Sketching with Partial Programs

Armando Solar-Lezama, Liviu Tancau, David Turner, Rastislav Bodik, Vijay Saraswat*, Sanjit Seshia
UC Berkeley   *IBM Research

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 synthesized implementation is correct by construction, which allows, among other benefits, rapid sketching of many 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 work, where sketches were meta-level rewrite rules, our sketches are simple-to-understand partial programs. Partial programs are programs with "holes" that are filled by the synthesizer. The unspecified behavior of partial programs is modeled with a single non-deterministic operator that we show to be surprisingly versatile. We also develop a synthesizer that is complete for the class of finite programs: it is guaranteed to complete any sketch in theory, and in practice has scaled to complex real-world programming problems.

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 is equivalent to a separate specification of the desired behavior.

The goal of sketching is to bridge the abstraction gap between a high-level task and its low-level implementation. Sketches sit between the two extremes: unlike specifications, sketches spell out the implementation strategy, and unlike implementations, they omit low-level details. By separating correctness and performance, sketches allow the abstraction gap to widen: First, sketching encourages cleaner specifications because it relies on specification only for behavioral specification. Second, sketching enables complex implementations that may be too tedious to develop and maintain without automatic synthesis of low-level detail.

We introduced the concept of sketching in StreamBit, a system for bit-stream programming [14]. In StreamBit, sketching proved to be very effective. We implemented, in a single afternoon, an implementation of the 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, we were unable to transfer the success of sketching from the lab to real programmers. Even though the concept of sketching was easy to explain, sketching as embedded in StreamBit required significant training. In fact, only one of the authors of [14] was able to write non-trivial sketches.

This paper attacks programmability challenges observed in StreamBit: (**1**) 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 [15]. While the dataflow programming model helped synthesis and subsequent parallelization, novice programmers faced sketching simultaneously with another new programming model.

This paper develops 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 straighforward 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 [14] was also restricted in expressiveness. (**1**) Except for some high-level refactorings, sketching worked only for programs that computed (semi)-permutations of bit-vectors. While this sufficed for DES, the modern block cipher standard, AES, was beyond our power. (**2**) The sketched implementations themselves could not implement permutations using non-permutations instructions, such as additions. This limitation prevented us from exploiting some efficient DES implementation strategies.

In contrast, the SKETCH language presented in this paper is complete. We can both *specify* any finite program and *sketch* any implementation of it. A finite program is any program whose input is bounded in size and the program is guaranteed to terminate on any input. Most high-performance kernels have this property.

In SKETCH, sketches are partial programs, i.e., programs where code fragments to be synthesized are indicated with a non-deterministic operator **??**. The operator is defined as returning a non-deterministically chosen integer value; the synthesizer replaces the operator with a suitable integer such that the resolved sketch behaves like the specification. The **??** operator is more versatile than it may seem: it can automatically derive values of hard-to-compute constants, such as bitmasks; it can be used to divide the work in a divide-and-conquer algorithm; and it can be used to synthesize the number of iterations of a loop. It can also be used to to synthesize a polynomial, which is useful in implementing big-integer multiplications algorithms.

Programming with non-determinism can be thought of as taking program verification one step further. In fact, if the sketch is fully deterministic, i.e., it is a regular program, then we are left with a simple verification problem where the compiler has to prove that an implementation is equivalent to the specification. However, the non-determinism 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 SKETCH language is supported by a new synthesis algorithm that is complete in that it can resolve an arbitrary sketch. The algorithm reduces the synthesis problem to a quantified Boolean satisfiability (SAT) 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 [12]. 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 also present an empirical evaluation of our system. We show that sketch can describe a very concise implementation of an AES stage, and that our solver resolves the sketch very fast, in about a minute.

In summary, this paper makes the following contributions:

- We develop a language for implementing a sketching high-performance kernels. Sketches are expressed as partial programs with a single, versatile non-deterministic operator that be used to synthesize, for example, bitmasks, control flow decisions, and polynomials.
- We build and evaluate a complete synthesizer: it can resolve in theory all sketches written in the SKETCH language. In practice, the solver scales extremely well for the cipher problems it was designed, and surprisingly well also for some harder problems.
- We developed a cookbook of programming with sketches (Section 2), implemented a spectrum of kernels in SKETCH, and evaluated the solver on these kernels.

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

2. Overview

The SKETCH language is a C-like procedural language with no pointers but with support for sketching. The language is targeted towards integer kernels over finite inputs. To support this domain, we support arrays. Vector operations are provided for arrays of bits, to give access to bitwise integer machine instructions.

This section gives a tutorial of the SKETCH language, going from simple to more complex kernels. All sketches in this section are beyond the power of [14].

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. 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 bitvector parallelism by relying on a little bit of algebra [16].

```
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.[1]

The ability to verify an implementation gives robustness, but the main contribution of SKETCH is 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 resolves `expression` to `~x & (x + 1)`; in the second to `~(x - 1) & x`. The semantics of caling 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 *unresolved* 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:

[1] Clearly, **`implements`** is not be confused with the same keyword in Java, which enforces (only) type-signature equivalence.

```
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 [16, 13].

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.

For word size $\mathtt{W} = 16$, the loop iterates 4 times. The synthesized code, unrolled, 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 sketched 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 [17]. 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 resolved to isolate the rightmost 1-bit even though the synthesizer is not instructed to do so; the only constraint given to the syntehsizer is that `popSparseSketched` must implement `pop`.

