Adaptive Concretization for Parallel Program Synthesis

Jinseong Jeon[1], Xiaokang Qiu[2],
Armando Solar-Lezama[2], and Jeffrey S. Foster[1]

[1] University of Maryland, College Park
[2] Massachusetts Institute of Technology

Abstract. Program synthesis tools work by searching for an implementation that satisfies a given specification. Two popular search strategies are *symbolic search*, which reduces synthesis to a formula passed to a SAT solver, and *explicit search*, which uses brute force or random search to find a solution. In this paper, we propose adaptive concretization, a novel synthesis algorithm that combines the best of symbolic and explicit search. Our algorithm works by partially concretizing a randomly chosen, but likely highly influential, subset of the unknowns to be synthesized. Adaptive concretization uses an online search process to find the optimal size of the concretized subset using a combination of exponential hill climbing and binary search, employing a statistical test to determine when one degree of concretization is sufficiently better than another. Moreover, our algorithm lends itself to a highly parallel implementation, further speeding up search. We implemented adaptive concretization for SKETCH, and we evaluated it on a range of benchmarks. We found adaptive concretization is effective in practice, outperforming SKETCH is many cases, sometimes significantly, and has good parallel scalability.

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 from automated grading [18], to programming by example [8], to synthesis of cache coherence protocols [22], among many others [6, 20, 14].

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 this paper, we propose *adaptive concretization*, a new approach to synthesis that combines many of the benefits of explicit and symbolic search. Adaptive

concretization has the added benefit of parallelizing very naturally, allowing us to leverage large-scale multi-core machines to solve otherwise intractable synthesis problems. In this paper, we explore adaptive concretization in the context of the SKETCH synthesis system [19], although we believe the technique can be readily applied to other symbolic synthesis systems such as Rosette [21] or Brahma [12]. (Section 2 illustrates the advantages of combining these two search strategies.)

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 2, we also 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.

Combining symbolic and explicit search requires solving two technical 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 make much difference in solving time for some problems.

The solution these problems is to design the explicit search portion of our algorithm as a series of trails, where every trail 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. At degree 1, unknowns are concretized with probability 0.5, and at degree ∞, the probability drops to zero. Introducing the degree of concretization poses its own challenge: a preliminary experiment showed that across a range of seven benchmarks and seven possible degrees, there is a different optimal degree for almost every benchmark. Moreover, as explicit search can be parallelized, the number of available cores also affects the optimal degree. (Section 3 describes the influence calculation, the degree of concretization, and this experiment.)

Since there is no fixed optimal degree, the crux of adaptive concretization is to estimate the optimal degree online. Our algorithm begins with a very low degree (i.e., a large amount of concretization), since trials are extremely fast. It then exponentially increases the degree (i.e., reduces the amount of concretization) until removing more concretization is estimated as no longer worthwhile. Since there is randomness across the trials, we use a statistical test to determine when a difference is meaningful. Once the exponential climb stops, our algorithm does binary search between the last two exponents to find the optimal degree, and it finishes by running with that degree. At any time during this process, the algorithm exits if it finds a solution. Adaptive concretization naturally parallelizes by using different cores to run the many different trials of the algorithm. (Section 4 discusses pseudocode for the adaptive concretization algorithm.)

We implemented our algorithm for SKETCH and evaluated it against 26 benchmarks from a number of synthesis applications including automated tutoring [18], automated query synthesis [6], and high-performance computing, as

well as benchmarks from the SKETCH performance benchmark suite [19] and from the SyGuS'14 competition [1]. By running our algorithm over seven thousand times across all benchmarks, we are able to present a detailed assessment of its performance characteristics. We found our algorithm outperforms SKETCH on 21 of 26 benchmarks, sometimes achieving significant speedups of 6× up to 72×. In two cases, adaptive concretization succeeds where SKETCH times out or runs out of memory. We also ran adaptive concretization on 1, 4, and 32 cores, and found it generally has reasonable parallel scalability. Finally, we compared adaptive concretization to the winner of the SyGuS'14 competition on a subset of the SyGuS'14 benchmarks and found that our approach performs better overall. (Section 5 presents our results in detail.)

2 Combining Symbolic and Explicit Search

To illustrate the idea of influence, consider the following SKETCH example:

```
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 the symbol ?? represents an unknown constant whose type is automatically inferred. Thus, the ?? in the branch condition is a boolean, and the other ??'s, labeled as unknowns *m1* and *m2*, are 32-bit integers. The specification on the right asserts that the synthesized code must compute $(x - (x \bmod 8))$.

