Adaptive Concretization for Parallel Program Synthesis

Jinseong Jeon¹, Xiaokang Qiu², Armando Solar-Lezama², and Jeffrey S. Foster¹

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 evaluated it on a range of benchmarks. We found adaptive concretization is very effective, outperforming Sketch in 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, 14, 20].

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].

* Supported in part by NSF CCF-1139021, CCF-1139056, CCF-1161775, and the partnership between UMIACS and the Laboratory for Telecommunication Sciences.
In this paper, we propose 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 2, 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 [19], although we believe the technique can be readily applied to other symbolic synthesis systems such as Brahma [12] or Rosette [21].

Combining symbolic and explicit search requires solving two 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. At degree 1, unknowns are concretized with high probability; at degree $\infty$, the probability drops to zero. The degree of concretization poses its own challenge: a preliminary experiment showed that across seven benchmarks and six degrees, there is a different optimal degree for almost every benchmark. (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 to no longer be 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. Thus a key benefit of our technique is that, by exploiting parallelism on big machines, it can solve otherwise intractable synthesis problems. (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 twelve thousand times across all benchmarks, we are able to present a detailed assessment of its performance characteristics. We found our algorithm outperforms SKETCH on 23 of 26 benchmarks, sometimes achieving significant speedups of $3 \times$ up to $14 \times$. In one case, adaptive concretization succeeds where SKETCH 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 is competitive with or outperforms the winner. (Section 5 presents our results in detail.)

2 Combining Symbolic and Explicit Search

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

\[
\begin{align*}
\text{bit}[32] \ foo(\text{bit}[32] \ x) \ &\text{implements spec} \{ \\
\text{if}(??)\{ \\
\quad \text{return} \ x \ & \ ??; \ // \ unknown \ m1 \\
\text{else} \{ \\
\quad \text{return} \ x \ | \ ??; \ // \ unknown \ m2 \\
\} \}
\end{align*}
\]

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 $(x - (x \mod 8))$.

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 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 lets us solve the problem in 32ms (or 30ms if in parallel), which is 35% faster than pure symbolic search. 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 $\text{foo}(x) = x - (x \mod 8)$ for the function $\text{foo}$ described by $c$. 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 $\land_{i \in E} Q(x_i, c)$, 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., [12, 21, 22]).

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, where each node corresponds to an unknown or an operation in the theory of booleans, integer arithmetic, or arrays, and where common sub-trees are shared (see [19] for more details). For the simple example above, the formula $Q(x, c)$ has 488 nodes and CEGIS takes 12 iterations. On each iteration, the algorithm concretizes $x_i$ and simplifies the formula to 195 nodes. In contrast, when we concretize the condition, $Q(x, c)$ shrinks 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 that 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 the original formula $Q(x, c)$ produced by symbolic execution over the sketch. The unknown $c$ is actually a vector of unknowns $c_i$, each corresponding to a different hole in the sketch. First, rank-order the $c_i$ from most to least influence, $c_{j0}, c_{j1}, \cdots$. Then pick some threshold $n$ smaller than the length of $c$, and concretize $c_{j0}, \cdots, c_{jn}$ with randomly chosen values. Run the previously described CEGIS algorithm over this partially concretized formula, and if a solution cannot be found, repeat the process with a different random assignment. 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 three challenges remain: 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 (e.g., used as guards of conditions),
and to give low influence to unknowns in arithmetic operations. For an unknown
\( n \), we define
\[
\text{influence}(n) = \sum_{d \in \text{children}(n)} \text{benefit}(d, n),
\]
where \( \text{children}(n) \) is the set of all nodes that depend directly on \( n \). Here \( \text{benefit}(d, n) \) 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,\(^3\) there are two possibilities. If \( n \) is \( d \)'s guard (\( d = \text{ite}(n, a, b) \)) then \( \text{benefit}(d, n) = 1 \), since replacing \( a \) with a constant will cause the formula to shrink by at least one node. On the other hand, if \( n \) corresponds to one of the choices (\( d = \text{ite}(c, n, b) \) or \( d = \text{ite}(c, a, n) \)), then \( \text{benefit}(d) = 0 \), since replacing \( n \) with a constant has no effect on the size of the formula.