AES. This next problem is a filter from the AES cipher. The filter computes a multiplication of $(in * 02) \mod P$ in the Galois field of polynomials in $\{0, 1\}$. Note that multiplication with 02 can be expressed as a shift by one. Then the filter reduces by the polynomial $P = x^8 + x^4 + x^3 + x + 1$. Since the new polynomial is of degree $d \leq 8$, we only have to check if $d = 8$, and if so, subtract P.

```
bit[8] GFMul02 (bit[8] in) {
  bit[9] tin= in;
  tin = tin >> 1;

  bit[9] P = {1,1,0,1,1,0,0,0,1};
  if( tin[8] == 1 ){
    tin = tin ^ P;
  }
  return (bit[8])tin;
}
```

The specification above is inefficient on modern processors due to the unpredictable branch. The sketch below replaces the branch with a logical sequence that distributes the 8th bit to the positions defined by the polynomial. The sketch is efficient because it exploits the symmetry of 1-bits in the polynomial: the idea is to distribute the bit to multiple positions simultaneously. The sketch can be written even more compactly (and more generally) with a loop, but the unrolled sketch is easier to explain.

```
bit[8] GFMul02sk(bit[8] in) implements GFMul02 {
        bit [8] t1 = in >> ??;
        bit [8] m1 = (in<<??) & ??;
        bit [8] m2 = (m1>>??) | m1;
        bit [8] m3 = (m2>>??) | m2;
        return t1 ^ m3;
}
```

In the full AES cipher, there is a large stage that applies this function together with other seven similar ones to a stream of data in blocks of size 32. The full stage, however, can be implemented efficiently with the sketch shown below. The `rot` function is a rotation, and the synthesizer is able to resolve the sketch in less than 8 minutes, which is impressive given that we are sketching the implementation of one third of this complex cipher.

```
bit[32] MCSketch(bit[32] in) implements MixColumns{
   bit [32] t1 = (in >> ?? );
   bit [32] m1 = (in<<??) & ??;
   loop (2) { m1 = (m1>>??) | m1; }
   bit [32] o1 = (t1 & ??)^((in) & ??)^(m1 & ??);
   bit [32] o2 = (t1 & ??)^((in) & ??)^(m1 & ??);
   bit [32] o3 = (t1 & ??)^((in) & ??)^(m1 & ??);
   return o1 ^
          rot(o2, ??*8) ^
          rot(o3, ??*8) ^
          rot(o3, ??*8);
}
```

Often, filters can be implemented efficiently with table lookups. SKETCH can also synthesize the content of these tables, and our synthesizer scales easily to tables of size 2048 bits.

```
// <N> is a template, paremeterizing word size
bit[N*2] k<N>(bit[N] x, bit[N] y) implements mult<N>{
 if (N<2) return x*y;
 bit[N/2] x1, x2, y1, y2;
 bit[2*N] t11=0, t12=0,  t22=0, r=0;
 x1 = x[0:N-1]; x2 = x[N:2*N-1];
 y1 = y[0:N-1]; y2 = y[N:2*N-1];
 t11 = k<N/2>(x1, y1);
 t12 = poly<N/2>(2, x1, x2, y1, y2);
 t22 = k<N/2>(x2, y2);
 loop(5){
   // {||} non-deterministically selects one of its operands
   bit[2*N] t = (t11 {||} t22 {||} t12) << (N/2*??);
   r = r + (t {||} -t);
 }
 return r;
}

bit[2*N] poly<N>(int n, bit[N] x0, x1, x2, x3) {
if (n<=0) return ??;
return  k<N>(
   (x0 {||}  -x0 {||} 0) + (x1 {||}  -x1 {||} 0)
 + (x2 {||}  -x2 {||} 0) + (x3 {||}  -x3 {||} 0)
   , poly(n-1, x0,x1, x2, x3));
}
```

Figure 1. Sketch for Karatsuba's multiplication.

Karatsuba Multiplication SKETCH was designed to work only for finite programs. However, the following example will show that it can be leveraged to help the user write code even for programs with recursion and unbounded input sizes.

The Karatsuba multiplication algorithm is used to multiply large integers because its complexity is $O(N^{1.5})$, as opposed to $O(N^2)$ for the standard algorithm. The algorithm uses a divide-and-conquer approach based on the fact that one can express N-digit numbers as $X = X_1 + X_2 b^{N/2}$, where X_1 and X_2 are $N/2$-digit numbers and b is the basis.

The key idea is that one can multiply two numbers $X * Y$ by performing three (large) multiplications of their halves, instead of the usual four multiplications. The sketch expresses the fact that the three multiplications are going to be $X_1 * Y_1$, $X_2 * Y_2$, and a polynomial that is going to be the product of two sums of terms. The three terms then have to be added together and multiplied by various powers of $N/2$.

The idea behind sketching this complex algorithm is to (i) constrain the implementation to perform no more than three large multiplications; and (ii) synthesize the polynomials needed to make the three-multiplication version equivalent to the classical long integer multiplication. The idea is encoded in the sketch in figure 1. (We will provide a more detailed explanation of the sketch in the final paper.)

Our system currently resolves and verifies the sketch for $N = 6$ in about a minute. However, the programmer knows that the algorithm is the same regardless of the value of N, so he can use the code synthesized for $N = 6$ and use it correctly for any N.

3. Language definition

This section describes SKETCH, our sketching language. We first define the base language, which supports neither verification nor sketching. We then add the **implements** construct, which verifies equivalence of two functions, and the **??** operator needed for the construction of sketches.

3.1 The Base Language

SKETCH is a C-like language without pointers. We model the sketch-free subset of SKETCH using a formal language with three syntactic categories, *expressions*, *commands* and *functions*. The formal language is a standard imperative language over bits, Booleans, integers, and arrays, with some set of unary and binary operators, assignment, conditionals, loops, and function calls:

$$\begin{array}{lll} \text{(Expr)} & e & ::= & v \mid x \mid x[e] \mid unop\ e \mid e\ binop\ e \mid f(e) \\ \text{(Comm)} & c & ::= & x = e \mid skip \mid if\ (e)\ then\ c\ else\ c \\ & & & loop(e)\ c \mid c\ ;\ c \mid return\ e \\ \text{(Func)} & f & ::= & def\ f(x)\ c \end{array}$$

We have chosen this language to keep the technical development as simple as possible while focusing on the new ideas (sketching).