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 those 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 for this unknown lets us solve the whole problem in 32ms (or 30ms if run in parallel), which is 35% faster than pure symbolic search. As we will show later, for larger benchmarks this can make the difference between solving a problem in seconds and not solving it at all.

The benefit of concretization may seem counterintuitive since SAT solvers also make random guesses, using sophisticated heuristics to decide which variables to guess first. To understand why explicit search for this unknown is beneficial, we need to first explain how SKETCH solves for these unknowns. First,

symbolic execution in SKETCH produces a predicate of the form $Q(x, c)$, where x is the 32-bit *input* bit-vector and c is a 65-bit *control* bit-vector encoding the unknowns. $Q(x, c)$ is true if and only if, for the program described by the choices of c, $foo(x)=x-(x \bmod 8)$. Thus, SKETCH's goal is to solve the formula $\exists c.\forall x.Q(x, c)$. This is a doubly quantified problem, so it cannot be solved directly with SAT.

SKETCH reduces this problem to a series of problems of the form $\wedge_{x_i \in E} Q(x_i, c)$, i.e., rather than solving for all x, SKETCH solves for all x_i in a carefully chosen set E. After solving one of these problems, the candidate solution c is checked symbolically against all possible inputs. If a counterexample input is discovered, that counterexample is added to the set E and the process is repeated. This is the Counter-Example Guided Inductive Synthesis (CEGIS) algorithm, and it is used by most published synthesizers(e.g. [22, 21, 12]).

SKETCH's solver represents constraints as a graph, similar to SMT solvers, and then iteratively solves SAT problems generated from this graph. The graph is essentially an AST of the formula, but with shared common sub-trees. For the simple example above, the formula $Q(x, c)$ consists of 488 nodes, and CEGIS takes 12 iterations on average. On each iteration, the algorithm plugs in a concrete x_i and simplifies this formula to 195 nodes. In contrast, when we concretize the condition, the size of $Q(x, c)$ reduces from 488 to 391 nodes, which simplify to 82 nodes per CEGIS iteration. Over 12 iterations, this factor of two in the size of the problem adds up. Moreover, when we concretize the condition to the wrong value, SKETCH discovers the problem is unsatisfiable after only one counterexample, which is why that case takes only 2ms to solve.

In short, unlike the random assignments the SAT solver uses for each individual sub-problem in the CEGIS loop, by assigning concrete values in the high-level representation, our algorithm significantly reduces the sub-problem sizes across *all* CEGIS loop iterations. It is worth emphasizing the unknown controlling the branch is special. For example, if we concretize one of the bits in *m1*, it only reduces the formula from 488 to 486 nodes, and 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. Thus, it is important to concretize only the most influential unknowns.

Putting this all together yields a simple, core algorithm for concretization. Consider an iteration of SKETCH with the formula $\wedge_{x_i \in E} Q(x_i, c)$. For notational convenience, alpha-convert that formula to $Q(\boldsymbol{y})$, where the y_i are the individual boolean unknowns in Q (both input and control bits). Rank-order the y_i from most to least influence, $y_{j0}, y_{j1}, \ldots$. Then pick some threshold n smaller than the length of $\boldsymbol{y}$, and concretize $y_{j0}, \ldots y_{jn}$ with randomly chosen values. Repeat until a solution is found. Notice that this algorithm parallelizes trivially by running the same procedure on different cores, stopping when one core finds a solution.

This basic algorithm is straightforward, but there remain three challenges: How to estimate the influence of an unknown, how to estimate the threshold of influence for concretization, and how to deal with uncertainty in those estimates. We discuss these challenges in the next two sections.

3 Influence and Degree of Concretization

An ideal measure of an unknown's influence would model its exact effect on running time, but there is no practical way to compute this. As we saw in the previous section, a reasonable alternative is to estimate how much we expect the constraint graph to shrink if we concretize a given node. However, it is still expensive to actually perform substitution and simplification.

