Top-Down Synthesis For Library Learning

MATTHEW BOWERS, Massachusetts Institute of Technology, USA
THEO X. OLAUSSON, Massachusetts Institute of Technology, USA
CATHERINE WONG, Massachusetts Institute of Technology, USA
GABRIEL GRAND, Massachusetts Institute of Technology, USA
JOSHUA B. TENENBAUM, Massachusetts Institute of Technology, USA
KEVIN ELLIS, Cornell University, USA
ARMANDO SOLAR-LEZAMA, Massachusetts Institute of Technology, USA

This paper introduces *corpus-guided top-down synthesis* as a mechanism for synthesizing library functions that capture common functionality from a corpus of programs in a domain specific language (DSL). The algorithm builds abstractions directly from initial DSL primitives, using syntactic pattern matching of intermediate abstractions to intelligently prune the search space and guide the algorithm towards abstractions that maximally capture shared structures in the corpus. We present an implementation of the approach in a tool called Stitch and evaluate it against the state-of-the-art deductive library learning algorithm from DreamCoder [Ellis et al. 2021]. Our evaluation shows that Stitch is 3-4 orders of magnitude faster and uses 2 orders of magnitude less memory while maintaining comparable or better library quality (as measured by compressivity). We also demonstrate Stitch's scalability on corpora containing hundreds of complex programs, which are intractable with prior deductive approaches, and show empirically that it is robust to terminating the search procedure early—further allowing it to scale to challenging datasets by means of early stopping.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: Program Synthesis, Library Learning, Abstraction Learning

ACM Reference Format:
Matthew Bowers, Theo X. Olausson, Catherine Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. 2022. Top-Down Synthesis For Library Learning. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (June 2022), 29 pages.

1 INTRODUCTION

One way programmers manage complexity is by hiding functionality behind functional abstractions. For example, consider the graphics programs at the bottom of Fig. 1A. Each uses a generic set of drawing primitives and renders a technical schematic of a hardware component, shown to the left of each program. Faced with the task of writing *more* of these rendering programs, an experienced human programmer likely would not continue using these low-level primitives. Instead, they would introduce new functional abstractions, like the one at the bottom of Fig. 1B which renders a regular polygon given a size and number of sides. Useful abstractions like these allow more concise and legible programs to render the existing schematics. More importantly, well-written abstractions should generalize, making it easier to write new graphics programs for similar graphics tasks.

Recently, the *program synthesis* community has introduced new approaches that can mimic this process, automatically building a library of functional abstractions in order to tackle more complex

Authors' addresses: Matthew Bowers, Massachusetts Institute of Technology, Cambridge, MA, USA, mlbowers@mit.edu; Theo X. Olausson, Massachusetts Institute of Technology, Cambridge, MA, USA, theoxo@mit.edu; Catherine Wong, Massachusetts Institute of Technology, Cambridge, MA, USA, catwong@mit.edu; Gabriel Grand, Massachusetts Institute of Technology, Cambridge, MA, USA, grandg@mit.edu; Joshua B. Tenenbaum, Massachusetts Institute of Technology, Cambridge, MA, USA; Kevin Ellis, Cornell University, Ithaca, NY, USA; Armando Solar-Lezama, Massachusetts Institute of Technology, Cambridge, MA, USA.

2022. 2475-1421/2022/6-ART1 $15.00
https://doi.org/

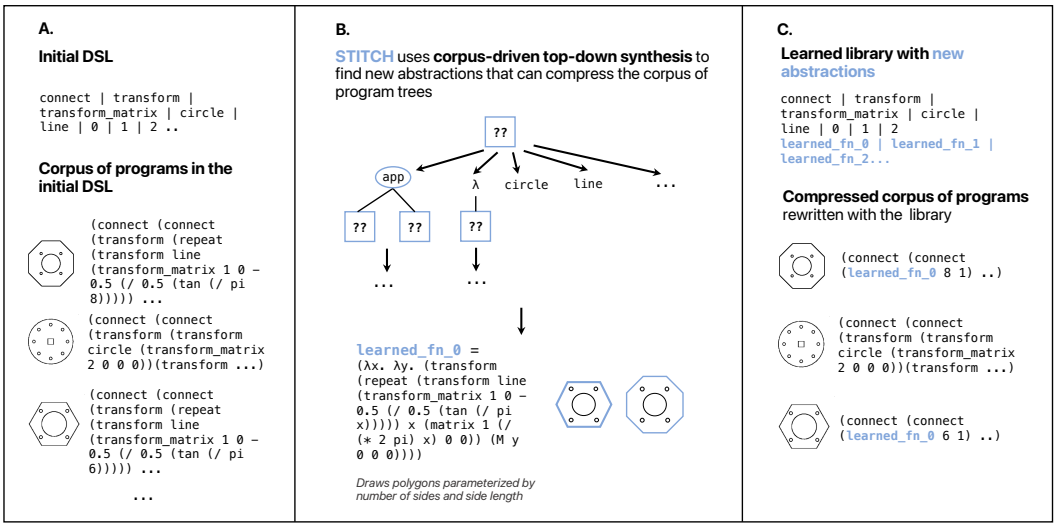

Fig. 1. (**A**) Given an initial DSL of lower-level primitives and a corpus of programs written in the initial DSL, (**B**) Stitch uses a fast and memory efficient *corpus-guided top-down search* algorithm to construct function abstractions which maximally capture shared structure across the corpus of programs. (**C**) Stitch automatically rewrites the initial corpus using the learned library functions.

synthesis problems [Dechter et al. 2013; Ellis et al. 2020, 2021; Lázaro-Gredilla et al. 2019; Shin et al. 2019]. One popular approach to library learning is to search for common tree fragments across a corpus of programs, which can be introduced as new abstractions [Dechter et al. 2013; Lázaro-Gredilla et al. 2019; Shin et al. 2019]. Ellis et al. 2021, however, introduces an algorithm that reasons about variable bindings to abstract out well-formed *functions* instead of just tree fragments. While it produces impressive results, the system in Ellis et al. 2021 takes a *deductive* approach to library learning that is difficult to scale to larger datasets of longer and deeper input programs. This deductive approach attempts to refactor existing programs to expose shared structure, and must represent and evaluate a large space of proposed refactorings to identify common functionality across the corpus. Prior, work such as Ellis et al. 2020, 2021, approaches this challenge by combining a dynamic bottom-up approach to refactoring with version spaces to more efficiently search over the refactored programs. However, these deductive approaches face daunting memory and search requirements as the corpus scales in size and complexity.

This paper introduces an alternate approach to library learning: instead of taking a *deductive* approach based on *refactoring the corpus*, we *directly synthesize abstractions*. We call this approach *corpus-guided top-down synthesis*, and it is based on the insight that when applied to the task of synthesizing abstractions, top-down search can be guided precisely towards discovering shared abstractions over a set of existing training programs — at every step of the search, syntactic comparisons between a partially constructed abstraction and the set of training programs can be used to strongly constrain the search space and direct the search towards abstractions that capture the greatest degree of shared syntactic structure.

We implement this approach in Stitch[1] , a corpus-guided top-down library learning tool written in Rust (Fig. 1). We evaluate Stitch through a series of experiments (Section 6), and find that: Stitch learns libraries of comparable quality to those found by the algorithm of Ellis et al. 2021 on

[1]https://github.com/mlb2251/stitch

their iterative bootstrapped library learning task, while being 3-4 orders of magnitude faster and using 2 orders of magnitude less memory (Section 6.1); Stitch learns high quality libraries within seconds to single-digit minutes when run on corpora containing a few hundred programs with mean lengths between 76–189 symbols (sourced from Wong et al. 2022), while even the simplest of these corpora lies beyond the reach of the algorithm of Ellis et al. 2021 (Section 6.2); and that Stitch degrades gracefully when resources are constrained (Section 6.3). We also perform ablation studies to expose the relative impact of different optimizations in Stitch (Section 6.4). Finally, we show that Stitch is complementary to deductive rewrite approaches (Section 6.5).

In summary, our paper makes the following contributions:

(1) **Corpus-guided top-down synthesis (CTS)**: A novel, strongly-guided branch-and-bound algorithm for synthesizing function abstractions (Sec. 3).

(2) **CTS for program compression**: An instantiation of the CTS framework for utility functions favoring abstractions that compress a corpus of programs (Sec. 4).

(3) **Stitch**: A parallel Rust implementation of CTS for compression that achieves 3-4 orders of magnitude of speed and memory improvements over prior work, and the analysis of its performance, scaling, and optimizations through several experiments (Sec. 6).

2 OVERVIEW

In this section, we build intuition for the algorithmic insights that power Stitch. As a running example, we focus on learning a single abstraction from the following corpus of programs:

$$\begin{array}{l} \lambda x.\ \texttt{+ 3 (* (+ 2 4) 2)} \\ \lambda xs.\ \texttt{map } (\lambda x.\ \texttt{+ 3 (* 4 (+ 3 x)))}\ \texttt{xs} \\ \lambda x.\ \texttt{* 2 (+ 3 (* x (+ 2 1)))} \end{array} \tag{1}$$

The notion of what a *good* abstraction is depends on the application, so our algorithm is generic over the utility function that we seek to maximize. Following prior work [Dechter et al. 2013; Ellis et al. 2021; Shin et al. 2019] we focus on *compression* as a utility function: a good abstraction is one which minimizes the size of the corpus when rewritten with the abstraction. The utility function used by Stitch is detailed in Section 4 and corresponds exactly to the compression objective, but at a high level the function seeks to maximize the product of *the size of the abstraction* and the *number of locations* where the abstraction can be used. This product balances two key features of a highly compressive abstraction: the abstraction should be general enough that it applies in many locations, but specific enough that it captures a lot of structure at each location.

The optimal abstraction that maximizes our utility in this example is:

$$\texttt{fn0} = \lambda\alpha.\ \lambda\beta.\texttt{(+ 3 (* } \alpha\ \beta\texttt{))} \tag{2}$$

And the shared structure that this abstraction captures is highlighted in blue in Eq. (1). When rewritten to use this abstraction, the size of the resulting programs is minimized:

$$\begin{array}{l} \lambda x.\ \texttt{fn0 (+ 2 4) 2} \\ \lambda xs.\ \texttt{map } (\lambda x.\ \texttt{fn0 4 (+ 3 x))}\ \texttt{xs} \\ \lambda x.\ \texttt{* 2 (fn0 x (+ 2 1))} \end{array} \tag{3}$$

Stitch synthesizes the optimal abstraction directly through *top-down search*. Top-down search methods construct program syntax trees by iteratively refining partially-completed programs that have unfinished *holes*, until a complete program that meets the specification is produced [Balog et al. 2016; Ellis et al. 2021; Feser et al. 2015; Nye et al. 2021; Polikarpova et al. 2016; Shah et al. 2020]. This kind of search is often made efficient by identifying branches of the search tree that

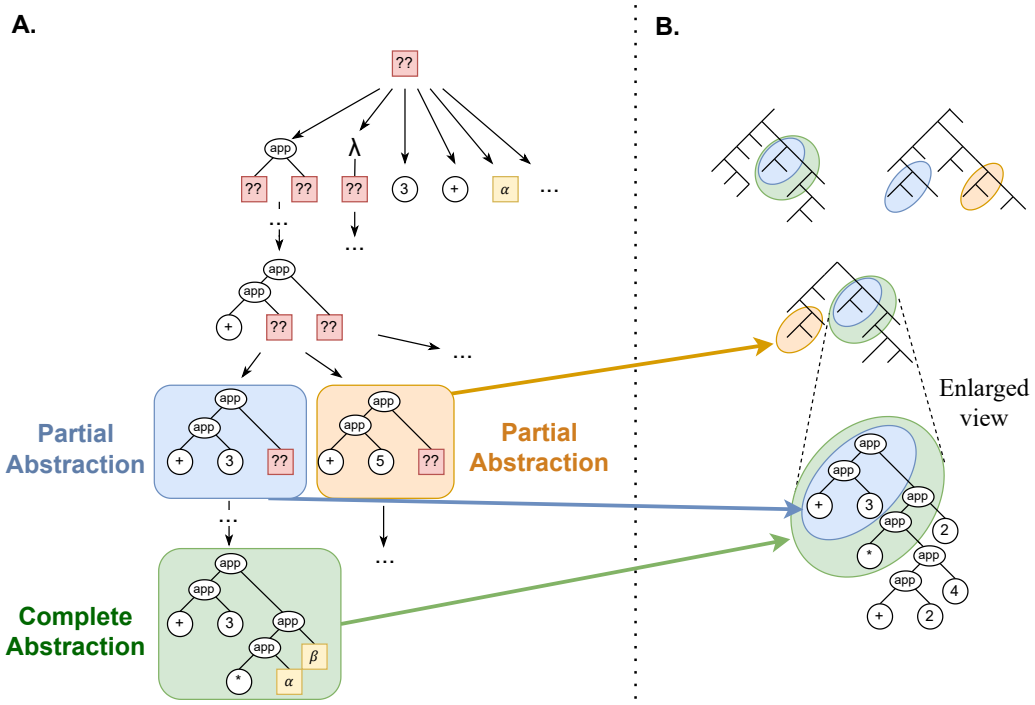

Fig. 2. (**A**) Schematic of the *corpus-guided top-down search* search. Partial abstractions can contain *holes* indicating unfinished subtrees, denoted ??, while complete abstractions do not contain holes. (**B**) Partial and complete abstractions *match* at locations in the corpus. The blue partial abstraction can be expanded into the green complete abstraction so the green abstraction matches in a *subset* of places that the blue abstraction does. For complete abstractions, *matching* indicates that the subtree can be rewritten to use the abstraction.

can be pruned away because you can efficiently prove that none of the programs in that branch can be correct. We aim to apply a similar idea to the problem of synthesizing good abstractions; the idea is to explore the space of functions in the same top-down way, but in search of a function that maximizes the utility measure. The key observation is that this new objective affords even more aggressive pruning opportunities than the traditional correctness objective, allowing us to synthesize optimal abstractions very efficiently. We call this approach *corpus-guided top-down search* (CTS).