In the rest of this paper we shall take the liberty of writing concrete programs that use local variables, global variables, for loops, recursive function definitions, and functions with multiple arguments. The semantics of these constructs is completely standard — even in the presence of sketching operators — and omitted from this paper for reasons of space and simplicity.

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.)

3.2 The Sketching Constructs

We now extend the syntax with sketching constructs:

$$\begin{array}{llll} \text{(Expr)} & e & ::= & ?? \\ \text{(Func)} & f & ::= & def\ f(x)\ implements\ g\ c \end{array}$$

Intuitively, **??** introduces non-determinism by requiring the implementation to pick some number, while **implements** constrains these choices. f **implements** g requires that the choices made in the body of f must be such that the resulting program is functionally equivalent to g. We also enforce the static semantic restriction that g must not contain any occurrences of the **??** operator in its dynamic extent.

To define the meaning of **implements** more formally, it helps to rely on the notion of refinement. An implementation i is said to refine a specification s if every behavior of i can be exhibited by s [10, 8]. A specification may thus in general permit multiple implementations. For example, a specification $s : f(x) \geq 0$ permits implementations $f(x) = x * x$ as well as $f = abs(x)$.

In SKETCH, whenever we have i **implements** s, the compiler verifies that i refines s. However, since s permits only one behavior (s is a finite program, and hence a Boolean function), refinement implies that the i and s are functionally equivalent, a complete verification.

3.3 Formal Semantics

Recall from Section 2 that the synthesizer translates a sketch into a base program by replacing **??**'s with suitably chosen integers. We capture this formally by defining the operational semantics of SKETCH in terms of oracles modeled as lists of values (see Figure 2). Whenever a **??** is encountered during execution, an oracle is consulted. The value returned by the oracle is a number popped from the list. Therefore a **??** that appears inside a loop can resolve to a different value on each iteration.

$$\frac{}{\langle n,\sigma,O\rangle \Downarrow \langle n,O\rangle} \qquad \frac{}{\langle x,\sigma,O\rangle \Downarrow \langle \sigma(x),O\rangle}$$

$$\frac{}{\langle ??,\sigma,O\rangle \Downarrow \langle \mathit{head}\ O, \mathit{tail}\ O\rangle}$$

$$\frac{\langle e_1,\sigma,O\rangle \Downarrow \langle v_1,O'\rangle \quad \langle e_2,\sigma,O'\rangle \Downarrow \langle v_2,O''\rangle}{\langle e_1\ op\ e_2,\sigma,O\rangle \Downarrow \langle v_1\ op\ v_2,O''\rangle}$$

$$\frac{\langle e,\sigma,O\rangle \Downarrow \langle v,O'\rangle}{\langle x=e,\sigma,O\rangle \Downarrow \langle \sigma[x:=v],O'\rangle}$$

$$\frac{\langle e,\sigma,O\rangle \Downarrow \langle v,O'\rangle}{\langle \mathit{return}\ e,\sigma,O\rangle \Downarrow \langle \sigma[\mathit{retval}:=v],O'\rangle}$$

$$\frac{\langle c_1,\sigma,O\rangle \Downarrow \langle \sigma',O'\rangle \quad \langle c_2,\sigma',O'\rangle \Downarrow \langle \sigma'',O''\rangle}{\langle c_1;c_2,\ \sigma,O\rangle \Downarrow \langle \sigma'',O''\rangle}$$

$$\frac{\langle e,\sigma,O\rangle \Downarrow \langle \mathit{true},O'\rangle \quad \langle c_1,\sigma,O'\rangle \Downarrow \langle \sigma',O''\rangle}{\langle \mathit{if}\ e\ \mathit{then}\ c_1\ \mathit{else}\ c_2,\sigma,O\rangle \Downarrow \langle \sigma',O''\rangle}$$

$$\frac{\langle e,\sigma,O\rangle \Downarrow \langle \mathit{false},O'\rangle \quad \langle c_2,\sigma,O'\rangle \Downarrow \langle \sigma',O''\rangle}{\langle \mathit{if}\ e\ \mathit{then}\ c_1\ \mathit{else}\ c_2,\sigma,O\rangle \Downarrow \langle \sigma',O''\rangle}$$

$$\frac{\langle e,\sigma,O\rangle \Downarrow \langle 0,O'\rangle}{\langle \mathit{loop}\ (e)\ c,\sigma,O\rangle \Downarrow \langle \sigma,O'\rangle}$$

$$\frac{\langle e,\sigma,O\rangle \Downarrow \langle n,O'\rangle \quad n>0 \quad \langle c;\ \mathit{loop}(n-1)\ c,\sigma,O'\rangle \Downarrow \langle \sigma',O''\rangle}{\langle \mathit{loop}\ (e)\ c,\sigma\rangle \Downarrow \langle \sigma',O''\rangle}$$

Figure 2. The operational semantics of SKETCH programs. In addition to the standard program state σ we add the oracle O, which is a list of values.

If the number of iterations of a loop depends on the input, then the oracle's list must be long enough to satisfy the largest possible number of iterations. This implies the existence of an upper bound on iterations, hence our restriction to finite programs.

The programmer can think of an unbound sketch as a program that executes in the context of an arbitrary (non-deterministic) oracle. A sketch can be partially evaluated with respect to its oracle, producing a residual program for each possible oracle. The derivation of one such residual program from an unbound sketch is shown in Figure 3.

