Generalized Data Structure Synthesis

Calvin Loncaric
loncaric@cs.washington.edu
Paul G. Allen School of Computer Science & Engineering
University of Washington
Seattle, WA, USA

Michael D. Ernst
mernst@cs.washington.edu
Paul G. Allen School of Computer Science & Engineering
University of Washington
Seattle, WA, USA

Emina Torlak
emina@cs.washington.edu
Paul G. Allen School of Computer Science & Engineering
University of Washington
Seattle, WA, USA

ABSTRACT

Data structure synthesis is the task of generating data structure implementations from high-level specifications. Recent work in this area has shown potential to save programmer time and reduce the risk of defects. Existing techniques focus on data structures for manipulating subsets of a single collection, but real-world programs often track multiple related collections and aggregate properties such as sums, counts, minimums, and maximums.

This paper shows how to synthesize data structures that track subsets and aggregations of multiple related collections. Our technique decomposes the synthesis task into alternating steps of *query synthesis* and *incrementalization*. The query synthesis step implements pure operations over the data structure state by leveraging existing enumerative synthesis techniques, specialized to the data structures domain. The incrementalization step implements imperative state modifications by re-framing them as fresh queries that determine what to change, coupled with a small amount of code to apply the change. As an added benefit of this approach over previous work, the synthesized data structure is optimized for not only the queries in the specification but also the required update operations. We have evaluated our approach in four large case studies, demonstrating that these extensions are broadly applicable.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; • **Software and its engineering → Source code generation**;

KEYWORDS

Program synthesis, automatic programming, data structures

ACM Reference Format:
Calvin Loncaric, Michael D. Ernst, and Emina Torlak. 2018. Generalized Data Structure Synthesis. In *ICSE '18: 40th International Conference on Software Engineering, May 27–June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180211

ICSE '18, May 27–June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180211

1 INTRODUCTION

Many programming tasks can be framed as data structure problems, especially in domains like user interfaces or web services where software must manage some internal state and also handle asynchronous events. Manually implementing complex application-specific data structures can be time-consuming and error-prone. Recent research seeks to automatically synthesize data structure implementations from high-level specifications, thus ensuring correctness and run-time efficiency with minimum programmer effort [11, 12, 16]. Existing techniques can synthesize only a narrow range of data structures—those that retrieve a subset of a single collection. Such a simple API limits their applicability, as real-world software often has more complex requirements.

For example, the chat server Openfire [13] uses a custom in-memory data structure to represent a many-to-many relationship between users and groups. This data structure needs to answer many different kinds of queries efficiently, such as which users belong to a given group or whether any two users share a group. It also needs to keep itself up-to-date as users and groups are added, removed, and renamed. Despite its complex implementation, Openfire's user management code has a simple specification. In general, data structure specifications are much smaller than their implementations because they do not need to manage memory or be algorithmically efficient.

This paper presents a new technique for data structure synthesis that overcomes many of the limitations of previous work. Our tool Cozy can synthesize implementations for complex multi-collection data structures—including those found in Openfire—from high-level specifications. Like previous work, a Cozy specification declares the *abstract state* (what information the data structure stores), *queries* (methods that perform pure computations on the state), and *updates* (methods that modify the state) that the data structure must support. Cozy then produces source code that developers can use right away.

In previous work, implementations for update methods were hard-coded [16] or derived using a set of hand-written rules [11]. In Cozy, update methods are synthesized rather than hardcoded. This enables Cozy to discover more specialized data representations than previous work, since Cozy can choose different representations depending on what kinds of updates appear in the specification.

Our technique iteratively improves the data structure specification using two cooperating components: a *query synthesizer* that selects a better representation and implementation for each query method and an *incrementalization* step that ensures the new representation is kept up-to-date when an update method is called. Crucially, the incrementalization step can produce specifications for new query operations to help implement the update procedure.

