Noname manuscript No.
(will be inserted by the editor)

An Empirical Study of Adaptive Concretization for Parallel Program Synthesis

Jinseong Jeon · Xiaokang Qiu ·
Armando Solar-Lezama · Jeffrey S. Foster

Received: Oct 23, 2015

Abstract Adaptive concretization is a program synthesis technique that enables efficient parallelization of challenging synthesis problems. The key observation behind adaptive concretization is that in a challenging synthesis problem, there are some unknowns that are best suited for explicit search and some that are best suited for symbolic search through constraint solving. At a high level, the main idea behind adaptive concretization is to dynamically identify which unknowns are best suited to which kind of search, and to parallelize the explicit search on those unknowns for which that style of search is more suitable.

We first introduced adaptive concretization in an earlier paper [13]. Our original algorithm involved a few arbitrary design decisions, leaving open the question of whether different choices could achieve better performance. In this paper, we systematically evaluate several dimensions of the design space to better understand the tradeoffs. We show that, in general, adaptive concretization is robust along those dimensions, and our initial choices were reasonable.

1 Introduction

Program synthesis aims to construct a program satisfying a given specification. One popular style of program synthesis is *syntax-guided synthesis*, which starts with a structural hypothesis describing the shape of possible programs, and then searches through the space of candidates until it finds a solution. Recent years have seen a number of successful applications of syntax-guided synthesis, ranging

Jinseong Jeon and Jeffrey S. Foster
University of Maryland, College Park
E-mail: {jsjeon, jfoster}@cs.umd.edu

Xiaokang Qiu
Purdue University (This work was partially done when the author was at MIT as a postdoc.)
E-mail: xkqiu@purdue.edu

Armando Solar-Lezama
Massachusetts Institute of Technology
E-mail: asolar@csail.mit.edu

from automated grading [21], to programming by example [9], to synthesis of cache coherence protocols [25], among many others [6,17,23].

Despite their common conceptual framework, each of these systems relies on different synthesis procedures. One key algorithmic distinction is that some use *explicit search*—either stochastically or systematically enumerating the candidate program space—and others use *symbolic search*—encoding the search space as constraints that are solved using a SAT solver. The SyGuS competition has recently revealed that neither approach is strictly better than the other [1].

In our prior work [13], we proposed *adaptive concretization*, a new approach to synthesis that combines many of the benefits of explicit and symbolic search while also parallelizing very naturally, allowing us to leverage large-scale, multi-core machines. Adaptive concretization is based on the observation that in synthesis via symbolic search, the unknowns that parameterize the search space are not all equally important in terms of solving time. In Section 3, we show that while symbolic methods can efficiently solve for some unknowns, others—which we call *highly influential* unknowns—cause synthesis time to grow dramatically. Adaptive concretization uses explicit search to concretize influential unknowns with randomly chosen values and searches symbolically for the remaining unknowns. We have explored adaptive concretization in the context of the Sketch synthesis system [22], although we believe the technique can be readily applied to other symbolic synthesis systems such as Brahma [14] or Rosette [24].

The original adaptive concretization algorithm involved a few arbitrary design decisions which were not fully evaluated. In this paper, we perform a systematic evaluation of the design space to better understand the tradeoffs. We use 31 benchmarks (of which 26 were from the original paper), and we use the same experimental platform; however, we use the latest version of Sketch, which has improved since the original paper was published. (Section 4 provides details of our experimental design.)

Our adaptive concretization algorithm is parameterized by a *degree of concretization*, which controls the exact probability of concretizing an unknown based on its influence. The exact influence computation is explained in detail in Section 8, but intuitively it tries to give more weight to unknowns used as conditional guards, since their values are likely to cause big changes to the choices of other unknowns. In the original algorithm, at degree 0, an unknown with any positive influence is concretized with probability 1/2; at degree ∞, only unknowns with influence 1500 or higher are concretized. The degree is an important parameter because the optimal amount of concretization depends heavily on the particular benchmark [13]—hence the "adaptive" part of adaptive concretization is to search for the optimal degree in an on-line manner, as synthesis proceeds.

While our original algorithm worked well, it was unsatisfying to have 1/2 and 1500 baked into the probability function as constants, especially because there is not much intuitive justification for those two choices. Moreover, the probability function, parameterized by an unknown's influence and the degree of concretization, was *discontinuous*—an unknown with influence 1500 or higher is always concretized, but an unknown with influence 1499 or lower is concretized at most 1/2 the time. Thus, we introduce a new, *smooth* probability function, with the same parameters, that eliminates the constants and the discontinuity. In Section 5, we empirically show that the new, smooth function behaves similarly to the discontinuous function.

In our original work, we showed that under the discontinuous probability function, if we make a graph with the degree on the x-axis and the expected time to find a solution on the y-axis, the graph forms a "vee" shape with a low point at the optimal degree. This justifies adaptive concretization's degree search process, which uses a combination of exponential hill climbing and binary search. In Section 6, we empirically demonstrate on a subset of the benchmarks that the "vee" shape still occurs under the new, smooth probability function, hence re-justifying adaptive concretization's search process.

Finally, we consider the last two "magic constants" in adaptive concretization. First, during degree search, the algorithm uses the Wilcoxon Signed-Rank Test [27] to compare the mean expected synthesis time from two sets of trials at two different degrees. That test returns a p-value indicating the probability any observed difference in the mean is due to random chance. Once the p-value exceeds a threshold T, adaptive concretization determines one degree is better than the other and then continues searching at a different pair of degrees. In our original algorithm, we fixed T at 0.2. In Section 7, we use a simulation of adaptive concretization to compare five different T values ranging from 0.001 to 0.5. We find that choosing a T anywhere between 0.05 and 0.2 seems to yield similarly good results.

Second, adaptive concretization's original influence calculation assigns arbitrary boolean unknowns 0.5 times the influence of unknowns in guard positions of if-then-else nodes. In Section 8, we empirically evaluate a range of different values for this ratio, B, ranging from 1/8 to 2. Surprisingly, we find no meaningful differences among this wide range of choices, suggesting the influence calculation is not sensitive to the choice of B.

Cumulatively, our results put adaptive concretization on a much firmer foundation by demonstrating that the algorithm is robust to a wide range of design decisions.

2 Related Work

There have been many recent successes in sampling-based synthesis techniques. For example, Schkufza et al. use sampling-based synthesis for optimization [17, 18], and Sharma et al. use similar techniques to discover complex invariants in programs [19]. These systems use Markov Chain Montecarlo (MCMC) techniques, which use fitness functions to prioritize sampling over regions of the solution space that are more promising. This is a more sophisticated sampling technique than what is used by our method. We leave it to future work to explore MCMC methods in our context. Another alternative to constraint-based synthesis is explicit enumeration of candidate solutions. Enumerative solvers often rely on factoring the search space, aggressive pruning and lattice search. Factoring has been very successful for programming by example [9, 11, 20], and lattice search has been used in synchronization of concurrent data structures [26] and autotuning [2]. However, both factoring and lattice search require significant domain knowledge, so they are unsuitable for a general purpose system like SKETCH. Pruning techniques are more generally applicable, and are used aggressively by enumerative solvers.