The definition of **`implements`** can now be stated in terms of oracles:

$$\frac{\exists O.\ f_O \approx g}{f\ \texttt{implements}\ g}$$

where f_O means f executed in the context of the oracle O and $\approx$ denotes functional equivalence. In words, this means that binding a sketch involves finding an oracle (one for all inputs) such that the sketch evaluates to the same result as the spec on all possible inputs. The problem of checking this functional equivalence reduces to SAT, and the problem of synthesizing a suitable oracle reduces to a generalized SAT problem, as will be shown in Section 4.

4. Synthesis

This section develops the synthesis process that resolves a sketch to make it functionally equivalent to the specification. First, we describe how programs are translated to Boolean functions. This functional representation of programs is the basis for detecting equivalence of specifications and (resolved) sketches. Second, we phrase synthesis as a quantified Boolean satisfiability problem and

$x = ??;\ \mathit{loop}(x)\ y = y\&??\ \rightarrow$
$\mathbf{x=2};\ \mathit{loop}(x)\ y = y\&??\ \rightarrow$
$\mathbf{x=2};\ y = y\&??;\ \mathit{loop}(2-1)\ y = y\&??\ \rightarrow$
$\mathbf{x=2};\ \mathbf{y=y\&13};\ \mathit{loop}(2-1)\ y = y\&??\ \rightarrow$
$\mathbf{x=2};\ \mathbf{y=y\&13};\ y = y\&??;\ \mathit{loop}(1-1)\ y = y\&??\ \rightarrow$
$\mathbf{x=2};\ \mathbf{y=y\&13};\ \mathbf{y=y\&7};\ \mathit{loop}(1-1)\ y = y\&??\ \rightarrow$
$\mathbf{x=2};\ \mathbf{y=y\&13};\ \mathbf{y=y\&7}$

Figure 3. Partial evaluation of the sketch "x=??; loop(x) y=y&??" using the oracle [2,13,7]

show that sketch resolution is decidable. Third, we describe a counterexample-driven solver for the satisfiability problem. The algorithm reduces the synthesis problem into two efficient SAT problems that iterate by exchanging information. Fourth, we show how the obtained solution is used to generate the resolved sketch in the C language. Next, we develop several scalability optimizations for the solver. We also describe how to deal with unscalable sketches at the programmer level, by heuristically identifying aspects of the sketch that may need to be specified by the programmer rather than synthesized by the solver. Finally, we empirically evaluate the counterexample-driven synthesis algorithm and its optimizations.

4.1 Programs as Boolean Functions

The synthesizer manipulates programs represented as Boolean functions. Both specifications and sketches are translated to Boolean functions with the same procedure, except that sketches produce functions with k additional *control inputs*; these inputs model values produced by the non-deterministic expressions $??$. The specification is translated to a function $P : \{0,1\}^m \rightarrow \{0,1\}^n$; the sketch is translated to a function $S : \{0,1\}^{(m+k)} \rightarrow \{0,1\}^n$. As described in Section 4.2, the goal of synthesis is to find a *control* — i.e., an assignment to the control arguments — such that the sketch computes the same function as the specification.

The translation to Boolean functions is done using the same standard approach used by many verification systems. In these systems, expressions corresponding to k-bit integers are modeled as a function from a set of named input bits to a vector of k output bits. Side effects are modeled as transformations of the program state σ, which is simply a map from variable names to Boolean functions. Our system also uses another less standard encoding for some integers and represent them in sparse form as pairs of values and guard functions (v, b); we distinguish between the two representations by giving all expressions a type, either `bin` for those encoded in standard binary form, or `spar` for those encoded in sparse form.

Unlike verification systems, our transformation also needs to keep track of which control inputs correspond to which execution of the $??$ expression. We do this by keeping a queue o where we push the fresh variable names produced by $??$ every time it is executed. After synthesizing the correct values for the control inputs, the variable names in o are replaced with their actual values, and o can now serve as an oracle during code generation, producing the correct value for a $??$ every time one is encountered.

We use the notation $\langle e,\sigma,o\rangle \rightarrow \langle e',\sigma',o'\rangle$ to denote that a program fragment e in state σ and with oracle o produces a Boolean expression e' in state σ' with oracle o'. We use the notation $o.v$ to indicate a new queue that contains all the values that o contained, followed by the value v. For example, the rule for the $??$ operator is shown below.

$$\frac{\hat{c}_1,\ldots,\hat{c}_k \text{ are fresh variables.}}{\langle ??,\sigma,o\rangle \rightarrow \langle [\hat{c}_1,\ldots,\hat{c}_k],\sigma,o.\hat{c}_1.\ldots.\hat{c}_k\rangle.}$$

Since expressions don't affect the state, we will abbreviate the rules for expressions as $\langle e, \sigma, o\rangle \rightarrow \langle e', o\rangle$. In our language, statements don't produce Boolean functions, so they will be abbreviated as $\langle e, \sigma, o\rangle \rightarrow \langle \sigma, o\rangle$.

The meaning of a procedure

$$[o_1, \ldots, o_k] fun(in_1, \ldots, in_m)\{c\}$$

with input parameters $in_1, \ldots, in_m$ and output parameters $o_1, \ldots, o_k$ is defined in terms of the Boolean functions that c assigns to the output variables when executed starting from a state where only the input variables have been assigned values.

$$\frac{[o_1, \ldots, o_k] fun(in_1, \ldots, in_m)\{c\} \quad \langle c, \{(in_1 := \sigma(x_1) \ldots (in_k := x_k)\}\rangle \rightarrow \sigma'}{\langle fun(x_1, \ldots, x_k), \sigma\rangle \rightarrow [\sigma'(o_1), \ldots, \sigma'(o_k)]}$$

To get the Boolean function for a sketch or a specification, we simply evaluate them as a function call, using as the start state the state that only maps the input variables to themselves.

