babble: Library learning with e-graphs

DAVID CAO, University of California, San Diego, USA

Library learning is a general technique for extracting out common auxiliary functions present across a set of input programs. Because of its generality, library learning has found use across a variety of domains, including in augmented program synthesis [Ellis et al. 2014] and graphical structure analysis. However, existing library learning solutions require a trade-off between exhaustiveness (being able to find more hidden common auxiliary functions, but at extremely high compute cost) and speed (being able to find trivial common functions quickly, but being unable to find less trivial common subroutines). Our work uses *egg* [Willsey et al. 2021], an efficient implementation of e-graphs in Rust, to create a more efficient library learning approach targeted towards a simple graphical DSL. We evaluate the effectiveness of *babble*, a prototype which implements this approach, and evaluate its effectiveness on a set of simple library learning examples.

1 INTRODUCTION

When programming, most programmers don't implement their programs as one massive function with every definition inlined; instead, programmers often extract out common functionality as auxiliary functions, allowing for the composition of more complex behavior. *Library learning* is the line of research which seeks to automate this process. Library learning algorithms take a set of programs as input, and synthesizes a set of common auxiliary functions across these programs (as long as the original programs re-expressed in terms of the auxiliary functions) as output. While use cases for this line of research are still emerging, several key use cases have already emerged, particularly in the synthesis domain.

One key use case for library learning is augmenting existing program synthesis techniques with the ability to generate new component functions during the synthesis process. Traditionally, many synthesis systems require the user to provide a set of components as a structural constraint input, with the synthesis algorithm using this set of components in order to generate a solution program. However, library learning allows for synthesis systems to learn new components based on previous synthesis results, without the need for user intervention. DreamCoder [Ellis et al. 2014], for instance, utilizes a two-stage synthesis loop: a neural-network-based program synthesis "wake" stage, and a library learning "sleep" stage, which uses an algorithm based on version-space algebras that learns new auxiliary functions based on complete user-provided programs, along with previously synthesized programs. This two-stage program synthesis process allows DreamCoder to synthesize complex auxiliary functions like map and fold, and use these in future synthesis solutions.

Another key use case of library learning is to uncover some hidden common structure between seemingly disparate programs. This is particularly important for graphical applications: for instance, a work from Wang et al. [Wang et al. 2021] (here onwards referred to as the "smiley paper") utilizes anti-unification techniques on a simple, specialized graphical DSL in order to compare how programmatic library learning and anti-unification techniques can parallel how human beings recognize abstract structures like smiley faces.

However, existing library learning approaches often require choosing between exhaustiveness and speed. For instance, because of its hybrid "wake-sleep" synthesis loop, DreamCoder is capable of discovering highly complex hidden structures between different programs. However, as a tradeoff,

Author's address: David Cao, University of California, San Diego, USA, dmcao@ucsd.edu.

it requires a potentially-impractical amount of compute power and time. On the other hand, approaches similar to the smiley paper don't suffer from issues with speed, but with exhaustiveness in their search; in particular, the approach used by the smiley paper only allows for anti-unification on the shape used. The ideal library learning system would be both exhaustive and fast, allowing for a reasonably thorough search of the program space in a reasonable amount of time.

This article presents a new technique for library learning which relies on modifying equivalence graphs—data structures which represent a set of equivalent programs—by progressively rewriting programs as applications to common functions. To limit the scope of the project, our approach focuses on basic library learning between two programs written in the graphics DSL from the smiley paper, in an effort to improve on its anti-unification techniques. Our prototype implementation of this approach, named *babble*, uses *egg* [Willsey et al. 2021], an efficient library for equivalence graphs in Rust, to facilitate performant library learning.

2 MOTIVATING EXAMPLE: RECOGNIZING SMILEYS

As mentioned earlier, for the sake of this project, we limit our scope to targeting the simple graphics DSL from the smiley paper and improving on its anti-unification capabilities. Presented in figure 1 is an example of such a problem.

```
(let cyanFace (set
  (move −2 2 (scale 0.1 line))
  (move 2 2 (scale 0.5 circle))
  (move 0 −2 line))
  (let greenFace (set
    (move −2 2 (scale 0.1 line))
    (move 2 2 (scale 0.1 line))
    (move 0 −2 line))
      (set cyanFace greenFace)))
```

(a) Two faces in the graphical DSL

(b) The graphical representation of the faces

Fig. 1. An example anti-unification problem

Both of the drawings are smiley faces, so the goal is to extract out a common function which represents a smiley face and use that function for both the cyan and green faces. In particular, we want to anti-unify these two expressions such that the common function parameterizes over the size and shape of the right eye, as shown in figure 2. Note that an example like this would have been impossible to anti-unify with the approach from the smiley face paper, as it is only capable of parameterizing over the shapes used in each drawing, and would require all of the translation and scaling parameters to match.

```
(let f (λz. λs. (set (move −2 2 (scale 0.1 line)) (move 2 2 (scale z s)) (move 0 −2 line)))
  (let cyanFace (f 0.5 circle)
    (let greenFace (f 0.1 line)
      (set cyanFace greenFace))))
```

Fig. 2. Faces in the DSL, with an anti-unified function for constructing smileys

Thus, the primary challenge of this work is devising a strategy for rewriting syntactically or structurally different programs such that they can be anti-unified.

