The Sketching Approach to Program Synthesis

Armando Solar-Lezama

Massachusetts Institute of Technology

Abstract. Sketching is a new form of localized software synthesis that aims to bridge the gap between a programmer's high-level insights about a problem and the computer's ability to manage low-level details. In sketching, the programmer uses partial programs to describe the desired implementation *strategy*, and leaves the low-level details of the implementation to an automated synthesis procedure. This paper describes the sketching approach to program synthesis, including the details of the SKETCH language and synthesizer. The paper will then describe some of the techniques that make synthesis from sketches possible, and will close with a brief discussion of open problems in programmer guided synthesis.

1 Introduction

Sketching is a new form of localized program synthesis that allows programmers to express their high-level insights about a problem by writing a *sketch*—a partial program that encodes the structure of a solution while leaving its low-level details unspecified. In addition to the sketch, programmers provide the synthesizer with either a reference implementation or a set of test routines that the synthesized code must pass. The SKETCH synthesis engine is able to derive the missing details in the sketch to produce a working implementation that satisfies the correctness criteria established by the programmer. By combining the programmers insight expressed in the sketch with the synthesizer's ability to reason exhaustively about all the low-level details of the problem, sketching allows complex algorithms to be implemented with minimal human effort.

To illustrate the use of sketching on an interesting programming problem, consider the problem of reversing a linked list. It is relatively easy to write a recursive solution to this problem, but the performance of the simple implementation is likely to be unacceptable. A more efficient implementation must use a loop instead of recursion, and must construct the new list backwards to avoid the linear storage. Sketching allows the programmer to express these insights as a partial program without having to think too much about the details of the implementation.

```
1: #define LHS {| tmp | (l | nl).(h | t)(.next)? |}
2: #define LOC {| LHS | null |}
3: #define COMP {| LOC ( == | != ) LOC |}

    list reverseEfficient(list l){
4:      list nl = new list();
5:      node tmp = null;
6:      bit c = COMP;
7:      while(c){
8:          repeat(??)
9:              if( COMP ){ LHS = LOC; }
10          c = COMP;
        }
    }
```

```
// test harness
void main(int n){
  if(n >= N){ n = N-1; }
  node[N] nodes = null;
  list l = newList();
  //Populate the list, and
  //write its elements
  //to the nodes array
  populateList(n, l, nodes);

  l = reverseSK(l);

  //Check that node i in
  //the reversed list is
  //equal to nodes[n-i-1]
  check(n, l, nodes);
}
```

Fig. 1. Complete sketch and specification for the linked list reversal problem

The sketch for this problem is shown in Figure 1. The body of reverseEfficient encodes the basic structure of the solution: allocate a new list, and perform a series of conditional pointer assignments inside a while loop. In order to define the space of possible conditionals and assignments, the sketch uses regular expression notation to define sets of expressions in lines 1 to 3. The sketch, in short, encodes everything that can be easily said about the implementation, and constrains the search space enough to make synthesis tractable.

Together with the sketch, the programmer must provide a specification that describes the correct behavior of the reversal routine. The SKETCH synthesizer allows the user to provide specifications in the form of parameterized or nondeterministic test harnesses. Figure 1 shows the test harness for the list reversal; the synthesizer will guarantee that the harness succeeds for all values of n¡N. On a laptop, the complete synthesis process takes less than a minute for N=4.

```
struct barrier{
      int nthreads; bit sense; bit[N] localSenses;
}
void main(){
   bit[N][T] grid = 0;
   barrier b = newBarrier(N);
   fork(int thread; N){
     for(int i=0; i<T; ++i){
       grid[i][thread] = 1;
       next(b, thread);
       assert grid[i][left(thread)] && grid[i][right(thread)];
}   }  }
```

Fig. 2. Specification for a sense reversing barrier

The sketching approach to synthesis applies just as effectively to concurrent programs, where the synthesizer's ability to orchestrate low-level details is even more desirable given the difficulty that programmers have in reasoning about

the effects of different thread interleavings. To illustrate the use of sketching in this domain, consider the problem of implementing a barrier. A barrier is a synchronization mechanism that forces a group of threads to stop at a particular point in the program until all threads in the group have reached this point. Figure 2 shows the specification for a barrier. The specification is given as a simple test method where N threads repeatedly call the next method of the barrier. The test uses a two dimensional array to check that no thread races past the barrier before its left and right neighbors have reached the next method. The goal of the synthesizer is to find an implementation of the next method that doesn't allow any thread interleavings under which the assertion might fail.