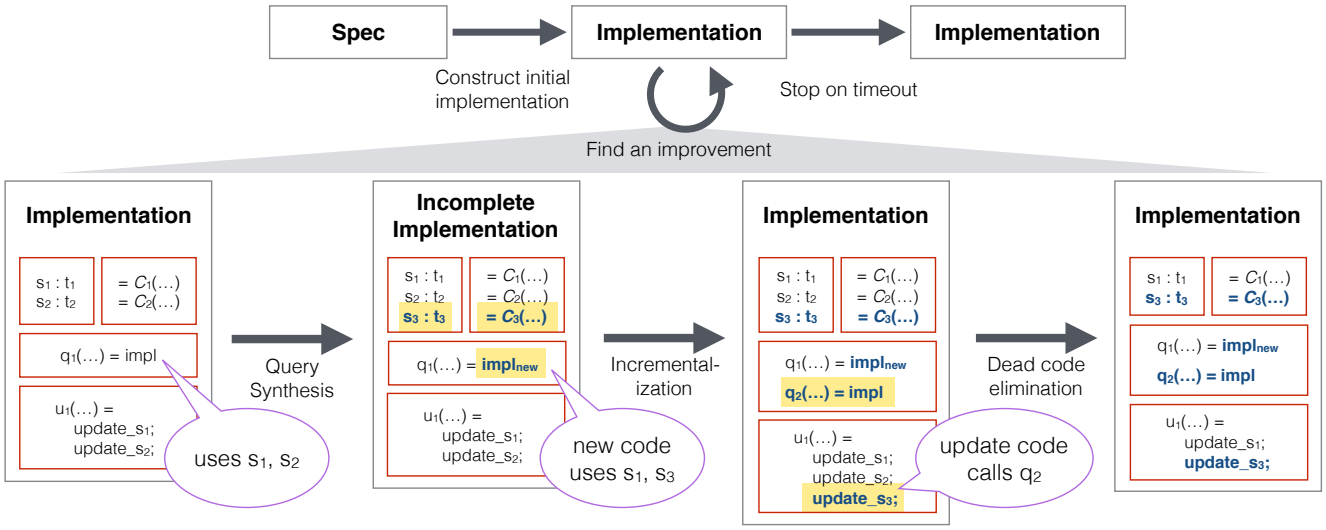

Figure 1: Architecture of Cozy. Each iteration through the loop performs query synthesis, incrementalization, and dead code elimination. Figure 2a shows example input, and Figures 2b and 3 show the corresponding output.

Cozy thus uses the query synthesizer to implement both pure query operations and imperative updates. Our technique is agnostic to the exact implementations of the query synthesizer and incrementalization step; Section 3 gives a detailed explanation of the concrete choices we made for Cozy.

The query synthesizer and incrementalization step interact using *concretization functions*. A concretization function expresses a data structure's representation—its concrete state—as a function of its abstract state. For example, the following concretization function represents the count of elements in an abstract set S: $C(S) = \sum_{x \in S} 1$. Concretization functions allow Cozy to reason about the effects of updates in pure mathematical terms. The imperative operation `S.remove(e)`—which removes an instance of e from S if any is present—causes a change to the data representation. The new value of the count thus becomes $C(S') = C(S - \{e\}) = \sum_{x \in (S - \{e\})} 1$.

Cozy's query synthesis step outputs both an efficient implementation for each query and a set of concretization functions indicating how the data should be represented. The incrementalization step then uses the concretization functions to produce a specification of the change to the concrete state as a result of each update. For the case of `S.remove(e)`, the change specification is the amount by which the count changes: $C(S') - C(S)$. These change specifications are queries over the abstract state of the data structure, and to implement them Cozy repeats the query synthesis step. The tool proceeds in this loop until exhausting its time budget—three hours for our evaluation.

Contributions

- A high-level data structure synthesis algorithm with alternating steps of query synthesis and incrementalization (§2).
- Query synthesis and incrementalization algorithms (§3).
- An implementation, called Cozy (https://cozy.uwplse.org).
- Four real-world case studies that evaluated Cozy's effect on development time, correctness, and efficiency (§4).

2 OVERVIEW

This section illustrates Cozy's high-level algorithm using a simplified example of a real-world data structure from Openfire. The data structure that manages users' contacts has been a frequent source of bugs (Section 4.4). Cozy can synthesize a complete implementation for Openfire's data structure given its specification.

