# Fast Synthesis of Fast Collections

Calvin Loncaric Emina Torlak Michael D. Ernst
University of Washington, USA
{loncaric,emina,mernst}@cs.washington.edu

## Abstract

Many applications require specialized data structures not found in the standard libraries, but implementing new data structures by hand is tedious and error-prone. This paper presents a novel approach for synthesizing efficient implementations of complex collection data structures from high-level specifications that describe the desired retrieval operations. Our approach handles a wider range of data structures than previous work, including structures that maintain an order among their elements or have complex retrieval methods. We have prototyped our approach in a data structure synthesizer called Cozy. Four large, real-world case studies compare structures generated by Cozy against handwritten implementations in terms of correctness and performance. Structures synthesized by Cozy match the performance of handwritten data structures while avoiding human error.

***Categories and Subject Descriptors*** E.1 [*Data Structures*]; I.2.2 [*Automatic Programming*]: Program Synthesis

***Keywords*** Data structure synthesis

## 1. Introduction

Fast data structures are key to good performance. All mainstream languages ship with well-written libraries implementing lists, maps, sets, trees, and other common data structures. These libraries are sufficient for most use cases. However, many high performance applications need specialized data structures with more complex operations. For such applications, the standard libraries are not enough.

Myria [21], a distributed database, is one such application. The Myria developers have spent much time implementing and maintaining code to store analytics data collected during

*PLDI'16*, June 13–17, 2016, Santa Barbara, CA, USA
ACM. 978-1-4503-4261-2/16/06...$15.00
http://dx.doi.org/10.1145/2908080.2908122

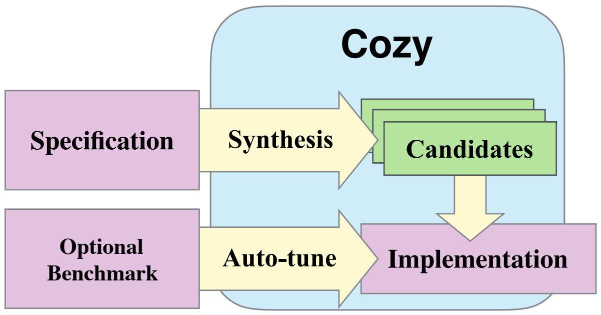

Figure 1: The architecture of the collection synthesizer Cozy. Figure 2 shows an example input specification. The synthesizer outputs a number of candidate data structure implementations with good asymptotic performance, and Cozy uses an optional client benchmark program to choose among them.

query execution. The analytics data powers an interactive viewer [20] to allow clients to understand the performance of their queries. The analytics data structure needs to support efficient insertion and retrieval operations, both to minimize overhead during execution and to allow clients to examine the data interactively. Its implementation was a repeated source of pain for nearly a year and still falls short of the developers' expectations.

Two factors are responsible for the Myria team's difficulties. First, the needs of the interactive viewer render the standard libraries insufficient. Each entry in Myria's analytics data describes what query caused it, what operation was taking place, and when that operation started and ended. The data structure must retrieve analytics data for a specific query, limited to those operations that overlap a given time range. Efficient implementation of this retrieval operation requires the use of an interval tree, a collection that is not found in any common collections library. To avoid implementing one by hand, the Myria team opted to store analytics data in a SQL database. Their choice obviated some implementation work, but made it difficult for them to control performance. Even for database developers, the behavior of SQL query optimizers can be difficult to predict. Occasional changes to the specification constituted the Myria team's second difficulty. For example, new features elsewhere in the codebase required analytics data to be additionally grouped by a new "subqueryId" field. Updating the data structure code to support the subquery field efficiently was an arduous two-week process.

Our Cozy tool (Figure 1) can automatically synthesize the implementations of many complex data structures—including Myria's analytics data structure—from short high-level specifications (Figure 2). In the case of Myria, the resulting implementation outperforms the existing handwritten one on the Myria team's benchmark data. Furthermore, changes to the data structure's requirements only require small changes to the specification. Cozy can eliminate many hours of work spent writing, tuning, and debugging data structure code.

Like previous work [17], Cozy targets collection data structures having add, remove, update, and retrieval methods over a single type. Unlike previous work, Cozy can synthesize data structures with complex retrieval operations involving disjunctions, negations, and inequalities—such as the inequalities found in Figure 2. A thorough discussion of what sets our approach apart can be found in Section 5.

There are an infinite number of possible data structure implementations in this domain, and it would be impractical for Cozy to naively explore the entire space. We introduce three specific innovations to make the problem tractable:

1. We define a small language to *outline* implementations of collection retrieval operations (Figure 4). The space of implementation outlines is much smaller than the space of all possible programs, making outlines feasible to synthesize.
2. We identify a property that makes checking the correctness of a candidate outline tractable: instead of checking correctness on every possible instance of the data structure, it suffices to check the correctness for every instance containing one arbitrary entry.
3. We present a way to prune inefficient outlines early using a static cost model, allowing our tool to quickly converge on good ones.

We have evaluated our approach on four large real-world subject programs from different domains, investigating both correctness and performance. Cozy and our evaluation files have been made available online [10].

The rest of this paper is organized as follows. The synthesis problem and algorithm are described at a high level in Section 2 and in detail in Section 3. Our experimental results are presented in Section 4. Related work is addressed in Section 5, and Section 6 concludes.

## 2. Overview

This section presents an overview of Cozy using an example a graph data structure not found in the Java or C++ standard libraries. The graph will be a directed unweighted multigraph, i.e. a graph that may have multiple edges between nodes. The graph will allow self-loops (edges from a node to itself), and it will support one retrieval operation `findEdges` for finding all incoming and outgoing edges from a given node. Nodes will be represented as integers, although the programmer could choose some other type.

```
fields
    queryId:long, subqueryId:long,
    fragmentId:int, opId:int,
    startTime:long, endTime:long,
    numTuples:long

assume startTime <= endTime

query getAnalyticsInTimespan(
        v_queryId:long, v_subqueryId:long,
        v_fragmentId:int,
        v_start:long, v_end:long)

    assume v_start <= v_end

    queryId == v_queryId and
    subqueryId == v_subqueryId and
    fragmentId == v_fragmentId and
    startTime < v_end and
    endTime >= v_start

costmodel myria-cost.java
```

Figure 2: Full specification for the Myria analytics data structure. Each entry in the structure has seven fields (`queryId`..`numTuples`). The structure supports one retrieval operation `getAnalyticsInTimespan`, which finds all the entries for a given query ID and fragment ID that overlap the given range. An `assume` statement states facts about the entries and preconditions on query variables that can be exploited by the synthesizer. The last line specifies a dynamic cost model in the form of a client benchmark program, used to select the best implementation among several candidates.

