Optimizing Synthesis with Metasketches

James Bornholt Emina Torlak Dan Grossman Luis Ceze

University of Washington, USA

{bornholt, emina, djg, luisceze}@cs.washington.edu

Abstract

Many advanced programming tools—for both end-users and expert developers—rely on program synthesis to automatically generate implementations from high-level specifications. These tools often need to employ tricky, custom-built synthesis algorithms because they require synthesized programs to be not only correct, but also *optimal* with respect to a desired cost metric, such as program size. Finding these optimal solutions efficiently requires domain-specific search strategies, but existing synthesizers hard-code the strategy, making them difficult to reuse.

This paper presents *metasketches*, a general framework for specifying and solving optimal synthesis problems. Metasketches make the search strategy a part of the problem definition by specifying a fragmentation of the search space into an ordered set of classic sketches. We provide two cooperating search algorithms to effectively solve metasketches. A global optimizing search coordinates the activities of local searches, informing them of the costs of potentially-optimal solutions as they explore different regions of the candidate space in parallel. The local searches execute an incremental form of counterexample-guided inductive synthesis to incorporate information sent from the global search. We present SYNAPSE, an implementation of these algorithms, and show that it effectively solves optimal synthesis problems with a variety of different cost functions. In addition, metasketches can be used to accelerate classic (non-optimal) synthesis by explicitly controlling the search strategy, and we show that SYNAPSE solves classic synthesis problems that state-of-the-art tools cannot.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program synthesis

Keywords Program synthesis

1. Introduction

Program synthesis is the classic problem of automatically producing an implementation from a high-level correctness specification. Recent research efforts have addressed this problem successfully for a variety of application domains, from browser layout [23] to executable biology [17]. But for many applications, such as synthesis-aided compilation [27, 31] or end-user programming [10, 11], it is not enough to produce *any* correct program. These applications require the synthesized implementation to also be *optimal* with respect to a desired cost function—for example, the number of instructions or the sum of their latencies.

Optimal synthesis involves producing a program that is both correct with respect to a (logical) specification and optimal with respect to a cost function. Existing tools for optimal synthesis are highly specialized, employing custom search strategies to quickly find the best solution in a large space of candidate programs. For example, a superoptimizer [16, 31] finds the least expensive instruction sequence (according to a cost model) equivalent to a given reference implementation. To make this task tractable, a superoptimizer must be able to focus its search on candidate programs cheaper than the currently-optimal solution. Building such a tool on top of existing synthesizers is impractical, because they provide no means for the tool to guide or control the search strategy. Instead, tool developers are forced to implement their own synthesis engines from scratch, giving up the potential to benefit from advances in general synthesis technology.

In this paper, we present *metasketches*, a general framework for specifying and solving optimal synthesis problems. A metasketch is an ordered set of *sketches* [35], together with a *cost function* to minimize and a *gradient function* to direct the search. A sketch is a syntactic template that defines a finite space of candidate programs. The union of the (possibly overlapping) sketches describes the candidate space of the metasketch. The ordered set of sketches, together with the cost and gradient functions, expresses a high-level search strategy: the sketches fragment the candidate space into regions that can be explored in parallel, while the sketch ordering and the two functions focus the search toward cheaper regions of the space. Because a metasketch consists of a set of classic sketches, solving a metasketch reduces to solving a set of classic synthesis problems, and so tools built with metasketches benefit from progress in the underlying synthesis techniques.

To solve an optimal synthesis problem expressed as a metasketch, we employ two cooperating algorithms: a global optimizing search over the entire candidate space, and many parallel instances of a local combinatorial search over the individual sketches in the metasketch. The local search algorithm implements an incremental form of counterexample-guided inductive synthesis (CEGIS). The global search drives this incremental local exploration in two ways. First, it uses the gradient function to select which sketches to explore locally whenever a satisfying solution (and therefore a tighter upper bound on the cost) is found. The gradient function is simple: given a numerical cost, it returns the set of all sketches from the metasketch that (may) contain a cheaper candidate program. Second, the global search communicates the cost of discovered solutions to all running local searches, which integrate these results to prune their own candidate spaces. The search process proceeds until an optimal solution is found or the global search space is exhausted. This search strategy is highly effective, solving both classic and optimal synthesis problems that cannot be solved by existing techniques.

POPL '16, January 20–22, 2016, St. Petersburg, FL, USA.

ACM 978-1-4503-3549-2/16/01…$15.00.
http://dx.doi.org/10.1145/2837614.2837666

In addition to enabling efficient search, metasketches also bring new expressive power to syntax-guided synthesis. By representing the search space as a set of sketches, a metasketch can describe candidate spaces such as "the set of all programs, of any length, that are in static single assignment (SSA) form and that contain no unused variables." A space of this form cannot be expressed with a single sketch (because it is infinite) nor can it be expressed with a context-free grammar (because neither the SSA nor the used-variable constraints are context-free). A set-of-sketches space description also supports an effective new form of encoding optimization, which we call *structure constraints*. These constraints take the form of assertions within an individual sketch, which rule out candidates that are semantically equivalent to cheaper programs from other sketches. The resulting metasketch avoids redundant work in the local searches, thus accelerating the global synthesis process.

We have implemented our optimal synthesis approach in a tool called SYNAPSE, built on top of the ROSETTE language [39, 40]. We have used SYNAPSE to develop and solve metasketches for a variety of optimal synthesis problems, from superoptimization to fixed-point approximation of computational kernels. Our experiments show that SYNAPSE is not only effective at solving optimal synthesis problems, but that it can also solve standard synthesis benchmarks [3] that are intractable for state-of-the-art syntax-guided synthesizers (due to their large, monolithic search spaces that we fragment with metasketches). The fragmented search space exposed by a metasketch also allows SYNAPSE to realize significant parallel speedup for large synthesis problems. We find that for many optimal synthesis problems, the search algorithm spends most of its time *proving* optimality of the final candidate solution, suggesting that the search can quickly output intermediate results that will *likely* be optimal. Finally, we show that SYNAPSE can effectively reason about a variety of cost functions through several examples: fitting a linear model to a training data set, optimizing worst-case execution time of a program, and training a small neural network.

In summary, this paper makes the following contributions:

- We introduce the first generic framework for optimal synthesis. The core of our framework is the metasketch abstraction, which combines a novel representation of a syntactic search space as a (countable ordered) set of finite sketches with a cost function to minimize and a gradient function to guide the search. Metasketches bring new expressive power to syntax-guided synthesis and enable effective search for optimal solutions.
- We present a new algorithm for optimal synthesis that uses metasketches to layer a parallel global search on top of a local combinatorial search. The combinatorial search is an incremental variant of counterexample-guided inductive synthesis (CEGIS). We prove that our algorithm is sound, and that it is complete and optimal for finite metasketches, as well as infinite metasketches that satisfy a simple compactness property.
- We present SYNAPSE, a prototype implementation of our approach, and evaluate it on standard benchmark suites for synthesis [3] and approximate computing [9]. Our results show that SYNAPSE can solve both classic and optimal synthesis problems that existing tools cannot solve.

The remainder of the paper is organized as follows. In Section 2, we formulate the problem of optimal syntax-guided synthesis. Section 3 introduces metasketches and presents examples of metasketches designed for superoptimization and approximate computing. We describe and characterize our synthesis algorithm in Section 4. Section 5 presents an evaluation of SYNAPSE on a diverse set of benchmarks. Section 6 discusses related work, and Section 7 concludes.

expressions	e	::= $l \mid x \mid$ (**lambda** ($x \ldots$) e) $\mid$ ($e\ e \ldots$) $\mid$ (**if** $e\ e\ e$) $\mid$ (**let*** ([$x\ e$] $\ldots$) e) $\mid$ (**assert** e)
	l	::= **true** $\mid$ **false** $\mid$ integer literal
	x	::= identifier $\mid$ = $\mid$ > $\mid$ + $\mid$ - $\mid$ * $\mid$ / $\mid$ & $\mid \ldots$
definitions	d	::= (**define** $x\ e$)
forms	f	::= $d \mid e$
programs	p	:= $f \mid p\ f$

Figure 1. Syntax of programs in the simple Scheme-like language SYN we use for examples.

2. Optimal Syntax-Guided Synthesis

This section briefly reviews syntax-guided synthesis [2, 35] and formalizes the problem of optimal syntax-guided synthesis. We also introduce a small Scheme-like synthesis language, SYN, that will be used to present (optimal) synthesis examples throughout the paper. Our approach is independent of SYN, however, and can be applied to any language that supports basic sketching constructs (such as Sketch [35] or ROSETTE [40]).

Synthesis. The program synthesis problem is to automatically discover a program P that implements a desired specification ϕ. Programs are written in a language $\mathcal{L}$, and specifications in a decidable theory $\mathcal{T}$ (or a decidable combination of theories $\mathcal{T}_i$). We assign a deterministic semantics $[\![P]\!]$ to each program $P \in \mathcal{L}$. For a set of programs $\mathcal{S} \subseteq \mathcal{L}$, we write $[\![\mathcal{S}]\!]$ to denote the set $\{[\![P]\!] \mid P \in \mathcal{S}\}$. A specification is a formula $\phi(x, [\![P]\!](x))$ in the theory $\mathcal{T}$ that relates program inputs to outputs. Given a specification ϕ, the program synthesis task is to find a program $P \in \mathcal{L}$ such that the formula $\forall x.\, \phi(x, [\![P]\!](x))$ is valid modulo $\mathcal{T}$.

Programs and Specifications. For examples in this paper, we take the language $\mathcal{L}$ to be SYN, a subset of core Scheme [29] shown in Figure 1. SYN expressions are constructed from booleans, signed finite-precision integers, lambda terms, applications, conditionals, and sequential let-binding expressions. The language also includes the usual built-in procedures for operating on booleans and integers. We take the specification theory $\mathcal{T}$ to be the quantifier-free theory of fixed-width bitvectors. For convenience, specifications can be expressed as assertions in SYN programs in the standard manner.[1] An assertion succeeds if the value of the expression argument is not **false**. The semantics of SYN is standard [29], except that all built-in integer operators also accept boolean arguments, treating **true** as 1 and **false** as 0.

Example 1. Suppose that we are trying to synthesize a SYN implementation of the max function. In the theory of bitvectors, the specification for max is straightforward:

$$\phi_{\max}(\langle x, y\rangle, [\![P]\!](x, y)) \equiv [\![P]\!](x, y) = \mathrm{ite}(x > y, x, y)$$

There are many programs in SYN that meet this specification, such as the following SSA-style implementation:[2]

```
(define (max1 i1 i2)
  (let* ([o1 (> i1 i2)]
         [o2 (if o1 i1 i2)])
    o2))
```

A program synthesizer should return `max1` or another correct implementation from SYN.

Syntax-Guided Synthesis. Syntax-guided synthesis is a form of program synthesis that restricts the search for P to a space of candidate implementations $\mathcal{C} \subseteq \mathcal{L}$ defined by a syntactic template [2]. This restriction makes the search more tractable, and it enables the

[1] In particular, SYN assertions can be reduced to formulas in the theory of bitvectors for all finite SYN programs, using existing methods [40].