The other noteworthy features of the translation, in addition to the ?? operator, are the modeling of arrays, loops and conditionals.

4.1.1 Arrays and sparse integer expressions

The sparse integer representation described above is mainly intended to model array operations precisely and efficiently. It is very similar to the guarded location sets used by Saturn [18] to represent pointers (Saturn does not model arrays). In this representation, an integer is represented as a set of *guarded values* of the form (v, b), where v is a value known at compile time and the *guard* b is a Boolean function.

The key invariant that we want to maintain for values in this representation is that at any given time, at most one guard will be true. Similarly, it is assumed that all the values v_i of all the guarded values in the set are unique. If in the process of evaluating an expression, two terms appear that have the same v, (v, b_1) and (v, b_2), they are implicitly replaced with a single term $(v, b_1 \vee b_2)$. Also, as a matter of convention, when writing a set of guarded values, we will assume that the first element in the set is the one with the smallest value.

For example, in this representation, an integer constant n evaluates to a value n guarded by the function $true$, i.e. the function that returns true on any input.

$$\frac{}{\langle n : \texttt{spar}, \sigma\rangle \rightarrow [(n, true)]}$$

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.

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

Another very important property of the sparse integer representation is that integer expressions get partially evaluated in a very precise manner. In fact, at any point in the program, if an expression e : spar does not depend on any inputs or control values, then $e = [(n, true)]$ for some constant n. In practice, we only do trivial minimizations on the guards as we construct the functions, so the internal representation would more likely be $e = [(n_1, f_1), (n_2, f_2), \ldots (n_k, f_k)]$, where f_1 is a tautology and $f_2, \ldots, f_k$ are unsatisfiable.

The conversion from binary to sparse representation is potentially exponential, since the sparse form is essentially a unary representation.

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

However, with some analysis it is possible to look at the places where the generated expression will be used and put a tighter bound on the number of terms. For example, if after the conversion the integer is used only to access an array of size N, then the sparse representation needs to keep at most $N + 1$ terms—all the terms smaller than N, plus a term to model the point when the array goes out of bounds. Our current implementation leaves to the user the decision about the representation by having two different data types for the two different representations and forcing users to do conversions explicitly.

As was mentioned earlier, the main rationale for the sparse representation was to make array accesses more efficient. An N element array is modeled in our system as a tuple of N expressions, which can be either in binary
form

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

or in sparse form.

$$\text{ar} : \texttt{spar} = [[(v_1^1, b_1^1), \ldots, (v_i^1, b_i^1)], \ldots, [(v_1^n, b_1^n), \ldots, (v_i^n, b_i^n)]]$$

Given an index expression $e = [(x_1, f_1), \ldots (x_k, f_k)]$ we define the array access for arrays in binary form by using each x_i to select an element in the array, and applying an and of that element with the corresponding f_i, and then taking the disjunction of all such terms.

$$\frac{\langle e : \texttt{spar}, \sigma\rangle \rightarrow [(x_1, f_1), \ldots (x_k, f_k)] \quad \sigma(\text{ar}) = [[b_1^1, \ldots, b_k^1], \ldots, [b_1^n, \ldots, b_k^n]]}{\langle ar[e] : \texttt{bin}, \sigma\rangle \rightarrow [(\bigvee(b_1^{x_i} \wedge f_i)), \ldots, (\bigvee(b_k^{x_i} \wedge f_i^e))]}$$

For arrays with sparse elements, the indexing is also straightforward; for each (x_j, f_j) in the in the index expression, we select the expression corresponding to the x_i entry in the array ar$[x_j]$, and create new guarded values by taking each value $(v_i^{x_j}, b_i^{x_j}) \in$ar$[x_j]$ and making its guard the conjunction of its original guard with f_i to get $(v_i^{x_j}, b_i^{x_j} \wedge f_j)$.

$$\frac{\langle e : \texttt{spar}, \sigma\rangle \rightarrow [(x_1, f_1), \ldots (x_k, f_k)] \quad \sigma(\text{ar}) = [[(v_1^1, b_1^1), \ldots, (v_i^1, b_i^1)], \ldots, [(v_1^n, b_1^n), \ldots, (v_i^n, b_i^n)]]}{\langle ar[e] : \texttt{spar}, \sigma\rangle \rightarrow [(v_i^{x_j}, b_i^{x_j} \wedge f_j) \forall i \leq m, j \leq k]}.$$

4.1.2 Conditionals and Loops

The treatment of conditionals is again fairly standard. It is actually simpler than what is used for verification because we are not interested in proving assertions from within the conditionals. This means, that in the case of an `if` statement, for example, we don't need to keep track of the branch condition while translating the two branches; we only need it at the join point where the two branches merge back together.

To handle `if` statements, we evaluate the two branches independently from the same initial state σ to get two new states, σ_1 and σ_2. The two states are merged into a new state that maps each variable x into the function $(b \wedge \sigma_1(x)) \vee (\neg b \wedge \sigma_2(x))$. Note that this expression reduces to $\sigma(x)$ when x is not modified in any branch of the conditional because in that case $\sigma(x) = \sigma_1(x) = \sigma_2(x)$.

$$\frac{\langle e : \texttt{bin}, \sigma\rangle \rightarrow [b] \quad \langle c_1, \sigma\rangle \rightarrow \sigma_1 \quad \langle c_2, \sigma\rangle \rightarrow \sigma_2}{\langle if\ e\ then\ c_1\ else\ c_2, \sigma\rangle \rightarrow \lambda x.(b \wedge \sigma_1(x)) \vee (\neg b \wedge \sigma_2(x))}$$

In the special case where e evaluates to either $true$ or $false$, we only evaluate one of the branches and ignore the other one. Combined with the partial evaluation that takes place automatically for sparse integers, this allows us to handle recursive function calls for which the level of recursion can be bounded at compile time.

