

# Extending the Applicability of Sketching

Liviu Tancau, Armando Solar-Lezama, Gilad Arnold

## Abstract

We present four extensions to the SKETCH programming system developed at UC Berkeley: (a) support for floating point numbers; (b) a new circuit-based solver; (c) performing synthesis and verification at two separate levels of granularity; (d) extending solutions of small problems to larger instances. These extensions improve the performance of the system, add new functionality, and address scalability concerns for real-world algorithms.

## 1. Introduction

The SKETCH compiler supports a model of sketching that allows the user to write a simple reference implementation, and to give the system implementation information in the form of a program with holes. The compiler works by modeling the problem as a 2QBF satisfiability problem and then using the results of this problem to fill in the holes in the sketch.

The overarching theme of this project is to address some of the issues that arise in modeling the original program/sketch pair as a 2QBF problem. The basic SKETCH algorithm starts by modeling the “implements” relation between program and sketch as a 2QBF problem of the form

$$S(F) := \exists h. \forall x. F(x, h) .$$

The 2QBF problem is resolved via counterexample guided refinement strategy (CEGAR) similar to what is used for model checking. We abstract the problem down to the synthesis problem

$$S(F, I) := \exists h. \forall x \in I. F(x, h) ,$$

where we try to resolve for an  $h$  that works only for a small set of inputs  $I$ , and then we verify that the witness produced by  $S(F, I)$  is indeed a witness for the original problem by solving the verification problem:

$$V(F, h) := \forall x. F(x, h)$$

Section 2, is concerned with the construction of  $F(x, h)$  for floating point sketches. This construction is non-trivial because most floating point algorithms are only approximations, and different implementation strategies can lead to different approximations, so we can no longer use the simple definition of  $F(x, h)$  as  $F(x, h) := K(x, h) = P(x)$ , where  $K(x, h)$  is a Boolean representation of the sketch, and  $P(x)$  is a Boolean representation of the program.

Section 3 describes our experience using some techniques from circuit verification to encode the  $S(F, I)$  and  $V(F, h)$  problems down to CNF.

The basic SKETCH algorithm is symmetric in the sense that  $S(F, I)$  and  $V(F, I)$  both use the same  $F$ , and are resolved using an identical solver. Section 4 describes how the use of different solution strategies for  $S$  and  $V$  can lead to improved performance.

Finally, Section 5 takes the idea of the asymmetric solver one step further and explores the use of a two-level verifier. Here, the idea is to use a bounded SAT-based verifier as a first level, and if the verifier returns positive, a secondary decision procedure tries to prove that the answer that worked for the bounded problem generalizes to larger sizes.

## 2. Modeling Floating-Point Arithmetic

In order to extend the domain of applicability of our sketching compiler, we have added support for manipulating floating point numbers.

Rather than providing built-in support for a new data type, we created a library file containing functions that operate on and return bit vectors whose values are interpreted as floating point numbers according to a scheme similar to IEEE 754. Due to the inherent loss of precision found in floating point algorithms, we had to replace the usual sketch functional equivalence with a form of approximate equivalence. We analyzed the scalability of sketches constructed in this manner by comparing their resolution time to that of sketches performing similar operations on integers of the same size. Furthermore, we analyzed the effect of varying the data representation parameters on the resolution time and accuracy of a more complicated benchmark (fast inverse square root). Our results and conclusions are presented in the following sections.

### 2.1 Data representation

We represent a floating point number as a sequence of  $1 + E + M$  bits: a sign bit,  $E$  exponent bits, and  $M$  mantissa bits (with the sign bit being the most significant).  $M$  and  $E$  are adjustable parameters, which are best kept between 4 and 8 in order to allow sketches to resolve in reasonable time without sacrificing too much precision. Ideally we would like to set  $E$  to 8 and  $M$  to 23 to follow the IEEE 754 specification for floats, but this is impractical. We further deviate from the IEEE standard by explicitly normalizing the mantissa (its MSB is always 1 unless the number is 0). This was necessary in order to simplify the implementation of the library. The interpretation of a number stored in the above mentioned format is  $(-1)^s * m * 2^{e-b}$ , where  $s$  is the sign bit,  $e$  and  $m$  are the values of the exponent and mantissa parts, and  $b$  is a bias constant, which we set to  $2^{M-1}$ .