[2] We write (**define** ($x\ y \ldots$) e) to abbreviate (**define** x (**lambda** ($y \ldots$) e)).

programmer to describe the desired implementation using a mix of syntactic and semantic constraints.

Syntactic constraints commonly take the form of a context-free grammar [2] or a *sketch* [35]. A sketch is a partial implementation of a program, with missing expressions called *holes* to be discovered by the synthesizer. Holes are constrained to admit expressions from a finite set of choices—for example, a hole could be replaced with a 32-bit integer constant or with an expression obtained from a finite unrolling of a context-free grammar. Unlike context-free grammars, sketches can express only finite candidate spaces $\mathcal{C}$. In return, however, they provide the programmer with more control over the shape of the search space, as well as the ability to express syntactic constraints that are not context-free.

Sketches. To enable sketching in SYN, we add a hole construct:

$$\text{expressions } e ::= \ldots \mid (\texttt{??}\ e \ldots)$$

The hole construct can be used in one of two ways. When it is applied to no expressions, `(??)`, it represents a placeholder for an integer constant. Otherwise, it is a placeholder that selects from among the provided expressions.

We call a program in SYN a sketch if it contains holes. A sketch $S \in \mathcal{L}$ defines a set of candidate programs $\overline{S}$, which is the set of all possible programs produced by replacing the holes H in P with concrete expressions. We can define the synthesis problem in terms of completing the holes: given a sketch S, the program synthesis task is to find a completion $\vec{h}$ for the holes H in S such that the formula $\forall x.\, \phi(x, [\![S[H := \vec{h}]]\!](x))$ is valid modulo $\mathcal{T}$. We abuse notation to write $[\![S]\!]$ for the set of semantics $[\![\overline{S}]\!]$ of all possible programs produced by a sketch.

Example 2. Sketches allow programmers to capture domain insights that can make synthesis more tractable. For example, a SYN sketch for `max` might specify that the last operation is always an **`if`**:

```
(define (max1-sketch i1 i2)
  (let* ([o1 ((?? > >= = < <=) (?? i1 i2) (?? i1 i2))]
         [o2 (if o1 i1 i2)])
    o2))
```

The advantage of a sketch is that the synthesizer need only discover an assignment to the holes that satisfies the specification ϕ, without having to explore all possible programs in SYN.

Optimal Syntax-Guided Synthesis. Even with syntactic constraints, there is rarely a unique solution to a given synthesis problem. Our simple `max1-sketch`, for example, has four correct solutions, and the synthesizer is free to return any one of them. But for many applications, some solutions are more desirable than others due to requirements such as program size, execution time, or memory or register usage. For these applications, the synthesis task becomes one of optimization rather than search.

We define the *optimal syntax-guided synthesis* problem as a generalization of syntax-guided synthesis. The optimal program synthesis problem is the task of searching a space of candidate programs $\mathcal{C}$ for a lowest-cost implementation P that satisfies the given specification ϕ. The search is performed with respect to a cost function κ, which assigns a numeric cost to each program $P \in \mathcal{L}$.

Definition 1 (Optimal Syntax-Guided Synthesis)**.** *Let $\mathcal{L}$ be a programming language, and $\mathcal{T}$ a decidable theory. Given a specification formula $\phi(x, [\![P]\!](x))$ in $\mathcal{T}$, a cost function $\kappa : \mathcal{L} \to \mathbb{R}$, and a search space $\mathcal{C} \subseteq \mathcal{L}$ of candidate programs, the* optimal (syntax-guided) synthesis *problem is to find a program $P \in \mathcal{C}$ such that the formula $\forall x.\, \phi(x, [\![P]\!](x))$ is valid modulo $\mathcal{T}$, and $\kappa(P)$ is minimal among all such programs.*

Note that when the cost function κ is constant, optimal synthesis reduces to syntax-guided synthesis.

To ensure that the optimal synthesis problem remains decidable, we must place restrictions on the cost function κ. Existing optimal synthesis techniques often require κ to reason only about program syntax. For our synthesis approach (Section 4), it is sufficient to require that the evaluation of κ on a program P be reducible to a term in a decidable theory $\mathcal{T}$. This allows us to encode cost functions that reason not only about program syntax but also about program semantics. For example, in Section 5.6, we demonstrate a simplified worst-case execution time metasketch, which reasons about feasible paths through the program's control flow.

Example 3. Optimal synthesis chooses among multiple correct candidate programs by minimizing a given cost function. Different cost functions will produce different optimal solutions. For example, suppose we want to find an implementation $P \in$ SYN of our $\phi_{\max}$ specification that minimizes the sum of operation costs:

programs	$\kappa(p)$	$= \sum_{f \in p} \kappa(f)$
definitions	κ ((**`define`** x e))	$= \kappa(e)$
expressions	κ ((**`if`** e_1 e_2 e_3))	$= 1 + \kappa(e_1) + \kappa(e_2) + \kappa(e_3)$
	κ ((**`let*`** $([x_i e_i] \ldots)$ e))	$= \kappa(e) + \sum_i \kappa(e_i)$
	κ ((**`lambda`** $(x \ldots)$ e))	$= \kappa(e)$
	κ ((**`assert`** e))	$= 0$
	κ ((e $e_i \ldots$))	$= \kappa(e) + \sum_i \kappa(e_i)$
	κ (x)	= 1 if x is a built-in operator = 0 otherwise

The program `max1` from Example 1 has a cost of 2, and it is an optimal solution under the cost function κ. However, suppose that we are targeting an environment where branches are expensive and to be avoided. We can update the cost function to penalize branches as follows:

$$\text{expressions} \quad \kappa\ ((\texttt{if}\ e_1\ e_2\ e_3)) = 8 + \kappa(e_1) + \kappa(e_2) + \kappa(e_3)$$

Under this cost function, the `max1` solution has a cost of 9. The new optimal solution has a cost of 4 and implements an arithmetic manipulation for the maximum of two (finite precision) integers:

```
(define (max2 i1 i2)
  (let* ([o1 (- i2 i1)]
         [o2 (<= i1 i2)]
         [o3 (* o1 o2)]
         [o4 (+ i1 o3)])
    o4))
```

Given $\phi_{\max}$, $\mathcal{C} =$ SYN, and our new cost function, an optimal synthesizer should return `max2`, or another correct program with the same cost as `max2`.

3. Metasketches

This section introduces *metasketches* (Def. 2), a new abstraction for specifying and solving optimal synthesis problems. Metasketches generalize sketches, enabling a description of an infinite space of candidate programs with a countable, ordered set of finite sketches. This representation permits fine-grained control over the shape of the candidate space, which is critical for effective search. A metasketch additionally provides a means of assigning cost to programs and of directing the search toward lower-cost regions of the candidate space. This section defines metasketches, describes their properties, and illustrates their utility for capturing insights that enable efficient search. Section 4 presents a synthesis algorithm that exploits the search strategy exposed by metasketches.

3.1 The Metasketch Abstraction

A metasketch consists of three components: (1) a *space* of candidate programs, represented as a countable, ordered set of finite sketches;

(2) a *cost* function from programs to numeric cost values; and (3) a *gradient* function from each cost value c to a set of sketches (i.e., a subspace) that may contain a program with a lower cost than c. The space component provides a way to fragment a monolithic search space into a set of finite regions that can be explored independently of one another. Because each region is described by a sketch, the programmer gains the ability (as we show later) to use context-sensitive *structure constraints* to reduce overlap between the regions—thus making both the global and local search tasks easier. The cost and gradient functions provide a way to navigate the local and global search spaces in a cost-sensitive way. Together, these components enable the programmer to easily convey key problem-specific insights to a generic search algorithm.

Definition 2 (Metasketch)**.** *A* metasketch *is a tuple $m = \langle \mathcal{S}, \kappa, g \rangle$, where:*

- *The* space *$\mathcal{S} \subseteq \mathcal{L}$ is a countable set of sketches in $\mathcal{L}$, equipped with a total ordering relation $\preceq$.*
- *The* cost function *$\kappa : \mathcal{L} \to \mathbb{R}$ assigns a cost to each program in the language $\mathcal{L}$.*
- *The* gradient function *$g : \mathbb{R} \to 2^{\mathcal{S}}$ returns an overapproximation of the set of sketches in $\mathcal{S}$ that contain programs with lower cost than a given value $c \in \mathbb{R}$:*

$$g(c) \supseteq \{S \in \mathcal{S} \mid \exists \vec{h}.\ \kappa(S[H := \vec{h}]) < c\} \tag{1}$$

Given a specification ϕ, a metasketch $m = \langle \mathcal{S}, \kappa, g \rangle$ defines an instance of the optimal program synthesis problem (Def. 1), in which the search space $\mathcal{C}$ is the union $\bigcup_{S \in \mathcal{S}} S$ of the search spaces of each sketch in the set $\mathcal{S}$, and the cost function is given by κ.

3.2 Properties of Metasketches

Metasketches bring new expressive power to (optimal) syntax-guided synthesis in two ways. First, they can express richer candidate spaces than either sketches or context-free grammars alone: unlike a classic sketch, a metasketch can capture an infinite space of candidate programs, and unlike a context-free grammar, it can express syntactic constraints on the search space that are context-sensitive (see Section 3.3 for examples). Second, unlike other forms of syntactic templates, metasketches describe both a search space and a *search strategy*.

In particular, by specifying the candidate space as an ordered set of sketches, the programmer provides a decomposition of the problem into independent parts, as well as an order in which those parts should be explored. At one extreme, if each sketch $S \in \mathcal{S}$ is a concrete program with no holes, the metasketch is an implementation of brute force search. Ordering these programs according to AST depth yields a bottom-up brute force search, a common synthesis technique [3, 42]. At the other extreme, if $\mathcal{S}$ contains only a single finite sketch, the programmer is choosing to solve the problem monolithically with the underlying synthesizer's search strategy, such as reduction to SMT [35, 39]. But there are many other search strategies between these two extremes that are also easily captured with a metasketch—for example, adaptive concretization [14] randomly replaces some holes in a sketch with concrete values, thus creating a family of sketches of roughly the same complexity (as measured by the number of holes) that are solved independently.