Loops are somewhat more involved. They are handled essentially by unrolling, but we can take advantage of the sparse representation to get better bounds on how much to unroll, and where to place loop exit branches.

$$\frac{\langle e, \sigma\rangle \rightarrow [(0, true)]}{\langle loop\ (e : \mathtt{spar})\ c, \sigma\rangle \rightarrow \sigma}$$

$$\frac{\begin{array}{c}\langle e, \sigma\rangle \rightarrow [(n, b_1)] \\ e' = [(n-1, true)] \quad \langle c, \sigma\rangle \rightarrow \sigma_1 \quad \langle loop\ (e')\ c, \sigma_1\rangle \rightarrow \sigma_2\end{array}}{\langle loop\ (e : \mathtt{spar})\ c, \sigma\rangle \rightarrow \sigma_2}$$

$$\frac{\begin{array}{c}\langle loop(e'')c;\ if\ (e > e'')\ then\ (loop(e')c), \sigma\rangle \rightarrow \sigma_2 \\ \langle e : \mathtt{spar}, \sigma\rangle \rightarrow [(v_1, b_1), \ldots, (v_k, b_k)] \\ e' = [(v_2 - v_1, b_2), \ldots, (v_k - v_1, b_k)] \ \ e'' = [(v_1, true)]\end{array}}{\langle loop\ (e)\ c, \sigma\rangle \rightarrow \sigma_2}$$

For this last rule, note that the unrolling is guaranteed to terminate because e' has strictly less terms than e. Finally, note that if all the v_i in e are non-negative, they will remain non-negative, since we are assuming v_1 is the smallest one. Also, note that if e satisfies the invariant that at most one b is true, this will also be true for e'.

4.1.3 A simple example

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

```
loop(??){
  b = b ^ in[??];
}
```

And assume the unknown is fixed at two bits. Then we have that

$$?? \rightarrow [\hat{c_1}, \hat{c_2}]$$

Which must be converted to sparse form to produce

$$?? : \mathtt{spar} \rightarrow [(0, \neg\hat{c_1} \wedge \neg\hat{c_2}), (1, \hat{c_1} \wedge \neg\hat{c_2}), (2, \neg\hat{c_1} \wedge \hat{c_2}), (3, \hat{c_1} \wedge \hat{c_2})]$$

After repeated application of the unrolling rule, we get that the semantics of the original loop will be equivalent to the semantics of:

```
if(e_0 > 0 )
  b = b $\oplus$ in[$\Star$]$;
  if(e_1 > 1 )
    b = b \oplus in[$\Star$];
      if(e_2 > 1 )
        b = b \oplus in[$\Star$];
```

where e_0 is the expression we got from ??, and each subsequent e we get by subtracting the first term from the rest of the terms.

$$e_0 = [(0, \neg\hat{c_1} \wedge \neg\hat{c_2}), (1, \hat{c_1} \wedge \neg\hat{c_2}), (2, \neg\hat{c_1} \wedge \hat{c_2}), (3, \hat{c_1} \wedge \hat{c_2})]$$

$$e_1 = [(1, (\hat{c_1} \wedge \neg\hat{c_2})), (2, (\neg\hat{c_1} \wedge \hat{c_2})), (3, (\hat{c_1} \wedge \hat{c_2}))]$$

$$e_2 = [(1, (\neg\hat{c_1} \wedge \hat{c_2})), (2, (\hat{c_1} \wedge \hat{c_2}))]$$

4.2 Synthesis as QBF

Having obtained functional representations 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 resolved to the specification P if there exists a control c such that the specification and sketch

```
function synthesize(sketch $S$, specification $P$)
    // Synthesize control that resolves $S$ 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
        $I = I \cup \{x\}$
        $c$ = synthesizeForSomeInputs($I$)
        if $c$ = nil then exit("buggy sketch")
        $x$ = verifyForAllInputs($c$)
    until $x \neq$ 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., $\forall x \in I.P(x) = S(x, c)$
    if $\bigwedge_{x \in I} P(x) = S(x, c)$ is satisfiable then
        return $c$ that satisfies the formula
    else // sketch $S$ cannot be resolved
        return nil
    end if
function verifyForAllInputs(control $c$)
    // Verify if sketch $S$ resolved with controls $c$ is functionally
    // equivalent to the specification $P$. If not, return the witness
    // (counterexample) $x$, i.e., $P(x) \neq S(x, c)$.
    if $P(x) \neq S(x, c)$ is satisfiable then
        return $x$ satisfying the formula
    else
        return nil
    end if
```

Figure 4. The counterexample-driven synthesis algorithm.

are functionally identical, i.e., when the following formula is satisfiable.

$$\exists c \in \{0, 1\}^k, \forall x \in \{0, 1\}^m; P(x) = S(x, c) \qquad (4.1)$$

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

4.3 Counterexample-Driven Solver

Problem (4.1) 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 (4.1) 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.

Existing solvers for 2QBF employ two SAT solvers, one for each quantifier, that communicate with each other [12]. Our solver also 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 resolved 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. Other solvers tailored to 2QBF make the same trade-off [12].

4.4 Generating Code for a Resolved Sketch

Given a synthesized control c, we are ready to resolve the sketch and translate it to a C program. The translation uses the oracles introduced earlier. For each oracle, we have a list of variables in the order in which values were required from it; these variables can be replaced with actual values, which are to be used in the execution of the program. The code generator then takes the original function, and adds a declaration of an array `oracle[]` that is going to represent the oracle, together with an array initializer containing all its values, and an index to keep track of the current value. Each instance of the $??$ operator is then replaced with the expression `oracle[idx++]`. After this, traditional compiler optimizations can be used to propagate constants, and specialize the code for the generated values.