Corpus-guided top-down search. Like other top-down synthesis approaches, our algorithm explores the space of abstractions by repeatedly refining abstractions with holes, as illustrated in Fig. 2A. We call abstractions with holes *partial abstractions* in contrast to *complete abstractions* which have no holes. Our top-down algorithm searches over abstraction *bodies*, so to synthesize the optimal abstraction `fn0` in our running example, we synthesize its body: `(+ 3 (* α β))`.

We say that a partial abstraction *matches* at a subtree in the corpus if there's a possible assignment to the holes and arguments that yields the subtree. For example, consider the subtree `(+ 3 (* (+ 2 4) 2)` from the first program in our running example, shown as a syntax tree at the bottom of figure 2B. The partial abstraction `(+ 3 ??)` shown in blue matches here with the hole `??` = `(* (+ 2 4) 2)`, and the complete abstraction `(+ 3 (* α β))` matches here with α = `(+ 2 4)` and β = 2. For complete abstractions, matching corresponds to being able to use the abstraction to rewrite this

subtree, resulting in compression. We refer to the set of subtrees at which a complete or partial abstraction matches as the set of *match locations*.

In traditional top-down synthesis, a branch of search can be safely pruned by proving that a program satisfying the specification cannot exist in that branch. In CTS, we can safely prune a branch of search if we can prove that it cannot contain the optimal abstraction. One way to prove this is by computing an upper bound on the utility of abstractions in the branch, and discarding the branch if we have previously found an abstraction with a utility that is higher than this bound. One way to compute this upper bound is to overapproximate the set of match locations, then upper bound the compressive utility gain from each match location.

Our key observation to compute this bound is that during search, expanding a hole in a partial abstraction yields an abstraction that is more precise and thus matches at a *subset* of the locations that the original matched. This *subset observation* is depicted in Fig. 2 where the refinement of the partial abstraction `(+ 3 ??)` (blue) to the goal `(+ 3 (* α β))` (green) results in a larger abstraction that matches at a subset of the locations (match locations shown at the top of Fig. 2B). An important consequence of this is that the match locations of a partial abstraction serves as an over-approximation of the match locations of any abstraction in this branch of top-down search.

To upper bound the compressive utility gained by rewriting at a match location, notice that the most compressive abstraction that matches at a subtree is the constant abstraction corresponding to the subtree itself. For example, the largest possible abstraction at `(+ 3 (* (+ 2 4) 2)` is just `(+ 3 (* (+ 2 4) 2)`. In other words, the size of an expression is a strict upper bound on how much compression can be achieved. Thus we get the upper bound for some partial abstraction $A_{??}$:

$$U_{\text{upperbound}}(A_{??}) = \sum_{e \in \text{matches}(A_{??})} \text{size}(e) \tag{4}$$

Equipped with this bound, our algorithm performs a branch-and-bound style top-down search, where at each step of search it discards all partial abstractions that have utility upper bounds that are less than the utility of the best complete abstraction found so far.

Our full algorithm presented in Section 3 has some additional complexity. It handles rewriting in the presence of variables soundly, it uses an exact utility function that accounts for additional compression gained by using the same variable in multiple places, and it handles situations where match locations overlap and are mutually exclusive. We also introduce two other important forms of pruning while maintaining the optimality of the algorithm, and we use the upper bound as a heuristic to prioritize more promising branches of search first.

Building up abstraction libraries. The top-down search algorithm described above yields a single abstraction. However, we can easily run this algorithm for multiple iterations on a corpus of programs to build up an entire *library of abstractions*. Fig. 3 illustrates the power of this kind of iterative library learning, which interleaves compression and rewriting. At each iteration, Stitch discovers a single abstraction that is used to rewrite the entire corpus of programs. Successive iterations therefore yield abstractions that build hierarchically on one another, achieving increasingly higher-level behaviors. As the library grows to contain richer and more complex abstractions, individual programs shrink into compact expressions.

Structure of the paper. In the subsequent sections, we formalize our corpus-guided top-down search algorithm (Section 3), its application to the problem of compression (Section 4), and how it may be layered on top of data structures such as version spaces (Section 5). We then report experimental results (Section 6) showcasing both diverse library learning settings as well as ablations of Stitch's search mechanisms. Finally, we conclude by situating Stitch within the landscape

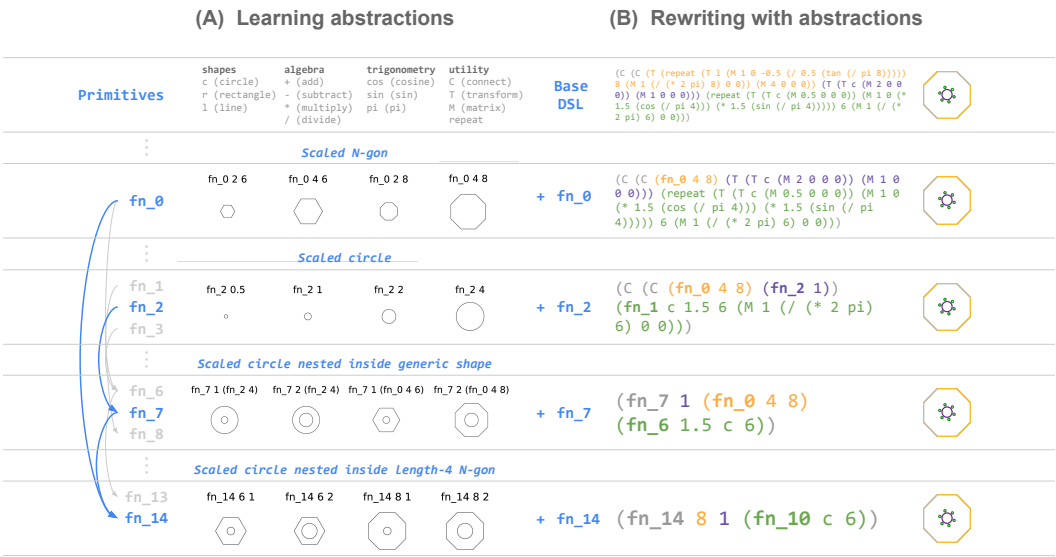

Fig. 3. Visualization of Stitch library learning on the nuts-bolts subdomain from Wong et al. 2022. (**A**) From the base DSL primitives (top row), Stitch iteratively discovers a series of abstractions that compress programs in the domain. Arrows demonstrate how abstractions from selected iterations build on one another to achieve increasingly higher-level behaviors. (**B**) Rewriting a single item from the domain with the cumulative benefit of discovered abstractions yields increasingly compact expressions. Colors indicate correspondence between object parts and program fragments: orange = outer octagon, green = ring of six circles, purple = inner circle.

of related (Section 7) and future (Section 8) work in the areas of library learning and program synthesis.

3 CORPUS-GUIDED TOP-DOWN SEARCH

We first provide the definitions necessary to understand our problem then introduce our algorithm.

Grammar. Our algorithm operates on lambda-calculus expressions with variables represented through de Bruijn indices [de Bruijn 1972]; expressions come from a context-free grammar of the form $e ::= \lambda.\ e' \mid e'\ e'' \mid \$i \mid t$, where $t \in \mathcal{G}_{\texttt{sym}}$ refers to the set of built-in primitives in the domain-specific language. For example, in an arithmetic domain $\mathcal{G}_{\texttt{sym}}$ would include operators like + and constants like 3. We say that e' is a subexpression of e if $e = C[e']$, where C is a context as defined in the contextual semantics of Felleisen and Hieb [1992]. An expression is *closed* if it has no free variables, in which case the expression is also a *program*. A set of programs $\mathcal{P}$ is a *corpus*.

Abstraction. Given a grammar $\mathcal{G}$, we define the *abstraction grammar* $\mathcal{G}_A$ as $\mathcal{G}$ extended to include *abstraction variables*, denoted by Greek letters α, β, etc. Formally, $A ::= \lambda.\ A' \mid A'\ A'' \mid \$i \mid t \mid \alpha$ where $t \in \mathcal{G}_{\texttt{sym}}$. A term A from this grammar represents the *body* of an abstraction; e.g. the abstraction $\lambda\alpha.\ \lambda\beta.\ (+\ \alpha\ \beta)$ is simply represented by the term $(+\ \alpha\ \beta)$ from the language of $\mathcal{G}_A$.

Partial abstraction. A *partial abstraction* $A_{??}$ is an abstraction that can additionally include *holes*. A *hole*, denoted by ??, represents an unfinished subtree of the abstraction. Thus, $A_{??} ::= A \mid ?? \mid \lambda.\ A'_{??} \mid A'_{??}\ A''_{??}$. Any abstraction is thus also a partial abstraction. Given a grammar $\mathcal{G}$, the grammar of partial abstractions is denoted by $\mathcal{G}_{A_{??}}$. Each hole in a partial abstraction A can be referred to by a *unique* index, which can be explicitly written as $??_i$.

$$\text{U-AbsVar}\ \frac{}{\textsc{LambdaUnify}(\alpha, e) \rightsquigarrow [\alpha \to e]}$$

$$\text{U-Hole}\ \frac{}{\textsc{LambdaUnify}(??_i, e) \rightsquigarrow [??_i \to e]}$$

$$\text{U-App}\ \frac{\begin{array}{c}\textsc{LambdaUnify}(A_1, e_1) \rightsquigarrow l_1 \\ \textsc{LambdaUnify}(A_2, e_2) \rightsquigarrow l_2 \\ l = \textsc{merge}(l_1, l_2)\end{array}}{\textsc{LambdaUnify}((A_1\ A_2), (e_1\ e_2)) \rightsquigarrow l}$$

$$\text{U-Lam}\ \frac{\begin{array}{c}\textsc{LambdaUnify}(A, e) \rightsquigarrow l' \\ l = \textsc{DownshiftAll}(l')\end{array}}{\textsc{LambdaUnify}((\lambda.\ A), (\lambda.\ e)) \rightsquigarrow l}$$

$$\text{U-Same}\ \frac{}{\textsc{LambdaUnify}(e, e) \rightsquigarrow []}$$

$$\downarrow e = \downarrow_0 e$$

$$\downarrow_d \lambda.b = \lambda.\ \downarrow_{d+1} b$$

$$\downarrow_d (f\,x) = (\downarrow_d f\ \ \downarrow_d x)$$

$$\downarrow_d \$i = \begin{cases} \$i, & \text{if } i < d \\ \$(i-1), & \text{if } i > d \\ \&(i-1), & \text{if } i = d \end{cases}$$

$$\downarrow_d \&i = \&(i-1),$$

$$\downarrow_d t = t, \text{ if } t \text{ is a primitive (i.e. } t \in \mathcal{G}_{\texttt{sym}})$$

Fig. 4. (**Left**) Inference rules for lambda-aware unification. (**Right**) Definition of the downshifting operator.

Lambda-aware unification. We introduce *lambda-aware unification* as a modification of traditional unification adapted to our algorithm. $\textsc{LambdaUnify}(A, e)$ returns a map from abstraction variables and holes to expressions $[\alpha_i \to e_i',\ \ldots,\ ??_j \to u_j',\ \ldots]$ such that

$$(\lambda\alpha_i.\ \ldots\ \lambda??_j.\ \ldots A)\ e_i'\ \ldots\ u_j'\ \ldots = e \tag{5}$$

through beta reduction. If no such mapping exists, LambdaUnify fails. The key challenge in defining $\textsc{LambdaUnify}(A, e)$ is that the expression e may be deep inside a program written using de Bruijn indices, so the function must perform some index arithmetic in order to generate its output mapping.

