Optimal Program Synthesis

Abstract

Program synthesis is the task of automatically producing a program that meets a specification. Many applications of program synthesis require not just any solution, but one that is *optimal* with respect to a desired cost function, such as program length. Solvers for optimal program synthesis exist, but are domain-specific and difficult to extend to new problems. We present a general framework for optimal synthesis, instantiated with a *metasketch*. Metasketches define fragmentations of the space of candidate programs and provide abstractions for effective optimal synthesis with two new, cooperating search algorithms. 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 a new incremental form of counterexample-guided inductive synthesis to incorporate information sent from the global search during the synthesis process. Together, these algorithms allow metasketches to provide not only a description of the space of candidate programs but also a high-level strategy for exploring that space. We present SYNAPSE, an implementation of our framework, and show that it effectively solves a wide range of optimal synthesis problems. The additional structure and strategy that a metasketch provides also improve classic (non-optimal) synthesis, and we show that SYNAPSE solves classic synthesis problems that state-of-the-art tools cannot.

1. Introduction

Program synthesis is the classic problem of automatically producing an implementation of a desired correctness specification. Recent research efforts have addressed this problem successfully for a variety of application domains, from browser layout [18] to executable biology [15]. But for many applications, such as synthesis-aided compilation [21, 23] or end-user programming [9, 10], 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—e.g., the number of instructions or the sum of instruction 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. Like program synthesis, the optimal synthesis problem is not new, and many dedicated solvers exist for specific instances of it. For example, a superoptimizer [14, 23] solves the problem of producing an instruction sequence that is equivalent to a given reference implementation, and that is the cheapest such sequence according to a cost model for an instruction set architecture. These dedicated solvers are highly effective at solving their target class of optimal synthesis problems. However, they are also difficult to design and implement, as well as to extend to different cost functions or generate different kinds of programs.

In this paper, we present a general framework for optimal synthesis that builds on syntax-guided synthesis [2, 26]. In syntax-guided synthesis, the synthesizer searches for a correct program in a space of candidate implementations defined by a syntactic template. The template can be defined by a context-free grammar [2] or by a *sketch* [26] (i.e., a partial implementation) of the desired program. Each approach has advantages: a grammar makes it possible to define an infinite space of candidate programs, while a sketch makes it possible to express constraints on program syntax that are not context-free. We unify the two approaches with the notion of a *metasketch*, which defines the space of candidate programs using an ordered, countable set of sketches. To support optimal synthesis, a metasketch also includes a *cost function* to minimize and a *gradient function* to direct the search. These three ingredients together express high-level search strategy, fragmenting the search space and allowing it to be explored in parallel, which facilitates both generic optimal synthesis and faster classic synthesis.

Our synthesis approach involves two cooperating algorithms: a global optimizing search over the entire search space and many parallel instances of a local combinatorial search over the individual sketches in the metasketch. The local search algorithm implements a new form of incremental 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 prune their own candidate spaces accordingly. The search process proceeds until an optimal solution is found or the global search space is exhausted. This search strategy is highly effective, easily solving both classic and optimal synthesis problems that cannot be solved by existing techniques.

The design of the metasketch abstraction is key to the effectiveness of our approach. 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). Crucially, such a space description is conducive to both parallel global search (because individual sketches can be solved independently) and efficient local search (because individual sketches can omit programs that will be included in other sketches). In essence, a

[Copyright notice will appear here once 'preprint' option is removed.]

metasketch enables the programmer to easily describe both a space of candidates and a *search strategy* over that space.

We have implemented our optimal synthesis approach in a tool called SYNAPSE, built on top of the ROSETTE language [29, 30]. We have used SYNAPSE to develop and solve metasketches for a variety of optimal synthesis problems, from superoptimization to fixed-point approximation of transcendental functions. Our experiments show that our search algorithm 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). Our experiments also show that 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.

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 based on a new 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 [8]. Our results show that SYNAPSE can solve both classic and optimal synthesis problems that cannot be solved with other techniques.

