Cozy: Synthesizing Collection Data Structures

Calvin Loncaric
University of Washington, USA
loncaric@cs.washington.edu

ABSTRACT

Many applications require specialized data structures not found in standard libraries. Implementing new data structures by hand is tedious and error-prone. To alleviate this difficulty, we built a tool called Cozy that synthesizes data structures using counter-example guided inductive synthesis. We evaluate Cozy by showing how its synthesized implementations compare to handwritten implementations in terms of correctness and performance across four real-world programs. Cozy's data structures match the performance of the handwritten implementations while avoiding human error.

CCS Concepts

•**Software and its engineering** → **Automatic programming;** •**Theory of computation** → *Data structures design and analysis;*

Keywords

Data structure synthesis

1. INTRODUCTION

All mainstream languages ship with libraries implementing lists, maps, sets, trees, and other common data structures. These libraries are sufficient for some use cases, but other applications need specialized data structures with different operations. For such applications, the standard libraries are not enough.

To alleviate the need to design, implement, and debug custom data structures, we propose to synthesize data structure implementations from high-level specifications. Our tool Cozy [6] does this using counter-example guided inductive synthesis (CEGIS) [12]. For instance, given the specification in Figure 1, Cozy implements `AnalyticsLog` using a hash table whose keys are tuples of query, subquery, and fragment IDs and whose values are interval trees over the entry start and end times.

FSE'16, November 13–18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2986032

```
entry fields
    queryId:long, subqueryId:long, fragmentId:int,
    opId:int, start:long, end:long, numTuples:long

query getAnalyticsInTimespan(
        v_queryId:long, v_subqueryId:long,
        v_fragmentId:int,
        v_start:long, v_end:long)
    queryId == v_queryId and
    subqueryId == v_subqueryId and
    fragmentId == v_fragmentId and
    start < v_end and
    end >= v_start
```

Figure 1: Sample input for Cozy. The specification describes a real-world data structure in Myria [7]. Given the specification as input, Cozy automatically generates an implementation with methods for adding, removing, and updating entries, as well as the `getAnalyticsInTimespan` query.

To make such a tool possible, we define a small language to outline implementations of collection operations. The space of implementation outlines is much smaller than the space of all possible programs, making outlines feasible to synthesize. We present a way to prune inefficient outlines early using a static cost model, allowing Cozy to quickly converge on good ones, and we identify a property that makes checking the correctness of an outline tractable: instead of checking correctness on every possible state of the data structure, it suffices to check correctness for every instance containing only one entry.

2. APPROACH

Our approach is based on CEGIS, a synthesis technique that works by coupling together two components: an *inductive synthesizer* that devises a program consistent with a set of input examples and a *verifier* that checks whether programs produced by the inductive synthesizer are correct on all possible inputs. Crucially, the verifier produces a counter-example whenever verification fails, and the inductive synthesizer can use the new example to refine its search.

To use CEGIS for data structure synthesis, we address several challenges: how to inductively synthesize method implementations, how to guide the CEGIS loop toward efficient solutions, and how to verify candidate programs.

The input to the inductive synthesizer is the specification and a set of *example inputs* generated by the verifier; the

output is a program consistent with the specification on the given example inputs. Cozy's inductive synthesizer works by brute force search. Since the space of all possible data structure implementations cannot be explored exhaustively in this fashion, Cozy employs several optimizations to make brute force search practical. First, instead of exploring all possible data structure implementations, Cozy explores *outlines*. An outline is a high-level functional program that describes how to retrieve a subset of the elements in the data structure. Outlines may include hash map look-ups, binary tree searches, linear-time filters, and other data structure operations. There may be many concrete programs implementing a given outline, and Cozy decides between different concrete programs in an *auto-tuning* step after finding a good outline. Second, Cozy makes use of the equivalence class optimization used in TRANSIT [13]: candidate outlines can be grouped based on their behavior on the current set of examples. Whenever two outlines behave the same on all current examples, one of them can be dropped from the search space. New examples from the verifier refine these coarse equivalence classes over time.

A static cost model defined over outlines helps to guide the search toward more efficient solutions. As an added benefit, the cost model can be used to further narrow the search space for the inductive synthesizer. The cost model we use reasons about the worst-case asymptotic cost of performing the query. Cozy uses the static cost model in two ways during inductive synthesis. First, the tool tracks a *cost ceiling* corresponding to the cost of the best valid outline found so far. Any outline having a greater cost can be safely discarded during search. Second, when the inductive synthesizer finds that two outlines belong in the same equivalence class, it can discard from its cache not just the worse outline but also any outline that uses the worse outline as a sub-component. This is a safe optimization because the search will inevitably later discover those same discarded outlines, but using the better version in place of the worse one.

Program verification is undecidable in general, but Cozy specifications and outlines are designed so that verification is efficiently decidable. For any given outline, a counter-example consisting of a single input and a single arbitrary data structure element for which the outline misbehaves is enough to show that an outline is invalid. Amazingly, the absence of such a counter-example is enough to show that an outline is valid for all possible inputs and all possible data structure states. This removes the need to reason about all possible data structure states; the verifier only needs to consider a single arbitrary element. This is called a *small-model property*: the verification problem can be reduced to the task of finding a very small counter-example.