The definition of LambdaUnify is presented in Fig. 4 (left). In U-App, $\textsc{merge}(l_1, l_2)$ merges two $[\alpha_i \to e_i]$ mappings to create a new mapping that includes all bindings from l_1 and l_2, and fails if the same abstraction variable α maps to different expressions in l_1 and l_2. The rule U-Same applies only when the abstraction argument to LambdaUnify is an expression (i.e. it contains no holes or abstraction variables) and is syntactically identical to the expression it is being unified with. In U-Lam, DownshiftAll returns a new mapping with all abstracted expressions e and hole expressions u downshifted by 1 using the $\downarrow$ operator presented in Figure 4 (right), which has been modified from the traditional de Bruijn index operator.

This operator recursively downshifts all free variables in an expression by 1, extended to switch to $\&i$ variables when a downshift would otherwise convert a free variable to a (different, incorrect) bound variable. If an abstracted expression e contains $\&i$ variables at this point then the procedure fails, as $\&i$ variables represent references to lambdas present within the body of the abstraction. Notably, the beta reduction in Eq. (5) must account for hole expressions u_j that contain $\&i$ indices. We thus use a modified form of beta reduction in which $\&i$ indices are *always* upshifted, and at the very end of beta reduction we replace each $\&i$ with $\$i$. With this modification of beta reduction, any ampersands with negative indices will ultimately be shifted back to positive indices during reduction.

To understand the definition of the $\downarrow$ operator and the motivation for the $\&i$ indices, consider what happens when you run.

$$\textsc{LambdaUnify}(\lambda.\ f\ ??_0,\ \lambda.\ e) \tag{6}$$

By Eq. (5), the goal is to produce a mapping $[??_0 \to u_0']$ such than when replacing $??_0$ with u_0' it produces $(\lambda.\ e)$. In other words, $(\lambda.\ \lambda.\ f\ \$1)\ u_0' = \lambda.\ e$. Now, suppose

$$\textsc{LambdaUnify}(f\ ??_0,\ e) \rightsquigarrow [??_0 \to \lambda.\ \$i] \tag{7}$$

There are three possibilities for i that need to be considered, corresponding to the 3 cases in the definition of $\downarrow_d \$i$. In the first case, when $i = 0$, it means that $\$i$ in Eq. (7) is bound to the lambda in the return mapping. In this case, DownshiftAll in U-Lam should not do anything because if we can replace $??_0$ with $(\lambda.\ \$0)$ in $(f\ ??_0)$ to produce e, the same replacement in $(\lambda.\ f\ ??_0)$ will produce $(\lambda.\ e)$.

The second case is when $i \geq 2$; this means that e has some variables that were defined outside of it and they have been captured by $??_0$. For example, if $i = 2$ it means that $(\lambda.\ f\ \$0)(\lambda.\ \$2) = e$, which implies that $e = f\ (\lambda.\ \$2)$. In this case, the solution to Eq. (6) is $[??_0 \to \lambda.\ \$1]$, since $(\lambda.\ \lambda.\ f\ \$1)(\lambda.\ \$1) = \lambda.\ e$; in other words, when computing $\textsc{LambdaUnify}(\lambda.\ f\ ??_0,\ \lambda.\ e)$ from Eq. (7), $\$i$ has to be downshifted to account for the fact that it will be substituted inside an additional lambda.

The third case, when $i = 1$, is more problematic. In this case, we have that $(\lambda.\ f\ \$0)(\lambda.\ \$1) = e$, which implies that $e = f\ (\lambda.\ \$1)$. This creates a problem, since there is no i such that $(\lambda.\ \lambda.\ f\ \$1)(\lambda.\ \$i) = \lambda.\ e$; the $\$i$ needs to refer to the lambda surrounding e. The downshift operator addresses this through the special treatment of the & index.

Match locations. The set of *match locations* of a partial abstraction $A_{??}$ in a corpus $\mathcal{P}$, denoted $\textsc{Matches}(\mathcal{P}, A_{??})$, is the maximal set of context-expression pairs $\{(C_1, e_1), (C_2, e_2),\ \ldots, (C_k, e_k)\}$ such that $\forall k.\ p = C_k[e_k]$ and $p \in \mathcal{P}$ and $\textsc{LambdaUnify}(e_k, A_{??})$ succeeds. In Section 3.1 we explain how we maintain this set of matches incrementally.

Rewrite Strategies. A corpus $\mathcal{P}$ can be *rewritten* to use a complete abstraction A as follows. We introduce a new terminal symbol t_A into the symbol grammar $\mathcal{G}_{\texttt{sym}}$ to represent the abstraction, and consider it semantically equivalent to $(\lambda\alpha_0.\ \ldots\ \lambda\alpha_k.\ A)$. We then re-express $\mathcal{P}$ in terms of t_A as follows. We can replace the match location $(C, e) \in \textsc{Matches}(\mathcal{P}, A)$ with t_A applied with the argument assignments $[\alpha_i \to e_i] = \textsc{LambdaUnify}(A, e)$ to yield a semantically equivalent expression, $C[(t_A\ e_0\ \ldots\ e_k)]$. Note that since this is a complete abstraction, it contains no holes.

One complication is that rewriting at one match location may *preclude* rewriting at another match location if they overlap with each other. A *rewrite strategy* $\mathcal{R}$ is a procedure for selecting a subset of $\textsc{Matches}(\mathcal{P}, A)$ to rewrite at, in which no matches are mutually exclusive. We refer to this set as $\textsc{RewriteLocations}_{\mathcal{R}}(\mathcal{P}, A)$. We refer to rewriting a corpus $\mathcal{P}$ under an abstraction A with rewrite strategy $\mathcal{R}$ by $\textsc{Rewrite}_{\mathcal{R}}(\mathcal{P}, A)$. We refer to rewriting at only a single particular match location m with abstraction A by $\textsc{RewriteOne}(m, A)$.

Utility. CTS optimizes a user-defined utility function $U_{\mathcal{P},\mathcal{R}}(A)$ which scores an abstraction A given a corpus $\mathcal{P}$ and rewrite strategy $\mathcal{R}$. There are no strict constraints on the form or properties of the utility function. While in Section 4 we will focus on compressive utility functions, our framework does not generally require this.

We say that a rewrite strategy is *optimal* with respect to a utility function if the rewrite strategy chooses locations to rewrite at such that the utility is maximized. A naïve optimal rewrite strategy exhaustively checks the utility of rewriting with each of the $2^{|\textsc{Matches}(\mathcal{P},A)|}$ possible subsets of $\textsc{Matches}(\mathcal{P}, A)$, however depending on the specific utility function computing an optimal or approximately optimal strategy may be significantly more computationally tractable. In Section 4 we detail the rewrite strategy used by Stitch, which is an optimal, linear-time rewrite strategy for compressive utility functions based on dynamic programming.

3.1 Algorithm

Given a corpus $\mathcal{P}$, rewrite strategy $\mathcal{R}$, and utility function $U_{\mathcal{P},\mathcal{R}}(A)$, the objective of CTS is to find the abstraction A that maximizes the utility $U_{\mathcal{P},\mathcal{R}}(A)$.

Expansion. We construct the body of a partial abstraction $A_{??}$ through a series of top-down *expansions* starting from the trivial partial abstraction ??. Given a partial abstraction $A_{??}$ we can *expand* a hole in this partial abstraction using any production rule from the partial abstraction grammar $\mathcal{G}_{A_{??}}$, yielding a new abstraction $A'_{??}$. We denote this expansion by $A_{??} \rightarrow A'_{??}$. In Figure 2A our overall top-down search is depicted as a series of expansions. We say that a complete abstraction A can be *derived* from a partial abstraction $A_{??}$, denoted $A_{??} \rightarrow^* A$, if A is in the transitive reflexive closure of the expansion operation, i.e. there exists a sequence of expansions from $A_{??}$ to A. Any abstraction can be derived from ??.

A naïve approach. The goal of our algorithm is to find the maximum-utility abstraction. One simple, inefficient approach to this is to simply enumerate the *entire* space of abstractions through top down synthesis and return the one with the highest utility. This naïve approach maintains a queue of partial abstractions initialized with just ??. At each step of the algorithm it pops an abstraction from the queue, chooses a hole in it and expands that hole using each possible production rule. Whenever an expansion produces a new partial abstraction it pushes it to the queue, and when it produces a complete abstraction it calculates its utility and updates the best abstraction found so far. Since an abstraction cannot match a program that is smaller than it, the algorithm can stop expanding when all expansions would lead to abstractions larger than the largest program in the corpus. This algorithm will enumerate all possible abstractions exactly once each.

Introducing subsumption pruning. While the naïve approach will find the optimal abstraction, it is extremely inefficient. We can improve on this by allowing the algorithm to choose to *prune* a partial abstraction instead of expanding it, in which case it simply discards the abstraction and chooses another from the queue. Of course, to maintain optimality we must be certain that we *never* prune the branch of search containing the optimal abstraction. To formalize safe pruning, we introduce the notions of *covering* and *subsumption.*

A complete abstraction A'' *covers* another complete abstraction A', written $\text{covers}(A'', A')$, if the utility of A'' is greater than that of A'. A partial abstraction $A'_{??}$ is *subsumed by* another partial abstraction $A''_{??}$ if and only if every complete abstraction that is derivable from $A'_{??}$ is covered by a complete abstraction that is derivable from $A''_{??}$.

Formally, $A''_{??}$ subsumes $A'_{??}$ iff $\forall A'.\ A'_{??} \rightarrow^* A' \implies \exists A''.\ A''_{??} \rightarrow^* A'' \wedge \text{covers}(A'', A')$.

Equipped with this formalism, we claim that it is safe to prune an abstraction $A'_{??}$ if we know that there exists some $A''_{??}$ which subsumes it. $A''_{??}$ need not have been enumerated yet during search and may even have been pruned; simply proving its existence is enough.

Lemma 3.1. *Naïve search augmented with subsumption pruning still finds the optimal abstraction.*

Proof. We proceed with a proof by induction. We seek to prove the equivalent statement that the optimum is never pruned and thus it will be enumerated by the search.

Our inductive hypothesis is that the optimum has not yet been pruned. In the base case this is trivially true, since no pruning has yet taken place. In the inductive step we must prove that a step of pruning maintains the validity of the inductive hypothesis. Recall that a step of pruning will discard some partial abstraction $A'_{??}$ which is subsumed by another abstraction $A''_{??}$. Suppose then for the sake of a contradiction that we did prune the optimum in this step; then the optimum must have been derivable from $A'_{??}$. By subsumption we know that all abstractions derivable from $A'_{??}$ must be covered by some abstraction derivable from $A''_{??}$. Hence, there must exist some abstraction

that covers the optimum. However, since the utility of the optimum is greater than or equal to that of all other abstractions, there is by definition *no* such abstraction; we have thus arrived at a contradiction. □

This lemma ensures the safety of composing together pruning strategies, as long as they all are forms of subsumption pruning, since no instance of subsumption pruning will remove the optimum. Determining which partial abstractions are subsumed by which others is specific to the utility function being used, and in Section 4 we identify two instances of subsumption used when instantiating this framework for a compression-based utility.

Branch-and-bound pruning. We further improve this algorithm to employ branch-and-bound style pruning, in which we bound the maximum utility that can be obtained in a branch of search, and discard the branch if we have previously found a complete abstraction with higher utility than this bound. Our algorithm is generic over the upper bound function $\bar{U}_{\mathcal{P},\mathcal{R}}(A_{??})$ which upper bounds the utility of *any complete abstraction* A that can be derived from $A_{??}$.

Formally, the bound must satisfy $\forall A.\ A_{??} \rightarrow^* A \implies U_{\mathcal{P},\mathcal{R}}(A) \leq \bar{U}_{\mathcal{P},\mathcal{R}}(A_{??})$. While this could trivially be satisfied with $\bar{U}_{\mathcal{P},\mathcal{R}}(A_{??}) = \infty$ for any choice of a utility function, a tighter bound will allow for more pruning. The soundness of branch-and-bound pruning in maintaining the optimality of the solution trivially follows from the fact that the pruned branch only contains complete abstractions that are at most as high-utility as the best abstraction found so far.

