Int J Softw Tools Technol Transfer (2013) 15:475–495
DOI 10.1007/s10009-012-0249-7

SYNTHESIS

Program sketching

Armando Solar-Lezama

Published online: 2 August 2012
© Springer-Verlag 2012

Abstract Sketching is a synthesis methodology 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 a partial program, a sketch, to describe the desired implementation strategy, and leaves the low-level details of the implementation to an automated synthesis procedure. In order to generate an implementation from the programmer provided sketch, the synthesizer uses counterexample-guided inductive synthesis (CEGIS). Inductive synthesis refers to the process of generating candidate implementations from concrete examples of correct or incorrect behavior. CEGIS combines a SAT-based inductive synthesizer with an automated validation procedure, a bounded model-checker, that checks whether the candidate implementation produced by inductive synthesis is indeed correct and to produce new counterexamples. The result is a synthesis procedure that is able to handle complex problems from a variety of domains including ciphers, scientific programs, and even concurrent data-structures.

Keywords SAT/SMT applications · Constraint-based synthesis · Sketching · Synthesis

1 Introduction

Sketching is a synthesis methodology that is designed to help programmers with small but very complex routines, the kind of routines one finds in low-level system's code or high-performance computational kernels. What distinguishes sketching from some of the other forms of synthesis presented in this issue is that all the information that flows from the programmer to the synthesizer is expressed as code. This has important implications for usability, because it means that programmers do not need to master additional formalisms in order to use the synthesizer, giving it the feel of a programming assistant as opposed to that of a formal verification tool.

The starting point of sketching is the *sketch* itself—a partial program where difficult expressions and statements are left unspecified. In their place, the programmer uses *generators* to describe a space of possible code fragments which the synthesizer can use to complete the missing code. The hypothesis behind sketching is that programmers often have an idea about the general form of a solution; a high-level strategy that will solve the problem at hand. To turn the strategy into a program, however, they have to orchestrate many low-level details; a process that is difficult and error prone. It therefore makes sense to focus the synthesizer on those low-level details, leaving control of the high-level strategy in the hands of the programmer. For many domains, partial programs offer a natural way to achieve this division of labor. The programmer controls the implementation strategy by defining the space of solutions the synthesizer can consider, while the synthesizer is responsible for discovering the low-level details of individual expressions in the program. Partial programs allow programmers to interact with the synthesizer without having to resort to separate formalisms, allowing synthesis to be embedded directly into a standard programming language.

In addition to providing the sketch, the user needs to define the expected behavior of the program. In keeping with the philosophy of SKETCH, the programmer defines this behavior through code, either in the form of a reference implementation, or as a set of parameterized unit tests that the synthesized code must pass. The synthesizer must ensure that the synthesized code passes its unit tests or matches the reference implementation for all possible inputs, although in practice it

A. Solar-Lezama (✉)
Massachussets Institute of Technology, Cambridge, USA
e-mail: asolar@csail.mit.edu

can only guarantee that the specification will be satisfied for all inputs up to a given bound. Writing unit tests and reference implementations are both regarded as best practices in traditional software development, but with SKETCH programmers can leverage this effort to get parts of their code synthesized.

The basic strategy for resolving sketches is based on search. The goal is to find the contents of all the unspecified code in the sketch such that the resulting program will behave correctly under all inputs. The space of possible code fragments defined by the generators in the program is bounded, as is the space of inputs that the synthesizer considers, so the synthesis problem is decidable. However, a naïve search is bound to fail given the astronomical sizes of both spaces. For example, even simple sketches can have input spaces in the range of 2^{32}, and even going up to 2^{128} elements. Similarly, the solution spaces for some sketches easily reach beyond 2^{300}. These scales can only be tackled by means of symbolic search mechanisms inspired by those originally developed for model checking in the early 1990s [10].

The sketch synthesizer uses *counterexample-guided inductive synthesis* (CEGIS). The CEGIS algorithm relies on an important empirical hypothesis; for most sketches, only a small set of inputs is needed to fully constrain the solution. In other words, it is possible to find a small set of inputs covering all the corner cases in the sketch, such that only a valid solution to the sketch can work correctly for all these inputs. CEGIS uses an efficient SAT-based inductive synthesis procedure to produce candidate solutions from small sets of inputs. The crucial observation behind the CEGIS algorithms is that the set of corner cases can be discovered automatically by coupling the inductive synthesizer with a validation procedure. Initially, the set of inputs contains only a random input, but once the inductive synthesizer produces its first candidate solution, the solution is checked by the validation procedure. If the candidate is incorrect, the counterexample produced by the validation procedure is fed to the inductive synthesizer, so the next candidate it produces will be guaranteed to work correctly for this corner case. After only a few iterations, the inductive synthesizer will have gathered a representative set of counterexample inputs and will produce a valid candidate which the validation procedure will accept and deliver to the user.

It is possible to construct sketches that violate the empirical hypothesis, and where the algorithm will have to explore every possible input in the worst case. In practice, however, the counterexample-based approach works remarkably well even for sketches with very large candidate and input spaces, converging to a solution after only a handful of iterations—and therefore a handful of calls to the validation procedure. For example, in one sketch for the AES encryption cipher, shown in Sect. 9, the synthesizer was able to derive the contents of over 1,024 32-bit integer constants after analyzing only 600 candidates. A second important property of CEGIS is that it separates the synthesis and verification tasks, making it possible to use off-the-shelf validation procedures, including incomplete but highly scalable procedures such as automated test generation [14].

The rest of the paper provides an overview of the sketch language and a detailed description of the algorithms that make it possible, and concludes with a discussion of our experience with the system and some of its limitations.

2 The sketch language

Sketching extends a simple procedural language with the ability to leave *holes* in place of code fragments that are to be derived by the synthesizer. Each hole is marked by a generator which defines the set of code fragments that can be used to fill a hole. SKETCH offers a rich set of constructs to define generators, but all of these constructs can be described as syntactic sugar over a simple core language that contains only one kind of generator: an unknown integer constant denoted by the token ??.

From the point of view of the programmer, the integer generator is a placeholder that the synthesizer must replace with a suitable integer constant. The synthesizer ensures that the resulting code will avoid any assertion failures under any input in the input space under consideration. For example, the following code snippet can be regarded as the "Hello World" of sketching.

```
harness void main(int x){
  int y = x * ??;
  assert y == x + x;
}
```

This program illustrates the basic structure of a sketch. It contains three elements you are likely to find in every sketch: (a) a harness procedure, (b) holes marked by generators, and (c) assertions.

The harness procedure is the entry point of the sketch, and together with the assertion it serves as an operational specification for the desired program. The goal of the synthesizer is to derive an integer constant C such that when ?? is replaced by C, the resulting program will satisfy the assertion for all inputs in the input space of the harness.[1] For the sketch above, the synthesized code will look like this.

```
void main(int x){
  int y = x * 2;
  assert y == x + x;
}
```

[1] The synthesizer relies on bounded decision procedures, so for harnesses with integer inputs, the input space is the set of all integers within a given bound determined by a command line flag.

The semantics of the sketch can also be framed in game theoretic terms. The inputs to the test harness correspond to non-deterministic choices available to a demonic adversary who is trying to get the program to fail. The ?? operator corresponds to a non-deterministic choice available to an angelic player who is trying to keep the program from failing. The job of the synthesizer is to find a memoryless strategy that guarantees that the angelic player always wins the game. Our restriction on the strategy is stronger than just being memoryless; the semantics of the sketch require that the value produced by the strategy depend only on the current program location (as opposed to the current state, which is what a memoryless strategy guarantees). This strong requirement means that the synthesizer can just replace the non-deterministic choice with a constant value, resulting in a fully deterministic program.

2.1 Sketching with integer generators

When combined with other language constructs, integer generators are remarkably expressive; they can even be used to define arbitrary context-free grammars of program fragments. Even by themselves, however, integer generators can be useful in turning a high-level insight into an efficient implementation.

As a small but realistic example of this, consider the problem of isolating the least significant 0-bit in a word x. For example, for the word 01010011, the desired output is a word containing a 1 in the position of the least significant 0; i.e. 00000100. There is a trick to do this using only three instructions. You may remember it: the trick takes advantage of the fact that adding a 1 to a string of ones preceded by a zero turns all the ones into zeros and turns the next zero into a one (i.e. 000111 + 1 = 001000). You may not remember the details, but with sketching you do not have to; you can let the synthesizer discover them. All you need to remember is the general form of the solution to encode the problem as a sketch. Specifically, you need to remember that the solution involved the addition of a constant to x, a negation, and a bitwise and. The expression ∼(x + ??) & (x + ??) encodes most of the expressions matching this criteria, and when given a suitable specification, the synthesizer can easily find the correct expression.

```
int W = 32;
bit[W] least_sig0(bit[W] x){
  return ∼(x + ??) & (x + ??);
}

bit[W] simple_least_sig0(bit[W] x){
  bit[W] ret = 0;
  for (int i = 0; i < W; i++)
    if (!x[i]) { ret[i] = 1; return ret; }
  return ret;
}
harness void main(bit[W] x){
    assert least_sig0(x) == simple_least_sig0(x);
}
```

In less than a second, the synthesizer is able to discover that the correct expression is ∼(x+0) & (x + 1). If you think this problem was too easy given the initial hint, consider this question: can the same trick be used to find the least significant 1? Without thinking too hard about the problem, one can ask the sketch synthesizer:

```
int W = 32;
bit[W] least_sig1(bit[W] x){
    // same sketch as before
  return ∼(x + ??) & (x + ??);
}

bit[W] simple_least_sig1(bit[W] x){
  bit[W] ret = 0;
  for (int i = 0; i < W; i++)
    if (x[i]) { ret[i] = 1; return ret; }
  return ret;
}
harness void main(bit[W] x){
    assert least_sig1(x) == simple_least_sig1(x);
}
```

It again takes less than a second for the synthesizer to tell us that yes, the same basic trick applies, but now the expression is ∼(x + 0xFFFFFFFF) & (x+0).

Notice that both examples used as a specification a simple operational description for the task at hand, what one would normally refer to as a reference implementation. All the harness is doing is running the sketch and the reference implementation side-by-side and comparing the result. Since this is a common idiom, the sketch language provides syntactic sugar for it. Instead of writing a harness, the user simply states that least_sig1 *implements* the functionality of sim−ple_least_sig1.

```
bit[W] least_sig1(bit[W] x) implements simple_least_sig1{
  return ∼(x + ??) & (x + ??);
}
```

In the above example, it was relatively clear that the tricky details in the implementation involved discovering a few constants. In many cases, however, the details in question do not involve any missing constants. 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 clever way. The challenge is to find the right sequence of

assignments. The insight, therefore, involves no integer constants, but the integer generator can still be used to encode it:

```
int W = 32;

void swap(ref bit[W] x, ref bit[W] 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; }
}
harness void main(bit[W] x, bit[W] y){
   bit[W] xold = x, yold = y;
   swap(x,y);
   assert y == xold && x == yold;
}
```

The sketch above uses the integer generator to encode the choice between assigning x ^ y to x or to y. The synthesizer uses the usual convention of using 0 to represent false and 1 to represent true, and in less than a second, it is able to discover that the three holes should evaluate to false, true and false, respectively. After replacing the generators with constants, the synthesizer will perform a small amount of cleanup, eliminating the unnecessary conditionals to produce the code shown below.

```
void swap(ref bit[W] x, ref bit[W] y){
   y = x ^ y;
   x = x ^ y;
   y = x ^ y;
}
```

2.2 Higher level generators

The language provides some syntactic sugar that makes sketches like the one above significantly easier to write. Two of the most important constructs are repeat blocks and regular expression generators.