The sketch for the barrier is based on the following high-level insights from [1]. First, the barrier needs to be able to count how many threads have called next; in the sketch, this is done with a READ_AND_DEC instruction which reads the value of b.count into the local variable tmpCount and then decrements it. It's tempting to believe that threads can simply wait for the count to reach zero before continuing, but that doesn't work when the barrier is called from inside a loop; such a scheme would make it possible for a thread to go one full iteration around the loop before the counter has been reset, thus allowing the thread to go through the barrier a second time. This problem is resolved by the second insight: rather than waiting on the count, the barrier should keep a global sense bit; the barrier is released by reversing the sense bit; threads that go a full iteration around the loop will find the barrier with the inverted sense and will wait until the sense is reverted again.

```
1:  #define VALUES {| lsense | b.nthreads | tmpCount | ??|}
2:  #define BITVALUES {| (!)?(b.sense | lsense) | ?? |}
3:  #define COND  {| ?? | (VALUES ( == | != ) VALUES) |}

  static void next(barrier b, int thread){
4:     bit lsense = b.localSenses[thread];
5:     int tmpCount;


6:     READ_AND_DEC( b.count, tmpCount );

7:     reorder{
8:        if(COND ){ b.count = b.nthreads; }
9:        if( COND ){ b.sense = BITVALUES; }
10:       if( COND ){WAIT( {| b.sense (== | !=) lsense |}  )}
11:       if( COND ){ lsense = BITVALUES; }
       }
13:    b.localSenses[thread] = lsense;
  }
```

Fig. 3. Complete sketch and for a sense reversing barrier

In order to express these insights in a sketch, the programmer starts by defining all the different operations that the thread might need to perform after reading and decrementing the counter. The next method must at some point do all of the following: a) reset the count, b) update the global sense, c) wait

on the global sense, d) update the local sense. The programmer knows that the barrier will have to perform all of these operations, but it's not clear in what order or under what conditions. The sketch in Figure 3 expresses both the insight described so far, as well as the programmer's ignorance. All the actions listed before are listed in the sketch inside a `reorder` block. The `reorder` block reflects the programmer's ignorance about the correct order for these statements, giving the synthesizer the freedom to reorder them as necessary. Each statement in the `reorder` is guarded by an `if` statement, giving the synthesizer the freedom to select under what conditions each statement should execute. On a Core Duo L9400 1.86GHz laptop, it takes the synthesizer less than three minutes to solve the sketch in Figure 3 and model check the solution to prove it correct for the bounded case where N=3 and T=3.

2 The Core Sketch Language

The SKETCH language is divided into two parts: the core SKETCH language [5], and a set of high-level constructs implemented as syntactic sugar on top of the core. The core SKETCH language is a simple imperative language extended with a single construct: a constant integer/boolean hole denoted by the symbol `??`.

From the point of view of the core SKETCH language, the role of the synthesizer is to replace each integer hole with a suitable constant so that the resulting program will be correct according to the given correctness criteria. For example, consider the simple program in Figure 4. On the left of the figure, you can see the original sketch; the specification is simply the assertion in the code, and the correctness condition is that the assertion should be satisfied for all inputs within the bound specified by the synthesizer. On the right you can see the resulting code after the synthesizer has replaced the integer hole with a suitable constant. The synthesis process is simply a search for suitable integer constants.

```
int bar(int x){               | int bar(int x){
        int t = x*??;         |         int t = x*2;
        assert t == x+x;      |         assert t == x+x;
        return t;             |         return t;
}                             | }
```

Fig. 4. Simple illustration of the integer hole.

The single integer hole is more powerful than it appears at first sight; by composing integer holes with other language constructs, it is possible to represent many interesting families of expressions. For example, in codes that manipulate arrays, it is common for array indexes to be affine functions of the loop induction variables. If an array access sits inside a loop-nest containing the induction variables i and j, we can represent the set of possible affine functions of i and j using integer holes as i*?? + j*?? + ??. It is also possible to combine