The Cozy specification of the graph data structure is given by:

```
fields src : int, dst : int
query findEdges(node : int)
    src == node or dst == node
```

Figure 3 shows the full language of allowed query expressions. Cozy outputs a source file having the following public interface, with optimized implementations for all methods:

```
class Graph {
    class Edge { int src; int dst; }

    void add(Edge r) {...}
    void remove(Edge r) {...}
    void updateSrc(Edge r, int newSrc) {...}
    void updateDst(Edge r, int newDst) {...}

    Iterator<Edge> findEdges(int node) {...}
}
```

Every collection synthesized by Cozy has the same shape: a single element type (in this case `Edge` with two `int` fields), an add method, a remove method, an update method for each

⟨*predicate*⟩ ::= True | False
  | ⟨*var*⟩ ⟨*comparison*⟩ ⟨*var*⟩
  | ⟨*predicate*⟩ And ⟨*predicate*⟩
  | ⟨*predicate*⟩ Or ⟨*predicate*⟩
  | Not ⟨*predicate*⟩

⟨*var*⟩ ::= ⟨*field*⟩ | ⟨*query-var*⟩

⟨*comparison*⟩ ::= '==' | '>' | '>=' | '<' | '<=' | '!='

Figure 3: Expressions for specifying retrieval methods. Some $var$s refer to fields on the element type, while others called "query variables" are inputs for the retrieval method and are only known at runtime. The complete input to Cozy is a set of *field*s, a predicate, and a set of assumptions about the data; Figure 2 shows an example.

field, and a retrieval method for each query in the specification (in this case `findEdges` with one `int` input).

## 2.1 Implementation

To implement the five methods of the `Graph` class, Cozy first searches for an efficient implementation outline for `findEdges`. The outline is expressed in the language shown in Figure 4. With an outline in hand, Cozy enumerates possible representations for the `Graph` class: what data the `Graph` class needs to store and how it needs to be organized. Figure 5 shows the possible representations. In the final step, Cozy generates code for all five methods. No additional searching needs to be done to find an implementation for the add, remove, and update methods; their implementations follow directly from the representation.

***Outlining `findEdges`*** An *outline* is a high-level functional program for retrieving a set of elements. The `findEdges` retrieval method takes one input, `node`. After synthesis, Cozy settles on the following outline for `findEdges`:

```
Concat(
  HashLookup(AllWhere(dst!=src), dst, node),
  HashLookup(AllWhere(True), src, node))
```

Each function call in the outline returns a collection of edges. The outline states that the result will be computed by concatenating two disjoint result sets. The first result set will be obtained via hash lookup on a table that indexes edges by their destination node `dst`. However, this table will only contain edges for which `dst!=src`. The expression `dst!=src` is a *guard*: the synthesized structure will check the guard at insertion time to determine whether to insert the edge into the map. The second result set will be obtained via hash lookup on a table that indexes edges by their source node `src`. This second table will contain every edge.

Cozy chooses the outline shown above because it avoids duplicating edges in the result set. A procedure that simply performs two hash lookups—one for incoming edges and one for outgoing edges—would contain every self-loop twice in the result.

⟨$outline$⟩ ::=
**AllWhere** ⟨$guard$⟩
  returns every entry in the data structure matching a guard—i.e. a predicate that can be checked at insertion time
**HashLookup** ⟨$outline$⟩ ⟨*field*⟩ ⟨$var$⟩
  performs a hash lookup on its argument to find entries where the given *field* equals the given $var$
**BinarySearch** ⟨$outline$⟩ ⟨*field*⟩ [>|≥|<|≤] ⟨$var$⟩
  performs a binary search to find entries where the given *field* is related to $var$ by the given comparison operator
**Filter** ⟨$outline$⟩ ⟨$predicate$⟩
  removes entries not matching the given predicate
**Concat** ⟨$outline$⟩ ⟨$outline$⟩
  requires that its two arguments be disjoint—there can be no entries which would be returned by both—and concatenates its two arguments

Figure 4: Language of outlines for retrieving sets of entries. Each outline is specific enough to assign an asymptotic bound, but still admits several possible concrete implementations. The exact data structure an outline returns is unspecified unless otherwise noted—this must be computed before code generation (subsection 3.2).

Deduplication could also be accomplished if the outline language had a "set union" operator, but Cozy does not need such an operator. Whenever set union would be appropriate, an equivalent outline using Concat and other filters will suffice. Concretely, whenever Union$(x, y)$ is a valid way to implement an outline, Cozy will instead discover Concat$(x,$ Filter$(y, P))$ where $P$ is a predicate that removes from $y$ all elements that might appear in $x$. In the graph example, $P$ is `dst!=src` and is implemented as a guard since it can be evaluated at insertion time. The Concat form has the added advantage that it can be executed in constant memory, as opposed to a set union which requires intermediate storage.

***Representation selection*** Given an outline, Cozy next determines a *representation* that describes how to arrange data in memory. Cozy builds representations out of known data structures; Figure 5 gives the full list. One possible representation for Cozy's proposed outline is a pair of hash maps, keyed by integer node identifiers. Under this representation, the `Graph` class should have two members:

```
HashMap<int, LinkedList> m1;
HashMap<int, LinkedList> m2;
```

The first map `m1` will be used for answering the first hash lookup, and the second map `m2` will be used for answering the second.

In cases where there are several possible representations—for instance, a BinarySearch could be performed on either a binary search tree or a sorted array—Cozy enumerates all of them and uses a programmer-provided auto-tuning benchmark to decide which one to use.

Data representations often have many opportunities for sharing data. For instance, the maps `m1` and `m2` could be fused into one map organizing edges by node and having each edge present in the map in one or two locations. Data structures having more than one retrieval method often exhibit several opportunities for sharing data. Exploiting sharing opportunities has been well described by past work [17], and so is not addressed here. In practice, we found output of Cozy to be efficient without solving the sharing problem.

***Code generation*** Cozy can output both Java and C++ code. To implement `findEdges`, the outline can be directly translated into code implementing an `Iterator` type. The implementations of the add, remove, and update methods are then written in terms of known add, remove, and update methods for `HashMap` and `LinkedList`. Specifically,

- The graph `add` method inserts an edge into the first map when `dst!=src`, and into the second map always.
- The graph `remove` method unhooks an edge from all linked lists it appears in.
- The graph `update*` methods move an edge to the correct hash buckets for its new values of `src` or `dst`.

## 2.2 Alterations

To demonstrate the flexibility of Cozy's specification language, consider changing the graph specification to exclude self-loops. Doing so requires only one addition to the specification:

```
assume src != dst
```

This assumption states that an edge will never connect a node to itself. Every assumption becomes a precondition for the `add` method. An implementation of the data structure could enforce assumptions at runtime to aid debuggability; Cozy does not.

The addition of the assumption `src != dst` results in the following modified outline for `findEdges`:

```
Concat(
  HashLookup(AllWhere(True), dst, node),
  HashLookup(AllWhere(True), src, node))
```

Since the collection may not contain edges with the same source and destination, the implementation can omit the guard on the first hash map.

## 3. Approach

This section describes the algorithms that Cozy uses to synthesize outlines, pick data representations, and generate concrete code.

### 3.1 Synthesis of Outlines

In general, the synthesis problem is to find a program $P$ with respect to a specification $s$ such that for all inputs $\vec{i}$,

$$\forall \vec{i}, s(\vec{i}, P(\vec{i})).$$

```
def synthesize_outline(spec, examples):
  cache = all_size1_outlines()
  best_progs = { initial_guess() }
  best_cost = cost(initial_guess())

  for size in [2..]:
    for P in enum_outlines(cache, size):
      if cost(P) > best_cost:
        # A better correct program exists
        continue
      elif not correct_on_examples(P, examples):
        # P is not correct, but it may be
        # a part of a correct program.
        cache += { P }
        cleanup(cache, examples)
      elif is_correct(P, spec):
        # P is correct!
        best_cost = min(
          best_cost, cost(P))
        best_progs += { P }
        cleanup(best_progs, examples)
      else:
        # P looks correct but is not. A new
        # example will help distinguish it.
        ex = find_counterexample(P, spec)
        return synthesize_outline(
          spec, examples + {ex})

  return best_progs
```

Figure 6: Cozy's algorithm for program synthesis, based on CEGIS. Programs are found using brute-force search with a twist: the cost model `cost` allows many candidate programs to be dropped in calls to `cleanup`, pruning the search space drastically.

