**BABBLE: Learning Better Abstractions with E-Graphs and Anti-unification**

DAVID CAO*, UC San Diego, USA  
ROSE KUNKEL*, UC San Diego, USA  
CHANDRAKANA NANDI, Certora, Inc., USA  
MAX WILLSEY, University of Washington, USA  
ZACHARY TATLOCK, University of Washington, USA  
NADIA POLIKARPOVA, UC San Diego, USA

*Library learning* compresses a given corpus of programs by extracting common structure from the corpus into reusable library functions. Prior work on library learning suffers from two limitations that prevent it from scaling to larger, more complex inputs. First, it explores too many candidate library functions that are not useful for compression. Second, it is not robust to syntactic variation in the input.

We propose *library learning modulo theory* (LLMT), a new library learning algorithm that additionally takes as input an equational theory for a given problem domain. LLMT uses e-graphs and equality saturation to compactly represent the space of programs equivalent modulo the theory, and uses a novel *e-graph anti-unification* technique to find common patterns in the corpus more directly and efficiently.

We implemented LLMT in a tool named BABELLE. Our evaluation shows that BABLELLLE achieves better compression orders of magnitude faster than the state of the art. We also provide a qualitative evaluation showing that BABLELLLE learns reusable functions on inputs previously out of reach for library learning.

**CCS Concepts:**  
- **Software and its engineering** → Functional languages; Automatic programming.

**Additional Key Words and Phrases:** library learning, e-graphs, anti-unification

**ACM Reference Format:**

1 INTRODUCTION

Abstraction is the key to managing software complexity. Experienced programmers routinely extract common functionality into libraries of reusable abstractions to express their intent more clearly and concisely. What if this process of extracting useful abstractions from code could be automated? *Library learning* seeks to answer this question with techniques to compress a given corpus of programs by extracting common structure into reusable library functions. Library learning has many potential applications from refactoring and decompilation [Jones et al. 2021; Nandi et al. 2021].

'Equal contribution

---

© 2023 Copyright held by the owner/author(s).  
2475-1421/2023/1-ART14  
https://doi.org/10.1145/3571207

library:  
\[ f_\theta = x_0 \cdot s_1 \rightarrow \]  
// polygon with \( s_1 \) sides scaled by \( x_0 \)

refactored programs:

library:

\[
\begin{align*}
\text{combine} & \quad \text{(f0 6 2)} \\
\text{combine} & \quad \text{(f0 6 1)}
\end{align*}
\]

\[
\begin{align*}
\text{combine} & \quad \text{(f0 6 2.25)} \\
\text{combine} & \quad \text{(f0 6 2)} \\
\text{combine} & \quad \text{(f0 6 0.5)}
\end{align*}
\]

\[
\begin{align*}
\text{combine} & \quad \text{(f0 6 2)} \\
\text{combine} & \quad \text{(f0 6 1)}
\end{align*}
\]
• Useful abstractions must be used in the corpus at least twice. For example, in a corpus of two programs $2 + 1$ and $3 + 1$, there is no need to consider $\lambda x \rightarrow 3 + x$ because it can only be used in one place, and hence would only increase the size of the corpus.

• Abstractions should be “as concrete as possible” for a given corpus. For example, in the same corpus with $2 + 1$ and $3 + 1$, the abstraction $\lambda x \rightarrow x + 1$ is superior to the more general $\lambda x y \rightarrow x + y$, since both apply to the same two terms, but applying the latter is more expensive (it requires two arguments).

In other words, a useful abstraction corresponds to the least general pattern that matches some pair of subterms from the original corpus; such a pattern can be computed via anti-unification (AU) [Plotkin 1970]. For example, anti-unifying $2 + 1$ and $3 + 1$ yields the pattern $X + 1$, and the desired candidate library function $\lambda x \rightarrow x + 1$ can be derived by abstracting over the pattern variable $X$. Similarly, in Fig. 1, the abstraction $\emptyset$ can be derived by anti-unifying, for example, the blue and the brown subterms of the corpus.

Robustness via E-Graphs. To make candidate generation more robust, LLMT additionally takes as input a domain-specific equational theory and uses it to find programs that are semantically equivalent to the original corpus, but share the most syntactic structure. For example, in the domain of arithmetic, we expect the theory to contain the equation $X + Y \equiv Y + X$, which states that addition is commutative. Given the corpus with terms $2 + 1$ and $1 + 3$, this rule can be used to rewrite the second term into $3 + 1$, enabling the discovery of the common abstraction $\lambda x \rightarrow x + 1$.

The main challenge with this approach is to search over the large space of programs that are semantically equivalent to the original corpus. To this end, LLMT relies on the e-graph data structure and the equality saturation technique [Tate et al. 2009; Willsey et al. 2021] to compute and represent the space of semantically equivalent programs. To enable efficient library learning over this space, we propose a new candidate generation algorithm that efficiently computes the set of all anti-unifiers between pairs of sub-terms represented by an e-graph, using dynamic programming.

Finally, selecting the optimal library in this setting reduces to the problem of extracting the smallest term out of an e-graph in the presence of common sub-expressions. This problem is extremely computationally intensive in its general form, and existing approaches have limited scalability [Yang et al. 2021]. Instead we propose targeted subexpression elimination: a novel e-graph extraction algorithm that uses domain-specific knowledge to reduce the search space and readily supports approximation via beam search to trade off accuracy and efficiency.

Babble. We have implemented LLMT in Babble, a tool built on top of the egg e-graph library [Willsey et al. 2021]. The overview of the Babble’s workflow is shown in Fig. 2, with gray boxes representing existing techniques and black boxes representing our contributions.

We evaluated Babble on benchmarks from two sources: compression tasks extracted from DreamCoder runs [Bowers 2022] and 2D CAD programs used to evaluate concept learning in humans [Wong et al. 2022]. Our evaluation shows that Babble outperforms DreamCoder on its

![Fig. 2. Babble architecture overview](image-url)
own benchmarks, achieving better compression in orders of magnitude less time. Adding domain-specific rewrites improves compression even further. We also show that BABBLE scales to the larger 2D CAD corpora, which is beyond reach of DREAMCODER. We also present and discuss a selection of abstractions learned by BABBLE, demonstrating that the LLMT approach can learn functions that match human intuition.

**Contributions.** In summary, this paper makes the following contributions:

- **library learning modulo equational theory:** a library learning algorithm that can incorporate an arbitrary domain-specific equational theory to make learning robust to syntactic variations;
- **e-graph anti-unification:** an algorithm that efficiently generates the set of candidate abstractions using the mechanism of anti-unification extended from terms to e-graphs;
- **targeted common subexpression elimination:** a new approximate algorithm for extracting the best term from an e-graph in the presence of common sub-expressions.

## 2 OVERVIEW

We illustrate LLMT via a running example building on Fig. 1. The input corpus in Fig. 1 is written in a 2D CAD DSL by Wong et al. [2022], which features the following primitives:

- `line` a line segment from the origin (0, 0) to the point (1, 0)
- `combine(F1, F2)` the union of figures `F1` and `F2`
- `xform(F, τ)` applying transformation `τ` to figure `F`
- `repeat(F, 0, τ)` the empty figure
- `repeat(F, n + 1, τ)` `combine(F, xform(repeat(F, n, τ), τ))`, similar to a “fold”

A transformation `τ` is a 4-tuple `[s, θ, tx, ty]` denoting uniformly scaling by a factor of `s`, rotating by `θ` radians, and translating by `(tx, ty)` in the `x` and `y` directions respectively. For example, the green hexagon in Fig. 1 is implemented as:

\[
xform\left(\text{repeat}\left(xform\left(\text{line}, [1, 0, -0.5, 0.5/\tan(\pi/6)]\right), 6\right), [2, 0/6, 0, 0]), [2, 0, 0, 0]\right)
\]

That is, a hexagon side `xform(line, [1, 0, -0.5, 0.5/\tan(\pi/6)])` is repeated six times, each time rotated by another `2\pi/6` radians, and the resulting unit hexagon is scaled by 2.

Taking a closer look at the two blue hexagons in Fig. 1, however, we notice something peculiar. The two occurrences of `xform(repeat . . ., [1, 0, 0, 0])` in Fig. 1 are no-ops: they merely scale a figure `F` by a factor of 1 and neither rotate nor translate it. These redundant transformations would likely not be there had the code been written by hand or decompiled from a lower-level representation (by a tool like Szalinski [Nandi et al. 2020]); and yet, they are crucial if we hope to learn the optimal abstraction `f_0` with a purely syntactic technique.

Fig. 3 shows a simplified and "more natural" version of the corpus from Fig. 1, which eliminates these no-op transformations. As illustrated in the figure, this causes a purely syntactic technique to learn a different function, `f_1`, which abstracts over an unscaled unit polygon. Because the scaling transformation is now outside the abstraction, it must be repeated five times. As a result, although the simplified input corpus `C` from Fig. 3 is smaller than the original corpus `A` (196 AST nodes instead of 208), its compressed version `D` is larger (81 AST nodes instead of 72)! In other words, syntactic library learning is not robust with respect to semantics-preserving code variations.

In contrast, our tool BABBLE can take the simplified corpus `C` as input and still discover, in a fraction of a second, the scaled polygon abstraction `f_0`, yielding the compressed corpus `B` of size 72. In the rest of this section, we illustrate how BABBLE achieves this using our new algorithm, **library learning modulo equational theory** (LLMT).

---

1We suspect that these transformations ended up in the dataset of Wong et al. [2022] because it was generated programmatically from human-designed abstractions, such as "scaled polygon".
Simplified DSL. In the rest of this section we use a tailored version of the 2D CAD DSL with the following additional constructs:

\[
\begin{align*}
\text{scale}(F, s) & \quad \text{scale } F \text{ by a factor of } s \\
\text{repRot}(F, n, \theta) & \quad \text{a special case of } \text{repeat} \text{ that only performs rotation between iterations} \\
\text{side}(n) & \quad \text{a side of a regular unit } n\text{-gon}
\end{align*}
\]

These are expressible in the original DSL, and could be even discovered with library learning, given an appropriate corpus; we treat them as primitives here for the sake of simplifying presentation.

2.1 Representing Equivalent Terms with E-Graphs

To exploit equivalences during library learning, BABBLE takes as input a domain-specific equational theory. For our running example, let us assume that the theory contains a single equation:

\[ F \equiv \text{scale}(F, 1) \quad (1) \]

which stipulates that any figure \( F \) is equivalent to itself scaled by one. With this equation in hand, it is possible to “rewrite” corpus C into corpus A, and from there learn the optimal compressed corpus B by purely syntactic techniques. The challenge is that there are infinitely many alternative corpora C may be rewritten to; how do we know which to pick to maximize syntactic alignment, and thus the chance of discovering an optimal abstraction?