integer holes with conditionals to express complex algorithmic choices. For example, consider the problem of swapping two bit-vectors x and y without using a temporary register. The insight is that the numbers can be swapped by assigning x xor y to x and y repeatedly in a clevery way. The challenge is to find the correct sequence of assignments. The insight, therefore, involves no integer constants, but the integer hole can still be used to encode it as illustrated by Figure 5. After replacing the integer holes with concrete values, the synthesizer performs a cleanup-pass to eliminate any control flow that may have been introduced solely for the purpose of giving choices to the synthesizer, so the resulting code looks like the program on the right of figure Figure 5.

```
//     sketch
int swap(ref int x, ref int y){
        if(??){ x = x ^ y; } else{ y = x ^ y; }
        if(??){ x = x ^ y; } else{ y = x ^ y; }
        if(??){ x = x ^ y; } else{ y = x ^ y; }
}
```

```
//     solution
int swap(ref int x, ref int y){
        y = x ^ y;
        x = x ^ y;
        y = x ^ y;
}
```

Fig. 5. Sketching a register-free swap using the integer hole.

3 Syntactic Extensions

The core SKETCH language is like the assembly language of synthesis; it is expressive enough to describe arbitrarily complex sets of choices, but it is too low level for practical programming. The SKETCH language addresses this problem by defining a number of syntactic extensions to make it easier to write sketches such as those in Figures 1 or 3. The most important of these constructs are: a) the regular expression generators, b) the `repeat` statement, and c) the `reorder` block.

Regular expression generators These constructs (hereafter Re-generators) allow the programmer to use a regular grammar to define families of expressions from which the synthesizer can chose the correct completion for a hole. RE-generators were used extensively in the sketches shown in Figures 1 or 3.

The Re-generator construct has the form {|e|}, where e is a regular expression. The regular expression can include choice $e_1|e_2$ as well as optional expressions e?. We purposely excluded Kleene closure because it increased the search space significantly without a clear programmability benefit.

The current version of the synthesizer implements Re-generators in a fairly straightforward manner. For Re-generators that are used as r-values, the synthesizer will enumerate all the expressions that can be generated from the regular expression e and use an integer hole to select which expression to use. So for example, the VALUES generator in Figure 3 is expanded into a conditional statement like the one shown below in Figure 6.

```
// {| lsense | b.nthreads | tmpCount | ??|}
// is replaced by rv, where rv is defined
// by the following block of code.
int t1 = ??;
assert t1<4;
int rv;
if(t1==0) rv = lsense;
else if(t1==1) rv = b.nthreads;
else if(t1==2) rv = tmpCount;
else rv = ??;
```

Fig. 6. Expanding a Re-generator

For generators that appear in an l-value, the idea is essentially the same; we enumerate the expressions defined by the generator and we use an integer hole to decide which one to assign to. The strategy of fully expanding the set is inefficient, and in some sketches leads to excessive growth in the resulting intermediate representation; surprisingly, though, this simple strategy is sufficient to efficiently synthesize complex sketches such as the ones in Figures 1 and 3.

Repeat statement The `repeat` statement was used in the sketch in Figure 1 to express the programmer's ignorance about how many assignments should be inside the body of the loop. In general, the statement `repeat`(n) c is equivalent to writing n different copies of the statement c. The key feature of the `repeat`, and what distinguishes it from a regular `for` loop, is that if the body c contains any holes, each copy of the body can be resolved to a different statement. For example, in the case of the `repeat` in Figure 1, the `repeat` will be expanded to n potentially different assignment statements.

The implementation of the `repeat` block is also relatively simple; the repeat statement is unrolled and the body replicated in a pre-processing phase. If the repeat count n is a hole, the repeat block is unrolled into a set of nested `if` statements, with the unroll factor determined by a command line flag.

Reorder block The `reorder` block gives the synthesizer the freedom to decide the correct ordering for a set of statements. The most compelling use of `reorder` is to provide the synthesizer with a "soup" of statements that must be assembled into a correct implementation. It is particularly useful in concurrent sketches where the order of seemingly independent statements can be crucial to the correctness of an algorithm.

In order to implement the `reorder` block, we use a representation that is exponential in the number of statements, but is still surprisingly efficient. The basic idea is as follows. Suppose that we start with a list of m statements $s_0; \ldots ; s_{m-1}$, and we want to insert a statement s_m somewhere in the list. We can encode this easily in 2*m+1 statements as shown in Figure 7.

