

Symbolic Proof Generation for Resizing Sketches

Gilad Arnold and Liviu Tancau

Abstract

We propose an approach for resizing finite implementations generated by the SKETCH synthesis framework. Our solution generates a formal proof of equivalence between size-parameterized versions of a naive specification and an efficient (synthesized) implementation, for any problem size. As opposed to previous attempts, the new framework integrates elaborate semantic properties—*e.g.*, machine numeral representation (bounded integers), two’s complement arithmetic, and recursive invocations—within a single formalism. As far as we know, this is the first instance of hybrid synthesis, where symbolic proof generation is applied on top of a combinatorially generated solution “seed”.

1. Introduction to Sketching

Sketching is a program synthesis paradigm in which a programmer expresses an outline of the implementation, called a *sketch*. The details missing in the sketch are filled in by the compiler such that the result is functionally equivalent to a separately provided specification. Programming by sketching is supported by the SKETCH language [3], an imperative language with “hole” and “alternation” constructs. These can be used in place of hard-to-get-right expressions, such as index expressions and loop bounds. The synthesizer infers the content of holes and decides alternations using a SAT-based combinatorial search over the space of possible sketch completions. Thanks to the combinatorial search, the synthesizer requires no domain-specific knowledge and can therefore be applied to a wide variety of programming problems, as long as they compute *finite programs*, *i.e.*, functions that take inputs of bounded size and perform a finite computation.

1.1 The Karatsuba multiplication example

We use the example of large integer multiplication to demonstrate the usability of SKETCH. While the traditional, grade school multiplication algorithm takes $O(n^2)$ to complete for n -bit large inputs, more sophisticated algorithms can improve on that significantly. One such popular algorithm is the Karatsuba multiplication [2]. Decomposing the operands x and y into their most- and least-significant halves— x_1 and x_2 , and y_1 and y_2 , respectively—the algorithm computes the expression

$$x_1y_12^n + (x_1y_1 + x_2y_2 - (x_1 - x_2)(y_1 - y_2))2^{n/2} + x_2y_2 .$$

It can be easily seen that this is (arithmetically) equivalent to xy . Nonetheless, the result is obtained using *three* distinct subsidiary multiplications—namely x_1y_1 , x_2y_2 , and $(x_1 - x_2)(y_1 - y_2)$ —instead of four, as done in the traditional algorithm. The complexity of the remaining operations is negligible. Therefore, applying this technique recursively results in an overall time complexity of $O(n^{\ln 3 / \ln 2})$ (roughly $O(n^{1.585})$).

Although the above algorithm is easy to understand, it is not as simple to code in practice: Machine-level representation of integers and the associated overflow and underflow semantics require that such implementation be crafted to address all possible corner cases. Even then, the result is not necessarily free of bugs unless otherwise (formally) proven. Therefore, the incurred development process is likely to be tedious and longer than expected.

The special features of SKETCH make it a suitable platform for implementing the above algorithm more effectively. In the following code

the programmer essentially captures the “core idea” of decomposition that underlies the Karatsuba method, but the re-composition of the final result is coded in a rather “loose” way. The programmer takes advantage of the special ?? operator, to denote “some value”, and the special [e1 |e2 |...] alteration construct, to denote “any of the given expressions.” Also notable is the **loop** (e) { ... } statement, which unrolls the body of the loop as many times as specified by the value of the expression e. Note that subsidiary multiplications are done by invoking the **mult** function, which could stand for either standard multiplication or a descendant instance of the Karatsuba algorithm for arguments of half the size.¹

```
// Karatsuba decomposition.
bit[w] x2 = x[0:w], x1 = x[w:w],
      y2 = y[0:w], y1 = y[w:w];
bit[4 * w] t1 = mult (x1, y1), t2 = mult (x2, y2),
          t3 = mult (x1 - x2, y1 - y2);
// Re-composition (under-specified).
bit[4 * w] res = 0;
loop (??) {
  bit[4 * w] t = [ t1 | t2 | t3 ] << (w * ??);
  res += [ t | -t ];
}
return res;
```

In addition, the programmer provides the following naive specification of the multiplication function.

```
bit[4 * w] res = x * y;
return res;
```

These are fed into the SKETCH compiler, along with a requirement of functional equivalence between the resolved sketch and the specification, for a fixed value of w .