While the space component of a metasketch expresses a search strategy, the gradient function expresses an *optimization strategy*—essentially, a cost-based filter for the optimal synthesizer to apply to the global search space once it finds some solution. For commonly used cost functions (such as program length or the sum of instruction latencies), it is easy to provide a function that precisely determines whether a finite sketch contains a program with a cost lower than a given value. Of course, this may be difficult or impossible for more complex cost functions. For this reason, a metasketch only requires g to be an overapproximation (Equation 1): it can return sketches that have no solution with cost less than c, but to be sound, must not filter out a sketch that does have such a solution. As a degenerate case, the gradient function $g(c) = \mathcal{S}$ is a trivial overapproximation that filters out no sketches. Using a trivial gradient will not affect the correctness of the search, nor will it affect its optimality over finite spaces, but as we discuss in Section 4, a more constrained gradient is required to guarantee optimality over infinite spaces.

$$
\begin{aligned}
\mathcal{S} &= \{S_i \mid i \in \mathbb{N}^+\} \\
\preceq &= \{\langle S_i, S_j \rangle \mid i \leq j\} \\
\kappa(P) &= i \text{ for } P \in S_i \\
g(c) &= \{S_i \mid i < c\} \\
S_i &= \texttt{(lambda (}x \ldots\texttt{)} \\
&\quad \texttt{(let* ([o}_1\ \texttt{(}expr\ x \ldots\texttt{)]} \\
&\qquad \ldots \\
&\qquad \texttt{[o}_i\ \texttt{(}expr\ x \ldots\ \texttt{o}_1\ \ldots\ \texttt{o}_{i-1}\texttt{)])} \\
&\quad \texttt{o}_i\texttt{))} \\
(expr\ e \ldots) &= \texttt{(?? ((?? - ~) (?? }e \ldots\texttt{)) } \\
&\quad \texttt{((?? + - * \& ...) (?? }e \ldots\texttt{) (?? }e \ldots\texttt{)) } \\
&\quad \texttt{(if (?? }e \ldots\texttt{) (?? }e \ldots\texttt{) (?? }e \ldots\texttt{)))}
\end{aligned}
$$

Figure 2. A basic metasketch for superoptimization. This formulation defines the search space to consist of all SYN programs in SSA form. The sketches are ordered according to size. The cost function κ : SYN $\to \mathbb{N}$ measures the number of conditionals and applications of built-in operators.

3.3 Examples

We illustrate the process of creating metasketches for two optimal synthesis problems: superoptimization [12, 16, 21, 31, 43] and approximate computing [5, 9, 25]. Superoptimization is the problem of finding the optimal sequence of instructions that implements a given specification. Approximate computing allows small calculation errors in programs (such as image processing kernels) that result in lower energy expenditure or execution time. A common formulation of the approximate computing task boils down to an optimal synthesis problem, in which the cost function encodes the desired performance metric, and the specification constrains the synthesized program to be sufficiently accurate with respect to the reference program. We show three simple metasketches for these applications. Despite their simplicity, however, our metasketches capture enough insight to enable optimal synthesis of superoptimization and approximation benchmarks that cannot be solved—even ignoring optimality—with existing techniques (Section 5).

Superoptimization Figure 2 shows a basic superoptimization metasketch, designed to find a shortest program in SSA form using a given set of operators (in our case, all built-in SYN operators). The space of all SSA programs cannot be expressed with a context-free grammar, since SSA constraints are context-sensitive. However, it is easily expressed as a set of sketches. Our metasketch assumes a cost function κ that measures the number of conditionals and applications of built-in operators (that is, the original cost function from Example 3). The search space consists of all finite SSA sketches S_i with i defined variables (whose names are canonical), and the sketches are ordered according to how many defined variables they contain. For our cost function, the number of defined variables in a sketch completely determines the cost of all programs in that sketch, which is reflected in the gradient function g.

The metasketch in Figure 2 encodes a simple iterative deepening strategy for superoptimization, in which smaller search spaces (corresponding to shorter programs) are explored first. However, this encoding is inefficient: a sketch of size i includes programs that can

be trivially reduced to a shorter program by dead code elimination. As a result, any search over the space defined by the sketch S_i will explore programs that are also covered by sketches $S_j \preceq S_i$. To reduce overlap between sketches, we can amend our encoding of S_i to force all defined variables to be used at least once. To do so, we simply lift the choice of the k^{th} variable's input arguments into fresh variables $\vec{v_k}$, and add, for each o_j, the following assertion to the body of the `let*` expression: $\bigvee_{k>j, v \in \vec{v_k}} o_j = v$. We call such assertions (that only constrain the holes) *structure constraints*. In this case, the structure constraints remove from a sketch of length i (a large class of) programs that are also found in shorter sketches. We have used the resulting metasketch to solve existing synthesis benchmarks orders-of-magnitude faster than other symbolic synthesis techniques, and in many cases as fast as the winner of the 2014 syntax-guided synthesis competition [3].

Adaptive Superoptimization Superoptimization (e.g., [16, 31]) often involves richer cost functions than program length. For example, we may want to use a static cost model of operator latencies, defined by modifying the cost function κ from Example 3 to assign different weights to built-in operators and to conditionals:

$$\kappa\,((\texttt{if}\ e_1\ e_2\ e_3)) = c_{if} + \kappa(e_1) + \kappa(e_2) + \kappa(e_3)$$
$$\kappa\,(x) \qquad\qquad = c_x \text{ if } x \text{ is a built-in operator}$$

To obtain a gradient for this cost function, we simply change g from Figure 2 to $g(c) = \{S_i \mid i * c_{\min} < c\}$, where $c_{\min}$ is the lowest operator cost according to κ. The new metasketch continues to encode length-based iterative deepening, but given its cost function, we can refine the search strategy by adding a second dimension to the sketches: the set of operators that may appear in the program.

In particular, to bias the search toward exploring cheaper subspaces first, we sort the available operators according to cost, and then create a set of sketches $S_{i,j}$ where i is the length of the sketch, as in Figure 2, and j specifies the prefix of the sorted operators used to build expressions. The ordering on sketches now becomes the lexicographical order over $\langle i, j\rangle$ (although other orders are possible as well), and we add one more structure constraint to $S_{i,j}$ that forces the j^{th} operator to define at least one of the i variables, thus reducing overlap between sketches along the instruction-set dimension. The resulting adaptive superoptimization metasketch enables us to find optimal, fast approximations of image processing kernels that cannot be found tractably with other approximation techniques.

Piecewise Polynomial Approximation Typical targets for approximate computing include small computational kernels that are invoked many times by an outer loop. These kernels often perform expensive floating-point arithmetic using transcendental functions. For example, the following C code shows a kernel that computes the inverse kinematics of a robotic arm with two joints:

```
void inversek2j(float x, float y, float* th1, float* th2)  {
  *th2 = acos(((x*x) + (y*y) - 0.5) / 0.5);
  *th1 = asin((y * (0.5 + 0.5*cos(*th2)) - 0.5*x*sin(*th2)) /
              (x*x + y*y));
}
```

Existing techniques [9] for approximating kernels such as `inversek2j` rely on hardware-accelerated neural networks, limiting their usability by requiring custom-designed hardware. We present a metasketch that implements the first software-based technique for approximating these kernels successfully, using only conditionals and fixed-point addition and multiplication.

Our metasketch, shown in Figure 3, implements a piecewise polynomial approximation of a mathematical function. The candidate space is decomposed along two dimensions: the number k of pieces and the degree n of each polynomial. The sketch $S_{k,n}$ contains $k-1$ unknown branching conditions defining k pieces, and each branch contains a polynomial of degree n with unknown coefficients. Because our optimal synthesis problem involves approximating a function over a set of points sampled from its domain, the cost κ minimizes a linear combination of pieces and degree, in order to prevent overfitting to the sample set.

$$\begin{aligned}
\mathcal{S} &= \{S_{k,n} \mid k \in \mathbb{N}^+, n \in \mathbb{N}\} \\
\preceq &= \{\langle S_{k,n}, S_{u,v}\rangle \mid k < u \vee (k = u \wedge n \le v)\} \\
\kappa(P) &= a * k + b * n \text{ for } P \in S_{k,n} \\
g(c) &= \{S_{k,n} \mid a * k + b * n < c\}
\end{aligned}$$

$$S_{1,n} = \texttt{(lambda (}x_1 \ldots x_m\texttt{)}\ (poly_n\ x_1 \ldots x_m))$$

$$\begin{aligned}
S_{k>1,n} = \ &\texttt{(lambda (}x_1 \ldots x_m\texttt{)} \\
&\quad \texttt{(if }(bnd\ x_1 \ldots x_m) \\
&\qquad (poly_n\ x_1 \ldots x_m)\ \texttt{; 1st piece} \\
&\qquad \texttt{(...} \\
&\qquad\quad \texttt{(if }(bnd\ x_1 \ldots x_m) \\
&\qquad\qquad (poly_n\ x_1 \ldots x_m) \\
&\qquad\qquad (poly_n\ x_1 \ldots x_m)\ \texttt{; kth piece} \\
&\qquad\qquad \texttt{)...)))}
\end{aligned}$$

$$(bnd\ x_1 \ldots x_m) = \texttt{(and (< }x_1\ \texttt{(??)) ... (< }x_n\ \texttt{(??)))}$$

$$\begin{aligned}
(poly_n\ x_1 \ldots x_m) = \ &\texttt{(+ (* (??) (expt }x_1\ n\texttt{)) ...} \\
&\texttt{(* (??) (expt }x_m\ n\texttt{)) ...} \\
&\texttt{(* (??) (expt }x_1\ \texttt{1)) ...} \\
&\texttt{(* (??) (expt }x_m\ \texttt{1)) ...} \\
&\texttt{(??))}
\end{aligned}$$

Figure 3. A basic metasketch for piecewise polynomial approximation. The search space consists of all SYN programs that implement a piecewise polynomial function with k pieces and the maximum degree of n. The sketches are ordered lexicographically by $\langle k, n\rangle$. The cost function $\kappa : \text{SYN} \to \mathbb{Z}$ is a linear combination of k and n. The function `(expt x n)` multiplies `x` by itself `n` times.

As in the case of (adaptive) superoptimization, we can reduce overlap between the sketches in Figure 3 with structure constraints. In particular, we force some piece in every $S_{k,n}$ to include an n^{th} degree term with a non-zero coefficient, and we force the branching conditions to differ in at least one term. In other words, our constraints prevent sketches of size $\langle k, n\rangle$ from containing (a class of) programs that belong to smaller sketches after constant propagation and dead code elimination. Finally, we also break symmetries in the $S_{k,n}$ search space by ordering the constants in the branching conditions, so that the i^{th} bound for the argument x_i is no greater than its $i + 1^{\text{st}}$ bound. Our optimal synthesis algorithm solves the resulting metasketch for several standard approximation benchmarks, including `inversek2j`, for which it finds an approximation with 16% error and 35× speedup.

4. Optimal Synthesis Algorithm

Our synthesis approach takes advantage of the metasketch abstraction by layering a global search atop individual local searches running in parallel. The global search executes the high-level strategy encoded in a metasketch, and coordinates the activities of local searches to satisfy the optimality requirement. Local searches execute an incremental form of counterexample-guided inductive synthesis (CEGIS) [34], which can accept additional constraints during the inductive synthesis loop. This section presents the global and local search algorithms, characterizes their properties, and describes performance-oriented implementation details.