Cozy uses the algorithm shown in Figure 1. It takes as input an executable specification of the data structure (Section 2.1), constructs an initial implementation (Section 2.2), and then iteratively improves its implementation using alternating steps of query synthesis (Section 2.3) and incrementalization (Section 2.4). A dead code elimination step (Section 2.5) prunes dead code as synthesis progresses.

2.1 Specification

Figure 2a shows a complete Cozy specification for part of the Openfire contact management data structure. It would be used as the input to Figure 1. In the specification, `state` declarations describe the abstract state of the data structure, `query` declarations specify methods that compute values using the abstract state, and `op` declarations specify methods that alter the abstract state. Methods may also include assumptions (preconditions) about their inputs. In some cases, Cozy can produce better implementations by leveraging these assumptions, but they are optional for specification writers. Callers must ensure that the assumptions hold at each call site.

In Openfire, users' contact lists are implicit and are computed based on the groups that each user belongs to. The data structure must be able to efficiently answer the question "should user u_1 appear in the contacts of user u_2?" for any u_1 and u_2. The query method `visible` in Figure 2a defines this relationship: u_1 is visible to (*i.e.* appears in the contacts of) u_2 if there exists a group g of which u_1 is a member and either g has been made visible to everyone (`g.visibility == Everyone`) or u_2 is also a member of g.

```
// Abstract state
state users : Set<User>
state groups : Set<Group>
state members : Set<(User, Group)>

// Query definition
query visible(u1, u2 : User):
  assume u1 ∈ users
  assume u2 ∈ users
  return (exists [ g | g ← groups,
    (u1, g) ∈ members and (
      g.visibility == Everyone or
      (u2, g) ∈ members) ])

// Update operation definition
op join(u : User, g : Group):
  assume u ∈ users
  assume g ∈ groups
  assume (u, g) ∉ members
  members.add((u, g))
```

(a)

```
// Users who are ∈ some group whose
// visibility is "Everyone"
state s1 : Map<User, Bool>

// Map from users to the groups of
// which each one is a member
state s2 : Map<User, Bag<Group>>

// Map from (user, group) tuples to
// boolean indicating whether that
// user is a member of that group
state s3 : Map<(User, Group), Bool>

// Now-unused state
state v1 : Set<User>
state v2 : Set<Group>
state v3 : Set<(User, Group)>

// New code
query visible(u1, u2 : User):
  return (s1[u1] or
    exists Filter_{λg.s3[(u2,g)]} s2[u1])
```

(b)

C_{s1}(users, groups, members) =
  MakeMap$_f$ users
  where
    f = λu . exists Filter$_{p(u)}$ members
    $p(u)$ = $\lambda(v, g)$. u == v and
        g.visibility == Everyone

C_{s2}(users, groups, members) =
  MakeMap$_g$ users
  where
    g = λu . Filter$_{q(u)}$ members
    $q(u)$ = $\lambda(v, g)$. u == v and
        g.visibility ≠ Everyone

C_{s3}(users, groups, members) =
  MakeMap$_{\lambda m.\text{true}}$ members

C_{v1}(users, groups, members) = users
C_{v2}(users, groups, members) = groups
C_{v3}(users, groups, members) = members

(c)

Figure 2: (a) An example input to Figure 1 that specifies the Openfire user and group management data structure. (b) New implementation of `visible` after several query synthesis steps. Cozy does not produce the comments, which we added for clarity. Since incrementalization and dead code elimination have not yet run, the implementation does not properly update the new state variables s_1, s_2, and s_3 and still contains some unused state variables. (c) Concretization functions for the new implementation.

This example has been simplified; our experiments (Section 4) use a full specification of the data structure that also includes explicit contacts and additional visibility modes for groups.

As specified, `visible` runs in $O(|\texttt{groups}| \times |\texttt{members}|)$ time. Cozy creates a more efficient implementation for `visible` (Figure 2b) that runs in $O(g)$ time, where g is the maximum number of groups that any one user is a member of.