1.2 Sketch resolution

The full details of how the SKETCH compiler infers values to complete a provably correct implementation are given in [3]. In the following we give a brief outline thereof.

Compilation of SKETCH code first transforms the imperative-style functions into pure Boolean functions. This includes modeling arithmetic and control data (standing for the special sketch constructs). By the end of this phase we are able to re-state the sketching problem as the two-quantified Boolean formula (2QBF) problem

$$\exists C . \forall x : 2w, y : 2w . f_{\text{sketch}}^w(C, x, y) = f_{\text{spec}}^w(x, y) ,$$

where C is the set of controls (Boolean values substituting holes and undetermined choices), x and y are inputs of size $2w$, $f_{\text{sketch}}^w(C, x, y)$ is the w -instantiated sketch function taking a set of controls and two inputs, and $f_{\text{spec}}^w(x, y)$ is the corresponding specification function.

The compiler then further reduces the problem into counter-example guided refinement of Boolean satisfiability (SAT) problems. Starting with some random values for inputs, a synthesis step attempts to assign

¹The exact sketch we used in this work differs slightly from this simplified version, as it checks for underflow before deciding the order of the subtraction expressions. This is necessary when using tighter integers sizes, which are required in the fully recursive Karatsuba algorithm. We avoid it here for clarity.

values to control variables while ensuring functional equivalence. A following verification step tries to find another set of inputs for which the equivalence does *not* hold given the current set of controls. This feedback loop is repeated until the synthesis step fails (*i.e.*, the sketch cannot be resolved) or the verification step fails (*i.e.*, equivalence is satisfied for all inputs). In the latter case, the most recent control values are plugged back into the original sketch, resulting in a full implementation. This process also yields a combinatorial proof of correctness.

By filling out the right values in place of undetermined expressions, the SKETCH compiler is able to generate the following elaborate Karatsuba re-composition (slightly simplified through partial evaluation).

```
bit[4 * w] res = 0;
res += t1 << (w * 2);
res += t1 << w;
res += t2 << w;
res += -(t3 << w);
res += t2;
return res;
```

1.3 The scalability challenge

The combinatorial synthesis process done in SKETCH can theoretically extend to any fixed problem size (*i.e.* value of w). In practice, however, the use of Boolean satisfiability blows up very quickly due to bloating incurred by Boolean arithmetic, a growing set of counter examples, and the hardness of SAT. Consequently we are unable to resolve the Karatsuba sketch for $w > 3$, making it impractical for real-life purposes.

In the following sections we describe our approach for extending finite sketches to arbitrary sizes, with the Karatsuba multiplication as an example. With this approach we are able to produce a symbolic proof of the functional equivalence between a sketched implementation—generated by the combinatorial synthesis process—and the specification, for *any* problem size.

2. Resizing Sketches

We assume the successful resolution of the Karatsuba sketch shown above, for some (small) w . This means that, for this fixed problem size, the SKETCH compiler has been able to generate the right set of controls C that satisfy the equivalence claim. While we cannot synthesize implementations for bigger problems using the same approach, we observe (informally) that the Karatsuba algorithm works the same way for operands of *any* representation width. Therefore, it should be possible to *resize* the resolved sketch by means of proving that it works correctly given any binding of w .

Fixing the above inferred control values C , we can now restate the correctness criteria to be

$$\forall w, x : 2w, y : 2w . f_{\text{sketch}}^w(C, x, y) = f_{\text{spec}}^w(x, y) . \quad (1)$$

The notable difference from the previous quantified formula is that we now wish to prove functional equivalence between the resolved sketch and the specification for *any* input size.

2.1 Previous work

In previous work [4] we presented an ad-hoc attempt to solve the Karatsuba resizing problem. We first applied a sequence of β -reductions to eliminate functional abstractions from both the sketch and the specification, resulting in a universally quantified equivalence over pure arithmetic expressions. This equivalence theorem was fed into a symbolic simplification framework (*e.g.*, Mathematica) which was able to reduce it to true. As a complementary step we applied a simple set of typing rules to the sketched program to ensure that no values exceed their assigned representation widths (*i.e.*, no overflows occur). This way we assured that the actual computation with machine-represented integers behaves the same as if computed with pure integers, thus completing the correctness argument.