4.1 Global Search

To solve a metasketch $m = \langle \mathcal{S}, \kappa, g\rangle$, the global search coordinates individual solvers operating on sketches drawn from the space $\mathcal{S}$. This coordination takes two forms. First, the global search uses the ordered set $\mathcal{S}$ and the gradient function g to select which sketches

```
1: global τ                                              ▷ Number of parallel threads

2: function SYNTHESIZE(φ, m = ⟨S, κ, g⟩)
3:     V ← ∅                                             ▷ Completed sketches
4:     P*, c* ← ⊥, ∞                                     ▷ Optimal program and cost
5:     R ← TAKE(S, τ)                                    ▷ Remove first τ sketches from S
6:     for all S ∈ R do
7:         η ← VARSFORHOLES(S)                           ▷ Logical variables for holes
8:         x ← VARSFORINPUTS(S)                          ▷ Logical variables for inputs
9:         ψ ← λe.λa.φ(a, TOSMT(S[H := e](a)))
10:        ∃∀SOLVEASYNC(S, ∃η.∀x.ψ(η, x))                ▷ Start solving S
11:    while R ≠ ∅ do
12:        S, result, h ← WAITFORRESULT(R)
13:        P ← S[H := h]
14:        t ← 0                                         ▷ Number of new sketches to launch
15:        if result = SAT ∧ κ(P) < c* then              ▷ New optimal solution
16:            P*, c* ← P, κ(P)
17:            S ← g(c*)
18:            for all S ∈ R do
19:                if S ∈ S then                         ▷ Allow S to continue
20:                    φ ← λe.λa.TOSMT(κ(S[H := e])) < c*
21:                    SENDCONSTRAINT(S, φ)
22:                else                                  ▷ Prune S
23:                    KILLSOLVER(S)
24:                    R ← R \ {S}; V ← V ∪ {S}; t ← t + 1
25:        else if result = UNSAT then                   ▷ No (more) solutions to S
26:            KILLSOLVER(S)
27:            R ← R \ {S}; V ← V ∪ {S}; t ← t + 1
28:        if t > 0 then
29:            N ← TAKE(S \ (R ∪ V), t)
30:            for all S ∈ N do
31:                η ← VARSFORHOLES(S)
32:                x ← VARSFORINPUTS(S)
33:                ψ ← λe.λa.φ(a, TOSMT(S[H := e](a))) ∧
                              TOSMT(κ(S[H := e])) < c*
34:                ∃∀SOLVEASYNC(S, ∃η.∀x.ψ(η, x))
35:            R ← R ∪ N
36:    return P*
```

Figure 4. The global optimal synthesis algorithm SYNTHESIZE takes as input a specification ϕ and a metasketch $\langle \mathcal{S}, \kappa, g \rangle$, and finds a program $P \in \bigcup_{S \in \mathcal{S}} \overline{S}$ that satisfies ϕ and minimizes κ. The synthesis runs τ local solvers (∃∀SOLVEASYNC), each executing in parallel on its own thread, and coordinates their search activities by sharing constraints.

```
1: function ∃∀SOLVEASYNC(id, ∃η. ∀x. ψ(η, x))
2:     G_S ← new IncrementalSMTSolver()
3:     y ← VALUESFOR(x)                                  ▷ Arbitrary initial binding for x
4:     Z ← {y}                                           ▷ CEGIS counterexamples
5:     Ψ ← {ψ}                                           ▷ All constraints
6:     ASSERT(G_S, ψ(η, x := y))                         ▷ Assert that ψ holds for y
7:     while true do
8:         block ← True
9:         result, h ← SOLVE(G_S)                        ▷ Solve for η
10:        if result = SAT then                          ▷ h is a candidate model
11:            result, z ← VERIFY(Ψ, η := h, x)
12:            if result = SAT then                      ▷ Candidate is incorrect
13:                ASSERT(G_S, ⋀_{φ∈Ψ} φ(η, x := z))
14:                Z ← Z ∪ {z}                           ▷ z is a counterexample
15:                block ← False                         ▷ Do not wait for more constraints
16:            else                                      ▷ Candidate is valid; send to global search
17:                SENDRESULT(id, SAT, h)
18:        else                                          ▷ Ψ is not valid
19:            SENDRESULT(id, UNSAT, ⊥)
20:            return
21:        if block ∨ a constraint has been received then
22:            φ ← RECEIVECONSTRAINT()
23:            ASSERT(G_S, ⋀_{z∈Z} φ(η, x := z))
24:            Ψ ← Ψ ∪ {φ}

25: function VERIFY(Ψ, η := h, x)
26:     G_V ← new SMTSolver()
27:     ASSERT(G_V, ⋁_{φ∈Ψ} ¬φ(η := h, x))
28:     return SOLVE(G_V)                                 ▷ Solve for x
```

Figure 5. The local synthesis algorithm ∃∀SOLVEASYNC takes as input a constraint ψ over a list of existentially quantified variables η and universally quantified variables x. The algorithm is an incremental form of counterexample-guided inductive synthesis (CEGIS) that accepts new constraints within the CEGIS loop. These new constraints are conjoined to ψ in the order in which they are received. ∃∀SOLVEASYNC emits models for η that make the resulting conjunctions valid. The search terminates if the current conjunction becomes unsatisfiable.

to send to individual solvers. The total order $\preceq$ on $\mathcal{S}$ defines the order in which to search sketches; the search order can significantly change the performance of the synthesis procedure, as Section 4.4 discusses. The gradient function g filters $\mathcal{S}$ once a satisfying solution is found to only search sketches with potentially cheaper solutions. Second, the global search receives candidate solutions from the individual solvers as they execute. The global search broadcasts information about these candidates to all local solvers, focusing their search efforts on cheaper solutions.

Figure 4 shows the global search algorithm SYNTHESIZE. The global search runs τ local solvers in parallel, each executing ∃∀SOLVEASYNC on a logical encoding of the synthesis problem for a particular sketch S from $\mathcal{S}$. Our algorithm assumes the existence of a procedure (as provided by, for example, Sketch [35] or ROSETTE [40]) that can encode the application of an arbitrary program from a sketch as a term in the theory $\mathcal{T}$. A local solver that completes a search with the sketch S returns a tuple $\langle S, result, h \rangle$ of results to the global search on line 12. The variable *result* is SAT if there is a completion of the sketch S that satisfies ϕ, or UNSAT otherwise. If *result* is SAT, the program $S[H := h]$ is a completion of S that satisfies ϕ; if *result* is UNSAT, h is $\bot$.

If a local solver produces a new solution that is the best seen so far (line 15), the global search filters $\mathcal{S}$ with the gradient function g (line 17). The gradient function restricts $\mathcal{S}$ to include only sketches that may contain programs cheaper than the new best cost c^*. The global search then announces c^* to all currently running solvers as a new constraint in theory $\mathcal{T}$. Consequently, the application of κ to an arbitrary program from S needs to be reducible to a term in $\mathcal{T}$ (as mentioned in Section 2). The local solvers use this constraint to prune their candidate space to include only programs cheaper than c^*. As an optimization, the global search can also kill local solvers that are no longer in the set $\mathcal{S}$ after applying the gradient function. This is sound because if a gradient function $g(c^*)$ filters out a sketch S, then by Equation (1), the sketch S has no solutions with cost less than c^*. Therefore, when the local search for S receives the new constraint that its cost be less than c^*, it will return UNSAT.

If a local solver produces no solution (line 25), it is marked as completed and new solvers are launched on new sketches from $\mathcal{S}$. In addition to the specification ϕ, these new solvers take an additional constraint that requires their solutions to be cheaper than the best known solution so far.

4.2 Local Searches

The global search invokes a local search procedure ∃∀SOLVEASYNC (Figure 5) on individual sketches S from the space $\mathcal{S}$ of the metasketch. The local search implements an incremental version of the CEGIS [35] algorithm for solving $\exists\eta.\forall x.\psi(\eta, x)$ synthesis queries—that is, for finding a binding h for η that makes the formula $\forall x.\psi(\eta := h, x)$ valid. The classic CEGIS algorithm uses one (incremental) solver instance, called the synthesizer, to search for an h that is correct for a set Z of values for x, by checking the

satisfiability of the formula $\exists\eta.\bigwedge_{z\in Z}\psi(\eta, x := z)$. If the synthesis formula is satisfiable, another solver instance, called the verifier, checks the satisfiability of the formula $\exists x.\neg\psi(\eta := h, x)$, looking for a value of x, called a counterexample, that invalidates the candidate solution h. If such a value exists, it is added to Z. This loop repeats until either the verifier returns UNSAT, indicating that h is valid, or the synthesizer returns UNSAT, indicating that there are no candidates left.

The ∃∀SOLVEASYNC algorithm extends CEGIS to accept additional constraints inside of the CEGIS loop. After each iteration of the CEGIS loop, our incremental algorithm can accept a new constraint $\psi'(\eta, x)$ (line 22) to obtain a new synthesis problem $\exists\eta.\forall x.\psi(\eta, x) \wedge \psi'(\eta, x)$. This constraint is added to the set of constraints seen so far (line 24), and asserted to the synthesizer for each counterexample collected so far (line 23). The synthesizer then searchers for a model of the new problem (line 9), and if the result is a valid solution (line 11), ∃∀SOLVEASYNC emits that solution to its output channel (line 17). As in classic CEGIS, an invalid solution leads to a counterexample, which is added to Z (lines 12–14). If there is no model for the problem, the search terminates (lines 19–20). When the algorithm satisfies all the constraints it has received so far, it blocks until it receives new constraints.

4.3 Characterization

We now show that the global search SYNTHESIZE is sound (it returns only correct programs), complete (it returns a correct program if one exists), and optimal (it returns the cheapest correct program). To start, we prove soundness and completeness of ∃∀SOLVEASYNC. Next, we use the soundness of ∃∀SOLVEASYNC to establish the soundness of SYNTHESIZE. We then observe that SYNTHESIZE is not guaranteed to terminate on an arbitrary metasketch with an infinite search space. To prove completeness and optimality, we therefore introduce *compact metasketches* (Def. 3), which place a simple compactness requirement on the gradient function g. This requirement is true of all metasketches with finite search spaces, and, in practice, is easy to satisfy for infinite search spaces as well. But SYNTHESIZE is still useful for non-compact metasketches: because it will always *discover* a solution if one exists (a property we call *online completeness*), an implementation that emits intermediate results (on line 16 of Figure 4) can be used to find the best solution within a given time budget.

Local Search. The global search invokes ∃∀SOLVEASYNC on the ∃∀ synthesis query for a given sketch S with respect to the correctness specification ϕ. To prove that the global search is sound, we need to show simply that no solution produced by ∃∀SOLVEASYNC violates ϕ (Lemma 1). Completeness of ∃∀SOLVEASYNC is more subtle, however. Unlike classic CEGIS, the incremental CEGIS explicitly filters out some solutions satisfying ϕ by receiving additional constraints. We prove completeness of ∃∀SOLVEASYNC in the case where it has received a set of constraints Ψ, but then receives no further constraints until it sends a result (Lemma 3). This completeness result is sufficient for proving completeness of the global search on compact metasketches.