3 LIBRARY LEARNING WITH E-GRAPH REWRITES

To facilitate this, our approach uses an *equivalence graph* and rewrites subexpressions within this equivalence graph to introduce and move functions. In practice, babble uses the egg library to do this in a performant manner. When given an input program, babble applies a set of rewrite rules to the program which introduces new functions, preserving program behavior, and rotates them up and down the AST (these rewrites can be thought of as "inverse" beta reductions). Like the DreamCoder paper, babble currently introduces these new functions indiscriminately, with no heuristics or guided search to introduce functions specifically amenable to anti-unification. babble also contains rewrite rules which extracts out a common function into its own let-binding if it is being applied in both programs. Then, after the e-graph has hit a maximum size or time limit, babble selects the expression in the e-graph which uses the fewest number of primitives (e.g. scale, move, circle, line, etc.)

To show how this technique would work in practice, we provide an abridged demonstration of how babble would learn a common face function from the example in figure 1. First, babble can replace any subexpression with an application to the identity function, using the corresponding subexpressions as the argument; this rewrite allows for the introduction of functions which parameterize over an arbitrary part of the expression. Introducing these functions yields the following:

```
(let cyanFace (set (move −2 2 (scale 0.1 line)) (move 2 2 (scale ((λz. z) 0.5) ((λs. s) circle))) (move 0 −2 line))
  (let greenFace (set (move −2 2 (scale 0.1 line)) (move 2 2 (scale ((λz. z) 0.1) ((λs. s) line))) (move 0 −2 line))
    (set cyanFace greenFace)))
```

Additionally, babble also contains rewrites which can rotate these functions outwards, allowing babble to move the function abstraction to the top level of each drawing expression. Rotating these functions upwards yields the following:

```
(let cyanFace ((λz. ((λs. (set (move −2 2 (scale 0.1 line)) (move 2 2 (scale z s)) (move 0 −2 line))) circle)) 0.5)
  (let greenFace ((λz. ((λs. (set (move −2 2 (scale 0.1 line)) (move 2 2 (scale z s)) (move 0 −2 line))) line)) 0.1)
    (set cyanFace greenFace)))
```

babble can then rotate the function abstractions and abstractions such that they are adjacent with each other, like so:

```
(let cyanFace ((λz. λs. (set (move −2 2 (scale 0.1 line)) (move 2 2 (scale z s)) (move 0 −2 line))) 0.5 circle)
  (let greenFace ((λz. λs. (set (move −2 2 (scale 0.1 line)) (move 2 2 (scale z s)) (move 0 −2 line))) 0.1 line)
    (set cyanFace greenFace)))
```

4 EVALUATION

In order to evaluate the effectiveness of babble, we tested several simple anti-unification tasks, along with a task taken from the smiley paper, and compared the output of babble to the desired anti-unified output. For completeness, these benchmarks and the desired/actual outputs are reproduced in appendix A.

For the most part, babble was able to produce correct solutions for simple examples relatively efficiently, but quickly stumbles for more complex inputs. We also observed unexpected behavior in some cases. First, the speed of these results can be misleading, as egg has a time limit on how

Benchmark	Correct output	Time (s)
Shape parameter	✓	0.113
Rotate parameter	✓	0.128
Shape & rotate parameters	✓	0.119
Two composed, scale & xy translate parameters	✓	0.116
Two composed, scale & x translate parameters	✓	0.125
Wang et al. example	✗	0.116

Fig. 3. A table of benchmark results.

long an e-graph will be explored for; by default, this limit is set at around a tenth of a second; since babble can rewrite any subexpression to introduce a function an indefinite number of times, the e-graph will never reach saturation, and the time taken will always be equal to the time limit given to the tool. However, when testing the final benchmark, increasing the time limit to as much as 30 seconds had no effect on the correctness of the output. Finally, during testing, we found that manually introducing and rotating some functions in the input (i.e. introducing "manual rewrites" to the input) allowed previously failing test cases to pass. This seems to indicate that although the concept is sound, it is also fragile, and the success of the process is highly dependent on how rewrites are scheduled and how the program search space is explored.