Our solution is to use a more myopic measure of influence, focusing on the immediate neighborhood of the unknown rather than the full graph. Following the intuition from Section 2, our goal is to assign high influence to unknowns that select among alternative program fragments (i.e., used as guards of conditions), and to give low influence to unknowns in arithmetic operations. For an unknown n, we define $\mathit{influence}(n) = \sum_{d \in \mathit{children}(n)} \mathit{benefit}(d)$, where $\mathit{children}(n)$ is the set of all nodes that depend directly on n. Here $\mathit{benefit}(d)$ estimates how much concretizing n may simplify d, and it is defined by case analysis of d:

– *Choices.* If d is an ite node,[3] there are two possibilities. If n is d's guard (i.e., $d = \mathrm{ite}(n, a, b)$), then $\mathit{benefit}(d) = 1$. The intuition is that unknown guards in **if** statements will flow to guards of ite nodes, and the more they do, the more lines they control, and the higher the sum of their children's *benefit*. On the other hand, if n corresponds to one of the choices (i.e., $d = \mathrm{ite}(c, n, b)$ or $d = \mathrm{ite}(c, a, n)$), then $\mathit{benefit}(d) = 0$, since replacing n with a constant has no effect on the size of the formula for the ite node.

– *Boolean nodes.* If d is any boolean node except negation, it has benefit 0.5. The intuition is that boolean nodes are often used in conditional guards, but sometimes they are not, so they have a lower benefit contribution than ite guards. If $d = \neg(n)$, then $\mathit{benefit}(d)$ 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 $\mathit{benefit}(d) = \mathit{influence}(d)$, i.e., the benefit of concretizing the choice depends on how many nodes depend on d.

– *Arithmetic nodes.* If d is an arithmetic operation, $\mathit{benefit}(d) = -\infty$. The intuition is that unknowns involved in arithmetic operations are best left to the solver. For example, if a sketch contains an expression ??+*in*, replacing ?? with a constant will not affect the size of the formula.

Note that while the above definitions may involve recursive calls to *influence*, the recursion depth will never be more than two due to a prior simplification pass. This pass also eliminates nodes with no children, and thus any unknown not involved in arithmetic will have at least one child and thus an influence of at least 0.5.

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

3.1 Degree of Concretization

The next step is to decide the threshold for concretization. We hypothesize the best amount of concretization varies—we will test this hypothesis shortly. Moreover, since our influence computation is only an estimate, we opt to incorporate some randomness, so that (estimated) highly influential unknowns might not be concretized, and (estimated) non-influential unknowns might be.

Thus, we parameterize our algorithm by a *degree of concretization* (or just *degree*). For each unknown n in the constraint graph, we calculate its estimated influence $N = \mathit{influence}(n)$. Then we concretize the node with probability

$$p = \begin{cases} 0 & \text{if } N < 0 \\ 1.0 & \text{if } N > 4000 \\ 1/(\max(2, \mathit{degree}/N)) & \text{otherwise} \end{cases}$$

To understand this formula, ignore the first two cases, and consider what happens when *degree* 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 *degree* is 0 then all nodes have a $1/2$ chance of concretization. On the other hand, suppose *degree* 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, and at the extreme of $\mathit{degree} = \infty$, no concretization will occur, just as in regular SKETCH.

For nodes with influence above 4000, the effect on the size of the formula is so large that we always find concretization profitable. Nodes with influence below zero are those involved in arithmetic, which we never concretize.

Overall, there are four "magic numbers" in our algorithm so far: the degree cutoff 4000 at which concretization stops being probabilistic, the ceiling of $1/2$ on the probability for all other nodes, and the benefit values of 1 and 0.5 for boolean and choice unknowns, respectively. We determined these number in an ad hoc way using a subset of our benchmarks (`p_*` and `hd_*`, explained later). The results on the rest of our benchmarks suggest that they are reasonably good values, and we leave a more systematic analysis to future work.

3.2 Preliminary Experiment: Optimal Degree

We conducted a preliminary experiment to test whether the optimal degree varies from one subject program to another. We chose seven benchmarks split across three different synthesis domains. The left column of Table 1 lists the benchmarks, grouped by domain. Section 5.1 describes the programs and the machine used for the experiments in more detail. We ran each benchmark with a range of degrees varying exponentially from 16 to 4096. For each degree, we ran each benchmark 128 times, with a 30-minute timeout.