Lemma 1 (Soundness of ∃∀SOLVEASYNC)**.** *Let $\exists\eta.\forall x.\psi(\eta, x)$ be the problem with which ∃∀SOLVEASYNC is initialized. If the algorithm emits a result of the form $\langle \text{id}, \mathit{SAT}, h\rangle$, then $\forall x.\psi(\eta := h, x)$ is valid modulo $\mathcal{T}$.*

Proof. If ∃∀SOLVEASYNC sends a result $\langle id, \text{SAT}, h\rangle$ from line 17 in Figure 5, then the verification on line 11 must have returned UNSAT. By the definition of VERIFY, this means that $\nexists z.\bigvee_{\varphi\in\Psi}\neg\varphi(\eta := h, z)$, and therefore that $\forall z.\bigwedge_{\varphi\in\Psi}\varphi(\eta := h, z)$. Since $\psi \in \Psi$ and additional constraints in Ψ can only rule out solutions, we have that $\forall x.\psi(\eta := h, x)$ is valid modulo $\mathcal{T}$. □

Lemma 2 (∃∀SOLVEASYNC loop invariant)**.** *At line 8 of Figure 5, the state of the incremental solver G_S is the assertion $\bigwedge_{\varphi\in\Psi}\bigwedge_{z\in Z}\varphi(\eta, x := z)$.*

Proof. By induction on loop iterations. On loop entry, $\Psi = \{\psi\}$ and $Z = \{y\}$, and the only assertion is $\psi(\eta, x := y)$. Now suppose the state is $\bigwedge_{\varphi\in\Psi}\bigwedge_{z\in Z}\varphi(\eta, x := z)$ at the start of the current iteration. The iteration can add only a single new counterexample z'; if it does, it will assert $\bigwedge_{\varphi\in\Psi}\varphi(\eta, x := z')$ (line 13), and set $Z' = Z \cup \{z'\}$; if not, it will set $Z' = Z$. (line 14) Then the iteration can add a single new constraint φ'; if it does, it will assert $\bigwedge_{z\in Z'}\varphi'(\eta, x := z)$ (line 23) and set $\Psi' = \Psi \cup \{\varphi'\}$ (line 24); if not, it will set $\Psi' = \Psi$. Therefore, at the start of the next iteration, the assertion store contains $\bigwedge_{\varphi\in\Psi'}\bigwedge_{z\in Z'}\varphi(\eta, x := z)$. □

Lemma 3 (Completeness of ∃∀SOLVEASYNC)**.** *Let $\exists\eta.\forall x.\psi(\eta, x)$ be the problem with which ∃∀SOLVEASYNC is initialized. Suppose that ∃∀SOLVEASYNC receives constraints $\varphi_1, \ldots, \varphi_k$, such that $\Psi = \{\psi, \varphi_1, \ldots, \varphi_k\}$. If no more constraints are received, and there exists some assignment h such that $\forall x.\bigwedge_{\varphi\in\Psi}\varphi(\eta := h, x)$ is valid modulo $\mathcal{T}$, then ∃∀SOLVEASYNC will eventually send a SAT result.*

Proof. Suppose we are at line 8 of Figure 5, when the previous iteration of the loop received the last message φ_k. Let Z' be the set Z of counterexamples at this point. We now have a set of specifications Ψ that will not change again. By Lemma 2, the accumulated state of G_S is the assertion $\bigwedge_{\varphi\in\Psi}\bigwedge_{z\in Z'}\varphi(\eta, x := z)$. From this point, the algorithm reduces to classic CEGIS, which is sound and complete on bounded input domains. □

Global Search. The correctness of the global search depends on the soundness and completeness of the local search. We first show that SYNTHESIZE is sound for the classic synthesis problem.

Theorem 1 (Soundness of SYNTHESIZE)**.** *Let $m = \langle\mathcal{S}, \kappa, g\rangle$ be a metasketch and ϕ a specification. If* SYNTHESIZE(ϕ, m) *returns a program P, then P is a solution to the classic synthesis problem; that is, $\forall x.\phi(x, [\![P]\!](x))$ is valid modulo $\mathcal{T}$.*

Proof. Follows immediately from Lemma 1. □

As stated, the SYNTHESIZE procedure is not complete: it is not guaranteed to return a solution if one exists, because it is not guaranteed to terminate. The issue is that both the space $\mathcal{S}$ and the sets returned by the gradient function g may be countably infinite. However, we can show that SYNTHESIZE will always *discover* a solution if one exists; that is, SYNTHESIZE is a semi-decision procedure. We call this property *online completeness*.

Theorem 2 (Online completeness of SYNTHESIZE)**.** *Let $m = \langle\mathcal{S}, \kappa, g\rangle$ be a metasketch and ϕ a specification. Suppose that there exists a program $P \in \bigcup_{S\in\mathcal{S}}\overline{S}$ in the search space defined by the metasketch such that $\forall x.\phi(x, [\![P]\!](x))$ is valid modulo $\mathcal{T}$. Then at some point during execution, the call to* WAITFORRESULT *on line 12 returns a tuple with result = SAT.*

Proof. Because $P \in \bigcup_{S\in\mathcal{S}}\overline{S}$, there exists a sketch $S \in \mathcal{S}$ such that $P \in \overline{S}$. Then there are three possibilities for how SYNTHESIZE treats the sketch S, all of which guarantee that the global search receives a SAT message:

- S is launched by a call to ∃∀SOLVEASYNC which then receives no constraint messages from the global search. Then by Lemma 3, because S is satisfiable, the global search eventually receives a SAT message with the sketch S.

- S is launched by a call to $\exists\forall$SOLVEASYNC and receives at least one constraint message from the global search. But constraint messages are only sent from line 21 of Figure 4, which is only reachable when WAITFORRESULT receives a SAT message.
- If S is never launched, it must have been removed from $\mathcal{S}$. This removal can only happen on line 23 of Figure 4, which is only reachable when WAITFORRESULT receives a SAT message.

□

Online completeness is useful because an implementation of SYNTHESIZE could emit intermediate results while continuing its search. As results in Section 5.4 show, SYNTHESIZE spends most of its execution time proving optimality of a candidate program, and so emitting intermediate results can make synthesis much faster at the expense of a weaker optimality guarantee. Online completeness ensures that SYNTHESIZE will always emit an intermediate solution if any solutions exist.

Compact Metasketches The global search is not guaranteed to terminate on an arbitrary metasketch, as explained above. To guarantee termination, we introduce an additional *compactness* constraint on the gradient function of a metasketch. This constraint is sufficient to prove that SYNTHESIZE is complete and optimal.

Definition 3 (Compact Metasketch)**.** *A* compact metasketch *is a metasketch $m = \langle \mathcal{S}, \kappa, g\rangle$ satisfying Definition 2 with the additional property that for all $c \in \mathbb{R}$, $g(c)$ is finite.*

Theorem 3 (Completeness of SYNTHESIZE)**.** *Let $m = \langle \mathcal{S}, \kappa, g\rangle$ be a compact metasketch and ϕ a specification. Suppose that there exists a program $P' \in \bigcup_{S\in\mathcal{S}} \overline{S}$ in the search space defined by the metasketch such that $\forall x.\, \phi(x, [\![P']\!](x))$ is valid modulo $\mathcal{T}$. Then there exists a program P such that* SYNTHESIZE(ϕ, m) *returns P.*

Proof. By Theorem 2, there is at least one sketch S and program P such that WAITFORRESULT will return the message $\langle \text{SAT}, S, P\rangle$. Let this be the first such SAT message. Then $c^* = \infty$, and so line 17 will set $\mathcal{S}' = g(\kappa(P))$. Since m is a compact metasketch, $\mathcal{S}'$ is finite, and since line 17 is only called when a new cost is smaller than c^*, no new sketches can be added to $\mathcal{S}'$. Therefore there are only finitely many sketches remaining to explore. Each sketch has only finitely many solutions and, whenever a sketch returns a SAT message, it either receives a new constraint ruling that solution out (if the solution it returned has cost $\kappa(P) < c^*$), or a constraint ruling that solution out is already waiting on its queue (if $\kappa(P) \geq c^*$). Therefore, local solvers can only return finitely many more solutions, after which they will return UNSAT and be added to $\mathcal{V}$. Eventually, the set $\mathcal{S} \setminus (\mathcal{R} \cup \mathcal{V})$ of unexplored sketches will be empty, the running sketches will return UNSAT, and SYNTHESIZE will return a program. □

Theorem 4 (Optimality of SYNTHESIZE)**.** *Let $m = \langle \mathcal{S}, \kappa, g\rangle$ be a compact metasketch and ϕ a specification. Suppose that* SYNTHESIZE(ϕ, m) *returns a program P with cost c. Then P is an optimal program: there is no program $P' \in \bigcup_{S\in\mathcal{S}} \overline{S}$ such that $\forall x.\, \phi(x, [\![P']\!](x))$ is valid modulo $\mathcal{T}$, and $\kappa(P') < c$.*

Proof. We proceed by contradiction. Suppose SYNTHESIZE returns P with $\kappa(P) = c$, but there is a sketch $S' \in \mathcal{S}$ that contains a correct program P' with $\kappa(P') = c' < c$. Since the assignment to c^* is guarded by line 15, c^* can only decrease, and by assumption, will never be smaller than c. By the definition of the gradient function, the sketch S' is never filtered out by line 17, since $c' < c \leq c^*$. Hence, when SYNTHESIZE receives a SAT message with a sketch S and cost c, either a local search for S' is still running, or it has not started. In both cases, the local search for S' will receive the constraint TOSMT$(\kappa(S[H := e])) < c$ (at line 21 or at line 34), and it will receive no further constraints (since we assumed that SYNTHESIZE returns a program with cost c). By Lemma 3, the local search will run to completion and will be satisfiable, returning a correct solution P' with cost $\kappa(P') < c$. This solution will be received by SYNTHESIZE on line 12, contradicting the assumption that SYNTHESIZE returns P. □

4.4 Implementation

We implemented our optimal synthesis approach in a new tool we call SYNAPSE, built on top of the ROSETTE language [39, 40], which extends Racket with features for program synthesis and verification using an underlying SAT or SMT solver [8, 41]. Here we briefly highlight some implementation details.

Sharing Counterexamples. The incremental CEGIS algorithm in Figure 5 can receive new constraints after each iteration. But the algorithm can be extended to also receive other messages. SYNAPSE exchanges CEGIS counterexamples between different local solvers in an effort to speed up each search. When a local solver sends a SAT or UNSAT message, it also includes the set Z of counterexamples it used to generate that result. The global search broadcasts the new counterexamples it receives to all running solvers, and maintains a set of all counterexamples that it provides to new local solvers. This optimization is sound because it does not affect the VERIFY check in $\exists\forall$SOLVEASYNC. SYNAPSE uses the shared counterexamples only to accelerate the VERIFY check in $\exists\forall$SOLVEASYNC, by first checking that the assertion on line 27 is not trivially invalidated by any of the existing counterexamples. This optimizations allows solvers to reduce the number of solver queries. We measure the effect of this optimization in Section 5.5.

Timeouts. While individual sketches are finite and therefore local searches will terminate, the queries made by local searches can take too long to be practical. We control this effect by adding a timeout parameter to $\exists\forall$SOLVEASYNC. Once the timeout expires, the local solver sends a timeout message to the global search. The global search treats a timed-out search in the same way as an unsatisfiable one: it kills the local solver and launches the next sketch.