- **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 \( d = \overline{(n)} \), then \( \text{benefit}(d, n) \) equals \( \text{influence}(d) \), 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 \( \text{benefit}(d, n) = \text{influence}(d) \), i.e., the benefit of concretizing the choice depends on how many nodes depend on \( d \).

- **Arithmetic nodes.** If \( d \) is an arithmetic operation, \( \text{benefit}(d, n) = -\infty \). The intuition is that these unknowns are best left to the solver. For example, given \( ?? + m \), replacing ?? with a constant will not affect the size of the formula.

Note that while the above definitions may involve recursive calls to \( \text{influence} \), the recursion depth will never be more than two due to prior simplifications. 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.

Before settling on this particular influence measure, we tried a simpler approach that attempted to concretize holes that flow to conditional guards, with

\(^3\) \text{ite}(a, b, c) \) corresponds to \textbf{if}(a) b \textbf{else} c, as in SMT-LIB.
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.

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. More-
over, 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 \( N = \text{influence}(n) \). Then we concretize the node with probability

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

To understand this formula, ignore the first two cases, and consider what hap-
pens 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 concretiza-
tion. 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 degree = \( \infty \), no concretization will occur, just as in regular SKETCH.

For nodes with influence above 1500, 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 1500 at which concretization stops being probabilistic, the ceiling of 0.5 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. For example, the 0.5 probability ceiling
is the first thing we tried, and it worked well. On the other hand, we initially
tried probability 0 for boolean unknowns, but found that some booleans also
indirectly control choices; so we increased the benefit to 0.5, which seems to
work well. 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 with subject program. We chose seven benchmarks across three different
Table 1: Expected running time (s) using empirical success rate. SIQR in small text. Fastest time in dark grey, second-fastest in light grey.

| Benchmark | 16 | 64 | 128 | 512 | 1024 | 4096 |
|-----------|----|----|-----|-----|------|------|
| p_button  | ∞  | ∞  | ∞   | 22  | 18   | 60   |
| p_color   | ∞  | ∞  | 23  | 8   | 10   | 3    |
| p_menu    | ∞  | ∞  | ∞   | 31  | 31   | 4    |
| lprepend  | 77 | 62 | 116 | 94  | 94   | 179  |
| l_min     | ∞  | 23 | 274 | 59  | 2,538| 1,440|
| a_mom_1   | ∞  | 1,176 | 620 | 308 | 1,376| 1,655|
| a_mom_2   | ∞  | ∞  | 19,262| 5,020| 9,610| 22,468|

Results. Each cell in Table 1 contains the median expected run time in seconds, as computed for each degree. 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.

The table shows that the optimal degree varies across all benchmarks; indeed, all degrees except 1024 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 40× the median. Other benchmarks also have high SIQRs. Importantly, if we visualize the median expected running times, they 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 next.

4 Adaptive, Parallel Concretization

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. If a solution is found, we exit the 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 256 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 \texttt{wilcoxon}(\texttt{dist}_a, \texttt{dist}_b) that takes two equal-length lists of \((\text{time}, \text{concretization space size})\) 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 0 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 \(s\), we calculate \(p = 1/s\).\(^4\)

\[^4\] Notice we can ignore the size of the symbolic space, since symbolic search will find a solution if one exists for the particular concretization.
Comparing Degrees. Next, \texttt{compare} takes two degrees as inputs and returns a value indicating whether the \texttt{left} argument has lower expected running time, the \texttt{right} argument does, or it is a \texttt{tie}. The function initially creates two empty sets of trial results, \texttt{dist\_a} and \texttt{dist\_b}. Then it repeatedly calls \texttt{run\_trial} to add a new trial to each of the two distributions (we write \( x \cup y \) to mean adding \( y \) to set \( x \)). Iteration stops when the number of elements in each set exceeds some threshold \( \texttt{Max\_dist} \), or the \texttt{wilcoxon} function returns a \( p \)-value below some threshold \( T \). Once the algorithm terminates, we return \texttt{tie} if the threshold was never reached, or \texttt{left} or \texttt{right} depending on the means.

In our experiments, we use \( 3 \times \max(8,|\text{cores}|) \) for \( \texttt{Max\_dist} \). Thus, \texttt{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 \texttt{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 \( \texttt{Max\_dist} \) and \( T \) further.