The sketch synthesizer uses this construction to recursively build a representation of the `reorder`. To do this, the synthesizer starts with statement s_0 and uses the construction above to add s_1 before or after it. Then, it repeats the

```
i=??;
if(i=0){ s_m;} s_0;
if(i=1) {s_m;} s_1;
...
if(i=m-1){ s_m;} s_{m-1};
if(i=m){ s_m; }
```

Fig. 7. Expanding a `reorder`

process to insert s_2 into the resulting sequence; the same process is repeated to insert each subsequent statement. The resulting representation has 2^i copies of s_i, and requires on the order of n^2 control bits.

The reason why this is efficient is that most reorder blocks in sketches have at most a handful of expensive statements. The SKETCH implementation initially sorts the statements so that s_0 is the most expensive and s_n is the cheapest one, so the encoding ends up with only one copy of the most expensive statement and many copies of all the smaller ones.

Together, these high-level constructs allow programmers to express their insight about the structure of an implementation and the building blocks that should be used to construct it. The constructs allow programmers to reason in terms of "soups" of statements, and sets of expressions, instead of having to manually translate their insights into the core SKETCH language. At the same time, by compiling these constructs into the core language, the synthesizer gets the benefit of a simple uniform representation that is very well adapted to the available decision procedures.

4 Solving Sketches with Counterexample Guided Inductive Synthesis

Once the high-level constructs have been compiled down to the core SKETCH language, the synthesizer must find the correct values for all the holes in the sketch. Specifically, it must find values that will avoid assertion failures under all possible inputs, or in the case of concurrent sketches, under all possible thread interleavings.

In order to solve this problem, the SKETCH synthesizer relies on Counterexample Guided Inductive Synthesis (CEGIS). The key insight behind CEGIS is that it is often possible to find a small set of carefully selected inputs such that any implementation that works correctly for those inputs will work correctly in general.

The core of the algorithm is a constraint-based inductive synthesis procedure. From a sketch, the procedure builds a set of constraints $Q(x, c)$ such that Q will be true if and only if the sketch works correctly on input x when assigning value c to the holes (in general both x and c are vectors of values). The procedure also takes in a set $E = x_0, \ldots, x_i$ of concrete inputs (or parameters to the test harness) and produces a solution that is guaranteed to work correctly for all the

inputs in the set. The inductive synthesis problem is much more tractable than the general synthesis problem, because the synthesizer doesn't have to reason about the behavior of the program under all possible inputs; only under the inputs provided.

In order for inductive synthesis to produce a correct answer, the system needs to a) have a mechanism to produce good inputs to drive the inductive synthesizer, and b) be able to decide when a correct solution has been discovered. CEGIS achieves these two goals by combining the inductive synthesizer with an automated validation procedure; a model checker in the case of SKETCH. The set of test inputs is seeded with a single random input, and the inductive synthesizer is asked to produce a candidate solution. The candidate is passed to the validator to decide whether the candidate is correct. If it is, then the algorithm has converged, and the program is returned. Otherwise, the validator produces a witness; a counterexample input that shows why the candidate is incorrect. This witness is a perfect input to the inductive synthesizer, because it is guaranteed to cover an aspect of the implementation that none of the previous inputs were covering. Thus, by adding this input to the set of observations of the inductive synthesizer, the synthesis process moves forward, and a new solution is produced that is closer to the desired solution. The complete algorithm is illustrated in Figure 8.

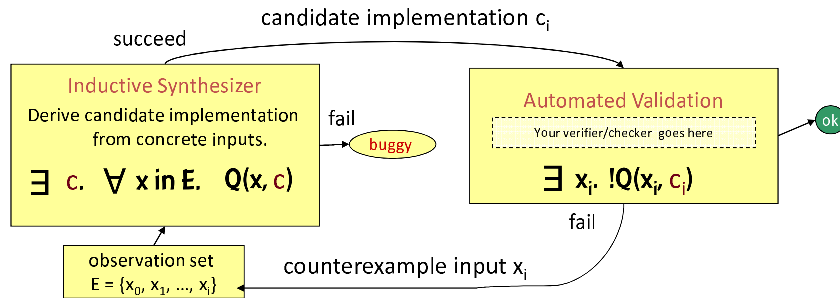

Fig. 8. Counterexample Guided Inductive Synthesis

In the case of concurrent sketches, the process is very similar; the key difference is that the validation procedure produces a trace rather than an input. This poses a challenge to the inductive synthesizer, which must use the counterexample trace to rule out not just the candidate that produced the trace, but any candidates that share the same problem. The details of how this is done can be found in [2] and [4], but the key idea is to extract from the counterexample trace precise information about the ordering of events that lead to the error. This information about the ordering of events is used to ensure that any candidate generated either avoids failure when those events are ordered in the same

way, or rules out that ordering through the use of locking. Overall, the CEGIS procedure is remarkably effective at quickly producing enough good test inputs to get the inductive synthesizer to produce a correct solution.