Timeouts weaken the optimality guarantee that SYNAPSE provides. A solution output by SYNAPSE is only guaranteed to be optimal among those sketches that did not time out. In practice, the metasketches we designed were unlikely to contain cheaper solutions in sketches that timed out, and extending the time out by an order of magnitude did not change our results.

Search Order. The completeness of SYNAPSE does not depend on the order $\preceq$ of the set of sketches $\mathcal{S}$ in a metasketch. The only requirement is that the order is total (as Definition 2 states), so that for every sketch $S \in \mathcal{S}$, SYNTHESIZE eventually either tries to solve that sketch, or prunes it by finding a cheaper solution.

However, the search order can have a significant effect on performance. In the example metasketch designs in Section 3.3, we were careful to select a search order $\preceq$ that preferred simpler sketches to more complex ones. This order avoids wasted work on complex sketches that are likely to time out. It also best exploits the counterexample sharing optimization described above, as smaller sketches quickly generate a set of counterexamples that later local searches can use. We found the Cantor and Szudzik orders [19, 38] to be particularly effective.

5. Evaluation

To demonstrate that SYNAPSE effectively solves optimal synthesis problems expressed as metasketches, we evaluated it on four sets of benchmarks drawn from existing work. We sought to answer the following questions:

1. Is SYNAPSE a practical approach to solving different kinds of synthesis problems? In particular, can it solve optimal synthesis problems? Do metasketches also enable more effective classic synthesis compared to existing syntax-guided synthesizers?
2. Does the fragmentation of the search space by a metasketch translate into parallel speedup?
3. Is *online completeness* empirically useful? What proportion of SYNAPSE's run time is spent finding an optimal solution versus proving its optimality?
4. How beneficial are our optimizations at the level of metasketches (realized through structure constraints) and within the implementation (realized through counterexample sharing)?
5. Can SYNAPSE reason about dynamic cost functions; that is, cost functions that execute the synthesized program?

This section presents our benchmarks, experiments, and results. The results provide affirmative answers to all five questions. SYNAPSE, our benchmarks, and our experimental data are available online[3] and have been artifact evaluated.

5.1 Benchmarks

Table 1 shows the benchmarks used in our evaluation. The benchmark problems come from two sources: the 2014 and 2015 syntax-guided synthesis (SyGuS) competitions [3], and common approximate computing benchmarks [9]. We selected 67 problems from four categories of the SyGuS competition, ranging in difficulty from easy (i.e., solvable by most solvers) to hard (i.e., unsolvable by most solvers). The approximate computing benchmarks consist of 7 programs that cannot be approximated with existing software-based techniques. We developed metasketches for each set of problems. Section 3.3 described some of these metasketches, and we describe the rest below.

Hacker's Delight. The first category contains 20 bit-manipulating problems, used as superoptimization benchmarks in previous synthesis work [12, 31], appearing in two different difficulties, d0 and d5. The metasketch for a d0 problem includes only the bitvector operators that appear in the reference solution for that problem. The metasketch for d5 problems includes all bitvector operators.

Array Search. The second category contains 14 array search problems from the SyGuS competition [3]. The problem arraysearch-n is to synthesize a program that returns the index of a search key in a sorted array of size n, or zero if the key is not present in the array. The most efficient solution to these problems implements binary search. The metasketch for array search problems is an infinite set of sketches generated by two mutually recursive functions that encode SyGuS grammars for integer and boolean expressions. Each sketch in this metasketch is parameterized by the depth of the deepest integer and boolean expressions, respectively.

Conditional Integer Arithmetic (CIA). The third category contains 13 conditional integer arithmetic problems[4] new to the 2015 syntax-guided synthesis competition [4]. Each problem involves synthesizing a program from a grammar that includes the program inputs; constants 0, 1, and 3; integer addition and subtraction; and the `qm` operation

```
(define (qm a b)
  (if (< a 0) b a))
```

The metasketch for CIA problems is an infinite set of sketches generated by this grammar, with one sketch per depth of production from the grammar. Several of the CIA benchmarks were unsolved by any solver in the SyGuS competition; we present only those solved by at least one SyGuS solver or by SYNAPSE.

Parrot. The fourth category contains 7 problems drawn from the approximate computing literature [9]. The specification for these problems allows the synthesized program to differ from the reference program by a given application-specific quality bound. We use two metasketches for the Parrot benchmarks: piecewise polynomial approximation (for benchmarks that use transcendental functions) and adaptive superoptimization (for all other benchmarks).

Methodology. We performed all experiments on an 18-core Intel Xeon E5-2666 CPU at 2.9 GHz, with 60 GB of RAM. For SyGuS benchmarks, we timed out each metasketch after one hour, for consistency with the SyGuS competition setup [3]. For Parrot benchmarks, we did not use a timeout for any metasketch. In both cases, individual sketches within a metasketch were timed out after 15 minutes. Section 4.4 describes the effect of timeouts on SYNAPSE's optimality guarantee; we found that extending the individual sketch timeout by an order of magnitude did not discover cheaper solutions for any problem. All timing results are wall-clock times for the entire SYNAPSE execution. Where speedups are presented, they are aggregated over all benchmarks in a category before being normalized to the relevant baseline [33].

5.2 Is SYNAPSE a practical approach to solving different kinds of synthesis problems?

To evaluate the effectiveness of SYNAPSE as a generic synthesis engine, we applied it to all of our benchmarks in sequential mode—that is, running only a single local search at a time. This gives a baseline for comparison against existing syntax-guided synthesis solvers, which are single-threaded. Figure 6 shows the sequential solving performance of SYNAPSE on our benchmarks.

For the Hacker's Delight benchmarks at difficulty d0, SYNAPSE solves all 20 problems. The performance is competitive with results from the syntax-guided synthesis (SyGuS) competition [3], showing that metasketches do not introduce additional overhead for easy problems. However, SYNAPSE also solves problem 20, which none of the SyGuS solvers could solve in either 2014 or 2015.

For Hacker's Delight benchmarks at difficulty 5, SYNAPSE is able to solve 18 of the 20 problems within a one hour timeout. This result is better than other SMT-based SyGuS solvers: the symbolic solver in 2014 [2, 12] times out on all 20 problems, the Sketch-based [35] solver in 2014 solves only problems 1–8, and the CVC4-based solver that won the 2015 competition [30] cannot solve problems 14 or 15. The winner of the SyGuS competition in 2014 used an enumerative brute force strategy, and solved the same 18 problems that SYNAPSE solves in comparable time (same order of magnitude).

SYNAPSE solves all Array Search problems. In comparison, the best SyGuS solver on these problems in 2014 was the Sketch-based [35] solver, which could solve these problems only up to length 7. The enumerative solver, which won the syntax-guided synthesis competition, could only solve lengths 2 and 3. In 2015, the CVC4-based solver [30] also solved all Array Search problems. However, its solutions were highly non-optimal: for arraysearch-15, SYNAPSE produces the expected binary search solution with AST depth 5 and size 349 bytes, while CVC4 produces a solution with AST depth 45 and size 7.1 MB.

SYNAPSE is also able to solve all seven Parrot problems. We attempted to solve the Parrot benchmarks using SyGuS solvers, Sketch [35], and the Stoke stochastic superoptimizer [31] without success. We encoded the adaptive superoptimization Parrot problems in the SyGuS benchmark format and in Sketch. Only the CVC4-based SyGuS solver [30] and Sketch produced solutions for these

[3] `http://synapse.uwplse.org`

[4] The conditional integer arithmetic benchmarks are labeled `qm` in the SyGuS competition dataset.

Benchmark Suite	Problems	Source	Problem Description	Metasketch	Cost Function
Array Search	14	SyGuS'14 [3]	Search a sorted array of size n for a given element and return its index	Array programs	Expression depth
Conditional Integer Arithmetic (CIA)	13	SyGuS'15 [4]	Integer programs that use complex branching structure	Integer programs	Expression depth
Hacker's Delight d0	20	SyGuS'14 [3]	Bit-manipulating programs, with sketches in the metasketch containing only the minimal set of bitvector operators necessary to implement the reference program.	Superoptimization	Program length
Hacker's Delight d5	20	SyGuS'14 [3]	Bit-manipulating programs, with sketches in the metasketch containing all operators from the theory of bitvectors.	Superoptimization	Program length
Parrot	7	Parrot [9]	Approximate computing kernels: kmeans and sobel ($\times 2$ convolution matrices)	Adaptive superoptimization	Static cost model
			Approximate computing kernels: fft ($\times 2$ outputs) and inversek2j ($\times 2$ outputs)	Piecewise polynomial approximation	Pieces + Degree

Table 1. The benchmarks used in our evaluation. For each benchmark suite, we wrote a metasketch whose set of sketches together form the relevant search space, as described in Section 3.3.

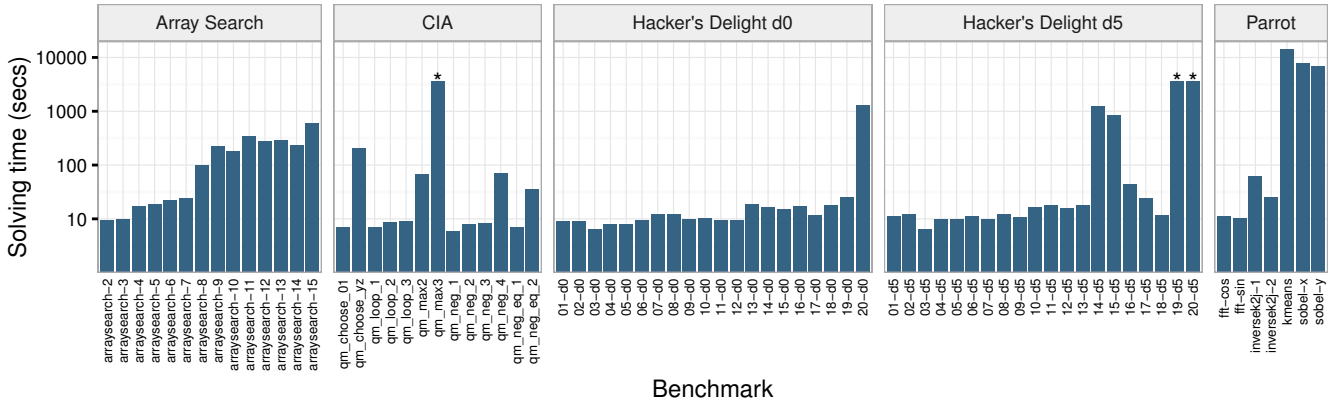

Figure 6. Sequential solving performance for all benchmarks. Asterisks indicate benchmarks that timed out after one hour.

problems, but the solutions failed to meet the specification. The other publicly-available SyGuS solvers did not return solutions in 4 hours, which is the maximum time taken by SYNAPSE to solve any Parrot benchmark. The piecewise polynomial Parrot problems are not expressible in the SyGuS format, because they require synthesizing (arbitrary numeric) constants. Sketch supports synthesis of constants, but it was unable to solve any of these problems in 4 hours. Implementing the benchmarks in C and passing them to Stoke also resulted in no solutions.