### 2.2 Approximate equivalence

The traditional way of using our system is to write an executable spec and a sketch that is to perform the same thing and declare them as being functionally equivalent. Since this cannot work for approximate algorithms, we needed to come up with a way of expressing approximate equivalence. The first method we considered was to simply discard some of the least significant bits from the mantissas of the return values of the spec and sketch. However, this is highly problematic since two numbers that are very close may differ in more than just the last few bits. For example, when  $M = E = 4$ , 10001011 (1.375) and 10001100 (1.5) are “consecutive” numbers, but differ in the last 3 bits. Even more convincingly, 01111111 (0.938) and 10001000 (1.0) are also consecutive but have no common prefix at all. For this reason, we decided to simply compute the difference between the result returned by the spec and that returned by the sketch and make sure it’s bounded by a user-specified epsilon value. In order to implement this kind of declarative-style specification using existing SKETCH constructs, we came up with the following template:

```
bit alwaysOne(bit[W] in) {
    return 1;
}
bit equiv(bit[W] in) implements alwaysOne {
    if(abs(spec(in)-sketch(in))<epsilon)
        return 1;
    else
        return 0;
}
```

$N$	f-sqrt	i-sqrt
16	18	3
32	78	7
64	4563	20

**Table 1.** Resolution times (ms) of floating point vs integer sketch

}

By establishing functional equivalence between `alwaysOne` and `equiv` over all inputs, we are forcing the solver to find a way to resolve the sketch such that the first branch of the if statement is always taken, which is only possible if the spec and sketch never differ by more than epsilon. Note that this template allows us to impose arbitrary constraints on the sketch. In fact, we do not even need a spec (reference implementation) if we can succinctly express some property satisfied by the sketch instead (as shown in subsequent sections).

### 2.3 Float vs Integer

In order to compare the solver’s ability to deal with floats as opposed to integers, we wrote two sketches that compute the square root of an  $N$ -bit number, interpreted as an integer in one and as a floating point number ( $M=E=N/2$ ) in the other. Note that instead of synthesizing a function, we used the solver to synthesize just a constant.

```

bit equiv() implements alwaysOne {
  bit[N] n= <some constant>;
  bit[N] s=??;
  if(abs((n-s*s)/n)<epsilon)
    return 1;
  else
    return 0;
}

```

As evident from Table 1, integer-only sketches are easier to solve than ones containing floating point numbers, and the relative difference between the two increases exponentially with the input size (i.e. the size of the problem).

### 2.4 Inverse Square Root

In order to demonstrate the usefulness of our system (i.e. its ability to synthesize practical approximate algorithms), we implemented a sketched version of the fast inverse square root routine found in the source code of Quake 3. This routine is of practical importance because it can be used to compute the scaling factor needed to normalize a vector, which is a frequent operation in computer graphics applications. The original version of the algorithm, written in C, is as follows:

```

float InvSqrt (float x) {
  float xhalf = 0.5f*x;
  int i = *(int*)&x;
  i = 0x5f3759df - (i >> 1);
  x = *(float*)&i;
  x = x*(1.5f - xhalf*x*x);
  return x;
}

```

The correctness and performance of this algorithm was analyzed in [2]. It reportedly runs four times faster than the trivial implementation ( $1/\sqrt{x}$ ) since it avoids both the square root operation and the inversion, which are expensive. The first key idea is treating the float as an integer and performing some simple algebra ( $0x5f3759df - (i \gg 1)$ ). Doing this alone is enough to compute the inverse square root within a relative error of 4%. However, the approximation can be refined using Newton’s method, which is precisely what the second last line of code does. The final result is reportedly within 0.18% of the true value of  $1/\sqrt{x}$  for any legal input.

$M$	$E$	$W$	epsilon	synth. const.	solve time (s)
4	4	9	0.25	C0	.477
5	5	11	0.25	2FF	.935
6	6	13	0.25	BFF	4.06
7	7	15	0.25	2FFF	4.18
8	8	17	0.25	BFFF	12.9
12	8	21	0.25	BFFFF	34.3
16	8	25	0.25	BFFFFF	34.7
20	8	29	0.25	BFFFFFF	1288

**Table 2.** Benchmark results for isqrt sketch (varying both  $M$  and  $E$ )

$M$	$E$	solve time (s)	$M$	$E$	solve time (s)
6	4	1.03	4	6	.509
6	5	2.49	5	6	.789
6	6	4.06	6	6	4.06
6	7	4.59	7	6	4.53
6	8	4.48	8	6	10.2

**Table 3.** Benchmark results for isqrt (independently varying  $M, E$ )

The obvious way to sketch this algorithm is to replace the magic number with the `??` construct. Furthermore, since this algorithm satisfies the property  $x*y*y = 1$ , where  $y$  stands for the output of the function, we can use this instead of a reference implementation. The resulting SKETCH code is shown below (for clarity, we have greatly simplified the code by using operators instead of function calls and omitting details pertaining to variable initialization and various integrity assertions):

```

bit[W] isqrt(bit[W] x) {
  bit[W] t=??-(x>>1);
  return t*(1.5-0.5*x*t*t);
}
bit equiv(bit[W] x) implements alwaysOne {
  bit[N] y=isqrt(x);
  if(abs(x*y*y-1)<epsilon)
    return 1;
  else
    return 0;
}

```

Despite the fact that our floating point representation differs from IEEE 754, the sketch does in fact resolve for any value of  $M$  and  $E$  (unless they are too large). Table 2 shows what constants were synthesized in each case and how long it took for the solver to finish executing. It is easy to see that there is an exponential blow-up in the solving time as a function of the input size. The problem becomes impractical just as the values of  $M$  and  $E$  approach the IEEE standard values, so no results were obtained beyond  $M = 20$ . However, it is easy to see a pattern in the generated constants. Almost all of them are of the form 010111... (i.e. all bits but the MSB and MSB-2 are ones). This would enable a perceptive programmer to extend the algorithm to any word size.