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 three examples of metasketches designed for superoptimization and approximate computing. We describe and evaluate our synthesis algorithm in Sections 4–5. The paper concludes with a discussion of related work (Section 6) and a brief summary of the contributions (Section 7).

2. Optimal Syntax-Guided Synthesis

This section briefly reviews syntax-guided synthesis [2, 26] 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 [26] or ROSETTE [30]).

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 $\mathcal{T}$.

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.

Programs and Specifications. For examples in this paper, we take the language $\mathcal{L}$ to be SYN, a subset of core Scheme [22] 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 referenced variable is not **false**. The semantics of SYN is standard [22], 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; for example, 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 any other 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 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* [26]. 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 only 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 enable only specification of finite candidate spaces $\mathcal{C}$. In return, however, they provide the programmer with more control over the shape of the search space, and the ability to express syntactic constraints that are not context-free.

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

expressions e ::= ... $\mid$ (**??** $e \ldots$)

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 between the provided expressions.

[1] In particular, SYN assertions can be reduced to formulas in the theory of bitvectors for all finite SYN programs, using existing methods [30].

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

We call a program in SYN a sketch if it contains holes. A sketch $S \in \mathcal{L}$ defines a set of candidate programs $\overline{S}$, the set of all possible programs produced by substituting the set of holes H in P with concrete values. 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 the values of the holes that satisfy 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 non-functional requirements, such as program size, execution time, memory or register usage, etc. 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. In particular, κ may take the form $\forall x.\mathit{cost}([\![P]\!](x))$, where $\mathit{cost}([\![P]\!](x))$ is a (quantifier-free) term in $\mathcal{T}$, enabling us to encode dynamic cost functions that execute P and minimize the result over all inputs.[3] In Section 5.6, we demonstrate the utility of cost functions that can reason about program semantics, by using a simplified worst case execution time cost function.

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 the cost of each operation:

[3] Our framework allows the universal quantification over x because our synthesizer works, conceptually, by solving a series of problems of the form $\exists P \in \mathcal{C}. \forall x.\, \phi(x, [\![P]\!](x)) \wedge \mathit{cost}([\![P]\!](x)) < c$ for a decreasing sequence of constant values c, until the problem becomes unsatisfiable.

programs	$\kappa(p)$	$= \sum_{f \in p} \kappa(f)$
definitions	κ ((**`define`** x e))	$= \kappa(e)$
expressions	κ ((e $e_i \ldots$))	$= \kappa(e) + \sum_i \kappa(e_i)$
	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 (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:

expressions	κ ((**`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 unsigned 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

We introduce the notion of a *metasketch* (Def. 2) for solving optimal program synthesis problems. Metasketches generalize sketches, enabling the programmer to describe an infinite space of candidate programs with a set of finite sketches. This representation permits fine-grained control over the shape of the search 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. In this section, we define metasketches, describe their properties, and illustrate their utility for capturing insights that enable efficient (optimal) synthesis. In Section 4, we present a novel synthesis algorithm that exploits the search strategy of a metasketch to efficiently solve optimal synthesis problems.

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 that maps programs to numeric cost values; and (3) a *gradient* function that maps each cost value to a set of sketches (i.e., a subspace) that may contain a program with a lower cost than that given. Intuitively, 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 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 $\bigcup_{S \in \mathcal{S}} \overline{S}$ of the search spaces of each sketch in the collection, and the cost function is given by κ.

3.2 Properties of Metasketches

A metasketch brings new expressive power to (optimal) syntax-guided synthesis in two ways. First, it 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, a metasketch also captures a *search strategy*.

In particular, by specifying the candidate space as a set of ordered sketches, the programmer suggests 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 $S \in \mathcal{S}$ is a concrete program with no holes, the metasketch is an implementation of brute force search. If those programs are ordered according to AST depth, for example, then it is an implementation of bottom-up brute force search. At the other extreme, if a metasketch is just a single finite sketch, the programmer is choosing to solve the problem monolithically with some predefined algorithm, such as reduction to SMT. But there are many interesting search strategies between these two extremes that are also easily captured with a metasketch—for example, adaptive concretization [12] 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 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.