While we defer to Section 4 to construct the upper bound used by Stitch, it is worth mentioning a key insight that helps in constructing tight bounds for many utility functions. When bounding the utility of $A_{??}$, it is useful to bound the set of possible locations where rewrites can occur for any abstraction A derived from $A_{??}$. The following lemma provides such a bound:

Lemma 3.2. *Matches*$(\mathcal{P}, A_{??})$ *is an upper bound on the set of locations where rewrites can occur in any* A *derived from* $A_{??}$.

This follows from the fact that as partial abstractions are expanded they become more precise and thus match at a subset of the locations. Since rewriting only happens at a subset of match locations, Matches$(\mathcal{P}, A_{??})$ serves as a bound on the set of locations where rewrites can occur.

An important consequence of Lemma 3.2 is that if a partial abstraction matches at zero locations, then all abstractions derived from it will have zero rewriting locations, so no rewriting can occur. Such branches can therefore be safely pruned.

Improving the search order. Finally, without sacrificing the optimality of this algorithm we can heuristically guide the order in which the space of abstractions is explored. The queue of partial abstractions used in top-down search can be replaced with a priority queue ordered by the upper bound. This way, the algorithm will first explore more promising, higher-bound branches of search, narrowing in on the optimal abstraction more quickly. While the algorithm is flexible to which heuristic is used, we find the upper bound heuristic works well in practice.

Efficient incremental matching. When expanding an abstraction $A_{??}$ to a new abstraction $A'_{??}$, it is easy to compute Matches$(\mathcal{P}, A'_{??})$ since we know it is a subset of Matches$(\mathcal{P}, A_{??})$. Thus, there is no need to perform matching from scratch against every subtree in the corpus to compute the match locations. Instead, we can simply inspect the relevant subtree at each match location of the original abstraction to see which match locations will be preserved by a given expansion. In fact, except when expanding into an abstraction variable α, the sets of match locations obtained by different expansions of a hole will be disjoint.

When expanding to an abstraction variable α, if α is an existing abstraction variable then this is a situation where the same variable is being used in more than one place, as in the *square* abstraction

$(\lambda\alpha.\ *\ \alpha\ \alpha)$. In this case we restrict the match locations to the subset of locations where within the location all instances of α are bound to syntactically identical subtrees.

Additionally, if the user provides a maximum arity limit, then an expansion that causes the abstraction to exceed this limit can be discarded.

Algorithm summary. The CTS algorithm combines all of the aforementioned optimizations in a top-down search algorithm with pruning. In summary, CTS takes as input a corpus $\mathcal{P}$, a rewrite strategy $\mathcal{R}$, a utility function $U_{\mathcal{P},\mathcal{R}}(A)$, and an upper bound function $\bar{U}_{\mathcal{P},\mathcal{R}}(A_{??})$, and returns the *optimal abstraction* with respect to the utility function. CTS performs a top-down search over partial abstractions and prunes branches of search when partial abstractions match at zero locations or are eliminated by branch-and-bound pruning or subsumption pruning. CTS is made further efficent through the aforementioned *search-order heuristic* and *efficient incremental matching*. None of these optimizations sacrifice the optimality of the abstraction found.

Since the algorithm maintains a best abstraction so-far, it can also be terminated early, making it an *any-time algorithm*. We also note that this algorithm is amenable to parallelization, as different workers can process items from the priority queue in parallel, especially since branch-and-bound pruning remains sound even when best abstraction found so-far is not always up-to-date across workers. Finally, to learn a library of abstractions, CTS can be run repeatedly, adding one abstraction at a time to the library and rewriting the corpus with each abstraction as it is learned before running CTS again. A listing of the full algorithm is provided in Appendix A.

4 APPLYING CORPUS-GUIDED TOP-DOWN SEARCH TO COMPRESSION

Having presented the general framework and algorithm of corpus-guided top-down search, we now instantiate this framework for optimizing a *compression* metric.

4.1 Utility

In compression we seek to minimize some measure of the size, or more generally the *cost*, of a corpus of programs after rewriting them with a new abstraction. As in prior work [Ellis et al. 2021] we penalize large abstractions by including abstraction size in the utility. The compressive utility function for a corpus $\mathcal{P}$ and abstraction A is given below.

$$U_{\mathcal{P},\mathcal{R}}(A) \triangleq -\text{cost}(A) + \text{cost}(\mathcal{P}) - \text{cost}(\textsc{Rewrite}_{\mathcal{R}}(\mathcal{P}, A)) \tag{8}$$

Here, $\text{cost}(\cdot)$ is a cost function of the following form:

$$\begin{aligned}
&\text{cost}(\lambda.\ e') = \text{cost}_{\lambda} + \text{cost}(e')\\
&\text{cost}(e'e'') = \text{cost}_{\mathsf{app}} + \text{cost}(e') + \text{cost}(e'')\\
&\text{cost}(\$i) = \text{cost}_{\mathtt{\$i}}\\
&\text{cost}(t) = \text{cost}_t(t),\ \text{for } t \in \mathcal{G}_{\mathtt{sym}}\\
&\text{cost}(\alpha) = \text{cost}_{\alpha}
\end{aligned} \tag{9}$$

where cost_{λ}, $\text{cost}_{\mathsf{app}}$, $\text{cost}_{\mathtt{\$i}}$, and cost_{α} are non-negative constants, and $\text{cost}_t(t)$ is a mapping from grammar primitives to their (non-negative) costs. Finally, we introduce $\text{cost}^*(\cdot)$ as a version of $\text{cost}(\cdot)$ where $\text{cost}_{\alpha} = 0$.

We can equivalently construct the utility in Eq. (8) by summing over the compression gained from performing each rewrite separately, as given below.

$$U_{\mathcal{P},\mathcal{R}}(A) = -\text{cost}(A) + \sum_{e \in \textsc{RewriteLocations}_{\mathcal{R}}(\mathcal{P},A)} \text{cost}(e) - \text{cost}(\textsc{RewriteOne}(e, A)) \tag{10}$$

By reasoning about the way that rewriting transforms a program, this utility can be broken down even further:

$$U_{\mathcal{P},\mathcal{R}}(A) = -\text{cost}(A) + \sum_{e \in \textsc{RewriteLocations}_{\mathcal{R}}(\mathcal{P},A)} U_{local}(A,e) \tag{11}$$

$$U_{local}(A,e) = \underbrace{\text{cost}^*(A)}_{\text{abstraction size}} \underbrace{- (\text{cost}_t(t_A) + \text{cost}_{\mathsf{app}} \cdot \text{arity}(A))}_{\text{application utility (negative)}} + \underbrace{\sum_{[\alpha \to e'] \in \text{args}(A,e)} (\text{usages}(\alpha) - 1) \cdot \text{cost}(e')}_{\text{multiuse utility}} \tag{12}$$

where t_A is the new primitive corresponding to abstraction A. This form of the utility function can be efficiently computed without explicitly performing any rewrites; and it sheds light on the different sources of compression. The three main terms in this expression are (1) the *abstraction size* that comes from the shared structure that is removed, (2) the negative *application utility* that comes from introducing the new primitive and lambda calculus `app` nonterminals to apply it to each argument, and (3) the *multiuse utility* which comes from only needing to pass in a single copy of an argument that might be use in more than one place in the body. We emphasize that this form of the utility function is equivalent to the original definition based on rewriting given in Eq. (8).

4.2 Upper bounding the utility

We seek an upper bound function $\bar{U}_{\mathcal{P},\mathcal{R}}(A_{??})$ such that for any A derived from $A_{??}$, $\bar{U}_{\mathcal{P},\mathcal{R}}(A_{??}) \geq U_{\mathcal{P},\mathcal{R}}(A)$. We begin from the decomposition of the utility function given in Section 4.1. Since costs are always non-negative, we can upper bound this by dropping the $-\text{cost}_\alpha(A)$ term as well as the negative term within the sum:

$$U_{\mathcal{P},\mathcal{R}}(A) \leq \sum_{e \in \textsc{RewriteLocations}_{\mathcal{R}}(\mathcal{P},A)} \text{cost}(e) \tag{13}$$

Intuitively, dropping the negative term from the sum is equivalent to assuming we compressed the cost of this location all the way down to cost 0. We can also bound $\textsc{RewriteLocations}_{\mathcal{R}}(\mathcal{P}, A)$ as $\textsc{Matches}(\mathcal{P}, A_{??})$ using Lemma 3.2, yielding our final upper bound in terms of $A_{??}$:

$$\bar{U}_{\mathcal{P},\mathcal{R}}(A_{??}) \triangleq \sum_{e \in \textsc{Matches}(\mathcal{P},A_{??})} \text{cost}(e) \tag{14}$$

4.3 Subsumption pruning

We identify two additional forms of subsumption pruning compatible with a compressive utility. The first is *argument capture*: when a partial abstraction takes the same argument for at least one abstraction variable across all match locations, this abstraction can be discarded. This is because every abstraction derivable is covered by another abstraction which is identical except it has the argument inlined into the body. For example, if an abstraction variable α takes the same argument `(+ 3 5)` across every match location, there is a superior partial abstraction that simply has `(+ 3 5)` inlined in its body instead of using an abstraction variable.

The second is *redundant argument elimination*: a partial abstraction can be dropped if it has two abstraction variables that always take the same argument as each other across all match locations, for example if it had variables α and β and $\alpha = \beta =$ `(+ 3 5)` at one location, $\alpha = \beta =$ `2` at another location, and so on for all match locations. This abstraction would then be subsumed

by the abstraction that doesn't take β as an argument and instead reuses α in place of β, and can hence be eliminated.

4.4 Rewrite strategy

Stitch employs a linear-time, optimal rewrite strategy for a compression objective. The goal of an optimal rewrite strategy is to efficiently select the optimal subset of $\textsc{Matches}(\mathcal{P}, A)$ to perform rewrites at in order to maximize the utility.

The main challenge is that when match locations overlap, the strategy must decide which of the two to accept. For example, consider the program `(foo (foo (foo bar))` and the abstraction $\lambda\alpha$. `(foo (foo` α`))`. This abstraction matches at the root of the program with α = `(foo bar)`, resulting in the rewritten program (`(`t_A `(foo bar))`. However, though the abstraction also matched at the subtree `(foo (foo bar)` in the original program, this match location is no longer present in the rewritten program, so only one of the two locations can be chosen by the rewrite strategy.

Our approach is a bottom-up dynamic programming algorithm, which begins at the leaves of the program and proceeds upwards. At each node e, we compute the cumulative utility so far if we reject a rewrite here (util_R), if we accept a rewrite here (util_A), or if we choose the better of the two options (util^*):

$$
\begin{aligned}
\text{util}_R[e] &= \sum_{e' \in children(e)} \text{util}^*[e'] \\
\text{util}_A[e] &= \begin{cases} 0 & \text{if } e \notin \textsc{Matches}(\mathcal{P}, A) \\ U_{local}(A, e) + \sum_{e' \in \text{args}(A,e)} \text{util}^*[e'] & \text{otherwise} \end{cases} \\
\text{util}^*[e] &= \max(\text{util}_R[e], \text{util}_A[e])
\end{aligned}
\tag{15}
$$

where $U_{local}(A, e)$ is the utility gained from a single rewrite as defined in Eq. (12). After calculating these quantities the rewrite strategy can start from the program root and recurse down the tree, rewriting at each node where rewriting is optimal (i.e. $\text{util}_A > \text{util}_R$) and then recursing into its arguments after rewriting. Since all arguments were originally subtrees in the program these quantities have been calculated for all of them, with the caveat that their de Bruijn indices may have been shifted. Shifting indices of a subtree does not change whether an abstraction can match there nor does it change the utility gained from using that abstraction, so this simply requires some extra bookkeeping to track.

This algorithm is optimal by a simple inductive argument - at each step of the dynamic programming problem we can assume that we know the cumulative utility of all children and potential arguments at this node, so we can use Eq. (15) to calculate the cumulative utility of either rejecting or accepting the rewrite at this node.