2.2 Initial Implementation

Whenever Cozy chooses a data representation, it also creates, for each field in the representation, a *concretization function* that computes the field's representation from the abstract state. Since Cozy specifications are executable, they can be converted to implementations whose concrete state is the same as the abstract state. For the specification in Figure 2a, Cozy's initial implementation has the variables v_1, v_2, and v_3 and trivial concretization functions:

C_{v1}(users, groups, members) = users
C_{v2}(users, groups, members) = groups
C_{v3}(users, groups, members) = members

Each query and update operation can be rewritten in terms of v_1, v_2, and v_3 by simple substitution. The `visible` method becomes

```
query visible(u1, u2 : User):
  assume u1 ∈ v1
  assume u2 ∈ v1
  return (exists [ g | g ← v2,
    (u1, g) ∈ v3 and (
      g.visibility == Everyone or
      (u2, g) ∈ v3) ])
```

While renaming the abstract members to v_1, v_2, v_3 does not functionally change the specification, it creates initial concretization functions for later steps to consume.

2.3 Query Synthesis

Cozy synthesizes an implementation by iteratively finding improvements to the data structure. The query synthesis step in Figure 1 makes an improvement to some non-deterministically chosen query operation on the data structure. Section 3.2 discusses how Cozy makes the choice and the improvement.

Figure 2 shows output from one of the query synthesis steps, *i.e.*, an improvement to the query `visible` that uses a new representation and has associated concretization functions.

The query synthesis step may introduce new state variables, but it does not drop unused ones. In Figure 2, the red state variables s_1, s_2, and s_3 are new; v_1, v_2, and v_3 are now unused. The dead code elimination pass will eliminate the unused variables later.

The new variables' concretization functions are more complex than the trivial ones introduced for the initial implementation. The state variable s_1, for instance, has the concretization function C_{s1}, which uses the MakeMap primitive to construct a new map from users to Booleans. The MakeMap primitive takes a collection of keys (`users`) and a value function (f) and builds a map where each key $u \in$ `users` is associated with value $f(u)$. For s_1, the value is true if the user is a member of a group with visibility set to "Everyone". In Cozy, maps are total. Lookups on missing keys return a default value: false for booleans, the empty set for sets, and so on. Thus,

```
op join(u : User, g : Group):
  join_s1(u, g)
  join_s2(u, g)
  join_s3(u, g)
  join_v1(u, g)
  join_v2(u, g)
  join_v3(u, g)

private op join_s1(u : User, g : Group):
  for k ∈ altered_keys_s1(u, g):
    s1[k] = new_value_for_key_s1(k, u, g)

// The join_s2 and join_s3 implementations have
// been omitted for brevity.

private op join_v1(u : User, g : Group): // no-op
private op join_v2(u : User, g : Group): // no-op
private op join_v3(u : User, g : Group): v3.add((u, g))
```

(a)

```
// Find keys of map s1 whose values
// change when user u joins group g.
private query altered_keys_s1(u : User, g : Group):
  assume u ∈ users
  assume g ∈ groups
  assume (u, g) ∉ members
  return [ k | k ← MapKeys(s1) ∪ MapKeys(s1'),
    s1[k] ≠ s1'[k] ]

// Compute a new value at key k ∈ s1 when user u joins group g.
private query new_value_for_key_s1(k, u : User, g : Group):
  assume u ∈ users
  assume g ∈ groups
  assume (u, g) ∉ members
  assume s1[k] ≠ s1'[k]
  return s1'[k]

// Sub-queries for s2 and s3 have been omitted for brevity.
```

(b)

Figure 3: (a) Implementation of the `join` update operation and (b) new sub-queries that need to be synthesized. The variable `s1′` is defined as the new value of `s1` after `join` is called: $C_{\texttt{s1}}(\texttt{users}', \texttt{groups}', \texttt{members}')$.

the expression $\texttt{s1}[u_1]$ efficiently determines whether user u_1 is a member of a group with visibility set to "Everyone".