5.3 Does the fragmentation of the search space by a metasketch translate into parallel speedup?

To evaluate the benefits of coarse-grained parallelism exposed by metasketches, we applied SYNAPSE to our benchmarks using 2, 4, and 8 threads. Figure 7 shows the resulting parallel solving performance. We omit Hacker's Delight at difficulty d0 because these small benchmarks do not benefit from parallelization. Results are speedups in total execution time aggregated over all benchmarks in a category, excluding benchmarks that timed out at any number of threads.

SYNAPSE realizes substantial parallel speedups for the Parrot problems, which are the hardest synthesis problems in our benchmark suite. These speedups are similar to or better than recent work

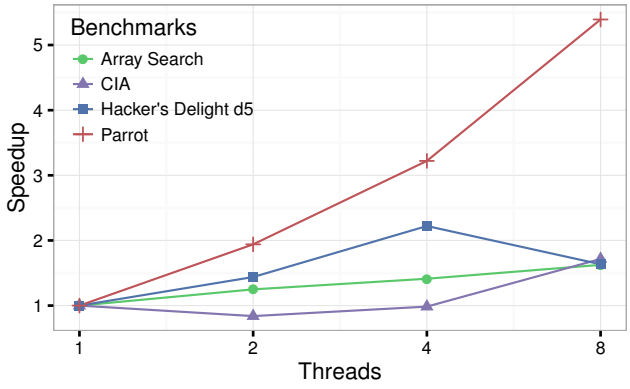

Figure 7. Parallel solving performance for all benchmarks except Hacker's Delight d0 (which are too small to benefit). Parrot sees substantial parallel speedup. Hacker's Delight d5 sees speedup up to four threads, but then a single local search dominates solving time. Array Search and Conditional Integer Arithmetic benchmarks see minimal speedup because most solving time is spent on a single local search.

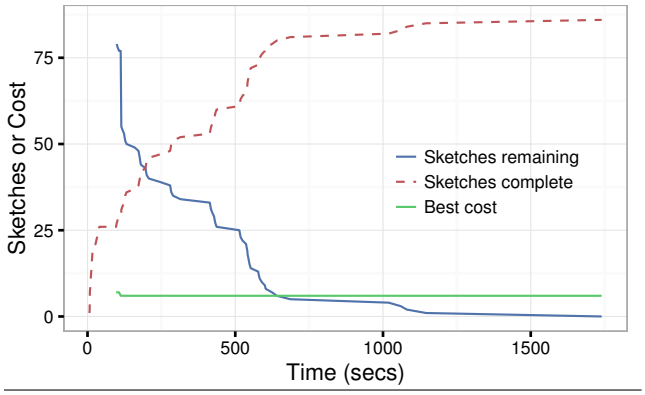

Figure 8. Search progress for the sobel-y benchmark on 4 threads. When solutions are discovered, the size of the remaining search space to explore drops significantly. SYNAPSE finds the optimal solution after 112 s, but must spend an additional 1,628 s proving it optimal.

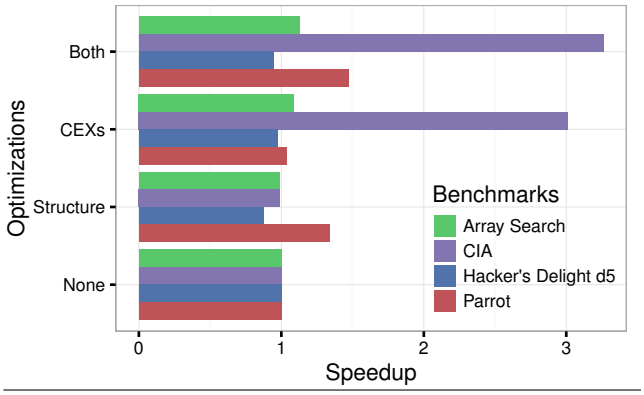

Figure 9. Effect of optimizations on SYNAPSE's performance for a single-threaded search. SYNAPSE can exchange counterexamples (CEXs) between local searches, and can impose structure constraints that prevent different local searches traversing some semantically-equivalent programs.

in parallel program synthesis [14]. For Hacker's Delight problems, the parallel speedups are significant up to four threads, but eventually a single local search (which executes sequentially) becomes the bottleneck. For Array Search and Conditional Integer Arithmetic problems, parallel speedups are minimal, because most sketches early in the search order are quickly found to be unsatisfiable, so solving time is again dominated by a single sequential local search.

5.4 Is *online completeness* empirically useful?

Unlike a classic program synthesizer, which can terminate as soon as it discovers a solution, an optimal program synthesizer such as SYNAPSE must also prove the optimality of a candidate solution. Metasketches provide an abstraction that allows this search to terminate despite exploring an infinite space of candidate programs. But the proof of optimality can still consume a significant portion of the search time: the gradient function of a metasketch returns sketches that *may* contain cheaper candidate programs, and so the search can spend significant amounts of time searching sketches that do not contain cheaper solutions.

Figure 8 shows the progress over time of a search for the sobel-y Parrot benchmark. The x-axis is the time since starting the search, and the y-axis plots both the number of sketches completed and remaining in the search, and the cost of the best solution so far. Note that the number of sketches remaining is infinite before a first solution is found. The search discovers the optimal solution after 112 s with cost 6. However, the gradient function returns 56 sketches that may contain solutions of lower cost. The search spends another 1,628 s exploring each of these sketches to prove they do not contain such solutions. The slope of the sketches-remaining line in Figure 8 shows that many of these sketches can be quickly pruned, due to the added constraint they receive from the global search that their solutions must be cheaper than 6. For some sketches, however, the local search is unable to quickly deduce unsatisfiability despite this added constraint. These sketches dominate the search time.

5.5 How beneficial are our metasketch and implementation optimizations?

SYNAPSE admits two optimizations beyond existing CEGIS-based solvers, as Section 4.4 describes. First, the global search can exchange counterexamples between local searches, which can improve their performance by reducing the number of calls to the verifier. Second, a metasketch can impose structure constraints on the individual local searches, which can rule out some semantically-equivalent programs from being considered by multiple searches.

Figure 9 shows the effect of these optimizations for a single-threaded search. Results are speedups in total execution time aggregated over all benchmarks in a category, excluding benchmarks that timed out in any configuration. Both Hacker's Delight and Array Search benchmarks see minimal benefits from the optimizations, because they consist mainly of sketches that are easily proven unsatisfiable. The Parrot problems benefit significantly (50%) from structure constraints, because the sketches in the Parrot metasketches contain significant semantic overlap. The Conditional Integer Arithmetic benchmarks, on the other hand, benefit significantly (3×) from counterexample exchange.

5.6 Can SYNAPSE reason about dynamic cost functions?

Metasketches place only very general restrictions on cost functions: the application of the cost function to a program must reduce to a term in a decidable theory (as discussed in Section 2). This restriction allows for static cost functions, such as static instruction cost models, but also for dynamic cost functions that execute the synthesized program to establish its cost. We illustrate SYNAPSE's support for a variety of dynamic cost functions with three small examples.

Least-Squares Regression. Least squares regression fits a model function f to a data set $\{x_i, y_i\}$ by minimizing the objective function $\sum_{i=1}^{n}(y_i - f(x_i))^2$. We implemented a modified version of the piecewise polynomial metasketch presented in Section 3.3 to perform least-squares regression. Each sketch in this metasketch is a piecewise polynomial with a fixed number of pieces and fixed degree. We defined the cost function to be the least-squares objective function, which is dynamic because it requires evaluating the synthesized program f at each x_i in the data set. The metasketch uses the trivial gradient function $g(c) = \mathcal{S}$, and because this metasketch is not compact (Def. 3), we provided a finite set $\mathcal{S}$ of sketches. We used as a data set 30 samples of the polynomial

$$p(x) = x^3 - 8x^2 + x - 9$$

from the interval $x \in [-1, 10]$ with added Gaussian noise ($\sigma = 5$). SYNAPSE synthesized the polynomial

$$q(x) = x^3 - 8x^2 + x - 7$$

in 30 seconds as the optimal (integer) solution to this problem. SYNAPSE also explored the other sketches in the metasketch, which correspond to other possible models for the data (including linear, quadratic, and quartic functions), but correctly found the cubic function to be the best fit.

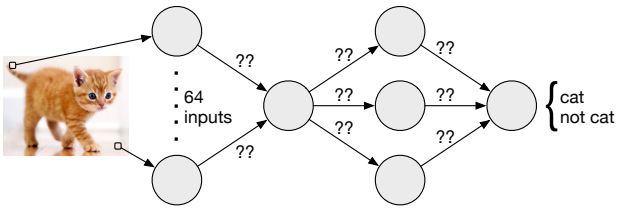

Figure 10. The sketch for a neural network is an SSA-form implementation of its evaluation function, with holes for each weight. In this example, the input nodes are the grayscale values of each pixel in the input image, and the output is a binary classification.

Worst-Case Execution Time. The superoptimization metasketches in Section 3.3 used static measures of program performance. The metasketch abstraction also supports dynamic measures, such as worst case execution time. To illustrate the effects of dynamic and static measures on the results of optimal synthesis, we selected problem 13 from the Hacker's Delight benchmark suite, which implements the sign function for 32-bit integers. We created two metasketches for this benchmark: $\langle \mathcal{S}, \kappa_s, g \rangle$ and $\langle \mathcal{S}, \kappa_d, g \rangle$, where $\mathcal{S}$ is a finite set of sketches and g is the trivial gradient $g(c) = \mathcal{S}$. Our sketches were drawn from a subset of the SYN grammar that includes conditional expressions and a minimal set of operators needed to implement benchmark 13. We defined κ_s to be an additive static cost function (as in Example 3 of Section 2) that assigns cost 2 to conditional expressions and cost 1 to all other expressions, including constants and variables. On the other hand, κ_d is a dynamic cost function that measures the worst-case execution time (over all feasible paths) using the same costs for SYN expressions as κ_s. We applied SYNAPSE to both metasketches and obtained two different optimal programs.

The static metasketch $\langle \mathcal{S}, \kappa_s, g \rangle$ produces the reference implementation for benchmark 13 as the optimal solution (with cost 8):

```
(define (sgn-s x)
  (| (>>> (- x) 31) (>> x 31)))
```

The dynamic metasketch $\langle \mathcal{S}, \kappa_d, g \rangle$, on the other hand, produces a different optimal solution (also with cost 8):

```
(define (sgn-d x)
  (if (< 0 x)
    (>>> -1 31) ; 1
    (>> x 31)))
```

The `sgn-d` function has cost 11 under the static cost function κ_s, and so is not an optimal solution to the static metasketch. SYNAPSE finds each solution in a few seconds.

Neural Networks. We developed a simple metasketch to train a feed-forward neural network classifier from a training set. The neural network metasketch $\langle \mathcal{S}, \kappa, g \rangle$ contains a sketch S_t in $\mathcal{S}$ for each of a (finite) set of neural network topologies t. A sketch S_t is an implementation of a feed-forward neural network in the SSA language of Figure 2 using fixed-point arithmetic and rectifier activation functions [26]. Each weight in the neural network is a hole in the sketch, as shown in Figure 10. The cost function κ simply counts the number of misclassified examples. Notably, we did not have to explicitly encode a training algorithm such as backpropagation.