5 COMBINING CORPUS-GUIDED TOP-DOWN SEARCH WITH DEDUCTIVE APPROACHES

In complex domains, assembling good libraries may involve more than just finding matching code-templates; sometimes, some initial refactoring is necessary to expose the common structure. For example, consider learning the abstraction $\lambda\alpha$. `(* 2` α`)` for doubling integers, given the expressions `(* 2 8)`, `(* 7 2)`, and `(right-shift 3 1)`. This is only possible if the system can use the commutativity of multiplication to rewrite `(* 7 2)` into `(* 2 7)`, and bitvector properties to rewrite `(right-shift 3 1)` into `(* 2 3)`; such rewrites are not natively supported by CTS.

$$\textsc{RU-AbsVar}\ \frac{}{\textsc{RewriteUnify}(\alpha, v) \rightsquigarrow [\alpha \to v]} \qquad \textsc{RU-Hole}\ \frac{}{\textsc{RewriteUnify}(??_i, v) \rightsquigarrow [??_i \to v]}$$

$$\textsc{RU-App}\ \frac{\textsc{RewriteUnify}(A_1, v_1) \rightsquigarrow l_1 \qquad \textsc{RewriteUnify}(A_2, v_2) \rightsquigarrow l_2 \qquad l = \textsc{vmerge}(l_1, l_2)}{\textsc{RewriteUnify}((A_1\ A_2), (v_1\ v_2)) \rightsquigarrow l}$$

$$\textsc{RU-Lam}\ \frac{\textsc{RewriteUnify}(A, e) \rightsquigarrow l' \qquad l = \textsc{DownshiftAll}(l')}{\textsc{RewriteUnify}((\lambda.\ A), (\lambda.\ e)) \rightsquigarrow l} \qquad \textsc{RU-Same}\ \frac{}{\textsc{RewriteUnify}(e, e) \rightsquigarrow [\,]}$$

$$\textsc{RU-Union}\ \frac{\textsc{RewriteUnify}(e, v_1) \rightsquigarrow l}{\textsc{RewriteUnify}(e, v_1 \uplus v_2) \rightsquigarrow l}$$

$$\textsc{VM-Same}\ \frac{[\alpha \to v_1] \in l_1 \qquad [\alpha \to v_2] \in l_2}{[\alpha \to (v_1 \cap v_2)] \in \textsc{vmerge}(l_1, l_2)} \qquad \textsc{VM-Diff}\ \frac{[\alpha \to v] \in l_1 \qquad \alpha \notin \text{dom}(l_2)}{[\alpha \to v] \in \textsc{vmerge}(l_1, l_2)}$$

Fig. 5. Defining RewriteUnify, which takes as input an abstraction and a version space of possible refactorings, and yields multiple substitutions corresponding to all the ways that the abstraction's holes and variables can match with programs encoded by the version space.

In this section, we discuss how CTS can be combined with refactoring systems based on deductive rewrites (e.g, e-graphs [Willsey et al. 2021]) to increase its expressivity further. The core idea is simple: run a rewrite system on the corpus to produce a set of refactorings of the corpus in an e-graph or version space, then run CTS over the resulting data structure.[2] Intuitively, this will lead to improved performance compared with a purely deductive approach, since the cost of rewriting grows exponentially the more rewrites you make. This is a problem for fully deductive approaches like Ellis et al. 2021 because extracting the abstractions often requires a long chain of rewrites, especially for higher-arity abstractions. However, a small number of rewrites is typically sufficient to expose the underlying commonalities, as it was in the example above; performing only a handful of rewrites and then using CTS to actually extract the library thus avoids the exponential blow-up of computing several rewrites in sequence, while still benefiting from the increase in expressivity afforded by the rewrites.

Version Spaces. To illustrate this hybrid approach, we combine CTS with *version spaces* [Lau et al. 2003; Polozov and Gulwani 2015] representing sets of programs semantically equivalent under the β-reduction rewrite. Version spaces are represented as terms from a grammar obtained by extending the grammar of expressions with the union ($\uplus$) operator that represents a set of equivalent expressions: $v ::= \varnothing \mid v' \uplus v'' \mid \lambda.\ v' \mid v'\ v'' \mid \$i \mid t$. We define a denotation operator, $[\![v]\!]$, mapping a version space v to a set of terms: for the union operator, $[\![v \uplus v']\!] = [\![v]\!] \cup [\![v']\!]$; for applications, $[\![v\ v']\!] = \{e\ e' : \forall e, e' \in [\![v]\!] \times [\![v']\!]\}$; for lambda abstractions, $[\![\lambda v]\!] = \{\lambda e : \forall e \in [\![v]\!]\}$; for the empty set, $[\![\varnothing]\!] = \varnothing$; and for deBuijn indices and terminals, $[\![v]\!] = v$. Ellis et al. [2021] give a procedure called $I\beta(e)$, which take as input an expression e and outputs a version space v inverting one step of β-reduction: that is, $e' \to^{\beta} e$ iff $e' \in [\![I\beta(e)]\!]$.

To run CTS on top of this rewriting system requires a generalization of LambdaUnify: instead of unifying an abstraction with a term, yielding a single substitution, we unify against a *set of terms* (a version space), yielding a set of candidate substitutions. The relation RewriteUnify (Fig. 5) accomplishes this, and allows us to straightforwardly redefine the core Stitch quantities in terms of RewriteUnify. The size of term e after expanding it into $I\beta(e)$ and rewriting it with abstraction

[2]Note that while we have previously presented CTS as operating over syntax trees, the core notions of upper bounds and *matching* that CTS operates on are not restricted to program trees.

A is:

$$\text{cost}(\textsc{Rewrite}(A,e)) = \min_{\substack{l: \\ \textsc{RewriteUnify}(A,I\beta(e))\rightsquigarrow l}} \text{cost}_t(t_A) + \text{cost}_{\text{app}} \cdot \text{arity}(A)) + \sum_{v \in \text{range}(l)} \min_{e' \in [\![v]\!]} \text{cost}(v)$$

And the local utility is then given by:

$$U_{\text{local}}(A,e) = \text{cost}(e) - \text{cost}(\textsc{Rewrite}(A,e)) \tag{16}$$

Example: Learning map from fold. Consider learning the higher-order function `map` from two example programs: doubling a list of numbers (via `(fold (λx.λℓ. (cons (+ x x) ℓ)) nil)`) and decrementing a list of numbers (via `(fold (λx.λℓ. (cons (- x 1) ℓ)) nil)`). Matching these programs with the `map` function—`(fold (λx.λℓ. (cons (α x) ℓ)) nil)`—requires re-expressing them as `(fold (λx.λℓ. (cons ((λz. (+ z z)) x) ℓ)) nil)` and `(fold (λx.λℓ. (cons ((λz. (- z 1)) x) ℓ)) nil)`, with the substitutions $\alpha \mapsto$ `(λz. (+ z z))` and $\alpha \mapsto$ `(λz. (- z 1))` respectively. These two re-expressions are licensed by a β-reduction rewriting step.

6 EXPERIMENTS

In this section, we evaluate corpus-guided top-down search for library learning. Specifically, our evaluation focuses on five hypothesis about the performance of Stitch:

(1) *Stitch learns libraries of comparable quality to those found by existing deductive library learning algorithms in prior work, while requiring significantly less resources.* In Section 6.1 we run Stitch on the library learning tasks from [Ellis et al. 2021] and directly compare Stitch to DreamCoder, the deductive algorithm introduced in that work. We find that Stitch learns libraries which usually match or exceed the baseline in quality (measured via a compression metric), while improving the resource efficiency in terms of memory usage and runtime by 2 and 3-4 orders of magnitude compared to the baseline (respectively).

(2) *Stitch scales to corpora of programs that contain more and longer programs than would be tractable with prior work.* In Section 6.2, we evaluate Stitch's ability to learn libraries within eight graphics domains from Wong et al. 2022, which are considerably larger and more complex than have been considered in previous work. We find that stitch on average obtains a test set compression ratio of 2.55x-11.57x in 0.28s-134.44s, with a peak memory usage of 11.22MB-715.42MB in these domains. The problems are large enough to time out with the DreamCoder baseline.

(3) *Stitch degrades gracefully when resource-constrained.* In Section 6.3 we investigate Stitch's performance when run as an *any-time* algorithm, i.e. one that can be terminated early for a best-effort result if a corpus is too large or there are limits on time or memory. We reuse the eight graphics domains from Wong et al. 2022 and find that with its heuristic guidance Stitch converges upon a set of high quality abstractions very early in search, doing so within 1% of the total search time in 3 out of 8 domains and 10% in all except one.

(4) *All the elements of Stitch matter.* In Section 6.4, we carry out an ablation study on Stitch and find the *argument capturing* and *upper bound* pruning methods essential to its performance, while *redundant argument elimination* also proves useful in certain domains. With all optimizations disabled, we find that Stitch cannot run in $\leq$ 90 minutes and $\leq$ 50GB of RAM on any of the domains from Wong et al. 2022.

(5) *Stitch is complementary to deductive rewrite-based approaches to library learning.* These prior experiments show the superior runtime performance of Stitch relative to deductive rewrite systems, but deductive systems have an important advantage over Stitch, the ability to incorporate arbitrary rewrite rules to expose more commonality among different programs and in that way discover better libraries. Such deductive approaches are especially more apt

at learning higher-order abstractions. In Section 6.5, we show that this expressivity gap can be reduced by running Stitch on top of such rewrite systems, allowing it to learn almost all the same functions they do while still using only <1% of the compute time.

We run all experiments on a machine with two AMD EPYC 7302 processors, 64 CPUs, and 256GB of RAM. We parametrize Stitch's cost(e) function (as defined in Section 4.1) as follows: $\text{cost}_{\text{app}} = \text{cost}_{\lambda} = 1$, $\text{cost}_{\$\text{i}} = \text{cost}_{\alpha} = \text{cost}_{t}(t) = 100$.

6.1 Iterative bootstrapped library learning

Experimental setup. Our first experiment is designed to replicate the experiments in DreamCoder, which is the state-of-the-art in deductive library learning. DreamCoder learns libraries iterativel: the system is initialized with a low-level DSL, and then alternates between synthesizing programs (via a neurally-guided enumerative search) that solve a training corpus of inductive tasks and updating the library of abstractions available to the synthesizer. Traces from the experiments carried out by Ellis et al. 2021 are publicly available[3] and include all of the intermediate programs that were synthesized as well as the libraries learned from those programs.

In this experiment, we take these traces and evaluate Stitch on each instance where library learning was performed, comparing the quality of the resulting library to the original one found by DreamCoder. We also re-run DreamCoder on these same benchmarks in order to evaluate its resource usage, capturing its runtime and memory usage in the same environment as Stitch.

The library learning algorithm in Ellis et al. 2021 implements a stopping criterion to determine how *many* abstractions to retain on any given set of training programs. In our comparative experiments, we run the DreamCoder baseline first, and then match the number of abstractions learned by Stitch at each iteration to those learned by the baseline under its stopping criterion so that timing comparisons are fair. We record the total time spent both performing abstraction learning and rewriting for both Stitch and DreamCoder.

We replicate experiments on five distinct domains from Ellis et al. 2021:

- **Lists**: a functional programming domain consisting of 108 total inductive tasks;
- **Text**: a string editing domain in the style of FlashFill [Gulwani et al. 2015] consisting of 128 total inductive tasks;
- **LOGO**: a graphics domain consisting of 80 total inductive tasks;
- **Towers**: a block-tower construction domain consisting of 56 total inductive tasks;
- **Physics**: a domain for learning equations corresponding to physical laws from observations of simulated data, consisting of 60 total inductive tasks

Assessing library quality with a compression metric. The standard Stitch configuration optimizes a compression metric that minimizes the size of the programs after being rewritten to use the abstraction. This is a standard metric in program synthesis, since shorter programs are frequently easier to synthesize. Optimizing against this metric is equivalent to maximizing the likelihood of the rewritten programs under a uniform PCFG.

The DreamCoder synthesizer is more sophisticated than simple enumeration; it takes as input a learned typed bigram PCFG and leverages it to synthesize programs more efficiently. When performing compression, it optimizes against this given PCFG in order to find abstractions that will be more profitable for its specific synthesizer. For the purpose of this evaluation, however, we restrict ourselves to the uniform PCFG because the one used by DreamCoder requires programs to be in a particular normal form. Another aspect of DreamCoder relevant to its compression metric is that DreamCoder synthesizes multiple programs that solve the same task and then selects the

[3] https://github.com/mlb2251/compression_benchmark

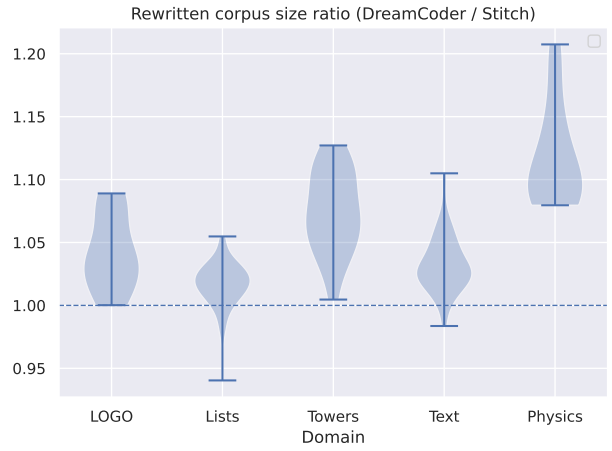

Fig. 6. Compression rates obtained when running Stitch on the five domains considered in 6.1 relative to those of DreamCoder. Higher is better for Stitch: a ratio above 1.0 indicates that Stitch achieves greater compression than DreamCoder. The specific compression metric used in the ratio is given in Eq. (17).