The concretization functions shown in Figure 2c will become the implementation of the constructor for the data structure. The constructor takes the abstract state as input and initializes the concrete state. Furthermore, the concretization functions enable incrementalization.

2.4 Incrementalization

The query synthesis step creates an incorrect data structure: the new state variables s_1, s_2, and s_3 are not kept up-to-date when `join` is called. The incrementalization step restores correct functioning by adding code to `join` that updates the new state variables. The new code must preserve the concretization functions in Figure 2c.

A simple but inefficient solution would be to recompute the value of each concrete state variable from scratch. Because an update usually makes a small change to the abstract state, Cozy produces an *incremental* update that makes small changes to the concrete state in response to a small change to the abstract state.

To incrementally update the concrete state, Cozy rephrases the update procedure as a set of queries that compute what changes should take place, plus a simple hardcoded snippet that applies those computed changes. A previous approach applied this same idea to synthesize remove operations [11], but with concretization functions it can be generalized to insertions and other updates as well. Our approach also allows for more complex update procedures like those that apply multiple changes at once or only make a change under certain conditions.

Figure 3a shows the code that Cozy produces to update the concrete state as a result of a user joining a group. Each concrete state variable gets its own update procedure (*e.g.* `join_s1` for `s1`). The code for `join_s1` is not synthesized; it comes from a lookup table (Section 3.3). However, the new code uses two fresh query operations `altered_keys_s1` and `new_value_for_key_s1` (Figure 3b) that determine what changes to apply. The former computes the set of map keys whose values change, and the latter computes the new value for each key. These two queries are added to the data structure specification, and thus they will be optimized by the query synthesizer on subsequent iterations.

The definitions of the fresh queries make use of both the old value of `s1` and the new value `s1′`. The new value is computed using the specification of `join` and the concretization functions. Mathematically, `join` sets the abstract state to

$$\texttt{users}' = \texttt{users}; \quad \texttt{groups}' = \texttt{groups}; \quad \texttt{members}' = \texttt{members} \cup \{(u, g)\}$$

and thus the new value `s1′` must be

```
s1' = C_s1(users', groups', members') =
  MakeMap_f users
  where
    f = λu . exists Filter_p(u) (members ∪ {(u, g)})
    p(u) = λ(v, g) . u == v and g.visibility == Everyone
```

Figure 3b shows the specifications for `altered_keys_s1` and `new_value_for_key_s1`, which are inefficient. On later iterations, Cozy's query synthesizer discovers efficient implementations for both. Specifically, Cozy implements `altered_keys_s1` to return the singleton set $\{u\}$ if g has visibility `Everyone` and u is not already in such a group, or $\emptyset$ otherwise. Cozy implements `new_value_for_key_s1` to simply return true.

The implementations of `altered_keys_s1` and `new_value_for_key_s1` do not require additional concrete state. In general, however, new concrete state might be generated for the fresh queries in later iterations, requiring another phase of incrementalization.

2.5 Dead Code Elimination

At each iteration, Cozy cleans up unused state variables and operations. For instance, the state variable v_2 can be eliminated since it is never read. All code that keeps v_2 up-to-date can be eliminated as well. Cozy also deduplicates state variables and fresh queries. Duplicates happen in cases where the same concrete state is useful to multiple different query operations.