3.3 Examples

We illustrate the process of creating metasketches for two optimal synthesis problems: superoptimization [11, 14, 17, 23, 33] and approximate computing [4, 8, 20]. 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. Finding the best approximation of a given reference program boils down to an optimal synthesis problem, in which the cost function encodes the desired performance metric, and the specification constrains the output of the synthesized program to stay within a given margin of error of the reference output. We show three simple metasketches for these applications. Despite their simplicity, however, these 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).

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

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 $\kappa : \text{SYN} \to \mathbb{N}$ measures the number of conditionals and applications of built-in operators.

Superoptimization Figure 2 shows a basic formulation of a superoptimization metasketch, in which the goal is to find a shortest program in SSA form composed of operators from a given set—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) *structural constraints*. In this case, the structural 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 most recent syntax-guided synthesis competition [3].

Adaptive Superoptimization Superoptimization (e.g., [14, 23]) 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:

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

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 structural constraint to $S_{i,j}$ that forces the j^{th} operator to be used by 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 [8] 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 just 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.

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

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

$S_{1,n}$ =
```
(lambda (x_1 ... x_m)
  (poly_n x_1 ... x_m))
```

$S_{k>1,n}$ =
```
(lambda (x_1 ... x_m)
  (if (bnd x_1 ... x_m)
      (poly_n x_1 ... x_m) ; 1st piece
      (...
        (if (bnd x_1 ... x_m)
            (poly_n x_1 ... x_m)
            (poly_n x_1 ... x_m) ; kth piece
            )...)))
```

$(bnd\ x_1 \ldots x_m)$ =
```
(and (< x_1 (??)) ... (< x_n (??)))
```

$(poly_n\ x_1 \ldots x_m)$ =
```
(+ (* (??) (expt x_1 n)) ...
   (* (??) (expt x_m n)) ...
   (* (??) (expt x_1 1)) ...
   (* (??) (expt x_m 1)) ...
   (??))
```

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.