To get a better understanding of how the solving time is influenced by the values of  $M$  and  $E$ , additional tests in which one of them was held constant were performed. As evidenced in Table 3, the size of the mantissa plays a more important role in determining the resolution time. However, it should be noted that repeating the tests can cause the numbers to change by as much as an order of magnitude (since the solver is non-deterministic), so the individual results are not reliable, and overall trends are difficult to predict exactly.

The value of epsilon used in all cases was 0.25; the sketch could not be resolved when epsilon was set any lower. This error value is much larger than the error of 0.0018 associated with the C code, but this is most likely due to our different representation of floats. To improve the precision of our sketched algorithm, we tried two additional mod-

method	$M$	$E$	epsilon	synth. const.	solve time (s)
1.5 ←??	6	6	0.125	C04, 1.53	55.1
1.5 ←??	8	8	0.062	C011, 1.53	74.5
Newton x2	6	6	0.062	BFF	9.6
Newton x2	8	6	0.031	BFFF	26.2
both	6	6	0.062	C00, 1.5	37.2
both	8	8	0.031	C005, 1.5	931

**Table 4.** Modified isqrt benchmark results

ifications: replacing the 1.5 with ?? and adding an extra Newton step (duplicating the second line of the sketch). The results of solving these modified sketches are shown in Table 4. When 1.5 is replaced with a wildcard, the solver resolves it to 1.53 and picks a different constant for the initial subtraction. The result is 2x improvement in accuracy for the case  $M = E = 6$  and 4x improvement for  $M = E = 8$ . Adding an additional Newton step gives a further 2x accuracy improvement in either case. It can be seen that adding a wildcard significantly increases the solving time (by introducing extra control bits), and that adding the extra Newton step at the same time makes the sketch resolution time almost prohibitive (due to an increase in the size of the Boolean formula).

The lesson learned from these experiments seems to be that a sketch should be kept as short as possible and with a minimal number of ?? constructs. The input size should be kept small and, if possible, the results should be generalized to larger problem instances. Section 5 will discuss ways in which this can be accomplished automatically.

### 3. Improving Circuit Encoding

In SKETCH, the program and its sketch are modeled as a Boolean circuits  $P(x)$  and  $K(x, h)$ , respectively, and from these, we construct a Boolean circuit  $F(x, h) := K(x, h) == P(x)$ .

The verification problem  $V(F, h)$  is encoded as a SAT problem where we look for an  $x$  such that  $\neg F(x, h)$ . SKETCH builds this problem by traversing  $F$  from the inputs to the outputs, producing CNF clauses for each node in a straightforward manner. For the project we rewrote the encoding process to make use of ABC, a system developed by Alan Mishchenko, Satrajit Chatterjee and Robert Brayton [3].

ABC is a software system for synthesis and verification of binary sequential logic circuits. ABC implements several algorithms for manipulating and simplifying SAT problems represented as Boolean circuits, making them easier to handle by DPLL-based SAT solver, which in this case is MiniSat.

ABC represents Boolean circuits using And-Inverter-graphs (AIG)–networks of 2-input AND gates and inverters. Using this representation, ABC applies various transformations to eliminate redundancies in the circuit, and also reduce the depth of the circuit from input to output, both of which make the DPLL-based solver work more efficiently. The main transformations implemented in ABC are: rewriting, refactoring, and balancing. The rewriting uses different forms of hashing to increase sharing of sub-expressions in the circuit. The refactoring takes cones with 10-20 inputs from the circuit, computes a canonical representation of the function represented by such cones using a BDD, and then converts the function back to a more efficient AIG. Finally, the balancing transformation uses associativity of the AND gates to try to reduce the number of AIG levels in the network. When encoding the network into CNF clauses, ABC also uses a smart encoding that groups several gates together before producing CNF clauses, and produce additional clauses from the minimum necessary, in order to speed up the propagation of certain facts.