Instead of trying to guess the best equivalent corpus or enumerating them one by one, BABBLE uses equality saturation [Tate et al. 2009; Willsey et al. 2021]. Equality saturation takes as input a term \( t \) and a set of rewrite rules, and finds the space of all terms equivalent to \( t \) under the given rules; this is possible due to the high degree of sharing provided by the e-graph data structure, which can compactly represent the resulting space.

Fig. 4 (left) shows the e-graph built by equality saturation for the blue term in Fig. 3—represented in the simplified DSL as \( \text{repRot} (\text{side}(6), 6, 2\pi/6) \)—using the rewrite rule (1). The blue part of the graph represents the original term, and the gray part is added by equality saturation. The solid rectangles in the e-graph denote e-nodes (which are similar to regular AST nodes), while the dashed rectangles denote e-classes (which represent equivalence classes of terms). Importantly, the edges in the e-graph go from e-nodes to e-classes, which enables compact representation of programs with variation in sub-terms: for example, making e-class \( c_1 \) the first child of the \( \text{repRot} \) node,
enables it to represent both terms \( \text{repRot(side(6), 6, 2\pi/6)} \) and \( \text{repRot(scale(side(6), 1), 6, 2\pi/6)} \) without duplicating their common parts. Furthermore, because e-graphs can have cycles, they can also represent infinite sets of terms: for example, the e-class \( c_1 \) represents all terms of the form: \( \text{side(6)}, \text{scale(side(6), 1)}, \text{scale(scale(side(6), 1), 1)} \), etc. Because this e-graph represents the space of all equivalent terms up to the rewrite (1), the term we seek for library learning, namely \( \text{scale(repRot(side(6), 6, 2\pi/6), 1)} \), is also represented in the e-class \( c_0 \).

### 2.2 Candidate Generation via E-Graph Anti-unification

After building an e-graph from the given corpus by running equality saturation with the given equational theory, the next step in library learning is to generate candidate patterns that capture syntactic similarities across the corpus. The challenge is to generate sufficiently few candidates to make library learning tractable, but sufficiently many to achieve good compression. We illustrate candidate generation using the e-graph in Fig. 4 (right), which represents the part of our corpus consisting of the (saturated) blue and brown terms. Recall that the optimal pattern—which corresponds to the scaled polygon abstraction \( f_0 \)—is:

\[
P_0 = \text{scale(repRot(side(X), X, 2\pi/X), Y)}
\]  

Prior work on DREAMCoder generates patterns by picking a random fragment from the corpus, and then replacing arbitrarily chosen subterms with pattern variables. For example, to generate the pattern \( P_0 \), DREAMCoder needs to pick the entire brown subterm as a fragment, and then decide to abstract over its subterms 8 and 2. This approach successfully restricts the set of candidates from all syntactically valid patterns to only those that have at least one match in the corpus; however, since there are too many ways to select the subterms to abstract over, this space is still too large to explore exhaustively beyond small corpora of short programs. In BABBLE, this problem is exacerbated by equality saturation, since the e-graph often contains exponentially or infinitely more programs than the original corpus.

**Generality Filters.** To further prune the set of viable candidates in BABBLE, we identify two classes of patterns that can be safely discarded, either because they are too concrete or too abstract. First, a pattern like \( \text{repRot(side(8), X, 2\pi/8)} \) can be discarded because it is too concrete for this corpus: the corresponding abstraction can be applied only once, essentially replacing the sole matching term, \( \text{repRot(side(8), 8, 2\pi/8)} \) with \( (\lambda x \rightarrow \text{repeat(side(8), x, 2\pi/8)}) \) 8, which only adds more AST nodes to the corpus. Second, a pattern like \( \text{repRot(side(X), X, Y)} \) can be discarded because it is too
abstract} for this corpus: everywhere it matches, a more concrete pattern $\text{repRot}(\text{side}(X), X, 2\pi/X)$ would also match; using the more concrete pattern leads to better compression, since the actual arguments in its applications are both fewer and smaller: $f 6 2\pi/6$ and $f 8 2\pi/8$ vs. $f 6$ and $f 8$.

Thus, our first key insight is to restrict the set of candidates to the most concrete patterns that match some pair of subterms in the (saturated) corpus. For example, pattern $P_0$ is the most concrete pattern that matches the two terms

\begin{align*}
\text{scale}(\text{repRot}(\text{side}(6), 6, 2\pi/6), 1) \\
\text{scale}(\text{repRot}(\text{side}(8), 8, 2\pi/8), 2)
\end{align*}

represented in Fig. 4 by the e-classes $c_0$ and $c_2$, respectively.

**Term Anti-unification.** Computing the most concrete pattern that matches two given terms is known as anti-unification (AU) [Plotkin 1970; Reynolds 1969]. AU works by a simple top-down traversal of the two terms, replacing any mismatched constructors by pattern variables. For example, to anti-unify the terms (3) and (4), we start from the root of both terms; since both AST nodes share the same constructor $\text{scale}$, it becomes part of the pattern and we recurse into the children. We eventually encounter a mismatch, where the term on the left is 6 and the term on the right is 8; so we create a fresh pattern variable $X$ and remember the anti-substitution $\sigma = \{(6, 8) \mapsto X\}$. When we encounter a mismatch in the denominator of the angle, we look up the pair of mismatched terms $(6, 8)$ in $\sigma$; because we already created a variable for this pair, we simply return the existing variable $X$. The final mismatch is $(1, 2)$ in the second child of $\text{scale}$; since this pair is not yet in $\sigma$, we create a second pattern variable, $Y$. At this point, the resulting anti-unifier is the pattern $P_0$ (2).

Anti-unifying a single pair of terms is simple and efficient. However, in LLMT we want to anti-unify all pairs of subterms that can occur in any corpus equivalent (modulo the given equational theory) to the original\(^3\). Explicitly enumerating all equivalent corpora represented by the e-graph and then performing AU on each pair of subterms is infeasible. Instead, **Babble** performs anti-unification directly on the e-graph.

**From Terms to E-Classes.** We first explain how to anti-unify two e-classes. This operation takes as input a pair of e-classes and returns a set of dominant anti-unifiers, i.e., a set of patterns that (1) match both e-classes, and (2) is guaranteed to include the best abstraction among the most concrete patterns that match pairs of terms represented by the two e-classes.

Consider computing $\text{AU}(c_1, c_4)$ for the e-classes $c_1$ and $c_4$ in the e-graph from Fig. 4 (right). AU still proceeds as a top-down traversal, but in this context we must check whether two e-classes have any constructors in common. In this case they do: both a $\text{side}$ constructor and a $\text{scale}$ constructor. Let us first pick the two $\text{side}$ constructors and recurse into their only child, computing $\text{AU}({6}, {8})$. These two e-classes have no matching constructors, so AU simply returns a pattern variable, similarly to term anti-unification; this yields the first pattern for $c_1$ and $c_4$: $\text{side} X$.

Recall, however, that $c_1$ and $c_4$ also have matching $\text{scale}$ constructors. This is where things get interesting: these constructors are involved in a cycle (their first child is the parent e-class itself). If we let AU follow this cycle, the set of anti-unifiers becomes infinite:

$\text{AU}(c_1, c_4) = \{\text{side} X\} \cup \{\text{scale} p \mid p \in \text{AU}(c_1, c_4)\}$

Fortunately, we can show that $\text{side} X$ dominates all the other patterns from this set, because their pattern variables—in this case, just $X$—match the same e-classes, but they are also larger (in Sec. 4

\[^2\] As we discuss in Sec. 3 this can in theory eliminate optimal patterns, but our evaluation shows that it works well in practice.

\[^3\] A careful reader might be wondering if we need to compute infinitely many anti-unifiers because there might be infinitely many equivalent corpora. As we explain shortly, there are only finitely many patterns that are viable abstraction candidates.
we show how this domination relation lets us prune other patterns, not just those caused by cycles. Hence AU(c₁, c₄) simply returns \{side X\}.

Following the same logic for the root e-classes of the two polygons, c₀ and c₂, AU(c₀, c₂) yields that pattern P₀(2), which is required to learn the optimal abstraction.

**From E-Classes to E-Graphs.** To obtain the set of all candidate abstractions, we need to perform anti-unification over all pairs of e-classes in the e-graph. Clearly, these computations have overlapping subproblems (for example, we have to compute AU(c₁, c₄) as part of AU(c₀, c₂) and AU(c₀, c₃)). To avoid duplicating work, babble uses an efficient dynamic programming algorithm that processes all pairs of e-classes in a bottom-up fashion.

### 2.3 Candidate Selection via Targeted Common Subexpression Elimination

We now illustrate the final step of library learning in babble: given the set of candidate abstractions generated by e-graph anti-unification, the goal is to pick a subset that gives the best compression for the corpus as a whole. For example, the candidates generated for the corpus from Fig. 3 include:

\[
\begin{align*}
  f₀ & = \lambda x y \to \text{scale}(\text{repRot}(\text{side}(x), x, 2\pi/x), y) \quad \text{scaled regular } x\text{-gon} \\
  f₁ & = \lambda x \to \text{repRot}(\text{side}(x), x, 2\pi/x) \quad \text{regular } x\text{-gon} \\
  f₂ & = \lambda y \to \text{scale}(\text{repRot}(\text{side}(6), 6, 2\pi/6), y) \quad \text{scaled hexagon}
\end{align*}
\]

It is not immediately clear which abstraction is better: f₀ matches more terms than f₂, but f₂ requires fewer arguments (so if we have enough scaled hexagons in the corpus and only one octagon, it might be better to leave the octagon un-abstracted). On the other hand, f₁ might be better, since it does not introduce the redundant transformation on the blue hexagons. Finally, if we pick f₀ and f₂ together, we can also abstract the definition of f₂ as f₀ 6, thereby getting additional reuse! As you can see, candidate selection is highly non-trivial, since it needs to take into account the choice between different equivalent programs in the e-graph, and the fact that some abstractions can be defined using others.

**Reduction to E-Graph Extraction.** To overcome this difficulty, we once again leverage e-graph and equality saturation. Our second key insight is that selecting the optimal subset of abstractions can be reduced to the problem of extracting the smallest term from an e-graph in the presence of common sub-expressions.