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 $35\times$ 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 a new incremental form of counterexample-guided inductive synthesis (CEGIS) [25], 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 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 search runs N local solvers in parallel, each executing $\exists\forall$SOLVEASYNC on a logical encoding of the synthesis prob-

```
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 N 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:        i ← 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}; i ← i + 1
25:        else if result = UNSAT then                   ▷ No (more) solutions to S
26:            KILLSOLVER(S)
27:            R ← R \ {S}; V ← V ∪ {S}; i ← i + 1
28:        if i > 0 then
29:            N ← TAKE(S \ (R ∪ V), i)
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 N local solvers (∃∀SOLVEASYNC), each running 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                         ▷ Keep iterating immediately
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.

lem for a particular sketch S from $\mathcal{S}$. Our algorithm assumes the existence of a procedure (as provided by, for example, Sketch [26] or ROSETTE [30]) 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. 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 this cost 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 this new best cost. 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 a sketch S, then by Equation (1), that sketch has no solutions with cost less than c^*. Therefore, when the local search for that sketch 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 [26] 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), $\exists\forall$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 new problem, the search terminates (lines 19–20). The algorithm blocks forever if it solves all the constraints it has received so far, and no new constraints are sent to it.

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 $\exists\forall$SOLVEASYNC. Next, we use the local soundness result to establish 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 define a compact metasketch, which places a simple compactness requirement on the gradient function g. This requirement is true of all metasketches with finite search spaces, and, in practice, 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 $\exists\forall$SOLVEASYNC on the $\exists\forall$ synthesis query for a given sketch S with respect to the correctness specification ϕ. To show soundness of the global search, we need to show simply that $\exists\forall$SOLVEASYNC never returns a result that is invalid for ϕ (Lemma 1). Completeness of $\exists\forall$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 $\exists\forall$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 $\exists\forall$SOLVEASYNC)**.** *Let $\exists\eta.\ \forall x.\ \psi(\eta, x)$ be the problem with which $\exists\forall$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 $\exists\forall$SOLVEASYNC sends a result $\langle \mathit{id}, \text{SAT}, h\rangle$ 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 ($\exists\forall$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 $\bigwedge_{\varphi\in\Psi}\bigwedge_{z\in Z}\varphi(\eta, x := z) \wedge \bigwedge_{\varphi\in\Psi}\varphi(\eta, x := z') \wedge \bigwedge_{z\in Z'}\varphi'(\eta, x := z)$, which is equivalent to $\bigwedge_{\varphi\in\Psi'}\bigwedge_{z\in Z'}\varphi(\eta, x := z)$. □

Lemma 3 (Completeness of $\exists\forall$SOLVEASYNC)**.** *Let $\exists\eta.\ \forall x.\ \psi(\eta, x)$ be the problem with which $\exists\forall$SOLVEASYNC is initialized. Suppose that $\exists\forall$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 $\exists\forall$SOLVEASYNC will eventually send a result $\langle \text{id}, \mathit{SAT}, h\rangle$.*

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 complete. □

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. 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:

- S is launched by a call to $\exists\forall$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, and $\forall x.\, \phi(x, [\![P]\!](x))$ is valid modulo $\mathcal{T}$.*