The construct repeat(N) c allows programmers to define a repeating algorithmic pattern. If N is a constant, the synthesizer will create N copies of the statement c and solve for the unknowns in each copy of c independently. Using this construct, the swap sketch can be expressed concisely as

```
void swap(ref bit[W] x, ref bit[W] y){
   repeat(3)
      if(??){ x = x ^ y; }else{ y = x ^ y; }
}
```

The repeat construct also allows the number of copies to be left unspecified by writing repeat(??), and if a minimal number of steps is desired, one can use min−re−peat instead. As was said before, all generators represent finite spaces of possible code fragments, and repeat is no exception; even when using repeat(??), the synthesizer will only consider up to a bounded number of repetitions, where the bound is determined by a command line flag.

Regular expression generators (RE-generators) are another constructs that allows the choice of expressions to be expressed more concisely without resorting to messy if statements. The RE-generator construct has the form {|e|}, where e is a regular expression literal. The semantics of the construct are that the synthesizer substitutes the syntactic occurrence of the construct with a string in the language $L(e)$ such that the substitution resolves the sketch. Using RE-generators, the example can be written even more concisely as:

```
void swap(ref bit[W] x, ref bit[W] y){
   minrepeat
      {| x | y |} = x ^ y;
}
```

In designing the language we made a design decision to support only two regular expression operators: choices $e_1 | e_2$ and optional expressions $e?$. Kleene closure is excluded for two reasons: first, we never found a situation where we needed to have an expression exhibiting the kind of unbounded repetition that Kleene closures imply. Moreover, since generators can only represent bounded sets of expressions, the Kleene closure would still have to be bounded in an arbitrary way. If the user needs to use Kleene closure, it can always be expressed using the repeat construct, so adding Kleene closure would have increased the number of constructs involving artificial bounds, and it would not have produced any significant programmability benefits.

2.3 Abstraction in SKETCH

Procedures allow the programmer to hide the details of a computation behind a simple interface, and are one of the most commonly used forms of abstraction in production languages. SKETCH supports procedures exactly as one would expect: generators within them are syntactically replaced with code fragments that ensure the correctness of the generated program.

```
int linexp(int x, int y){
   return ??*x + ??*y + ??;
}
harness void main(int x, int y){
   assert linexp(x,y) >= 2*x + y;
   assert linexp(x,y) <= 2*x + y+2;
}
```

For example, for the routines above, there are many different solutions for the holes in lin−exp that will satisfy the

first assertion, and there are many that will satisfy the second assertion, but the synthesizer will chose one of the candidates that satisfies them both.

```
int linexp(int x, int y){
  return 2∗x + y;
}
```

The procedure lin−exp originally had holes, and therefore corresponded to a set of functions. However, the synthesizer completed the holes to give the procedure a *single* concrete meaning to be used across all calling sites. This gives procedures the same power of abstraction that they would have in the absence of sketching. But, as the following example illustrates, sketching creates the need for a mechanism to abstract *sets* of functions.

```
int[N∗N] transpose(int N, int[N∗N] mat){
  int[N∗N] out;
  for(int i=0; i<N; ++i) for(int j=0; j<N; ++j){
    out[ {|??|N|}∗i + {|??|N|}∗j+{|??|N|}] =
    mat[{|??|N|}∗i + {|??|N|}∗j+{|??|N|}];
  }
  return out;
}

harness void main(int N, int[N∗N] mat, int i, int j){
  int[N∗N] out = transpose(N, mat);
  assert !(i < N && j < N) || mat[i∗N + j] == out[j∗N + i];
}
```

In the above sketch, trans−pose is a procedure which abstracts the matrix transpose function for $N \times N$ matrices. However, within the transpose procedure, the expression |??|N|*i + |??|N|*j+|??|N|! is repeated twice. This expression is quite big, so one would like to abstract it into its own procedure to avoid the repetition. However, one cannot abstract this expression into a procedure because each use of the expression has to resolve to a different linear expression. Therefore, an abstraction mechanism is needed to represent the entire set of functions encoded by |??|N|*i + |??|N|*j+|??|N|!, rather than a single one like the procedure does.

The SKETCH language allows programmers to abstract sets of functions into custom generators. For each use of the generator, the synthesizer is free to choose a different function. For the above example, the expression |??|N|*i + |??|N|*j+|??|N|! can be abstracted into a generator that represents the set of linear expressions involving i and j with either constants or N as coefficients.

```
generator int legen(int i, int j, int N){
  return {|??|N|}∗i + {|??|N|}∗j+{|??|N|};
}

int[N∗N] transpose(int N, int[N∗N] mat){
  int[N∗N] out;
  for(int i=0; i<N; ++i) for(int j=0; j<N; ++j){
    out[ legen(i,j,N) ] = mat[ legen(i,j,N) ];
  }
  return out;
}
```

Each call to the generator resolves to a different expression, resulting in a correct implementation for the $N \times N$ transpose.

```
int[N∗N] transpose(int N, int[N∗N] mat){
  int[N∗N] out;
  for(int i=0; i<N; ++i) for(int j=0; j<N; ++j){
    out[ N∗i + j ] = mat[ i + N∗j ];
  }
  return out;
}
```

Programmers are encouraged to think of generators as procedures which are inlined into their calling context before the sketch is synthesized, so each call to the generator will be resolved independently from other calls.

Generators derive much of their expressive power from their ability to recursively define a space of expressions. For example, consider again the least_sig1 example; as presented earlier, the example assumed considerable knowledge about the shape of the solution. If the user lacks that knowledge, he can rely on a very general generator to leverage the synthesizer more heavily as illustrated by the code below.

```
generator bit[W] gen(bit[W] x, int bnd){
  assert bnd >= 0;
  if(??) return x;
  if(??) return ??;
  if(??) return ∼gen(x, bnd−1);
  if(??)
    return {| gen(x, bnd−1) (+ | & | ^) gen(x, bnd−1) |};
}

bit[W] least_sig1(bit[W] x) implements simple_least_sig1{
  return gen(x, 3);
}
```

The user-defined generator gen recursively defines the space of all expressions involving x, bit-vector constants, and the operators +, &, ^ and the bitwise negation ∼. The parameter bnd controls the depth of recursion, limiting the synthesizer to expressions of a certain size. Without this bound, a solver flag limiting the depth of recursion of generators would determine the maximum size of expressions that the synthesizer is allowed to consider.

2.4 Putting it all together

To illustrate the use of sketching, 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.

The sketch for this problem is shown in Fig. 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 1–3. 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 non-deterministic test harnesses. Figure 1 shows the test harness for the list reversal; the synthesizer will guarantee that the harness succeeds for all values of $n < N$. Specifically, the synthesizer guarantees that none of the inputs within the given bound will trigger any assertions or cause any illegal memory accesses. Additionally, the system also guarantees that the unbounded while loop whose condition is left unspecified will terminate for all $n < N$. On a laptop, the complete synthesis process takes less than a minute for $N = 4$.

3 Semantic of sketches

A sketch can be understood as a set of programs, each corresponding to a different valuation of the holes. This section defines the *synthesis semantics* of sketches in a way that makes it easy to characterize the set of correct solutions to the sketch and the relationship between the values of holes and the state of the program at a given point in the execution.

To illustrate the key ideas behind the formalism, consider the following example.

```
generator int linexp(int t){
    return t*??0 + ??1;
}
generator int linexp2(int t1, int t2){
    return linexp_g1(t1) + linexp_g2 (t2);
}

harness void HelloWorldGen(int x, int z){
    int y = linexp2_g0(x, z);
    assert y == x + x + z;
}
```

The example is a simple variation of the "Hello World" program that uses generators. The holes and the call sites for the generators have been labeled with identifiers to help us to refer to them in the text.