Recently, some researchers have explored ways to use symbolic reasoning to improve sampling-based procedures. For example, Chaudhuri et al. have shown how to use numerical search for synthesis by applying a symbolic smoothing transformation [4, 5]. In a similar vein, Chaganty et al. use symbolic reasoning to limit the sampling space for probabilistic programs to exclude points that will not satisfy a specification [3].

Finally, there has been significant interest in parallelizing SAT/SMT solvers. The most successful of these combine a portfolio approach—solvers are run in parallel with different heuristics—with clause sharing [10, 28]. Interestingly, these solvers are more efficient than solvers like PSATO [29] where every thread explores a subset of the space. One advantage of our approach over solver parallelization approaches is that the concretization happens at a very high-level of abstraction, so the solver can apply aggressive algebraic simplification based on the concretization. This allows our approach to even help a problem that ran out of memory on the sequential solver. The tradeoff is that our solver loses the ability to tell if a problem is UNSAT because we cannot distinguish not finding a solution from having made incorrect guesses during concretization.

3 Adaptive Concretization

We begin by reviewing how adaptive concretization works. To motivate the algorithm, consider the following example input to the Sketch synthesis tool:

```
bit[32] foo(bit[32] x) implements spec{
  if(??){
      return x & ??; // unknown m1
  }else{
      return x | ??; // unknown m2
} }
```

```
bit[32] spec(bit[32] x){
      return minus(x, mod(x, 8));
}
```

Here *??* represents an unknown constant, which is inferred to be a 1-bit boolean in the branch condition, and a 32-bit integer in the unknowns labeled *m1* and *m2*. The specification on the right asserts the synthesized code must compute $x-(x \ mod \ 8)$.

The sketch above has 65 unknown bits and 2^{33} unique solutions, which is too large for a naive enumerative search. However, the problem is easy to solve with *symbolic search*. Symbolic search works by symbolically executing the template to generate constraints among the unknowns, and then generating a series of SAT problems that solve the unknowns for well-chosen test inputs. Using this approach, Sketch solves this problem in about 50ms, which is certainly fast.

However, not all unknowns in this problem are equal. While the bit-vector unknowns are well-suited to symbolic search, the unknown in the branch is much better suited to explicit search. In fact, if we incorrectly concretize that unknown to *false*, it takes only 2ms to discover the problem is unsatisfiable. If we concretize it correctly to *true*, it takes 30ms to find a correct answer. Thus, enumerating concrete values lets us solve the problem in 32ms (or 30ms if in parallel), which is much faster than pure symbolic search. For larger benchmarks the speedup is even greater, and can make the difference between solving a problem in seconds and not solving it at all.

It is worth emphasizing that the unknown controlling the branch is special. For example, if we concretize one of the bits in *m1*, the solution time does not improve. Worse, if we concretize incorrectly, it will take almost the full 50ms to discover the problem is unsatisfiable, and then we will have to flip to the correct value and take another 50ms to solve, thus doubling the solution time.

From this example, we see that some unknowns are highly *influential*, as concretizing them makes the remaining symbolic synthesis problem significantly easier, leading to a much faster overall synthesis algorithm. Moreover, concretizing leads very naturally to parallelization, since we can try different concrete values on different cores.

Thus, if we can combine symbolic and explicit search, we can potentially do better than either alone. However, we need to solve two key challenges. First, there is no practical way to compute the precise influence of an unknown. Instead, our algorithm estimates that an unknown is highly influential if concretizing it will likely shrink the constraint representation of the problem. Second, because influence computations are estimates, even the highest influence unknown may not affect the solving time for some problems. Thus, our algorithm uses a series of trials, each of which makes an independent decision of what to *randomly* concretize. This decision is parameterized by a *degree of concretization*, which adjusts the probability of concretizing a high influence unknown.

More details of our influence measure and the degree of concretization, and an empirical comparison of different design decisions, appear in Sections 5 and 8.

3.1 Adaptive, Parallel Concretization

In our prior work, we found that no single degree of concretization was best across all subject programs [13]. However, we did find that each benchmark had an *optimal* degree, leading to the fastest solution time, and that the farther away from the optimal degree, the slower the solution time. (We will explore this finding again in Section 6.) Thus, adaptive concretization works by *searching* for the optimal degree, as we discuss next.

Figure 1 gives pseudocode for adaptive concretization. The core step of our algorithm, encapsulated in the **run_trial** function, is to run SKETCH with the specified degree. That is, SKETCH will calculate the influence of each unknown; randomly select unknowns to concretize based on their influence and the degree, using the probability functions described in Section 5; assign randomly chosen values to the selected unknowns; and then use the standard symbolic solving engine for the remaining unknowns.

If a solution is found, we exit the search. Otherwise, we return a rough estimate of how long it will take to find a solution if we continue running at the current degree. We compute the estimate by first determining the size m of the concretization space, that is, the number of possible different ways we could concretize the unknowns that were selected for this by SKETCH. For example, if SKETCH chose n bits to concretize, then $m = 2^n$.

Then we make two assumptions: First, we assume that there is exactly one solution, so the probability of finding with a random concretization is $1/m$. Second, we assume that every trial at this degree will take the same amount of time. Thus the expected time to find a solution is the running time of the trial multiplied by m.

```
run_trial(degree)
  run SKETCH with specified degree
  if solution found then
    raise success
  else
    return ( running time *
               |concretization space| )


compare(deg_a, deg_b)
  dist_a ← ∅
  dist_b ← ∅
  while |dist_a| ≤ Max_dist ∧
        wilcoxon(dist_a, dist_b) > T do
    dist_a ∪← run_trial(deg_a)
    dist_b ∪← run_trial(deg_b)
  if wilcoxon(dist_a, dist_b) > T then
    return tie
  elsif avg(dist_a) < avg(dist_b) then
    return left
  else
    return right
```

```
climb()
  low, high ← 0, 1
  while high < Max_exp do
    case compare($2^{low}$, $2^{high}$) of
      left : break
      right:
        low ← high
        high ← high + 1
      tie: high ← high + 1
  return (low, high)

bin_search(low, high)
  mid ← (low + high) / 2
  case compare(low, mid) of
    left : return bin_search(low, mid)
    right: return bin_search(mid, high)
    tie: return mid

main()
  (low, high) ← climb()
  deg ← bin_search($2^{low}$, $2^{high}$)
  while (true) do run_trial(deg)
```

Fig. 1: Adaptive Concretization Algorithm.

Comparing Degrees. Since SKETCH solving has some randomness in it, a single trial does not provide a good estimate of the time-to-solution. Moreover, the precise number of trials needed to make a good estimate will vary from benchmark to benchmark, so setting the number of trials to a fixed value would mean that either we waste time running to many or we compute inaccurate information by running too few.

Instead, our algorithm uses the *Wilcoxon Signed-Rank Test* [27] to determine when we have enough data to distinguish two degrees. Given two sets of estimates for the solution time generated by two different degrees, the **wilcoxon***(dist_a, dist_b)* routine determines the probability that the two sets of samples were drawn from the same distribution of times to solution; this probability is known as the p-value. Once the Wilcoxon test determines there is enough data to confidently distinguish the two distributions, we chose the degree that promises the best solution time. If a maximum set of samples *Max_dist* is reached without reaching a good enough p-value, the two degrees are deemed to be indistinguishable. This whole process of repeatedly running trials and applying Wilcoxon tests takes place inside the function **compare**, which takes two degrees as inputs and decides which one is better.