For each benchmark/degree pair, we wish to estimate the time to success if we ran the same benchmark many times at that degree. To form this estimate,

Benchmark	Degree					
	16	64	128	512	1024	4096
p_button	∞	∞	88 58	52 36	35 25	234 112
p_color	∞	26 15	41 22	22 10	24 10	15 4
p_menu	∞	∞	77 47	26 11	27 10	82 28
l_prepend	290 237	160 144	236 193	240 554	402 937	1,514 349
l_min	2,801 11,259	220 1,906	116 1,570	4,820 7,096	6,337 5,584	8,023 2,757
a_mom_1	∞	779 1,295	240 528	2,172 2,507	2,221 2,494	2,592 4,278
a_mom_2	∞	∞	∞	3,937 12,040	41,910 74,904	12,307 11,322

Table 1: Expected running time (s) using empirical success rate. SIQR in small text. Fastest time in dark grey, second-fastest in light grey.

for each such pair we compute the fraction of runs p that succeeded; this approximates the true probability of success of any given trial. Now suppose a trial takes time t. If every run takes that time, and p is the probability of success, we compute the *expected time to success* from that trial as t/p. While this is a coarse estimate, it provides a simple calculation we can also use in an algorithm (Section 4). If p is 0 (no trial succeeded), the expected time to success is ∞.

Results. The right set of columns in Table 1 shows the results. Each cell contains the median expected run time in seconds, as computed for each trial. Since variance is high, we also report the semi-interquartile range (SIQR) of the running times, shown in small text. We highlight the fastest and second-fastest times.

This table shows that the optimal degree varies across all benchmarks; indeed, all degrees chosen except 16 were optimal for at least one benchmark. We also see a lot of variance across runs. For example, for `l_min`, degree 128, the SIQR is more than 13× the median. Other benchmarks also have high SIQRs.

Importantly, if we visualize the median expected running times, they roughly form a vee around the fastest time—performance gets worse the farther away from optimal in either direction. Thus, we can *search* for an optimal degree, as we discuss in the next section. There is one noticeable outlier. Benchmark `p_color` has a few very high-impact unknowns that are concretized at every degree, making the performance of the remaining symbolic search indistinguishable.

4 Adaptive, Parallel Concretization

As we saw in the previous section, if we were to plot the expected running time versus degree, we would see roughly a valley with the optimal degree at the lowest point. Thus, we can *search* for the best degree by varying the degree and running trials to estimate the completion time at that degree, until we reach a local optimum. Figure 1 shows our search algorithm, which we discuss next.

Sampling. The core step of our algorithm, encapsulated in the **run_trial** function, is to run SKETCH with the specified degree. If a solution is found, we exit the

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

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 ← low + 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: Search Algorithm using Wilcoxon Signed-Rank Test.

search. Otherwise, we return both the time taken by that trial and the size of the concretization space, e.g., if we concretized n bits, we return 2^n. We will use this information to estimate the time-to-solution of running at this degree.

Since SKETCH solving has some randomness in it, a single trial is not enough to provide a good estimate of time-to-solution, even under our heuristic assumptions. In Table 1 we used 128 trials at each degree, but for a practical algorithm, we cannot fix a number of trials, lest we run either too many trials (which wastes time) or too few (which may give a non-useful result).

To solve this issue, our algorithm uses the *Wilcoxon Signed-Rank Test* [24] to determine when we have enough data to distinguish two degrees. We assume we have a function **wilcoxon**(*dist_a*, *dist_b*) that takes two equal-length lists of (time, concretization space size) pairs, converts them to distributions of estimated times-to-solution, and implements the test, returning a p-value indicating the probability that the means of the two distributions are different.

Recall that in our preliminary experiment in Section 3, we calculated the estimated time to success of each trial as t/p, where t was the time of the trial and p was the empirical probability of success. We use the same calculation in this algorithm, except we need a different way to compute p, since the success rate is always until we find a solution, at which point we stop. Thus, we instead

calculate p from the search space size. We assume there is only one solution, so if the search space size is s, the probability of success is $1/s$.[4] We then compute p as the average of $1/s_i$ for all search spaces in the distribution.

Comparing Degrees. Next in our algorithm, **compare** takes two degrees as inputs and returns a value indicating whether the **left** argument has lower expected running time, the **right** argument does, or it is a **tie**. The function initially creates two empty sets of trial results, *dist_a* and *dist_b*. Then it repeatedly calls **run_trial** to add a new trial to each of the two distributions (we write $x \cup\!\leftarrow y$ to mean adding y to set x). Iteration stops when the number of elements in each set exceeds some threshold *Max_dist*, or the **wilcoxon** function returns a p-value below some threshold T. Once the algorithm terminates, we return **tie** if the threshold was never reached, or **left** or **right** depending on the means.

