FSE: G:
Cozy: Synthesizing Collection Data Structures

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

Abstract

Many applications require specialized data structures not found in standard libraries. Implementing new data structures is tedious and error-prone. To alleviate this difficulty, we built a tool that does so automatically: it synthesizes efficient data structures from short, simple, declarative specifications. Our tool Cozy uses and extends the technique of counter-example guided inductive synthesis.

We evaluated Cozy on four real-world programs. We replaced programmer-written data structure implementations by implementations synthesized by Cozy. The programmer-written versions had dozens of bugs, but Cozy's versions are correct by construction. The programmer-written versions were 10–100× larger than the Cozy specifications. Cozy's versions were equally fast in 2 cases, a constant 20% slower in one case, and asymptotically faster in the other case.

Keywords Data structures, program synthesis, automatic programming

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 applications often 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 [9] does this using counter-example guided inductive synthesis (CEGIS) [16]. 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. The input specification is shown in its entirety. The output has been reduced to its interface only, as Cozy produces nearly 5000 lines of Java code to implement the specification. Since the synthesized code does not need to be human-readable, it does not need to be well-abstracted; the large volume of output code is due to inlining at code generation time to improve performance.

Existing synthesis tools cannot produce such complex data structure implementations unassisted. One technique is to require a sketch, or partial implementation, from the programmer [15, 17]. A synthesizer can then fill in missing parts of the sketch automatically using CEGIS. Unfortunately, this imposes an additional burden on the programmer since they are required to commit to many details about the implementation. A second technique enumerates many different possible in-memory representations and uses a planner—similar to query planners in databases—to implement each data structure operation in terms of the representation [6, 7]. This trades programmer effort for compute time: since the planner is a black box, the only way to search the space of implementations is by unguided brute-force search. In experiments, this technique requires over 80 calls to a dynamic benchmark before finding a good implementation. (The authors did not report how long this process took.) Even worse, limitations in the planner mean that complex data structures such as the one in Myria cannot be synthesized at all.

By contrast, Cozy requires no input from the programmer beyond a single declarative specification and can find a complex implementation very quickly: less than 90 seconds for all real-world specifications we studied. It does so by inverting the typical synthesis process. Previous approaches first select a representation—by programmer input or by brute-force search—and then implement each method—by synthesis or by planning. In contrast, Cozy first synthesizes an implementation for each method, and then infers the necessary representation for the implementation to function correctly.

Cozy synthesizes method implementations in terms of *outlines*, small functional programs that describe how to retrieve entries from the data structure. The outline language has been designed so that synthesis can take place before the data representation is known, and the data representation can later be inferred from the outline.

Contributions

- 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. Furthermore, the correct in-memory representation of the data can be inferred from the outline.
- We present a way to prune inefficient outlines early using a static cost model, allowing Cozy to quickly converge on efficient solutions.
- 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.

Section 2 describes our approach at a high level, including how Cozy derives outlines from specifications and how outlines are converted into concrete implementations. Section 3 describes results of our evaluation: on four different case studies, Cozy improves the correctness of data structure implementations while matching performance of human-written code. Finally Section 4 discusses how our approach relates to previous work.

2. Approach

Cozy synthesizes data structures in four major steps. *Outline synthesis*, our primary contribution, implements each query method—e.g. `getAnalyticsInTimespan` from Figure 1—using CEGIS. *Representation selection* examines the query method implementations to choose the representation of the data in memory. *Code generation* uses hard-coded rules to implement `add`, `remove`, and `update` methods for the chosen representation. These steps may produce several different data structures with equivalent estimated costs, so a final *auto-tuning* step runs a programmer-supplied benchmark on each one to choose the best for a particular workload.

2.1 Outline Synthesis

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 a program is correct on all possible inputs.

The inductive synthesizer acts like a "guesser": it generates candidate implementations and returns one that behaves correctly on all examples (test cases) it has ever seen. The verifier acts like a "checker": it either proves that a candidate is correct or produces a concrete example where it misbehaves. In the latter case, the inductive synthesizer restarts and produces a new candidate implementation. Each new example helps refine the candidates until the process eventually finds a correct implementation.