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 $\kappa(P') < c$. By the definition of the gradient function, the sketch S' is not filtered out on line 17 once SYNTHESIZE finds P. Since SYNTHESIZE returns a program of cost c (by our hypothesis), then every running sketch $S \in \mathcal{R} \cap \mathcal{S}$ will receive TOSMT$(\kappa(S[H := e])) < c$ as its last constraint (lines 20–21). By completeness of local search, all running sketches will terminate. As a result, every unexplored sketch $S \in \mathcal{S} \setminus (\mathcal{R} \cup \mathcal{V})$ will be launched with the constraint TOSMT$(\kappa(S[H := e])) < c$ (line 33), without receiving any other constraints, and so will also terminate. The solver for S' will therefore run to completion after P is found, and it will emit a solution of cost less than 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 SYNAPSE, built on top of the ROSETTE language [29, 30], which extends Racket with features for program synthesis and verification using an underlying SAT or SMT solver [7, 31]. 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. Sharing counterexamples allows solvers to skip early CEGIS iterations and therefore reduce the number of solver queries, but it can also make queries larger and therefore slower. 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. But 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.

5. Evaluation

To demonstrate that SYNAPSE effectively solves optimal synthesis problems expressed as metasketches, we evaluated it on three 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 fragmentatation 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.

5.1 Benchmarks

Table 1 shows the benchmarks used in our evaluation. The benchmark problems come from two sources: 54 problems from three categories of the recent syntax-guided synthesis (SyGuS) competition [3], and 7 problems from recent approximate computing literature [8]. We developed metasketches for each category; Section 3.3 described some of these metasketches.

Hacker's Delight. The first two categories contain 20 bit-manipulating problems, used as benchmarks in previous synthesis work [11], appearing in two different difficulties. For each problem in difficulty d0, the superoptimization metasketch includes only the bitvector operators that appear in the reference implementation. For problems in difficulty d5, the metasketch includes all bitvector operators.

Array Search. The third category contains 14 array search problems from the syntax-guided synthesis competition [3]. The problem arraysearch-n is to synthesize a program that takes as input a sorted array of size n and a search key, and returns the index of the key in the input array, or zero if it does not appear. 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.

Parrot. The fourth category contains 7 problems drawn from the approximate computing literature [8]. 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 problems: for those that use math functions, we use a piecewise polynomial approximation, and for others the adaptive superoptimization metasketch that optimizes against a static instruction cost model.

Setup. We performed all experiments on a 36-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.

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 problems 19 and 20, which none of the SyGuS solvers could solve.

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 significantly better than the SMT-based SyGuS solvers: the symbolic solver [2, 11] times out on all 20 problems, while the Sketch-based [26] solver can only solve problems 1–8. The winner of the SyGuS competition 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 was the Sketch-based [26] solver, which could only solve these problems up to length 7. The enumerative solver, which won the syntax-guided synthesis competition, could only solve lengths 2 and 3.

SYNAPSE is also able to solve all seven Parrot problems. For comparison, we attempted to solve the Parrot benchmarks using both SyGuS solvers and the Stoke stochastic superoptimizer [23], without success. This result is unsurprising: Parrot problems require synthesizing constants, which is only feasible for the symbolic SyGuS solver, but the Hacker's Delight benchmarks show that the symbolic solver does not scale beyond small problems.

5.3 Does the fragmentatation 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, 6, and 8 threads. We omit Hacker's Delight at difficulty d0 because these small benchmarks do not benefit from parallelization. Figure 7 shows the resulting parallel solving performance.

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 in parallel program synthesis [12]. For simpler problems such as Hacker's Delight, the results are dominated by parallel overheads and limitations of our current implementation.[4] We anticipate future work on making better use of parallel resources that will improve these overheads.

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 programs.