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 the **compare** function if it does not seem to be finding any distinction. We use 0.2 for the threshold T. This is higher than a typical p-value (which might be 0.05), but recall our algorithm is such that returning an incorrect answer will only affect performance and not correctness. We leave it to future work to tune *Max_dist* and T further.

Searching for the Optimal Degree. Given the **compare** subroutine, 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 (especially in situations when our algorithm is most useful).

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 variables until it finds values such that at degree 2^{high}, search is estimated to take a longer time than at 2^{low}, i.e., making things more symbolic than *low* causes too much slowdown. 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 case. 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. (Of course this is a heuristic assumption.) 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

[4] Notice we can ignore the size of the symbolic space, since symbolic search will find a solution if one exists for the particular concretization.

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.

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 were 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.

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, which exits the algorithm. (Of course, **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.

Timeouts. Like all synthesis tools, SKETCH includes a timeout that kills a search that seems to be taking too long. Timeouts are tricky to get right, because it is hard to know whether a slightly longer run would have succeeded. Our algorithm exacerbates this problem because it runs many trials. If those trials are killed just short of the necessary time, it adds up to a lot of wasted work. At the other extreme, we could have no timeout, but then the algorithm may also waste a lot of time, e.g., searching for a solution with incorrectly concretized values.

To mitigate the disadvantages of both extremes, our implementation uses an adaptive timeout. All worker threads share an initial timeout value of one minute. When a worker thread hits a timeout, it stops, but it doubles the shared timeout value. In this way, we avoid getting stuck rerunning with too short a timeout. Note that we only increase the timeout during **climb** and **bin_search**. Once we fix the degree, we leave the timeout fixed.

5 Experimental Evaluation

We empirically evaluated adaptive concretization against a range of benchmarks with various characteristics. Compared to regular SKETCH (i.e., pure symbolic search), we found our algorithm is substantially faster in many cases; competitive in most of the others; and slower on a few benchmarks. We also compared adaptive concretization with concretization fixed at the final degree chosen by the adaption phase of our algorithm (i.e., to see what would happen if we could

guess this in advance), and we found performance is reasonably close, meaning the overhead for adaptation is not high. We measured parallel scalability of adaptive concretization of 1, 4, and 32 cores, and found it generally scales well. We also compared against the winner of the SyGuS'14 competition on a subset of the benchmarks and found that adaptive concretization performs better overall.

Throughout this section, all performance reports 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 for the experiments in Section 3.) For the pure SKETCH runs only, performance is also on 13 runs with a 2-hour timeout and 32 GB memory bound.

5.1 Benchmarks

The names of our benchmarks are listed in the left column of Table 2, with the size in the next column. The benchmarks are grouped by the synthesis application the are from. Each application domain's sketches vary in complexity, amount of symmetry, etc. We discuss the groups in order.

– 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 [11]. 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. As benchmarks, we selected three PASKET sketches that aim to synthesize parts of Java Swing that include buttons, the color chooser, and menus.
– *Data Structure Manipulation.* The second set of benchmarks is from a project aiming to synthesize provably correct data-structure manipulations [13]. Each synthesis problem consists of a program template and logical specifications describing the functional correctness of the expected program. There are two benchmarks. `l_prepend` accepts a sorted singly linked list L and prepends a key k, which is smaller than any element in L. `l_min` traverses a singly linked list via a while loop and returns the smallest key in the list.
– *Invariants for Stencils.* The next sets 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 [7]. These benchmarks involve primarily integer arithmetic and large numbers of loops.
– *SyGuS Competition.* The next sets 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 three groups 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.