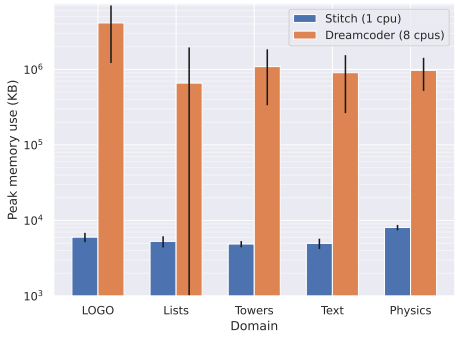

Fig. 7. Peak memory usage of Stitch and DreamCoder while running on the five domains considered in 6.1, averaged over all benchmarks. Lower is better; black lines indicate ± one standard deviation. Note the logarithmic y-axis.

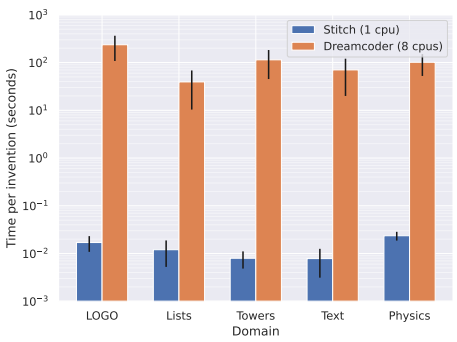

Fig. 8. Wall-clock time required to find (and rewrite under) one abstraction in each of the five domains from 6.1, averaged over all benchmarks. Lower is better; black lines indicate ± one standard deviation. Note the logarithmic y-axis.

abstraction that works best on *some* program for a given task. This is expressed formally in the equation below.

$$\text{cost}(A) + \sum_{\text{task}} \min_{p \in \text{task}} \text{cost}\left(\textsc{Rewrite}(p, A)\right) \tag{17}$$

It is trivial to implement this best-of-task metric in Stitch, so we use this for the comparisons with DreamCoder.

Results. We compare DreamCoder and Stitch for library quality and resource efficiency.

Library quality. We first examine the quality of the libraries learned by both Stitch and DreamCoder using the compression metric in Eq. (17): Fig. 6 shows the ratio between them across

all of the benchmarks for each domain. A ratio of 1.0 indicates that programs are the *exact same length* under the libraries learned by DreamCoder and Stitch, a ratio greater than 1.0 indicates that Stitch learns more compressive libraries, and a ratio less than 1.0 indicates that Stitch learns libraries which are less compressive. For example, a ratio of 1.1 indicates that DreamCoder rewrote to produce a corpus 10% larger than that of Stitch, so it achieved less compression.

These results show that Stitch generally learns libraries of comparable and often greater quality than DreamCoder when matching the number of abstractions learned by the latter. In the *logo*, *towers*, and *physics* domains, Stitch always finds abstractions that are of equal—and often considerably greater—compressive quality than DreamCoder does; in the *list* and *text* domains, Stitch more often than not still obtains better compression, but sometimes loses out to DreamCoder. This is a result of the fact that Stitch cannot learn higher-order abstractions, which are useful in these domains; although we emphasize our focus is on scalability, we will later present an extension of Stitch capable of handling most higher-order functions in Section 6.5, based on the formalism developed in Section 5. Nonetheless, we conclude that **Stitch learns libraries whose quality is comparable to and often better than those found by DreamCoder.**

Resource efficiency. In addition to the quality of the libraries found, we are interested in how the two methods compare in terms of time and space requirements. Since we a priori believe Stitch to be significantly faster, for this evaluation we allow DreamCoder to use 8 CPUs but limit Stitch to single threading[4]. The results are shown in Fig. 7 and Fig. 8; in summary, we find that Stitch takes *tens of milliseconds* to discover abstractions across all five domains—achieving a *3-4 order of magnitude* speed-up over DreamCoder—while also requiring more than *2 orders of magnitude* less memory. We thus conclude that **Stitch is dramatically more efficient than the state-of-the-art deductive baseline** when replicating the iterative library learning experiments of Ellis et al. 2021.

6.2 Large-scale corpus library learning

Experimental setup. While the previous experiment allowed us to benchmark Stitch directly against a state-of-the-art deductive baseline, the iterated learning setting considered by Ellis et al. 2021 only evaluates library learning on relatively small corpora of short programs discovered by the synthesizer. Our second experiment is instead designed to evaluate Stitch in a more traditional learning setting, in which we aim to learn libraries of abstractions from a large corpus of existing programs all at once.

We source our larger-scale program datasets from Wong et al. 2022, which present a series of datasets designed as a benchmark for comparing human-level abstraction learning and graphics program writing against automated synthesis and library learning models. These datasets are divided into two distinct high-level domains (*technical drawings* and *block-tower planning tasks*), each consisting of four distinct subdomains containing 250 programs:

- **Technical drawing domains: *nuts and bolts; vehicles; gadgets; furniture***: CAD-like graphics programs that render technical drawings of common objects, written in an initial DSL consisting of looped transformations (scaling, translation, rotation) over simple geometric curves (lines and arcs).
- **Tower construction domains: *bridges; cities; houses; castles***: planning programs that construct complex architectures by placing blocks, written in an initial DSL that moves a virtual hand over a canvas and places horizontal and vertical bricks.

[4]While this may seem unfair to Stitch, it is worth noting that it would be unlikely to benefit from multithreading when running on the order of milliseconds anyway; DreamCoder, on the other hand, would struggle greatly in these domains without the aid of parallelism.

We choose these datasets not only for their size and scale (full dataset statistics in Table 1), but also for the complexity of their potential abstractions: Wong et al. 2022 explicitly design their corpora to contain complex hierarchical structures throughout the programs, making them an interesting setting for library learning.

Domain	#Programs	Average program length	Average program depth
nuts & bolts	250	76.03 ± 24.22	15.18 ± 2.13
gadgets	250	142.85 ± 87.32	20.88 ± 2.37
furniture	250	171.74 ± 48.41	31.83 ± 5.33
vehicles	250	141.70 ± 40.23	21.22 ± 1.35
bridges	250	137.03 ± 59.71	92.35 ± 39.80
cities	250	161.70 ± 55.56	109.80 ± 37.66
castles	250	189.09 ± 60.18	128.27 ± 40.77
houses	250	168.13 ± 55.75	114.85 ± 37.60

Table 1. Summary statistics about the domains from Wong et al. 2022. Program length is the number of terminal symbols in the program; program depth is the length of the longest path from root to leaf in the program tree. Both are reported as the mean over the entire dataset ± one standard deviation.

When performing library learning in a synthesis setting by compressing a corpus of solutions, it's desirable to find abstractions that would be useful for solving *new* tasks, as opposed to abstractions that overfit to the existing solutions. To evaluate how well the abstractions we learn apply to heldout programs in the domain, we split the corpora into train and test sets, running Stitch on the train set and evaluating its compression on the test set. Since Wong et al. 2022 do not present a train/test split of their datasets, we use stochastic cross validation to evaluate the generalization of the libraries found by Stitch. For each domain, we randomly sample 80% of the dataset to train on and reserve the last 20% as a held out test set; we repeat this procedure 50 times. We then ask Stitch to learn a library consisting of ≤ 10 abstractions with a maximum arity of 3, and average the results across the different random seeds.

To obtain a baseline to compare its performance against, we once again turned to DreamCoder [Ellis et al. 2021]. However, we found that DreamCoder was unable to discover even a single abstraction when run directly on any of the datasets from Wong et al. 2022, despite being given hours of runtime and 256GB of RAM. We also experimented with heavily sub-sampling the training dataset before passing it to DreamCoder, but failed to find a configuration under which DreamCoder finds any interesting abstractions at all due to the fact that it immediately blows up on programs as long as these. As a result, we resort to presenting Stitch's performance metrics without any baseline to compare against; we stress that this is a direct result of the fact that Stitch is the first library learning tool that scales to such a challenging setting.

Results. The results are summarized in Table 2. We find that Stitch scales up to even the most complex sub-domains, running in ≈ 2 minutes with a peak memory usage way below 1GB on *castles*. On four out of eight of the domains, Stitch finishes in single-digit seconds and consumes only tens of megabytes. This stands in stark contrast to DreamCoder, which we were unable to run

Domain	Compression Ratio		Runtime (s)	Peak mem. usage (MB)
	Training set	Test set		
nuts & bolts	12.00 ± 0.25	11.57 ± 0.49	0.28 ± 0.07	11.22 ± 0.25
gadgets	4.03 ± 0.15	3.91 ± 0.31	2.92 ± 0.44	30.48 ± 0.90
furniture	4.95 ± 0.07	4.85 ± 0.26	5.52 ± 0.57	31.59 ± 0.60
vehicles	4.28 ± 0.14	4.14 ± 0.25	2.32 ± 0.25	27.31 ± 0.84
bridges	4.36 ± 0.06	3.78 ± 0.14	28.50 ± 1.38	199.46 ± 10.57
cities	3.15 ± 0.05	3.06 ± 0.14	97.24 ± 3.50	431.99 ± 9.90
castles	2.57 ± 0.07	2.55 ± 0.08	134.44 ± 7.76	715.42 ± 33.89
houses	8.92 ± 0.21	8.85 ± 0.57	27.53 ± 1.72	243.62 ± 2.89

Table 2. Results for the large-scale library learning experiment in Sec. 6.2. The compression ratio refers to how many times smaller the corpus is after rewriting under the learned library compared to the original corpus; higher is thus better. All results are given as the mean ± one standard deviation over 50 runs with different random seeds for the dataset splitting.

on the very simple nuts & bolts domain even with 256GB RAM and several hours worth of compute budget. These results thus support our claim that **Stitch scales to corpora of programs that would be intractable with prior library learning approaches**.

We hope that by providing our results on these datasets in full, future work in this field will benefit from having a directly comparable baseline.

6.3 Robustness to early search termination

Experimental setup. This experiment is designed to evaluate how early into the search procedure Stitch finds what will eventually prove to be the optimal set of abstractions. This is highly relevant in settings where the set of training programs is too large to run the search to completion. The experiment showcases one of Stitch's more subtle strengths: corpus-guided top-down abstraction search is an *anytime* algorithm, and thus does not need to be run to completion to give useful results.

We re-use the domains from Wong et al. 2022 and once again evaluate the quality of the library learned (measured in program compression), similarly to what was done in the previous experiment. However, since we are interested in how quickly Stitch finds *what it perceives to be* the optimal abstractions, we measure compressivity of the training dataset itself (rather than a held-out test set) and capture the compression ratio obtained by each *candidate* abstraction found *during* search (rather than just the compression ratio obtained when search has been run to completion). Thus, we are able to investigate how early on during the search procedure Stitch converges on a chosen library. We restrict Stitch to learning a single abstraction with a maximum arity of 3.

Results. The results are shown in Fig. 9. These results validate our hypothesis that **Stitch is empirically robust to terminating the search procedure early**: in every sub-domain except for *nuts & bolts* Stitch converges to the optimal library very early on, having only completed a tiny

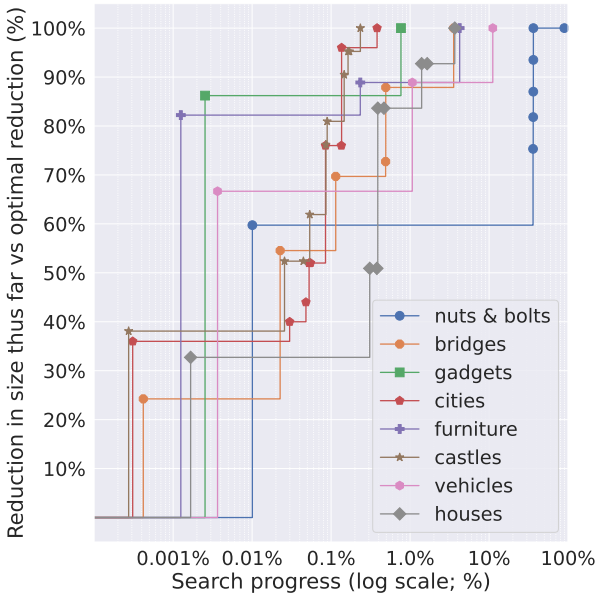

Fig. 9. Reduction in size obtained on the training set when rewritten under the best abstraction found thus far vs. the number of nodes expanded during search. The y-axis is normalized with respect to the optimal abstraction; the x-axis is normalized with respect to the total number of nodes explored by Stitch. Lines thus end earlier the quicker Stitch finds the optimal abstraction; however in all runs Stitch *continues to search* until reaching 100% on the x-axis, exploring the rest of the abstraction space without finding a new best abstraction. Note the logarithmic x-axis.

fraction of the total search.[5] We believe that this has great importance for Stitch's applicability in data-rich settings since it suggests that a nearly-optimal library can be found even if the search must be terminated early (e.g. after a fixed amount of time has passed), making early stopping an empirically useful way of speeding up the library learning process.