CEGIS has been used to synthesize data structure code before [15, 17], but doing so requires additional input from

```
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
```

```
class AnalyticsLog {
    class Entry { long queryId; ... }

    void add(Entry e)
    void remove(Entry e)
    void update(Entry e, long newQueryId, ...)

    Iterator<Entry> getAnalyticsInTimespan(
            long v_queryId, ...)
}
```

Figure 1. Sample input specification for Cozy (above) and partial corresponding output (below). The specification describes a real-world data structure in Myria [10]. Given the specification as input, Cozy automatically generates a complete implementation of the interface shown below, with methods for adding, removing, and updating entries, as well as the `getAnalyticsInTimespan` query for retrieving them. Cozy's finished implementation totals nearly 5000 lines of performant code.

the programmer in the form of a partial implementation. Our technique does not take any input from the programmer beyond the specification, introducing several new challenges:

- What should the space of candidate programs be, given that the data representation is not known yet?
- How will the synthesizer discover efficient solutions, given that existing implementations of CEGIS make no promises about the performance of the synthesized code?
- Can candidates be verified at all, given that program equivalence is undecidable in general?

Inductive Synthesis of Outlines 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.

Given the specification in Figure 1, the outline

$$\texttt{HashLookup}(state,\ queryId = v\_queryId)$$

searches the collection $state$ for all elements whose query ID equals $v\_queryId$. The outline indicates that $state$ should be stored as a hash map keyed by $queryId$ and the method should be implemented as a single hash lookup. 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 (Section 2.4).

The input to the inductive synthesizer is the specification and a set of *example inputs*; the output is an outline consistent with the specification on the example inputs. The example inputs are not provided by the programmer. Rather, they are generated by the verifier. Each example input consists of (1) a concrete data structure state listing all elements present in the data structure and (2) values for each input to the query routine.

For instance, one example input for the Myria data structure in Figure 1 might be

$$state = [\{queryId : 1, ...\}],\ v\_queryId = 2,\ ...$$

indicating that there is one element in the data structure having $queryId$ 1 and the value of $v\_queryId$ passed to the query method is 2. The other values in the example are not shown. The inductive synthesizer must devise a data structure implementation that behaves correctly according to the specification on this example—in this case, the return value should be an empty iterator, since the only entry in the data structure state does not match $v\_queryId$.

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 makes use of the equivalence class optimization used in TRANSIT [18]: 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.

Cost Optimization 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. Cozy's cost model 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.

Verification 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.

The restriction that makes Cozy's small-model property hold is that neither input specifications nor outlines can mention the entirety of the data structure state. They can only state or require properties that are true for all elements in the data structure. For example, Cozy specifications may not state that "the data structure holds a maximum of 16 elements," but they may state that "every element's age field is greater than zero."

Termination Typically, CEGIS algorithms promise that they always terminate if a program exists that correctly implements the specification. However, termination is more subtle for Cozy, since beyond finding an arbitrary solution it seeks the best solution according to the cost model. The first correct outline Cozy finds is not likely to be the optimal one.

Cozy runs until it has enumerated every outline cheaper than the best outline found. To ensure this process terminates, the cost model must be *monotonic*—estimated cost always increases with outline size—and *divergent*—estimated cost must approach infinity as outline size approaches infinity. Our cost model has both of these properties.

In practice, Cozy finds good solutions very quickly: typically in the first minute of execution. Even so, it could take many hours to explore the entire space for the optimal outline. In our evaluation we impose a 30 second timeout for synthesis.

2.2 Representation Selection

Every outline implicitly encodes requirements for the representation on which it operates. Consider again the HashLookup outline shown earlier:

$$\texttt{HashLookup}(state,\ queryId = v\_queryId)$$

This outline returns a collection of entries, but it requires the entries to be organized in a map whose entries are bucketed by $queryId$. For this outline, Cozy would produce the representation

$$state : \text{Map}\langle queryId, \text{Set}\langle\text{Entry}\rangle\rangle$$

indicating that the data must be organized as a map by $queryId$.

The task of inferring a representation for a particular outline is akin to type inference: each outline primitive encodes some constraints on the representations of the structure it *operates on* and some constraints on the shape of the structure it *returns*.

There are often multiple possible representations for a given outline; for instance, `HashLookup` does not specify the desired load factor for the hash map. When this happens, Cozy enumerates several possibilities and the later auto-tuning step decides between them. The number of possibilities is typically small: not more than 12 for any of our case studies.

2.3 Code Generation

After synthesizing outlines and representations for query operations—e.g. `getAnalyticsInTimespan` from Figure 1—Cozy uses hard-coded rules to implement `add`, `remove`, and `update`.

The add and remove methods are built out of known implementations for each possible representation type. Other researchers have investigated synthesizing operations such as add and remove on binary trees [15], but that approach is unnecessary here: the implementations of these methods are well-understood.