Initial tests using ABC to solve SAT problems generated by SKETCH showed some dramatic improvements in performance. To give an example, for one of our benchmarks, if you solve  $S(F, I)$  for a random  $I$ , zchaff would take 671 seconds, while ABC took 45 seconds. However, when ABC was fully incorporated into SKETCH, the results were rather poor. For the same benchmark mentioned above, the

With abc without transformations:

```

0 0 0 0 1 1 1 0
0 0 1 1 1 1 0 1
0 0 0 0 1 0 0 1
0 0 0 1 0 1 0 0
0 0 0 1 0 1 0 1
1 0 1 1 1 1 0 1
1 1 0 0 1 1 1 0
0 1 1 1 1 1 0 1
1 1 1 1 1 1 1 0
0 1 0 0 0 0 1 0
0 0 1 1 1 1 1 0

```

abc with transformations:

```

1 0 0 1 0 1 1 1
0 1 1 0 1 1 0 1
0 0 1 0 0 1 1 0
1 0 0 1 1 0 1 0
1 1 0 1 1 1 1 0
1 1 1 0 1 1 1 0
0 1 0 0 1 1 1 0
0 1 1 0 1 0 0 1
1 1 1 1 1 0 0 1
0 0 0 1 1 1 1 0

```

zchaff:

```

0 1 1 1 0 0 1 1
0 0 0 0 0 1 1 1
0 0 0 0 0 0 0 1
0 0 0 0 1 1 1 0
0 0 0 0 1 0 1 1
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 1
0 0 0 1 0 1 1 0
0 0 0 0 0 0 1 1
0 0 0 0 1 1 1 1
0 0 1 0 1 1 1 0

```

**Table 5.** Counter examples generated by three verification procedures

seventh  $S(F, I)$  took only 2.8 seconds with the zchaff solver, but when we used ABC instead, it took 162 seconds.

After some investigation we discovered this was the result of an interesting effect. It turns out that when you solve  $S(F, I)$  using a set  $I$  of counterexamples produced by Zchaff, the resulting SAT problem is actually much easier than the problem you get if you use a random set  $I$ . This is because there is a lot of regularity in the counterexamples produced by Zchaff. ABC, on the other hand, because of the extensive transformations it applies, produces counterexamples that are much more irregular. Table 5, for example, shows the counterexamples produced by zchaff compared to those produced by the ABC verifier. From the figure, it is clear that whereas those produce by zchaff are very regular, those produced by ABC are much more irregular.

This experience suggested that there could be benefits from using different solvers for synthesis and verification, the results of which will be discussed in the next section.

### 4. Asymmetric Synthesize-Verify Loop

From our experiences using ABC and Zchaff, we found that they both had strengths and weaknesses. ABC is much more efficient than Zchaff in general, and is able to deliver very large performance improvements on similar problems. However, Zchaff produces more regular counterexamples, which for many benchmarks lead to simpler SAT problems. Therefore, we decided to try different combinations of solvers for the synthesis and verification.

	(Z, Z)	(ABC, ABC)	(ABC, ABCwo)	(ABC, Z)	(ABCwo, Z)
logcnt 8	(15.9, .23)	(40.8, .14)	(44.6, .14)	(5.2, .16)	(9.0, 0.27)
parity 16	(0.24, 0.1)	(1.6, 0.09)	(3.0, 0.12)	(0.19, 0.13)	(2.5, 0.08)
parity 24	> 1.5 hr	(99.7, 2.37)	(43.7, 3.404)	(28.8, 0.16)	(1.39, .27)
des	> 2.5hr	(3070, .140)	(5235, 0.19)	(2638, 0.08)	(1083, 0.3)
Karatsuba 6	(1.76, 4.23)	(4.0, 5.15)	(6.65, 1.2)	(3.32, 4.2)	(1.4, 4.37)
Karatsuba 8	(9.4, 1744)	(5.1, 67.6)	(4.7, 33.7)	(7.0, 1507)	(3.3, 1410.8)
sort 12	(<1, 23.35)	(<1, 2.26)	(<1, 2.8)	(<1, 23.3)	(<1, 16.4)
log2 16	(178.0, 0.59)	(91.7, .32)	(223.0, 0.8)	(271.51, 0.5)	(96.9, 0.8)

**Table 6.** Results from using the mixed solvers. Each column is labeled with (verifier, synthesizer)

Table 6 shows the results of these experiments for some benchmarks. The solvers used were the following: ABC, Zchaff, ABC without transformations. One of the first things we noticed, and which is not apparent from the table, is that for some of the benchmarks (logcnt, log2), there was enormous variability across runs depending on how the random number generator was seeded. The times shown for these two benchmarks are actually an average over 6 different runs. For logcnt, we saw variations of an order of magnitude in the running times. For log2, one run with abc for both synthesizer and verifier took 2.9 seconds, while another identical run took 193 seconds. For the other benchmarks, the times were more stable.

From the numbers in the table, we can make several observations. First, not all benchmarks behave the same on the different combinations of solvers. For logcnt, parity, and des, it is clear that the hybrid approach works much better than either the pure ABC, or pure Zchaff approaches.

Karatsuba and Sort are odd because for both of these benchmarks, the limiting step is not synthesis but verification. Sort at least behaves as expected, with the ABC verifier performing much faster than the Zchaff verifier. The same can not be said about Karatsuba with size 6. The different solvers perform very differently, and unlike logcnt, and log2, the times are very consistent across different runs. However, the only pattern that seems to appear is that ABC is slower than Zchaff, but ABC without the transformations is faster than both. With Karatsuba of size 8, ABC is now faster than Zchaff, but ABC without transformations is still dramatically faster than both. It is worth pointing out that the synthesis portion of Karatsuba does not seem to suffer from bad counterexamples.

