Program Sketching by Automatically Generating Mocks from Tests

Nate F. F. Bragg[1], Jeffrey S. Foster[1], Cody Roux[2], and Armando Solar-Lezama[3]

[1] Tufts University, Medford, MA 02155, USA
{nate,jfoster}@cs.tufts.edu
[2] Draper Laboratory, Cambridge, MA 02140, USA
croux@draper.com
[3] Massachusetts Institute of Technology, Cambridge, MA 02139, USA
asolar@csail.mit.edu

Abstract. Sketch is a popular program synthesis tool that solves for unknowns in a *sketch* or partial program. However, while Sketch is powerful, it does not directly support modular synthesis of dependencies, potentially limiting scalability. In this paper, we introduce Sketcham, a new technique that modularizes a regular sketch by automatically generating *mocks*—functions that approximate the behavior of complete implementations—from the sketch's test suite. For example, if the function f originally calls g, Sketcham creates a mock g_m from g's tests and augments the sketch with a version of f that calls g_m. This change allows the unknowns in f and g to be solved separately, enabling modular synthesis with no extra work from the Sketch user. We evaluated Sketcham on ten benchmarks, performing enough runs to show at a 95% confidence level that Sketcham improves median synthesis performance on six of our ten benchmarks by a factor of up to 5× compared to plain Sketch, including one benchmark that times out on Sketch, while exhibiting similar performance on the remaining four. Our results show that Sketcham can achieve modular synthesis by automatically generating mocks from tests.

Keywords: Program synthesis, mocks, Sketch

1 Introduction

Program synthesis by sketching, as embodied by the Sketch synthesis tool [30], is a popular technique that has been applied to a wide variety of problems [5,7,13,14,15,16,18,22,29]. A Sketch input (henceforth a *sketch*) is a program written in a C-like language augmented with *holes*, unknown constants, and *generators*, unknown expressions. The solution for a sketch is specified using test cases called *harnesses*, also written in the Sketch language, that make assertions about the results of to-be-synthesized code. Sketch searches for a solution using *counterexample-guided inductive synthesis (CEGIS)*, which alternately synthesizes a candidate solution and then uses a verifier to check the assertions; any counterexamples from verification feed into the next round of synthesis [27].

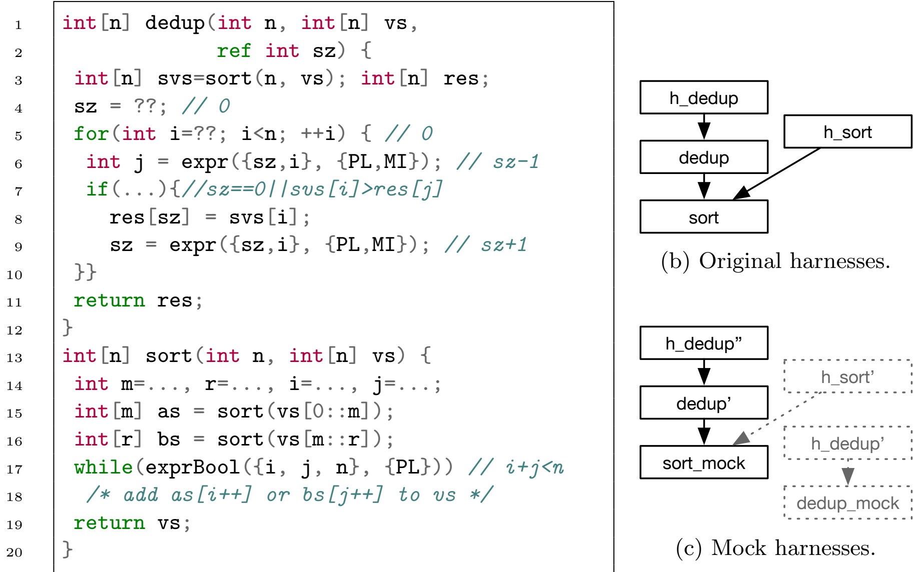
© The Author(s) 2021
A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12759, pp. 808–831, 2021.
https://doi.org/10.1007/978-3-030-81685-8_38

One key challenge of using Sketch is that it does not specifically support modular synthesis. More precisely, even if an input sketch is divided into a number of functions that call each other, Sketch solves them all together. This approach potentially limits scalability, as SAT formulas created by Sketch can grow quite quickly as function calls are inlined. A Sketch user could potentially work around this by manually replacing calls to to-be-synthesized functions with calls to Sketch *models* [24], which are *mocks*, i.e., functions that, in place of full implementations, approximate the desired behavior with a specification in the form of assertions about individual cases. However, writing additional specifications is both time consuming and redundant with developing the original sketch.

In this paper, we introduce Sketcham (short for *Sketch and Mocks*), a novel technique that converts a regular sketch problem into a modular sketch problem by *automatically* generating mocks from harnesses. More specifically, suppose Sketcham is given a sketch in which function f calls g and g is tested by harness h. Sketcham first converts h into a mock g_m that has the same function signature as g but whose body encodes the assertions from h. Then, Sketcham augments the original sketch with new code in which f calls g_m instead of g, thereby allowing f to be synthesized separately from g. Thus, by converting tests (harnesses) to mocks (specs), Sketcham enables modular synthesis without extra work from the user. Section 2 gives an overview of Sketcham.

Sketcham generates the new, modular sketch problem using a sequence of three algorithms. First, Sketcham traverses the original sketch to build a mapping A from function names to a set of assertions in which each function is called. Note that we place some limitations of the assertions—e.g., they can contain at most one function call—to guarantee we can always translate them from harness assertions to mock assertions. Next, Sketcham traverses A, generating a mock f_m for each function $f \in dom(A)$, where f_m encodes the assertions in $A(f)$. Finally, Sketcham generates new mock harnesses that are the same as the original harnesses, except they call mocks instead of the underlying functions. Section 3 presents Sketcham's core algorithms.

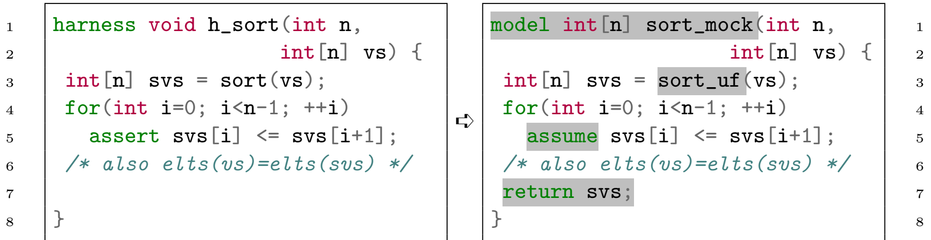
We implemented Sketcham as an additional pass to Sketch, which we evaluated on ten benchmarks. We found a high variance in running time, both under Sketch and under Sketcham. To account for this difference, we used the Clopper-Pearson method [6], running each configuration (synthesis tool–benchmark combination) up to 1,487 times, reaching 95% confidence that the true median running time lies within 20% of the experimental median, excluding failures and runs exceeding a 60 minute timeout. We found that, for six of ten benchmarks, Sketcham runs up to 5× faster than Sketch; for one benchmark Sketcham is up to a factor of 0.98× slower; for the remaining three benchmarks, performance is indistinguishable. We examined one benchmark, deduplication of elements in an array, in detail. We found that the performance improvement is largely due to a mock that does a thorough job representing the function it mocks, and that the performance improvement occurs during the CEGIS synthesis phase rather than the CEGIS verification phase. Section 4 presents our evaluation.



(a) dedup and sort (simplified).

(b) Original harnesses.

(c) Mock harnesses.



(d) Translating sort's test harness into a mock.

Fig. 1: Sketcham applied to deduplication via sorting.

In summary, Sketcham demonstrates that modular synthesis can be achieved by automatically generating mocks from tests (specs from harnesses) without additional user effort.