Now, recall that the goal of synthesis is to assign a value to every hole. Moreover, in the case of generators, the synthesizer needs to assign different values to the same hole depending on the calling context. In order to model these value assignments, we use a *control function* $\phi : H \times \mathcal{T} \rightarrow \mathbb{Z}$ that assigns a value to each hole in the program at a given calling context τ. In the example above, there are two holes

```
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:   while(COMP){
7:     repeat(??)
8:       if( COMP ){ LHS = LOC; }
     }
   }
```

```
harness void main(int n){ // test harness
    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

and two different valid calling contexts for these holes: $\tau_1 = g_0 \cdot g_1$ and $\tau_2 = g_0 \cdot g_2$. Therefore, any ϕ for this sketch must define a value for each of the following four pairs of holes and calling contexts: $(??_0, \tau_1)$, $(??_0, \tau_2)$, $(??_1, \tau_1)$, $(??_1, \tau_2)$.

In the synthesis semantics, every value in the program is modeled as a function of the control. We call these functions *parameterized values*, and we use the Greek letter Ψ to designate the set of all such values. For example, in sketch above, y has the parameterized value $\lambda\phi.x * \phi(??_0, \tau_1) + \phi(??_1, \tau_1) + z * \phi(??_0, \tau_2) + \phi(??_1, \tau_2)$. The state of the program is modeled with an environment σ that maps variable names to parameterized values, as summarized in Fig. 2.

The formalism is very different from that used in automata-based synthesis. The main motivation for choosing this formalism in place of a more traditional one is to simplify reasoning about some of the higher level features in the language. For example, in this formalism it is relatively easy to describe the difference between a user-defined generator and a function, or to reason about the semantics of high-order functions, or even high-order functions that take generators as parameters. It also makes it relatively easy to prove the correctness of program transformations, such as the partial evaluation mechanism used to produce code from a sketch [16].

3.1 Basic rules

The synthesis semantics are described formally through a very simple model language described by the abstract syntax

L	The set of variables in the program. x is used to refer to an arbitrary variable.
H	The set of holes in the program. All holes are assumed to be uniquely labeled as $??_i$.
$\mathcal{T}$	The set of calling contexts. A calling context $\tau = g_{i_0} \cdot g_{i_1} \cdot \ldots \cdot g_{i_n}$ is a sequence of generator call sites. $\tau_\emptyset$ is the empty context.
$\boldsymbol{\Phi} = H \times \mathcal{T} \to \mathbb{Z}$	The set of control functions. A control ϕ corresponds to a particular assignment of integers to holes, where each hole is identified by its label $??_i$ together with its calling context τ.
$\mathcal{P}(\boldsymbol{\Phi})$	The powerset of $\boldsymbol{\Phi}$. Φ is used to refer to a given subset of $\boldsymbol{\Phi}$.
$\Psi = \boldsymbol{\Phi} \to \mathbb{Z}$	The set of parameterized values. A parameterized ψ value produces an integer for each control.
$\Sigma = L \to \Psi$	The set of all states. A state σ maps each variable to a parameterized value.

Fig. 2 Notation for the synthesis semantics of SKETCH

expressions	e	$::= n \mid \mathrm{x} \mid \mathrm{x}[e] \mid e \star e \mid ??$
statements	c	$::= \mathrm{x} := e \mid \mathrm{x}[e] := e \mid \mathrm{skip} \mid f(e) \mid c; c \mid$ if e then c else $c \mid$ assert $e \mid$ while e do c
functions	f	$::= \mathrm{def\ f(x)}\ c$
generators	g	$::= \mathrm{defgen\ g(x)}\ c$
programs	p	$::= p\ f \mid p\ g \mid \epsilon$

Fig. 3 Abstract syntax for a simplified subset of the SKETCH language

in Fig. 3. The language has been simplified in a few cosmetic ways to make the presentation simpler. For example, the operator $\star$ is used to denote an arbitrary binary operator. Expressions are assumed to have no side effects; expressions that might lead to an error, such as out of bounds array accesses or division by zero, are assumed to be preceded by an appropriate assertion so the expressions themselves can be modeled as being side effect free. Procedure calls are assumed to be statements rather than expressions; they return values by writing to a special variable @, and they have no other side effects besides writing to this variable and possibly causing assertion failures.

Parameterized values allow us to define the synthesis semantics following many of the formalisms of standard denotational semantics. As in denotational semantics, the meaning of an expression is defined recursively through a *denotation function*.

$$\mathcal{A}[\![\circ]\!]^\tau : Aexp \to (\Sigma \to \Psi). \tag{3.1}$$

The denotation function defines the meaning of any expression as a function from a state to a parameterized value. The state $\sigma : L \to \Psi$ of the program is a mapping from the set of variable names L to parameterized values. The τ in the denotation function indicates the calling context under which the interpretation is taking place.

The denotation function is defined recursively for various types of expressions, quite similar to the way these functions are defined in denotational semantics. The only new rule is the rule for evaluating a hole, which produces a function that takes in a control ϕ, and produces the value of the hole on that control under the current calling context τ.

$$\begin{aligned} \mathcal{A}[\![x]\!]^\tau\sigma &= \sigma(x) \\ \mathcal{A}[\![??_i]\!]^\tau\sigma &= \lambda\phi.\phi(??_i, \tau) \\ \mathcal{A}[\![e_1 \star e_2]\!]^\tau\sigma &= \lambda\phi.\ \mathcal{A}[\![e_1]\!]^\tau\sigma\phi \ \star\ \mathcal{A}[\![e_2]\!]^\tau\sigma\phi. \end{aligned}$$

Unlike expressions, commands have side effects. To model these, the denotation function defines the meaning of a command as a transformation on a state and a set of candidate controls. From the initial state and control set, the command produces an updated state and a subset of the original control set containing only those controls which are valid for that command, i.e. those that do not cause assertion failures. Expressions do not need to track these sets because, as was

said earlier, we have assumed that evaluation of expressions will never lead to errors.

$$\mathcal{C}[\![\circ]\!]^\tau : Command \rightarrow (\langle \Sigma\,,\, \mathcal{P}(\boldsymbol{\Phi})\rangle \rightarrow \langle \Sigma\,,\, \mathcal{P}(\boldsymbol{\Phi})\rangle) \tag{3.2}$$

The two most basic rules are those for assertions and assignments.

$$\mathcal{C}[\![x := e]\!]^\tau \langle \sigma\,,\, \Phi\rangle = \langle \sigma[x \mapsto\ \mathcal{A}[\![e]\!]^\tau \sigma]\,,\, \Phi\rangle$$
$$\mathcal{C}[\![\text{assert } e]\!]^\tau \langle \sigma\,,\, \Phi\rangle = \langle \sigma\,,\, \{\phi \in \Phi :\ \mathcal{A}[\![e]\!]^\tau \sigma\phi = 1\}\rangle.$$

Assignments modify only the state while leaving the set of candidate controls unmodified. Assertions, on the other hand, narrow the set of valid controls, to include only those that will cause the assertion to succeed.

Sequencing of commands is easy to define; it is just a composition of two functions.

$$\mathcal{C}[\![c_1; c_2]\!]^\tau \langle \sigma\,,\, \Phi\rangle =\ \mathcal{C}[\![c_2]\!]^\tau (\ \mathcal{C}[\![c_1]\!]^\tau \langle \sigma\,,\, \Phi\rangle)$$

Example To illustrate how these rules operate, consider the denotation function for the body of the Hello−World example in the previous section.

```
int y = x * ??;
assert y == x + x;
```

For this example, the initial state will just map the input variable x to a symbolic input value x.

$$\mathcal{C}[\![\text{y = x * ??}_0\text{; assert y == x + x;}]\!]^\tau \langle [\text{x} \mapsto x]\,,\, \boldsymbol{\Phi}\rangle$$
$$=\ \mathcal{C}[\![\text{assert y == x + x}]\!]^\tau\ \mathcal{C}[\![\text{y = x * ??}_0]\!]^\tau \langle [\text{x} \mapsto x]\,,\, \boldsymbol{\Phi}\rangle$$

In that equation, $\mathcal{C}[\![\text{y = x * ??}_0]\!]^\tau \langle [\text{x} \mapsto x]\,,\, \boldsymbol{\Phi}\rangle$ evaluates to the following pair.

$$\mathcal{C}[\![\text{y = x * ??}_0]\!]^\tau \langle [\text{x} \mapsto x]\,,\, \boldsymbol{\Phi}\rangle$$
$$= \langle [\text{x} \mapsto x, \text{y} \mapsto (\ \mathcal{A}[\![\text{x * ??}_0]\!]^\tau [\text{x} \mapsto x])]\,,\, \boldsymbol{\Phi}\rangle$$
$$= \langle [\text{x} \mapsto x, \text{y} \mapsto \lambda\phi.x * \phi(??_0, \tau_\emptyset)]\,,\, \boldsymbol{\Phi}\rangle$$

Therefore,

$$\mathcal{C}[\![\text{assert y == x + x}]\!]^\tau\ \mathcal{C}[\![\text{y = x * ??}_0]\!]^\tau \langle [\text{x} \mapsto x]\,,\, \boldsymbol{\Phi}\rangle$$
$$=\ \mathcal{C}[\![\text{assert y == x + x}]\!]^\tau \langle [\text{x} \mapsto x,$$
$$\text{y} \mapsto \lambda\phi.x * \phi(??_0, \tau_\emptyset)]\,,\, \boldsymbol{\Phi}\rangle$$
$$= \langle [\text{x} \mapsto x, \text{y} \mapsto \lambda\phi.x * \phi(??_0, \tau_\emptyset)],$$
$$\{\phi \in \boldsymbol{\Phi} : x * \phi(??_0, \tau_\emptyset) == x + x\}\rangle$$

The resulting pair tells us what we needed to know about the semantics of the Hello−World program. On the one hand, it shows the exact relationship between the state and the choice of value for the hole. On the other hand, it constrains the set of valid control functions to those satisfying the relationship $x * \phi(??_0, \tau_\emptyset) == x + x$.

The rules for control statements are somewhat more complicated due to the handling of Φ. In an if statement, each branch is evaluated under the subset of Φ that would cause the program to take that branch, and the resulting sets of controls are combined through set union. In the rules below we use the notational shortcut $a?b:c$ to represent the function that returns b if a is true and c otherwise.

$$\mathcal{C}[\![\text{if } e \text{ then } c_1 \text{ else } c_2]\!]^\tau \langle \sigma\,,\, \Phi\rangle = \langle \sigma'\,,\, \Phi'\rangle,$$

where σ' and Φ' are defined through the following equations:

$$\Phi_t = \{\phi \in \Phi :\ \mathcal{A}[\![e]\!]^\tau \sigma\phi = true\}$$
$$\Phi_f = \{\phi \in \Phi :\ \mathcal{A}[\![e]\!]^\tau \sigma\phi = false\}$$
$$\langle \sigma_1\,,\, \Phi_1\rangle =\ \mathcal{C}[\![c_1]\!]^\tau \langle \sigma\,,\, \Phi_t\rangle$$
$$\langle \sigma_2\,,\, \Phi_2\rangle =\ \mathcal{C}[\![c_2]\!]^\tau \langle \sigma\,,\, \Phi_f\rangle$$
$$\Phi' = (\Phi_1) \cup (\Phi_2)$$
$$\sigma' = \lambda x.\lambda\phi.\ \mathcal{A}[\![e]\!]^\tau \sigma\phi\ ?\ \sigma_1 x\phi : \sigma_2 x\phi.$$

while loops are handled in a similar way as they are handled in regular denotational semantics, by defining their denotation function recursively.

$$W(\langle \sigma\,,\, \Phi\rangle) =\ \mathcal{C}[\![\text{while } e \text{ do } c]\!]^\tau \langle \sigma\,,\, \Phi\rangle = \langle \sigma'\,,\, \Phi'\rangle$$
$$\Phi_t = \{\phi \in \Phi :\ \mathcal{A}[\![e]\!]^\tau \sigma\phi = true\}$$
$$\Phi_f = \{\phi \in \Phi :\ \mathcal{A}[\![e]\!]^\tau \sigma\phi = false\}$$
$$\langle \sigma_1\,,\, \Phi_1\rangle = W(\ \mathcal{C}[\![c]\!]^\tau \langle \sigma\,,\, \Phi_t\rangle)$$
$$\Phi' = (\Phi_1) \cup (\Phi_f)$$
$$\sigma' = \lambda x.\lambda\phi.\ \mathcal{A}[\![e]\!]^\tau \sigma\phi\ ?\ \sigma_1 x\phi : \sigma x\phi$$

For some loops, it is possible to solve the equation above to derive a closed form expression for W. For example, consider the loop below.