5 Experience

A thorough evaluation of the SKETCH synthesis engine for both sequential and concurrent programs can be found in [2]. Overall, we have been able to successfully synthesize the low-level details of algorithms in a variety of domains including ciphers, scientific computation, linked data-structures, as well as concurrent objects and data-structures. Despite the success of the tool, a number of important challenges remain in order to make program directed synthesis a standard tool in every programmer's toolkit.

Improving programmability. While the sketch language provides a handful of high-level constructs to help programmers express their insight without having to reason about the low-level details, the language is still too low-level for many domains. For example, for the body of the loop in the sketch from Figure 1, the programmer had to go through the very mechanical process of describing the set of memory locations that could be reached from the lists l and nl. We have found this process to be error prone, as it is easy for programmers to forget choices which turn out to be necessary to construct the solution; for example, many programmers might forget to include `null` in the set LOC. Moreover, when programmers make mistakes and their sketches cannot be solved, it can be difficult for them to find the problems, since debugging a partial program can be more difficult than debugging a concrete one.

A solution to these challenges will have to involve multiple facets, including higher level mechanisms for expressing insights so programmers make fewer errors, language constructs that allow for more interactive exploration of the space of solutions, and diagnostic mechanisms that can pinpoint errors in a sketch.

Exploiting Higher Level Insight Another big challenge is improving the performance of synthesis by harnessing high-level insights, either about a specific program or about an entire domain. In the case of individual programs, we want to exploit high-level invariants that the programmer might know, in order to reduce the search space and make the synthesis more tractable.

In our PLDI 07 paper [3], we showed how synthesis could be made much more effective for programs in a particular domain by incorporating domain specific insight into the synthesizer. We believe such domain-specific insight can make an enormous difference. This will be particularly true in the case of parallelism. For parallel programs, reasoning about concurrency, and about the effect of all possible interleavings is extremely expensive. However, large classes of parallel programs are written in a very disciplined manner that prevents threads from non-deterministically modifying shared memory. Exploiting this discipline should allow for dramatic performance improvements in the synthesis of many concurrent programs.

Moving beyond semantic equivalence and safety In many situations, programmers care about many other factors that go beyond functional correctness. Performance, for example, is a central consideration in many domains. Another closely related property involves statistical properties of an implementation. For example, a hash table will be correct regardless of the implementation of the hash function, but we would like the synthesizer to find an implementation that leads to a good distribution of keys. In some cases, some implementations may be preferred on purely aesthetic grounds; they are easier to read, or contain simpler control flow. The challenge is to develop synthesis strategies that can optimize on these non-functional criteria while still remaining tractable.

6 Conclusions

SKETCH is part of a new breed of programmer guided synthesis tools that aim to make synthesis practical by exploiting the tremendous advances in program analysis and decision procedures over the last ten years, and combining them with the programmer's high-level insights. Another research effort in this area is the Paraglide project [7], which applies powerful domain specific algorithms to achieve programmer guided synthesis of concurrent data structures. Another notable effort in this direction is the work on proof theoretic synthesis by Srivastava et. al. which aims to use techniques from program verification to synthesize complex algorithms from sketch-like skeletons [6].

All of these tools attempt to create a synergy between the strengths of the human programmer and the power of modern analysis tools, with the ultimate goal of making it easier to design and program systems that are more reliable and efficient.

References

1. M. Herlihy and N. Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.
2. A. Solar-Lezama. *Program Synthesis By Sketching.* PhD thesis, EECS Dept., UC Berkeley, 2008.
3. A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, volume 42, pages 167–178, New York, NY, USA, 2007. ACM.
4. A. Solar-Lezama, C. Jones, G. Arnold, and R. Bodík. Sketching concurrent datastructures. In *PLDI 08*, 2008.
5. A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *ASPLOS '06*, San Jose, CA, USA, 2006. ACM Press.
6. S. Srivastava, S. Gulwani, and F. Jeffrey. From program verification to program synthesis. *Submitted to POPL*, 2010.
7. M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. *SIGPLAN Not.*, 43(6):125–135, 2008.