To illustrate this reduction, let us limit our attention to only two candidate abstractions, f₀ and f₁, defined above. babble converts each of the candidate patterns and its corresponding abstraction
into a rewrite rule that introduces a local \( \lambda \)-abstraction followed by application into the corpus; for our two abstractions these rules are:

\[
\text{scale}(\text{repRot}(\text{side}(X), X, 2\pi/X), Y) \Rightarrow f_0 \; X \; Y \tag{5}
\]

\[
\text{repRot}(\text{side}(X), X, 2\pi/X) \Rightarrow f_1 \; X \tag{6}
\]

The result of applying these rules to the e-graph from Fig. 4 (right) is shown in Fig. 5. For example, you can see that the e-class \( c_2 \) (which represents the scale-2 octagon) now stores an alternative representation: \( f_0 \ 8 \ 2 \). The e-class \( c_0 \) (unit hexagon) has representations using either \( f_0 \) or \( f_1 \), because this class matches both rewrite rules (5) and (6) above. Note also that because the definitions of the \( \lambda \)-abstractions are also stored in the e-graph, equality saturation can use the above rewrite rules inside their bodies, to make one function use another: for example, one term stored for the definition of \( f_0 \) is

\[
\lambda x \; y \rightarrow \text{scale}(f_1 \; x, \; y). \]

Once we have built the version of the e-graph with local lambdas for all the candidate abstractions, all that is left is to extract the smallest term out of this e-graph. The tricky part is that we want to count the size of the duplicated lambdas only once. For example, in Fig. 5, \( f_0 \) is applied twice (in \( c_0 \) and \( c_2 \)); if term extraction were to choose both of these e-nodes, we want to treat their first child (the definition of \( f_0 \)) as a common sub-expression, whose size contributes to the final expression only once. Intuitively, this is because we can “float” these lambdas into top-level \texttt{let}-bindings, thereby defining \( f_0 \) only once, and replacing each local lambda with a name.

Extraction with common sub-expressions is a known but notoriously hard problem, which is traditionally reduced to integer linear programming (ILP) [Wang et al. 2020; Yang et al. 2021]. Because the scalability of the ILP approach is very limited, we have developed a custom extraction algorithm, which scales better by using domain-specific knowledge and approximation.

**Extraction Algorithm.** The main idea for making extraction more efficient is that for library learning we are only interested in sharing a certain type of sub-terms: namely, the \( \lambda \)-abstractions. Hence for each e-class we only need to keep track of the best term for each possible library (i.e. each subset of \( \lambda \)-abstractions). More precisely, for each e-class and library, we keep track of (1) the smallest size of the library (2) the smallest size of the program refactored using this library (counting the \( \lambda \)-abstractions as a single node). We compute and propagate this information bottom-up through the e-graph. Once this information is computed for the root e-class (that represents the entire corpus), we can simply choose the library with the smallest total size.

For example, for the e-class \( c_2 \) from Fig. 5, with the empty library \( \emptyset \), the size of library is 0 and the size of the smallest program (\( \text{scale}(\text{repRot}(\text{side}(8), 8, 2\pi/8), 2) \)) is 9; with the library \( \{f_0\} \), the size of the library is 9 and the size of the smallest program \( (f_0 \ 2 \ 8) \) is 3; while with the library \( \{f_1\} \), the size of the library is 7 and the size of the smallest program \( (\text{scale}(f_1 \ 8, \ 2)) \) is 4. Clearly for this e-class introducing library functions is not paying off yet \( (0 + 9 < 7 + 4 < 9 + 3) \), since each one can be only used once. This situation changes, however, as we move up towards the root. Already for the parent e-class of \( c_0 \) and \( c_2 \), the cost of introducing \( \{f_0\} \) and \( \{f_1\} \) is amortized: the size of the smallest program is 17 with the empty library and 7 with either \( \{f_0\} \) or \( \{f_1\} \), so \( \{f_1\} \) is already worthwhile to introduce \( (9 + 7 < 0 + 17) \). Including even more programs with scaled polygons eventually makes the library \( \{f_0\} \) the most profitable of the four subsets.

Since in a larger corpus, keeping track of all subsets of candidate abstractions is not feasible, BABBLE provides a way to trade off scalability and precision by using a beam search approach.
3 LIBRARY LEARNING OVER TERMS

In this section we formalize the problem of library learning over a corpus of terms and motivate our first core contribution: generating candidate abstractions via anti-unification. Sec. 4 generalizes this formalism to library learning over an e-graph. For simplicity of exposition, our formalization of library learning is first order, that is, the initial corpus does not itself contain any λ-abstractions, and all the learned abstractions are first-order functions (the BABLE implementation does not have this limitation).

3.1 Preliminaries

Terms. A signature \( \Sigma \) is a set of constructors, each associated with an arity. \( \mathcal{T}(\Sigma) \) denotes the set of terms over \( \Sigma \), defined as the smallest set containing all \( s(t_1, \ldots, t_k) \) where \( s \in \Sigma, k = \text{arity}(s) \), and \( t_1, \ldots, t_k \in \mathcal{T}(\Sigma) \). We abbreviate nullary terms of the form \( s() \) as \( s \). The size of a term \( \text{size}(t) \) is defined in the usual way (as the number of constructors in the term). We use subterms \( (t) \) to denote the set of all subterms of \( t \) (including \( t \) itself). We assume that \( \Sigma \) contains a dedicated \textit{variadic} tuple constructor, written \( \langle t_1, \ldots, t_n \rangle \), which we use to represent a corpus of programs as a single term (\( \langle \rangle \) does not contribute to the size of a term).

Patterns. If \( X \) is a denumerable set of variables, \( \mathcal{T}(\Sigma, X) \) is a set of patterns, i.e. terms that can contain variables from \( X \). A pattern is linear if each of its variables occurs only once: \( \forall X \in \text{vars}(p).\text{occurs}(X, p) = 1 \). A substitution \( \sigma = [X_1 \mapsto p_1, \ldots, X_n \mapsto p_n] \) is a mapping from variables to patterns. We write \( \sigma(p) \) to denote the application of \( \sigma \) to pattern \( p \), which is defined in the standard way. We define the size of a substitution \( \text{size}(\sigma) \) as the total size of its right-hand sides.

A pattern \( p \) is more general than (or matches) \( p' \), written \( p \sqsubseteq p' \), if there exists \( \sigma \) such that \( p' = \sigma(p) \); we will denote such a \( \sigma \) by \( \text{match}(p', p) \). For example \( X + 1 \sqsubseteq X + Y \) with \( \text{match}(X + 1, X + Y) = [Y \mapsto 1] \). The relation \( \sqsubseteq \) is a partial order on patterns, and induces an equivalence relation \( p_1 \sim p_2 \) which is defined in the standard way up to variable renaming. In the following, we always distinguish patterns only up to this equivalence relation.

The join of two patterns \( p_1 \sqcup p_2 \) is called their anti-unifier—is the least general pattern that matches both \( p_1 \) and \( p_2 \); the join is unique up to \( \sim \). Note that \( (\mathcal{T}(\Sigma, X), \sqsubseteq, \sqcup, \sqcap, \top = X) \) is a join semi-lattice (part of Plotkin’s \textit{subsumption lattice} [Plotkin 1970]). Consequently, the join can be generalized to an arbitrary set of patterns.

A context \( C \) is a pattern with a single occurrence of a distinguished variable \( \circ \). We write \( C[p] \rightarrow p \) in context \( C \)—as a syntactic sugar for \( C(C[p]) \). A rewrite rule \( R \) is a pair of patterns, written \( p_1 \Rightarrow p_2 \). Applying a rewrite rule \( R \) to a term or pattern \( p \), written \( R(p) \) is defined in the standard way: \( R(p) = (\text{match}(p, p_1))(p_2) \) if \( p \sqsubseteq p_1 \) and undefined otherwise. A pattern \( p \) can be re-written in one step into \( q \) using a rule \( R \), written \( p \rightarrow^R_R q \), if there exists a context \( C \) such that \( p = C[p'] \) and \( q = C[R(p')] \). The reflexive-transitive closure of this relation is the rewrite relation \( \Rightarrow \), where \( \Rightarrow \) is a set of rewrite rules.

Compressed Terms. Compressed terms \( \hat{\mathcal{T}}(\Sigma, X) \) are of the form:

\[
\hat{i} ::= X \mid s(\hat{i}_1, \ldots, \hat{i}_k) \mid (\lambda X_1 \ldots X_n \rightarrow \hat{i}) \hat{i}_1 \ldots \hat{i}_n
\]

In other words, compressed terms may contain applications of a \( \lambda \)-abstraction with zero or more binders to the same number of arguments. Note that this is a first-order language in the sense that all abstractions are fully applied. Importantly, we define \( \text{size}(\hat{i}) \) in such a way that multiple occurrences of a \( \lambda \)-abstraction are \textit{only counted once}. For simplicity of accounting, the size of an
\[\langle f(g(a) + g(a)) + (g(1) + h(2)) \]\[, f(g(b) + g(b)) + (g(3) + h(4)) \]\[, g(5) + h(6)\] where \[\hat{f}_1 = \lambda Y Z \to g(Y) + h(Z)\]\[\hat{f}_2 = \lambda X Y Z \to f(X + X) + \hat{f}_1 Y Z\]

Fig. 6. Library learning. Left: initial term (size 29). Right: an optimal solution with two abstractions, one of which uses the other (size 26). A solution with \[\hat{f}_2 = \lambda X Y Z \to f(g(X) + g(X)) + \hat{f}_1 Y Z\] also has size 26.

application \((\lambda X \to \hat{t}_1) \hat{t}_2\) is defined as \(\text{size}(\hat{t}_1) + \sum \text{size}(\hat{t}_2)\), that is, abstraction nodes themselves do not add to the size.\(^4\)

Beta-reduction on compressed terms, denoted \(\hat{t}_1 \to^\beta \hat{t}_2\), is defined in the usual way:

\[(\lambda X \to \hat{t}_1) \hat{t}_2 \to^\beta [X \mapsto \hat{t}_2](\hat{t}_1)\]

where substitution on compressed terms is the standard capture-avoiding substitution for \(\lambda\)-calculus.

Note that because our language is first-order and has no built-in recursion, it is strongly normalizing (the proof of this statement, as well as other proofs omitted from this section, can be found in the supplementary material). Hence, the order of \(\beta\)-reductions is irrelevant, and without loss of generality we can define evaluation on compressed terms, \(\hat{t}_1 \to^\beta \hat{t}_2\), to follow applicative order, i.e. innermost \(\beta\)-redexes are reduced first. As a result, any application reduced during evaluation has the form \((\lambda X \to p_1) p_2\), that is, neither the body \(p_1\) nor the actual arguments \(p_2\) contain any redexes (and hence any \(\lambda\)-abstractions). This simplifies several aspects of our formalization; for example, there is no need for \(\alpha\)-renaming, since with no binders in \(p_1\), no variable capture can occur.