4.5 Scalability Optimizations

The synthesizer described so far scales only for very small programs. Here we describe the reasons and develop optimizations for making the synthesizer significantly more scalable. We evaluate the benefits if these optimizations in Section 5.2.

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

Random counterexamples. The counter-examples generated by the SAT solver sometimes don't lead to rapid convergence. Our solution is seed the set of inputs I with several random inputs. Adding more random inputs can reduce the number of iterations of the synthesize-verify loop at the expense of making each iteration more expensive. For this reason, this optimization is mainly beneficial in cases where the loop takes many iterations, but each iteration is very fast. This issue will be analyzed in further detail in the full paper.

4.6 Diagnostics

When there are bugs in the sketches, the compiler reports that the sketch can not be satisfied, but it may be difficult for the programmer to find what the problem is. The main diagnostic tool of our compiler, is to allow the user to select a set of outputs, and ask the compiler for an assignment of the controls, together with a set of inputs and outputs such that the output of the sketch and the spec differs only for the selected outputs. We have found this to be a useful tool for isolating bugs in the sketches because it allows the user to understand why the sketch cannot be satisfied.

5. Evaluation

To evaluate our work, we ask four questions: (1) Can the SKETCH language be used in practically interesting domains? (2) Is our synthesizer scalable enough for real-world problem sizes? (3) What kinds of code fragments can be synthesized with the sketching support provided in SKETCH, and what is the programmer experience? (4) Does a sketched implementation match the performance of hand-written code?

5.1 Expressiveness of the SKETCH language

Here we examine the expressiveness of the SKETCH language, leaving the examination of its synthesis power to Section 5.2. SKETCH is sufficiently powerful for both specifying and implementing finite programs, but do finite programs extend to practically interesting domains? We believe that SKETCH applies at least in private key cryptography, public key cryptography, error correction, signal processing, vision and graphics. The reason is finite programs cover three broad (albeit vaguely defined) classes of programs, which have substantial use in the above domains: (i) programs mapping a few words to a few words, such as encrypting a fixed sized message; (ii) array-manipulating programs, with known array sizes, such as matrix multiplication; and (iii) stateless streaming programs, such as encrypting a variable-length message. Technically, streaming programs manipulate unbounded streams, so they are not finite, but they are typically composed of (finite) filters that process the stream a few bytes at a time. Note that in our work on bitstreaming, we sketched precisely these filters [14]. We can also implement stateful filters (i.e., those that compute a "running" function on a stream), by restricting ourselves to the (finite) filter wrapped inside the feedback loop.

We implemented SKETCH programs that span these classes. Table **??** lists the programs that we use for evaluation in this section. The programs are divided into three categories: word manipulation routines, streaming kernels, and programmability tests. The first group includes, in addition to some of the kernels we have mentioned already, `log2`, a routine that computes the log_2 of a number in logarithmic time with respect to the word size, and `parity`, a routine that computes the parity bit for a word.

The second group includes `CRC`, a kernel that computes a cyclic redundancy check, `DES.IP`, the initial permutation from DES, and `AES.MixColumns`, one stage of AES that is based on operations on the Galois field $GF(2^8)$.

The last group evaluates the ease of sketching on complex problems that go beyond error correction and cryptography. This group is also a good stress test for the synthesizer, which is not designed to scale on sketches multiplications of two integers larger than 16 bits or so. It includes the version of Karatsuba we saw earlier, and `poly`, a simple sketch that finds the coefficients of a degree 4 polynomial of degree 5.

5.2 Scalability of the Synthesizer

Here we report the scalability of the synthesizer on our sketches. We divide sketches into two groups: those within the (intended) power of the synthesizer, and those containing synthesis problems beyond what we envisioned for the synthesizer in this paper.

The performance of the synthesizer is given in Figure 9. For each benchmark, we show the number of controls we are trying to synthesize, The total number of iterations of the synthesize-verify loop, and the total ammount of time spent on synthesize and verify. For a few benchmarks, we include the scaling behavior as we increase the input size.

We now examine the synthesizer on the stress tests. A distinguishing feature of these sketches is that they contain multiplications whose both operands are relatively wide integers. An example is the polynomial synthesis sketch. It is well known that SAT solvers do not handle multiplications gracefully, and so we did not expect these sketches to scale to large word sizes with the SAT-based synthesizer solver.

The synthesis of sketches exceeded our expectations. The running times are shown also in Figure 9.

Another interesting point to note is the scaling behavior of the number of iterations, vs the time to solve each individual iteration. For example, for problems like Karatsuba, the number of iterations grows very slowly compared with the exponential growth in the amount of time it takes to solve each individual SAT problem. For CRC-table, on the other hand we can see the exponential behavior on the number of iterations as well.

	Karatsuba	Log2 24	LogCount	parity	reverse	tblcrc	tblcrc2	AES MixCo	DES.IP	POLY 16
Input size	6	24	8	24	64	3	8	32	64	16
Iterations	5	94	9	14	51	4	255	3	16	17
Synthesize	11347	1268455	4322	1502	193160	4892	244932	442984	692808	617930
Veritfy	5916	2684	99	258	67291	1370	8644	77305	87319	272473
unknowns	63	409	109	45	522	32	2048	2602	178	96

Figure 5. The running time of the syntehsizer.