Bench mark	LoC	Sketch Time (s)		Adaptive Degree	Adaptive # Trials	Adaptive Time (s)		Non-Adaptive # Trials	Non-Adaptive Time (s)	
p_button	3,430	timeout		128	958	**102**	83	554	40	12
p_color	3,188	**13**	0	96	731	49	12	206	15	12
p_menu	4,093	OOM		128	658	**79**	38	1,863	169	59
l_prepend	708	102	10	16	32	**20**	6	133	23	2
l_min	795	501	104	16	170	**75**	61	409	70	46
a_mom_1	229	**224**	19	96	192	282	136	561	254	44
a_mom_2	231	**900**	94	640	1,109	2,641	725	599	1,460	135
ar_s_4	313	5	0	16	27	5	0	29	3	0
ar_s_5	334	8	2	16	30	**5**	0	36	15	2
ar_s_6	337	15	2	32	49	**12**	8	31	16	2
ar_s_7	322	**39**	14	64	86	87	10	37	56	8
ar_sum	328	263	149	16	28	**43**	14	89	41	14
hd_13_d5	310	85	15	16	20	**6**	0	20	7	0
hd_14_d1	304	98	75	16	28	**8**	8	60	10	2
hd_14_d5	329	691	428	16	6	**161**	103	16	226	39
hd_15_d5	329	1,001	348	16	11	**127**	80	18	134	64
s_cg	124	62	8	16	32	**35**	8	500	39	6
s_log2	49	858	314	64	92	**389**	578	98	119	118
s_logcnt	30	92	110	16	27	**16**	10	110	23	10
s_rev	136	359	250	128	139	**246**	128	81	22	10
deriv2	1,444	18	1	16	20	**7**	1	13	5	0
deriv3	1,410	21	1	16	9	**10**	2	8	5	0
deriv4	1,410	11	0	16	14	**8**	0	18	5	0
deriv5	1,410	12	1	16	21	**8**	1	15	5	0
q_noti	262	17	5	16	32	**11**	4	246	10	0
q_serv	2,005	1,947	1,250	16	28	**27**	6	46	44	10

Table 2: Comparing Sketch, adaptive, and non-adaptive concretization.

5.2 Performance Results

The right columns of Table 2 show our results. The columns that include running time are greyed for easy comparison, with the semi-interquartile range (SIQR) in a small font. (We only list the running times' SIQR to save space.) The first grey column lists Sketch's running time on one core. The next group of columns reports on adaptive concretization, run on 32 cores. The first column in the group gives the median of the final degrees chosen by adaptive concretization. The next column lists the median number of calls to **run_trial**. The last column lists the median running time. Lastly, the right group of columns shows the performance of our algorithm on 32 cores, assuming we skip the adaptation step and jump straight to running with the median degree shown in the table. For example, for p_button, these columns report results for running starting with degree 128 and never changing it. We again report the number of trials and the running time.

Comparing SKETCH and adaptive concretization, we find that adaptive concretization typically performs better. In the figure, we boldface the fastest time between those two columns. We see several significant speedups: 72× for `q_serv`, 14× for `hd_13_d5`, 12× for `hd_14_d1`, 8× for `hd_15_d5`, 7× for `l_min`, and 6× for `ar_sum` and `s_logcnt`. For `p_button`, regular SKETCH reaches the 2-hour timeout in 7 of 13 runs, while our algorithm succeeds, mostly within 2 minutes. In another case, `p_menu`, SKETCH reliably exceeds our 32GB memory bound and then aborts. Overall, adaptive concretization performed better in 21 of 26 benchmarks, and performed equally well on one benchmark (`ar_s_4`).

Of the remaining benchmarks, on two (`a_mom_1` and `ar_s_7`), adaptive concretization's performance was within a factor of approximately two. For `p_color`, adaptive concretization is 4× slower. Investigating further, the slowdown occurs because the degree search process gets trapped in a local minimum (96 as the final degree) and does not reach the optimum (at least 4096, from Table 1). Finally, `ar_mom_2` is 3× slower. In this case, SKETCH's synthesis phase is extremely fast, hence parallelization has no benefit. Instead, the running time is dominated by the checking phase (when the candidate solution is checked symbolically against all possible inputs), and using adaptive concretization only adds overhead.

Next we comparing adaptive concretization to non-adaptive concretization at the final degree. In 11 cases, the adaptive algorithm is actually faster, due to random chance. In 14 cases, the adaptive algorithm is within a factor of approximately three of non-adaptive concretization, which is reasonably close. In the remaining case, `s_rev`, the optimal is 11× faster. We investigated further and found this is due to a synchronization issue. Our implementation has a barrier after each round of parallel trials. The larger the degree, the more likely a single trial takes longer, which due to the barrier increases the time for all parallel trials in that round. This is in contrast to the solution-searching phase, where our algorithm exits immediately upon finding a solution in any parallel thread. We believe the barrier can be eliminated with more engineering effort.

5.3 Parallel Scalability and Comparison to SyGuS Solvers

We next measured how adaptive concretization's performance varies with the number of cores, and compare it to the winner of the SyGuS competition. Table 3 shows the results. The first two columns are the same as Table 2. The next three columns show the performance of adaptive concretization on 1, 4, and 32 cores. (The 32-core column is the same as Table 2.) We discuss the rightmost column shortly. We boldface the fastest time among SKETCH, 1, 4, and 32 cores.