3. RESULTS

Using four real-world programs as case studies, we have evaluated to what extent Cozy improves implementation correctness, reduces programmer effort, and affects performance. Relative to the original human-written implementations, we found that Cozy's synthesized data structures have fewer bugs, require far fewer lines of code to write, and have comparable performance.

Our four case study programs are Myria [7] (a distributed database), ZTopo [14] (a topological map viewer), Bullet [1] (a physics simulation library), and Sat4j [9] (a boolean satisfiability solver). Each one relies on a core data structure for part of its functionality. The diversity of data structures and use cases illustrates Cozy's wide applicability.

We performed a small amount of refactoring in each case study program to make the core data structure have the same interface that Cozy generates. Then we wrote a specification for each one and had Cozy synthesize a new implementation.

All the programs except ZTopo have a dedicated issue tracker. Across those three we found 33 distinct correctness bugs. Cozy's implementations do not suffer from any of these reported issues; they are correct by construction.

Cozy specifications are very small relative to the original implementations. The shortest original implementation is Myria at 269 lines of data structure code, and the longest is Bullet with more than 2500. None of the specifications for these data structures have more than 25 lines of code. Since the specifications are one to two orders of magnitude shorter, we believe that they will be much faster and easier for programmers to write.

For ZTopo and Bullet, the performance of the synthesized implementation was nearly identical to the original. In the case of Sat4j the synthesized implementation was a constant factor of 20% slower. This was because of a property that Cozy was not aware of and thus could not exploit: the map keys in Sat4j's data structure are all small integers, meaning that a simple array could be used for look-ups instead of a hash map. In Myria, however, the synthesized implementation outperformed the original. While the original implementation had worst-case linear time for some look-ups, Cozy found a synthesized implementation with worst-case $O(\log n)$ time in the size of the data structure. This led to big speedups, especially when the size of the data structure grew to be very large.

4. RELATED WORK

Automatic data structure implementation began with iterator inversion [3, 4, 8]. Iterator inversion used rewrite rules to transform set comprehensions into optimized data structures. The rules are difficult to write and even more difficult to prove exhaustive. The rewrite engine is fairly naive, so performance gains are not guaranteed. In contrast, our techniques do not require explicit rewrite rules and can guarantee optimality with respect to a simple cost model.

There was also work on automatic representation selection for the SETL language [11, 10, 2]. These techniques performed source code analysis to bound the possible contents of each data structure; the bounds could then be used to choose a good representation. However, these algorithms could only generate more efficient set or map implementations; data structures with more complex interfaces such as the one in Figure 1 were not possible.

More recent work focused on synthesizing data structures from relational logic specifications. RelC [5] exhaustively enumerates candidate data structure representations. A query planner then determines how to use each representation to implement the data structure's methods. Each candidate implementation is evaluated using an auto-tuning benchmark, and the best one is returned to the programmer. Since the RelC planner is opaque, the tool cannot rule out candidate representations quickly and relies entirely on benchmarking to select a good representation. In contrast, Cozy uses a coarse cost model to guide the search toward better implementations. Furthermore, Cozy can synthesize a wider class of data structures than RelC.

5. REFERENCES

[1] The Bullet physics library. http://bulletphysics.org (Retrieved October 29, 2015).

[2] J. Cai, P. Facon, F. Henglein, R. Paige, and E. Schonberg. Type transformation and data structure choice. In *Constructing Programs From Specifications*, pages 126–124. North-Holland, 1991.

[3] J. Earley. High level iterators and a method for automatically designing data structure representation. *Comput. Lang.*, 1(4):321–342, Jan. 1975.

[4] A. C. Fong and J. D. Ullman. Induction variables in very high level languages. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, POPL '76, pages 104–112, New York, NY, USA, 1976. ACM.

[5] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Data representation synthesis. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 38–49, New York, NY, USA, 2011. ACM.

[6] C. Loncaric, E. Torlak, and M. D. Ernst. Fast synthesis of fast collections. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2016, pages 355–368, New York, NY, USA, 2016. ACM.

[7] Myria distributed database. http://myria.cs.washington.edu (Retrieved April 10, 2015).

[8] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, July 1982.

[9] Sat4J boolean reasoning library. https://www.sat4j.org (Retrieved February 3, 2016).

[10] E. Schonberg, J. T. Schwartz, and M. Sharir. An automatic technique for selection of data representations in SETL programs. *ACM Trans. Program. Lang. Syst.*, 3(2):126–143, Apr. 1981.

[11] J. T. Schwartz. Automatic data structure choice in a language of very high level. In *Proceedings of the 2Nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '75, pages 36–40, New York, NY, USA, 1975. ACM.

[12] A. Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[13] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur. TRANSIT: Specifying protocols with concolic snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 287–296, New York, NY, USA, 2013. ACM.

[14] ZTopo topographic map viewer. https://hawkinsp.github.io/ZTopo/ (Retrieved May 8, 2015).