In our experiments, we use $3 \times max(8, |cores|)$ for *Max_dist*. Thus, **compare** runs at most three "rounds" of at least eight samples (or the number of cores, if that is larger). This lets us cut off **compare** if it does not seem to be finding any distinction. In Section 5, we use 0.2 for the threshold T to match our previous work. In Section 6, we empirically evaluate a range of p-values to find the best choice.

Searching for the Optimal Degree. Now we can implement the search algorithm. The entry point is **main**, shown in the lower-right corner of Figure 1. There are two algorithm phases: an *exponential climbing* phase (function **climb**) in which we try to roughly bound the optimal degree, followed by a binary search (function **bin_search**) within those bounds.

We opted for an initial exponential climb because binary search across the whole range could be extremely slow. Consider the first iteration of such a process, which would compare full concretization against no concretization. While the former would complete almost instantaneously, the latter could potentially take a long time and would not stop until it found a solution, thereby defeating the purpose of concretization.

Exponential Climb. The **climb** function aims to return a pair *low*, *high* such that the optimal degree is between 2^{low} and 2^{high}. It begins with *low* and *high* as 0 and 1, respectively. It then increases both until it finds values such that at degree 2^{high}, search is estimated to take a longer time than at 2^{low}. Notice that the initial trials of the **climb** will be extremely fast, because almost all variables will be concretized.

To perform this search, **climb** repeatedly calls **compare**, passing in 2 to the power of *low* and *high* as the degrees to compare. Then there are three cases. If **left** is returned, 2^{low} has better expected running time than 2^{high}. Hence we assume the true optimal degree is somewhere between the two, so we return them. Otherwise, if **right** is returned, then 2^{high} is better than 2^{low}, so we shift up to the next exponential range. Finally, if it is a **tie**, then the range is too narrow to show a difference, so we widen it by leaving *low* alone and incrementing *high*. We also terminate climbing if *high* exceeds some maximum exponent *Max_exp*. In our implementation, we choose *Max_exp* as 14, since for our subject programs this makes runs nearly all symbolic.

Binary Search. After finding rough bounds with **climb**, we then continue with a binary search. Notice that in **bin_search**, *low* and *high* are the actual degrees, whereas in **climb** they are degree exponents. Binary search is straightforward, maintaining the invariant that *low* has expected faster or equivalent solution time to *high* (recall this is established by **climb**). Thus each iteration picks a midpoint *mid* and determines whether *low* is better than *mid*, in which case *mid* becomes the new *high*; or *mid* is better, in which case the range shifts to *mid* to *high*; or there is no difference, in which case *mid* is returned as the optimal degree.

Finding a solution. Finally, after the degree search has finished, we repeatedly run Sketch with the given degree. The search exits when **run_trial** finds a solution, which it signals by raising an exception to exit the algorithm. (Note that **run_trial** may find a solution at any time, including during **climb** or **bin_search**.)

Parallelization. Our algorithm is easy to parallelize. The natural place to do this is inside **run_trial**: Rather than run a single trial at a time, we perform parallel trials. More specifically, our implementation includes a worker pool of a user-specified size. Each worker performs concretization randomly at the specified degree, and thus they are highly likely to all be doing distinct work.

The algorithm as described so far involves a number of design decisions, some of which are important to the performance of the algorithm and some of which are not. In the rest of the paper, we conduct a detailed empirical evaluation to evaluate the most salient of these design decisions against possible alternatives.

4 Experimental Design

We evaluated adaptive concretization across 31 benchmarks collected from five categories of synthesis problems.[1] Each category stems from a distinct application domain and varies in complexity, amount of symmetry, etc. We briefly describe each category below, and Table 1 lists each benchmark, along with its lines of code and brief description.

– PASKET. The first three benchmarks, beginning with `p_`, come from the application that inspired this work: PASKET, a tool that aims to construct executable code that behaves the same as a framework such as Java Swing, but is much simpler to statically analyze [12]. PASKET's sketches are some of the largest that have ever been tried, and we developed adaptive concretization because they were initially intractable with SKETCH.
– *Data Structure Manipulation.* The second set of benchmarks, beginning with `l_` or `t_`, is from a project aiming to synthesize provably correct data-structure manipulations [16]. Each synthesis problem consists of a program template and logical specifications describing the functional correctness of the expected program.
– *Invariants for Stencils.* The next set of benchmarks, beginning with `a_mom_`, are from a system that synthesizes invariants and postconditions for scientific computations involving stencils. In this case, the stencils come from a DOE Miniapp called Cloverleaf [8]. These benchmarks involve primarily integer arithmetic and large numbers of loops.
– *SyGuS Competition.* The next set of benchmarks, beginning with `ar_` and `hd_`, are from the first Syntax-Guided Synthesis Competition [1], which compared synthesizers using a common set of benchmarks. We selected nine benchmarks that took at least 10 seconds for any of the solvers in the competition, but at least one solver was able to solve it.
– SKETCH. The last group of benchmarks, beginning with `s_`, `deriv`, and `q_`, are from SKETCH's performance test suite, which is used to identify performance regressions in SKETCH and measure potential benefits of optimizations.

Throughout the paper, all performance reports (such as Tables 2 and 5) are based on 13 runs on a server equipped with forty 2.4 GHz Intel Xeon processors and 99 GB RAM, running Ubuntu 14.04.1. LTS. We used the same machine to collect per-degree SKETCH data for the simulations in Section 6. The baseline sketch results for this paper were run using SKETCH release 1.7.0, which is a newer version than the one from our prior work [13].

5 Concretization Probability

As mentioned in Section 3.1, adaptive concretization probabilistically chooses the unknowns to concretize, as determined by the degree of concretization and the

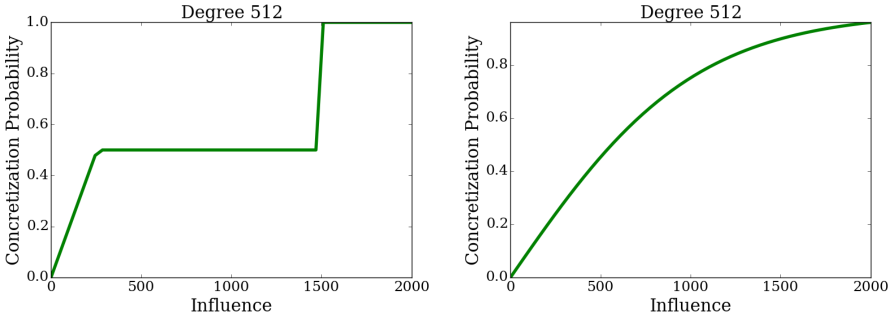
[1] Our testing infrastructure, benchmarks, and raw experimental data are open-sourced and explained at: `http://plum-umd.github.io/adaptive-concretization/`.