2 Overview

To illustrate Sketcham, consider Figure 1a, which shows a simplified sketch whose solution deduplicates an array of integers. This sketch makes use of Sketch holes `??`, which are unknown constants, and generators such as `expr(vars, ops)`, which is an unknown expression composed of variables `vars` combined with

operands `ops`, including `PL` for addition and `MI` for subtraction. The correct solutions for the holes and generators are shown in end-of-line comments.

At the top of Figure 1a, function `dedup` takes a length `n` and array `vs`, and it returns the deduplicated array and, by reference, the deduplicated array's length `sz` (in Sketch, functions can only have at most one return value, hence the return-by-reference `sz`). The `dedup` function begins by calling another function, `sort`, to sort the array (line 3). Then it initializes `sz` to a hole and loops through the array (lines 4-5). In each iteration, it computes an expression `j` of `sz` and `i` (line 6) used in a conditional guard (line 7; details of guard not shown). If the condition holds, the element at position `i` is copied into `res` and `sz` is updated; otherwise the element is ignored. Finally, `dedup` returns the result array `res`.

The `sort` function (line 13) takes the length and array and returns a sorted array. This particular sketch is for merge sort. Here the programmer knows that merge sort involves sorting two sub-arrays but isn't sure about the details. After some initialization (not shown), it makes two recursive calls to sort sub-arrays (lines 15 and 16). Then it loops over the sorted sub-arrays, merging the elements into array `vs`, which is returned. The loop guard (line 17) uses a different generator, `exprBool(vars, ops)`, that generates arithmetic comparisons (`<`, `<=`, etc) among expressions generated by calling `expr(vars, ops)`.

Harnesses and Mocks. To test the expected behavior of `dedup` and `sort`, the sketch also includes two harnesses, `h_dedup` and `h_sort`. Figure 1b shows the call graph of the sketch with the harnesses, and the left side of Figure 1d shows a portion of `h_sort` (we omit `h_dedup` for brevity). This harness calls `sort` and then makes assertions about the results, e.g., that the output array is sorted. Harnesses are distinguished from regular functions by the keyword `harness`, and their arguments are treated as universally quantified. Thus, `h_sort` tests that for all `n` and arrays `vs` of length `n`, the `sort` function is correct.

To solve this synthesis problem, Sketch converts `dedup`, `sort`, and a harness into a single SAT formula and then uses CEGIS to find a solution. This approach works, but the formula passed to the solver is large, because it contains both functions' worth of code, and complex, because reasoning about the code in `dedup` requires simultaneously reasoning about the code in `sort`. Thus, mashing together both functions into a single SAT formula potentially limits the scalability of Sketch.

The key idea of Sketcham is to observe that this sketch is actually modular—it has been divided into two functions, each with their own tests. Sketcham takes advantage of this modularity by creating a new synthesis problem that includes mock versions of functions in the sketch, which can then be used to enable separate reasoning about each function.

The right side of Figure 1d shows `sort_mock`, the mock version of `sort`. The mock has the same signature as `sort`, but instead of containing the actual sorting code, it contains assertions from `h_sort` about `sort`'s expected behavior. In detail, in place of calling `sort`, the mock calls a fresh uninterpreted function `sort_uf` on line 3. Then it makes assumptions (rather than assertions) about the result array `svs` (line 5), and finally returns `svs` (line 7). The mock itself is a

```
int doub(int m) {
  return m * 2;
}
harness void h(int n) {
  int out = doub(n * 10);
  assert out == (n + n) * ??;
}
```

(a) Double.

```
model int doub_mock(int m) {
  int out = doub_uf(m);
  assume (0 == m%10)  ==>
      out == (m/10 + m/10) * ??;
  return out;

}
```

(b) Mock double.

Fig. 2: The double function and its mock.

Sketch model (indicated by the `model` keyword), and where the mock is called, Sketch will replace the call with the assumptions in the model's body [24].

Next, Sketcham creates new code that uses the mock, as shown in Figure 1c. (Here the dashed, greyed boxes are for functions and harnesses that are generated but do not improve solving time; see Section 4.2.) In particular, `dedup'` is the same as `dedup`, except it calls `sort_mock` instead of `sort`, and `h_dedup''` is the same as `h_dedup` but it calls `dedup'` instead of `dedup`.

The final sketch includes `h_dedup''`, `h_dedup'` (a trivial harness that calls a mocked `dedup`), and `h_dedup`—in that order—as well as the harnesses for `sort`. Sketcham searches for a solution for each harness in order, i.e., it tries to solve `h_dedup''` first. Notice that, critically, when Sketcham solves `h_dedup''`, it need not consider the code of `sort`, but rather only its specification as encoded in the mock. In practice, this means that Sketcham can solve `h_dedup''` up to 18.1× faster than Sketch solves `h_dedup`, a significant speedup.

Moreover, `sort_mock` encodes the specification of `sort`, so once Sketcham solves `h_dedup''`, it has found a solution for `h_dedup` as well. To preserve correctness, Sketcham keeps the original harnesses such as `h_dedup`, because mocks with partial specifications can lead to partially incorrect solutions to the harnesses using them. However, even in these cases, the counterexamples they generate can still help more quickly narrow the synthesis search space for the original harness, and lead to an ultimately valid solution.

Quantifier Elimination. In Figure 1d, the translation from harness to mock was straightforward: the call to the mocked function becomes a call to an uninterpreted function, and `assert`s become `assume`s. Sometimes, however, the translation is more complex. Consider the sketch in Figure 2a, which includes a function `doub` that doubles its input and a harness `h` that calls `doub(n*10)` and asserts the result is `(n+n)*??` for some hole.

Notice this assertion only describes arguments of the form `n*10` for some `n`, i.e., implicitly there must exist some `m` such that `m = n*10` for the assertion to hold. Sketcham performs *quantifier elimination* [1,4] on such nested existentials, following the approach of Kuncak et al. [17]. Figure 2b shows the resulting mock.

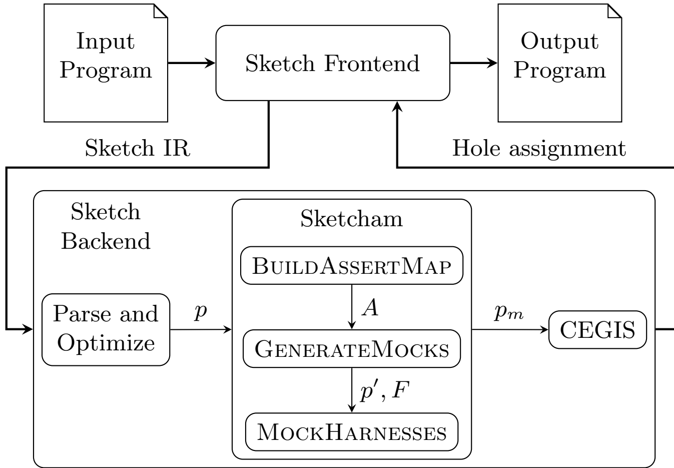


Fig. 3: Sketcham architecture

Here, in the assumption, `n` is replaced by witness candidate `m/10`. Because `m` is an integer, we also add a precondition that `m` is evenly divisible by 10.

We note that Sketcham includes quantifier elimination for completeness, and in our evaluation we consider the sketch in Figure 2a. However, we did not find quantifier elimination necessary for our other benchmarks.

3 The Sketcham Algorithm

Next we more formally describe Sketcham, which is implemented as a pass within Sketch as shown in Figure 3. The presentation that follows reflects this Sketch implementation without loss of generality of the core algorithm for converting tests to mocks. The Sketch *frontend* consumes the input sketch and transforms it into the Sketch intermediate representation (Sketch IR), which is passed to the Sketch *backend*. Sketch IR encodes first-order logic augmented with theories of arithmetic, arrays, functions, and more, as discussed below. When the backend loads the IR, it performs loop unrolling, function inlining, and other transformations that are needed by the solver [26], yielding a program p. Standard Sketch then uses CEGIS to solve the synthesis problem, outputting a hole assignment that the frontend uses to produce the solved sketch. Sketcham modifies this process by inserting, after optimization, a *mock rewriting* phase, described below, that transforms p into the augmented program p_m for CEGIS.