**Problem Statement.** We can now formalize the library learning problem as follows: given a term \(t \in T(\Sigma)\), the goal is to find the smallest compressed term \(\hat{t} \in \hat{T}(\Sigma, X)\) that evaluates to \(t\) (i.e. \(\hat{t} \to^\beta t\)). The reason such \(\hat{t}\) may be smaller than \(t\), is that it may contain multiple occurrences of the same \(\lambda\)-abstraction (applied to different arguments), whose size is only counted once. An example is shown in Fig. 6.

Although in full generality the solution might include nested lambdas with free variables (defined in the outer lambdas), in the rest of the paper we restrict our attention to global library learning, where all lambdas are closed terms. This is motivated by the purpose of library learning to discover reusable abstraction for a given problem domain. The solution in Fig. 6 already has this form.

### 3.2 Pattern-Based Library Learning

At a high-level, our approach to library learning is to use patterns that occur in the original corpus as candidate bodies for \(\lambda\)-abstractions in the compressed corpus. Looking at the example in Fig. 6, it is not immediately obvious that using just the patterns from the original corpus is sufficient, since the body of \(f_2\) contains an application of \(f_1\). Perhaps surprisingly, this is not an issue: the key idea is that we can compress \(t\) into \(\hat{t}\) by inverting the evaluation \(\hat{t} \to^\beta t\), and because the evaluation order is applicative, the rewritten sub-term at every step will not contain any \(\beta\)-redexes.

**Compression.** More formally, given a pattern \(p\), let us define its compression rule (or \(\kappa\)-rule for short) as the rewrite rule

\[\kappa(p) \triangleq p \Rightarrow (\lambda \overline{X} \to p) \overline{X}\]

where \(\overline{X} = \text{vars}(p)\)

In other words, \(\kappa(p)\) will replace any term matching \(p\) with an application of a function whose body is exactly \(p\). For example, if \(p = g(Y) + h(Z)\), its \(\kappa\)-rule is \(g(Y) + h(Z) \Rightarrow (\lambda Y Z \to g(Y) + h(Z)) Y Z\).

\(^4\)Hereafter, \(\overline{a}\) stands for a sequence of elements of syntactic class \(a\) and \(\epsilon\) denotes the empty sequence.
Note that on the right-hand side of this rule only the free occurrences of $Y$ and $Z$ will be substituted during rewriting; the bound $Y$ and $Z$ will be left unchanged, following the usual semantics of substitution for $\lambda$-calculus. For example, this rule can rewrite the third term in Fig. 6 (left) as follows:

\[
g(S) + h(6) \rightarrow_{1}^{\kappa(g(Y)+h(Z))} (\lambda Y \ Z \rightarrow g(Y) + h(Z))\]

A sequence of $\kappa$-rewrites $t \rightarrow_{1}^{\kappa(p_1)} \ldots \rightarrow_{1}^{\kappa(p_n)} \hat{t}$, where all $p_i \in P$, is called a compression of $t$ into $\hat{t}$ using patterns $P$ and written $t \rightarrow_{\kappa(P)}^{\hat{t}}$. We can now show that the library learning problem is equivalent to finding the smallest compression of $t$ using only patterns that occur in $t$.

**Theorem 3.1 (Soundness and Completeness of Pattern-Based Library Learning).** For any term $t \in T(\Sigma)$ and compressed term $\hat{t} \in \hat{T}(\Sigma, X)$:

**(Soundness)** If $t$ compresses into $\hat{t}$, then $\hat{t}$ evaluates to $t$: $\forall P : t \rightarrow_{\kappa(P)}^{\hat{t}} \Rightarrow \hat{t} \rightarrow^{\beta} t$.

**(Completeness)** If $\hat{t}$ is a solution to the (global) library learning problem, then $t$ compresses into $\hat{t}$ using only patterns that have a match in $t$: $\hat{t} \in \arg \min_{\hat{t}'} \text{size}(\hat{t}') \Rightarrow t \rightarrow_{\kappa(P)}^{\hat{t}}$, where $P = \{ p \in T(\Sigma, X) \mid t' \in \text{subterms}(t), t' \subseteq p \}$.

The proof can be found in the supplementary material.

**Example.** Consider once again the library learning problem in Fig. 6. Here the set of patterns used to compress the original corpus into the solution on the right is:

\[
p_1 = g(Y) + h(Z) \quad p_2 = f(X + X) + g(Y) + h(Z)
\]

Rewriting the first term of the corpus proceeds in two steps (the redexes of $\kappa$-steps are highlighted):

\[
f(g(a) + g(a)) + (g(1) + h(2)) \rightarrow_{1}^{\kappa(p_2)} (\lambda X \ Y \ Z \rightarrow f(X + X) + g(Y) + h(Z)) \ g(a) \ g(a) \ (g(1) + h(2)) \rightarrow_{1}^{\kappa(p_1)} (\lambda X \ Y \ Z \rightarrow f(X + X) + (\lambda Y \ Z \rightarrow g(Y) + h(Z)) \ Y \ Z) \ g(a) \ g(a) \ (g(1) + h(2))
\]

In other words, we first rewrite the entire term using $p_2$, and then rewrite inside the body of the introduced abstraction using $p_1$ (note that this order of compression is the inverse of the applicative evaluation order). The second term of the corpus compresses analogously; the third term compresses in a single step using $p_1$. Although this is not obvious from the rewrite sequence above, the resulting compressed corpus is indeed smaller than the original thanks to sharing of both $\lambda$-abstractions, as illustrated on the right of Fig. 6.

**Library Learning as Term Rewriting.** Theorem 3.1 reduces library learning to a term rewriting problem. Namely, given a term $t$ and a finite set of rewrite rules $R = \{ \kappa(p) \mid t' \in \text{subterms}(t), t' \subseteq p \}$, our goal is to find a minimal-size term $\hat{t}$ such that $t \rightarrow^{R} \hat{t}$, which is a standard formulation in term rewriting. Unfortunately, this particular problem is notoriously difficult because (a) the rule set $R$ is very large for any non-trivial term $t$, and (b) our size function is non-local (it takes sharing into account) In the rest of this section we discuss how we can prune the rule set $R$ to reduce it to a tractable size. Sec. 5 discusses how we tackle the remaining term rewriting problem using the equality saturation technique [Tate et al. 2009; Willsey et al. 2021].

### 3.3 Pruning Candidate Patterns

In this section, we discuss which patterns can be discarded from consideration when constructing the set of $\kappa$-rules $R$ for the term rewriting problem.

**Cost of a Pattern.** Consider a compression $t \rightarrow_{\kappa(P)}^{\hat{t}}$ where each pattern $p \in P$ is used some number $n$ times, with substitutions $\sigma^p_1, \ldots, \sigma^p_n$. We can break down the total amount of compression
into contributions of individual patterns:

\[ \text{size}(t) - \text{size}(i) = \sum_{p \in \mathcal{P}} \text{cost}(p, \{\sigma_1^p, \ldots, \sigma_n^p\}) \]

The cost of a pattern \( p \), in turn, consists of three components. The cost of introducing the abstraction is the size of its body, i.e. \( \text{size}(p) \). The cost of using an abstraction—\( \text{use}(p, \sigma) \) (7)—includes the application itself and the size of the arguments. The cost saved by using an abstraction—\( \text{save}(p, \sigma) \) (8)—is just the cost of the term matched by \( p \) (i.e. the redex of the corresponding \( \kappa \)-step).

\[
\text{use}(p, \sigma) = 1 + \sum_{X \in \text{vars}(p)} \text{size}(\sigma(X)) \\
\text{save}(p, \sigma) = \text{size}(\sigma(p)) = \text{size}(p) + \sum_{X \in \text{vars}(p)} \text{occurs}(X, p) \cdot (\text{size}(\sigma(X)) - 1)
\]

The total cost of \( p \) is the cost of introducing the abstraction paid a single time, plus the cost of each use, minus what you save for each application:

\[
\text{cost}(p, \{\sigma_1, \ldots, \sigma_n\}) = \text{size}(p) + \sum_{\sigma_i} (\text{use}(p, \sigma_i) - \text{save}(p, \sigma_i))
\]

When \( p \) is linear (all \( \text{occurs}(X, p) = 1 \)), the cost depends only on \( n \) but not on the substitutions \( \sigma_i \):

\[
\text{cost}(p, \{\sigma_1, \ldots, \sigma_n\}) = \text{size}(p) + \sum_{\sigma_i} (1 - \text{size}(p) + |\text{vars}(p)|)
\]

\[
= \text{size}(p) + n \cdot (1 - \text{size}(p) + |\text{vars}(p)|)
\]

We can show that a pattern \( p \) with a non-negative cost can be safely discarded, that is: there exists another compression using only \( \mathcal{P} \setminus \{p\} \), whose result is at least as small.

**Trivial Patterns.** Based on this analysis, any linear pattern \( p \) with \( \text{skel}(p) \leq 1 \) can be discarded, where \( \text{skel}(p) = \text{size}(p) - |\text{vars}(p)| \) is the size of \( p \)’s “skeleton”, i.e. it’s body without the variables. Intuitively, the skeleton of \( p \) is simply too small to pay for introducing an application. In this case, \( \text{cost}(p, \_ ) > 0 \) independently of how many times it is used. We refer to such patterns as trivial. Examples of trivial patterns are \( X \) and \( X + Y \).

**Patterns with a Single Match.** We can show that patterns with only a single match in the corpus can also be discarded. If \( p \) has a single match in \( t \), then it can appear at most once in any compression of \( t \). If \( p \) is linear, \( \text{cost}(p, \_ ) = 1 + |\text{vars}(p)| \), which is always positive, so \( p \) can be discarded. But what about non-linear patterns, where even a single \( \kappa \)-step can decrease size thanks to variable reuse? It turns out that any non-linear pattern with a single match can always be replaced by a nullary pattern (with no variables) that is more optimal.

Without loss of generality, assume that \( p \) has a single variable \( X \) that occurs \( m > 1 \) times, and let its sole \( \kappa \)-step be \( \sigma(p) \rightarrow^{\kappa}_{\text{skel}} (\lambda X \rightarrow p) \sigma(X) \). The size of the right-hand side is \( \text{size}(p) + 1 + \text{size}(\sigma(X)) \), or, rewritten in terms of \( p \)’s skeleton: \( 1 + (\text{skel}(p) + m) + \text{size}(\sigma(X)) \). Instead, we can rewrite the same redex \( \sigma(p) \rightarrow^{\kappa}_{\text{skel}} p' \) using \( m \) applications of the rule \( R \triangleq \sigma(X) \Rightarrow (\lambda e \rightarrow \sigma(X)) \epsilon \). This is a \( \kappa \)-rule for a nullary pattern \( \sigma(X) \) with no variables (hence the corresponding \( \lambda \)-abstraction has zero binders). The size of \( p' \) obtained in this way is \( \text{size}(\sigma(X)) + m + \text{skel}(p) \) (where the former is the size of the shared \( \lambda \)-abstraction, \( m \) is the number of applications, and \( \text{skel}(p) \) is the size of the term around the applications). As you can see, this term is one smaller than the one we get by applying \( p \). Intuitively, this result says that instead of using a non-linear pattern that occurs only once, it is better to perform common sub-expression elimination.