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. The left y-axis plots the number of sketches completed by the search so far, and the number of sketches remaining to search (which can be infinite, since the space of candidate programs is infinite). The right y-axis plots the cost of the best candidate solution so far. The search discovers the optimal solution after 194 s with cost 6. However, the gradient function then returns 55 sketches that may contain cheaper candidate programs. The search spends another 1,046 s exploring each of these sketches to prove they do not contain such solutions.

[4] SYNAPSE currently invokes the ROSETTE symbolic compilation in a single main thread of control, which then launches a local solver thread. With large numbers of solver threads, many sketches must be compiled in the main thread before any are solved.

Benchmark Suite	Problems	Source	Problem Descriptions	Metasketch	Cost Function
Hacker's Delight d0	20	SyGuS [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 [3]	Bit-manipulating programs, with sketches in the metasketch containing all operators from the theory of bitvectors.	Superoptimization	Program length
Array Search	14	SyGuS [3]	Search a sorted array of size n for a given element and return its index	Array programs	Expression depth
Parrot	7	Parrot [8]	Approximate computing kernels: kmeans and sobel ($\times$2 convolution matrices)	Adaptive super-optimization	Static cost model
			Approximate computing kernels: fft ($\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.

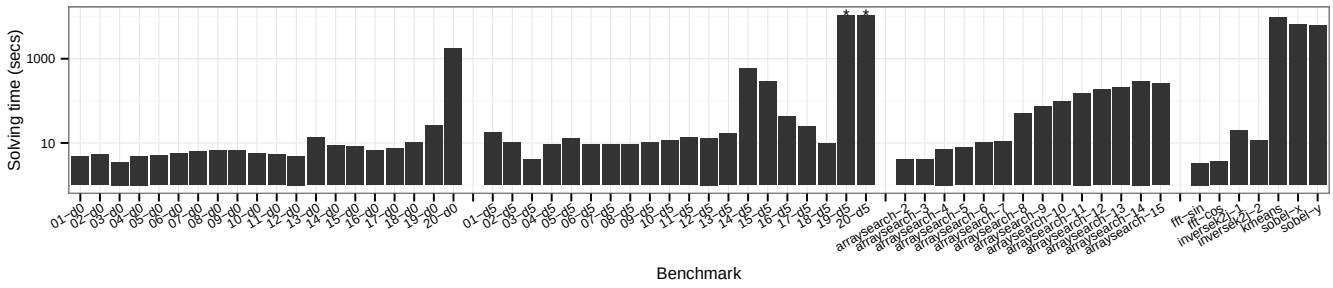

Figure 6. Sequential solving performance for all benchmarks. Asterisks are benchmarks that timed out after one hour.

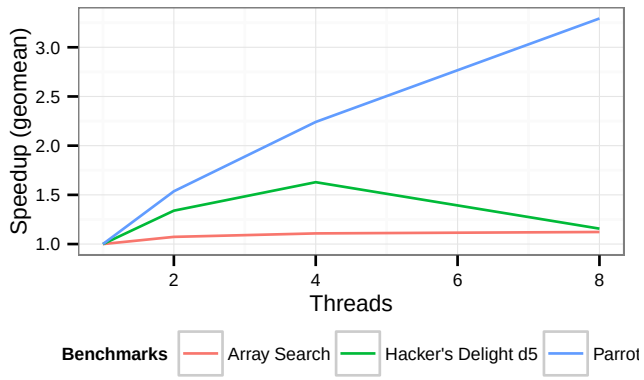

Figure 7. Parallel solving performance for Hacker's Delight difficulty 5 and Parrot benchmarks. While Parrot sees substantial parallel speedup, Hacker's Delight sees slowdown, because the solving performance is dominated by parallel overhead.

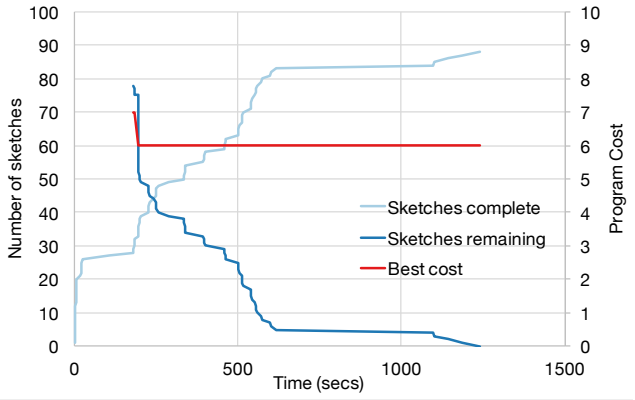

Figure 8. Search progress for the sobel-y benchmark on 4 threads. When solutions are discovered, the size of the remaining search space to explore drops significantly. SYNAPSE finds the optimal solution after 194 s, but must spend an additional 1,046 s proving it optimal.

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. First, the global search can exchange counterexamples between local searches, which can improve their performance by reducing the number of CEGIS iterations. 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 on the Hacker's Delight and Parrot benchmarks for a single-threaded search. For the simpler Hacker's Delight problems, the optimizations improve performance by 10%; for larger Parrot problems, the increase is almost 50%. The primary cause of the speedup is different between the two benchmark suites. The simpler benchmarks benefit most from exchanging counterexamples, as this reduces the number of

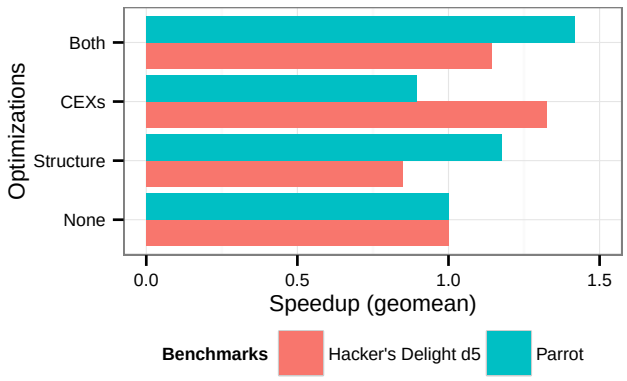

Figure 9. Effect of optimizations on SYNAPSE's performance for a single-threaded search. SYNAPSE can exchange counterexamples (CEXs) between CEGIS loops, and can impose structure constraints that prevent different local searches traversing some semantically-equivalent programs.

CEGIS iterations, each of which is fast for these small problems. For larger benchmarks, the effect of the counterexample exchange is smaller, as the local search spends most of its time in only a few late CEGIS iterations. Instead, these larger benchmarks benefit more from structure constraints, which rule out significant (semantic) overlap among the individual sketches that comprise their metasketches.

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 mentioned 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 dynamic cost functions with two simple 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 $\sum_{i=1}^{n}(y_i - f(x_i))$. We implemented a modified version of the piecewise polynomial metasketch presented in Section 3.3 to perform least-squares regression. We defined the cost function to be the least-squares objective function, and used the trivial gradient function $g(c) = \mathcal{S}$. This cost function is dynamic because it requires evaluating the synthesized program f at each x_i in the data set. Because this metasketch is not compact (Def. 3), we provided a bounded set 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. SYNAPSE synthesized the polynomial

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

in 60 seconds as the optimal solution to this problem. Because the piecewise function metasketch includes polynomial sketches with varying numbers of pieces and branches, SYNAPSE implicitly performs model selection when solving.

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 bounded 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 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)))
```