We formalize Sketcham on the fragment of Sketch IR shown in Figure 4. Here types are omitted, and we assume the sketch is type-correct. A program sketch p is a sequence of harness and function definitions. A harness definition h tags a function definition as a test harness. A function definition d is given named parameters[1] and a body, which is a sequence of statements. Statements s are

[1] For simplicity, we assume parameter names are unique across the whole program.

$$
\begin{array}{lll}
p & ::= & (h \mid d)* \\
h & ::= & \textsf{harness } d \\
d & ::= & \textsf{def } f \texttt{ (} x \texttt{ , } \ldots \texttt{ , } x \texttt{) \{ } s* \texttt{ \}} \\
s & ::= & x \texttt{ := } e \mid \textsf{return } e \mid \textsf{assert } \phi \mid \textsf{assume } \phi \\
e, \phi, \psi & ::= & f \texttt{ (} e \texttt{ , } \ldots \texttt{ , } e \texttt{) } \mid uop\ e \mid e\ bop\ e \mid n \mid x \mid \texttt{??}\ x \\
uop & ::= & \neg \mid \texttt{-} \\
bop & ::= & \wedge \mid \vee \mid \oplus \mid \implies \mid = \mid + \mid \texttt{-} \mid \texttt{*} \mid / \mid \%
\end{array}
$$

$x, y \in$ variable names	$G \in$ graphs	$A : f \rightarrow \Phi$
$f, g \in$ function names	$\Phi \in$ set of ϕ	$F : f \rightarrow f$

Fig. 4: Sketcham's fragment of Sketch IR

assignments, returns, assertions, and assumptions. The most critical expressions e in our algorithm are function calls f (e , ..., e) with their arguments. The detailed grammar for the remaining expressions is unimportant in the remainder of this section, but for completeness we show expressions for unary and binary logical and arithmetic operations $uop\ e$ and $e\ bop\ e$; constants n; variables x; and named holes ?? x. Below, we sometimes use the metavariables ϕ and ψ in place of e to indicate an expression used for Boolean-valued formulas.

Given the input Sketch IR program p as shown in Figure 3, Sketcham creates the output sketch by first calling BuildAssertMap (Algorithm 1) to build mapping A from function names to assertions from tests of those functions. Next, GenerateMocks (Algorithm 2) uses A to construct mocks for functions in the domain of A, yielding program p', which includes the original sketch p plus those mocks. This step also returns a mapping F from the original function names to the corresponding mock names. Finally, MockHarnesses (Algorithm 3) creates the output sketch p_m, which augments p' with copies of the original sketch's harnesses, except the copies call the mocks instead of the original functions.

Critically, during this last step, holes are *not* renamed when the harnesses are copied. Moreover, the newly generated harnesses are prepended to the sketch. Thus, when CEGIS tries solving each harness in p_m in order, it will first find solutions that are consistent with the mocks. Then when it reaches the original harnesses (which must remain in case there is information in them not captured by the mocks—see discussion of GenerateMocks below), CEGIS can use the information it already derived from the mocks to find the ultimate solution to the original problem.

In the remainder of this section, we describe each step of the algorithm in detail. Below, we capitalize the names of sets of a given metavariable (e.g., Φ is a set of formulas ϕ, etc.), and we use vector notation to indicate arrays (e.g., $\vec{s}$ is an array of statements s).

Building the assertion mapping. Each mock expresses the specification of an original function as it is encoded by that function's tests. To start, Sketcham collects assertions from those tests into an assertion mapping. Algorithm 1 builds

Algorithm 1 Mock rewriting: building the assertion map

Input: p - the sketch
Output: A - finite map of function names to sets of assert formulas

```
 1: function BuildAssertMap(p)
 2:     A ← ∅
 3:     Φ ← {φ | assert φ ∈ p}                    ▷ all solver-reachable asserts in p
 4:     Φ0 ← {φ ∈ Φ | 0 = |f(...) ∈ φ|}           ▷ asserts with 0 function calls
 5:     Φ1 ← {φ ∈ Φ | 1 = |f(...) ∈ φ|}           ▷ asserts with 1 function call
 6:     for all f ∈ Φ1 do
 7:         Φf ← Φ0 ∪ {φ ∈ Φ1 | f ∈ φ}            ▷ asserts with 0 calls, or 1 call to f
 8:         Ψ ← Φ \ Φf
 9:         while Ψ ≠ ∅ do
10:             X ← FV(Ψ)                          ▷ inputs and holes free in Ψ
11:             Ψ ← {φ ∈ Φf | X ∩ FV(φ) ≠ ∅}
12:             Φf ← Φf \ Ψ
13:         end while
14:         A[f] ← Φf
15:     end for
16: end function
```

the assertion mapping A from the input sketch p. The algorithm begins by initializing A to empty and Φ to the set of all assertions from all tests in p. It then selects two subsets of Φ. The set Φ_0 contains all assertions that do not include calls to any functions, and the set Φ_1 contains all assertions that include exactly one function call. We exclude assertions with multiple function calls so that mocks are standalone, to conform to the technical requirements Sketch imposes on models. As a consequence, we exclude some terms that present no such concerns (e.g., conjunctions of otherwise unrelated terms), as translating them to assumptions may be much more complex or even impossible. We leave extending BuildAssertMap to more assertion patterns to future work.

For each function f called in an assertion in Φ_1, on line 7 we next compute the set Φ_f from Φ_0 (the assertions that hold throughout each test, including at calls to f) and the subset of Φ_1 that refers to f. For example, consider the assertion in `h_sort` in Figure 1d. This code refers to the result of calling `sort(n, vs)`, so $\Phi_1 = \{\phi_i(\texttt{sort(n,vs)})\}$, where the ϕ_is capture the assertions in `h_sort`. Additionally, if we picked, say, a loop unrolling bound of 4, then Sketch would implicitly `assert n<4`, resulting in $\Phi_0 = \{\texttt{n<4}\}$. In general, Φ_0 might contain additional assertions that are irrelevant to the calls in Φ_1. For example, loop unrolling for harness `h_dedup` (not shown) might add another bound `m<4` to Φ_0 for `sort`. However, such irrelevant assertions will not change the resulting mock.

In some cases, we cannot add assertions in Φ_f to A because other assertions on the same variables interfere. For example, suppose the sketch includes `assert f(x)` and `assert g(x)`. Then Φ_f might not completely characterize `f`—the assertion in Φ_f is valid only if `assert g(x)` also holds, which puts an unknown (until the full sketch is solved) constraint on `x`. Thus, in this case, our algorithm discards the assertions in Φ_f. More specifically, on line 9, the loop

Algorithm 2 Mock rewriting: generate mocks

Input: p - the sketch
Input: A - output of Algorithm 1
Output: p' - the sketch augmented with mock definitions
Output: F - finite mapping from an original function name to its mock

```
1: function GenerateMocks(p, A)
2:   F ← ∅, p' ← p
3:   for all f ↦ Φ ∈ A do
4:     def f(x⃗){...} ← the definition of f in p
5:     f_u ← FreshName(f)
6:     s⃗ ← [ ]
7:     Φ_0 ← {φ ∈ Φ | 0 = |f(...) ∈ φ|}
8:     Φ_1 ← {φ ∈ Φ | 1 = |f(...) ∈ φ|}
9:     for all φ ∈ Φ_1 do                        ▷ convert asserts into assumes
10:      f(e⃗) ← the lone function call in φ
11:      φ_u ← φ[f(e⃗) := f_u(x⃗)]                ▷ substitute uninterpreted function
12:      Ψ ← {x_i = e_i | 0 ≤ i < |x⃗|}           ▷ equate parameters to arguments
13:      φ' ← (⋀Φ_0) ∧ (⋀Ψ) ⟹ φ_u                ▷ the condition where φ holds
14:      φ'' ← ⟦FV(φ); Ψ ⊢ φ'⟧
15:      s⃗.append(assume φ'')
16:    end for
17:    f_m ← FreshName(f)
18:    F[f] ← f_m
19:    d_m ← def f_m(x⃗){                         ▷ create the mock definition
               s⃗
               return f_u(x⃗)
             }
20:    p'.insert(d_m)
21:  end for
22: end function
```

removes any $\phi \in \Phi_f$ whose free variables overlap with free variables outside of Φ_f. The process iterates in case free variable dependencies cascade. For example, the existence of `assert g(x)` would eliminate `assert f(x-y)`, which would in turn eliminate `assert f(y)`. The result is the transitive closure of the allowable assertions about each function.