Cozy generates efficient update methods as well. A trivial implementation of an update routine might remove the element from the data structure, alter the relevant field, and re-insert the element. Cozy instead generates code to find the new location for the element (if different from its present location) and move it there. Most updates require very little motion; for instance, updating a field on an element in a linked list does not require any movement.

2.4 Auto-tuning

Beyond just asymptotic performance, high performance data structures ought to be tuned to particular workloads. Cozy's auto-tuning step takes each generated candidate implementation and evaluates it against a programmer-provided benchmark. Cozy outputs the best-scoring implementation. The benchmark acts as a fine-grained cost model and provides a natural way to express the needs of a particular workload.

The auto-tuning benchmark is not required; if omitted, Cozy will select an implementation arbitrarily. The implementation will still be guaranteed correct.

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.

Case Studies and Methodology Our four case study programs are Myria [10] (a distributed database), ZTopo [19] (a topological map viewer), Bullet [1] (a physics simulation library), and Sat4j [12] (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.

For each case study we replaced a central data structure with a synthesized version, then measured correctness (in terms of existing tests and bug reports), programmer effort (in terms of commits and lines of code saved), and performance (on realistic workloads). Table 1 summarizes our results.

Correctness All the programs except ZTopo have a dedicated issue tracker. Across those three we found 33 distinct correctness bugs. These include memory mismanagement, nondeterminism, and functional correctness errors. We verified that cozy's implementations do not suffer from any of these reported issues; they are correct by construction and they pass all existing tests for each project. Among those 33 issues, Sat4j includes three related to performance. Since Cozy optimizes its data structures according to a static cost model, even performance bugs such as these can be avoided.

Programmer Effort 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 have strong reason to believe that they will be much faster and easier for programmers to write. Furthermore, Table 1 shows that the programmers did not produce perfect implementations right away: each required several commits to get right.

Performance Myria, Bullet, and Sat4j have existing benchmarks that the developers use to tune performance. We measured the change in overall performance on these benchmarks after integrating Cozy's data structures. For ZTopo, we assembled our own benchmark by recording several minutes of normal use of ZTopo's graphical interface. Table 1 reports the change in end-to-end benchmark time. Performance of the internal data structure in each case is a dominating factor.

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 15% 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 the synthesized implementation greatly 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

Synthesis techniques have seen success in specialized domains such as optimizing bit-vector programs [8] and de-

Program	LoC (impl/spec)	Commits	Bugs	Synth. time (s)	Benchmark Δ
Myria	269 / 22	88	11	75	137%
ZTopo	1383 / 23	57	15	84	91%
Bullet	2582 / 25	15	-	47	94%
Sat4j	292 / 11	22	7	7	85%

Table 1. Results of our evaluation on four case studies. "LoC" shows the number of lines of code in the programmer-written implementation versus the specification. "Commits" shows the number of commits in the version history of the programmer-written implementation. "Bugs" shows the number of bugs reported on the project's issue tracker for the programmer-written structure. Note that ZTopo does not have a dedicated issue tracker. "Synthesis time" shows the time required for Cozy to produce a complete implementation. "Benchmark Δ" shows the overall performance change on a realistic benchmark after integrating the synthesized structure as a ratio of original time to new time; higher is better, >100% indicates that the synthesized implementation is faster, and <100% indicates that the programmer-written implementation is faster.

riving string transformations from examples [5]. Our work extends these techniques beyond program snippets to whole class implementations.

Automatic data structure implementation began with iterator inversion [3, 4, 11]. 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 cost model.

There was also work on automatic representation selection for the SETL language [2, 13, 14]. 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.

Researchers have investigated using synthesis techniques to automatically produce implementations of individual data structure operations [15, 17]. These approaches require the developer to already know the correct in-memory representation for their data structure, and they do not optimize for performance. Instead, they are focused on correctly implementing tricky operations like binary tree transformations. In the future, these techniques might be used to enhance Cozy by allowing it to synthesize very low-level operations instead of using hard-coded rules for operations like binary tree insertion.

More recent work focused on synthesizing data structures from relational logic specifications. RelC [6] 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: Cozy specifications can include disjunctions and inequalities, while RelC specifications can only have conjunctions of equalities.

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] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[6] 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.

[7] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Concurrent data representation synthesis. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 417–428, New York, NY, USA, 2012. ACM.

[8] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software*

Engineering - Volume 1, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM.

[9] 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.

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

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

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

[13] 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.

[14] 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.

[15] R. Singh and A. Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 289–299, New York, NY, USA, 2011. ACM.

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

[17] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent data structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 136–148, New York, NY, USA, 2008. ACM.

[18] 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.

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