In Cozy, outlines are functional programs and the query predicates are specifications: the retrieval method for that query must return exactly those entries which are in the data structure and match the predicate.

To synthesize outlines from a query, Cozy employs counterexample guided inductive synthesis (CEGIS) [32], a common technique for simplifying the task of solving for $P$ given $s$. The CEGIS approach splits this hard problem into two easier ones: a *synthesizer* and a *verifier*. The synthesizer takes a finite set of input examples and finds $P$ such that $P$ behaves correctly on the given examples:

$$s(\vec{i_1}, P(\vec{i_1})) \wedge \ldots \wedge s(\vec{i_n}, P(\vec{i_n})).$$

The verifier takes a program $P$ and either proves that $P$ is correct or finds a concrete counterexample: $i$ such that $\neg s(\vec{i}, P(\vec{i}))$.

A CEGIS loop starts with zero or more examples and alternates back and forth, asking the synthesizer for a guess based on all the current examples and then checking the guess against the verifier. If the verifier rejects the guess, the synthesizer adds the resulting counterexample to its running set of examples and the procedure restarts using the larger set. If the verifier accepts the program, the algorithm has found a correct answer.

$$
\begin{array}{rll}
\langle \textit{collection} \rangle & ::= & \text{Iterable} \\
 & | & \text{ArrayList} \\
 & | & \text{LinkedList} \\
 & | & \text{SortedIterable } \langle \textit{field} \rangle \\
 & | & \text{SortedArray } \langle \textit{field} \rangle \\
 & | & \text{BinaryTree } \langle \textit{field} \rangle \\
 & | & \text{HashMap } \langle \textit{field} \rangle \ \langle \textit{collection} \rangle \\
 & | & \text{Tuple } \langle \textit{collection} \rangle + \\
 & | & \text{Guarded } \langle \textit{collection} \rangle \ \langle \textit{guard} \rangle \\
 & | & \text{Filtered } \langle \textit{collection} \rangle \ \langle \textit{predicate} \rangle
\end{array}
$$

$$
\begin{array}{l}
t <: t \\
\text{ArrayList} <: \text{Iterable} \\
\text{LinkedList} <: \text{Iterable} \\
\text{SortedIterable } f <: \text{Iterable} \\
\text{SortedArray } f <: \text{SortedIterable } f \\
\text{BinaryTree } f <: \text{SortedIterable } f \\
(t_0 <: u_0 \wedge ...) \rightarrow (\text{Tuple } t_0 ... <: \text{Tuple } u_0 ...) \\
(t <: u) \rightarrow (\text{HashMap } f\ t <: \text{HashMap } f\ u) \\
(t <: u) \rightarrow (\text{Guarded } t\ p <: \text{Guarded } u\ p) \\
(t <: u) \rightarrow (\text{Filtered } t\ p <: \text{Filtered } u\ p)
\end{array}
$$

Figure 5: Data structure implementations supported by Cozy (left) and subtyping relationships between them (right). The relationship $A <: B$ indicates that $A$ can be treated as $B$ for the purposes of retrieving entries. "Guarded" and "Filtered" modify the collections they wrap. Guarded collections use the *guard* expression to determine whether to insert an element. Thus, Guarded collections only contain elements for which the guard expression is true. Filtered collections use the *predicate* to filter out entries at retrieval time. A filter predicate may reference query variables, while a guard expression may not.

***Synthesis*** Figure 6 shows the synthesis half of Cozy's implementation of CEGIS. Unlike standard CEGIS, Cozy's algorithm returns *all* programs having minimum cost according to a static cost model.

The algorithm begins with a guess for how to implement the outline. The initial guess must be a correct outline for the input query, and serves to bound the maximum cost of any generated outline. For any query $Q$, the initial guess `Filter(AllWhere(True), Q)`—corresponding to a bag of elements filtered at retrieval time—is always correct.

The Cozy algorithm uses brute-force search, enumerating programs in order of increasing size. However, instead of naively enumerating *all* programs, Cozy memoizes smaller programs in a `cache` and only visits programs that can be built using components in the cache. The `enum_outlines` function enumerates programs of the given size that can be built from programs in the given cache. Keeping the cache small keeps the synthesis manageable.

The cache is kept small using the `examples` returned by the verifier, as done in TRANSIT [33]. The `cleanup` procedure groups programs by equivalence class based on their behavior on the examples and sorts each equivalence class by cost. Whenever two programs in an equivalence class have different costs `cleanup` removes the worse one. These optimizations never cause Cozy to skip plans since the cost model is monotonic. The next subsection discusses the requirements on the cost model in more detail.

Beyond keeping the cache size small, the list of examples also helps to avoid unnecessary calls to the verifier. The `correct_on_examples` function checks `P` against the current examples; if it is wrong on any of them, then `P` is surely not the desired program and the verification step can be skipped.

***Static cost model*** Figure 7 shows the implementation of Cozy's static cost model (`cost` in Figure 6). The static cost model estimates the asymptotic cost of potential retrieval programs, and helps to prune the search space during synthesis.

The cost of many primitives depends on the cardinality of the dataset they operate on. For example, BinarySearch requires $\log n$ time, where $n$ is the number of elements in the dataset. Cozy's cost model uses the cardinality heuristic shown in Figure 7. In situations where the workload is well-understood, a custom cardinality heuristic may be appropriate. We did not implement any custom cardinality estimation routines for our case studies.

An important property of Cozy's static cost model is that it is *monotonic*: the cost of a given outline is strictly greater than its arguments' costs. In the synthesis algorithm of Figure 6, the first if-check excludes all programs worse than the set of correct programs found so far. If a program built using $P$ could have a lower cost than $P$, then discarding $P$ could cause the synthesizer to skip some correct outlines. Thus, that if-check assumes monotonicity of the cost model.

***Verification*** The verification half of the CEGIS algorithm checks whether a candidate program is correct. Doing so amounts to implementing `is_correct` and `find_counterexample` from Figure 6. Many verifiers—such as those for straight-line assembly snippets—can rely on the finiteness of the program input to decide the verification problem using off-the-shelf solvers. For Cozy outlines, however, the verifier needs to guarantee correctness for every possible state that the data structure could be in; there are an infinite number of such states.