Generate mocks. Next, Algorithm 2 iterates through each function in the domain of A, generating a corresponding mock to add to the augmented sketch p'. As it does so, it also builds a map F from function names to the names of the generated mocks.

For each $f \mapsto \Phi \in A$, GenerateMocks begins by finding the definition of f and creating a corresponding freshly named uninterpreted function f_u. It then initializes $\vec{s}$, the assumptions to be inserted into the new mock body, to empty. Then, from each asserted formula $\phi \in \Phi$, the algorithm creates a formula ϕ_u by substituting the single function call $f(\vec{e})$ in ϕ with a call $f_u(\vec{x})$, where $\vec{x}$ are the formal parameters of f (line 11). Notice this call to f_u is the same no matter the

Algorithm 3 Mock rewriting: mock harnesses

Input: p' - the sketch from Algorithm 2
Input: F_1 - the name map from Algorithm 2
Output: p_m - the sketch augmented with mock harnesses

```
 1: function MockHarnesses(p′, F)
 2:     G ← CallGraph(p′), p_m ← p′
 3:     for i ← 1, maximum mock call graph depth do
 4:         F_{i+1} ← ∅
 5:         for all def g(y⃗){s⃗} ∈ Callers(G, dom F_i) do    ▷ similarly, harness def
 6:             g′ ← FreshName(g)
 7:             d′ ← def g′(y⃗){                              ▷ respectively, harness def
                        {s[f := f′ | f ↦ f′ ∈ F_i] | s ∈ s⃗}
                    }
 8:             p_m.insert(d′, before g)
 9:             F_{i+1}[g] ← g′
10:         end for
11:     end for
12: end function
```

original call to f, which ensures the generated mock conforms to the technical requirements Sketch imposes on models. To encode the actual information at the call site, we next add a precondition. The algorithm constructs ϕ' (line 13), which is an implication denoting that ϕ_u holds if the ancillary asserts Φ_0, and the equalities $x_i = e_i$ from the call to f hold. One nuance we elide here is that Sketch augments all function calls with an additional explicit *path condition* parameter that captures conditional branches taken up to the point of the call, which makes it easier for Sketch to translate the IR into a SAT formula. For soundness, we include this path condition as a premise of ϕ' and assign f_u the path condition $\top$. Note that our implementation trims Φ_0 before adding it to ϕ' to the subset containing only the variables in $\vec{e}$.

Next, the algorithm performs quantifier elimination on ϕ', yielding ϕ'' (line 14). More precisely, $[\![\mathrm{FV}(\phi); \Psi \vdash \phi']\!]$ eliminates variables in $\mathrm{FV}(\phi)$ from ϕ', searching for witnesses in Ψ. Then, ϕ'' is added to $\vec{s}$ as an `assume`, and the loop continues until all mappings for f have been handled.

Finally, on lines 17-19 the algorithm computes a fresh Sketch name f_m for f, adds a mapping to F, and creates function definition d_m for f_m. The function f_m takes the same arguments as f, assumes all formulas in $\vec{s}$, and returns f_u on f_m's arguments. Thus, when f_m is called, the assertions about f from its original test suite in p are assumed on f_m's arguments, as we saw in Section 2. The definition d_m is added to p', and mock generation continues until all mappings in A have been traversed.

New mock harnesses. The last step of Sketcham adds calls to the mocks generated by GenerateMocks. One naïve approach would be to simply replace each call to f with a call to f_m for all $f \mapsto f_m \in F$. However, this will not work for two reasons. First, we need a full solution for the holes in all functions, including

those that are mocked. Replacing calls to f with calls to f_m would remove many constraints on the holes in f, underconstraining their solutions. Second, as we saw earlier the template for f might contain additional information excluded by BuildAssertMap, so replacing f by f_m might underconstrain f's callers.

Our solution is to create an output sketch that includes both the original sketch—including all calls to f in their original form—and duplicate sketch code that calls f_m in place of f. The duplicated code refers to the same holes as the original sketch. Hence, information derived from the duplicated code can potentially greatly speed up solving of the original code.

Algorithm 3 shows MockHarnesses, which creates this duplicate code. The algorithm begins by constructing a call graph G from the sketch p' from the previous step. Note that none of the mocks in p' are called yet, so the call graph is the same as for the original sketch. Next, the algorithm duplicates the sketch one level of the call stack at a time, starting at the mocks and working up toward the harnesses. To limit duplication, e.g., for mocks called by recursive functions whose duplication would loop infinitely, the algorithm bounds the duplication depth. For each level i, it iterates through all functions $g \in \textsc{Callers}(G, \operatorname{dom} F_i)$, meaning functions g that call a function in the domain of F_i. It duplicates each such g, replacing calls to functions $f \in \operatorname{dom} F_i$ with calls to $F_i[f]$, and then adds the duplicated function to the sketch. Since g has now been renamed, $g \mapsto g'$ is added to a new mapping F_{i+1}, and calls to it are duplicated in the next iteration, repeating until reaching the root of the call graph or the maximum duplication depth. Note the process is the same for both regular function definitions and for functions that are harnesses.

For example, suppose harness h calls function g, which in turn calls function f, and assume GenerateMocks created f_m and g_m. Then in the first iteration, MockHarnesses creates a duplicate h' that calls g_m and a duplicate g' that calls f_m. In the next and final iteration, it creates a duplicate h'' that calls g'.

When we insert the duplicate functions, we insert them *before* the original functions. This ensures that when we insert the duplicate harnesses that call the mocks, Sketch will solve those harnesses before solving the original ones.

4 Evaluation

We evaluated Sketcham on ten benchmarks, running each from 11 to 1487 times until reaching statistically significant results. We found that, for six of ten benchmarks, Sketcham performs up to 5$\times$ faster than Sketch, for one benchmark Sketcham is slower by a factor of up to 0.9$\times$, and for the remaining three benchmarks performance is indistinguishable. We examined the benchmark *dedup* (Figure 1) in depth and found that, as suspected, overall performance improvement is due to improved synthesis time when using `sort_mock`.

Implementation. Sketcham comprises approximately 1075 lines of C++ code within the Sketch backend. The user enables Sketcham with `-mock` and specifies the max mock duplication depth via `--bnd-mock-depth`, which defaults to 3.

Because they clone and then rearrange the input Sketch IR program, the run time of Algorithms 1-3 is approximately linear in the number of functions and the number of asserts in the sketch. Our implementation covers the features given as part of the Sketch IR fragment in Figure 4, with the modification that we explicitly depict assignment, which Sketch IR does not require because it structurally hashes expressions to yield a compact in-memory representation [26]. We also note that Sketch includes additional features that we leave to future work, such as complex harness types, and that quantifier elimination is currently restricted to arithmetic expressions.

Benchmarks. We used the following benchmarks:

- *double*, the integer doubling program given in Figure 2.
- *absval*, the absolute value function.
- *fib*, the linear-time Fibonacci function. The specification requires its output to be equivalent to the exponential time algorithm.
- *datetime*, a simplified implementation of the C `strptime` function. This function accepts a format that it uses to parse a date/time string.
- *boyerMoore*, which implements the Boyer Moore string search algorithm [3].
- *regex*, a regular expression matching engine and compiler.
- *spellcheck*, a program that suggests a corrected version of its input using the Levenshtein edit distance from entries in a dictionary.
- *minpair*, uses edit distance to find the closest pair out of an array of values.
- $dedup_m$, deduplication with merge sort from Figure 1, and $dedup_i$, deduplication with insertion sort.