While the above was enough to resize—in a manual and somewhat hand-wavy manner—the Karatsuba multiplication algorithm, it

had several problematic aspects. First, it works by separating the pure arithmetic equivalence from the machine-level representation thereof. Although this generally works, we believe that in most cases a conservative type checker would fail to assert the safety of otherwise perfectly correct code, due to the loss of precision across arithmetic operations and its inability to re-use arithmetic related knowledge (*e.g.*, bounds on certain values, dependencies between values, *etc.*) when propagating typing judgments. Another artifact of this limitation is that type checking was not fundamentally “easier” than the first equivalence proving stage, since certain typing rules required arithmetically involved premises to hold prior to successful application of the rule.

Second, this approach could only handle recursive functions of a very specific form, where the (single) recursive invocation can be substituted with an instance of the specification function. Consequently, the inductive argument underlying the proof for a recursive function is only implicitly stated, and the proof not fully established. We believe this feature limits the applicability of the mentioned approach for many practical problems.

2.2 A unified decision procedure

In this work we propose a more rigorous approach to proving functional equivalence of functions involving recursive computations with machine-level arithmetic. Hopefully, by integrating machine-level semantics with arithmetic related proofs, and by using inductive arguments to directly address recursive computations, we may be able to handle a wider range of “sketchable” programs in the future. We believe that the dependent-type oriented, functional-style proofs that Coq can handle, as well as its relatively elaborate standard library, make it a good vehicle for building such a system. Implementing this framework goes through the following stages.

Infrastructure. We elaborate on the bounded bit-vector arithmetic formalism that was implemented by Adam Chlipala in the *ProofOS* project, described in [1]. This is an elaborate and general-purpose baseline, which we extend with additional operations, as well as lemmas to describe properties of expressions in this domain theory.

Translation. We need to translate functions written and compiled in SKETCH to our new Coq-based framework. Since implementing a complete translation procedure is beyond the scope of this work, we consider a precise manual translation to be sufficient for now.

Proving equivalence. We express the equivalence formula (1) using the following Coq theorem.

```
Theorem sk_karatsuba_eq :
forall (w : nat) (x y : Bvector (2 * w)),
  sk_karatsuba w x y = sp_multiplyFull w x y.
```

Here, `Bvector` is a width-dependent bit-vector type used to represent machine integers. The function `sk_karatsuba` is the resolved Karatsuba sketch, and `sp_multiplyFull` the ordinary reference implementation for integer multiplication (specification). Both are translated from their respective description in SKETCH.

Automation. Our ultimate goal is to be able to prove the complete resizing argument with minimal user intervention. Ideally, simplifying the goal and proving the equivalence theorem can be performed by applying a certain sequence of tactics. Although we did not manage to achieve automation within the limited scope of this project, we outline our plan for obtaining a certain degree of it in Section 4.

In the following we describe our implementation towards a decision procedure as above.

3. Proving Resizability in Coq

We describe the major components in our Coq implementation for proving resizing of the Karatsuba multiplication sketch.

3.1 Domain theory of machine-level arithmetic

In the Coq representation of sketches, numbers are modeled using the dependent type `Bvector` from the Coq standard library. `Bvector` is

simply a list of Boolean values parametrized by the length of the list. This matches up nicely with the representation used in SKETCH, which is the bit array. The length of an array in SKETCH is always known (unlike in C), therefore translation to Bvector is trivial. Bvector is also used to model machine registers in ProofOS, meaning that we were able to reuse some of the operators and theorems defined in the Asm module of that project, as well as add our own.

Our extensions to the Asm library are contained in the module BArithExt. They comprise a set of operators (including infix notations) to be used in translated sketches, as well as lemmas reasoning about the operators. These are summarized in the following table.

Operation	Notation	Operands
addition	++	Two bit-vectors of equal size
subtraction	--	Two bit-vectors of equal size
multiplication	**	Two bit-vectors of equal size
negation	bneg	A bit-vector
left-shift	<<	A bit-vector and a nat
right-shift	>>	A bit-vector and a nat
cast	bcast	A bit-vector and a nat

Note that all binary operators take operands of the same size, and the size of the resulting value is always the same as the size of the operand(s). Changing the size of a bit-vector is only possible via the cast operator, which will either remove bits from the most significant positions or pad the operand with zeros, depending on whether the size argument is less than or greater than the size of the bit-vector argument. All operators treat their arguments as unsigned binary integers, meaning that right shifting does not perform sign extension and the negation operator simply computes the two's complement. Division operators (quotient and modulo) were not defined because it is difficult to reason about them.