Fortunately, outlines have a *small-model property* that renders the complete state of the data structure irrelevant for verification; showing correctness for all possible states containing a single entry is sufficient. The small-model property is made possible by the fact that the semantics of every outline $P$ is characterized by some predicate $p(\vec{i}, x)$ over input $\vec{i}$ and collection elements $x$. The set of entries returned by $P$ on input $\vec{i}$ when the data structure contains elements $E$ is exactly $\{x \mid x \in E \wedge p(\vec{i}, x)\}$. Figure 8 shows the conversion from outlines to predicates.

$$cost(n,P) = 1 + \begin{cases} \text{AllWhere \_} & : 0 \\ \text{HashLookup } p \text{ \_ \_} & : cost(n,p) \\ \text{BinarySearch } p \text{ \_ \_ \_} & : cost(n,p) + \log(cardinality(n,p)) \\ \text{Filter } p \text{ \_} & : cost(n,p) + cardinality(n,p) \\ \text{Concat } p_1\ p_2 & : cost(n,p_1) + cost(n,p_2) \end{cases}$$

$$cardinality(n,P) = \begin{cases} \text{AllWhere \_} & : n \\ \text{HashLookup } p \text{ \_ \_} & : cardinality(n,p)/3 \\ \text{BinarySearch } p \text{ \_ \_ \_} & : cardinality(n,p)/2 \\ \text{Filter } p \text{ \_} & : cardinality(n,p)/2 \\ \text{Concat } p_1\ p_2 & : cardinality(n,p_1) + cardinality(n,p_2) \end{cases}$$

Figure 7: Static cost heuristic and cardinality estimation for the worst-case asymptotic time to run an outline $P$ on a structure containing $n$ entries. For cardinality estimation, HashLookup is given a more aggressive size reduction since equality constraints generally return fewer elements than inequalities.

$$\frac{}{\text{AllWhere } p \iff p(\vec{i},x)}$$

$$\frac{P \iff p(\vec{i},x)}{\text{HashLookup } P\ f\ v \iff p(\vec{i},x) \wedge x.f = v}$$

$$\frac{P \iff p(\vec{i},x)}{\text{BinarySearch } P\ f > v \iff p(\vec{i},x) \wedge x.f > v}$$

$$\frac{P \iff p(\vec{i},x)}{\text{Filter } P\ q \iff p(\vec{i},x) \wedge q(\vec{i},x)}$$

$$\frac{\forall\vec{i}\forall x, \neg(p(\vec{i},x) \wedge p_2(\vec{i},x)) \quad P_1 \iff p_1(x) \qquad P_2 \iff p_2(x)}{\text{Concat } P_1\ P_2 \iff p_1(\vec{i},x) \vee p_2(\vec{i},x)}$$

Figure 8: Outlines and their equivalent predicates. The judgment $P \Leftrightarrow p(\vec{i},x)$ means that the outline $P$ returns exactly those entries $x$ in the data structure where $p(\vec{i},x)$ holds. (Only the rule for BinarySearch with $<$ is shown; the rules for the other operators are symmetric.)

For a given input query predicate $q(\vec{i},x)$ over query variables $\vec{i}$ and elements $x$, the overall specification to check is that the outline returns only those elements in the data structure that satisfy $q$. Formally,

$$s(\vec{i},P(\vec{i})) := \forall E \forall x, (x \in P(\vec{i})) \iff (x \in E \wedge q(\vec{i},x)).$$

Since $P(\vec{i}) = \{x \mid x \in E \wedge p(\vec{i},x)\}$, the specification formula can be simplified as follows

$$\begin{aligned} s(\vec{i},P(\vec{i})) :=& \forall E \forall x, (x \in P(\vec{i})) \iff (x \in E \wedge q(\vec{i},x)) \\ =& \forall E \forall x, (x \in \{x \mid x \in E \wedge p(\vec{i},x)\}) \iff \\ & \quad (x \in E \wedge q(\vec{i},x)) \\ =& \forall E \forall x, (x \in E \wedge p(\vec{i},x)) \iff \\ & \quad (x \in E \wedge q(\vec{i},x)) \\ =& \forall E \forall x, (x \in E) \rightarrow (p(\vec{i},x) \iff q(\vec{i},x)) \end{aligned}$$

To solve this formula, Cozy instead solves the negation:

$$\neg s(\vec{i},P(\vec{i}) = \exists E \exists x, (x \in E) \wedge \neg(p(\vec{i},x) \iff q(\vec{i},x))$$

In the negation, the only constraint on the set $E$ is that $x \in E$. Therefore, if we take $E$ to be the universal set containing all elements, any solution to the simpler formula $\exists x, \neg(p(\vec{i},x) \iff q(\vec{i},x))$ is a solution to the entire formula. Cozy solves the verification problem by asking Z3 [12] for an instance of $x$ and $\vec{i}$, and the exact contents of the data structure never need to be considered.

***Termination*** Standard CEGIS algorithms stop immediately when a solution has been found. Many approaches enumerate programs in order of size and always return the shortest correct program. This is a good choice when size correlates strongly with performance, but Cozy's static cost model does not have a strong correlation with outline size. The first correct outline Cozy finds is not likely to be the optimal one. Therefore, Cozy runs until it has enumerated every outline cheaper than the best outline found. To ensure this process terminates, the cost of a plan may never asymptotically approach a fixed value as the size of the plan increases. The $1+$ term in Figure 7 has the effect of adding the size of the outline to its cost, thereby ensuring that the cost does not asymptotically approach a fixed value.

$$\frac{t <: u \qquad concrete(t)}{\text{AllWhere} : t \rightarrow u} \qquad \frac{p : t \rightarrow \text{HashMap}\langle k, v\rangle}{\text{HashLookup}\ p\ k\ \_ : t \rightarrow v}$$

$$\frac{p : t \rightarrow \text{SortedArray}\langle f\rangle}{\text{BinarySearch}\ p\ f\ \_\_ : t \rightarrow \text{SortedArray}\langle f\rangle}$$

$$\frac{p : t \rightarrow \text{BinaryTree}\langle f\rangle}{\text{BinarySearch}\ p\ f\ \_\_ : t \rightarrow \text{SortedIterable}\langle f\rangle}$$

$$\frac{p_1 : t \rightarrow \text{Iterable} \qquad p_2 : u \rightarrow \text{Iterable}}{\text{Concat}\ p_1\ p_2 : \text{Tuple}\langle t, u\rangle \rightarrow \text{Iterable}}$$

$$\frac{p : t \rightarrow \text{Iterable}}{\text{Filter}\ p\ \_ : t \rightarrow \text{Iterable}}$$

Figure 9: Rules for inferring data structure representations. The judgment $p : t \rightarrow u$ means "outline $p$ operates on a structure with representation $t$ and returns a structure with representation $u$". The predicate $concrete(t)$ means that $t$ does not reference the abstract types Iterable or SortedIterable.

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

### 3.2 Representation Selection

Every outline implicitly encodes requirements for the representation on which it operates. In the multigraph example from Section 2, the outline operates on a data structure having two hash maps. The task of inferring a representation for a particular outline is akin to type inference: each planning primitive encodes some constraints on the representations of the structure it *operates on* and some constraints on the shape of the structure it *returns*.

Cozy enumerates all possible representations for every outline. Doing so is equivalent to inferring possible types for each `AllWhere` call in the outline. Figure 5 shows the space of possible representations. Figure 9 shows inference rules for determining whether an outline is legal for a particular representation, and these rules can be converted into a syntax-directed algorithm for enumerating all possible representations for a given outline.

Note that not all outlines have representations. For instance, an outline with the shape

$$\text{HashLookup}(\text{BinarySearch}(...), ...)$$

does not correspond to a meaningful program: the collection that results from a binary search is always a SortedIterable (see Figure 9), so it does not make sense to treat it as a hash table. Cozy ignores outlines that do not have any representation.

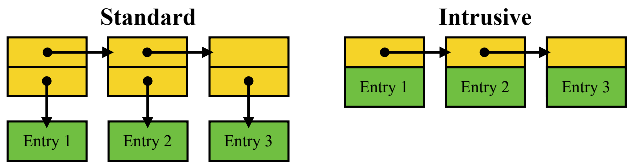

Figure 10: Standard versus intrusive linked list implementation. In the latter, the data entries themselves hold the "next" pointers used by the list implementation.