Sketch has a multitude of configuration options that can have a large effect on performance. The middle portion of Table 1 gives values for the four options that differ across the benchmarks: *int type*, whether Sketch uses symbolic integers (in either a bit-vector encoding or a sparse encoding [26]) or native integers [28]; *int bits*, the number of bits per integer; *loop unroll*, the maximum loop unrolling depth; and *func inline*, the maximum depth of function call inlining.

We selected values for these options that reflected each benchmark's design and demonstrated pronounced run time differences from Sketch to Sketcham, as follows. *double* and *absval* use Sketch's defaults. *fib* tests recursively computing the Fibonacci sequence up to the tenth entry, so function call inlining is set accordingly. *regex* is required to reject bad matches, which requires higher unrolling and inlining. *datetime*, *boyerMoore*, *spellcheck*, and *minpair* need higher loop unrolling to iterate over long strings. These last three and both *dedup*s also do much better using native integers. The *dedup*s also run unreasonably slowly with more bits or higher unroll, so we reduced the amount of unrolling. In all our benchmarks, any configuration options not discussed here were left as their defaults, including the mock duplication depth, with the default of 3.

Methodology. All measurements were taken on a 3.2 GHz AMD Ryzen 5 1600 system with 32GB of RAM. We found that while most benchmarks consistently

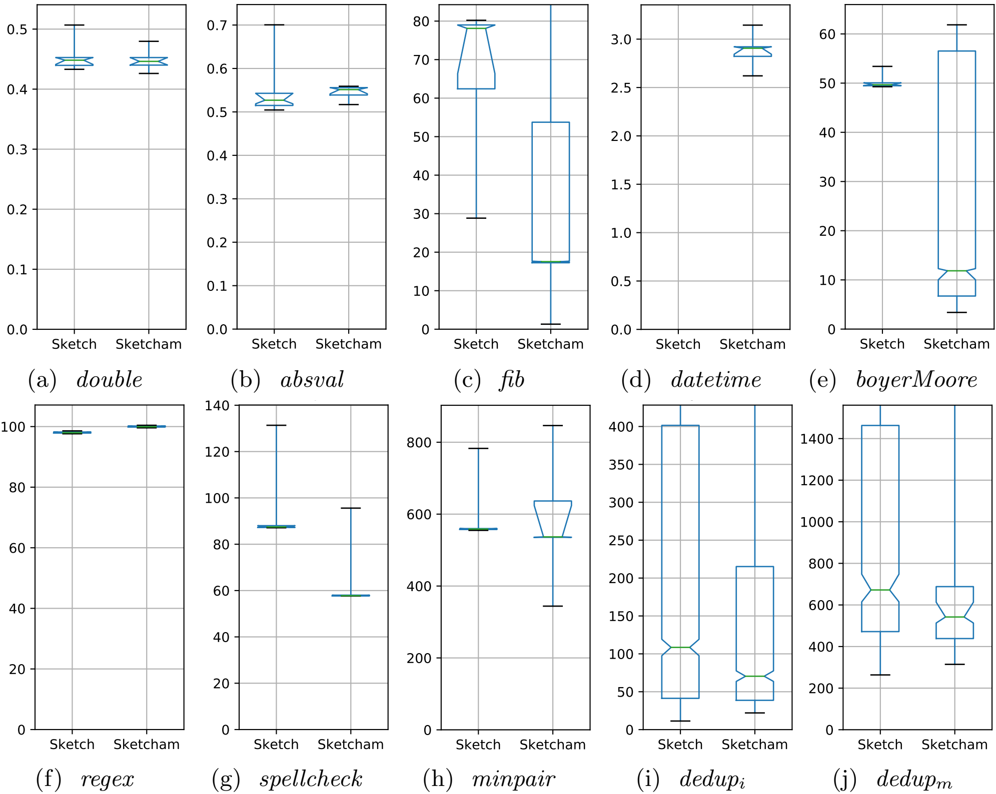
	# lines	# holes	int type	int bits	loop unroll	func inline	Sketch runs total	Sketch runs failed	Sketcham runs total	Sketcham runs failed
double	8	1	symbolic	5	8	5	17	0	17	0
absval	69	9	symbolic	5	8	5	17	0	17	0
fib	46	4	symbolic	6	8	10	20	0	65	0
datetime	177	3	symbolic	11	20	5	11	11	17	0
boyerMoore	136	16	native	7	13	5	17	0	153	19
regex	357	5604	symbolic	5	30	7	17	0	17	0
spellcheck	94	5	native	5	9	5	17	0	17	0
minpair	113	3	native	5	10	5	17	0	22	2
$dedup_i$	73	1134	native	2	4	5	1487	88	762	23
$dedup_m$	80	9008	native	2	4	5	648	281	88	16

Table 1: Benchmark config options and characteristics.

perform within half an order of magnitude under both Sketch and Sketcham, in a few cases synthesis time varies by as much as two and a half orders of magnitude. To account for this variance during our evaluation, we repeatedly ran each benchmark until achieving statistical significance, between 11 and 1487 times, as listed in the rightmost portion of Table 1. Each run was executed with the system otherwise almost totally idle to minimize interference. While most runs completed successfully, we exclude those that exceed a 60 minute timeout or fail to synthesize due to exhausting system memory or a crash within Sketch. To give an idea of the problem size, the leftmost portion of Table 1 lists the numbers of lines and holes per benchmark.

As other work has observed [9], performance evaluation methodologies that lack rigor can lead to misleading and incorrect conclusions. To avoid this problem, we collect enough data to calculate a percentile's confidence interval (CI) at a given confidence level (CL). We employ the classic Clopper-Pearson [6] (or "exact") method using the probabilities of the Binomial distribution to iteratively calculate confidence intervals for a given dataset. While other methods are often used, many of these assume an underlying Gaussian distribution. The underlying distributions for our measurements are not known and do not appear to be Gaussian, a case the exact method handles correctly.

Run time variance is not correlated across configurations, so the number of runs needed for significance can differ from Sketch to Sketcham, as reflected in the "total" columns of Table 1. We ran each configuration repeatedly until measurements met two statistical significance conditions. First, that they reach a 95% CL that the population median lies within at most a 20% CI around the sample median. For example, for a sample median of 100 s, the population median might lie between 90 s and 110 s, or between 98 s and 118 s, depending on the underlying distribution. Second, the CI must range entirely between the first and third quartiles to increase the confidence that the median measurements adequately reflect the underlying distribution. In seven out of ten benchmarks these two conditions were sufficient to yield CIs that did not overlap across Sketch



(a) *double* (b) *absval* (c) *fib* (d) *datetime* (e) *boyerMoore*

(f) *regex* (g) *spellcheck* (h) *minpair* (i) $dedup_i$ (j) $dedup_m$

Fig. 5: Total time (s). Times are drawn as notched box plots, which give the distribution's median inside a notch indicating its confidence interval. As usual, the box extends to the first and third quartiles, and whiskers extend to the full distribution. To better focus on the data, we truncate some whiskers. Note differing y-axis scales both here and below.

and Sketcham, which allows for statistically significant performance claims about these benchmarks.

4.1 Performance

Figure 5 shows the running times of Sketch and Sketcham on our benchmarks. The distribution of times is shown as notched box plots. The boxes extend from the first to the third quartile, with the median shown as a mid-line. The CI is indicated by the notch. The whiskers extended to the minimum and maximum values (some whiskers are truncated to allow for a closer view of the median).

Following standard practice, we conclude that two configurations have a statistically significant difference in performance if their CIs do not overlap, as there is then high probability that the median times of the distributions are different.

We see that for six of the ten benchmarks, Sketcham is faster than Sketch, while one is marginally slower and three display no significant performance change. We investigated each benchmark's performance in detail, discussed next. The performance differences we report are ratios of the run time of Sketch to Sketcham for a given benchmark. Due to uncertainty we report speedup ranges for the median, comparing the opposite extents of each CI. This ranges from, at minimum, the ratio of the faster end of Sketch's CI to the slower end of Sketcham's CI, up to, at maximum, the ratio of the slower end of Sketch's CI to the faster end of Sketcham's CI.