The implementation of most operators involves converting the bit-vectors to nats (natural numbers), performing the operation using functions from the Arith library, then converting the result back to bit-vector. Therefore, we can speak of the *value* of a bit-vector as being the natural number encoded by it. This number may not necessarily be equal to the nat that was used to create the bit-vector since the conversion only extracts as many least significant bits as can fit in the vector. However, when proving things about arithmetic expressions on bit-vectors it is useful to prove that the conversion is indeed exact—*i.e.*, no bits are discarded—whenever that is in fact true. This normally simplifies the resulting Coq expression. Most of the “support lemmas” defined in BArithExt were written for that purpose. They are described below.

bsub_safe	The value of $x - y$ is equal to the value of x minus the value of y , as long as $x \geq y$ (<i>i.e.</i> , no underflow).
mult_upper	The multiplication of two bit-vector values (of width w) is bounded above by 2^{2w} .
sub_upper	The subtraction of two bit-vector values (of width w) is less than 2^w .
add_upper	The addition of two bit-vector values (of width w) is less than 2^{w+1} .

We also wrote a lemma stating that left shifting a bit-vector of size w by s is equivalent modulo 2^w with multiplication by 2^s (eqMod_bshift). Deriving a similar lemma for right shifting proved too difficult due to insufficient library support for reasoning about division. It was also unnecessary for our goal.

3.2 Proof tactics

In order to promote our goal of devising an automated prover, we wrote several tactics that perform various simplifications or discharge certain subgoals arising from application of rewrite rules. Some of these general-purpose tactics, contained in the module BTactics, are described below.

pow2BoundProver Automatically proves subgoals of the form $x < \text{pow2 } n$ or $x \leq \text{pow2 } n$. For the latter form, the tactic converts it into a strict inequality first, then invokes itself. For strict inequalities, it does pattern matching on x and applies `mult_upper`, `sub_upper`, or `add_upper` (defined in BArithExt) if x is computed using an arithmetic operation; or it applies `btn_upper` (defined in the Asm library) if x is the value of a bit-vector. Many lemmas can yield subgoals provable by `pow2BoundProver`, making this a good “cleanup” tactic.

doSplit Converts a bit-vector of even size into a linear combination of its two halves; also introduces aliases for the two halves as hypotheses and rewrites the goal accordingly.

rewriteSub Attempts to simplify subtractions using `bsub_safe`, as long as it can prove the absence of underflow (using `auto`).

destructAll Destructs all comparisons in the goal, such as `eq_nat_dec` or `le_gt_dec`; used to get rid of conditionals, which simplifies the goal by splitting it into subgoals corresponding to all possible branches.

elimConv Tries to eliminate conversions from nat to bit-vector then nat again, by proving that no truncation occurs, hence simplifying the expression to the original nat; uses `ntb_invert` (from Asm) for rewriting and `pow2BoundProver` for providing the proof obligation.

3.3 Proof of single Karatsuba decomposition

The main theorem we set out to prove is the correctness of the Karatsuba sketch for arbitrary input sizes, expressed in (1). The definitions of the two functions are shown below.

Definition `sp_multiplyFull` w ($x \ y : \text{Bvector } (2 * w)$)
 $: \text{Bvector } (4 * w) := \text{bmult_c } (4 * w) \ x \ y.$