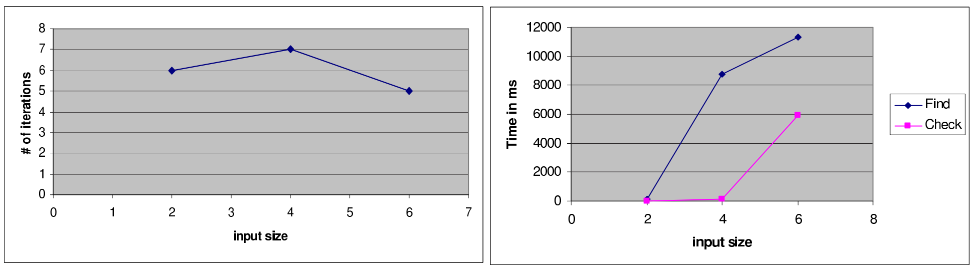

Figure 6. Karatsuba

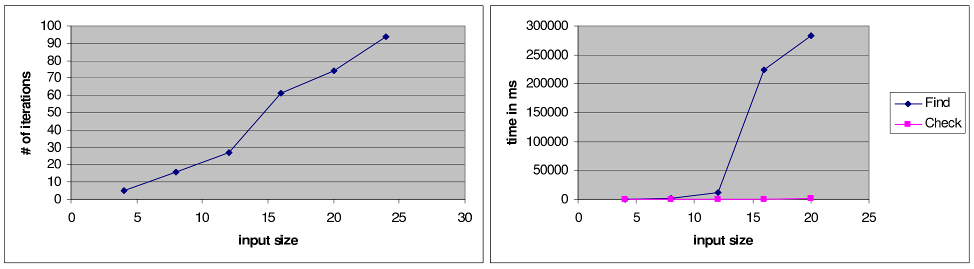

Figure 7. Log2

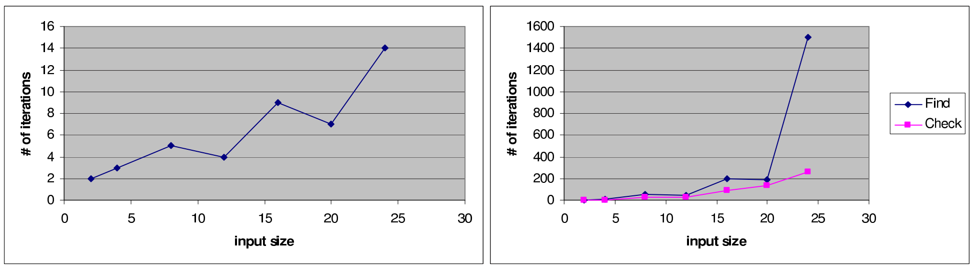

Figure 8. Parity

5.3 Programming with Sketches

One factor in the evaluation of linguistic support for sketching is whether sketches can be used to synthesize non-trivial implementations. We have shown the range of sketchable constructs in Section **??**. All sketches shown in that section can also be syntehsized with our current synthesizer.

Another evaluation factor is programability. Perhaps contrary to intuition, our experience shows that sketches are much easier to develop than the specifications. Once the specification is debugged, the sketch can be obtained easily, and much faster than teh specification. Sketching alternative implementaitons takes even less time.

One experience with sketching worth elaborating on is revising non-scalable sketches. For some sketches with a large number of **??** operators, the synthesizer would not terminate in a reasonable time (a few minutes). In these sketches, it was typically easier for the programmer to make the sketch somewhat less non-deterministic, by giving the sketch some more specific constraints.

6. Related Work

Synthesis based on partial programs has been explored in the AI community. For example, ALisp [1], developed by Andre and Russell 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 resolution 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 [5]. 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 [9] was one of the first systems to leverage the progress in SAT solving for code optimization. Denali

	Karatsuba	Log2 24	LogCount	parity	reverse	tblcrc	tblcrc2	AES MixCo	DES.IP	POLY 16
Input size	6	24	8	24	64	3	8	32	64	16
Iterations	5	94	9	14	51	4	255	3	16	17
Synthesize	11347	1268455	4322	1502	193160	4892	244932	442984	692808	617930
Veritfy	5916	2684	99	258	67291	1370	8644	77305	87319	272473
unknowns	63	409	109	45	522	32	2048	2602	178	96

Figure 9. Parity

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 [6] and Saturn [18]. Our work uses SAT solving not just for program verification, but also for program synthesis.

It is likely that we will hit the capacity limits of current SAT solvers, such as zChaff [11], for larger word sizes involving operations such as integer multiplication. We anticipate being able to scale further by using two strategies. The first involves switching to *word-level solvers* (also known *bit-vector decision procedures*). These solvers avoid exploding a word of size w into w Boolean variables by using either word-level axioms and simplification rules (e.g., [2, 3]) or efficient, special-purpose data structures (e.g., [4]). The second strategy is based on employing analyses that generalize a synthesis obtained for a smaller word or array size to larger sizes by exploiting symmetries and dependences in control and input variables.

7. 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 a significant fraction of the AES cipher; the sketch is suprisingly concise and the solver scales extremely well on this benchmark.

References

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

[2] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the* 35^{th} *Design Automation Conference (DAC '98)*, pages 522–527. Association for Computing Machinery, June 1998.

[3] S. Berezin, V. Ganesh, and D. Dill. A decision procedure for fixed-width bit-vectors. Technical report, Computer Science Department, Stanford University, April 2005.

[4] Y.-A. Chen and R. E. Bryant. An efficient graph representation for arithmetic circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(12):1442–1454, December 2001.

[5] Y.-A. Chen and R. E. Bryant. Autobayes: a system for generating data analysis programs from statistical models. *Journal of Functional Programming*, 13(3):483–508, May 2003.

[6] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc.* 10^{th} *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.

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

[8] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

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

[10] R. Milner. An algebraic definition of simulation between programs. In *Proc. of the 2nd International Joint Conference on Artificial Intelligence*, 1971.

[11] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference (DAC '01)*, pages 530–535, June 2001.

[12] 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.

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

[14] 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.

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

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

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

[18] 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.