```
wihile i < N do
    assert ??0 > i
    i = i+1
```

For this loop, the closed form solution for $W(\sigma, \Phi)$ is

$$\begin{cases} \sigma(i) < N & \langle \sigma[i \mapsto N]\,,\, \Phi \cap \{\phi : \phi(??_0) > N - 1)\}\rangle \\ else & \langle \sigma\,,\, \Phi\rangle \end{cases}$$

One can check that this function satisfies the recursive constraints for W. Unfortunately, the problem of finding a closed form for the W function of a loop is undecidable in general. In our synthesizer, we will get around this problem by bounding the number of iterations of loops. For states σ that cause the loop to iterate more than the allowed number of times, we define $W(\sigma, \Phi) = (\sigma', \emptyset)$. In practice, this will mean that our synthesizer may fail to find a solution to a sketch when one actually exists, or more commonly, that the user will have to make sure that the bounds in the number of

iterations are enough to handle all the inputs that the synthesizer may consider.

There are still some semantics left to describe, namely the semantics of procedures and generators, and the current limited support for language features such as arrays and heap allocated objects. However, this covers the major ideas behind the synthesis semantics, which will allow us to define the set of valid solutions to a sketch and subsequently to reason formally about our novel counterexample guide inductive synthesis algorithm.

3.2 Procedures and generators

Procedure calls behave as we would expect from standard denotational semantics. For a function defined as def f(x) c, the semantics of a call to f(e) are defined by evaluating the body of the function under the empty calling context $\tau_\emptyset$, and the initial state $\sigma_\perp[x \mapsto \mathcal{A}[\![e]\!]^\tau\sigma]$, where $\sigma_\perp$ is the empty state.

$$\langle\sigma', \Phi'\rangle = \mathcal{C}[\![c]\!]^{\tau_\emptyset}\langle\sigma_\perp[x \mapsto \mathcal{A}[\![e]\!]^\tau\sigma], \Phi\rangle$$
$$\mathcal{C}[\![f(e)]\!]^\tau\langle\sigma, \Phi\rangle = \langle\sigma[@ \mapsto \sigma'(@)], \Phi'\rangle$$

In the definition, the return value of f is stored in the special variable @ as explained before.

The evaluation of generators is only slightly different. Instead of evaluating the body under the empty calling context, the body is evaluated under the calling context $\tau \cdot g_i$, where g_i identifies the current call site for the generator. Therefore, the semantics for a call to a generator defined as defgen g(x) c from a call site g_i are defined by the formulas below.

$$\langle\sigma', \Phi'\rangle = \mathcal{C}[\![c]\!]^{\tau\cdot g_i}\langle\sigma_\perp[x \mapsto \mathcal{A}[\![e]\!]^\tau\sigma], \Phi\rangle$$
$$\mathcal{C}[\![g(e)]\!]^\tau\langle\sigma, \Phi\rangle = \langle\sigma[@ \mapsto \sigma'(@)], \Phi'\rangle$$

An interesting observation is that procedure calls have the effect of forgetting the calling context, so generators called from a procedure will behave the same regardless of the calling context of the procedure.

3.3 Bounded semantics for generators

As we saw in Sect. 2.3, a generator represents a set of functions, and the synthesizer is free to select any of these functions to replace a call to the generator. However, there is a problem with the way we defined the synthesis semantics for generators: they make generators *too* powerful. So powerful, in fact, that they can represent sets which include functions that are not even computable, a clear problem if we expect to synthesize code from them.

The problem is that the semantics defined so far allow programmers to write sketches which can only be resolved with an infinite ϕ. A trivial example of this would be the universal generator:

```
generator int univ(int x){
    if( abs(x) > 0 ){ return univ(abs(x)−1); }
    else { return ??; }
}
```

According to the synthesis semantics, the generator above can be made to represent any function in the set $\mathbb{N} \to \mathbb{Z}$, even though we know some functions in this set are not computable.

To address this problem, we provide a slight modification to the semantics which we call *bounded generator semantics*. Bounded generator semantics bounds the recursion of generators by specifying a bounded set of call stacks $\bar{\tau}$. Thus, for a generator defined as defgen g(x) c, the semantics of a call to g at call site g_i now involve a check of whether $\tau \in \bar{\tau}$.

$$\langle\sigma', \Phi'\rangle = \mathcal{C}[\![c]\!]^{\tau\cdot g_i}\langle\sigma_\perp[x \mapsto \mathcal{A}[\![e]\!]^\tau\sigma], \Phi\rangle$$
$$\mathcal{C}[\![g(e)]\!]^\tau\langle\sigma, \Phi\rangle = \begin{cases}\langle\sigma[@ \mapsto \sigma'(@)], \Phi'\rangle & \text{if } \tau \in \bar{\tau} \\ \langle\sigma, \emptyset\rangle\end{cases}$$

One can see from the formulas that trying to evaluate a generator when the current stack does not belong to $\bar{\tau}$ has the same effect as an assertion failure.

4 The sketch resolution equation

In SKETCH, it is required that all sketches have at least one harness procedure h. The semantics of a program P are thus defined in terms of the effect of calling the harness procedure.

$$\mathcal{C}[\![P]\!]^\tau\langle\sigma, \Phi\rangle = \mathcal{C}[\![\text{h}(in)]\!]^{\tau_\emptyset}\langle\sigma, \Phi\rangle.$$

From this definition, we can define a set of valid controls Φ to be one which satisfies the sketch resolution equation.

Equation 1 (Sketch Resolution) *The set of controls Φ is said to be valid if it is invariant under $\mathcal{C}[\![P]\!]^{\tau_\emptyset}$ for any initial state, as expressed in the equation below.*

$$\forall\,\sigma \quad \mathcal{C}[\![P]\!]^{\tau_\emptyset}\langle\sigma, \Phi\rangle = \langle\sigma', \Phi\rangle.$$

Now, let Φ^* denote the maximal set of valid controls, or maximal solution to the sketch.

The sections that follow will explain how to use the synthesis semantics to symbolically approximate Φ^*.

5 Solving sketches with CEGIS

The synthesis semantics from the previous section allow the search for a valid control to be framed as a constraint satisfaction problem. Specifically, the synthesis semantics define

the meaning of a program P through a denotation function $\mathcal{C}[\![P]\!]^{\tau_\eta}\langle\sigma_{in}, \Phi\rangle \rightarrow \langle\sigma_{out}, \Phi'\rangle$. The function describes how an initial set Φ of candidate solutions is constrained down to a subset Φ' containing only those solutions which are correct for input σ_{in}. Therefore, a valid candidate $\phi \in \Phi^*$ is one that satisfies the constraints imposed by each of the possible inputs.

The synthesis semantics allow us to derive a set of constraints on ϕ in terms of the input state σ. If we use the predicate $Q(\phi, \sigma)$ to represent these constraints, the synthesis problem becomes a doubly quantified constraint system.

$$\exists\phi \ \forall\sigma \ \ Q(\phi, \sigma). \tag{5.1}$$

Solving constraint systems involving such universally quantified variables is difficult, and many existing approaches do not scale to the size and complexity of the sketches we want to solve. Fortunately, sketches are not arbitrary constraint systems; they are partial programs written to convey the high level structure of a solution while leaving the details unspecified. Therefore, a decision procedure that takes advantage of the structure embodied in sketches can succeed where the general solution strategies fail.

5.1 Solving sketches with inductive synthesis

The crucial observation that makes sketch synthesis possible is that for many sketches, an implementation that works correctly for the common case and for all the different corner cases is likely to work correctly for all inputs. For example, consider the sketch of a remove method for a doubly linked list. The sketch may have a large number of possible solutions, and a very large space of inputs; however, in addition to the common case where an element is removed from the middle of the list, there are only a handful of corner cases that can cause problems, such as the cases involving removal of the head, the tail, and removal from a list of size one. Therefore, the synthesis problem can be simplified enormously by focusing only on a handful of inputs that are representative of the common case and of the problematic corner cases. This insight can be made more formal through the following empirical hypothesis.

Hypothesis 1 *(Bounded Observation Hypothesis)* For a given sketch P, it is possible to find a small set of inputs E that fully represents the entire domain of inputs Σ such that any set of controls Φ satisfying

$$\forall\, \sigma \in E \ \ \mathcal{C}[\![P]\!]^{\tau_\eta}\langle\sigma, \Phi\rangle = \langle\sigma', \Phi\rangle \tag{5.2}$$

will also be a solution to the sketch resolution equation of Sect. 4.

The hypothesis implies that we can frame the sketch synthesis problem as an inductive synthesis problem. Inductive synthesis is the process of generating a program from concrete observations of its behavior, where an observation describes the expected behavior of the program on a specific input [3]. The inductive synthesizer uses each new observation to refine its hypothesis about what the correct program should be until it converges to a solution. Inductive synthesis had its origin in the work by Gold [8] on language learning, and the pioneering work by Shapiro [15] on inductive synthesis and its application to algorithmic debugging among others (e.g. [21]).

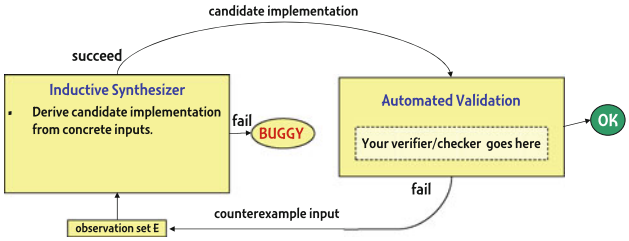


Fig. 4 Counterexample driven synthesis algorithm

Three important problems must be resolved in order to apply inductive synthesis to the problem of sketch resolution. First, it is necessary to have a mechanism to generate observations to drive the inductive synthesis. This mechanism should be able to generate inputs that exercise the corner cases in the implementation so the inductive synthesis quickly converges to a correct candidate. Second, the system needs a mechanism to determine convergence, i.e. to decide when the candidate derived from the set of observations actually generalizes to work correctly for all inputs. And finally, the system needs an inductive synthesis procedure capable of efficiently solving Eq. (5.2) for realistic sketches.

To address the first two problems, we designed a counterexample-guided inductive synthesis algorithm (CEGIS). This algorithm handles convergence checking and observation generation by coupling the inductive synthesizer with a validation procedure as illustrated in Fig. 4. In the algorithm, a validation procedure checks the candidate implementation produced by the inductive synthesizer. If the validation succeeds, the candidate is considered correct, and is returned to the user. If validation fails, then the validation procedure is expected to produce a bounded and concrete input which exhibits the bug in the candidate program. The witness to the bug can then be used as an observation for the inductive synthesizer.

The CEGIS algorithm owes an intellectual debt to the idea of counterexample-guided abstraction refinement (CEGAR) introduced by Clarke et al.CEGARClarke03 to cope with the state explosion problem in model checking. CEGAR exploits the observation that a counterexample is much easier to find in an abstract model, but abstract models can produce spurious counterexamples which are infeasible in the concrete model. This drawback can be alleviated by combining the

abstract model checker with a validation procedure that can check whether a counterexample is indeed feasible for the original model. If it is not, the validation procedure can refine the abstraction to disallow the spurious counterexample, and the cycle can be repeated. If we view the input set E as an abstraction of the original input domain, the CEGIS algorithm can be seen as an application of the CEGAR idea to the problem of program synthesis.

5.2 Formalization of algorithm

The algorithm illustrated in Fig. 4 can be succinctly expressed in terms of the synthesis semantics. In the algorithm below, Φ_i is the set of all controls which satisfy the specification for the input states $E = \{\sigma_0, \ldots, \sigma_{i-1}\}$. The control ϕ_i is a candidate selected non-deterministically from Φ_i, and it constitutes the result of the inductive synthesis, as it is guaranteed to work correctly for all inputs in E. The state σ_i is an input which exposes an error in the candidate program represented by ϕ_i. The initial control set Φ_0 is initialized to $\boldsymbol{\Phi}$, the set of all controls, while σ_0 is initialized to a random initial state.

Each iteration of the CEGIS loop starts with the inductive synthesis phase. In this phase, a new set Φ_i is computed by removing from Φ_{i-1} those controls which cause the specification to be violated for the input σ_{i-1}. Φ_i is represented symbolically as a set of constraints, and it is derived by applying $\mathcal{C}[\![P]\!]^{\tau_\emptyset}$ to σ_{i-1} and to the symbolic representation of Φ_{i-1}. The symbolic representation is then queried for an element $\phi_i \in \Phi_i$ which is the result of the inductive synthesis phase. It is important to note that while the set represented by Φ_i is shrinking after every iteration, the representation is actually growing, since every iteration of the CEGIS loop is adding more constraints.

Algorithm 1 (CEGIS Algorithm) .

$\sigma_0 := \sigma_{random}$
$\Phi_0 = \boldsymbol{\Phi}$
$i := 0$
do
  $i = i + 1$

  $(\sigma', \Phi_i) := \mathcal{C}[\![P]\!]^{\tau_\emptyset}(\sigma_{i-1}, \Phi_{i-1})$
  if $\Phi_i = \emptyset$ then return UNSAT_SKETCH
  def $\phi_i \in \Phi_i$
  } Inductive Synthesis

  def σ_i s.t. $\mathcal{C}[\![P]\!]^{\tau_\emptyset}(\sigma_i, \{\phi_i\}) = (\sigma', \emptyset)$ } Validation

while $\sigma_i \neq$ null
return $PE(P, \phi_i)$

The validation phase of the algorithm checks whether the candidate solution associated with ϕ_i satisfies the specification for all possible inputs. If it does, then the candidate generated from control ϕ_i is the solution that the algorithm was looking for; if it does not, then the process is repeated until either a solution is found or Φ_i becomes empty. In the latter case, we can assert that the sketch has no valid solutions.

The sets Φ_i generated by the CEGIS algorithm form a series that approaches Φ^*, the maximal solution of the sketch equation, in strictly monotonic fashion. This means that if $\boldsymbol{\Phi}$ is bounded, then the procedure above is guaranteed to terminate, and Φ_i will converge towards Φ^*. In fact, because the algorithm is only looking for a single $\phi \in \Phi^*$, it can actually terminate before Φ_i has converged to Φ^* if the ϕ_i selected from Φ_i also happens to be in Φ^*.

5.3 Convergence

The theoretical convergence properties of the algorithm are not great. The number of iterations is bounded by the maximum of the size of the control space and the size of the input space. Even for bounded sketches, these sizes can be astronomical. Moreover, if we do not bound $\boldsymbol{\Phi}$, the CEGIS algorithm can easily iterate forever. A curious example of this is the sketch below, which requires that the ith bit of $??_0$ be equal to $i \ mod \ 2$.