We used this metasketch to train a simple image classifier to distinguish between cats and other images. We used 40 training examples (20 cats, 20 not-cats) from the CIFAR-10 dataset [18], resized to 8×8 pixels and converted to grayscale. SYNAPSE synthesized a neural network in 35 mins that achieved 95% recognition accuracy on the 40 examples. The synthesized network has 64 input nodes (one per pixel in the input image), a hidden layer of one node, a second hidden layer of three nodes, and a single output node. In training this network, SYNAPSE also explored the rest of the metasketch, consisting of all topologies with at most 2 layers and 4 nodes per layer (a total of 20 topologies).

Of course, this example is no advance in machine learning: we do not have anywhere near enough training examples, the recognition accuracy is measured on the training set, and the training time is many orders of magnitude slower than backpropagation. Rather, this example demonstrates that the underlying synthesizer can discover a training strategy given only an implementation of forward evaluation and the error function to minimize. It also demonstrates that SYNAPSE can handle large, under-constrained programs: the SSA-form program for the synthesized neural network contains 284 instructions, and some other topologies in the metasketch consist of over 1500 instructions.

6. Related Work

Program synthesis is well studied in the literature. Some program synthesizers, and some applications of synthesis, implicitly or explicitly optimize an objective function. This section reviews related work on program synthesis, domain-specific synthesizers, and on optimal or quantitative synthesis.

Program Synthesis. Synthesis techniques have been successfully applied to a wide variety of problems, including compilation for ultra low power spatial architectures [27]; generation of high-performance data-parallel code [36, 39]; generation of efficient web layout engines [24]; education [1]; and end-user programming [11].

Our work builds on recent advances in syntax-guided synthesis (e.g., [2, 12, 15, 31, 39, 42]), which are based on counterexample-guided search [35]. In addition to a correctness specification, a syntax-guided synthesizer takes as input a space of candidate programs, defined by a syntactic template, and searches it for a program (if any) that satisfies the specification. Existing approaches employ a variety of search procedures, including bottom-up enumeration [42], symbolic solving [12, 15, 35, 39], and stochastic search [31]. The recently developed syntax-guided synthesis (SyGuS) framework [2] unifies these approaches, providing a common language for expressing synthesis problems, a suite of standard benchmarks, and a set of search procedures for solving SyGuS problems. We drew several of our benchmarks from the SyGuS framework.

A number of SyGuS solvers implicitly minimize (or nearly minimize) a fixed objective function. A bottom-up enumerative solver [42] implicitly minimizes program length: shorter solutions will be discovered before longer ones. Some symbolic synthesis algorithms [12] use a fixed library of components, which is expanded only when the problem is unsatisfiable, implicitly directing the search toward simpler programs. In both cases, however, the optimization is implicit, and not easily extended to different cost functions. The winner of the 2015 SyGuS competition builds support for refutation-based synthesis into the CVC4 SMT solver [30]. While the refutation approach quickly produces correct solutions, those solutions are often extremely long. Extending the refutation approach with support for optimization would be non trivial.

Domain-Specific Synthesizers. Optimality is desirable in a number of synthesis applications, and so domain-specific synthesizers often implement an optimization strategy. Chlorophyll [27] is a synthesis-aided compiler for a low-power spatial architecture that performs modular superoptimization. The superoptimizer executes a binary search over programs given a cost model: it uses counterexample-guided inductive synthesis (CEGIS) to synthesize a program of cost k, and if one exists, to synthesize a program of cost $k/2$, and so on. To make the superoptimization scale to real-world programs, the process uses "sliding windows" to break a program into smaller pieces. Metasketches also give structure to the search,

but our synthesis approach can provide whole-program optimality guarantees that the sliding windows technique cannot, and can reason about more general cost functions.

Feser et al. [10] present a synthesis algorithm for producing data structure transformations from input-output examples. The algorithm guarantees optimality of the generated transformation with respect to an additive cost function over program syntax, which is defined similarly to the cost function in Example 3 of Section 2. In contrast, our approach is generic: it is applicable to a broad range of synthesis problems, and it can optimize a variety of cost functions, as long as their semantics is expressible in a decidable theory.

McSynth [37] is a synthesizer that generates machine code instructions from semantic specifications of their behavior. While McSynth alone does not consider optimality, the authors note that it could be extended to generate optimal solutions with a naive algorithm that generates *every* solution to the synthesis problem and returns the one among them with minimum cost. Metasketches provide considerably more structure to the optimal synthesis process, and can accommodate an unbounded space of sketches (and therefore solutions).

Optimal Synthesis. Recent work has considered optimality in synthesis, with various forms of ranking and weighting formulations [13, 22, 28], and in SMT problems in general. Chaudhuri et al. [6], for example, propose a smoothed proof search technique for synthesizing parameter holes in a program while optimizing a quantitative objective. Smoothed proof search reduces optimal synthesis to a sequence of optimization problems that can be solved numerically to satisfy the specification in the limit. The use of numerical optimization allows this technique to perform probabilistic reasoning, which SYNAPSE does not support. However, the technique's sketching language is less expressive than a metasketch, allowing only linear operations on holes. The optimality guarantee also holds only over a single monolithic sketch, which restricts synthesis to a finite set of candidate programs; in contrast, a metasketch can represent an unbounded set of candidate programs.

Symba [20] is an SMT-based optimization algorithm for objective functions in the theory of linear real arithmetic (LRA). Symba optimizes the objective function by maintaining an under-approximation of the maximal cost, and using the SMT solver to generate new models for the specification that violate that under-approximation (i.e., are more optimal). This approach builds on a line of work on optimization for SMT problems [7, 32]. Unlike Symba, our approach supports non-linear cost functions (in, for example, the theory of bitvectors) and can optimize over an unbounded space of candidate programs. However, Symba is able to detect when the cost function it is maximizing has no upper bound, whereas a (compact) metasketch must explicitly rule out this possibility. Integrating Symba with our approach is a promising direction for future work.

7. Conclusion

We presented metasketches, a general framework for specifying and solving optimal synthesis problems. A metasketch fragments the search space of a synthesis problem into an ordered set of sketches, provides a cost function to optimize, and specifies a gradient function to direct the search toward cheaper regions of the space. This three-part abstraction enables the programmer to succinctly express both the desired (optimal) synthesis problem and a high-level strategy for solving it. By making the search strategy programmable, metasketches enable rapid creation of competitive (optimal) synthesis tools, ranging from superoptimization to approximation of computational kernels. Moreover, metasketches bring new expressive power to syntax-guided synthesis, including sketching of unbounded search spaces and use of dynamic cost functions that reason about program semantics. Our synthesis approach, implemented in SYNAPSE, exploits the structure of metasketches to search for solutions in parallel, employing effective generic and problem-specific optimizations. Our results demonstrate that custom search strategies expressed via metasketches make it possible for SYNAPSE to solve hard synthesis problems, both optimal and classic, which cannot be solved with general state-of-the-art synthesis algorithms.

Acknowledgements

We thank Ras Bodik, Brandon Holt, Zach Tatlock, Xi Wang, and the anonymous reviewers for their feedback on earlier versions of this paper. This work was supported in part by NSF grants #1064497 and #1518703, by DARPA under contract #HR0011-15-1-0001, and by the David Notkin Endowed Graduate Fellowship.

References

[1] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of DFA constructions. In *IJCAI*, 2011.

[2] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, 2013.

[3] R. Alur, R. Bodik, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, volume 40. 2015.

[4] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama. The second competition on syntax-guided synthesis. In *SYNT*, 2015. URL `http://formal.epfl.ch/synt/2015/slides/fisman-etal.pdf`.

[5] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA*, 2013.

[6] S. Chaudhuri, M. Clochard, and A. Solar-Lezama. Bridging boolean and quantitative synthesis using smoothed proof search. In *POPL*, 2014.

[7] A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani, and C. Stenico. Satisfiability modulo the theory of costs: Foundations and applications. In *TACAS*, 2010.

[8] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[9] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012.

[10] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, 2015.

[11] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[12] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[13] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8), Aug. 2012.

[14] J. Jeon, X. Qiu, A. Solar-Lezama, and J. S. Foster. Adaptive concretization for parallel program synthesis. In *CAV*, 2015.

[15] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.

[16] R. Joshi, G. Nelson, and K. Randall. Denali: a goal-directed superoptimizer. In *PLDI*, 2002.

[17] A. S. Köksal, Y. Pu, S. Srivastava, R. Bodik, J. Fisher, and N. Piterman. Synthesis of biological models from mutation experiments. In *POPL*, 2013.

[18] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[19] I. Kuraj, V. Kuncak, and D. Jackson. Programming with enumerable sets of structures. In *OOPSLA*, 2015.

[20] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik. Symbolic optimization with SMT solvers. In *POPL*, 2014.

[21] A. Massalin. Superoptimizer: A look at the smallest program. In *ASPLOS*, 1987.

[22] A. K. Menon, O. Tamuz, S. Gulwani, and A. T. Kalai. A machine learning framework for programming by example. In *ICML*, 2013.

[23] L. A. Meyerovich. *Parallel Layout Engines: Synthesis and Optimization of Tree Traversals*. PhD thesis, University of California, Berkeley, 2013.

[24] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodik. Parallel schedule synthesis for attribute grammars. In *PPoPP*, 2013.

[25] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin. SNNAP: Approximate computing on programmable SoCs via neural acceleration. In *HPCA*, 2015.

[26] V. Nair and G. E. Hinton. Rectified linear units improve restricted Boltzmann machines. In *ICML*, 2010.

[27] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *PLDI*, 2014.

[28] O. Polozov and S. Gulwani. FlashMeta: A framework for inductive program synthesis. In *OOPSLA*, 2015.

[29] J. D. Ramsdell. An operational semantics for Scheme. *SIGPLAN Lisp Pointers*, V(2):6–10, 1992.

[30] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. Barrett. Counterexample-guided quantifier instantiation for synthesis in SMT. In *CAV*, 2015.

[31] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *ASPLOS*, 2013.

[32] R. Sebastiani and S. Tomasi. Optimization in SMT with $\mathcal{LA}(\mathbb{Q})$ cost functions. In *IJCAR*, 2012.

[33] J. E. Smith. Characterizing computer performance with a single number. *Commun. ACM*, 31(10), Oct. 1988.

[34] A. Solar-Lezama. *Program synthesis by sketching*. PhD thesis, University of California, Berkeley, 2008.

[35] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[36] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI*, 2007.

[37] V. Srinivasan and T. Reps. Synthesis of machine code from semantics. In *PLDI*, 2015.

[38] M. Szudzik. An elegant pairing function. In *NKS*, 2006.

[39] E. Torlak and R. Bodik. Growing solver-aided languages with Rosette. In *Onward!*, 2013.

[40] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, 2014.

[41] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *TACAS*, 2007.

[42] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur. Transit: Specifying protocols with concolic snippets. In *PLDI*, 2013.

[43] H. S. Warren, Jr. *Hacker's Delight*. Addison-Wesley, 2007.