The times shown are total run time, which can be broken down into synthesis, verification, and overhead time. For Sketcham, overhead can further be broken down into mock construction and normal Sketch overhead. The total runtime overhead of mock construction is less than 0.4% for all benchmarks except regex (3%) and both dedups (∼20%). In most cases, this time was dominated by the GenerateMocks and BuildAssertMap phases.

The *double* benchmark's performance is approximately the same in both cases. In fact, the CIs overlap almost completely, suggesting the performance may be dominated by constant factors in Sketch.

The *absval* benchmark is also approximately the same. It is another simple program that Sketch solves very quickly, and as such the mocks only add to the verification time.

The *fib* benchmark asserts that, on integers 0 to 9, the to-be-synthesized linear-time Fibonacci implementation returns the same result as an exponential-time implementation. In Sketch, the calls to the exponential-time algorithm cause a slowdown. But since Sketcham replaces calls to the exponential-time algorithm with calls to a (constant-time) mock, Sketcham achieves a speedup of 3.8–4.5×. While it is difficult to make out in the plot, the median and CI lie immediately above the first quartile for Sketcham.

The *datetime* benchmark fails to synthesize in Sketch due to memory exhaustion, but it consistently synthesizes in just a few seconds using Sketcham. Investigating further, we found the bottleneck is a function that parses strings into integers in a loop that converts digits and adds them to a running total. For example, the digit sequence `abc` is converted to the integer `100*a+10*b+c`. This conversion loop is unrolled to the maximum bound by Sketch, and the input strings are of varying sizes, which is encoded as a separate formula for each possible length. The SAT conversion algorithm translates symbolic arithmetic formulas according to combinations of possible values of their subformulas, which results in very large SAT formulas in this case. Later in the conversion, these are merged back together in another quadratic operation. Due to the number of formulas and overall formula size, this eventually exhausts memory. While Sketcham technically faces the same issue, it does so after decomposing the sketch into smaller formulas, and thus these limits are never approached.

The *boyerMoore* benchmark runs 4–5× faster under Sketcham than Sketch. The reason is similar to the previous case. *boyerMoore* includes a generator that constructs arithmetic expressions that add and subtract a small set of values

including a hole. Sketch constructs these expressions recursively so they grow quickly, with the total number of terms determined exponentially by the degree of function inlining, and the resulting expressions have high symmetry, both factors that slow down solving, further compounded by the location of this expression deep within the sketch. Because Sketcham breaks the problem's dependencies, this expression can be synthesized separately from the rest of the program, which proceeds much more quickly.

The *regex* benchmark's overall performance using Sketcham is statistically significantly slower by a factor of 0.98×, which is a minimal difference in practice. The main mocked function here performs compilation of a regular expression into instructions for a virtual machine. Because compilation is recursive, it is difficult to give a specification that Sketcham can use. It is instead given by example with an exhaustive set of subproblems, which greatly increases the number of harnesses to solve. While most harnesses keep similar performance and the slowest harness is 8% faster in Sketcham, this is not enough margin to improve overall solve time.

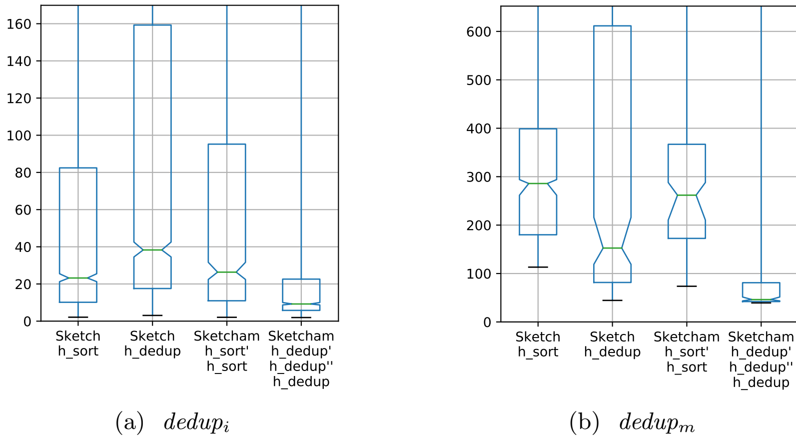
The *spellcheck* benchmark using Sketcham sees a speedup of 1.5×, while *minpair* performs roughly the same (0.89–1.04×). Both rely on the same Levenshtein edit distance algorithm. The harness for this algorithm, which is the most time-consuming in either sketch, runs last in both settings, which reveals the source of the performance difference between the two benchmarks. *minpair* is dominated by synthesis time and *spellcheck* by verification time, which means that harnesses for the minimum pair function are more difficult to synthesize than for the spellcheck function, and so the former accumulates more state within the solver that is compounded when solving the Levenshtein harness. This slows it down enough to decrease the overall performance. On the other hand, the improvement of *spellcheck* is distributed across all individual harnesses, and across both synthesis and verification time, more than making up for the time it takes to construct and solve the mock harnesses.

Finally, the *dedup*s show a notable performance improvement with Sketcham. In both $dedup_i$ and $dedup_m$, the problem is large and complex enough that plain Sketch struggles with it. Sketcham eliminates the interactions of holes across the deduplication and sorting functions, which speeds up synthesis by a factor of 1.3–1.9× for $dedup_i$ and 1.003–1.5× for $dedup_m$.

4.2 Case Study: Deduplication

Next, we examine the performance of $dedup_i$ and $dedup_m$ in detail, as they illustrate the strengths and weaknesses of Sketcham. We break our discussion into comparisons of solving time across harnesses and comparisons of CEGIS synthesis time to CEGIS verification time.

Time to Solve Each Harness. Both $dedup_i$ and $dedup_m$ are structured the same way, and Sketcham creates the harnesses and mocks shown in Figure 1c for both. Figure 6 breaks down the total times for $dedup_i$ and $dedup_m$, grouped by the



(a) $dedup_i$ (b) $dedup_m$

Fig. 6: Harness time (s)

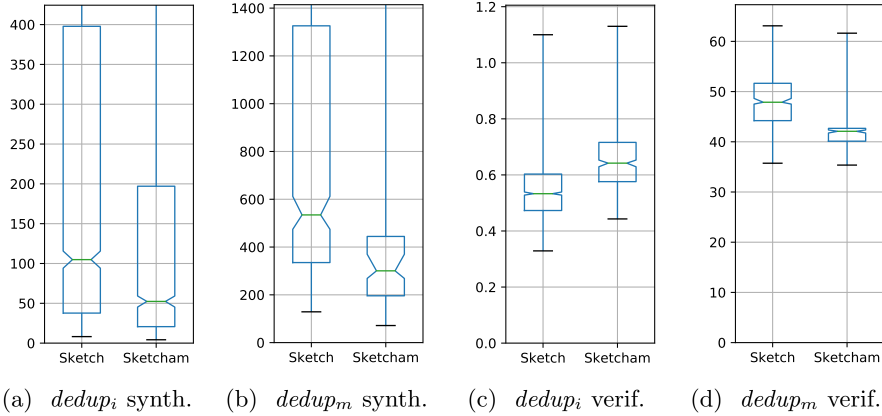
harnesses for `sort` and for `dedup`. We exclude overheads such as time spent in mock construction, parsing the input, and reassembling the output.

We make several observations. First, comparing the first and third columns within each subfigure, we see the time for solving `h_sort` is the same for Sketch and Sketcham. This makes sense because `h_sort'` adds no information—it calls mocked `sort` and then immediately asserts the same specification as in the mock. Note that, while the trivial `h_sort'` harness could be elided here, creating an analogous harness would be useful if the harness accidentally contained a contradiction. In such a case, Sketcham would almost instantly decide the harness is unsatisfiable, whereas Sketch could spend an arbitrary amount of time reasoning about the computation in the actual called function before detecting the contradiction.

Second, comparing the second and fourth columns within each subfigure, we see that the CI of `h_dedup` using Sketcham lies well below the CI using Sketch. The speed improves by a factor of 3.2–4.7× for $dedup_i$ and 2.2–4.9× for $dedup_m$. Examining this result in detail, we find that Sketcham works exactly as intended: `h_dedup''` calls the mocked `sort`, enabling it to synthesize quickly and assign holes correctly, which are then simply verified when checking `h_dedup` (and `h_dedup'` is trivial, similarly to `h_sort'`).