**Parameterization Lattice.** Eliminating from consideration all patterns with fewer than two matches in the corpus suggests an algorithm for generating a complete set \( \mathcal{P} \) of candidate patterns:
(1) start from the set $\mathcal{P}^2(t) = \{ t_1 \sqcup t_2 \mid t_1 \in \text{subterms}(t) \}$ of all pairwise joins of subterms of the input program, (2) explore all elements of the subsumption semi-lattice above those patterns, by gradually replacing sub-patterns with variables, until we hit trivial patterns at the top of the lattice.

We will refer to this semi-lattice above $t$ whose join is distinct from all their pairwise joins (otherwise some pattern is more optimal than $t$). Right: an example where it is insufficient to consider pairwise joins to obtain an optimal pattern.

Unfortunately, computing the set $\mathcal{P}(t)$ for $t = (f(a + b), f(a + c), f(b + c))$ is expensive. (left).

Fig. 7. Left: fragment of the parametrization lattice for the term $t = (f(a + b), f(a + c), f(b + c))$; only filled black circles correspond to candidate patterns: hollow circles match fewer than two terms and gray circles are trivial. Right: an example where it is insufficient to consider pairwise joins to obtain an optimal pattern.

**Approximation.** In practice, computing the set $\mathcal{P}^2(t)$ is feasible: although there are quadratically many pairs of subterms, most of them do not have a common constructor at the root, and hence their join is trivially $X$. An example is the join of $t$ with any of its subterms in Fig. 7 (left). Unfortunately, generalizing the patterns from $\mathcal{P}^2(t)$ according to the parameterization lattice (Fig. 7) is expensive. For this reason, BABBLE adopts an approximation and simply uses $\mathcal{P}^2(t)$ as the set of candidates.

This approximation makes our pattern generation theoretically incomplete. Consider a pattern $p \in \mathcal{P}(t) \setminus \mathcal{P}^2(t)$; there are two reasons why we might need $p$ in the optimal compression of $t$:

1. there is no $p' \in \mathcal{P}^2(t)$ with the same set of matches as $p$, or
2. there is such a $p'$ but it has few enough matches that its larger size does not pay off.

The first kind of incompleteness occurs when $p$ matches a set of subterms $t_1, \ldots, t_n$ ($n > 2$), whose join is distinct from all their pairwise joins (otherwise some $t_i \sqcup t_j \in \mathcal{P}^2(t)$ would also match all $t_1, \ldots, t_n$). An example is shown in Fig. 7 (right), where the three subterms in question are $f(a + a)$, $f(a + c)$, and $f(c + c)$. In this case, an optimal compression might use the pattern $f(X + Y)$ to rewrite all three subterms, but our approximation would only include the patterns $f(a + X)$, $f(X + X)$, and $f(X + c)$, each of which can rewrite only two of the three subterms.

The second kind of incompleteness occurs when there exists $p' \in \mathcal{P}^2(t)$ that has the same set of matches as $p$, despite being strictly more specific, and yet using $p'$ instead of $p$ still does not pay off. The understand when this happens, consider the difference in costs between $p$ and $p'$, assuming that they are both used to rewrite the same $n$ subterms (i.e. their save cost is the same):

$$
\text{cost}(p', \sigma') - \text{cost}(p, \sigma) = \text{size}(p') - \text{size}(p) + \sum_i (\text{use}(p', \sigma'_i) - \text{use}(p, \sigma_i))
$$

$$
= \text{size}(p') - \text{size}(p) + \sum_i (\text{size}(\sigma'_i) - \text{size}(\sigma_i))
$$

Because $p'$ is strictly more specific than $p$, we know that $\text{size}(p') \geq \text{size}(p)$, but all its substitutions $\sigma'_i$ must be strictly smaller than $\sigma_i$. Hence, with enough uses, $p'$ is bound to become more compressive than $p$; when there are just a few uses, however, $p$ can be more optimal. For example, consider the corpus $\{ C[f(1, 2, 3)], C[f(4, 5, 6)] \}$, where $C$ is some sufficiently large context. Here, a more general pattern $p = C[X]$ is more optimal than the more specific $p' = C[f(X, Y, Z)]$, because $\text{size}(p') - \text{size}(p) = 3$, each use of $p'$ is only one node cheaper, and there are only two uses.

Despite the lack of theoretical completeness guarantees, we argue that restricting candidate patterns to $P^2(t)$ is a reasonable trade-off. Note, that the counter-examples above are quite contrived, and they no longer apply once the corpus contains sufficiently many and sufficiently diverse instances of a pattern (for example, adding $f(b, b)$ to the first corpus would make $f(X, Y)$ appear in $P^2(t)$, and adding just one more occurrence of $p'$ into the second corpus would make it as optimal as $p$). Our empirical evaluation confirms that this approximation works well in practice.

4 LIBRARY LEARNING MODULO EQUATIONAL THEORY

E-Graphs. Let $I$ be a denumerable set of e-class ids. An e-graph $G$ is a triple $(C, M, r)$, where $C$ is a set of e-classes, $M : I \rightarrow C$ is an e-class map, and $r \in I$ is the root class id. An e-class $c \in C$ is a set of e-nodes $n \in N$, and an e-node is a constructor applied to e-class ids. The syntax of e-classes and e-nodes is summarized in Fig. 8 (left). An e-graph has to satisfy the congruence invariant, which states that the e-graph has no two identical e-nodes (or alternatively, that all e-classes are disjoint).

The denotation of an e-graph—the set of terms it represents—is the denotation of its root e-class $[M(r)]$, where the denotation of e-classes and e-nodes is defined mutually-recursively in Fig. 8 (right). Note that the denotation can be infinite if the e-graph has cycles. An e-graph induces an equivalence relation $\equiv^G$, where $t_1 \equiv^G t_2$ iff there exists an e-class $c \in C$ such that $t_1 \in [c] \land t_2 \in [c]$.

E-graphs provide means to extract the cheapest term from an e-class according to some cost function: $\text{extract}_{\text{cost}}(a) = \arg\min_{t \in [M(a)]} \text{cost}(t)$. If cost is local, meaning that the cost of a term can be computed from the costs of its immediate children, extraction can be done efficiently by a greedy algorithm, which recursively extracts the best term from each e-class.

E-Matching. E-matching is a generalization of pattern matching to e-graphs, where matching an e-class $c$ against a pattern $p$ yields a set of e-class substitutions $\theta : X \rightarrow I$ such that $\theta(p)$ is a “subgraph” of $c$. To formalize this notion, we introduce partial terms $\pi \in T(\Sigma, I)$, which are terms whose leaves can be e-class ids (or, alternatively, patterns with e-class ids for variables). The containment relation $\pi < a$ for some e-class id $a$ is defined as follows:

$$a < a \quad s(\pi_1, \ldots, \pi_k) < a \iff s(a_1, \ldots, a_k) \in M(a) \land \pi_i < a_i$$

With this definition, a pattern $p$ matches an e-class $a$, $a \subseteq p$, if there exists an e-class substitution $\theta$, such that $\theta(p) < a$. We denote the set of such substitutions as $\text{matches}(a, p)$.

Rewriting and Equality Saturation. Equality saturation (EqSat) [Tate et al. 2009; Willsey et al. 2021] takes as input a term $t$ and a set of equations that induce an equivalence relation $\equiv$, and produces an e-graph $G$ such that $[G] = \{t' \mid t \equiv t\}$ and $t \equiv^G t'$ iff $t \equiv G t'$. The core idea of EqSat is to convert equations into rewrite rules and apply them to the e-graph in a non-destructive way: so that the original term and the rewritten terms are both represented in the same e-class. Applying a rewrite rule $p_1 \Rightarrow p_2$ to an e-class $a$ works as follows: for each $\theta \in \text{matches}(a, p_1)$, we obtain the

$$\theta(p_1) \Rightarrow \theta(p_2)$$

\[5\]In a real e-graph implementation, the definitions of e-graphs and the congruence invariant are more involved, because efficient merging of e-classes requires introducing a non-trivial equivalence relation over e-class ids; these details are irrelevant for our purposes. Also, other formalizations of e-graphs do not feature a distinguished root e-class.

rewritten partial term $\pi' = \theta(p_2)$ and then add this partial term to the same e-class $a$, restoring the congruence invariant (i.e., merging e-classes that now have identical e-nodes).

4.1 Top-Level Algorithm

We can formalize the problem of library learning modulo equational theory (LLMT) as follows: given a term $t$ and a set of equations that induce an equivalence relation $\equiv$, the goal is to find a compressed term $\hat{t} \in \hat{T} (\Sigma, X)$, such that $\hat{t} \rightarrow^\beta t'$ for some $t' \in \llbracket G \rrbracket$. Similarly to Sec. 3, we take a pattern-based approach to this problem, that is, we select a set $P$ of candidate patterns and then perform compression rewrites using these patterns.

The rest of the algorithm is split into two phases: candidate generation and candidate selection. Candidate generation (lines 8–10) first computes the set of candidate patterns $P$ using the anti-unification mechanism extended to e-graphs. Then it creates a compression rule ($\kappa$-rule, see Sec. 3) from each candidate pattern, and once again applies EqSat to obtain a new e-graph $G'$. This new e-graph represents all possible ways to compress the terms from $G$ using patterns in $P$. Finally, the candidate selection phase (lines 13–15) selects the optimal subset of compression rules $R'$, constructs an e-graph $G''$ that represents all possible compressions using only the selected compression rules, and finally extracts the smallest compressed term from this e-graph.

The rest of this section focuses on the candidate generation via e-graph anti-unification (line 8). The candidate selection functions select_library and extract are discussed in Sec. 5.