Finally, log2 is quite odd. As was mentioned before, there were enormous variations between runs, much larger than for any other benchmarks, so it's hard to say anything conclusive about it, except that the hybrid scheme that worked quite well for all other benchmarks (abc, zchaff), was consistently bad for this one.

One final thing we noticed, is that for many of our benchmarks, using ABC without the transformations worked better than using it with the transformations. This may be just a matter of tuning; it may be that ABC is spending too much time starting SAT solvers and killing them before they are done.

Another idea we wanted to try along these same lines, but we could not due to lack of time, was to use a different boolean representation of the problem for synthesis and verification. Most of the infrastructure needed is there, since we had to build it in order to perform the experiments described above, but we were not able to run a full experiment in these mode.

## 5. Resizing Sketches

Although SAT-based synthesis can theoretically solve the a sketch for any arbitrary input size, experiments have shown that scalability is hard to get in practice. We examine an approach for *resizing* synthesized implementations using different techniques in order to work around this problem. Such an approach may have many facets in terms of defining the nature of the resizing step, the correctness criteria, and the approximation and heuristic techniques applicable in the solution. We focus on a simple instance concerned with implementing a fast integer mul-

$$\begin{aligned}
K\langle 2W \rangle &= \lambda x : \overleftarrow{2W}. \lambda y : \overleftarrow{2W}. \\
&\lambda x_1 : \overleftarrow{W}. \lambda x_2 : \overleftarrow{W}. \lambda y_1 : \overleftarrow{W}. \lambda y_2 : \overleftarrow{W}. \\
&\lambda a : \overleftarrow{4W}. \lambda b : \overleftarrow{4W}. \lambda c : \overleftarrow{4W}. \\
&\text{plus} \\
&\quad (\text{shl } a \ 2W) \\
&\quad (\text{plus} \\
&\quad\quad (\text{shl} \\
&\quad\quad\quad (\text{minus} \\
&\quad\quad\quad\quad c \\
&\quad\quad\quad\quad (\text{plus } a \ b)) \\
&\quad\quad\quad\quad W) \\
&\quad\quad\quad b) \\
&\quad (S\langle W \rangle \ x_1 \ y_1) \\
&\quad (S\langle W \rangle \ x_2 \ y_2) \\
&\quad (S\langle 2W \rangle \\
&\quad\quad (\text{plus } x_1 \ x_2) \\
&\quad\quad (\text{plus } y_1 \ y_2)) \\
&\quad (\text{shr } x \ W) \\
&\quad (\text{lsb } x \ W) \\
&\quad (\text{shr } y \ W) \\
&\quad (\text{lsb } y \ W)
\end{aligned}$$

**Figure 1.** Parameterized version of a synthesized implementation of Karatsuba's multiplication

tiplication algorithm known as the *Karatsuba multiplication* for large integers: this technique improves the asymptotic complexity of multiplication from  $O(n^2)$  down to  $O(n^{1.585})$  with  $n$  being the representation bit-width of the arguments. Although seemingly a simple case for synthesis, our experiments indicate that a straightforward SAT-based method cannot scale for arguments of sizes beyond a few bits. An encouraging observation is that all solutions to the Karatsuba multiplication algorithm take the same form (up to the integer representation width annotations associated with variables and operators), a fact that we wish to exploit in our process.

### 5.1 Input and assumptions

We assume that both the specification and the synthesized code are given by functional-style expressions with unsigned integer arithmetic. This is a reasonable assumption as the sketch language is indeed expressively equivalent to a restricted functional language. Specifically, the (trivial) specification of multiplication of two integers is given by

$$S\langle W \rangle = \lambda x : \overleftarrow{W}. \lambda y : \overleftarrow{W}. \text{times } x \ y \ .$$

Similarly, we assume the synthesized Karatsuba implementation to be a functional-style expression, whose types (i.e., representation widths for integers) are parameterized by some type variable, allowing linear dependencies between concrete integer widths. Suppose the sketching framework was able to synthesize a provably correct Karatsuba implementation for operands of some arbitrary width  $2W$ , for some  $W$ , namely multiplying a pair of  $2W$ -bit unsigned integers. As a preliminary step, we re-parameterize this instance with our width type

## Subexpression reduction

$$\left. \begin{array}{l} \frac{e_1 \rightarrow e'_1}{e_1 \oplus e_2 \rightarrow e'_1 \oplus e_2} \\ \frac{e_2 \rightarrow e'_2}{e_1 \oplus e_2 \rightarrow e_1 \oplus e'_2} \end{array} \right\}, \oplus \in \{+, -, \times, /, \% \}$$

$$\frac{e_1 \rightarrow e'_1}{(\lambda x.e_1)e_2 \rightarrow (\lambda x.e'_1)e_2} \quad \frac{e_2 \rightarrow e'_2}{(\lambda x.e_1)e_2 \rightarrow (\lambda x.e_1)e'_2}$$