Third, also comparing the second and fourth columns, we see the variance in performance for Sketch is much greater than for Sketcham. Investigating further, we found this occurs for two reasons. First, the specification in `h_sort` is weak enough[2] that sometimes an incorrect hole assignment for `sort` satisfies the verifier and is only discovered while synthesizing `h_dedup`, forcing the solver to backtrack at great cost and simultaneously consider the holes in both functions. Second, even when the solver finds a correct assignment for `sort`, it includes the

[2] In addition to the specification we have supplied, a complete specification of `sort` relies on the existence of a permutation function over the array's indices.



(a) $dedup_i$ synth. (b) $dedup_m$ synth. (c) $dedup_i$ verif. (d) $dedup_m$ verif.

Fig. 7: Synthesis and verification time (s)

entire formula again while solving `h_dedup`, resulting in a much larger problem and corresponding variability. In contrast, with Sketcham, `h_dedup''` is decoupled from `sort`, eliminating these issues.

Fourth, we observe that both Sketch and Sketcham can solve `h_sort` about 10× faster for $dedup_i$ than for $dedup_m$. Overall, merge sort is more challenging for Sketch than insertion sort (note that since Sketch finitizes the problem by, e.g., unrolling loops, asymptotic complexity does not play a role). More surprisingly, synthesizing `h_dedup` is also faster for $dedup_i$ compared to $dedup_m$. We believe this occurs because synthesis of `h_dedup` must sometimes recover from a bad hole assignment from `h_sort`, which will be quicker for $dedup_i$, and because the easier synthesis of $dedup_i$ means the solver accumulates less state, such as conflict clauses, that would otherwise slow down solving subsequent harnesses.

Finally, we begin to get a clearer picture of the divergence between $dedup_i$ and $dedup_m$. In $dedup_i$, `h_dedup` synthesis is the performance driver, and the improvement using Sketcham has a significant impact on total performance improvement. In $dedup_m$ it is overshadowed by `h_sort`, which dominates to the point that improvement elsewhere is not as significant a contributor. Combined with the overhead of mock construction, this leads to a less pronounced improvement in total performance.

Synthesis and Verification Time. Figure 7 shows the times for the CEGIS synthesis phase and verification phase for each benchmark under Sketch and Sketcham. Not shown are the overheads of mock construction, parsing, etc., which for $dedup_i$ we found took 3–4s in Sketch versus 17–19s in Sketcham, and for $dedup_m$ took 90–96s in Sketch versus 201–207s using Sketcham. We believe much of the difference between these could be eliminated with additional engineering effort.

Looking at verification times in Figures 7c and 7d, we see that while the verification times for Sketch and Sketcham are different, they are still relatively close: Sketcham is 0.81–0.86× slower for $dedup_i$ and 1.12–1.16× faster for $dedup_m$. In

contrast, comparing synthesis times in Figures 7a and 7b, we see a more significant speedup for Sketcham over Sketch: 1.59–2.55× for $dedup_i$ and 1.28–2.28× for $dedup_m$. Moreover, if we compare synthesis and verification time, we see that the overall solving time for both benchmarks is dominated by synthesis time. Indeed, we observed even greater synthesis speedups on other benchmarks including *fib* (4.2–5.1×) and *boyerMoore* 5.2–6.9×, but the most extreme of which was *spellcheck*, which saw synthesis speed up by 308.4–345.7× using Sketcham. Thus, we find that Sketcham's performance improvements come from reducing synthesis time by introducing mocks that decrease the number of holes that need to be considered at once.

4.3 Discussion

In general, we found that Sketch's performance is unpredictable in practice, which is influenced by factors such as the solver's random seed. For example, in terms of overall solving time, our experimental runs included several outliers (not shown in Figure 5) near the 60 minute timeout. In these cases, Sketch essentially makes a very poor initial guess for the holes, and verification produces counterexamples that do not add much information. Both Sketch and Sketcham exhibit this issue.

Moreover, often what seem like minor changes in the program sketch or configuration options can result in totally different solver behavior, and hence performance. One example of this was *boyerMoore*, which turned out to be non-linearly sensitive to the loop unrolling parameter. This benchmark was also extremely fickle about the problem formulation—holes in what seemed to be innocuous locations would lead to timeouts in both Sketch and Sketcham. Another example is *dedup*, which initially had a specification that omitted a requirement that the output array did not have a negative length. Without this constraint, the performance benefit of Sketcham was overwhelmed by the variability of the solver exploring ultimately impossible scenarios.

Overall, our results suggest that while Sketcham can't always outperform plain Sketch, it performs best on problems split into functions whose tests cover the behavior the sketch actually relies on while being easier to compute than the functions' actual implementations. While Sketcham affected the performance of both CEGIS phases, the best improvements were observed when the solving time of dependencies was dominated by the synthesis phase. For programs with these properties, Sketcham can exhibit a performance improvement of as much as 5× overall, with synthesis time improvements alone of up to 345.7×. Moreover, in some cases, such as *datetime*, Sketcham can solve problems that are out of reach of plain Sketch. For programs where these properties do not hold, Sketcham performance is typically similar to plain Sketch.

5 Related Work

There are several threads of related work.

Program Synthesis with models. As discussed earlier, our work builds on work by Singh et al. [24], who propose manually created models for Sketch. While Sketcham relies on the core algorithm of that work, Sketcham frees the Sketch user from needing to write models, because we create mocks automatically from normal sketches. Mariano et al. [18] use algebraic specifications to model libraries. In contrast, our approach derives specifications from the input program's assertions, without requiring the programmer to add annotations.

Deriving mocks and specs from tests. Saff et al. [21] use the capture and replay of actual test executions to automatically generate mock dependencies with the goal of speeding up test execution. Fazzini et al. [8] further generalize this capture-and-replay technique to consistently model the environment of a mobile app under test, allowing for testing apps that use an inconsistent resource like a database or network device. Both of these target normal testing rather than synthesis. Nguyen et al. [19] leverage symbolic execution over input-output test pairs to perform program repair. However, they use these tests to model individual expressions instead of modeling entire functions. The insight underlying these approaches is similar to ours, however Sketcham is capable of both input-output pairs and general properties, and does not rely on either concrete or symbolic execution of tests.

Component-based synthesis. Gulwani et al. [10] model programs using logical input-output relations to synthesize loop-free bit-vector programs. Shi et al. [23] combine many solutions that each only partially meet a specification into one that meets the entire specification. Both approaches limit the synthesis search space by building their solutions from the bottom up, from a selection of base components. Smith and Albarghouthi [25] prune the search space using bottom up algebraic rewriting of the program into an equivalent normal form. In contrast to these, Sketcham derives its benefits from breaking apart input sketches from the top down, at function level granularity.

Modular synthesis using symbolic or actual execution. Samak et al. [22] derive specifications of class methods using symbolic execution and use them to synthesize a replacement shim class one method at a time. Van Geffen et al. [31] use symbolic execution to model abstract virtual machines to modularly synthesize a compiler one instruction at a time. In contrast, because our approach derives mocks directly from the input's assertions, we need not consider the code itself when modeling it. Hua et al. [11] modularize the synthesis of library calls through execution of actual partial programs. In contrast, we attempt to avoid called functions entirely by relying on their inferred specifications.

Other approaches. Bodík et al. [2] finalize incomplete programs using angelic nondeterminism. In contrast, Sketcham does not introduce arbitrary angelic values, but instead constrains any angelic-like behavior using a function's inferred specification. Huang et al. [12] use a divide-and-conquer strategy to iteratively split synthesis problems according to heuristics. In contrast, Sketcham splits problems structurally in a single pass. Polikarpova et al. [20] speed up synthesis through modular verification using refinement types. In contrast, our approach achieves a similar kind of modularity without being type-directed.