4.2 Candidate Generation via E-Graph Anti-unification

The goal of candidate generation is to find a set of patterns that are useful for compression. Following the discussion in Sec. 3, we restrict our attention to patterns $\text{AU}(t_1, t_2)$, where $t_1$ and $t_2$ are subterms
E-node Anti-unification $\text{AU}(\Gamma \vdash a, b)$
\[
\text{AU}(\Gamma, (a, b) \vdash s(a_1, \ldots, a_k), s(b_1, \ldots, b_k)) = \{s(p_1, \ldots, p_k) \mid p_i \in \text{AU}(\Gamma \vdash a_i, b_i)\}
\]
\[
\text{AU}(\Gamma, (a, b) \vdash s_1(\ldots), s_2(\ldots)) = \{X_{a,b}\} \quad \text{if } s_1 \neq s_2
\]

E-class Anti-unification $\text{AU}(\Gamma \vdash a, b)$
\[
\text{AU}(\Gamma \vdash a, b) = \emptyset \quad \text{if } (a, b) \in \Gamma
\]
\[
\text{AU}(\Gamma \vdash a, b) = \text{dominant}(\bigcup_{n_a \in M(a), n_b \in M(b)} \text{AU}(\Gamma, (a, b) \vdash n_a, n_b))
\]

Fig. 10. E-class anti-unification defined as two mutually-recursive functions of e-nodes and e-class ids.

of some $t \in \llbracket G \rrbracket$. The naïve approach is to enumerate all such $t$, and for each one, perform anti-unification on all pairs of subterms; this is suboptimal at best, and impossible at worst (when $\llbracket G \rrbracket$ is infinite). Hence in this section we show how to compute a finite set of candidate patterns directly on the e-graph, without discarding any optimal patterns.

**E-Class Anti-unification.** Let us first consider anti-unification of two e-classes, $\text{AU}(a, b)$, which takes as input e-class ids $a$ and $b$ and returns a set of patterns. We define $\text{AU}(a, b) = \text{AU}(e \mapsto a, b)$, where $\text{AU}(e \mapsto a, b)$ is an auxiliary function that additionally takes into account a context $\Gamma$. A context is a list of pairs of e-classes that have been visited while computing the AU, and is required to prevent infinite recursion in case of cycles in the e-graph.

Fig. 10 defines this operation using two mutually recursive functions that anti-unify e-classes and e-nodes. Note that e-node AU is always invoked in a non-empty context. The first equation anti-unifies two e-nodes with the same constructor: in this case, we recursively anti-unify their child e-classes and return the cross-product of the results. The second equation applies to e-nodes with different constructors: as in term AU, this results in a pattern variable. A nice side-effect of dealing with e-graphs is that we need not keep track of the anti-substitution to guarantee that each pair of subterms maps to the same variable: because any duplicate terms are represented by the same e-class, we can simply use the e-class ids $a$ and $b$ in the name of the pattern variable $X_{a,b}$.

The second block of equations defines anti-unification of e-classes (let us first ignore the dominant which will be explained shortly). The first equation applies when $a$ and $b$ have already been visited: in this case, we break the cycle and return the empty set. Otherwise, the last equation anti-unifies all pairs of e-nodes from the two e-classes in an updated context and merges the results. Note that this will add a pattern variable unless all e-nodes in both e-classes have the same constructor; we have omitted this detail in Sec. 2 for simplicity, but this is implemented in BABBLE and often yields more optimal patterns. For example, consider anti-unifying $c_1$ and $c_2$ from Fig. 4 (right): although they have the constructor scale in common, $X$ is actually a better pattern for abstracting these two e-classes than scale $XY$, because the total size of the actual arguments to pattern $X$ (side$(6)$ and scale$(\text{repRot}(\text{side}(8), 8, 2\pi/8), 2)$) is the same as those to the pattern scale $XY$ (side$(6)$, 1, repRot$(\text{side}(8), 8, 2\pi/8)$, and 2), and the pattern $X$ itself is smaller. This happens because the class $c_1$ represents several different terms, and the term “compatible with” $X$ in this case is smaller than the term “compatible with” scale $XY$.

**Dominant Patterns.** Recall that $\text{AU}(a, b)$ produces patterns with variable names $X_{a_i,b_i}$, which record the e-class ids they abstract; let us refer to such a pattern $p$ as uniquely matched and define $\theta_i(p) = \{X_{a_i,b_i} \mapsto a_i\}$ and $\theta_j(p) = \{X_{a_i,b_i} \mapsto b_i\}$; these substitutions are necessarily among matches($a, p$) and matches($b, p$), respectively. Given two uniquely matched patterns $p_1, p_2 \in \text{AU}(a, b)$, we say that $p_1$ dominates $p_2$ in the context of $(a, b)$ if (1) $\text{vars}(p_1) \subseteq \text{vars}(p_2)$, and (2) $\text{size}(p_1) \leq \text{size}(p_2)$. We can show that if $p_1$ dominates $p_2$, then we can safely discard $p_2$ from the set of candidate patterns. First, since $p_1$ is no larger than $p_2$, the definition of its $\lambda$-abstraction is also
no larger. Second, given a term $t_a \in [a]$, compressing this term using $p_1$ vs $p_2$, requires choosing actual arguments from $\text{range}(\theta_1(p_1))$ vs $\text{range}(\theta_2(p_2))$: because the former is a subset of the latter, the first application can always be made no larger (symmetric argument applies for $t_b \in [b]$).

Hence it is sufficient that $\text{AU}(a, b)$ only returns the set of dominant patterns (i.e. a pattern dominated by any pattern in the set can be discarded). This is what the function dominant does in the last equation of Fig. 10. This pruning technique is especially helpful in the presence of equational theories. Suppose our theory contains the equation $X + Y \equiv Y + X$, and that the original term $t$ contains subterms $1 + 2$ and $3 + 1$. After saturation, the e-graph will represent $1 + 2$ and $2 + 1$ in some e-class $a$ and $3 + 1$ and $1 + 3$ in another e-class $b$; $\text{AU}(a, b)$ will then produce both patterns $X + 1$ and $1 + X$, but it is clearly redundant to have both, since they match the same e-classes with the same substitutions $\theta$. Pruning of dominated patterns will eliminate one of them.

**Avoiding Cycles.** Interestingly enough, the same notion of dominant patterns justifies why we need not follow cycles in the e-graph when computing $\text{AU}(a, b)$, or, alternatively, why a finite set of candidate patterns is sufficient to compress any term in $[G]$, even if this set is infinite. Removing the first equation that short-circuits cycles can only lead to solutions $p'$ of the form

$$p' = [X \mapsto p](q)$$

where $p \in \text{AU}(a, b)$ is another solution, and $q$ is some context, added by the cycle. It is clear that any such $p'$ is dominated by $p$, and hence can be discarded.

**E-Graph Anti-unification.** The algorithm $\text{AU}(a, b)$ computes a set of patterns that can be used to compress terms represented by the e-classes $a$ and $b$. Our ultimate goal, however, is to compute candidate patterns for abstracting all subterms of some $t \in [G]$. The most straightforward way to achieve this is to apply $\text{AU}(a, b)$ to all pairs of e-classes in $G$. We can do better, however: some pairs of e-classes need not be considered, because they cannot occur together in a single term $t$. For an example, consider the following e-graph, with $\mathcal{I} = \mathbb{N}$ and $r = 0$:

$$0 \mapsto \{f(1), g(2)\} \quad 1 \mapsto \{g(3)\} \quad 2 \mapsto \{f(3)\} \quad 3 \mapsto \{a\}$$

This e-graph can result, for example, by rewriting a term $f(g(a))$ using an equation $f(g(X)) \equiv g(f(X))$. In this e-graph, the e-classes 1 and 2 (representing $g(a)$ and $f(a)$, respectively) clearly cannot co-occur in the same term: since the e-graph only represents two terms, $f(g(a))$ and $g(f(a))$.

To formalize this intuition, we define the co-occurrence relation between two e-class ids as follows. An e-class $a$ is a sibling of $b$ if there is an e-node that has both $a$ and $b$ as children. An e-class $a$ is an ancestor of $b$ if $a = b$ or $b$ is a child of some e-node $c \in M(c)$ and $c$ is an ancestor of $a$; $a$ is a proper ancestor of $b$ if $a$ is an ancestor of $b$ and $a \neq b$. Two e-classes $a$ and $b$ are co-occurring if (1) one of them is a proper ancestor of another, or (2) they have ancestors that are siblings.

Finally, to compute the set $\text{AU}(G)$ of all candidate patterns for an e-graph $G$, LLMT first computes the co-occurrence relation between all e-classes in $G$, and then computes $\text{AU}(a, b)$ of all pairs of e-classes that are co-occurring. As mentioned in Sec. 2, we use a dynamic programming algorithm that memoizes the results of $\text{AU}(a, b)$ to avoid recomputation.

## 5 CANDIDATE SELECTION VIA E-GRAPH EXTRACTION

After generating candidate abstractions, the LLMT algorithm invokes select_library to pick the subset of candidate patterns that can best be used to compress the input corpus. This section describes our approach to selecting the optimal library dubbed targeted common subexpression elimination.

**Library Selection as E-Graph Extraction.** Recall that candidate selection starts with an e-graph $G'$, which represents all the ways of compressing the initial corpus and its equivalent terms using the candidate patterns $P$. We will refer to a subset $\mathcal{L} \subset P$ as a library. The optimal size of an e-class $c$ compressed with $\mathcal{L}$ can be computed as the sum of the sizes of (1) the smallest term $t \in [c]$ using
**E-node cost set** \( \text{costset}_N(n) \)

\[
\text{costset}_N(s()) = \{ (\emptyset, 1) \}
\]

\[
\text{costset}_N(s(\overline{a})) = \{ (L, u + 1) \mid (L, u) \in \text{cross}(\overline{a}) \}
\]

\[
\text{costset}_N((\lambda \overline{x} \rightarrow a) b) = \text{addlib}(a, \text{costset}(a), \text{costset}(b))
\]

**E-class cost set** \( \text{costset}(a) \)

\[
\text{costset}(\overline{a}) = \text{prune}(\text{reduce}(\bigcup \text{costset}_N(n_j)))
\]

**Auxiliary definitions** cross, addlib, reduce, prune

\[
cross(z_1, z_2) = \text{prune}(\text{reduce}(\{ (L_1 \cup L_2, u_1 + u_2) \mid (L_1, u_1) \in z_1, (L_2, u_2) \in z_2 \}))
\]

\[
\text{addlib}(a, z_1, z_2) = \text{prune}(\text{reduce}(\{ (L_1 \cup L_2 \cup \{a\}, u_2) \mid (L_1, u_1) \in z_1, (L_2, u_2) \in z_2 \}))
\]

\[
\text{reduce}(z) = \{ (L_1, u_1) \in z \mid \forall (L_2, u_2) \in z \setminus (L_1, u_1). L_1 \subset L_2 \lor u_1 < u_2 \}
\]

\[
\text{prune}(N, K, z) = \text{top}_k(\{ (L, u) \in z \mid |L| \leq N \}, K)
\]

**Fig. 11.** Cost set propagation, defined as two mutually recursive functions. Blue text corresponds to the partial order reduction optimization and red text corresponds to the beam approximation. \( \text{top}_k(S, K) \) is a helper function that returns top \( K \) elements from the sorted set \( S \).

only the library functions in \( L \) and where \( \lambda \)-abstractions do not count toward the size of \( t \), and (2) the smallest version of each \( p \in L \). Note that the cost of defining an abstraction in \( L \) is only counted once, and that abstractions in \( L \) can be used to compress other abstractions in \( L \). Our goal is to find \( L \) such that the root e-class compressed with \( L \) has the smallest size.