Searching for the Optimal Degree. Given the \texttt{compare} subroutine, we can implement the search algorithm. The entry point is \texttt{main}, shown in the lower-right corner of Figure 1. There are two algorithm phases: an exponential climbing phase (function \texttt{climb}) in which we try to roughly bound the optimal degree, followed by a binary search (function \texttt{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 \texttt{climb} function aims to return a pair \texttt{low}, \texttt{high} such that the optimal degree is between \( 2^{\text{low}} \) and \( 2^{\text{high}} \). It begins with \texttt{low} and \texttt{high} as 0 and 1, respectively. It then increases both variables until it finds values such that at degree \( 2^{\text{high}} \), search is estimated to take a longer time than at \( 2^{\text{low}} \), i.e., making things more symbolic than \texttt{low} causes too much slowdown. Notice that the initial trials of the \texttt{climb} will be extremely fast, because almost all variables will be concretized.

To perform this search, \texttt{climb} repeatedly calls \texttt{compare}, passing in \( 2 \) to the power of \texttt{low} and \texttt{high} as the degrees to compare. Then there are three cases. If \texttt{left} is returned, \( 2^{\text{low}} \) has better expected running time than \( 2^{\text{high}} \). Hence we assume the true optimal degree is somewhere between the two, so we return them. Otherwise, if \texttt{right} is returned, then \( 2^{\text{high}} \) is better than \( 2^{\text{low}} \), so we shift up to the next exponential range. Finally, if it is a \texttt{tie}, then the range is too narrow to show a difference, so we widen it by leaving \texttt{low} alone and incrementing \texttt{high}.

We also terminate climbing if \texttt{high} exceeds some maximum exponent \( \texttt{Max\_exp} \). In our implementation, we choose \( \texttt{Max\_exp} \) as 14, since for our subject programs this makes runs nearly all symbolic.

After finding rough bounds with \texttt{climb}, we then continue with a binary search. Notice that in \texttt{bin\_search}, \texttt{low} and \texttt{high} are the actual degrees, whereas in \texttt{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
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 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.

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.\(^5\) Compared to regular SKETCH (i.e., pure symbolic
search), we found our algorithm is substantially faster in many cases; competi-
tive 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, mean-
ing 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

\(^5\) Our testing infrastructure, benchmarks, and raw experimental data are open-sourced
and explained at: http://plum-umd.github.io/adaptive-concretization/.
of the benchmarks and found that adaptive concretization is better than the winner on 6 of 9 benchmarks and competitive on the remaining benchmarks.

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 they 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_prepending 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 | Adaptive | Non-Adaptive |
|------------|-----|--------|----------|--------------|
| p_button   | 3,436 | 50 ∞ | 4,160 | 51 8 249 21 6 |
| p_color    | 3,194 | 13 0 | 3,072 | 551 33 6 109 12 4 |
| p_menu     | 4,099 | 00N | 5,120 | 752 84 18 207 31 10 |
| l_prepend  | 708 | 96 8 | 32 98 20 4 110 25 4 |
| l_min      | 795 | 810 | 512 | 153 59 40 17 28 6 |
| a_mom_1    | 229 | 336 35 | 256 316 274 76 331 285 16 |
| a_mom_2    | 231 | 1,000 56 | 2,048 | 383 1,517 254 303 1,422 89 |
| ar_s_4     | 313 | 6 1 | 16 18 3 0 25 3 0 |
| ar_s_5     | 334 | 9 0 | 16 17 4 0 29 4 1 |
| ar_s_6     | 337 | 17 2 | 32 23 6 0 33 9 1 |
| ar_s_7     | 322 | 63 8 | 64 84 50 12 35 59 8 |
| ar_sum     | 328 | 618 282 | 16 11 50 19 27 31 8 |
| hd_13_d5   | 310 | 88 47 | 16 3 11 2 7 8 0 |
| hd_14_d1   | 304 | 156 41 | 16 5 29 12 10 22 7 |
| hd_14_d5   | 329 | 1,294 388 | 16 23 229 47 16 239 83 |
| hd_15_d5   | 329 | 733 370 | 32 9 177 16 6 213 39 |
| s_cg       | 124 | 18 4 | 64 161 15 2 160 14 0 |
| s_log2     | 49 | 971 314 | 128 114 349 105 25 131 89 |
| s_logcnt   | 30 | 225 236 | 32 90 21 26 84 16 6 |
| s_rev      | 136 | 327 202 | 256 165 53 22 11 43 14 |
| deriv2     | 1,444 | 28 4 | 16 15 8 0 20 8 2 |
| deriv3     | 1,410 | 28 2 | 32 9 7 1 10 7 0 |
| deriv4     | 1,410 | 14 0 | 16 7 6 0 20 5 0 |
| deriv5     | 1,410 | 14 1 | 16 7 6 0 19 5 0 |
| q_noti     | 262 | 12 6 | 32 100 8 2 79 6 3 |
| q_serv     | 2,005 | 105 68 | 32 11 22 2 9 23 2 |