Cozy specifications of retrieval methods may include a directive to sort by a specified field $f$. In this case, Cozy only allows representations for which the outline produces a subtype of SortedIterable$\langle f\rangle$. Thus Cozy can also synthesize collections that maintain their elements in sorted order.

### 3.3 Code Generation

The outline and the representation together fully describe the implementation. Code generation does not require search; the implementations for add, remove, and update are defined by the representation. The implementation for each retrieval method is derived from its outline.

Cozy generates *intrusive* data structures. As shown in Figure 10, an intrusive data structure is one which avoids a layer of indirection by storing auxiliary data—such as linked list pointers—on its elements directly. Standard collection types usually introduce a layer of indirection in the form of "node" types, such as the nodes of a linked list or a binary tree or a hash map. In the graph example from Section 2, Cozy will add next and previous pointers to the `Edge` type for each linked list an edge could be a part of, and it does not generate an additional linked list node type.

Intrusive data structures trade some flexibility for increased performance. An element of an intrusive data structure may not exist simultaneously in multiple instances of the data structure, since it only has one set of pointers to use. However, the implementations of removal and update methods can be made much more efficient since there is no need to search for a given element in the data structure first; each element is a "handle" to its own location in the data structure.

In practice, we found intrusiveness to be beneficial. For all of our case studies the data structure under analysis is a singleton object, so there is no risk of elements existing in multiple instances at once. We observed a noticeable performance boost from faster removal and update methods. It should also be noted that intrusiveness is not fundamental to our approach; the code generator could equivalently have been implemented to use traditional data structures.

***Implementing `add` and `remove`*** The add and remove methods are built out of known implementations for the types in

Figure 5. Other researchers have investigated synthesizing operations such as add and remove on binary trees [28], but Cozy has been hard-coded with implementations of these methods for each of the possible representations.

***Implementing `update`*** Cozy generates efficient update methods. A simple implementation of an update routine removes the element from the data structure, alters the relevant field, and re-inserts 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 action to be taken.

***Implementing retrieval methods*** The retrieval methods, such as `findEdges` in the graph example from Section 2, are implemented according to the functional algorithm given in the outline. However, rather than collecting a large set of elements to return, Cozy returns an iterator type with `hasNext` and `next` methods for streaming the result set. As with the implementations of add and remove, the implementations of the iterator methods have already been written for each representation and simply need to be composed according to the outline.

***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. The best scoring implementation is the final output of the tool. The benchmark acts as a fine-grained cost model and provides a natural way to express the needs of a particular workload.

### 3.4 Implementation Details

Cozy is slightly more powerful in implementation than the basic approach described thus far. It can handle data structure specifications with multiple query operations and it has several additional output structures in its library.

***Handling multiple queries*** Many interesting data structures require more than one query operation. Cozy supports this by synthesizing a data structure for each query operation separately. These substructures are then combined into a complete implementation that stores each one independently. When an entry is added, removed, or updated in the complete implementation, that change is pushed to each substructure. When a query method is invoked, it is simply dispatched to the appropriate substructure.

This combination strategy is well-suited for the intrusive data structures Cozy generates. Entries are not duplicated across substructures; instead, they simply receive additional pointer fields for each substructure they belong to. As a result, Cozy can emit complex data structures such as a linked list threaded through a binary search tree.

***Improved expressiveness*** Cozy can be extended to work with more complex types and additional implementation strategies. Cozy contains three such extensions: augmented binary search trees, multi-field hash maps, and vector maps. The implementations of these features in Cozy demonstrate the steps which might be taken to implement other extensions.

Augmented binary search trees are binary search trees in which each node is augmented with an aggregate minimum or maximum of some property of its subtrees. They are often used to accelerate searching for intervals; the nodes in the tree might be sorted by the start value of the interval, and be augmented with the maximum end value of any subtree. Queries for nodes overlapping a particular point can now skip subtrees whose maximum end times land before the desired point.

Cozy supports augmented binary search trees by extending the BinarySearch primitive to handle arbitrary conjunctions of greater-than and less-than comparisons, and by extending representation selection and code generation with an additional `AugTree` type.

Multi-field hash maps are hash maps whose keys contain several fields, rather than just one. We observed Cozy frequently emitting nested hash lookups, resulting in `HashMap<T1, HashMap<T2, ...>>` types. Multi-field hash maps fuse nested maps into a single map with type `HashMap<Pair<T1, T2>, ...>`. This extension only affects the code generator, and was simply implemented as another choice it can make when enumerating implementations.

Vector maps are a faster alternative to hash maps. When a key type is enumerable—such as `bool`, which has only two possible values—a HashMap can be replaced by a fixed-length vector with one entry for each possible key. The code generator makes this transformation aggressively whenever possible.

## 4. Evaluation

To evaluate Cozy, we examined the extent to which it improves correctness and performance. We replaced the implementations of core data structures in four real-world subject programs from disparate domains, summarized in Table 1.

### 4.1 Subject Programs

We replaced data structures in Myria [21] (a distributed database), Bullet [6] (a physics simulation library), ZTopo [34] (a topographic map viewer), and Sat4J [23] (a boolean satisfiability solver). These programs were selected because they are real-world programs that require high performance from a central data structure.

Myria, which was introduced in Section 1, is a distributed database implemented in Java. When issuing a query to the database, in addition to performing the requested query, Myria logs analytics data to local instances of the Postgres relational database. This analytics data powers an interactive profiling interface, so fast retrieval is a high priority. In the

| program | data structure | commits | LoC | bugs |
|---|---|---|---|---|
| Myria | analytics storage | 88 | 269 | 11 |
| Bullet | volume tree | 57 | 2582 | 15 |
| ZTopo | map tile cache | 15 | 1383 | - |
| Sat4J | var data | 22 | 22 | 7 |

Table 1: Subject programs. “Commits” is the number of commits in the program’s repository that directly relate to the data structure being replaced. “LoC” is the total number of lines of code for the data structure and associated helper methods. “Bugs” is the number of bugs reported in the program’s issue tracker for which the correct fix was in the implementation of the data structure. Note that ZTopo does not have a dedicated issue tracker.

| program | time (s) | spec | tuning | LoC delta |
|---|---|---|---|---|
| Myria | 30* / 45 | 22 | 65 | +37 |
| Bullet | 30* / 54 | 23 | 123 | -603 |
| ZTopo | 30* / 17 | 25 | 75 | +1 |
| Sat4J | 1 / 6 | 11 | 32 | -58 |

Table 2: Work we performed to synthesize and integrate replacement data structures for each project. “Time” shows how long Cozy spent synthesizing the structure, separated by synthesis time on the left of the “/” and auto-tuning time on the right. Asterisks (*) indicate runs that were capped at a timeout of 30 seconds; if allowed to run beyond that point, Cozy does not terminate in less than an hour. “Spec” shows the number of lines in the Cozy specification. “Tuning” shows the number of lines in the auto-tuning benchmark. “LoC delta” shows the net change on the codebase as a result of our refactoring, not including lines of specification or lines in the synthesized data structure.

analytics store, each entry has a query ID (what query it came from), a sub-query ID (for compound queries), a fragment ID (what part of the execution pipeline caused it), a start time, an end time, and additional information about what transpired. The interactive profiling interface needs to quickly find analytics entries for a given query and fragment within a given time range.