Given a *particular* library, we can find the size of the smallest term via a relatively straightforward top-down traversal of the e-graph. Hence, a naïve approach to library selection would be to enumerate all subsets of \( \mathcal{P} \) and pick the one that produces the smallest term at the root. Unfortunately, this approach becomes intractable as the size of \( \mathcal{P} \) grows.

**Exploiting Partial Shared Structure.** Instead, BABBLE selects the optimal library using a bottom-up dynamic programming algorithm. To this end, it associates each e-node and e-class with a cost set, which is a set of pairs \( (L, u) \) where \( L \) is a library and \( u \) is the use cost of this library, i.e. the size of the smallest term represented by the e-node / e-class if it is allowed to use \( L \) (excluding the size of \( L \) itself). Cost sets are propagated up the e-graph using the rules shown in Fig. 11. The base case is a nullary e-node \( s() \), which cannot use any library functions and whose size is always 1. For an e-node that has children, BABBLE computes the cross product over the cost sets of all its child e-classes. Finally, for an application e-node \( (\lambda \overline{x} \rightarrow a) b \), the cost set must include the library function \( a \) (in addition to some combination of libraries from the cost sets of \( a \) and \( b \)); note that the use cost of the abstraction node only includes the use cost of \( b \), since abstraction bodies are excluded from the use cost.

To compute the cost set of an e-class, BABBLE takes the union of the cost sets of all its e-nodes. However, doing this naively would result in the size of the cost set growing exponentially. To mitigate this, we define a partial order reduction (reduce), which only prunes provably sub-optimal cost sets. Given two pairs \( (L_1, u_1) \) and \( (L_2, u_2) \) in the cost set of an e-class, if \( L_1 \subset L_2 \) and \( u_1 \leq u_2 \), then \( L_2 \) is subsumed by \( L_1 \) and can be removed from the cost set, intuitively because \( L_1 \) can compress this e-class even better and with fewer library functions. In practice this optimization prunes libraries with redundant abstraction, where two different abstractions can be used to compress the same subterms.

**Beam Approximation.** Even with the partial order reduction, calculating the cost set for every e-node and e-class can blow up exponentially. To mitigate this, BABBLE provides the option to limit
We selected our benchmark domains from two previous works: DreamCoder [Ellis et al. 2021] and 2D CAD [Wong et al. 2022]. Each domain from DreamCoder has multiple benchmarks; 2D CAD has one large benchmark per domain. For some domains, we additionally supplied Babble with an equational theory; we report the number of equations in the final column.

### Table 1

<table>
<thead>
<tr>
<th>Domain</th>
<th># Benchmarks</th>
<th># Eqs</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>59</td>
<td>14</td>
</tr>
<tr>
<td>Physics</td>
<td>18</td>
<td>8</td>
</tr>
<tr>
<td>Text</td>
<td>66</td>
<td>-</td>
</tr>
<tr>
<td>Logo</td>
<td>12</td>
<td>-</td>
</tr>
<tr>
<td>Towers</td>
<td>18</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Domain</th>
<th># Benchmarks</th>
<th># Eqs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuts &amp; Bolts</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>Vehicles</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>Gadgets</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>Furniture</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>

both the size of each library inside a cost set and the size of the cost set stored for each e-class. This results in a beam-search style algorithm, where the cost set of each e-class is pruned, as shown in Fig. 11. This pruning operation first filters out libraries that have more than \( N \) patterns, then ranks the rest by the total cost (i.e. the sum of use cost and the size of the library), and finally returns the top \( K \) libraries from that set.

### Evaluation

We evaluated Babble and the LLMT algorithm behind it with two quantitative research questions and a third qualitative one:

- **RQ 1.** Can Babble compress programs better than a state-of-the-art library learning tool?
- **RQ 2.** Are the main techniques in LLMT (anti-unification and equational theories) important to the algorithm’s performance?
- **RQ 3.** Do the functions Babble learns make intuitive sense?

### Benchmark Selection

We use two suites of benchmarks to evaluate Babble, both shown in Tab. 1. The first suite originates from the DreamCoder work [Ellis et al. 2021], and is available as a public repository [Bowers 2022]. DreamCoder is the current state-of-the-art library learning tool, and using these benchmarks allows us to perform a head-to-head comparison. The DreamCoder benchmarks are split into five domains (each with a different DSL); we selected two of the domains—List and Physics—which we understood best, to add an equational theory.

The second benchmark suite, called 2D CAD, comes from Wong et al. [2022]. This work collects a large suite of programs in a graphics DSL for the purpose of studying connections between the generated objects and their natural language descriptions. There are 1,000 programs in the “Drawings” portion of this dataset, divided into four subdomains (listed in Tab. 1) of 250 programs each.

We ran Babble on all benchmarks on an AMD EPYC 7702P processor at 2.0 GHz. Each benchmark was run on a single core. The DreamCoder results were taken from the benchmark repository [Bowers 2022]; DreamCoder was run on 8 cores of an AMD EPYC 7302 processor at 3.0 GHz.

### 6.1 Comparison with DreamCoder

To answer **RQ 1**, we compare to the state-of-the-art DreamCoder tool [Ellis et al. 2021] on its own benchmarks. The DreamCoder benchmarks are suited to its workflow; while the input to a library learning task is conceptually a set of programs (or just one big program), each input to DreamCoder is a set of groups of programs. Each group is a set of programs that are all output from the same program synthesis task (from an earlier part of the DreamCoder pipeline). When compressing a program via library learning, DreamCoder is minimizing the cost of the program
Fig. 12. **Babble** consistently achieves better compression ratios than **DreamCoder** on benchmarks from the **DreamCoder** domains, and it does so 1–2 orders of magnitude faster. Each marker shows the compression ratio (x-axis) and run time (y-axis) of a benchmark. Each benchmark is one **DreamCoder** input, i.e., a set of groups of programs as described above. Lower and to the right is better. In the domains where we supplied **Babble** with an equational theory (List and Physics), additional markers show the performance of **Babble** using purely syntactic learning (without equations, “**BabbleSyn**”) or only equality saturation without library learning (“**EqSat**”).
Table 2. Results for running BABBLE on the four large examples from the 2d cad dataset [Wong et al. 2022], both without and with an equational theory. Each row includes the input and output AST sizes, the compression ratio (CR), and BABBLE’s run time in seconds. Fig. 13 plots this data. The final row shows performance on a modified dataset.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input Size</th>
<th>Without Eqs</th>
<th></th>
<th>With Eqs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Out Size</td>
<td>CR</td>
<td>Time (s)</td>
<td>Out Size</td>
</tr>
<tr>
<td>Nuts &amp; Bolts</td>
<td>19009</td>
<td>2059</td>
<td>9.23</td>
<td>18.74</td>
<td>1744</td>
</tr>
<tr>
<td>Vehicles</td>
<td>35427</td>
<td>6477</td>
<td>5.47</td>
<td>79.50</td>
<td>5505</td>
</tr>
<tr>
<td>Gadgets</td>
<td>35713</td>
<td>6798</td>
<td>5.25</td>
<td>75.07</td>
<td>5037</td>
</tr>
<tr>
<td>Furniture</td>
<td>42936</td>
<td>10539</td>
<td>4.07</td>
<td>133.25</td>
<td>9417</td>
</tr>
<tr>
<td>Nuts &amp; Bolts (clean)</td>
<td>18259</td>
<td>2215</td>
<td>8.24</td>
<td>18.12</td>
<td>1744</td>
</tr>
</tbody>
</table>

made by concatenating the most compressed program from each group together, in other words:

\[
\sum_{\text{group } g} \min_{\text{program } p \in g} \text{cost}(p)
\]

DREAMCODER takes this approach to give its library learning component many variants of the same program, in order to introduce more shared structure between solution programs across different synthesis problems.

To implement DREAMCODER’s benchmarks in BABBLE, we use the e-graph to capture the notion of program variants in a group. Since every program in a group is the output of the same synthesis task, BABBLE considers them equivalent and places them in the same e-class.

**Results.** We ran BABBLE on five domains from the DREAMCODER benchmark suite. The results are shown in Fig. 12. In summary, BABBLE consistently achieves better compression ratios than DREAMCODER on benchmarks from the DREAMCODER domains, and it does so 1–2 orders of magnitude faster.

**The Role of Equational Theory.** To answer RQ 2, we again turn to the DREAMCODER benchmarks, focusing on the domains where we supplied BABBLE with an equational theory: List and Physics. In these domains, we ran BABBLE in two additional configurations:

- “BabbleSyn” ignores the equational theory, just doing syntactic library learning.
- “EqSat” just optimizes the program using Equality Saturation with the rewrites from the equational theory. This configuration does not do any library learning.

Fig. 12a and 12b show the results for these additional configurations, as well as DREAMCODER and the normal BABBLE configuration. All BABBLE configurations rely on targeted subexpression elimination to select the final learned library. The “EqSat” configuration is unsurprisingly very fast but performs relatively little compression, as it does not learn any library abstractions. The “BabbleSyn” configuration does indeed compress the inputs, in fact it is still better than DREAMCODER in both domains. However, the addition of the equational theory (the “BABBLE” markers in the plots) significantly improves compression and adds relatively little run time, well within an order of magnitude.

**6.2 Large-Scale 2d cad Benchmarks**

The previous section demonstrated that BABBLE’s performance far surpasses the state of the art. In this section, we present and discuss the results of running BABBLE on benchmarks from the 2d cad domain. These benchmarks, taken from Wong et al. [2022], are significantly larger (roughly 10x - 100x) than those from the DREAMCODER dataset and out of reach for DREAMCODER.
Fig. 13. Data from Tab. 2 in scatter plot. Each line segment shows compression ratio and run time for a domain without (hollow marker) and with (solid marker) an equational theory. Using an equational theory improves compression in all cases, and even improves run time in two cases.