```
void main(int i){

  int z = (??0 / pow(2, i) ) % 2;

  assert z == i % 2;
}
```

If we did not bound the set of possible values for $??_0$, then the CEGIS algorithm would iterate forever on this sketch which actually has no solution according to the synthesis semantics.

However, the CEGIS algorithm was design to exploit our intuition that a few inputs covering all the relevant corner cases should allow us to infer the correct solution to the sketch. As Fig. 5 shows the convergence properties for sketches representing more useful functions are surprisingly good. The largest number of iterations was for the tableBasedAddition benchmark, which implements an addition of two 4-bit numbers as a single table lookup, where all the entries in the table are left empty for the synthesizer to discover. For this benchmark, the number of iterations was, as we would expect, equal to the size of the input space, since each input provides information about only one entry in the table. For less contrived benchmarks, however, the CEGIS algorithm was very good at abstracting the entire input space into a few representative inputs.

Figure 5 also shows error bars of one standard deviation for each benchmark. The variability in the number of iterations

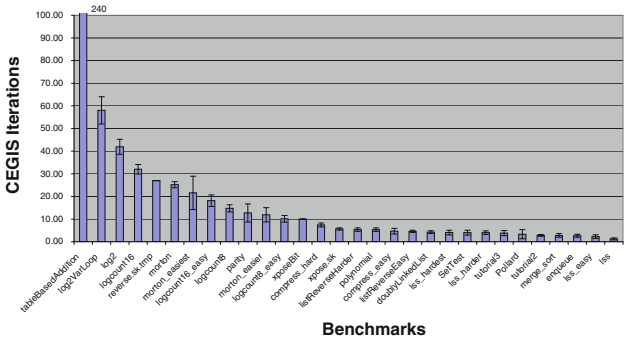


Fig. 5 Iterations per benchmark

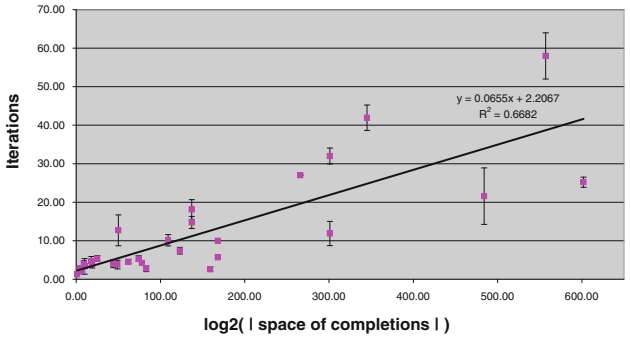


Fig. 6 Iterations versus distinct candidates

for a given benchmark comes from the non-deterministic choice the CEGIS algorithm makes in selecting a $\phi_i \in \Phi_i$, and from the choice of the σ_i that the validation phase decides to produce. However, you may notice that the number of iterations was fairly stable for each of the benchmarks. Of the 30 problems we tested, only 5 had a standard deviation of more than 2 iterations, and only 3 had a standard deviation larger than 4 iterations. This consistency suggests that there is something intrinsic to each benchmark that determines the number of observations needed for inductive synthesis to converge, irrespective of the non-deterministic choices made by the synthesizer.

We hypothesize that the number of iterations in the CEGIS loop is related to the number of candidates solutions to the sketch. Figure 6 shows the relationship between the number of CEGIS iterations and the logarithm of the size of the candidate space of the benchmark. The correlation is fairly strong, with an R^2 slightly over 0.66. The log of the size of the candidate space is not the same as the number of bits of holes, because some sketches exhibit a lot of redundancy: there are many combinations of hole values that after partial evaluation produce the same program. For example, the benchmark tuto−rial3, had a single generator which represented a family of $4 \times 10^{14} \approx 2^{49}$ syntactically distinct expressions, but used 259 integer holes, each represented with 5-bits. The fact that the number of iterations is better predicted by the number of unique candidates than the number of holes points to one of the strengths of the CEGIS approach: the ability to eliminate large classes of equivalent candidates with a single representative input.

These experiments support the bounded observation hypothesis; specifically, they demonstrate that for many interesting problems, the number of observations needed to find a valid control is quite small. Moreover, they show that the each new iteration is able to eliminate a fraction of the remaining candidate space, including huge numbers of equivalent solutions. Having shown this, it remains to be shown how effectively the inductive synthesis procedure is able to generate candidate solutions from sets of observations.

6 SAT-based inductive synthesis

The CEGIS procedure depends on an inductive synthesizer to generate candidate implementations from a small set of inputs, and a validation procedure to produce counterexample inputs exposing problems in invalid candidates. The previous section showed how the inductive synthesizer and the validator could be expressed in terms of the synthesis semantics through the following two equations.

$$\mathcal{C}[\![P]\!]^{\tau_\emptyset}(\sigma_{i-1}, \Phi_{i-1}) = (\sigma', \Phi_i) \tag{6.1}$$

$$\mathcal{C}[\![P]\!]^{\tau_\emptyset}(\sigma_i, \{\phi_i\}) = (\sigma', \emptyset) \tag{6.2}$$

Inductive synthesis is defined using Eq. (6.1), which describes how a set of candidate controls Φ_{i-1} is constrained to a set Φ_i by removing from it those controls that are invalid for input σ_{i-1}. The inductive synthesizer must then select a control $\phi_i \in \Phi_i$ which represents the solution to the inductive synthesis problem. Validation is defined through Eq. (6.2); it requires the synthesizer to select an input σ_i that shows that control ϕ_i cannot be in the set of solutions to the sketch synthesis equation, i.e. input σ_i causes an assertion failure on the candidate represented by control ϕ_i.

These two equations describe inductive synthesis and validation, respectively, but they are not algorithmic; the semantic rules describe manipulations on sets and functions in the abstract, but they do not tell us how these objects should be represented, or how the manipulations should be implemented. This section describes an implementation of the synthesis semantics that turns the inductive synthesis and validation problems into constraint satisfaction problems. The implementation is based on the well-known idea of representing sets symbolically as constraints; specifically, a set Φ of controls is represented as constraints that must be satisfied by all the controls in Φ. For example, the constraint $(\phi(??_0) = 5 \wedge \phi(??_1) < 3)$ represents the set of all controls that assign 5 to the hole $??_0$ and a value less than 3 to hole $??_1$. By representing sets of controls symbolically, we are able to derive systems of constraints for Φ_i by manipulating the constraints representing Φ_{i-1} according to the rules of

Table 1 Intermediate language used to represent parameterized values

Base	$e ::= c \mid h_{i,\tau} \mid in_i$
Arithmetic	$e ::= +(e_1, e_2) \mid -(e) \mid *(e_1, e_2) \mid \text{div}(e_1, e_2) \mid \text{mod}(e_1, e_2)$
Comparison	$e ::= <(e_1, e_2) \mid >(e_1, e_2) \mid \geq(e_1, e_2) \mid \leq(e_1, e_2) \mid =(e_1, e_2)$
Boolean	$e ::= \vee(e_1, e_2) \mid \wedge(e_1, e_2) \mid \oplus(e_1, e_2) \mid \neg(e)$
Selection	$e ::= mux_n(e_{idx}, e_1, \ldots, e_n) := e_{e_{idx}}$ $if_c(e_{ind}, e_{\neq}, e_{=}) := (e_{ind} = c) \ ? \ e_{=} : e_{\neq}$

the synthesis semantics. Extracting a control $\phi \in \Phi_i$ then becomes a constraint satisfaction problem.

To show how the constraint systems are constructed, we begin by describing the representations of controls. The number of holes in the program is bounded, and because we are restricting ourselves to bounded semantics, so is the number of calling contexts. Therefore, if we assume there are k distinct pairs of holes and calling contexts, we can represent ϕ as a *control vector*, $\langle h_0, \ldots, h_k \rangle$, where each control value h_i corresponds to the value of a specific hole under a specific calling context. The notation $h_{i,\tau}$ will sometimes be used to make explicit the exact hole and calling context for a given control value.

Now, recall that the values of expressions and variables are represented in the semantics as parameterized values, which are functions mapping controls to concrete values $\psi : \boldsymbol{\Phi} \rightarrow \mathbb{Z}$. The synthesizer represents parameterized values symbolically as expressions in the language described in Table 1. These expressions are represented in the synthesizer as DAGs, rather than trees, to allow sharing of common subexpressions. The base expressions in this language can be of three types:

- Controls $h_{i,\tau}$ indicating a specific component in the control vector.
- Integer constants.
- Input nodes in_i, which serve as place holders for concrete inputs.

The state σ is a mapping of variable names to parameterized values. Through the derivation process, the state is read and updated according to the synthesis semantics. For example, consider the various rules for evaluating expressions. Those rules are easily adapted to construct expressions in the intermediate language of Table 1.

$$\begin{aligned} \mathcal{A}[\![x]\!]^\tau\sigma &= \sigma(x) \\ \mathcal{A}[\![??_i]\!]^\tau\sigma &= h_{i,\tau} \\ \mathcal{A}[\![e_1 + e_2]\!]^\tau\sigma &= +(\mathcal{A}[\![e_1]\!]^\tau\sigma, \mathcal{A}[\![e_2]\!]^\tau\sigma) \end{aligned}$$

For example, the expression $\text{t} * ??_0 + ??_1$ is translated into an expression in the intermediate language through the following syntax directed translation.

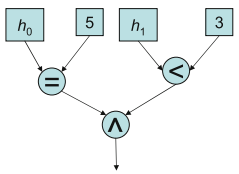
$$\begin{aligned} \mathcal{A}[\![\text{t} * ??_0 + ??_1]\!]^\tau\sigma &= +(\mathcal{A}[\![\text{t} * ??_0]\!]^\tau\sigma, \mathcal{A}[\![??_1]\!]^\tau\sigma) \\ &= +(*(\mathcal{A}[\![\text{t}]\!]^\tau\sigma, \mathcal{A}[\![??_0]\!]^\tau\sigma), h_{1,\tau}) \\ &= +(*(\sigma(\text{t}), h_{0,\tau}), h_{1,\tau}). \end{aligned}$$

The final expression is a function of the control values $h_{0,\tau}$ and $h_{1,\tau}$, and the input value of variable t.

Sets of controls are represented as constraints on the control values. To represent and manipulate these constraints, we exploit the intermediate language used to represent parameterized values by associating with each set Φ a characteristic function ψ_Φ related to Φ through the following equation.

$$\Phi = \{\phi : \psi_\Phi(\phi) \neq 0\}.$$

In other words, a control ϕ belongs to Φ if and only if it satisfies the constraint $\psi_\Phi(\phi) \neq 0$. For example, if Φ is the set of controls satisfying $\phi(??_0) = 5$ and $\phi(??_1) < 3$, this set will be represented with the characteristic function $\wedge(=(h_0, 5), <(h_1, 3))$, shown graphically below.