Benchmark		**Description**
Name	**LoC**	
`p_button`	3,436	aims to synthesize a model of JButton and ActionListener
`p_color`	3,194	aims to synthesize a model of JColorChooser
`p_menu`	4,099	aims to synthesize a model of JMenu and JMenuItem
`p_file`	8,706	aims to synthesize a model of JFileChooser
`p_check`	5,455	aims to synthesize a model of CheckBox and Button for Android
`l_prepend`	708	accepts a sorted singly linked list L and prepends a key k, which is smaller than any element in L
`l_min`	795	traverses a singly linked list via a while loop and returns the smallest key in the list
`l_reverse`	1,817	reverses a singly linked list via a while loop and returns the new head of the list
`l_insert`	1,842	inserts a new key into a sorted list via a while loop and returns the new head of the list
`t_rotate`	1,215	perform the left rotate manipulation on a binary search tree
`a_mom_1`	229	stencil 1
`a_mom_2`	231	stencil 2
`ar_s_4`	313	array search SyGuS benchmark
`ar_s_5`	334	larger array search benchmark
`ar_s_6`	337	larger array search benchmark
`ar_s_7`	322	larger array search benchmark
`ar_sum`	328	array sum SyGuS benchmark
`hd_13_d5`	310	hackers delight bit-vector SyGuS benchmark
`hd_14_d1`	304	another bit-vector SyGuS benchmark
`hd_14_d5`	329	another bit-vector SyGuS benchmark
`hd_15_d5`	329	another bit-vector SyGuS benchmark
`deriv2`	1,444	automatically grades Python code to compute a derivative
`deriv3`	1,410	different automated grading Python benchmark
`deriv4`	1,410	different automated grading Python benchmark
`deriv5`	1,410	different automated grading Python benchmark
`s_cg`	124	conjugate gradient benchmark from Sketch benchmark suite
`s_log2`	49	computes the logarithm base two of a bit vector
`s_logcnt`	30	counts the number of ones in a bit-vector in $\log n$ steps
`s_rev`	136	reverses a list
`q_noti`	262	SQL Query synthesis benchmark 1
`q_serv`	2,005	SQL Query synthesis benchmark 2

Table 1: Benchmarks.

unknown's influence. More concretely, when Sketch is run as part of the **run_trial** function in Figure 1, it computes each unknown's influence as a real number N. Details of the influence calculation are described in Section 8; for this section, it is only important to know that positive influence means there is some chance concretization will be beneficial.

Sketch then computes a probability of concretization based on the non-negative integer-valued degree d, where unknowns with high influence (i.e., with large N) are assigned a higher probability, as the overall probability of concretization decreases as d increases. (So, if d is 0, many unknowns are concretized, and if it is ∞, then none are.)



(a) Discontinuous

(b) Smooth

Fig. 2: Probability functions at degree 512.

5.1 Discontinuous Probability Function

In prior work [13], we computed the probability p of concretization using the following formula, which we refer to as the *discontinuous probability function*:

$$p = \begin{cases} 0 & \text{if } N < 0 \\ 1.0 & \text{if } N > 1500 \\ 1/(\max(2, d/N)) & \text{otherwise} \end{cases}$$

To understand this function, ignore the first two cases, and consider what happens when d is low, e.g., 10. Then any node for which $N \geq 5$ will have a 1/2 chance of being concretized, and even if N is just 0.5—the minimum N for an unknown not involved in arithmetic—there is still a 1/20 chance of concretization. Thus, low degree means many nodes will be concretized. In the extreme, if d is 0 then all nodes have a 1/2 chance of concretization. On the other hand, suppose d is high, e.g., 2000. Then a node with $N = 5$ has just a 1/400 chance of concretization, and only nodes with $N \geq 1000$ would have a 1/2 chance. Thus, a high degree means fewer nodes will be concretized. There are also two special cases: Nodes of influence less than 0 are never concretized, and nodes of influence greater than 1500 are always concretized.

Figure 2a draws the discontinuous probability function at degree 512. There is a linear slope from 0 until the ceiling of 0.5, followed by a straight line until the degree cutoff 1500, and then the probability becomes 1.0.

While this function worked well, it is unsatisfying for a few reasons. First, the choices of probability ceiling 0.5 and influence cut off 1500 are ad hoc, based on what worked well for a subset of our benchmarks. Second, it has two large discontinuities as shown in the figure. The one is a straight line; depending on the degree, its length, which corresponds to the range of somewhat ambiguous influences, might be too long; e.g., for degree 512, a node with 256 has the same 0.5 probability as a node with influence 1499 has. The other one is the discontinuous jump at influence 1500; a variable with influence 1499 is concretized with probability at most 0.5, whereas a variable with influence 1500 is always concretized. Such discontinuity is why we call this the *discontinuous* probability function.

5.2 Smooth Probability Function

To address these issues, we developed a new, *smooth* probability function:

$$p = \left(\frac{1}{1 + e^{-N/d}} - 0.5 \right) \times 2$$

Like the discontinuous function, the smooth function is parametrized only by degree d and influence N of node n; the larger N is, the more likely node n is concretized; and the larger d is, the less likely concretization is overall.

However, the smooth function addresses all the aforementioned issues. First, it does not include any ad hoc constants. In addition to base e, there are two extra constants in the formula, 0.5 and 2, but they are only used to ensure the output lies between 0 and 1. Second, it does not have any discontinuity. To visually compare both functions, Figure 2b depicts the smooth function at the same degree 512. As clearly shown in the figure, the smooth function has neither a straight line, where nodes with quite different influences may have the same concretization probability, nor discontinuous jumps at any points.

Thus, the first question we address in this paper is:

Research Question 1 *How does the smooth probability function compare to the discontinuous probability function?*

Table 2 compares both functions on our full benchmark suite, running on 32 cores. For each benchmark, we list its lines of code, followed by the results under the discontinuous probability function, the smooth probability function, and the speedup, which is the ratio of the running time under the smooth function to the time under the discontinuous function.

For each probability function, we list the median of the final degrees chosen by adaptive concretization (column d), the median number of calls to **run_trial** (column $||$), and the median running time. The columns that include running time are greyed for easy comparison, with the semi-interquartile range (SIQR) in a small font. We boldface the fastest time in each row.

Note that the discontinuous results differ from the data reported in our previous paper [13], because Sketch has undergone some significant improvements since we ran the earlier experiments.

Overall, the degrees chosen by both functions are very similar in the sense that they usually are within a factor of two, which indicates that the climbing phase ended in about the same range. The two probability functions are about the same in terms of performance. Indeed, each function outperforms the other one for half of the benchmarks. The median speedup is 1.0, the average is 1.038, and the variance is 0.05.