spec	::=	*name* : $s_1, s_2, \ldots$ invariant e $m_1, m_2, \ldots$	specifications
s	::=	$x : \tau$	abstract state
τ	::=	Int | Bool | String	basic types
	|	Enum $\{case_1, case_2, \ldots\}$	enumerations
	|	$\langle \tau_1, \tau_2, \ldots \rangle$	tuples
	|	$\{f_1 : \tau_1, f_1 : \tau_2, \ldots\}$	records
	|	Bag$\langle \tau \rangle$	bags (multisets)
m	::=	query $q(args\ldots)$: assume e; return e;	queries
	|	op $u(args\ldots)$: assume e; *stmt*;	updates
stmt	::=	$x \leftarrow e$	assignment
	|	x.add(e)	insertion
	|	x.rm(e)	deletion
	|	if e : *stmt*	conditional
	|	*stmt*; *stmt*	sequencing
e	::=	x	variables
	|	$e == e \mid e < e \mid \ldots$	comparisons
	|	$e \wedge e \mid e \vee e \mid \neg e$	bool operations
	|	$e\ ?\ e\ :\ e$	conditionals
	|	$e + e \mid e - e$	arithmetic
	|	$(e, e, \ldots) \mid e.n$	tuples
	|	$\{f : e, f : e, \ldots\} \mid e.f$	records
	|	$\emptyset \mid \{e\} \mid e \cup e \mid e - e$	bag operations
	|	$\text{Map}_f\ e \mid \text{Filter}_f\ e$	map and filter
	|	$\text{FlatMap}_f\ e$	map union
	|	$\Sigma\ e$	sum
	|	Distinct e	remove duplicates
	|	$\text{ArgMin}_f\ e \mid \text{ArgMax}_f\ e$	min and max
f	::=	$\lambda x.e$	lambda abstraction

Figure 4: Core specification language *spec*.

3 DETAILS

Cozy iteratively improves a specification (Section 3.1) to produce an implementation. At each iteration Cozy attempts to find an improvement to some query (Section 3.2). The improvement may require new concrete state, which must be properly maintained in each update method (Section 3.3). Finally, unused state and code are removed (Section 3.4).

3.1 Specification and Output Languages

Figure 4 shows the core specification language. All input specifications are desugared to this core language (Figure 5). Cozy's output language is a superset of its input language that includes additional constructs for maps:

$$\tau ::= \text{Map}\langle \tau, \tau \rangle$$
$$e ::= \text{MakeMap}_f\ e \mid \text{MapKeys}\ e \mid e[e]$$

$$\begin{aligned}
len(X) &\rightarrow \Sigma\ \text{Map}_{\lambda x.1}\ X \\
empty(X) &\rightarrow len(X) = 0 \\
areUnique(X) &\rightarrow X = \text{Distinct}\ X \\
\forall x \in X, p(x) &\rightarrow empty(\text{Filter}_{\neg p}\ X) \\
\exists x \in X, p(x) &\rightarrow \neg empty(\text{Filter}_p\ X) \\
x \in X &\rightarrow \exists y \in X, y = x \\
[f(x) \mid x \in X, p(x)] &\rightarrow \text{Map}_f\ \text{Filter}_p\ X \\
[f(x,y) \mid x \in X, y \in Y] &\rightarrow \text{FlatMap}_{\lambda x.\text{Map}_{\lambda y.f(x,y)}\ Y}\ X
\end{aligned}$$

Figure 5: Expressions that Cozy accepts in input specifications but desugars into simpler forms. Cozy supports arbitrary list comprehensions, though only two examples of desugaring list comprehensions are shown.

Maps could be included in the input language, but they are not needed: a comprehension can group and look up values in a declarative rather than procedural manner. This clarifies what each expression computes and reduces the number of invariants that programmers need to maintain. In the output language, the MakeMap primitive takes an expression e representing the keys of the map and a projection f that gives the value at each key. MapKeys returns the keys of a map. The map index operator $e[e]$ returns the value of a given key in the given map. If the key is not in the map, this operator returns a default value; *e.g.* false for booleans and the empty set for bags.

We plan to extend Cozy with additional primitives for heaps, trees, and other efficient data structures in the future. For the case studies we examined, maps alone are sufficient to discover efficient implementations.

3.2 Synthesis

Cozy attempts to synthesize a better implementation for each query method in the specification, in parallel, with one thread per query. A static cost model (Figure 6) defines "better." Whenever a thread discovers a better implementation: (1) That implementation is immediately passed through the incrementalization step, and new queries it produces get new threads. (2) The whole specification undergoes dead code elimination, and any old queries that were eliminated have their threads terminated.

Each thread synthesizes improvements for its query using enumerative synthesis, an optimized form of brute-force search. The core algorithm described here was pioneered by previous work [3, 28, 31], but Cozy employs several novel improvements. We describe the core algorithm first, followed by our extensions.