Most standard set operations are easy to perform on the symbolic representation. For example, if ψ_{Φ_1} and ψ_{Φ_2} are the characteristic functions for the sets Φ_1 and Φ_2, respectively, then the characteristic functions for the complement, intersection, and union of these sets are easy to construct from ψ_{Φ_1} and ψ_{Φ_2} as illustrated below.

$$\begin{aligned} \text{Complement } \neg\Phi_1 &= \neg(\psi_{\Phi_1}) \\ \text{Intersection } \Phi_1 \cap \Phi_2 &= \wedge(\psi_{\Phi_1}, \psi_{\Phi_2}) \\ \text{Union } \Phi_1 \cup \Phi_2 &= \vee(\psi_{\Phi_1}, \psi_{\Phi_2}) \end{aligned}$$

The rules of the synthesis semantics are used to construct the characteristic functions through syntax-directed translation. For example, the basic statements of assignment and assertion manipulate the state and the set of valid controls according to the following rules.

$$\begin{aligned} \mathcal{C}[\![x := e]\!]^\tau\langle\sigma, \psi_\Phi\rangle &= \langle\sigma[x \mapsto \mathcal{A}[\![e]\!]^\tau\sigma], \psi_\Phi\rangle \\ \mathcal{C}[\![\text{assert } e]\!]^{\tau_\emptyset}\langle\sigma, \psi_\Phi\rangle &= \langle\sigma, \wedge(\mathcal{A}[\![\sigma]\!]^e, \psi_\Phi)\rangle \end{aligned}$$

The same is true of the if statement; for the statement if e then c_1 else c_2, we can follow the synthesis semantics to evaluate the two branches of the conditional.

$$\psi_e = \mathcal{A}[\![e]\!]^\tau \sigma$$
$$\psi_{\Phi_t} = \wedge(\psi_\Phi, \psi_e)$$
$$\psi_{\Phi_f} = \wedge(\psi_\Phi, \neg(\psi_e))$$
$$\langle \sigma_1, \psi_{\Phi_1} \rangle = \mathcal{C}[\![c_1]\!]^\tau \langle \sigma, \psi_{\Phi_t} \rangle$$
$$\langle \sigma_2, \psi_{\Phi_2} \rangle = \mathcal{C}[\![c_2]\!]^\tau \langle \sigma, \psi_{\Phi_f} \rangle.$$

Then, the rule for the if statement becomes

$$\mathcal{C}[\![\text{if } e \text{ then } c_1 \text{ else } c_2]\!]^\tau \langle \sigma, \psi_\Phi \rangle = \langle \lambda x.mux_2(\psi_e, \sigma_2(x), \sigma_1(x)), \vee(\psi_{\Phi_1}, \psi_{\Phi_2}) \rangle$$

The rules for loops and procedure calls follow the same logic; the symbolic representations are manipulated according to the synthesis semantics, replacing set operations with operations on the characteristic functions. Because we are using bounded semantics, we do not have to worry about termination of loops or recursion.

An important advantage of representing sets as a constraint on the value of a characteristic function is that it is possible to query for a control in the set through a constraint satisfaction procedure. Any solution ϕ to the constraint $\psi_\Phi(\phi) \neq 0$ is guaranteed to belong to Φ; if the constraints are unsatisfiable, then it means that Φ is empty.

This property allows us to implement inductive synthesis by a straightforward application of the semantics; the synthesizer can simply evaluate $\mathcal{C}[\![P]\!]^\tau(\sigma_{i-1}, \Phi_{i-1})$ from its symbolic representation of Φ_{i-1}, and then query the representation for a control $\phi_i \in \Phi_i$.

The idea of representing sets as systems of constraints is not new. In fact, it was one of the major advances behind symbolic model checking [10]. However, SKETCH was the first system to represent the set of candidate solution to a synthesis problem as a SAT problem.

6.1 Validation

One of the advantages of separating inductive synthesis from validation is that any validation procedure that produces a counterexample input can be plugged into the algorithm. However, the SKETCH synthesizer exploits the symbolic machinery of inductive synthesis to perform bounded symbolic model-checking on the candidate solution.

The validation problem can be framed directly in terms of the synthesis semantics as the problem of finding an input σ_i that shows that the control ϕ_i is not a solution to the sketch equation, so

$$\mathcal{C}[\![P]\!]^{\tau_\emptyset}(\sigma_i, \{\phi_i\}) = (\sigma', \emptyset). \tag{6.3}$$

The SKETCH synthesizer uses symbolic reasoning to search for an input state σ_i satisfying the equation above using the exact same satisfiability procedure used for inductive synthesis.

Interestingly, the validation procedure that results from this symbolic manipulation is equivalent to the SAT-based bounded model checker developed by Clarke et al. [6]. Their tool, called CBMC, also translates a program into a set of Boolean constraints and uses SAT to solve the system for a counterexample. Some of their low-level encodings to SAT are different from outs, but the high-level ideas are the same.

Saturn [23] is another SAT-based validation tool that operates through similar principles. One of the key features of Saturn is that it is able to abstract procedures into summaries, allowing for modular verification, which our system does not support. What is most interesting about the similarity between our procedure with Saturn and CBMC is that the same techniques that proved successful for bug finding can be effective for inductive synthesis.

7 Experience

This section provides an overview of some of the problems that can be solved with sketch and gives some idea of the overall performance of the synthesizer. All the experiments described were performed on a very modest ThinkPad laptop with a single core Intel T1300 at 1.66GHz with 2MB of L2 cache and 1GB of memory. All the performance numbers in this section are averages from 4 to 7 different executions using different random seeds for the initial counterexample and the random restart in the SAT solver.

Figure 7 shows the solution times for several variations of 18 representative benchmarks. The exact quantities in the chart should be taken with a grain of salt, because they vary quite a bit from machine to machine, and with different versions of the solvers. However, they give a good idea of the relative complexity of solving different problems and of the scale of problems that can be solved at interactive speeds. The benchmarks were chosen to be representative of real programming problems which the synthesizer can solve in less than 15 min. Many of these benchmarks were written by our group, but nine of them were developed by students from a graduate introductory programming languages class held at UC Berkeley in the fall of 2007. The benchmarks can be roughly categorized into three groups: bit manipulations, integer manipulations and linked data structures.

Bit manipulations What characterizes these benchmarks is that they treat machine words as bit-vectors. All these benchmarks use the imple−ments directive to provide specifications in the form of reference implementations that manipulate each bit individually. The sketches contain the necessary insight to take advantage of bit-level parallelism. These benchmarks were our first application of sketching because the low-level details almost always involve discovering bit-masks and precise shift amounts, so we could write sketches

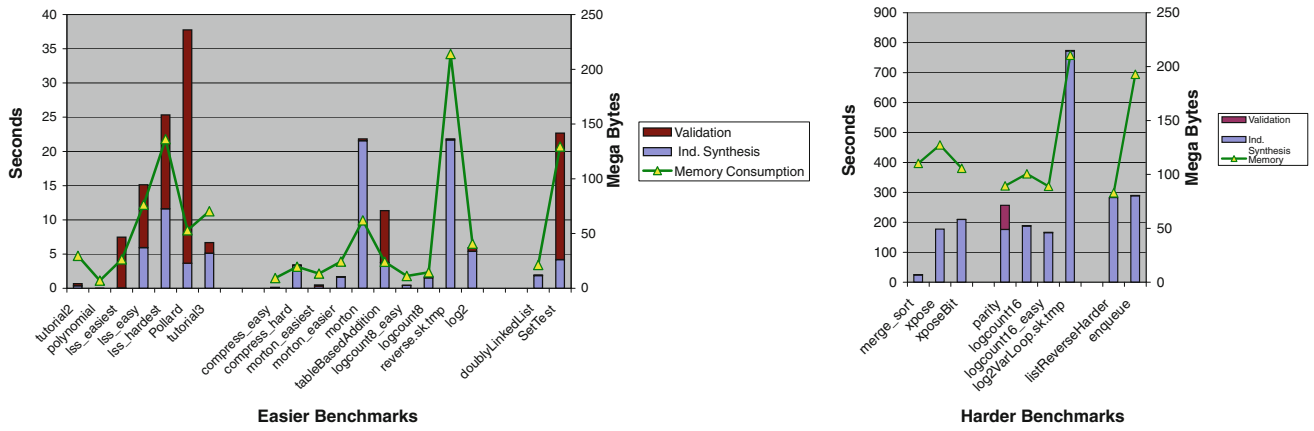


Fig. 7 Solution time and memory consumption for selected benchmarks

for them using only the integer holes, even before we had any of the higher level sketching constructs.

Example A typical benchmark of this category is the morton benchmark, written by graduate student Jacob Burnim. A 32-bit morton number is computed by interleaving the bits of two 16 bit integers x and y, so that bit r_{2*i} of the result equals bit x_i of x, and bit r_{2*i+1} corresponds to bit y_i of y. According to Anderson, "Morton numbers are useful for linearizing 2D integer coordinates, so x and y are combined into a single number that can be compared easily and has the property that a number is usually close to another if their x and y values are close" [2].

It is easy to interleave the bits of two 16-bit integers by selecting the bits one by one, but it is possible to do it more efficiently by taking advantage of the ability to shift all the bits in a word with a single instruction; while the bit-by-bit approach takes $O(W)$ operations for a word of size W, the task can be achieved with $O(log(W))$ operations using bit-vector parallelism. The high-level insight can be stated as follows.

> First, scatter the 16 bits of each of the two inputs across the even bits of a 32 bit word. Then, or together the resulting two words, shifting one of the words by one to align its bits with the gaps in the other word. The scatter can be done with log-shifting, a technique for efficiently scattering or gathering bits by shifting many bits at a time as illustrated in Fig. 8. A logshifter can be implemented by repeatedly shifting some bits, oring them with the original word, and then masking the result.

The insight can be expressed succinctly in a sketch. The log−shift generator encapsulates the basics of logshifting, but leaves unspecified the tricky details of exactly what bits to mask and how much to shift on each step; this means that the generator could actually be reused to implement other scattering patterns different from the one required for this

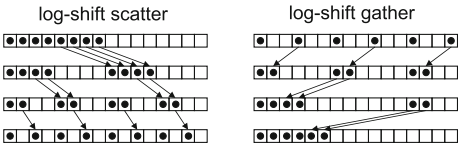


Fig. 8 Examples of log-shifting for scattering and gathering bits

problem. The sketch also leaves unspecified the number of steps required for the logshifter; this is a potential problem because it gives the synthesizer the freedom to include more steps than necessary. The student produced a second version of this benchmark (morton_easiest) that specifies that on each iteration the shift amount should be reduced by half. This version of the benchmark is guaranteed to produce the desired answer, and resolves much faster because of the added information.