Quantitative Results. Tab. 2 and Fig. 13 show the results of running Babble on the benchmarks from the 2D CAD domain. The plot in Fig. 13 makes two observations clear. First and relevant for RQ 2, the addition of an equational theory improved all four benchmarks (the solid marker is always to the right of the hollow marker). Second, and perhaps surprisingly, equational theories can sometimes make Babble faster! This is consistent with previous observations about equality saturation [Willsey et al. 2021]: while equality saturation typically makes an e-graph larger, it can sometimes combine two relevant e-classes into one and reduce the amount of work that some operation over an e-graph must do.

We also observed that the Nuts-and-bolts dataset contains several redundant transformations, like the “scale by 1” featured in the running example of Sec. 2. These redundancies can be useful for finding abstractions in the absence of an equational theory. However, they should not be required in Babble since LLLM can introduce the redundancies wherever required. We therefore removed all existing redundant transformations from Nuts-and-bolts and ran Babble on the transformed dataset. The results are in the final row of Tab. 2. On the modified dataset, Babble achieves identical compression when using the equational theory, but without the equations it performs worse than on the unmodified dataset.

Qualitative Evaluation. Fig. 14 highlights a sample of abstractions that Babble discovered from the 2D CAD benchmarks. We ran Babble on each of the benchmarks and applied the learned abstractions on a few input programs to visualize their usage. Questions about usability and readability of learned libraries are difficult to answer without rigorous user studies which we leave for future work. Nevertheless, Fig. 14 shows that Babble identifies common structures that are similar across different benchmarks, which makes its output easier to reuse and interpret.

First, we revisit the Nuts-and-bolts example from Sec. 1: Fig. 14 shows that Babble learns the scaled polygon (n-gon) abstraction which is applicable to several programs in the dataset. We also see that Babble consistently finds a similar abstraction representing a “ring of shapes” for both Nuts-and-bolts and Vehicles. Finally, as the Gadgets example shows, Babble finds abstractions for both the entire model as well as its components. In this case, it learned the function gadget_body that abstracts the entire outer shape, and it also learned dial that abstracts the handles of the outer shape.
Fig. 14. A selection of evaluated programs from each of the domains in the 2d cad dataset, along with a selection of functions that BABBLE learns within the first ten rounds for each domain. Bolded functions represent learned abstractions. Note this figure uses the concrete syntax from the 2d cad dataset; it is similar to the simplified form shown in the overview. We named the learned functions and their parameters for clarity.
7 RELATED WORK

**BABBLE** is inspired by work on library learning, specifically the **DreamCoder** line of work, as well as equality saturation-based program synthesis and decompilation.

**DreamCoder.** **DreamCoder** [Ellis et al. 2021] is a program synthesizer that learns a library of abstractions from solutions to a set of related synthesis tasks. The library is intended to be used for solving other similar synthesis tasks. **DreamCoder** uses version spaces [Lau et al. 2001; Mitchell 1977] to compactly store a large set of programs and leverages ideas from e-graphs (such as e-matching) but only for exploring the space of refactorings of the original program using the candidate libraries, not for making library learning robust to syntactic variation. Our evaluation shows that **BABBLE** can find more optimal abstractions faster than **DreamCoder**.

**DreamCoder** has sparked several direction of follow-up work that attempt to improve the efficiency of its library learning procedure and the quality of the learned abstractions. One of them is by Wong et al. [2021], which uses natural language annotations and a neural network to guide library learning. Another one is **Stitch** [Bowers et al. 2023], which was developed concurrently with this work and is the most closely related to **BABBLE**; we discuss **Stitch** in some detail below.

**Stitch.** The core difference between the two approaches is that **Stitch** focuses on improving efficiency of purely syntactic library learning, whereas **BABBLE** attempts to improve its expressiveness by adding equational theories. While **BABBLE** separates library learning into two phases—candidate generation via anti-unification and candidate selection via e-graph extraction—the **Stitch** algorithm *interleaves* the generation and selection phases in a branch-and-bound top-down search. Starting from the “top” pattern $X$, **Stitch** gradually refines it until further refinement does not pay off. To quickly prune suboptimal candidate patterns, **Stitch** computes an upper bound on their compression by summing up the compression at each match of this pattern in the corpus (this bound is imprecise because it does not take into account that matches might overlap). For candidates that are not pruned this way, **Stitch** computes their true compression by searching for the optimal subset of matches to rewrite, a so-called “rewrite strategy”. **BABBLE**’s extraction algorithm can be seen as a generalization of **Stitch**’s rewrite strategy: while the former searches over both subsets of patterns and how to apply them to the corpus at the same time, the latter considers a single pattern at a time and only searches for the best way to apply it. Since the search space in the former case is much larger, **BABBLE** uses a beam search approximation, while in **Stitch** the rewrite strategy is precise. To sum up, the main pros and cons of the two approaches are:

- **BABBLE** can learn libraries modulo equational theories, while **Stitch** cannot;
- **Stitch** provides optimality guarantees for learning a single best abstraction at a time, while **BABBLE** can learn multiple abstractions at once, but sacrifices theoretical optimality.

**Other Library Learning Techniques.** **Knoch** [Dumancic et al. 2021] is a library learning tool for logic programs, which, like **BABBLE**, proceeds in two phases. Their candidate generation phase is similar to the upper bound computation in **Stitch**, while their selection phase uses an off-the-shelf constraint solver. It would be interesting to explore whether their constraint-based technique can be generalized beyond logic programs.

Other work develops limited forms of library learning, where only certain kinds of sub-terms can be abstracted. For example, ShapeMod [Jones et al. 2021] learns macros for 3D shapes represented in a DSL called ShapeAssembly, and only supports abstracting over numeric parameters, like dimensions of shapes. Our own prior work [Wang et al. 2021] extracts common structure from graphical programs, but only supports abstracting over primitive shapes and applying the abstraction at the top level of the program. Such restrictions make the library learning problem more computationally tractable, but limit the expressiveness of the learned abstractions.
There are several neural program synthesis tools [Dechter et al. 2013; Iyer et al. 2019; Lázaro-Gredilla et al. 2018; Shin et al. 2019] that learn programming idioms using statistical techniques. Some of these tools have used “explore-compress” algorithms [Dechter et al. 2013] to iteratively enumerate a set of programs from a grammar and find a solution that exposes abstractions that make the set of programs maximally compressible. This is similar to common subexpression elimination which babble uses for guiding extraction.

Loop rerolling. Loop rerolling is related to library learning in that it also aims to discover hidden structure in a program, except that this structure is in the form of loops. A variety of domains have used loop rerolling to infer abstractions from flat input programs. In hardware, loop rerolling is used to optimize programs for code size [Rocha et al. 2022; Stiff and Vahid 2005; Su et al. 1984]. In many of these tools, the compiler first unrolls a loop, applies optimizations, then rerolls it — the compiler therefore has structural information about the loop that can be used for rerolling [Rocha et al. 2022]. The graphics domain has used loop-rerolling to discover latent structure from low-level representations. CSGNet [Sharma et al. 2017] and Shape2Prog [Tian et al. 2019] used neural program generators to discover for loops from pixel- and voxel-based input representations. [Ellis et al. 2017] used program synthesis and machine learning to infer loops from hand-drawn images. Szalinski [Nandi et al. 2020] used equality saturation to automatically learn loops in the form of maps and folds from flat 3D CAD programs that are synthesized by mesh decompilation tools [Nandi et al. 2018]. WebRobot [Dong et al. 2022] has used speculative rewriting for inferring loops from traces of web interactions. Similar to babble (and unlike Szalinski), WebRobot finds abstractions over multiple input traces.

Applications of Anti-unification. Anti-unification is a well-established technique for discovering common structure in programs. It is the core idea behind bottom-up Inductive Logic Programming [Cropper and Dumancic 2022], and has also been used for software clone detection [Bulychev et al. 2010], programming by example [Raza et al. 2014], and learning repetitive code edits [Meng et al. 2013; Rolim et al. 2017]. It is possible that these applications could also benefit from babble’s notion of anti-unification over e-graphs to make them more robust to semantics-preserving transformations.

Synthesis and Optimization using E-graphs. While traditionally e-graphs have been used in SMT solvers for facilitating communication between different theories, several tools have demonstrated their use for optimization and synthesis. Tate et al. [2009] first used e-graphs for equality saturation: a rewrite-driven technique for optimizing Java programs with loops. Since then, several tools have used equality saturation for finding programs equivalent to, but better than, some input program [Nandi et al. 2020; Panchekha et al. 2015; VanHattum et al. 2021; Wang et al. 2020; Willsey et al. 2021; Wu et al. 2019; Yang et al. 2021]. babble uses an anti-unification algorithm on e-graphs (together with domain specific rewrites), which prior work has not shown. Additionally, prior work has either used greedy or ILP-based extraction strategies, whereas babble uses a new targeted common subexpression elimination approach which we believe can be used in many other applications of equality saturation, especially given its amenability to approximation via beam search.

8 CONCLUSION AND FUTURE WORK

We presented library learning modulo theory (LLMT), a technique for learning abstractions from a corpus of programs modulo a user-provided equational theory. We implemented LLMT in babble. Our evaluation showed that babble achieves better compression orders of magnitude faster than the state of the art. On a larger benchmark suite of 2D CAD programs, babble learns sensible functions that compress a dataset that was—until now—too large for library learning techniques.
LLMT and BABBLE present many avenues for future work. First, our evaluation showed that equational theories are important for achieving high compression, but these must be provided by domain experts. Recent work in automated theory synthesis like Ruler [Nandi et al. 2021] or TheSy [Singher and Itzhaky 2021] could aid the user in this task. Second, LLMT uses e-graph anti-unification to generate promising abstraction candidates, but this approach is incomplete and misses some patterns that could achieve better compression. An exciting direction for future work is to combine LLMT with more efficient top-down search from STITCH [Bowers et al. 2023]. This is challenging because STITCH crucially relies on the ability to quickly compute an upper bound on the compression of a given pattern by summing up the local compression at each of its matches in the corpus. This upper bound does not straightforwardly extend to e-graphs because in an e-graph different matches of a pattern may come from different syntactic variants of the corpus, and one needs to trade-off the compression from abstractions against the size difference between different syntactic variants.

DATA AVAILABILITY STATEMENT
The data that support the findings of this study are openly available on Zenodo [Cao et al. 2022].

ACKNOWLEDGMENTS
We are grateful to the anonymous reviewers for their insightful comments. We would like to thank Matthew Bowers for many helpful discussions and especially for publishing the DREAMCODER compression benchmark: we know it was a lot of work to assemble! This work has been supported by the National Science Foundation under Grants No. 1911149 and 1943623.

REFERENCES
Canonical source is on Github: https://github.com/dcao/babble/blob/popl23/POPL23.md


Received 2022-07-07; accepted 2022-11-07