Bullet is a real-time physics simulation library implemented in C++ and commonly used in visual effects work. The library offers a collision detection API. Before doing fine-grained collision detection, Bullet searches for coarse-grained collisions between bounding boxes using a “fast dynamic bounding volume tree (DBVT)”—a custom data structure. The data structure’s most important function finds bounding boxes intersecting a given query bounding box. We replaced Bullet’s handwritten structure with one synthesized by Cozy.

ZTopo is a topological map viewer implemented in C++. Topology data comes from an online database; to avoid downloading the entire database every time the user opens the application, the data is divided into tiles that are downloaded on-demand and kept in memory and disk caches. To maintain its cache hierarchy, ZTopo stores tiles with a given $x$ and $y$ coordinate and tracks what state they are in (in memory, on disk, or available over the network). The data structure keeps tiles organized according to their state. A replacement data structure was also synthesized for ZTopo in previous work [17].

Sat4J is a boolean satisfiability solver implemented in Java. It takes as input a formula over boolean variables and finds a satisfying assignment (if one exists). During solving, Sat4J tracks miscellaneous data about each variable appearing in the formula. The data is specific to the implementation of the solver and includes an assignment level, a cause constraint, collections of listeners watching for changes on the variable, and a collection of “undos” that can be used for rewinding to an earlier solver state. This data structure needs to be able to efficiently look up a variable’s information by its integer identifier.

## 4.2 Integration Methodology

The central data structure in each program implements a collection as its core functionality, but also includes many application-specific methods. For example, Bullet’s DBVT implementation includes code for computing collisions between two trees and code for estimating the future locations of bounding boxes based on velocities. Cozy can replace the collection functionality, but not the application-specific methods.

Therefore, our methodology when replacing each data structure was to replace its internals with a Cozy collection keep but the data structure’s public interface intact, reimplementing their functionality on top of Cozy’s implementation.

Table 2 summarizes the amount of work we performed in order to replace each data structure’s internals with a collection synthesized by Cozy. The collection specifications were easy to write and are very short, on the order of twenty lines. We implemented an auto-tuning benchmark for each synthesized collection that benchmarks on random data. The final code size after integrating the new collections is not always shorter; Myria and ZTopo use high-level database and collections libraries already, so the integration of the synthesized collection does not save any lines. On the other hand, the core handwritten algorithm and tuned memory management employed by Bullet can be completely removed, resulting in over 600 lines of net savings. Sat4J’s variable metadata structure is not complex, but it includes a fair amount of logic for reallocating storage when the number of variables exceeds the size of any allocated arrays; all of that logic can be removed.

We were very careful to preserve correctness and the original public interface while doing the refactoring, so we were not as aggressive in saving lines as we might have been. We believe that, if this methodology were used from the start of development, the original application developers could use

Cozy to save many more lines in each codebase. Furthermore, the developers would not have had to fix the bugs noted in Table 1 and subsection 4.3.

## 4.3 Correctness

To measure the extent to which Cozy helps achieve correctness, we examined bug reports for each subject program. We show how Cozy addresses potential correctness issues by generating correct-by-construction code.

***Myria*** The Myria issue tracker lists 11 bugs directly related to analytics storage. Since the existing implementation of the Myria analytics store uses Postgres, none of the Myria bugs refer to incorrect functional behavior. Instead, they relate to the interface between the Java code and Postgres, e.g., tables not being created during initialization or improperly escaped query strings. Our synthesizer builds a complete Java class with insert, remove, and query methods, thereby alleviating the need to reason about this interface.

***Bullet*** The Bullet issue tracker lists 12 bugs related to the implementation of the DBVT, and the file's source history in the Bullet repository reveals 3 commits that reference unreported bugs. The bug reports span nondeterminism, cross-platform portability, and memory mismanagement. Nondeterministic behavior made testability a concern. The original implementation uses some platform-dependent features (in particular, vector intrinsics) that make portability a problem. Code style problems such as stray semicolons have also caused portability problems. The data structure also has its own memory allocation logic, which has led to memory leaks and unexpected behavior in the past. Our synthesized implementation avoids these problems: it uses only standard C++ language features and uses the system memory allocator. We fuzz-tested the synthesized implementation and found no nondeterminism or memory bugs.

***ZTopo*** ZTopo was written by a single programmer, and so it has no dedicated bug tracker. Instead of tests, the code contains many assertions to verify that each entry's position in the data structure reflects the value of its state field. Cozy guarantees these properties in the generated data structure, so the complex data structure logic that ensures them—as well as the asserts themselves—can be removed without sacrificing correctness.

***Sat4J*** Sat4J's bug tracker lists 7 issues directly related to its storage of variables. Three of these relate to performance. The original implementation stores the data in arrays that grow when new variables are added to the formula, and the original implementation did not grow their sizes exponentially, resulting in $O(n^2)$ time to add $n$ variables to the formula. Cozy removes the need to implement this logic by hand. The other 4 bugs were correctness problems that Cozy's generated structure does not suffer from. Sat4J has an extensive test suite, and our synthesized implementation passes all available regression tests.

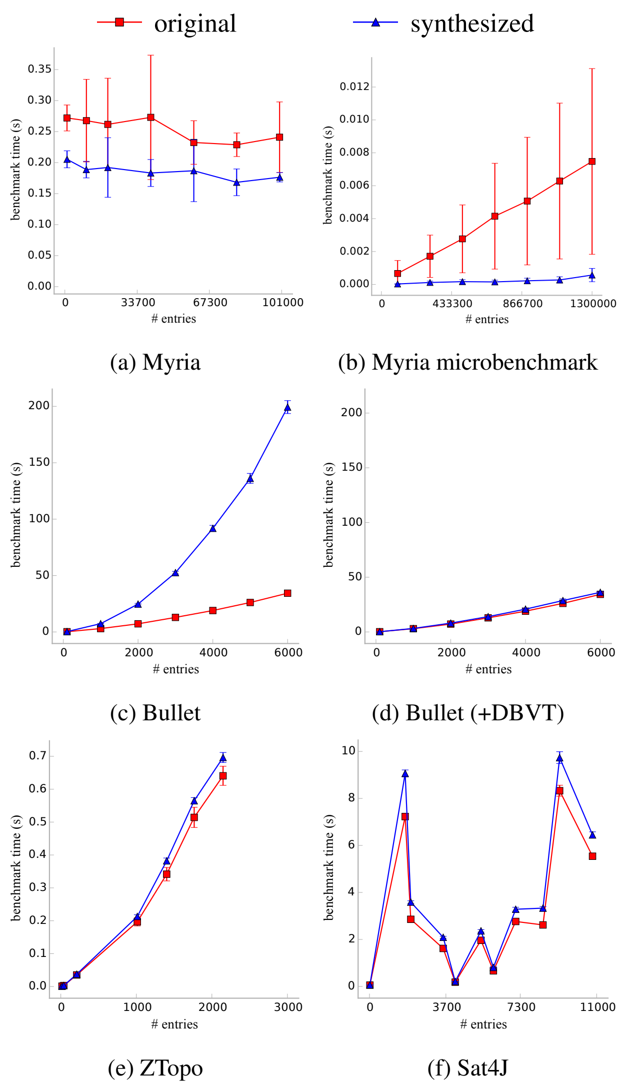

Figure 11: Benchmark results. Lower is faster and better. Each point represents the average of 30 runs, with error bars showing the standard deviation. The individual benchmark tasks are described in text. All benchmark data was collected on a Macbook Pro with a two-core 2.5 GHz processor and 8 Gb of RAM.