We applied *Mann-Whitney U test* [15], which tests whether one of given sample sets is consistently better than the other, to the performance results under the two probability functions. Notice that this test is different from Wilcoxon signed-rank test [27] that we used to compare two degrees during the adaptive concretization. The main difference is the category of input samples: Wilcoxon signed-rank test is applicable to repeatable samples on the single same benchmark, whereas Mann–Whitney U test is applicable to two performance result samples from the whole benchmark set. According to the statistical test, we cannot reject the alternative hypothesis that either performance set is exceeding the other, due to a very poor

Bench mark	Discontinuous				Smooth				speedup (D/S)
	d	||	Tm (s)		*d*	||	Tm (s)		
p_button	4,864	597	52	9	2,048	592	**46**	10	1.130
p_color	3,072	462	36	12	640	336	**21**	2	1.714
p_menu	212	590	**70**	14	3,072	601	72	28	0.972
p_file	6,144	577	139	20	5,120	492	**90**	13	1.544
p_check	256	254	**32**	2	256	208	33	2	0.970
l_prepend	32	88	**13**	1	256	151	17	1	0.765
l_min	128	204	41	12	256	204	**34**	10	1.206
l_reverse	32	1	**21**	1	16	1	21	2	1.000
l_insert	32	1	**25**	2	32	1	29	1	0.862
t_rotate	16	23	**49**	4	16	96	58	6	0.845
a_mom_1	256	306	248	20	1,024	222	**198**	12	1.253
a_mom_2	4,096	355	1,130	144	2,048	219	**848**	98	1.333
ar_s_4	32	11	**4**	0	16	3	5	0	0.800
ar_s_5	16	18	**5**	0	16	11	5	1	1.000
ar_s_6	32	38	9	2	16	15	**9**	0	1.000
ar_s_7	64	106	49	10	16	37	**40**	10	1.225
ar_sum	16	15	**40**	6	32	8	55	32	0.727
hd_13_d5	16	14	**8**	0	32	15	8	1	1.000
hd_14_d1	32	70	**16**	6	52	107	22	6	0.727
hd_14_d5	32	14	265	62	32	10	**237**	70	1.118
hd_15_d5	32	13	**130**	48	32	12	178	56	0.730
s_cg	64	141	13	1	32	73	**11**	2	1.118
s_log2	64	110	141	156	128	109	**136**	227	1.037
s_logcnt	32	110	27	8	32	37	**25**	46	1.080
s_rev	128	164	**40**	13	128	118	44	18	0.909
deriv2	16	20	**7**	2	16	25	8	1	0.875
deriv3	32	15	**7**	2	32	20	8	2	0.875
deriv4	16	17	6	0	32	18	**5**	0	1.200
deriv5	32	19	6	0	32	9	**5**	2	1.200
q_noti	32	115	**7**	0	64	125	7	2	1.000
q_serv	16	9	**21**	4	16	5	22	4	0.955

Table 2: Comparing Adaptive Concretization with discontinuous vs. smooth probability function.

confidence: 0.50. Therefore, from a statistical point of view, it is difficult to identify which one strictly outperforms the other, hence it is fairly safe to choose either function. The smooth probability function is preferable, since it is much intuitive due to the lack of design choices, i.e., magic numbers. In the remainder of the paper, all experiments use the new, smooth probability function.

Although there are no noticeable outliers, in the next subsection we investigated some cases where the smooth function performs better and some cases where the discontinuous function performs better, to get a better understanding of the algorithm.

5.3 Discussion

In the following discussion, we refer to unknowns as *holes*, which is SKETCH's internal terminology for unknowns. Holes are named by prefixing their unique id with `H__`, e.g., `H__26`. Sometimes the same syntactic hole may appear multiple times in the SAT formula due to inlining a function call or unrolling a loop. In this case, the hole name is appended with addition unique IDs, e.g., `H__26_22` and `H__26_23` are two instances of the same original syntactic hole `H__26`.

While investigating the experimental results, we found the benchmarks can be divided into three general categories: those with many influential holes; those with few influential holes; and those with a lot of symmetry. The smooth probability function tends to work better for the first two, and the discontinuous function better for the last one. We discuss each category in depth next, using concrete examples.

Many Influential Holes. Suppose there are many holes that are influential, but not above the cutoff of 1500 hard-coded into the discontinuous function. Then the smooth function tends to do better because it gives these holes a much higher probability of concretization, whereas the discontinuous function caps the probability at 0.5.

As an example, Figure 3 shows hole concretization statistics and histograms for `a_mom_2` at low, medium, and high degrees: 16, 512, and 2048, respectively. In each table, for each probability function, we list the success rate (how often we find a solution out of how many trials); the median time for a successful trial; the median time for a failed trial; and the maximum search space size.

We also give a partial histogram of the most often concretized holes, where the number indicates the count of times a hole was concretized in the trials, and the parenthesized number indicates in how many of those times synthesis was successful when that hole was concretized. For example, under the discontinuous function at degree 2048, hole `H__29_26_29` was concretized 264 times (out of 523 trials), and 1 concretization (out of 264) was in a trial that succeeded.

Looking at the table for degree 16, we see that under the discontinuous function, even the most influential holes (`H__26_23` and `H__26_22`) are concretized in at most half the trials, whereas they are always concretized under the smooth function. As a result, the maximum search space under the smooth function is four orders of magnitude larger. However, the failed trials also speed up, here by a factor of seven, thus leading the smooth function to give up sooner at this low degree. Under both functions the probability of success is extremely low, and there are no successful trials.

While the search algorithm climbs up to degree 2048, however, the two functions behave differently. Under the discontinuous function, those influential holes still have a 0.5 probability of concretization. In contrast, under the smooth function, a concretization probability of those holes has dropped from 0.5 to around 0.16. Thus the overall level of concretization is much smaller, as is the concretization search space (1,024 for smooth versus 32,768 for discontinuous).

In sum, many holes in this benchmark are equally influential, which makes it hard to find an optimal degree as well as a right amount and subset of holes for concretization. Our new smooth function achieves slightly better performance, for two reasons. First, when degrees are low, it can quickly climb up, thanks to faster

		Discontinuous	Smooth
Degree 16	Success Rate	0 / 592	0 / 559
	Success Time (ms)	N/A	N/A
	Fail Time (ms)	2,598	343
	Max Search Space	5.37e8	2.75e12
	Concretization Histogram		
	H__26_23	321 (0)	559 (0)
	H__26_22	312 (0)	559 (0)
	H__27_19_35	36 (0)	16 (0)
	H__8_23_25	18 (0)	17 (0)
	...	...	...
		Discontinuous	Smooth
Degree 512	Success Rate	0 / 624	0 / 320
	Succeeded Tm(ms)	N/A	N/A
	Failed Tm(ms)	10,243	9,485
	Max Search Space	131,072	4,194,304
	Concretization Histogram		
	H__26_23	301 (0)	140 (0)
	H__26_22	322 (0)	160 (0)
	H__27_27_35	56 (0)	11 (0)
	H__30_27_33	49 (0)	7 (0)
	...	...	...
		Discontinuous	Smooth
Degree 2048	Success Rate	1 / 523	37 / 320
	Success Time (ms)	764,183	757,921
	Fail Time (ms)	64,292	109,731
	Max Search Space	32,768	1,024
	Concretization Histogram		
	H__29_26_29	264 (1)	81 (6)
	H__30_26_35	276 (1)	76 (9)
	H__28_26_33	239 (1)	69 (3)
	H__27_26_35	286 (0)	80 (7)
	...	...	...

Fig. 3: Hole concretization histograms for a_mom_2.

individual trials caused by aggressive concretization. Second, when a degree is high enough, it can balance the amount of concretization and the running time of individual trials by generously lowering the concretization probability of influential holes.

Few Influential Holes. If there are few influential holes, then it is better to use symbolic search rather than explicit search. We observed that under the smooth function, holes with smaller influence tend to have a lower probability of concretization; thus the smooth function yields more symbolic search, which in turn achieves slightly better performance.