$\beta$ -reduction

$$\frac{}{(\lambda x.e_1)e_2 \rightarrow [e_2/x]e_1}$$

## Integer arithmetic

$$\frac{e_1 = n_1 \ e_2 = n_2}{e_1 \oplus e_2 = n} , n = n_1 \oplus n_2, \oplus \in \{+, -, \times, /, \% \}$$

$$\frac{}{a \oplus 0 = a} , \oplus \in \{+, -\} \quad \frac{}{a \oplus 1 = a} , \oplus \in \{\times, /\}$$

$$\frac{}{a - a = 0} \quad \frac{}{a \times 0 = 0} \quad \frac{}{a/a = 1} \quad \frac{}{a\%a = 0}$$

$$\frac{}{a\%1 = 0}$$

$$\frac{}{(a/c)c + a\%c = a}$$

$$\frac{}{(a+b)c = ac + bc} \quad \frac{}{a \oplus b = b \oplus a} , \oplus \in \{+, \times\}$$

$$\frac{}{a - (b+c) = a - b - c}$$

**Figure 2.** Deduction rules used to prove equivalence of  $\lambda$ -expressions with integer arithmetic. The = rules stand for bidirectional derivation (equivalence)

annotations—taken from the initial sketch—yielding the program in Figure 1. Note that this includes parametrizing some sub-expressions with constants depending on the value of  $W$  (e.g., bit masking and shifting), as well as parameterizing the template of the synthesized program itself. Also note that the synthesized implementation invokes the specification multiplication function ( $S$ ) for computing sub-products.

## 5.2 Equivalence over pure integers

As a first step, we attempt to prove that the function represented by the synthesized expression is indeed semantically equivalent to the specification, with the temporary assumption that calculations are made with “pure” (unbounded) integer values and variables. For this purpose we consider a stripped-down (untyped) version of the expression in Figure 1. We also model bit-wise operations using their arithmetic counterparts, in particular representing  $shr\ x\ W$  using  $div\ x\ 2^W$ ,  $lsh\ x\ W$  using  $mod\ x\ W$ , and  $shl\ x\ W$  using  $times\ x\ 2^W$ . Note that  $W$  is a free variable in this expression.

We use a set of semantics preserving rules by which the two functions can be rewritten to an isomorphic form. These rules, shown in Figure 2, include rules for reducing sub-expressions (e.g., across function abstraction and arithmetic operations),  $\lambda$ -calculus reduction rules (e.g.,  $\beta$ -reductions), and arithmetic equivalence rules (e.g., associativity, commutativity, distributivity, quotient-remainder decomposition, etc). These rules are sufficient for reducing the (stripped-down) expression in Figure 1 to the specification expression shown in Section 5.1, by first applying normal order (leftmost-outermost)  $\beta$ -reduction, then simplifying the resulting inner arithmetic expression. The overall result is isomorphic to the specification, as it is syntactically equivalent up to variable naming.

We therefore propose a staged scheme for mechanizing the proof of equivalence for our case of restricted functional language with integer arithmetic.

**Normal order  $\beta$ -reduction.** For expressions disallowing functions to be passed as arguments (other than primitive functions, and the initial name binding done for each internal function)—as in the Karatsuba case—normal order reduction is expected to terminate in a normal form as no recursion can occur. We therefore reduce both our synthesized expression and the specification expression this way, up to a point where no further reduction is possible.

It is important to note that obtaining a normal form by symbolic reduction is not necessarily feasible given recursive (even primitive recursive) expressions, in particular such that depend on the value of some input, be it either the integer width parameter ( $W$ ) or some formal argument to the function. Indeed, many loops currently present in sketched implementations are due to our use of bit-vector numerals representation, but these can be abstracted away by using atomic integer representation (this is actually done in our example). Nonetheless, other instances of argument-dependent recursion can still occur in general, and possible ways to deal with it are discussed later.

At this point, we expect to get a functional expression taking a sequence of arguments (of the same number as those taken by the specification) and using them in a purely arithmetic expression.

**Arithmetic simplification.** Given the resulting (fully reduced) functional expressions, we denote by  $K'$  the arithmetic function corresponding to the synthesized function, and by  $S'$  the one corresponding to the specification (in this case primitive multiplication). We then hand the expression  $K'(a, b) - S'(a, b)$  (with  $a$  and  $b$  being arbitrary fresh symbols) to an automated arithmetic simplification procedure, hoping that it is able to simplify it to zero. So far we experimented with two simplification algorithms, implemented in different frameworks for symbolic manipulation of arithmetic expressions, each of them bearing different capabilities and limitations.

**Mathematica.** This commercial system was able to simplify the expression to zero, but required that we add the rewrite rule for integer quotient/remainder ourselves.