## 4.4 Performance

Figure 11 plots the difference in performance between the handwritten and synthesized data structure implementations. For each project, we found or created a benchmark program. Each benchmark program has a parameter for tuning the size of the workload, and for each run we recorded the maximum size of the data structure during execution and the amount of time the whole benchmark program took to run. All of the benchmarks primarily measure retrieval time; insertion time is either negligible (for ZTopo) or negligible and not included (for Myria and Bullet).

***Myria*** The benchmark uses a real-world dataset and query frequently used by the Myria developers for testing. The dataset describes relationships between Twitter users. The benchmark first generates a large amount of analytics data by repeatedly issuing queries to find cliques in the data. It then issues requests for various amounts of analytics data about the queries, measuring how much analytics data is present in the database and how long it takes to retrieve a small subset. For both implementations, the overhead incurred due to the data structure while generating analytics data is negligible compared to the normal query execution time. We found that the synthesized implementation is generally faster than the original implementation. Two factors contribute to the speedup: the constant overhead introduced by Postgres in the original implementation, and the asymptotic speedup that Cozy achieves by using augmented binary search trees to store intervals. In the original Postgres implementation, entries are indexed by start time, but the query planner is unable to exploit an index on end time as well. In contrast, the synthesized implementation stores the maximum end time of any entry in every subtree, and so it is able to exploit information about end times to prune the result set faster.

Due to high overhead in the Myria system and various confounding factors such as garbage collection interruptions, the data in Figure 11a is very noisy. To better illustrate the difference between Cozy's synthesized implementation and the Myria builtin implementation, we separated Myria's existing data structure implementation from the codebase and performed a microbenchmark on random data. We ran Postgres on an entirely in-memory disk to exclude disk performance factors. Figure 11b shows the results of the microbenchmark. The asymptotic wins from the synthesized structure as element count increases are clearly visible. The Postgres query plans are worst-case linear time, and the wide standard deviation reflects the variability in performance with different random data and random queries.

***Bullet*** In the past, the Bullet developers have used a benchmark suite called CDTestFramework to evaluate their collision detection code against other libraries [8]. CDTestFramework constructs an animated scene out of many moving boxes and times how fast the collision detection module can find all the intersections between boxes at each frame. We adapted CDTestFramework to test our modified implementation of Bullet against the original version. We found that the synthesized implementation is consistently slower by a factor of six to seven. This reveals a limitation in Cozy: the DBVT is not in its set of known representations, nor is anything that might have comparable performance on this benchmark.

The DBVT is a space-partitioning tree that groups boxes spatially according to their Cartesian distance to each other. Nodes are separated into leaf nodes which each store a single box and internal nodes which store the total enclosing volume of their entire subtree. By contrast, the synthesized structure is an augmented binary search tree. It organizes boxes by starting y-coordinate only, and each node stores a volume enclosing its subtrees. Although the augmented volume data allows the synthesized structure to skip many subtrees during iteration, in practice we found that it results in many false-descents into subtrees with no matching nodes. The original implementation makes almost no false-descents. On this workload, the synthesized structure visits roughly ten times as many nodes as it actually returns, which explains the difference between the original and synthesized implementations. In this case, the synthesized implementation would be most useful for rapid prototyping and testing.

For comparison, we also added Bullet's DBVT into Cozy as another choice the auto-tuner can make. Figure 11d shows the same benchmark with the DBVT enabled. The auto-tuner easily discovers that the DBVT structure is better suited to this problem and so the resulting structure easily matches the performance of the original implementation. This improvement did not require any change to the specification of the data structure or to the Bullet codebase. The only observable change after adding the DBVT to Cozy—besides increased performance—was that auto-tuning took four seconds longer than the 54 seconds reported in Table 2.

***ZTopo*** ZTopo has no publicly-available benchmarks, so we constructed a benchmark workload by recording all the calls made to the cache module during several minutes of normal usage. The benchmark replays a prefix of the recorded calls, measuring the total time taken and how large the structure got during execution. The handwritten implementation and the synthesized implementation behave nearly identically because they are implemented nearly identically: both have a hash map for organizing tiles by key and a series of linked lists for organizing tiles by state.

***Sat4J*** We benchmarked Sat4J on several examples from the SAT Competition's 2002 industrial benchmark suite [24]. We chose the examples by randomly selecting several different problems of different sizes from among the examples that Sat4J terminates on. With the synthesized structure, performance on these problems is a constant factor of 20–25% slower than the original implementation. However, performance is still dominated far more by the input formula than by the exact implementation. The performance gap exists because the original implementation is a structure of arrays and Cozy's implementation is an array of structures. In the original implementation, each query is answered by a single array lookup, while in the synthesized implementation each query is answered by an array lookup followed by a pointer dereference.

## 5. Related Work

Several bodies of work relate to the approach outlined in this paper: previous work on synthesizing data structures, work on general program synthesis, work on generation of data structure instances, and work on database query planning and materialized views.

***Data structure synthesis*** Data structure synthesis has been an active area of research for many years. One class of techniques stems from Early's work on iterator inversion [13], in which high-level set comprehensions can be rewritten automatically to use more efficient data structures. Subsequent work generalized these ideas [15, 22]. The primary weakness of iterator inversion approaches is that they require manual construction of rewrite rules for different syntactic forms. The rules are difficult to write and even more difficult to prove exhaustive. The rewrite engine is fairly naive, and so performance gains are not guaranteed.

Automatic data structure selection has also been investigated for the SETL language [7, 25, 26]. This work differs from ours in that it relies on complex program analyses to bound the possible contents of each set or map. These bounds can then be used to select good implementations for the abstract set and map interfaces. In SETL some of this computation occurs at runtime, while our work requires a more precise specification than "set" or "map," and we can avoid extra runtime overhead. Automatic data structure selection has also been investigated for Java, where a dynamic analysis can select better implementations of abstract collection types [27].

Specialized languages for data structure implementation have also been proposed [3, 29, 30]. In this approach, programmers are given a set of composable primitives for describing the implementation of their data structures. However, the programmers still need to describe the desired implementation. In our work, Cozy generates the implementation automatically.

More recent work has identified ways to describe and synthesize data structures having complex data sharing using relational logic specifications [17]. The RelC tool described in that line of work is very similar to Cozy: both emit collections over a single type with add, remove, update, and query operations. RelC works by exhaustively enumerating candidate data structure representations. For each one, a well-formedness test is used to determine whether the representation can be used to correctly implement the data structure. For those that succeed, a query planner determines how to use the 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.

Cozy improves on that line of work by supporting a greater range of input specifications and by requiring fewer calls to the auto-tuning benchmark during synthesis. RelC did not support input specifications with inequalities, disjunctions, and negations, so RelC would not be able to synthesize implementations for the Myria or Bullet data structures. Cozy handles these efficiently by inverting the order of operations: instead of synthesizing a representation and using a query planner to obtain code, we synthesize a plan in the form of an outline and derive the representation from the outline *after* synthesis. This inversion helps here since query planners like the one employed by RelC require complicated handwritten rules to handle negation and disjunction.

Cozy does not make more than 12 invocations of the auto-tuning benchmark on any of our case studies, while RelC makes more than 80 in a typical case. The inverted order of operations helps here too: the performance of an outline is easier to predict than the performance of a representation, since intimate knowledge of the query planner is necessary for RelC to predict the performance of a representation. Having the plan up-front enables the use of a static cost model to guide the synthesizer toward more efficient solutions earlier.