```
int W = 16;
generator bit[2*W] logshift(bit[2*W] in){
   int pt = 4*W;
  repeat(??){
    // Shift some of the bits, and mask
    // their original positions.
    in = (in | (in << ??)) & ??;
  }
}
bit[2*W] morton(bit[W] x, bit[W] y)
                implements mortonSpec{
  bit[2*W] x2 = logshift(x);
  bit[2*W] y2 = logshift(y);
  return x2 | (y2 << 1);
}
```

□

All the benchmarks in the second group in Fig. 7 are bit manipulation benchmarks. These benchmarks are difficult to

solve despite their relatively small size (compress is the largest one of these sketches and it is only 47 lines of code). There are two reasons for this. First, their candidate spaces are often huge; a single 32-bit mask will have billions of possible solutions. Moreover, the holes are often very tightly coupled, in the sense that every bit in the output potentially depends on the value of every single hole, as was the case in the morton example. This makes these benchmarks very challenging for the solver. On the other hand, they are a great domain for sketching because it is very challenging to program by hand, and there is often a very good match between the insight and the sketch. Moreover, because these benchmarks are inherently bounded, the SAT-based validation procedure can provide absolute correctness guarantees.

Integer manipulations These benchmarks manipulate integers or arrays of integers; with the manipulations typically involving some arithmetic. Their specifications also consist of reference implementations, while the sketches often must take advantage of some mathematical principle to achieve better performance at the expense of clarity. All the benchmarks in the first group in Fig. 7 are integer manipulation benchmarks.

Example A great example from this domain is the Karatsuba multiplication algorithm for large integers (karatsuba). The algorithm is a building block of many public key cipher implementations. It uses a divide and conquer approach to multiply integers with N digits in $O(N^{1.585})$, as opposed to the standard $O(N^2)$ from the grade school multiplication algorithm.

The algorithm starts by decomposing two N-digit numbers x and y into two halfs: $x = x_1 b^{N/2} + x_0$, $y = y_1 b^{N/2} + y_0$, where b is the base. The standard multiplication can be defined recursively in terms of the two halfs.

$$x * y = b^N x_1 * y_1 + b^{N/2}(x_1 * y_0 + x_0 * y_1) + x_0 * y_0$$

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

Let us illustrate how Karatsuba might have been able to invent (and implement) his algorithm with the assistance of sketching. He would first observe that it may be possible to replace the four expensive multiplications with three expensive multiplications. He would guess that one cannot avoid computing terms $x_0 * y_0$ and $x_1 * y_1$, so he would focus on replacing the term $x_1 * y_0 + x_0 * y_1$ with a one-multiplication term. This optimization would be performed at the expense of adding big-integer additions or subtractions, a good trade-off since their complexity is linear rather than quadratic. In mathematical notation, the idea can be expressed in the following sketch, where the generator $poly(n, x_1, \ldots, x_k)$ produces a polynomial in k variables of degree n.

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

It turns out that the idea for this optimization is correct and the correct formula is shown below.

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

Creating an implementation using the SKETCH system is just as simple. The sketch in Fig. 9 contains the same insight expressed above, but it also addresses the representation issues for the integers and their operations. Integers are represented as N element arrays of ints; addition, complement and shifting are all provided through separate routines. The half ranges are read from the original input array using special array notation available in the language, where A[a::b]

```
int[N*2] k (int[N] x, int[N] y) implements mult {
    if (N<=1) return mult(x,y);
    int[N/2] x1, x2, y1, y2;
    int[N] a=0, b=0, c=0;
    int[N*2] out = 0;

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

    a = multHalf(x1, y1);
    b = multHalf(x2, y2);
    c = multHalf(poly1(x1,x2,y1,y2), poly1(x1,x2,y1,y2));

    out = a;
    out = plus(out, shift(b, N/2));
    repeat(??){
     int[N] t = {| a | b | c |};
     out = plus( out,
           shift( {| t | minus(t) |}, {| N | N/2 | 0 |} ) );
    }
    out = normalize(out);
    return out;
}

int[N/4] poly1(int[N/4] a, int[N/4] b,
               int[N/4] c, int [N/4] d){
        int[N/4] out = 0;
        if(??) out = plus(out, {| a | minus(a)|} );
        if(??) out = plus(out, {| b | minus(b)|} );
        if(??) out = plus(out, {| c | minus(c)|} );
        if(??) out = plus(out, {| d | minus(d)|} );

        return out;
}
```

Fig. 9 Sketch for Karatsuba's multiplication

correspond to a range of b elements in A starting with the element at position a. Multiplications by the base term are encoded through a shift operation.

Ideally, we would like for the routine to be parametrized by N, and the solver to guarantee the result for all N. Unfortunately, the SKETCH solver cannot reason about unbounded operations, so the correct answer was derived by setting N to 4, and limiting the range of integer values to two bits. □

The karatsuba benchmark illustrates many relevant aspects of integer benchmarks. First, because we use bounded model checking as our correctness criteria, we cannot provide strong correctness guarantees. For most of these benchmarks, validation was performed for all integer inputs in the range [0, 8]. Only the tutorial benchmarks were validated for inputs in the range [0, 32]. For these benchmarks these ranges happened to be sufficient in the sense that the programs that were correct for all inputs between 0 and 8 turned out to be correct programs, but this could only be ascertained through hand examination of the result.

The ranges of inputs are fairly small even by the standards of bounded model checking. This is partly a consequence of the use of the unary representation of integers used internally by the solver. This representation is very efficient when representing integers ranging over a small set of values, but it grows very quickly, making it impractical to validate sketches over a wide range of input values. It is very likely that the growing power and availability of SMT solvers capable of reasoning about integers will have a big impact on these benchmarks. In spite of this, the synthesizer is able to quickly produce correct implementations from sketches with a lot of freedom for many interesting kernels.

Linked data structures These benchmarks involve manipulation of data structures in the heap. The linked list reversal from the introduction is an example of this class of benchmark.

Example Another interesting benchmark in this category is the Set−Test benchmark. This benchmark implements a tree-based set using a hash table as a reference implementation. One of the problems that make tree manipulation tricky is symmetry: the code for the different cases is very similar except some cases have to use the left child and some have to use the right child, and it is easy to get confused about which child should be used where. Sketching allowed us to eliminate this redundancy using generators. The fragment of the Set−Test sketch shown below uses a generator to produce the code that decides whether to add a new node as a child of the current node or to continue traversing. The generator will produce the correct code both for the case when n.val is less than v and when it is not.

```
generator bit choice(ref TreeNode n, int v){
    if({| n(.left | .right) |} == null){
        {| n(.left | .right) |} = newTreeNode(v);
        return ??;
    }else{
        n = {| n(.left | .right) |} ;
        return ??;
    }
}