**Maxima.** This open-source system was generally able to simplify the expression to zero, yet it took an explicit instruction to make it fully expand expressions using the distributivity rule.

This leads us to believe that the simplification algorithm of the latter system is not designed for conducting effective exploration of distributivity induced search space expansion, compared to the one used in the former system. On the other hand, it does seem to be sufficiently elaborate to include rules missing in the other system, and the fact that its simplification algorithm is at the public domain probably implies better adaptivity for our needs, as well as more transparency in understanding the way a rewrite system of this kind works.

By the end of this phase we have established that the “abstract” implementation (using pure integers) is functionally equivalent to the specification.

## 5.3 Concrete integer representation

In general, the lack of overflows and underflows (i.e., well-typedness) of the synthesized expression stems from the SAT-based verification phase.<sup>1</sup> Nonetheless, this fact is only assured to hold for the particular width for which the problem has been resolved. We now need to show that the integer widths associated with variables indeed lead to a well-typed program. As opposed to semantic equivalence, where we make use of a definitive decision procedure for showing equivalence, here we approximate type safety using conservative rules.

<sup>1</sup> More precisely, we can model the requirement for no overflows while formulating the SAT problem.

## Axioms

$$\cdot \vdash e : \overrightarrow{[\log_2(e+1)]}$$

## Subtyping

$$\frac{\vdash_A w_1 \leq w_2 \quad \cdot \vdash e : \overrightarrow{w_1}}{\cdot \vdash e : \overrightarrow{w_2}}$$

## Abstraction / Application

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau'} \quad \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau}$$

## Arithmetic

$$\frac{\Gamma \vdash e_1 : \overrightarrow{w} \quad \Gamma \vdash e_2 : \overrightarrow{w}}{\Gamma \vdash \text{plus } e_1 e_2 : \overrightarrow{w+1}}$$

$$\frac{\Gamma \vdash e_1 : \overrightarrow{w} \quad \Gamma \vdash e_2 : \overrightarrow{w}}{\Gamma \vdash \text{times } e_1 e_2 : \overrightarrow{2w}}$$

$$\frac{\Gamma \vdash e_1 : \overrightarrow{w} \quad \Gamma \vdash e_2 : \overrightarrow{w} \quad (\vdash_A e_1 \geq e_2)}{\Gamma \vdash \text{minus } e_1 e_2 : \overrightarrow{w}}$$

$$\frac{\Gamma \vdash e : \overrightarrow{w}}{\Gamma \vdash \text{shr } e w' : \overrightarrow{w-w'}} \quad \frac{\Gamma \vdash e : \overrightarrow{w}}{\Gamma \vdash \text{shl } e w' : \overrightarrow{w+w'}}$$

$$\frac{}{\Gamma \vdash \text{lsb } e w' : \overrightarrow{w'}}$$

**Figure 3.** Inference rules for integer representation width. The  $\vdash_A$  notation implies a proof adherent to the theory of integer arithmetic

We extend our functional-style expressions with annotations of integer representation width types  $\overrightarrow{W}$ , as well as function types (which are mappings thereof). The obvious subtype relation corresponds to a total order on representation width expressions. We use our knowledge of the effect of integer operators to come up with a set of inference rules, shown in Figure 3. Note that the rule for inferring width of an integer subtraction optionally requires an added premise in order to guarantee the absence of underflow. Such a proof generally relies on an additional decision procedure for integer arithmetic, which is generally incomplete. However, it is easily provable for the case of Karatsuba.

We then apply these rules to infer types for the stripped-down version of the expression in Figure 1—note, however, that we keep the outermost argument types intact, as these define the interface for the sketch. The resulting fully-typed expression is omitted for brevity, and we suffice for addressing interesting properties of it.

A first observation is the fact that inferred types for inner variables are tighter than those given by the sketch (and therefore present in the synthesized implementation): specifically, we associate widths of  $2W$  (respectively,  $2W+2$ ) to  $a$  and  $b$  (respectively,  $c$ ) instead of  $4W$ . This isn't surprising, as the inference process attempts to obtain the tightest possible (conservative) width annotations. Furthermore, this does not impose any practical difficulty, since we can safely expand the types for the inner variables—specifically,  $a$ ,  $b$ , and  $c$ —following our subtype relation, knowing that the type of any expression which uses them can remain as is.

Next, we observe that the whole typed function is such whose return type is potentially larger (wider) than that of the sketch, and (accordingly) that of the specification: in particular, our system was able to infer a return type of width  $4W+1$  whereas the expected return type is of width  $4W$ . This is an expected effect of using a conservative set of rules for inferring types. Fortunately, we can reuse a previously learned equivalence property of the two expressions—the parameterized implementation and the specification—in order to tighten this type annotation: we can safely tighten the return type to  $4W$ , as we know that any output of the sketched implementation is

equal to that of the specification, and therefore has the same (actual) type.