6.4 Ablation study

Experimental setup. Stitch implements several different optimizations, which we have argued hasten the search for abstractions. To verify that this claim holds in practice, we now carry out a brief ablation study.

Since the space of every possible combination of optimizations is too large to present succinctly, we focus our attention on four ablations:

- **no-arg-capture**, which disables the pruning of abstractions which are only ever used with the exact same set of arguments (and these arguments could therefore just be in-lined for greater compression)
- **no-upper-bound**, which disables the upper-bound based pruning
- **no-redundant-args**, which disables the pruning of multi-argument abstractions that have a redundant argument that could be removed because it is always the same as another argument
- **no-opts**, which disables all of Stitch's optimizations

[5] Given that significant attention has already been given to wall-clock run-times of Stitch on similar workloads in 6.1, we here use the number of nodes explored instead of wall-clock time to ensure deterministic and easily reproducible results.

System \ Domain	bridges	castles	cities	gadgets	furniture	houses	nuts & bolts	vehicles
no-arg-capture	⏱	⏱	⏱	1375.05	⏱	⏱	33.74	7.18
no-upper-bound	12.04	23.27	27.96	241.82	63.43	35.44	159.33	302.25
no-redundant-args	1.92	1.45	1.39	1.01	1.00	1.01	1.04	1.00
no-opt	RAM	⏱	⏱	RAM	⏱	⏱	RAM	⏱

Table 3. Results from the ablation study. Each cell contains the ratio between the number of nodes explored during search by that particular system and the number of nodes explored by the baseline on the same domain. Lower is better; 1.00 means performance is identical to the baseline. Cells labeled RAM crashed due to reaching the 50GB virtual address space limit, while those labeled ⏱ reached the 90 minute time restriction.

To isolate the impact of disabling optimizations, we run a single iteration of abstraction on each of the 8 domains from 6.2 and collect the number of nodes explored during the search. Running a single iteration reduces the impact of compound effects on the results, and focusing on the number of nodes explored (rather than for example latency) allows for deterministic results. We also fix the maximum arity of abstractions to 3 in all runs, aiming to strike a good balance between compute requirements and how much the optimization will be exposed. We limit each run to 50GB of virtual address space, as well as 90 minutes of compute.

Results. The results are shown in Table 3. We first note as a sanity check that each ablation does indeed lead to reduced performance in general (i.e. explores a search space larger than the baseline does). Furthermore, the results suggest that upper-bound based pruning is the most important in the *nuts & bolts* and *vehicles* domains, while pruning out argument capture abstractions is the most important in the other six domains; it is noteworthy that this latter ablation by itself causes Stitch to reach the memory limit or timeout on more than half of the domains, which suggests that function applications in these domains contain very little variation in the arguments supplied. On the other hand, disabling the pruning of redundant arguments has a relatively modest impact on the size of the search space, but still leads to an almost 2x improvement in the bridge domain.

Perhaps the most important takeaway from this ablation study is that when *all* optimizations are disabled, Stitch fails to find an abstraction within the resource budget on any of the domains. This verifies our hypothesis that corpus-guided pruning of the search space is the key factor involved in making top-down synthesis of abstractions tractable.

6.5 Learning Libraries of Higher-Order Functions

Experimental setup. Our experiments up till now have focused on performant and scalable library learning. This comes at the expense of some expressivity: deductive rewrite systems can, in principle, express broader spaces of refactorings. For example, a rewrite based on β-reduction allows inventing auxiliary λ abstractions, which helps learn higher-order functions: in Ellis et al. 2021, DreamCoder is shown to recover higher-order functions such as `map`, `fold`, `unfold`, `filter`, and `zip_with`, starting from the Y-combinator. This works by constructing a version space which encodes every refactoring which is equivalent up to β-reduction rewrites. But DreamCoder's coverage comes at steep cost: inverting β-reduction is expensive.

In this experiment, we seek to give evidence that it is possible to make Stitch recover almost all of these higher-order functions by layering it on top of version space obtained after a single step of DreamCoder's β-reduction-inverting refactoring, following the formalism outlined in Section 5[6]. We then compare this modified version of Stitch against DreamCoder on its ability to learn these higher-order functions from programs generated by intermediate DreamCoder iterations, and measure the runtime of each approach.

Probabilistic Re-ranking. While the approach outlined in Section 5 should suffice to layer Stitch on top of general version space-based deductive rewrite systems, some extra care needs to be taken to combine it with DreamCoder. This is because DreamCoder implements a probabilistic Bayesian objective for judging candidate abstractions (exploiting the connection between compression and probability [Shannon 1948]), seeking the library L which maximizes $P(L)\prod_{p\in\mathcal{P}} P(p|L)$ for a given or learned prior prior $P(L)$ and likelihood $P(p|L)$. Stitch, on the other hand, effectively judges compression quality by counting the syntactic size of the programs, as has been discussed at great length in Section 4.

To heuristically hybridize these methods, we simply run Stitch on the version spaces as-is but then re-score each complete abstraction popped off the queue under the Bayesian objective, using DreamCoder's models of the prior and likelihood. To make this integration easier, we re-implement Stitch in Python, giving a prototype version called pyStitch which is only used for this experiment.

System		Fold	Unfold	Map	Filter	ZipWith	Time (s)
pyStitch	base	×	×	×	$\frac{\checkmark}{\times}$	✓	254
	+Bayes	×	×	×	×	✓	475
	+VS	✓	✓	×	×	✓	1103
	+Bayes+VS	✓	✓	✓	$\frac{\checkmark}{\times}$	✓	1616
DreamCoder	Step 1	×	×	✓	×	✓	67
	Step 2	×	×	✓	✓	×	116
	Step 3	×	×	✓	✓	✓	2254
	Step 4	✓	✓	✓	✓	✓	228048

Table 4. Comparing library learners on functional programming exercises. DreamCoder, n-steps: deductive baseline rewriting n steps of β-reduction. pyStitch: Python reimplementation of Stitch, which enables better interoperability with DreamCoder's version space algebra (+VS) and probabilistic models (+Bayes). pyStitch+VS+Bayes learns essentially all of the same higher-order functions, using $< 1\%$ of the compute. ($\frac{\checkmark}{\times}$: half credit. For filter, pyStitch learns a filter-esque function with type $\alpha \to (\alpha \to \beta \to \text{bool}) \to \text{list}(\beta) \to \text{list}(\beta)$.)

[6]Note that DreamCoder was only used as a convenient base upon which to graft the Stitch; we believe the same hybridization recipe should apply to other rewrite frameworks as well, such as e-graphs [Willsey et al. 2021], but leave it to future work to investigate this in detail.

Results. The results are shown in Table 4. We note first the large discrepancy in runtimes; running DreamCoder with 4 steps of rewriting takes roughly 2.5 days of compute on this domain, while even the slowest version of pyStitch still finishes in less than half an hour. In terms of the number of higher-order abstractions found, DreamCoder only finds all five when run in its most expensive configuration; reducing the computation cost quickly decreases its expressivity. For pyStitch, the Bayesian re-ranking alone (pyStitch+Bayes) does not yield any improvements, while running it on the version spaces (pyStitch+VS) yields 3 out of 5 functions. However, it is only when these two adaptations are used in conjunction (pyStitch+Bayes+VS) that the Stitch-based method is able to find almost all of the higher-order functions, failing only in that it learns an unconventional sibling of the `filter` function.

In summary, we find that running CTS after a single step of version space rewriting and then probabilistically re-ranking the results suffices to recover almost all of the core higher-order functions that DreamCoder learns, while using <1% of its compute. Since our methodology may easily be extended to other version space-based deductive systems, and even e-graph based ones with some modifications to the underlying formalism, we thus conclude that **running Stitch on top of a deductive rewrite system closes the expressivity gap, while retaining superior performance**.

7 RELATED WORK

Stitch is related to two core ideas from prior work: **deductive refactoring and library learning systems**, which introduce the idea of learning abstractions that capture common structure across a set of programs, but have largely been driven by deductive algorithms; and **guided top-down program synthesis systems**, which use cost functions to guide top-down enumerative search over a space of programs, but have largely been used to synthesize whole programs for individual tasks in prior work.

7.1 Deductive refactoring and library learning.

Recent work shares Stitch's goal of learning *libraries* of program abstractions which capture reusable structure across a corpus of programs [Allamanis and Sutton 2014; Cropper 2019; Dechter et al. 2013; Ellis et al. 2021, 2018; Iyer et al. 2019; Jones et al. 2021; Shin et al. 2019; Wong et al. 2021]. Several of these prior approaches also introduce a utility metric based on *program compression* in order to determine the most useful candidate abstractions to retain [Dechter et al. 2013; Ellis et al. 2021; Iyer et al. 2019; Lázaro-Gredilla et al. 2019; Wong et al. 2021].

Prior work generally follows a bottom-up approach to abstraction learning, combining a bottom-up traversal across individual training programs with a second stage to extract shared abstractions from across the training. This approach includes systems that work through direct *memoization* of subtrees across a corpus of programs [Dechter et al. 2013; Lázaro-Gredilla et al. 2019; Lin et al. 2014]; *antiunification* (caching tree templates that can be unified with training program syntax trees) [Ellis et al. 2018; Henderson 2013; Hwang et al. 2011; Iyer et al. 2019]; or by more sophisticated *refactoring* using one or more rewrite rules to expose additional shared structure across training programs [Chlipala et al. 2017; Ellis et al. 2021, 2018; Liang et al. 2010].

These latter library learning algorithms draw more generally on *deductive synthesis* approaches that apply local rewrite rules in a bottom-up fashion to program trees in order to refactor them — historically, to synthesize programs from a declarative specification of desired function [Burstall and Darlington 1977; Manna and Waldinger 1980]. Deductive approaches to library learning, however, confront fundamental memory and search-time scaling challenges as the corpus size and depth of the training programs increases; prior deductive approaches such as [Chlipala et al. 2017; Ellis et al. 2021, 2018] use version spaces [Lau et al. 2003; Mitchell 1977] to mitigate the memory usage during

bottom-up abstraction proposal. Still, deductive approaches are challenging to bound and prune (unlike the top-down approach we take in Stitch), as they generally traverse individual program trees locally and must store possible abstraction candidates in memory before the extraction step.

7.2 Guided top-down program synthesis.

Stitch uses a corpus-guided top-down approach to learning library abstractions that is closely related to recent *guided enumerative synthesis techniques*. This includes methods that leverage type-based constraints on holes [Feser et al. 2015; Polikarpova et al. 2016], over- and under-approximations of the behaviors of holes [Chen et al. 2020; Lee et al. 2016], and probabilistic techniques to heuristically guide the search [Balog et al. 2016; Ellis et al. 2020, 2021; Nye et al. 2021; Shah et al. 2020]. These approaches have have largely been applied in to synthesize entire programs based on input/output examples or another form of specification, in contrast to the abstraction-learning goal in our work [Allamanis et al. 2018; Balog et al. 2016; Chen et al. 2018; Ellis et al. 2018; Ganin et al. 2018; Koukoutos et al. 2017]. Like Stitch however, these approaches sometimes use cost functions (such as the likelihood of a partially enumerated program under a hand-crafted or learned probabilistic generative model over programs) in order to direct search towards more desirable program trees. However, Stitch's cost function leverages a more direct relationship between partially-enumerated candidate functions and the existing training corpus, unlike the cost functions typically applied in inductive synthesis, which must be estimated from input/output examples.

8 CONCLUSION AND FUTURE WORK

We have presented *corpus-guided top-down synthesis* (CTS)—an efficient new algorithm for synthesizing libraries of functional abstractions capturing common functionality within a corpus of programs. CTS directly synthesizes the abstractions, rather than exposing them through a series of rewrites as is done by deductive systems. Key to its performance is the usage of a guiding utility function, which allows CTS to effectively search (and prune out large portions of) the space of possible abstractions.

We implement this algorithm in Stitch, an open-source library learning tool which exposes both Rust and Python bindings. We evaluate Stitch across five experimental settings, demonstrating that it learns comparably compressive libraries with 2 orders of magnitude less memory and 3-4 orders of magnitude less time compared to the state-of-the-art deductive algorithm of Ellis et al. 2021. We also find that Stitch scales to learning libraries of abstractions from much larger datasets of deeper program trees than is possible with prior work, and that the *anytime* property of top-down corpus-guided search — abstractions discovered via top-down search are already compressive early in search and improve as it continues — offers the opportunity for high-quality library learning even in complex domains through early search termination.