For example, Figure 4 shows the concretization statistics and histogram for ar_s_7 at degree 16. This benchmark is very exceptional in that it has many equally unimportant holes—more than 1,200 of them. Since their influence is small, the probability of concretizing them under the smooth function is lower than under the discontinuous function. For example, the concretization probability of the most influential hole, H__17_10_1_0, is around 0.3 under the discontinuous function, whereas under the smooth function it is around 0.15. As a result, fewer holes are

		Discontinuous	Smooth
Degree 16	Success Rate	0 / 536	8 / 323
	Success Time (ms)	N/A	39,600
	Fail Time (ms)	3,792	6,674
	Search Space	2.028e+31	2.882e+17
	Concretization Histogram		
	H__17_10_1_0	164 (0)	50 (2)
	H__17_10_2_1_0	160 (0)	51 (1)
	H__0_10_2_1_1_2	49 (0)	21 (1)
	H__13_10_1_0_4_1	24 (0)	15 (1)
	H__1_10_0_4_2_2	24 (0)	8 (1)
	(. . . and more than 1,200 similar holes)		

Fig. 4: Hole concretization statistics and histogram for `ar_s_7`.

		Discontinuous	Smooth
Degree 16	Success Rate	10 / 110	6 / 60
	Success Time (ms)	37,540	83,740
	Fail Time (ms)	4,586	12,359
	Search Space	7.556e+22	6.872e+11

Fig. 5: Hole concretization statistics for `ar_sum`.

concretized under the smooth function, and the search space is sixteen orders of magnitude smaller.

Symmetry. If the synthesis problem has a lot of symmetry, then concretization helps in general, because there's a high probability of finding a solution. In this case, the discontinuous function does better because it tends to concretize more.

For example, Figure 5 shows the concretization statistics for `ar_sum`. This benchmark is very similar to `ar_s_7` in that it has many low-influence holes. However, it also has many solutions, and thus even aggressive concretization under the discontinuous function has a relatively good chance of concretizing correctly.

In this particular example, the discontinuous function has a search space that is 11 orders of magnitude larger. This huge search space makes both successful and failed trials around twice as fast as the ones under the smooth function. But the empirical success rates are similar (10/110 vs. 6/60). Thus, the discontinuous function, which has faster individual trials, outperforms the smooth function.

6 Degree/Time Tradeoff Curve

A critical hypothesis underlying adaptive concretization is that there exists an optimal degree such that the farther away from the optimal degree, the slower the running time (i.e., the running time forms a "vee" around the optimal degree). While we found this held for the discontinuous probability function [13], we should reconfirm this for the smooth function before investigating other research questions:

Research Question 2 *Under the smooth probability function, do the expected running times across all concretization degrees form a vee shape around the optimal degree?*

Bench mark	Degree								
	16	32	64	128	256	512	1024	2048	4096
p_button	7	6	5	5	6	6	7	8	18
p_menu	1	1	2	163	196	196	196	196	196
l_prepend	>1M	>1M	>1M	908,067	474	19	6	4	4
l_min	1	1	0	4	52	337	597	142	150
a_mom_1	7,717	7,324	4,198	1,832	2,403	1,610	228	103	61
a_mom_2	>1M	>1M	>1M	>1M	>1M	>1M	8,697	1,571	641
hd_14_d5	23,142	10,010	4,184	N/A	N/A	N/A	N/A	N/A	N/A
hd_15_d5	5,299	29	0	0	N/A	N/A	N/A	N/A	N/A
s_log2	918	626	472	399	∞	∞	N/A	N/A	N/A
s_rev	>1M	>1M	>1M	39,290	13,111	146	N/A	N/A	N/A

Table 3: Expected running time (s) using empirical success rate. Fastest time in dark grey, second-fastest in light grey, ∞ if all failed trials exceed the 2-hour timeout, and N/A if there are no failed cases.

To answer this question, we created a database of individual trials of Sketch run on a particular benchmark under a given degree (i.e., the run on Sketch in **run_trial** in Fig. 1). We gathered the data from the 13 runs used to generate Table 2, and we also ran extra trials for various benchmark/degree combinations to gather more information (more details below). We used a 2-hour timeout for the extra trials. For each trial the database records whether it succeeded or failed, the running time t, and the size of the concretization space n.

We assume the running is single-threaded, one trial after another, and compute the expected time to success as $t * n$, where t is the median running time of failed trials, and n is search space of the failed trials after random concretization. Then we can group the trials in the database by their benchmark and concretization degree, and compute each benchmark/degree pair's median expected running time. Table 2 summarizes the results. Here we give data for the two longest-running (according to Table 2) benchmarks from each category.

There are a few exceptional cases in *SyGuS* and Sketch benchmarks. For *SyGuS* benchmarks, hd_*_d5, starting from degrees 128 or 256, randomly concretizing influential holes always succeeds in finding a solution, thanks to the low level of concretization as well as the symmetry in those benchmarks. For s_log2, at degrees 256 and 512, all the failed trials exceed the 2-hour timeout, hence ∞ expected running time. Starting from degree 1024, similar to *SyGuS* benchmarks, random concretization always succeeds as well. The other Sketch benchmark, s_rev, has the same behavior at the same degrees (from 1024 to 4096). In case of no failed cases, we cannot *expect* the running time of the random concretization because the (empirical) success rate is 1. (Such cases are labeled as N/A.) In terms of the optimal degree of the adaptive concretization, those degrees are out of scope, since the adaptive concretization eventually settles earlier on other beneficial degrees.

Except for those cases, we can generally see that the running times indeed form a "vee" around the optimal degree, i.e., performance gets worse the farther away from optimal in either direction. This matches our previous result for the discontinuous function, and it suggests that hill climbing and binary search can successfully find an optimal degree. The table also shows that the optimal degree

```
$t \leftarrow 0$ /* ``wall-clock'' time measurement */
sample_trial(degree)
  sample a failed trial with specified degree, and
  get the running time $\Delta$ and concretization space size $S$.
  $(\Delta, S) \leftarrow$ sample_from_database()
  $t \leftarrow t + \Delta$
  return $(\Delta/S)$
```

Fig. 6: Sampling trials in the database in lieu of actual Sketch runs.

varies across all benchmarks; indeed, all degrees except 1024 are optimal for at least one benchmark. This confirms our assumption that there is no fixed optimal degree, and necessitates our adaptive search algorithm.

In addition, we also use the resampling method [7] to check if our sample size is reasonable large to give us a reliable answer to Research Question 2. Specifically, for each benchmark and for each degree, if there are n trials in the database we resample from these trials n times, with replacement, and call the collected trials a bootstrap sample. Then based on these bootstrap samples, we compute the median expected running time for each benchmark/degree again, using the same formulation we set forth above. Although the expected running times are slightly different from those in Table 3 for most cases, bootstrap samples still show us the same "vee", with the same optimal degree. This experiment shows that what we reported in Table 3 is solid enough and reliable.

7 Wilcoxon Test Threshold

Now that we have data about the optimal degree of each benchmark, we can ask whether adaptive concretization actually finds it. Recall that the algorithm is parameterized by a threshold T for the p-value of the Wilcoxon test. Thus, we actually want to ask:

Research Question 3 *How is adaptive concretization's search affected by the threshold T?*

We could try to answer this by running adaptive concretization many times, but since we already have a database of trials, there is a better way: We can perform a *simulation* in which we run the algorithm, but instead of running Sketch itself, we randomly (with replacement) pick an appropriate benchmark/degree trial result from the database and return that from **run_trial**.