Definition `sk_karatsuba` ($w : \text{nat}$) ($x \ y : \text{Bvector } (2 * w)$)
 $: \text{Bvector } (4 * w) :=$
`let` $n2 := 4 * w$ `in`
`let` $x1 := \text{upper_half } w \ x$ `in`
`let` $x2 := \text{lower_half } w \ x$ `in`
`let` $y1 := \text{upper_half } w \ y$ `in`
`let` $y2 := \text{lower_half } w \ y$ `in`
`let` $\text{cmpx} := \text{ble } x2 \ x1$ `in`
`let` $\text{dx} := \text{if } \text{cmpx} \ \text{then } x1 \ - \ x2 \ \text{else } x2 \ - \ x1$ `in`
`let` $\text{cmpy} := \text{ble } y2 \ y1$ `in`
`let` $\text{dy} := \text{if } \text{cmpy} \ \text{then } y1 \ - \ y2 \ \text{else } y2 \ - \ y1$ `in`
`let` $\text{tmp2} := \text{if } (\text{xorb } \text{cmpx } \text{cmpy}) \ \text{then}$
 $\text{bcast } n2 \ (\text{sk_multiplyHalf } \text{dx } \text{dy}) \ \text{else}$
 $\text{bneg } (\text{bcast } n2 \ (\text{sk_multiplyHalf } \text{dx } \text{dy}))$ `in`
`let` $\text{tmp1} := \text{bcast } n2 \ (\text{sk_multiplyHalf } x1 \ y1)$ `in`
`let` $\text{tmp3} := \text{bcast } n2 \ (\text{sk_multiplyHalf } x2 \ y2)$ `in`
 $\text{tmp1} \ll (2 * w) \ ++ \ (\text{tmp1} \ ++ \ \text{tmp2} \ ++ \ \text{tmp3}) \ll w$
 $\ ++ \ \text{tmp3}.$

The proof is carried out by manipulating both sides of the equality, performing simplification, rewriting, and cancellation until they become the same. The most significant steps follow.

1. Unfold bit-vector operators to expose operators on nats.
2. Split x and y into linear combinations of their halves, *e.g.*
 $x = x2 + (\text{pow2 } w) * x1.$
3. Distribute the multiplication in the specification and rewrite it in canonical form with respect to `pow2 w`.
4. Simplify with `elimConv`.
5. Cancel out $\text{tmp1} \ll (2 * w)$ and tmp3 from the right-hand side with equivalent terms from the left-hand side.
6. Eliminate conditionals by doing case analysis with `destructAll`.

7. Simplify subtractions with `rewriteSub`.
8. Convert negations to subtractions from 2^{4w} .
9. Convert the goal to equality of integer expressions modulo 2^{4w} .
10. Instantiate a multiplier that makes the equality hold (either 0 or 1).
11. Use the `ring` tactic to prove the equality by rewriting both sides into some canonical form.

Some parts of the proof are automated (*e.g.*, invoking `eLimConv`), while others required goal-specific manipulation (*e.g.*, re-arranging the right-hand side and canceling out parts that appeared in the left-hand side). However, in an ideal scenario the only thing that should be manually specified in the proof is the splitting of x and y .

3.4 Proof of fully-recursive Karatsuba

The recursive version of the Karatsuba sketch is identical to the one already mentioned in all but three regards.

- w is replaced with `pow2 n`.
- When n is zero, it performs basic multiplication.
- In the general case, it calls itself rather than `sk_multiplyHalf` to do half-size multiplications.

In order to avoid complications arising from using `Fixpoint` with dependent type arguments, we defined the recursive function using tactics. The skeleton of the definition is shown below.

```

Definition sk_karatsubaRec (n : nat)
  (x y : Bvector (2 * (pow2 n))) : Bvector (4 * (pow2 n)).
  induction n as [| n sk_karatsubaRec]; intros.
  exact (bmult_c 4 x y).
  exact ( ... ).

```

Defined.

The proof relies on the equivalence theorem for the non-recursive case, hence is much simpler, as described below.

1. Perform induction on n .
2. Trivially prove the base case.
3. Unfold/simplify the function, then replace the recursive calls with multiplications by invoking the inductive hypothesis.
4. Apply the (non-recursive) Karatsuba equivalence theorem after doing some simple rewrites to bring the goal to the right form.

A complication arises while using `simpl` to perform simplification in the third step. Although it performs the desired work of unfolding the sketch function and eliminating inlined `fun` and `fix` definitions, it also has the negative side-effect of unfolding multiplications such as $4 * (\text{pow2 } w)$ into a sequence of additions, even adding some unnecessary $+ 0$ operations. Unfortunately, these over-simplifications cannot be undone using rewrite rules, presumably due to type checking issues with dependent types. The workaround we came up with was to use the `change` tactic to rewrite the goal into the desired form. This had the obvious disadvantage of having to duplicate the goal (a long expression) in the proof, which not only looks bad but will break easily if the goal is changed in any way.