These results show that, in the one-core experiments, adaptive concretization performs better than regular SKETCH in 17 of 26 cases. Although adaptive concretization is worse in the other cases, due to its extra overhead, its performance improves with the number of cores. The 4-core runs are consistently better than 1-core runs (with only one exception); the speedup ranges from 1.01× to 5×. When the number of cores increases to 32, we see the best performance in 18 of the 26 cases, with a speedup over 4-core runs ranging from 1× to 9×.

Bench mark	SKETCH Time (s)		**# Cores** (Time (s)) **1**		**4**		**32**		**Enum** Time(s)	
p_button	timeout		336	534	118	65	**102**	83		
p_color	**13**	0	142	45	107	38	49	12		
p_menu	OOM		1,228	688	439	624	**79**	38		
l_prepend	102	10	35	41	34	16	**20**	6		
l_min	501	104	395	244	295	156	**75**	61		
a_mom_1	**224**	19	927	394	426	90	282	136		
a_mom_2	**900**	94	timeout		2,094	556	2,641	725		
ar_s_4	5	0	4	2	**2**	0	5	0	1,804	44
ar_s_5	8	2	7	2	**5**	0	**5**	0	∞	
ar_s_6	15	2	16	6	**9**	3	12	8	∞	
ar_s_7	**39**	14	90	36	89	35	87	15	∞	
ar_sum	263	149	285	150	141	94	**43**	14	∞	
hd_13_d5	85	15	10	4	6	1	**6**	0	8	0
hd_14_d1	98	75	22	16	10	2	**8**	8	8	0
hd_14_d5	691	428	622	448	381	184	**161**	103	201	1
hd_15_d5	1,001	348	783	398	157	404	**127**	80	424	13
s_cg	62	8	50	14	85	40	**35**	8		
s_log2	858	314	901	948	790	820	**389**	578		
s_logcnt	92	110	236	418	144	238	**16**	10		
s_rev	359	250	232	124	**164**	50	246	128		
deriv2	18	1	13	4	**4**	4	7	1		
deriv3	21	1	13	8	**6**	2	10	2		
deriv4	11	0	5	1	**3**	0	8	0		
deriv5	12	1	6	2	**3**	0	8	1		
q_noti	17	5	19	6	12	2	**11**	4		
q_serv	1,947	1,250	73	22	29	20	**27**	6		

Table 3: Parallel scalability of adaptive concretization.

There are a few cases where four cores is faster than 32. The most noticeable are a_mom_2 and s_rev.We further investigated these benchmarks, and found the same synchronization issue mentioned above causes this issue.

SyGuS Benchmarks and Solvers. The rightmost column of Table 3 shows the performance of the Enumerative CEGIS Solver, which won the SyGuS'14 Competition [1]. As the Enumerative Solver does not accept problems in SKETCH format, we only compare on benchmarks from the competition (which uses the SyGuS-IF format, which is easily translated to a sketch). For all the nine benchmarks from the competition, adaptive concretization is actually faster. It is also worth mentioning that the Enumerative Solver actually won on the four benchmarks beginning with hd_, but our experiments show that adaptive concretization outperforms it on these benchmarks, when 32 cores are available.

6 Related Work

There have been many recent successes in sampling-based synthesis techniques. For example, Schkufza et al. use sampling-based synthesis for optimization [14,

15], and Sharma et al. use similar techniques to discover complex invariants in programs [16]. 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 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 [8, 17, 10], and lattice search has been used in synchronization of concurrent data structures [23] 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 the enumerative solver compared against in Section 5.

Recently, some researchers have combined symbolic reasoning with sampling based procedures, but mostly in the context of leveraging symbolic reasoning to improve sampling. For example, Chaudhuri et al. have shown how to use numerical search to do synthesis by applying a symbolic smoothing transformation [5, 4]. 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]. We leave as future work exploring the tradeoffs between these different approaches.

Finally, there has been significant interest in parallelizing SAT/SMT solvers. The most successful of these combine a portfolio approach—where solvers are run in parallel with different heuristics—with clause sharing [9, 25]. Interestingly, these solvers have proven to be more efficient than solvers like PSATO [26] 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 like `p_menu` 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.