More concretely, we replace **run_trial** with a new function **sample_trial** in Figure 6. Here global variable t simulates the wall-clock time for the whole algorithm. Each time we retrieve a trial from the database, we add its time to the running wall-clock time and then return the estimated time to success.

We also simulate parallelized version of the algorithm using a single thread. Specifically, if there are n workers in the pool and the manager dispatches m trials at a time, the **sample_trial** function will sample failed trials m times with replacement, and get their running time Δ_1 through Δ_m. Then assuming that every worker is fully utilized in the long rung and $n \leq m$, **sample_trial** can simply

Bench mark	**Optimal Degree**	$T = 0.001$		$T = 0.05$	
		Degree	Tm	Degree	Tm
`p_button`	608 128	2056 2064	99	2064 2056	99
`p_menu`	30 16	2056 3076	251	44 40	231
`l_prepend`	2048 4096	4096 104	4199	48 4096	2310
`l_min`	64 32	2048 N/A	79K	2048 52	70K
`a_mom_1`	3328 4096	3328 2560	69K	3072 3328	56K
`a_mom_2`	4096 2048	3328 3320	289K	3328 3712	299K
`hd_14_d5`	64 32	64 N/A	224K	64 16	216K
`hd_15_d5`	64 32	64 N/A	110K	64 16	107K
`s_log2`	128 64	512 16	135K	512 16	135K
`s_rev`	512 256	512 48	81K	512 48	74K

Bench mark	$T = 0.1$		$T = 0.2$		$T = 0.5$	
	Degree	Tm	Degree	Tm	Degree	Tm
`p_button`	2080 2064	95	16 2080	90	16 48	N/A
`p_menu`	44 44	165	16 40	131	16 32	97
`l_prepend`	48 4096	2175	48 4096	1697	48 16	981
`l_min`	2048 16	64K	2048 16	52K	2048 16	41K
`a_mom_1`	3072 3584	50K	3072 16	47K	4096 16	26K
`a_mom_2`	3328 3712	301K	3328 3712	292K	32 16	270K
`hd_14_d5`	64 16	217K	64 16	208K	64 16	181K
`hd_15_d5`	64 16	107K	64 16	102K	64 16	94K
`s_log2`	512 48	133K	512 48	125K	512 16	N/A
`s_rev`	512 48	73K	512 48	69K	512 48	71K

Table 4: Simulating 32-core adaptive concretization as T varies: Mode degree found and Median time taken. Most often found degrees in large text and second most often found degrees in small text.

return the running results from the m trials, and advance the global time by $\left(\sum_{i\in[1,m]} \Delta_i\right)/n$.

Using this approach, we simulated 32-core adaptive concretization runs with T set at five thresholds: 0.001, 0.05, 0.1, 0.2 and 0.5. (Recall that a small number means we need more trials before reaching that significance level.) For each benchmark b and for each threshold p, we run adaptive concretization on b with p as the p-value, for 301 times. For each benchmark/threshold combination, we counted the most often found degree and the median time taken by adaptive concretization to find a fixed degree.

Note that the sampled trials might be insufficient for the Wilcoxon test to produce a small enough p-value, and resampling more trials from the database won't help. In that case, we run extra trials with the current concretization degree, and add them to the database and restart the simulation from scratch. This iteration continues until the mode degree becomes obvious, i.e., the mode degree won't change no matter what the unsettled runs' results would be. For each simulation that is still stuck on comparing degrees d_1 and d_2 due to the insufficient number of samples for d_1 or d_2 in the database, we assume every candidate degree (multiple of 16) within $[d_1, d_2]$ may be chosen with the same probability. For example, if a simulation stops at $[16, 48]$, we count each degree from $\{16, 32, 48\}$ as being chosen 1/3 times.

Table 4 summarizes our simulation results and compares them with the optimal degree based on all the trials in the database. The mode degrees are shown in large text and the second most often found degrees are shown in small text. Similarly, we show the optimal degrees in large text and the second-to-optimal degree in small text.

Experiments also show that the random nature of the adaptive concretization algorithm makes it robust with moderate thresholds ($T = 0.05$, 0.1 or 0.2): the found degrees are very similar, and lower thresholds usually take just slightly less time to find a degree than higher thresholds take. However, extreme thresholds ($T = 0.001$ or 0.5) are clearly not good choices: on the one hand, when the threshold is extremely high, the Wilcoxon test usually cannot conclude which degree is better, and then algorithm tends to climb to very high degrees for most benchmarks; on the other hand, when the threshold is extremely low, the Wilcoxon test's results can be easily affected by random noises, and the algorithm will stop at very low degrees too often. In summary, the simulation results suggest that any threshold between 0.05 and 0.2 is reasonable.

8 Influence Computation

As discussed earlier, the *influence* of an unknown is an estimate of how much concretizing that unknown may improve synthesis performance. Ideally our influence computation would model its exact effect on running time, but there is no practical way to compute this.

Instead, following the intuition from Section 3.1, we aim to assign high influence to unknowns that select among alternative program fragments (e.g., used as guards of conditions), and to give low influence to unknowns in arithmetic operations.

This immediately raises the question:

Research Question 4 *What should be the relative weighing between different influence unknowns?*

In particular, our influence measurement distinguishes guards of conditionals from other booleans; the former are likely highly influence, while the latter may or may not be. In our previous paper, we fixed a particular ratio of the influence between these two. In this section, we explore the tradeoff between different ratios.

In our influence computation, for unknown n we define

$$\mathit{influence}(n) = \sum_{d \in \mathit{children}(n)} \mathit{benefit}(d, n)$$

where $\mathit{children}(n)$ is the set of all nodes that depend directly on n. Here $\mathit{benefit}(d, n)$ is meant to be a crude measure of how much the overall formula might shrink if we concretize the parent node n of node d. The function is defined by case analysis on d:

- *Choices.* If d is an ite node,[2] there are two possibilities. If n is d's guard ($d = \text{ite}(n, a, b)$) then $\mathit{benefit}(d, n) = 1$. This is a high-influence position, so 1 is our baseline for the ratio between high and low influence. If n corresponds to one of the choices ($d = \text{ite}(c, n, b)$ or $d = \text{ite}(c, a, n)$), then $\mathit{benefit}(d) = 0$, since replacing n with a constant has no effect on the size of the formula.

[2] $\text{ite}(a, b, c)$ corresponds to **if** *(a)* b **else** c, as in SMT-LIB.

Bench	Influence Weights B													
mark	**1/8**		**1/4**		**1/2**		**3/4**		**1**		**4/3**		**2**	
`p_button`	42	4	35	8	46	10	41	6	38	10	42	6	50	8
`p_menu`	71	22	87	37	72	28	62	13	55	15	60	18	91	37
`l_prepend`	18	2	19	2	17	1	17	2	17	2	17	2	18	2
`l_min`	29	7	26	4	34	10	27	10	35	17	29	7	43	20
`a_mom_1`	211	32	204	24	198	12	188	18	185	37	199	30	214	26
`a_mom_2`	1,004	153	699	122	848	98	811	80	889	126	905	137	822	98
`hd_14_d5`	313	125	175	48	237	70	286	110	231	92	197	63	211	68
`hd_15_d5`	300	157	188	141	178	56	227	28	284	40	257	40	149	105
`s_log2`	222	161	444	461	136	227	159	271	137	310	390	212	84	70
`s_rev`	52	51	69	20	44	18	70	32	64	22	116	59	52	34

Table 5: Comparing influence weights of boolean nodes.