4. Evaluation and Future Work

Using the infrastructure and specific proof techniques described in Section 3 we managed to express and prove resizability of the fully recursive Karatsuba multiplication sketch for any input size. By that we have succeeded in solving the particular problem used to motivate our project. In the following we discuss several issues that need to be addressed prior to introducing a practical resizing component into `SKETCH`.

Controlled simplification. As shown in Section 3.4, in some cases we have experienced unwelcome over-simplified expressions resulting from the `simpl` tactic. While we were generally able to avoid those by changing the goal into its desirable form, such a technique is

unlikely to be useful in an automated setting. Instead, we would like to explore the construction of custom rewriting tactics, namely such that lead to a canonical form that is better suited to our purposes. This will eliminate the need for some of the manual rewriting incurred by the current solution.

Simplifying across modulo-equality. *Coq*'s standard library has several general-purpose simplification tactics, as well as arithmetic specialized tactics—such as `Ring`—to automatically prove equality goals over certain mathematical domains. In our case, however, it was not until we converted the goal to pure integers (\mathbb{Z}) that automatic simplification became applicable. Instead, we have applied specific rules designed for modulo-equality, in a manual fashion. While this was sufficient to prove the particular Karatsuba goal, we consider this an undesirable solution. We believe—currently without much confidence—that it is possible to make this class of equalities subject to automatic simplification given that sufficient domain-related properties are proved. One direction is the use of *Coq*'s `setoid` feature to qualify modulo-equality as an admissible equality predicate for automatic rewriting. Another direction is deriving an instance of the `Ring` tactic to work for bounded bit-vectors. None has been attempted with encouraging results as of yet.

Improved applicability. The Karatsuba example we have experimented with easily lends itself for resizing: It has a single function pertaining a simple recursive structure, and the size parameter corresponds with it cleanly (*i.e.*, arguments size are 2^n with n being the depth of the recursion). Although this is challenging for itself, it is clear that many other programs—and, moreover, many sketches we've already written—do not incur such clean characteristics. Instead, dependencies of values with the recursive structure are often obscured, sometimes boiling down to “magic numbers” within the code. Such instances are hard (if not impossible) to prove in an inductive manner. To address this wider range of problems, our efforts need to be two-fold: First, we need to extend the coverage of our framework in terms of types of arithmetic “scenarios” that can be covered by our lemmas and tactics. Furthermore, we will need to propose techniques for effective coding of sketches, such that make resizing applicable to (even slightly) more “obfuscated” cases.

Better use of combinatorial proof component. In our solution, we make use of the resolved sketch in considering the set of controls to be fixed when proving the generalized equivalence. Nonetheless, our *Coq*-based solution does not take any advantage of the combinatorial proof that is generated (or can be generated) by the SAT-based resolution process: Instead, we ignore any such a-priori knowledge and prove the functional equivalence from scratch. Indeed, it seems unlikely that such combinatorial proof, which relates to low-level artifacts of the original high-level functions, can be effectively re-used in the symbolic proof process. Yet we believe that this conjecture deserves a better examination before the direction is disbanded (or adopted).

We plan to cover the above aspects in our future research. As far as we can tell, using *Coq* for implementing this solution has been beneficial and cost-effective, due to the convenience of *Coq*'s functional style constructs, the interactive development process, and the rich standard library.

5. Conclusion

We described an approach for resizing finite arithmetic computations over machine integers, that are synthesized using the `SKETCH` compiler, for arbitrary problem sizes. We motivated and demonstrated our solution using the Karatsuba multiplication algorithm, yet we believe that—with sufficient elaboration—the same technique can apply to a wider class of practical computations involving integer arithmetic. Finally, using *Coq* to implement our decision procedure was an appropriate choice, due to its expressiveness and (relative) ease of development.

Acknowledgments

We thank Adam Chlipala for advising and sharing his experience with us while working on this project.

References

- [1] A. J. Chlipala. Modular development of certified program verifiers with a proof assistant. In *ACM SIGPLAN International Conference on Functional Programming*, pages 160–171, 2006.
- [2] A. A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akad. Nauk*, 145:293–294, 1962.
- [3] A. Solar-Lezama, L. Tancau, D. Turner, R. Bodik, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.
- [4] L. Tancau, A. Solar-Lezama, and G. Arnold. Extending the applicability of sketching. EE219C course project, EECS Department, UC Berkeley, May 2006.