6 Conclusion

This paper presents Sketcham, a new technique for decomposing program sketches during synthesis by turning a function's test suite into a mock that a caller can invoke in place of that function, thereby allowing separate reasoning about callers and callees. Sketcham gathers asserts from tests into a specification for each function which it embodies as a Sketch model. We implemented Sketcham as an additional pass with Sketch and evaluated it on a set of ten benchmarks. Our rigorous evaluation strategy ensured at a confidence level of 95% that our measurements demonstrate performance gains of as much as 5×, including one benchmark that otherwise timed out on Sketch. Based on these results, we believe that automatically generating mocks from tests with Sketcham is a promising new approach for achieving modular synthesis.

Acknowledgments

We would like to thank Norman Ramsey, Milod Kazerounian, and the anonymous reviewers for their helpful comments. This research was supported in part by a Draper Fellowship.

References

1. Bjørner, N.: Linear quantifier elimination as an abstract decision procedure. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 316–330. Springer, Heidelberg (2010). https://doi.org/https://doi.org/10.1007/978-3-642-14203-1_27
2. Bodík, R., et al.: Programming with angelic nondeterminism. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, pp. 339–352. ACM, New York (2010). https://doi.org/https://doi.org/10.1145/1706299.1706339
3. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM **20**(10), 762–772 (1977). https://doi.org/https://doi.org/10.1145/359842.359859
4. Bradley, A.R., Manna, Z.: The calculus of computation: decision procedures with applications to verification. Springer, Berlin (2007), oCLC: 255687662
5. Cheung, A., Solar-Lezama, A., Madden, S.: Using program synthesis for social recommendations. In: Chen, X., Lebanon, G., Wang, H., Zaki, M.J. (eds.) 21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012, pp. 1732–1736. ACM, Hawaii, USA (2012). https://doi.org/https://doi.org/10.1145/2396761.2398507, http://dl.acm.org/citation.cfm?id=2396761
6. Clopper, C.J., Pearson, E.S.: The use of confidence or fiducial limits illustrated in the case of the binomial. Biometrika **26**(4), 404–413 (1934). https://doi.org/https://doi.org/10.1093/biomet/26.4.404, publisher: Oxford Academic
7. Ellis, K., Ritchie, D., Solar-Lezama, A., Tenenbaum, J.: Learning to infer graphics programs from hand-drawn images. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31, pp. 6059–6068. Curran Associates, Inc. (2018)

8. Fazzini, M., Gorla, A., Orso, A.: A framework for automated test mocking of mobile apps. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), NIER track, pp. 1204–1208 (Sep 2020), iSSN: 2643–1572
9. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous Java performance evaluation. ACM SIGPLAN Notices **42**(10), 57–76 (2007). https://doi.org/https://doi.org/10.1145/1297105.1297033
10. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 62–73. ACM, New York (2011). https://doi.org/https://doi.org/10.1145/1993498.1993506
11. Hua, J., Zhang, Y., Zhang, Y., Khurshid, S.: EdSketch: execution-driven sketching for Java. Int. J. Softw. Tools Technol. Transf. **21**(3), 249–265 (2019). https://doi.org/https://doi.org/10.1007/s10009-019-00512-8
12. Huang, K., Qiu, X., Shen, P., Wang, Y.: Reconciling enumerative and deductive program synthesis. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1159–1174. PLDI 2020, Association for Computing Machinery, New York, June 2020. https://doi.org/https://doi.org/10.1145/3385412.3386027
13. Inala, J.P., Polikarpova, N., Qiu, X., Lerner, B.S., Solar-Lezama, A.: Synthesis of recursive ADT transformations from reusable templates. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 247–263. Springer, Heidelberg (2017). https://doi.org/https://doi.org/10.1007/978-3-662-54577-5_14
14. Inala, J.P., Singh, R., Solar-Lezama, A.: Synthesis of domain specific CNF encoders for bit-vector solvers. In: Creignou, N., Berre, D.L. (eds.) Theory and Applications of Satisfiability Testing - SAT 2016–19th International Conference, Bordeaux, France, July 5–8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9710, pp. 302–320. Springer (2016). https://doi.org/https://doi.org/10.1007/978-3-319-40970-2_19
15. Jeon, J., Qiu, X., Fetter-Degges, J., Foster, J.S., Solar-Lezama, A.: Synthesizing framework models for symbolic execution. In: Proceedings of the 38th International Conference on Software Engineering, pp. 156–167. ICSE 2016, Association for Computing Machinery, New York, May 2016. https://doi.org/https://doi.org/10.1145/2884781.2884856
16. Jeon, J., Qiu, X., Solar-Lezama, A., Foster, J.S.: JSketch: sketching for Java. In: European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE). Tool Demo Track, pp. 934–937. ACM, Bergamo, Italy, September (2015)
17. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 316–329. PLDI 2010, ACM, New York (2010). https://doi.org/https://doi.org/10.1145/1806596.1806632
18. Mariano, B., et al.: Program synthesis with algebraic library specifications. Proc. ACM Program. Lang. **3**(OOPSLA), 132:1–132:25 (2019). https://doi.org/https://doi.org/10.1145/3360558
19. Nguyen, T.V., Weimer, W., Kapur, D., Forrest, S.: Connecting program synthesis and reachability: automatic program repair using test-input generation. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 301–318. Springer, Heidelberg (2017). https://doi.org/https://doi.org/10.1007/978-3-662-54577-5_17

20. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2016, pp. 522–538. ACM Press, Santa Barbara, CA, USA (2016). https://doi.org/https://doi.org/10.1145/2908080.2908093
21. Saff, D., Artzi, S., Perkins, J.H., Ernst, M.D.: Automatic test factoring for Java. In: Proceedings of the 20th IEEE/ACM International Conference on Automated software engineering, ASE 2005, pp. 114–123. Association for Computing Machinery, New York, November 2005. https://doi.org/https://doi.org/10.1145/1101908.1101927
22. Samak, M., Kim, D., Rinard, M.C.: Synthesizing replacement classes. Proc. ACM Programm. Lang. **4**(POPL), 52:1–52:33 (2019). https://doi.org/https://doi.org/10.1145/3371120
23. Shi, K., Steinhardt, J., Liang, P.: FrAngel: component-based synthesis with control structures. Proc. ACM Programm. Lang. **3**(POPL), 73:1–73:29 (2019). https://doi.org/https://doi.org/10.1145/3290386
24. Singh, R., Singh, R., Xu, Z., Krosnick, R., Solar-Lezama, A.: Modular synthesis of sketches using models. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 395–414. Springer, Heidelberg (2014). https://doi.org/https://doi.org/10.1007/978-3-642-54013-4_22
25. Smith, C., Albarghouthi, A.: Program synthesis with equivalence reduction. In: Enea, C., Piskac, R. (eds.) VMCAI 2019. LNCS, vol. 11388, pp. 24–47. Springer, Cham (2019). https://doi.org/https://doi.org/10.1007/978-3-030-11245-5_2
26. Solar Lezama, A.: Program synthesis by sketching. Ph.D. thesis, EECS Department, University of California, Berkeley, December 2008. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html
27. Solar-Lezama, A.: The sketching approach to program synthesis. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 4–13. Springer, Heidelberg (2009). https://doi.org/https://doi.org/10.1007/978-3-642-10672-9_3
28. Solar-Lezama, A.: The sketch programmers manual. Tech. rep, MIT, February 2020
29. Solar-Lezama, A., Arnold, G., Tancau, L., Bodik, R., Saraswat, V., Seshia, S.: Sketching stencils. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2007, pp. 167–178. Association for Computing Machinery, San Diego, June 2007. https://doi.org/https://doi.org/10.1145/1250734.1250754
30. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Shen, J.P., Martonosi, M. (eds.) Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, 21–25 October, 2006, pp. 404–415. ACM (2006). https://doi.org/https://doi.org/10.1145/1168857.1168907
31. Van Geffen, J., Nelson, L., Dillig, I., Wang, X., Torlak, E.: Synthesizing JIT compilers for in-kernel DSLs. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 564–586. Springer, Cham (2020). https://doi.org/https://doi.org/10.1007/978-3-030-53291-8_29

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.