- *Boolean nodes.* If d is any boolean node except negation, its benefit should be some fraction B of the baseline benefit 1. In our previous work, we set B to be 0.5, so that ite nodes are two times as important as boolean nodes. Our intuition was that boolean nodes are often used in conditional guards, but sometimes not. We explore the choice of B shortly.
  If $d = \neg(n)$, then $\mathit{benefit}(d, n)$ equals $\mathit{influence}(d)$, since the benefit in terms of formula size of concretizing n and d is the same.
- *Choices among constants.* SKETCH's constraint graph includes nodes representing selection from a fixed sized array. If d corresponds to such a choice that is among an array of constants, then $\mathit{benefit}(d, n) = \mathit{influence}(d)$, i.e., the benefit of concretizing the choice depends on how many nodes depend on d.
- *Arithmetic nodes.* If d is an arithmetic operation, $\mathit{benefit}(d, n) = -\infty$. The intuition is that these unknowns are best left to the solver. For example, given *??+in*, replacing *??* with a constant will not affect the size of the formula.

Note that before settling on this particular influence measure, we tried a simpler approach that attempted to concretize holes that flow to conditional guards, with a probability based on the degree of concretization. However, we found that a small number of conditionals have a large impact on the size and complexity of the formula. Thus, having more refined heuristics to identify high influence holes is crucial to the success of the algorithm.

To evaluate the choice of B, the ratio between choice nodes and other booleans, we ran a subset of our benchmarks on seven ratios: 1/8, 1/4, 1/2 (our previous choice), 3/4, 1, 4/3, and 2. Notice the last two ratios weigh arbitrary booleans as more important than choice nodes. We used the same subset of benchmarks as Tables 3 and 4 in Section 6. We ran each benchmark/B combination thirteen times on 32 cores. Table 5 shows the results. As in Table 2, the columns show median running time, with the SIQR in a small font, and we highlight fastest and second-fastest times in each row.

From these results, the fastest running times appear across all degrees except for 1/8, though the second-smallest weight, 1/4, typically has many fastest running times. This reinforces our intuition that choice nodes should be more influential than boolean nodes. However, the performance differences are not that huge. In order to see whether there exists a degree that outperforms everything else, we applied Mann-Whitney U test again to all possible pairs of ratios. The confidence from those tests ranges from 0.35 to 0.96, which simply implies that there is no best ratio at all. This rather indicates that our influence computation is not sensitive to the ratio between choice nodes and other booleans.

9 Conclusion

In this paper, we empirically evaluated several of the key design choices in adaptive concretization, which we previously introduced [13]. The key insight behind adaptive concretization is that, by concretizing high *influence* unknowns, we can often speed up the overall synthesis algorithm, especially when we add parallelism. Since the best *degree of concretization* varies with the problem, adaptive concretization uses exponential hill climbing and binary search to find a suitable degree by running many trials in parallel.

We evaluated four key design decisions. First, we introduced a new function to assign a concretization probability to an unknown based on its influence and the degree. Our new function assigns probability in a smooth, continuous manner, eliminating both heuristic constants and a discontinuity in the original function. We showed that both functions behave similarly. We also showed that when graphed against expected running times, the degree forms a "vee" around the optimal point, justifying adaptive concretization's degree search process.

Finally, we explored a range of values for T, the threshold at which adaptive concretization decides that the p-value returned by the Wilcoxon Signed-Rank Test is significant enough to distinguish two degrees; and B, the ratio of the influence of arbitrary boolean unknowns versus those in guards of if-then-else nodes. We showed that many different choices for T and B work equally well, including $T = 0.2$ and $B = 0.5$, which were the choices in our original algorithm.

Overall, this paper makes adaptive concretization simpler by introducing a new, smooth concretization probability function, and we showed that our algorithm is robust to a wide range of design decisions.

Acknowledgements Supported in part by NSF CCF-1139021, CCF-1139056, CCF-1161775, and the partnership between UMIACS and the Laboratory for Telecommunication Sciences.

References

1. R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–17, 2013.
2. J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. P. Amarasinghe. Opentuner: an extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 303–316, 2014.
3. A. Chaganty, A. V. Nori, and S. K. Rajamani. Efficiently sampling probabilistic programs via program analysis. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2013, Scottsdale, AZ, USA, April 29 - May 1, 2013*, pages 153–160, 2013.

4. S. Chaudhuri, M. Clochard, and A. Solar-Lezama. Bridging boolean and quantitative synthesis using smoothed proof search. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 207–220, 2014.
5. S. Chaudhuri and A. Solar-Lezama. Smooth interpretation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 279–291, 2010.
6. A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 3–14, 2013.
7. B. Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1):1–26, 1979.
8. W. Gaudin, A. Mallinson, O. Perks, J. Herdman, D. Beckingsale, J. Levesque, and S. Jarvis. Optimising hydrodynamics applications for the cray xc30 with the application tool suite. *The Cray User Group*, pages 4–8, 2014.
9. S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330, 2011.
10. Y. Hamadi, S. Jabbour, and L. Sais. Manysat: a parallel SAT solver. *JSAT*, 6(4):245–262, 2009.
11. W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 317–328, 2011.
12. J. Jeon, X. Qiu, J. Fetter-Degges, J. S. Foster, and A. Solar-Lezama. Synthesizing Framework Models for Symbolic Execution. In *38th International Conference on Software Engineering (ICSE '16)*, May 2016.
13. J. Jeon, X. Qiu, A. Solar-Lezama, and J. S. Foster. Adaptive Concretization for Parallel Program Synthesis. In *Computer Aided Verification (CAV)*, volume 9207 of *Lecture Notes in Computer Science*, pages 377–394, San Francisco, CA, USA, July 2015. Springer International Publishing.
14. S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM.
15. H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Annals of Mathematical Statistics*, 18(1):50–60, 1947.
16. X. Qiu and A. Solar-Lezama. Synthesizing Data-Structure Manipulations with Natural Proofs. Under submission.
17. E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 305–316, 2013.
18. E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 9, 2014.
19. R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 88–105, 2014.
20. R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 634–651, 2012.
21. R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 15–26, 2013.
22. A. Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5-6):475–495, 2013.

23. A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent data structures. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 136–148, 2008.
24. E. Torlak and R. Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 54, 2014.
25. A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur. TRANSIT: specifying protocols with concolic snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 287–296, 2013.
26. M. T. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 125–135, 2008.
27. F. Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80–83, 1945.
28. C. M. Wintersteiger, Y. Hamadi, and L. M. de Moura. A concurrent portfolio approach to SMT solving. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 715–720, 2009.
29. H. Zhang, M. P. Bonacina, and J. Hsiang. Psato: A distributed propositional prover and its application to quasigroup problems. *J. Symb. Comput.*, 21(4-6):543–560, June 1996.