For future work, we aim to build upon the hybrid approach introduced in Section 5 by more closely integrating Stitch into a deductive rewrite system like egg [Willsey et al. 2021], further extending the sorts of equivalences it can reason over. Beyond this, we also aim to explore applying CTS to corpora of human-written code, such as Haskell and OCaml codebases. We are also exploring the application of CTS to finding abstractions in dataflow DAGs and more general graph structures like molecules, since the basic ideas of matching and bounding are still applicable. Finally, we are interested in the new ways that abstraction learning can be applied when it is made this fast; for example, by directly integrating abstraction learning into a synthesis search procedure.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*. 472–483.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).

Rod M Burstall and John Darlington. 1977. A transformation system for developing recursive programs. *Journal of the ACM (JACM)* 24, 1 (1977), 44–67.

Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 487–502. https://doi.org/10.1145/3385412.3385988

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Execution-guided neural program synthesis. In *International Conference on Learning Representations*.

Adam Chlipala, Benjamin Delaware, Samuel Duchovni, Jason Gross, Clément Pit-Claudel, Sorawit Suriyakarn, Peng Wang, and Katherine Ye. 2017. The end of history? Using a proof assistant to replace language design with library design. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Andrew Cropper. 2019. Playgol: Learning programs through play. *arXiv preprint arXiv:1904.08993* (2019).

Nicolaas Govert de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, Vol. 75. Elsevier, 381–392.

Eyal Dechter, Jonathan Malmaud, Ryan Prescott Adams, and Joshua B Tenenbaum. 2013. Bootstrap learning via modular concept discovery. In *Proceedings of the International Joint Conference on Artificial Intelligence*. AAAI Press/International Joint Conferences on Artificial Intelligence.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. 2020. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381* (2020).

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. 2021. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 835–850.

Kevin M Ellis, Lucas E Morales, Mathias Sablé-Meyer, Armando Solar Lezama, and Joshua B Tenenbaum. 2018. Library learning for neurally-guided bayesian program induction. (2018).

Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theor. Comput. Sci.* 103, 2 (sep 1992), 235–271. https://doi.org/10.1016/0304-3975(92)90014-7

John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices* 50, 6 (2015), 229–239.

Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, SM Ali Eslami, and Oriol Vinyals. 2018. Synthesizing programs for images using reinforced adversarial learning. In *International Conference on Machine Learning*. PMLR, 1666–1675.

Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H Muggleton, Ute Schmid, and Benjamin Zorn. 2015. Inductive programming meets the real world. *Commun. ACM* 58, 11 (2015), 90–99.

Robert John Henderson. 2013. Cumulative learning in the lambda calculus. (2013).

Irvin Hwang, Andreas Stuhlmüller, and Noah D Goodman. 2011. Inducing probabilistic programs by Bayesian program merging. *arXiv preprint arXiv:1110.5667* (2011).

Srinivasan Iyer, Alvin Cheung, and Luke Zettlemoyer. 2019. Learning programmatic idioms for scalable semantic parsing. *arXiv preprint arXiv:1904.09086* (2019).

R Kenny Jones, David Charatan, Paul Guerrero, Niloy J Mitra, and Daniel Ritchie. 2021. ShapeMOD: macro operation discovery for 3D shape programs. *ACM Transactions on Graphics (TOG)* 40, 4 (2021), 1–16.

Manos Koukoutos, Mukund Raghothaman, Etienne Kneuss, and Viktor Kuncak. 2017. On repair with probabilistic attribute grammars. *arXiv preprint arXiv:1707.04148* (2017).

Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2003. Programming by demonstration using version space algebra. *Machine Learning* 53, 1 (2003), 111–156.

Miguel Lázaro-Gredilla, Dianhuan Lin, J Swaroop Guntupalli, and Dileep George. 2019. Beyond imitation: Zero-shot task transfer on robots by learning concepts as cognitive programs. *Science Robotics* 4, 26 (2019), eaav3150.

Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*, Bernd Fischer and Ina Schaefer (Eds.). ACM, 70–80. https://doi.org/10.1145/2993236.2993244

Percy Liang, Michael I Jordan, and Dan Klein. 2010. Learning programs: A hierarchical Bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 639–646.

Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B Tenenbaum, and Stephen H Muggleton. 2014. Bias reformulation for one-shot function induction. (2014).

Zohar Manna and Richard Waldinger. 1980. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2, 1 (1980), 90–121.

Tom M Mitchell. 1977. Version spaces: A candidate elimination approach to rule learning. In *Proceedings of the 5th international joint conference on Artificial intelligence-Volume 1*. 305–310.

Maxwell Nye, Yewen Pu, Matthew Bowers, Jacob Andreas, Joshua B Tenenbaum, and Armando Solar-Lezama. 2021. Representing Partial Programs with Blended Abstract Semantics. In *International Conference on Learning Representations*.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538.

Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 107–126.

Ameesh Shah, Eric Zhan, Jennifer Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. 2020. Learning Differentiable Programs with Admissible Neural Heuristics. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 4940–4952. https://proceedings.neurips.cc/paper/2020/file/342285bb2a8cadef22f667eeb6a63732-Paper.pdf

Claude Elwood Shannon. 1948. A mathematical theory of communication. *The Bell system technical journal* 27, 3 (1948), 379–423.

Eui Chul Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. 2019. Program synthesis and semantic parsing with learned code idioms. *Advances in Neural Information Processing Systems* 32 (2019).

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (jan 2021), 29 pages. https://doi.org/10.1145/3434304

Catherine Wong, Kevin M Ellis, Joshua Tenenbaum, and Jacob Andreas. 2021. Leveraging language to learn program abstractions and search heuristics. In *International Conference on Machine Learning*. PMLR, 11193–11204.

Catherine Wong, William McCarthy, Gabriel Grand, Jacob Andreas, Joshua B Tenenbaum, Robert Hawkins, and Judy Fan. 2022. Identifying concept libraries from language about object structure. In *CogSci*. To appear.

A CORPUS-GUIDED TOP-DOWN SEARCH: FULL ALGORITHM

In this appendix, we give an expansive description of the full corpus-guided top-down search (CTS) algorithm to ease re-implementation.

The full algorithm is given in Algorithm 1. It takes as input a corpus of programs $\mathcal{P}$, a utility function $U_{\mathcal{P},\mathcal{R}}(A)$, and utility upper bound function $\bar{U}_{\mathcal{P},\mathcal{R}}(A_{??})$. The algorithm searches for and returns the complete abstraction A_{best} that maximizes the utility function. The algorithm maintains a priority queue Q of partial abstractions ordered by their utility upper bound (or alternatively a custom priority function), a best complete abstraction so-far A_{best}, and a corresponding best utility so far U_{best} for that abstraction. The priority queue is initialized to hold the single-hole partial abstraction ??, from which any abstraction can be derived. A_{best} and U_{best} are initialized as the best arity-zero abstraction and corresponding utility (line 2), since arity-zero abstractions can be quickly and completely enumerated (each unique, closed subexpression in the corpus is an arity-zero abstraction).

We then proceed to the core loop of the algorithm from lines 4-29. At each step of this loop we pop the highest-priority partial abstraction $A_{??}$ off of the priority queue and process it. We discard $A_{??}$ if its utility upper bound doesn't exceed our best utility found so far (lines 6-8). We then choose a hole h in $A_{??}$ to expand with the procedure Choose-Hole. Choose-Hole can be a custom function; we find that choosing the most recently introduced hole is effective in practice.

The algorithm then uses the procedure Expansions to iterate over all possible single step expansions of the hole h in $A_{??}$, such as replacing the hole with + or with (app ?? ??). For each expanded abstraction $A'_{??}$, it is easy to compute its set of match locations Matches$(\mathcal{P}, A'_{??})$ since we know this will be a subset of the match locations of $A_{??}$ (and these will be *disjoint* subsets, except when expanding into an abstraction variable α). We can easily inspect the relevant subtree at each match location of the original abstraction to see which expansions are valid and which match locations will be preserved by a given expansion.

When expanding to an abstraction variable α, if α is a new variable not already present in the partial abstraction, the set of match locations is unchanged. If α is an existing abstraction variable then this is a situation where the same variable is being used in more than one place, as in the *square* abstraction $(\lambda\alpha.\ *\ \alpha\ \alpha)$. In this case we restrict the match locations to the subset of locations where within a location all instances of α match against the same subtree. Additionally, if a maximum arity is provided then an expansion that causes the abstraction to exceed this limit is not considered.

When considering each possible expanded abstraction $A'_{??}$, there is room for strong corpus-guidance. First, we don't need to consider any expansions that would result in zero match locations since all abstractions in this branch of search will have zero rewrite locations per Lemma 3.2 (lines 11-13). Furthermore, we use $\bar{U}_{\mathcal{P},\mathcal{R}}(A'_{??})$ to upper bound the utility achievable in this branch of search and discard it if it is less than our best utility so far (lines 14-16). Since each $A'_{??}$ covers a disjoint set of match locations (except in the case of expanding into an abstraction variable), the set of match locations often drops rapidly and can allow for the calculation of a tight upper bound depending on the utility function. As a final step of pruning, if we can identify that $A'_{??}$ is *subsumed* by some other abstraction $A''_{??}$, we may discard $A'_{??}$ (lines 17-19).

For any partial abstraction $A'_{??}$ that has not been pruned, we then check whether it is a complete abstraction (there are no remaining holes) or whether it is still a partial abstraction. Partial abstractions get pushed to the priority queue ranked by their utility upper bound, and complete abstractions are used to update A_{best} and U_{best} if they have a higher utility than any prior complete abstractions.

Once there are no more partial abstractions remaining in the priority queue, the algorithm terminates. Note that since the algorithm maintains a best abstraction so-far, it can also be terminated

early, making it an *any-time algorithm*. We also note that this algorithm is amenable to parallelization as it can be easily parallelized over the while loop on lines 4-29, especially since the algorithm remains sound even when the upper bound U_{best} used pruning is not always up to date.

This algorithm can be iterated to build up a library of abstractions.

Algorithm 1 Corpus-guided top-down abstraction synthesis. Color-coded: Branch and bound, Zero-usage pruning, Subsumption pruning

Input: Corpus of input programs $\mathcal{P}$, utility function $U_{\mathcal{P},\mathcal{R}}(A)$, and utility upper bound function $\bar{U}_{\mathcal{P},\mathcal{R}}(A_{??})$
Output: The maximally compressive abstraction A_{best}

1: $Q \leftarrow$ Priority-Queue $\{ ?? \}$ ▷ New priority queue with the single partial abstraction ??
2: $A_{best} \leftarrow$ Best-Arity-Zero-Abstraction$(\mathcal{P})$ ▷ Initialize best abstraction so far
3: $U_{best} \leftarrow U_{\mathcal{P},\mathcal{R}}(A_{best})$
4: **while** non-empty(Q) **do**
5: $A_{??} \leftarrow$ pop-max(Q) ▷ Next partial abstraction to expand
6: **if** $\bar{U}_{\mathcal{P},\mathcal{R}}(A_{??}) \leq U_{best}$ **then**
7: **continue** ▷ Branch and bound
8: **end if**
9: $h \leftarrow$ Choose-Hole$(A_{??})$ ▷ Choose a hole to expand
10: **for** $(A'_{??}, M') \in$ Expansions$(A_{??}, h, \mathcal{P})$ **do** ▷ abstraction $A'_{??}$ and match locations M'
11: **if** length$(M') == 0$ **then**
12: **continue** ▷ No match locations in corpus
13: **end if**
14: **if** $\bar{U}_{\mathcal{P},\mathcal{R}}(A'_{??}) \leq U_{best}$ **then**
15: **continue** ▷ Branch and bound
16: **end if**
17: **if** subsumed-abstraction$(A'_{??}, \mathcal{P})$ **then**
18: **continue** ▷ Subsumption pruning
19: **end if**
20: **if** has-holes(A_e) **then**
21: $Q \leftarrow Q \cup A'_{??}$ ▷ add partial abstraction to heap
22: **else**
23: **if** $U_{\mathcal{P},\mathcal{R}}(A'_{??}) > U_{best}$ **then**
24: $U_{best} \leftarrow U_{\mathcal{P},\mathcal{R}}(A'_{??})$ ▷ new best complete abstraction
25: $A_{best} \leftarrow A'_{??}$
26: **end if**
27: **end if**
28: **end for**
29: **end while**