RelC solves a slightly different problem than Cozy, since it was designed to find representations with a high degree of data sharing. We suspect that RelC's implementations are more memory-efficient than Cozy's, but we were not able to evaluate this since RelC is not publicly available. Additionally, follow-up work on RelC investigated synthesizing concurrent data structures [18], which we do not address here.

***Synthesis*** Our work builds on existing CEGIS techniques for synthesizing individual functions [31]. However, Cozy optimizes for cost rather than program size. Additionally, Cozy abstracts over the data structure representation, allowing for automatic representation selection. Traditional synthesis approaches require precise specification of all inputs and outputs, including the contents of memory.

More recently, researchers have developed cost-optimizing synthesis techniques. $\lambda^2$ [14] synthesizes functional programs over recursive data structures. It optimizes with respect to a monotonic cost model by enumerating programs in order of cost. Unfortunately, since the cost of a Cozy outline depends on the cardinalities of the sets returned by its subcomponents, the enumeration algorithm used in $\lambda^2$ is not applicable. SYNAPSE [4] requires a *gradient* function mapping from cost values to subspaces of the synthesis search space. The gradient function is a flexible way to encode a search strategy for the synthesizer. In contrast to this approach, Cozy simply enumerates candidates in order of size and prunes aggressively based on the cost model.

***Data structure generation*** Researchers have investigated automatic data structure generation for testing [5, 11, 16, 19]. This differs from synthesis; tools in this domain construct *instances* of data structures with nontrivial representations, but they do not construct *implementations* of those data structures. The instances are typically used for exhaustive testing of code that operates on complex data structures.

***Databases and query planning*** The implementation outlines presented in Figure 4 can be viewed as a planning language for executing retrieval operations. Our model of collection entries as bags of fields is deliberately reminiscent of relational logic. However, our problem differs from conventional database query planning in several ways. First, instead of using handwritten rewrite rules, Cozy uses inductive synthesis to find the optimal set of plans with respect to our static cost model. Second, Cozy is not optimizing a retrieval plan for a pre-selected representation. Instead, the tool is free to pick its own preferred representation.

Other database research has focused on automatically choosing data representations; most notably AutoAdmin [1, 9], which can propose indexes and materialized views to improve query execution performance. This work is closer to Cozy's domain, but is still constrained by disk requirements and the limited expressiveness of indexes.

Strategies for efficiently maintaining materialized views have also become popular [2]. A materialized view is an optimized data structure that keeps the result of a particular database query up-to-date even while the underlying table is updated. The key difference between materialized view maintenance and our work is the presence of run-time query variables. Materialized views only materialize exact queries with all values filled-in, while Cozy creates data structures to respond efficiently even when the exact query is not known in advance.

## 6. Conclusion

This paper presented novel techniques for data structure specification and synthesis, as well as an evaluation of those techniques. Our approach splits the synthesis into two stages: first choosing a high-level outline with good asymptotic performance on a static cost model, and then choosing a low-level implementation with good physical performance on a dynamic benchmark. Four real-world case studies suggest that data structure synthesis has the potential to save programmer development time by offering high performance, a clean interface, and a correct-by-construction guarantee.

## Acknowledgments

This material is based on research sponsored by DARPA under agreement numbers FA8750-12-2-0107, FA8750-15-C-0010, and FA8750-16-2-0032. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

## References

[1] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 496–505, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. ISBN 1-55860-715-3. URL `http://dl.acm.org/citation.cfm?id=645926.671701`.

[2] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *Proc. VLDB Endow.*, 5(10):968–979, June 2012. ISSN 2150-8097. doi: `10.14778/2336664.2336670`. URL `http://dx.doi.org/10.14778/2336664.2336670`.

[3] D. Batory, V. Singhal, and M. Sirkin. Implementing a domain model for data structures, 1992.

[4] J. Bornholt, E. Torlak, D. Grossman, and L. Ceze. Optimizing synthesis with metasketches. In *Proceedings of the 43rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, POPL '16, 2016.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 123–133, New York, NY, USA, 2002. ACM. ISBN 1-58113-562-9. doi: `10.1145/566172.566191`. URL `http://doi.acm.org/10.1145/566172.566191`.

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

[7] 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.

[8] CDTestFramework. CDTestFramework. `http://www.bulletphysics.org/mediawiki-1.5.8/index.php/CDTestFramework` (Retrieved October 29, 2015).

[9] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 146–155, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc. ISBN 1-55860-470-7. URL `http://dl.acm.org/citation.cfm?id=645923.673646`.

[10] Cozy. `http://cozy.uwplse.org`.

[11] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 185–194, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4. doi: `10.1145/1287624.1287651`. URL `http://doi.acm.org/10.1145/1287624.1287651`.

[12] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. URL `http://dl.acm.org/citation.cfm?id=1792734.1792766`.

[13] J. Earley. High level operations in automatic programming. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 34–42, New York, NY, USA, 1974. ACM. doi: `10.1145/800233.807043`. URL `http://doi.acm.org/10.1145/800233.807043`.

[14] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 229–239, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-

3468-6. doi: 10.1145/2737924.2737977. URL http://doi.acm.org/10.1145/2737924.2737977.

[15] 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. doi: 10.1145/800168.811544. URL http://doi.acm.org/10.1145/800168.811544.

[16] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in udita. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 225–234, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806835. URL http://doi.acm.org/10.1145/1806799.1806835.

[17] 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. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993504. URL http://doi.acm.org/10.1145/1993498.1993504.

[18] 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. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254114. URL http://doi.acm.org/10.1145/2254064.2254114.

[19] D. Marinov and S. Khurshid. Testera: A novel framework for automated testing of java programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, pages 22–, Washington, DC, USA, 2001. IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=872023.872551.

[20] D. Moritz, D. Halperin, B. Howe, and J. Heer. Perfopticon: Visual query analysis for distributed databases. *Computer Graphics Forum (Proc. EuroVis)*, 34(3), 2015. URL http://idl.cs.washington.edu/papers/perfopticon.

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

[22] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4 (3):402–454, July 1982. ISSN 0164-0925. doi: 10.1145/357172.357177. URL http://doi.acm.org/10.1145/357172.357177.

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

[24] SatCompetition. The international SAT competition. http://www.satcompetition.org/ (Retrieved February 3, 2016).

[25] 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. ISSN 0164-0925. doi: 10.1145/357133.357135. URL http://doi.acm.org/10.1145/357133.357135.

[26] 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. doi: 10.1145/512976.512981. URL http://doi.acm.org/10.1145/512976.512981.

[27] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 408–418, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542522. URL http://doi.acm.org/10.1145/1542476.1542522.

[28] 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. ISBN 978-1-4503-0443-6. doi: 10.1145/2025113.2025153. URL http://doi.acm.org/10.1145/2025113.2025153.

[29] M. Sirkin, D. Batory, and V. Singhal. Software components in a data structure precompiler. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, pages 437–446, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press. ISBN 0-89791-588-7. URL http://dl.acm.org/citation.cfm?id=257572.257671.

[30] Y. Smaragdakis and D. Batory. Distil: a transformation library for data structures. In *In USENIX Conference on Domain-Specific Languages*, pages 257–270, 1997.

[31] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2008. AAI3353225.

[32] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *SIGOPS Oper. Syst. Rev.*, 40(5):404–415, Oct. 2006. ISSN 0163-5980. doi: 10.1145/1168917.1168907. URL http://doi.acm.org/10.1145/1168917.1168907.

[33] 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. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462174. URL http://doi.acm.org/10.1145/2491956.2462174.

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