bit add(Tree t, int v){
    TreeNode n =t.root;
    if(n == null){
        t.root = newTreeNode(v);
        return ??;
    }
    while(n != null){
        if( n.val == v){ return ??; }
        if(n.val < v){
            if(choice(n, v)){
                return ??;
            }
        }else{
            if(choice(n, v)){
                return ??;
            }
        }
    }
    return ??;
}
```

□

Like the integer manipulation benchmarks, data-structure benchmarks also have to cope with the limitations of the validation procedure. Our validation procedure cannot guarantee the absolute correctness of the synthesized implementation, only its correctness against a bounded test harness. For example, the test harness for the enqueue benchmark checks the equivalence of the sketched queue with an array implementation on an input-directed sequence of operations.

This type of test harness is often referred to in the verification literature as a "most general client" [1] because it verifies that the data-structure works correctly for all sequences of up to N operations, which is a very good, but it is not the same as verifying that the queue is correct.

Another interesting feature of the data-structure benchmarks, especially when compared with the bit manipulations, is that one can leave a great amount of code unspecified while keeping the search space relatively small. For example, in the listReverseHard benchmark, the assignments in the body of the loop specified remarkably little, leaving a lot of freedom to the synthesizer, but the synthesizer only had 60 different

possibilities to search through for each assignment. By contrast, a single bit-mask in the morton benchmark can have 2^{32} different possible values. This means that sketches can be allowed to have a lot of freedom without overwhelming the synthesizer. At the same time, we can see that the solution times for these benchmarks can be quite large given their small input and candidate spaces, which seem to suggest that our very naïve representation of the heap may have a lot of room for improvement.

Overall, these benchmarks are not such a good match for sketching, in the sense that there is a lot of boilerplate that programmers have to write before the synthesizer is able to synthesize an implementation. At the same time, programmers often have strong intuitions about how these data-structure manipulations work, which are not reflected in the sketch. Nevertheless, these benchmarks provide a good stress test of the capabilities of the sketch synthesizer.

8 Comparison with QBF

One of the original motivations for the CEGIS algorithm was the difficulty of solving constraint systems with multiple quantifiers. Specifically, we have seen that the synthesis problem reduces to a 2QBF problem of the form shown below.

$$\exists \phi \ \forall \sigma \ \ Q(\phi, \sigma). \tag{8.1}$$

The predicate Q is a boolean formula, so the equation above is a satisfiability problem on a quantified boolean formula (QBF). Over the last few years, there has been a lot of interest in QBF solvers. Every year, there is even a QBF competition held side by side with the annual SAT competition at the International Conference on Theory and Applications of Satisfiability Testing. Therefore, an important question is: How does the CEGIS algorithm compare with general QBF solvers?

To answer this question, we generated QBF problems for four representative benchmarks of varying degrees of difficulty: polynomial, doublyLinkedList, lss_hardest and parity. The QBF problems were generated from the optimized constraint system, so the QBF solver could benefit from all the high-level optimizations available to the SKETCH synthesizer. It is worth noting that even though there are only two quantifiers in Eq. (8.1), this is actually a 3-QBF problem, because converting Q to conjunctive normal form requires the introduction of temporary variables which are existentially quantified.

$$\exists \phi \ \forall \sigma \ \exists t \ \ Q_{cnf}(\phi, \sigma, t) \tag{8.2}$$

The QBF formulas from the four benchmarks were fed to `2clsQ`, the winner of the 2006 QBF competition [13], and `quantor` version 3.0, the winner of the 2008 competition [4]. In both cases the results support the CEGIS approach to resolving sketches.

In the case of `2clsQ`, the performance difference was overwhelming. Of the four benchmarks, `2clsQ` was only able to resolve polynomial, the easiest one. For this benchmark, `2clsQ` took 94 s to find a solution, compared to 0.1 s it took SKETCH with MiniSat. `2clsQ` was unable to solve any of the other three benchmarks in the 20 min of allotted time, while SKETCH was able to solve par−ity, the hardest of these benchmarks, in 257 s using MiniSat, and in only 11 s using ABC.

Benchmark	SKETCH time (s)	CEGIS Iters.	`2clsQ` time
polynomial	0.1	5.3	94 s
doublyLinkedList	2.6	4	>20 min
lss_hardest	25	4.3	Out of memory
parity	257	15	>20 min

`quantor` did much better on the easier benchmarks, but it was still unable to compete with CEGIS on the harder problems. For polynomial and doublyLinkedList, `quantor` finished in about the same time as SKETCH. For both parity and lss_hard, however, `quantor` exhausted all available memory after the first 2 min of execution. After this, the system started thrashing and became unresponsive, so the execution had to be stopped. By contrast, SKETCH was able to solve both of these benchmarks using less than 150MB of memory.

Benchmark	SKETCH (s)	SKETCH Mem. (MB)	`Quantor` time
polynomial	0.1	7	0.15 s
doublyLinkedList	2.6	16	3.4 s
lss_harder	25	136	Out of memory
parity	257	89	Out of memory

There is an alternative encoding into a QBF problem with avoids the third quantifier; the idea is to negate Eq. (8.1) before converting the predicate into CNF.

$$\forall \phi \ \exists \sigma \ \ \bar{Q}(\phi, \sigma). \tag{8.3}$$

Then, $\bar{Q}$, the negation of Q, can be converted to CNF without introducing an additional quantifier alternation.

$$\forall \phi \ \exists \sigma \ \exists t \ \ \bar{Q}_{cnf}(\phi, \sigma, t). \tag{8.4}$$

Now, the QBF solver must find a ϕ that falsifies the equation above. However, this encoding proved to be even worse than the previous one; with this encoding, `quantor` was unable to solve even the polynomial problem without running out of memory.

8.1 2QBF solvers and quantifier elimination in SMT

Ranjan et al. [12] have shown that for 2QBF problems, specialized algorithms can be more efficient than the algorithms used by general QBF solvers. In [12], they present two algorithms that are also based on two interacting solvers where one produces candidate solutions and the other one checks them. However, the interaction between the two algorithms is more limited than in our approach. In particular, every iteration of both algorithms adds a single clause to the solver in charge of producing candidate solutions. This clause is computed by taking the failed solution and computing a *cover set* for it, i.e. a partial assignment that is computed by eliminating those variables whose values were not used in determining that the solution had failed. The new clause rules out this partial assignment from appearing again in a candidate solution. By contrast, our approach does not require us to compute a cover set, and the clauses added after each iteration of our approach are a strictly stronger than the single clause added by these algorithms, so we can expect our algorithm to converge in fewer iterations.

Very recently, Wintersteiger et al. [22] have worked on adding support for quantifiers to their SMT solver. Their approach combines a basic counterexample-guided refinement approach similar to CEGIS with other techniques such as substitution and term rewriting. Their approach is more recent than our original CEGIS algorithm [19] and is targeted towards a more general use of quantifiers in SMT problems. Their approach was actually inspired by the work of Jha et al. [9] and Srivastava et al. [20], both of whom were influenced by our earlier work.

The CEGIS approach is unlikely to beat the QBF solvers on arbitrary QBF problems. However, on sketching problems, the CEGIS algorithm is able to exploit the bounded observation hypothesis and efficiently synthesize a correct candidate from only a small set of inputs. Moreover, the CEGIS approach has the useful property of separating synthesis and validation, allowing the best techniques to be used for each of these two functions. For example, in another paper [18] we used CEGIS to do synthesis in the context of concurrent algorithms. For that domain, we were able to get significant scalability benefits from using an explicit state model-checker (SPIN) in place of SAT for the verification phase. This important flexibility is lost if we see the problem as a monolithic 2QBF problem.

9 Case study: sketching AES

As a case study [19], we used the SKETCH synthesizer to create a full implementation of the AES cipher [7] by synthesizing its most difficult fragments. For this experiment, we created a reference implementation by directly transcribing the NIST standard into code. The NIST standard defines the cipher in terms of 14 rounds which take a 128-bit input block and a round key and processes it, followed by a final round.

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

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

In the optimized version, all the operations in the round are folded into a set of table lookups. A programmer implementing AES by traditional means would have to derive the formula for generating the table entries; this may be difficult if one is not familiar with the algebra involved. The programmer would then have to write an ad hoc code generator to produce the table from the specification through some algebraic manipulation, and then would have to incorporate the generated table into the code and check the correctness of the cipher using known input/output pairs.

By contrast, SKETCH is able to synthesize the tables automatically and verify their correctness against the reference implementation. Figure 10 shows the sketch for the regular round. The sketch for the final round is similar, except that it uses only one table instead of four, and it combines outputs from the tables using masks—which are left unspecified—instead of xors.

The roundSK sketch places a lot of stress on the solver since there are 32,768 bits in the table that have to be generated. Furthermore, each input considered by the solver helps complete only a small number of table entries, so the synthesize/verify loop has to iterate 655 times. Nonetheless, the solver is able to complete the sketch in about an hour. Table 2 shows the exact times spent by the two SAT solvers involved. All instances of synthesis were solved using MiniSat. For verification, we used MiniSat for the first 645 iterations. For the last 10 iterations we switched our SAT solver to ABC [11] because it provides much better performance for hard SAT problems.

```
int[4] roundSK(bit[32][4] in, bit[32][4] rkey)
    implements round{
  bit[32][4][256] T = ??; // synthesize 32768 bits in T
  bit[32][4] output = 0;
  bit[32] mask = 0x000000FF;
  int[4][4] ch = {{0,1,2,3},{1,2,3,0},
                  {2,3,0,1},{3,0,1,2}};
  for(int i=0; i<4; ++i){
    int i0 = (int) in[ch[i][0]] & mask; // 1st 8 bits
    int i1 = (int)(in[ch[i][1]] >> 8) & mask; // 2nd 8 bits
    int i2 = (int)(in[ch[i][2]] >> 16) & mask; // 3rd 8 bits
    int i3 = (int)(in[ch[i][3]] >> 24) & mask; // last 8 bits
    output[i] = T[0][i0] ^ T[1][i1]
              ^ T[2][i2] ^ T[3][i3];
    output[i] = output[i] ^ rkey[i];
  }
  return output;
}
```

Fig. 10 Sketch for one round of AES

Table 2 Solution time for roundSK in AES benchmark

Total Synth:	791 s	=13.183 min
Total Verify:	3,942 s	=65.7 min
Synth easy:	1.17 s	Avg time per SAT problem
Synth hard:	3.4 s	Avg time per SAT problem
Verify easy:	5.33 s	Avg time per SAT problem
Verify hard:	50 s	Avg time per SAT problem

The times for Synth and Verify hard correspond to the times for the last 10 iterations

Performance of generated code. The resulting code was run against a hand optimized AES implementation from open SSL. The runtime for 50,000 encryptions was as follows:

OpenSSL AES	19.652 ms
Sketch	21.307 ms
Spec	19936.100 ms

The difference between the hand coded AES and the sketched version is less than 10 %, the difference due to the fact that the hand optimized code was written in a way that caused the compiler to do a better job at register allocation. We can also see that the original specification, which is very close to the specification of AES [7], is over 1,000 times slower.

10 Conclusions

The paper has explored the power of the SKETCH system to synthesize the low-level details for small but complex programs in a variety of domains. In addition to that, it has shown some potential avenues for significant improvement in the synthesizer's performance.

There are a number of other strategies that could significantly improve the performance of the solver. For example, using constraint solvers that can reason about integers could make a big difference for integer problems. Similarly, improved search strategies that take advantage of more semantic information about the sketch could make the search more efficient. Additionally, there is great room for improvement in the encoding of higher-level sketching constructs and advanced language features; the current encoding is very simple and naïve.

Beyond performance, however, it is important to keep in mind that a synthesizer is a productivity tool. The ultimate test for any optimization is the extent to which it is able to improve programmer productivity. A detailed analysis of how improvements in performance affect programmer productivity is one of the great omissions in this work. Quantifying this impact requires user studies and analysis of the use of sketching in the field. Overall, some of the bigger open issues in sketching involve the following areas:

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 Fig. 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 [17], 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.

References

1. Amit, D., Rinetzky, N., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: In 19th International Conference on Computer Aided Verification (CAV) (2007)
2. Anderson, S.E.: Bit twiddling hacks (1997–2005). http://www-graphics.stanford.edu/~seander/bithacks.html
3. Angluin, D., Smith, C.H.: Inductive inference: theory and methods. ACM Comput. Surv. **15**(3), 237–269 (1983)
4. Biere, A.: Resolve and expand. In: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing, SAT'04, pp. 59–70. Springer, Berlin (2005)
5. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003)
6. Clarke, E., Kroening, D.,Yorav, K.: Behavioral consistency of c and verilog programs using bounded model checking. In: Proceedings of the 40th Annual Design Automation Conference, DAC '03, pp. 368–371. ACM, New York (2003)
7. Advanced Encryption Standard (AES): U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology, November (2001). http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
8. Gold, E.M.: Language identification in the limit. Inf. Control **10**(5), 447–474 (1967)
9. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE '10, vol. 1, pp. 215–224. ACM, New York (2010)
10. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
11. Mishchenko, A., Chatterjee, S., Brayton, R.: Dag-aware AIG rewriting: a fresh look at combinational logic synthesis. In: DAC '06: Proceedings of the 43rd Annual Conference on Design Automation, pp. 532–535. ACM Press, New York (2006)
12. Ranjan, D.P., Tang, D., Malik, S.: A comparative study of 2qbf algorithms. In: The Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), May (2004)
13. Samulowitz, H., Bacchus, F.: Binary clause reasoning in qbf. In: Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing, SAT'06, pp. 353–367. Springer, Berlin (2006)
14. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for c. In: ESEC/SIGSOFT FSE, pp. 263–272 (2005)
15. Shapiro, E.Y.: Algorithmic Program Debugging. MIT Press, Cambridge (1983)
16. Solar-Lezama A.: Program Synthesis By Sketching. PhD thesis, EECS, UC Berkeley (2008)
17. Solar-Lezama, A., Arnold, G., Tancau, L., Bodík, R., Saraswat, V., Seshia, S.: Sketching stencils. In: PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, vol. 42, pp. 167–178. ACM, New York (2007)
18. Solar-Lezama, A., Jones, C., Arnold, G., Bodík, R.: Sketching concurrent datastructures. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation. Tucson, June 7–13 (2008)
19. Solar-Lezama, A., Tancau, L., Bodík, R., Saraswat, V., Seshia, S.: Combinatorial sketching for finite programs. In: ASPLOS'06. ACM Press, San Jose (2006)
20. Srivastava, S., Gulwani, S., Foster, J.: From program verification to program synthesis. POPL, Madrid (2010)
21. Summers, P.D.: A methodology for lisp program construction from examples. J. ACM **24**(1), 161–175 (1977)
22. Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: Efficiently solving quantified bit-vector formulas. In: Bloem, R., Sharygina, N. (eds.) FMCAD, pp. 239–246. IEEE (2010)
23. Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 351–363 (2005)