5 FUTURE WORK

As demonstrated, although the proof-of-concept works for relatively simple examples, it struggles with expressions of larger complexity. Additionally, even in the cases when the tool returns the desired results, the tool currently suffers from several other key limitations:

- Only two different programs can be anti-unified.
- Only one common auxiliary function can be extracted between these two programs.
- Dealing with variable capture is currently unimplemented, which limits the complexity of the generated auxiliary functions.

However, despite these limitations, the performance of the tool on the simple benchmarks above seems to indicate that the proof-of-concept is sound. There are also several other improvements beyond addressing these limitations which can make the system more robust for complex examples:

More guided search. At the moment, babble uses a naive rewrite rule scheduling strategy provided by the egg library which, on every iteration, simply fires every rule once for each subexpression it matches. Rather than this naive approach, we propose adding a more intelligent rewrite scheduler that schedules rewrites such that the output is more conducive to anti-unification.

More domain-specific rewrites. Because this tool targets a graphical DSL, it can take advantage of special properties like symmetries. For instance, a circle can be rotated any amount, and it will still be the same circle. The same can be said for a line segment, which looks the same as when it is rotated 180 degrees. Thus, we propose new rewrite rules which can take these symmetries into account and reveal further anti-unification opportunities.

Combined, these additions to babble could make it a performant and capable library learning solution.

A BENCHMARK PROGRAMS

A.1 Shape parameter

```
(let s1 (rotate 50 (move 2 4 (scale 3 circle)))
  (let s2 (rotate 50 (move 2 4 (scale 3 line)))
    (set s1 s2)))
```

(a) Input

```
(let f (λc. (rotate 50 (move 2 4 (scale 3 c))))
  (let s1 (f circle)
    (let s2 (f line)
      (set s1 s2))))
```

(b) Expected and actual output

A.2 Rotate parameter

```
(let s1 (rotate 50 (move 2 4 (scale 3 line)))
  (let s2 (rotate 100 (move 2 4 (scale 3 line)))
    (set s1 s2)))
```

(a) Input

```
(let f (λr. (rotate r (move 2 4 (scale 3 line))))
  (let s1 (f 50)
    (let s2 (f 100)
      (set s1 s2))))
```

(b) Expected and actual output

A.3 Shape & rotate parameter

```
(let s1 (rotate 50 (move 2 4 (scale 3 line)))
  (let s2 (rotate 100 (move 2 4 (scale 3 circle)))
    (set s1 s2)))
```

(a) Input

```
(let f (λs r. (rotate r (move 2 4 (scale 3 s))))
  (let s1 (f line 50)
    (let s2 (f circle 100)
      (set s1 s2))))
```

(b) Expected and actual output

A.4 Two composed, scale & xy translate parameters

```
(let s1 (set (scale 9 circle) (move 3 3 (scale 2 circle)))
  (let s2 (set (scale 3 circle) (move 1 1 (scale 2 circle)))
    (set s1 s2)))
```

(a) Input

```
-- The order of the two circles is reversed in
-- the output. This has no effect on evaluation
(let f (λxy. λz. (set
  (move xy xy (scale 2 circle)) (scale z circle)))
  (let s1 (f 9 3)
    (let s2 (f 3 1)
      (set s1 s2))))
```

(b) Expected and actual output

A.5 Two composed, scale & x translate parameters

```
(let s1 (set (scale 6 circle) (move 0 0 line))
  (let s2 (set (scale 4 circle) (move 2 0 line))
    (set s1 s2)))
```

(a) Input

```
(let f (λz. λx. (set (move x 0 line) (scale z circle)))
  (let s1 (f 6 0)
    (let s2 (f 4 2)
      (set s1 s2)))
```

(b) Expected and actual output

A.6 Wang et al. example

```
(let s1 (set
  (move 4 4 (scale 2 line))
  (move 3 2 line)
  (move 4 3 (scale 9 circle))
  (move 5 2 line))
  (let s2 (set
    (move 4 4 (scale 2 circle))
    (move 3 2 circle)
    (move 4 3 (scale 9 circle))
    (move 5 2 circle))
      (set s1 s2)))
```

(a) Input & actual output

```
-- The order of the two circles is reversed in
-- the output. This has no effect on evaluation
(let f (λs. (set
  (move 4 4 (scale 2 s))
  (move 3 2 s)
  (move 4 3 (scale 9 circle))
  (move 5 2 s)))
  (let s1 (f line)
    (let s2 (f circle)
      (set s1 s2)))
```

(b) Expected output

REFERENCES

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. 2014. DreamCoder: Building interpretable hierarchical knowledge representations with wake-sleep Bayesian program learning. (2014), 68.

Haoliang Wang, Nadia Polikarpova, and Judith Fan. 2021. Learning Part-Based Abstractions for Visual Object Concepts. *Under submission* (2021).

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 1–29. https://doi.org/10.1145/3434304