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 median is ∞ if more than half the runs timed out, while the SIQR is ∞ if more than one quarter of the runs timed out. 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 4,160 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, ranging from $14\times$ for $l_{\text{min}}$, $12\times$ for $\text{ar}_\text{sum}$, and $11\times$ for $s_{\text{logcnt}}$ to $4\times$ for $\text{hd}_{15,0,5}$ and $\text{deriv3}$ and $3\times$ for $\text{ar}_s_{6}$ and $s_{\text{log2}}$. For $\text{p}_{\text{button}}$, regular SKETCH reaches the 2-hour timeout in 4 of 13 runs, while our algorithm succeeds, mostly within one minute. In another case, $\text{p}_{\text{menu}}$, SKETCH reliably exceeds our 32GB memory bound and then aborts. Overall, adaptive concretization performed better in 23 of 26 benchmarks, and about the same on one benchmark.

On the remaining benchmarks ($\text{p}_{\text{color}}$ and $\text{a}_{\text{mom}}_{2}$), adaptive concretization’s performance was within about a factor of two. Comparing other similarly short-running benchmarks, such as $\text{deriv4}$ and $\text{deriv5}$, where the final degree (16) was chosen very early, the degree search process needed to spend more time to reach bigger degree, resulting in the slowdown. Finally, $\text{a}_{\text{mom}}_{2}$ is $1.5\times$ 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 compare adaptive concretization to non-adaptive concretization at the final degree. In 7 cases, the adaptive algorithm is actually faster, due to random chance. In the remaining cases, the adaptive algorithm is either about the same as non-adaptive or is at worst within a factor of approximately three.

### 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 five columns show the performance of adaptive concretization on 1, 4, and 32 cores. Real time is wall-clock time for the parallel run (the 32-core real-time column is the same as Table 2), and CPU time is the cumulative SKETCH back-end time summed over all cores. We discuss the rightmost column shortly. We boldface the fastest real time among SKETCH, 1, 4, and 32 cores.

The real-time 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 or times out in the other cases, its performance improves with the number of cores. The 4-core runs are consistently close to or better than 1-core runs; in some cases, benchmarks that time out on 1 core succeed on 4 cores. At 32 cores, we see the best performance in 20 of the 26 cases, with a speedup over 4-core runs ranging up to $7\times$. There is only one case where 4 cores is faster than 32: $\text{a}_{\text{mom}}_{2}$. However, as the close medians and large SIQR indicate, this is noise due to randomness in SKETCH.

Comparing real times and CPU time, we can see that our algorithm does fully utilize all cores. Investigating further, we found one source of overhead is that each trial re-loads its input file. We plan to eliminate this cost in the future by only reading the input once and then sharing the resulting data structure.
Finally, 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). We should note that the enumerative solver is not parallelized and may be difficult to parallelize.

Adaptive concretization is faster for 6 of 9 benchmarks from the competition. It is also worth mentioning the Enumerative Solver actually won on the four benchmarks beginning with hd_. Our results show that adaptive concretization outperforms it on one benchmark and is competitive on the others.

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) tech-
niques, 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, 10, 17], 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 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]. We leave exploring the tradeoffs between these approaches as future work.

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 [9, 25]. Interestingly, these solvers are 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 because we cannot distinguish not finding a solution from having made incorrect guesses during concretization.

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 auto-tuning. 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.: Synthesizing Framework Models for Symbolic Execution, 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., Bodik, 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