Another potential deviation (which we did not encounter here) is where inferred types for inner variables and expressions are wider than those expected. In this case, we simply propose to adopt the new inferred types, instead of the previously provided ones, this way assuring sanity of the implementation yet not harming its outcome in any way.

By the end of the second phase we have established that the “concrete” implementation is equivalent to the “abstract” implementation, for any  $W$ . Combining this fact with the result of Section 5.2 we conclude that the former is equivalent to the specification.

## 5.4 Beyond Karatsuba

In the process of proving equivalence of a parameterized Karatsuba formulation, for arbitrary argument widths, we are in fact hiding an implicit inductive argument. Specifically, by using the specification function itself for recursive computation of sub-products, we avoid a full-scale proof by induction, namely a base case (in this case, a single bit) and an inductive case over some explicitly defined well-founded order (in this case, a chain of doubling argument widths). Here we could afford such a detour mainly due to the fact that the only recursive invocation used in the implementation is to a descendant instance of itself. In the general case, however, we might encounter recursions at sub-expression level, which could not be substituted by a reference implementation. Furthermore, when such chains depend on the value of some input, we cannot hope that straightforward symbolic reduction (i.e.,  $\beta$ -reduction) will converge. Therefore, in such cases, a more involved technique needs to take place. We believe that making inductive arguments explicit in the derivation process, namely requiring that the derivation adheres to a well-founded descending sequence, is the right way to reduce such instances.

In addition to the special-case recursive form, the case of a multiplication algorithm of this kind seems particularly simple to handle because it only deals with theory of integer arithmetic. Obviously, aiming at handling different theories, or even extension of arithmetic (e.g., bit-vector operations, operations on matrices, etc), we will need to extend our rule base accordingly. Alternatively, we might need to consider other means for proving equivalence, such as reasoning on properties of interest via abstraction. Although the problem of proving equivalence is generally undecidable, we hope to be able to cover a considerable set of sketch problems in practice.

## 5.5 Related work

Our approach for generalizing the verification step for arbitrary problem sizes relates to previous work on transformation-based synthesis, such as [1, 4]: in these frameworks, correctness preserving transformations—namely, reductions concerned with functional abstraction, as well as theory-specific rules—are applied to some declarative specification in attempt to obtain a (hopefully) efficient variant with equivalent semantics. Our approach differs from that as we exploit our a-priori knowledge of *some* correct implementation, which we *expect* to be semantically equivalent to the specification. This notion implies a transformational process which is mostly concerned with *simplification*, rather than arbitrary exploration, in turn suggesting the advantage of an informed search. Nonetheless, we are yet to investigate further means to ensure faster convergence of the rewriting process, given this special scenario. A slight generalization of that suggests that parameterized sketches could be obtained directly by applying a heuristic transformation process, such that is directed towards finding a variant which matches some “implementation with holes” (i.e., the sketch). By that, we may be able to avoid the SAT-based synthesis/verification stage altogether, and head straight towards the ultimate goal. Such an approach, however, requires considerable formalism and practical infrastructure before it can be put to practice.

## 6. Conclusion

Our project was quite successful in most of the goals we set out to accomplish. For example, for the floating point encoding the results were very positive. We were able to show that one can implement approximate equivalence for small but real floating point benchmarks using only bit blasting.

In terms of the use of circuit verification techniques for encoding and solving the synthesis and verification problems, the results were more mixed. It is clear that there are potentially some very large performance gains to be achieved, but attaining them consistently in practice for many of our benchmarks proved to be very difficult. We believe the main reason for this is the large number of XOR gates present in all the benchmarks. The only benchmark that does not involve large numbers of XORs is sort, and incidentally, it is the only benchmark where we get clean consistent results that look as we expected in the beginning: ABC is the fastest, followed by ABC without transformations, and Zchaff is the slowest. ABC is not designed to handle circuits with such large numbers of XOR gates, which probably explains the results.

Finally, we were able to show that it is possible to generalize the results from sketches from the finite domain, using algebraic (and other) simplification techniques to prove that they can be extended to arbitrary sizes, at least for one interesting example. This is a very important result, because it will allow us to move our solver out of the realm of small bit-level benchmarks and make it usable for synthesis problems of more practical sizes.

## Acknowledgments

We would like to thank Ras Bodik, Alan Mischenko, Satrajit Chatterjee, Robert Brayton and Sanjit Seshia, who helped us a lot with various stages of the project.

## References

- [1] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM (JACM)*, 24(1):44–67, 1977.
- [2] C. Lomont. Fast inverse square root. Technical report, Purdue University, 2003. Available at <http://www.math.purdue.edu/~clomont/Math/Papers/2003/InvSqrt.pdf>.
- [3] A. Mischenko, S. Chatterjee, and R. Brayton. Dag-aware aig rewriting: A fresh look at combinatorial logic synthesis. In *DAC 2006*, 2006.
- [4] D. R. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering (TSE)*, 16(9):1024–1043, 1990.