7 Conclusion

We introduced adaptive concretization, a program synthesis technique that combines explicit and symbolic search. Our key insight is that not all unknowns are equally important with respect to solving time. 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* is hard to compute, we presented an online algorithm that uses exponential hill climbing and binary search to find a suitable degree by running many trials. We implemented our algorithm for SKETCH and ran it on a suite of 26 benchmarks across several different domains. We found that adaptive concretization often outperforms SKETCH, sometimes very significantly. We also found that the parallel scalability of our algorithm is reasonable.

References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. pp. 1–17 (2013), http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6679385
2. Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U., Amarasinghe, S.P.: Opentuner: an extensible framework for program autotuning. In: International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014. pp. 303–316 (2014), http://doi.acm.org/10.1145/2628071.2628092
3. Chaganty, A., Nori, A.V., Rajamani, S.K.: 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. pp. 153–160 (2013), http://jmlr.org/proceedings/papers/v31/chaganty13a.html
4. Chaudhuri, S., Clochard, M., Solar-Lezama, A.: 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. pp. 207–220 (2014), http://doi.acm.org/10.1145/2535838.2535859
5. Chaudhuri, S., Solar-Lezama, A.: 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. pp. 279–291 (2010), http://doi.acm.org/10.1145/1806596.1806629
6. Cheung, A., Solar-Lezama, A., Madden, S.: 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. pp. 3–14 (2013), http://doi.acm.org/10.1145/2462156.2462180
7. Gaudin, W., Mallinson, A., Perks, O., Herdman, J., Beckingsale, D., Levesque, J., Jarvis, S.: Optimising hydrodynamics applications for the cray xc30 with the application tool suite. The Cray User Group pp. 4–8 (2014)
8. Gulwani, S.: 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. pp. 317–330 (2011), http://doi.acm.org/10.1145/1926385.1926423
9. Hamadi, Y., Jabbour, S., Sais, L.: Manysat: a parallel SAT solver. JSAT 6(4), 245–262 (2009), http://jsat.ewi.tudelft.nl/content/volume6/JSAT6_12_Hamadi.pdf
10. Harris, W.R., Gulwani, S.: 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. pp. 317–328 (2011), http://doi.acm.org/10.1145/1993498.1993536
11. Jeon, J., Qiu, X., Foster, J.S., Solar-Lezama, A.: Pasket: Synthesizing framework models with design patterns, under submission
12. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1. pp. 215–224. ICSE '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1806799.1806833

13. Qiu, X., Solar-Lezama, A.: Synthesizing data-structure manipulations with natural proofs, under submission
14. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013. pp. 305–316 (2013), http://doi.acm.org/10.1145/2451116.2451150
15. Schkufza, E., Sharma, R., Aiken, A.: 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. p. 9 (2014), http://doi.acm.org/10.1145/2594291.2594302
16. Sharma, R., Aiken, A.: 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. pp. 88–105 (2014), http://dx.doi.org/10.1007/978-3-319-08867-9_6
17. Singh, R., Gulwani, S.: Synthesizing number transformations from input-output examples. In: Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. pp. 634–651 (2012), http://dx.doi.org/10.1007/978-3-642-31424-7_44
18. Singh, R., Gulwani, S., Solar-Lezama, A.: 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. pp. 15–26 (2013), http://doi.acm.org/10.1145/2462156.2462195
19. Solar-Lezama, A.: Program sketching. International Journal on Software Tools for Technology Transfer 15(5-6), 475–495 (2013)
20. Solar-Lezama, A., Jones, C.G., Bodik, R.: Sketching concurrent data structures. In: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation. pp. 136–148. PLDI '08 (2008)
21. Torlak, E., Bodík, R.: 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. p. 54 (2014), http://doi.acm.org/10.1145/2594291.2594340
22. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M.K., Alur, R.: 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. pp. 287–296 (2013), http://doi.acm.org/10.1145/2462156.2462174
23. Vechev, M.T., Yahav, E.: 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. pp. 125–135 (2008), http://doi.acm.org/10.1145/1375581.1375598
24. Wilcoxon, F.: Individual Comparisons by Ranking Methods. Biometrics Bulletin 1(6), 80–83 (1945)
25. Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: A concurrent portfolio approach to SMT solving. In: Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. pp. 715–720 (2009), http://dx.doi.org/10.1007/978-3-642-02658-4_60
26. Zhang, H., Bonacina, M.P., Hsiang, J.: Psato: A distributed propositional prover and its application to quasigroup problems. J. Symb. Comput. 21(4-6), 543–560 (Jun 1996), http://dx.doi.org/10.1006/jsco.1996.0030