Notably, neither solution is optimal with respect to both cost functions. SYNAPSE finds each solution in a few seconds.

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 in 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 [21]; generation of high-performance data-parallel code [27, 29]; generation of efficient web layout engines [19]; education [1]; and end-user programming [10].

Our work builds on recent advances in syntax-guided synthesis (e.g., [2, 11, 13, 23, 29, 32]), which are based on counterexample-guided search [26]. 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 [32], symbolic solving [11, 13, 26, 29], and stochastic search [23]. 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.

A number of SyGuS solvers implicitly minimize (or nearly minimize) a fixed objective function. A bottom-up enumerative solver [32] implicitly minimizes program length: shorter solutions will be discovered before longer ones. Some symbolic synthesis algorithms [11] use a library of components, which a synthesized program is allowed to use only a fixed number of times. This bound on component reuse implicitly directs the search towards shorter programs (but not the shortest program): the bound will only be increased if there is no program that satisfies the specification within the current bound. In both cases, however, the optimization is implicit, and not easily extended to different cost functions.

Domain-Specific Synthesizers. Optimality is desirable in a number of synthesis applications, and so domain-specific synthesizers often implement an optimization strategy. Chlorophyll [21]

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. [9] 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 [28] 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 and in SMT problems in general. Chaudhuri et al. propose a smoothed proof search technique for synthesizing parameter holes in a program while optimizing a quantitative objective [5]. 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 only holds 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 [16] 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 [6, 24]. Unlike Symba, our approach supports non-linear cost functions (in, e.g., 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 explicitly rules out this possibility. Integrating Symba with our approach is a promising direction for future work.

7. Conclusion

We presented a generic framework for optimal synthesis that we instantiated with a metasketch, which defines a fragmentation of the space of candidate programs. Metasketches provide abstractions for efficient optimal synthesis performed by two novel cooperating solving algorithms. A global optimizing search coordinates the activities of the local searches, which run in parallel executing an incremental form of counterexample-guided inductive synthesis. Together, metasketches and the solving algorithms allow for efficient traversal of a space of candidate programs while searching for an optimal solution.

The metasketch abstraction allows programmers to direct the high level strategy of the synthesis process. The metasketches we designed were able to fragment candidate spaces in a variety of domains, showing the generality of the abstraction. While other program synthesizers perform some fragmentation implicitly (for example, by iteratively generating longer sketches until a solution is found), metasketches make the fragmentation explicit and allow the solver to reason about it directly. Moreover, metasketches allow programmers to express complex optimal synthesis problems involving an unbounded number of sketches and dynamic cost functions that reason about program behavior. We implemented our approach in SYNAPSE and showed that it effectively solves optimal synthesis problems ranging from superoptimization to approximation of computational kernels. Our results also demonstrate that, due to the strategic search that metasketches enable, SYNAPSE solves classic synthesis problems that state-of-the-art algorithms cannot.

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 *Marktoberdorf*, 2014.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] 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.

[21] 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.

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

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

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

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

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

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

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

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

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

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

[32] 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.

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