Enumerative synthesis explores every possible expression in Cozy's output grammar, in order of size from smallest to largest. For each expression, a verifier (*e.g.* Z3 [6]) checks whether the expression satisfies the specification—that is, they always produce the same result. If so, the expression is emitted. Then the search continues to look for an even better solution. Since Cozy employs bounded verification (described below), the verifier always produces a result and never times out or returns unknown.

To make the search feasible, Cozy employs equivalence class deduplication [16, 31], an optimization that skips most expressions in the search space. The skipping is done safely so that Cozy never

State Expressions
$cost_S(e)$ = number of AST nodes in e

Query Expressions
$cost_Q(e) \mid e$ is a state expression $= 1$
$cost_Q(x) = 1$
$cost_Q(e_1 \ op \ e_2) = 1 + cost_Q(e_1) + cost_Q(e_2)$
$cost_Q(\text{Filter}_p \ e) = 1 + cost_Q(e) + card(e) \times cost_Q(p(x))$
(x is a fresh variable)
$cost_Q(\Sigma \ e) = 1 + cost_Q(e) + card(e)$
...

Facts About Cardinalities
$\forall e, card(e) \geq 0$
$\forall x, card(x) \geq 1000$ (if x is an abstract state variable)
$card(\emptyset) = 0$
$card(\{e\}) = 1$
$card(e_1 + e_2) = card(e_1) + card(e_2)$
$unsat(|e_1| > |e_2|) \rightarrow card(e_1) \leq card(e_2)$

Partial Order on Costs
$sat(c_1 < c_2) \wedge \neg sat(c_2 < c_1) \rightarrow c_1 \prec c_2$
(subject to the provable facts about all cardinalities in formulas c_1 and c_2)

Figure 6: Static cost model. In Cozy, costs are represented as symbolic formulas over the cardinalities of various collections. Cozy uses a solver (*sat* and *unsat* functions) to order costs.

misses a solution, if one exists. Equivalence class deduplication requires a list of example inputs. In Cozy, an example input consists of values for both the abstract state of the data structure and the query arguments. The example inputs are produced by the verifier: every time an expression fails verification, the verifier yields a new example input. Cozy caches built expressions. Whenever two expressions produce the same output on every example, Cozy consults a static cost model (described below) to decide which to keep. In this way, an expression's set of outputs on the examples puts it in an equivalence class, and only one representative of each equivalence class is cached at any given time. Larger expressions are only built out of those that survive this deduplication. Furthermore, Cozy only tries to verify expressions that produce correct output on every example, reducing the number of calls to the verifier. Since the skipping is so aggressive, the search must restart every time a new example is discovered to ensure that no solutions are missed.

Cozy includes three novel additions to the core enumerative synthesis algorithm: *query-time distinction*, a *symbolic cost model*, and *diversity injection*. Additionally, since verification is undecidable for our specification language, Cozy uses *bounded verification* instead of full functional verification. This technique was also employed by previous work [26].

Query-Time Distinction. Cozy's query synthesis algorithm must solve two intertwined problems: choosing a good representation for the data and choosing a good algorithm that exploits that representation. Our solution is to tag each node in a synthesized expression as either a *state expression* or a *query expression*. The data structure stores each state expression as a member and incrementally maintains it at each update operation. A *query expression* is evaluated each time the query is called.

For instance, an expression to compute the length of a list could be implemented in several different ways, depending on which parts are tagged as state expressions:

$$\underbrace{\Sigma \ \text{Map}_{\lambda x.1} \ S}_{\text{state}} \quad \text{or} \quad \Sigma \ \text{Map}_{\lambda x.1} \ \underbrace{S}_{\text{state}} .$$

The first case indicates that the data structure stores the length of S as a member and returns the stored value when the query is called. The second case indicates that the data structure stores S as a member and computes the length on-demand. Since these two expressions are equivalent, only the lower-cost one—in this case, the first—is kept during deduplication. Cozy's cost model does not account for the cost of maintaining the state; instead, that job is delegated to the sub-queries generated during incrementalization. Expressions that contain query arguments may not be tagged as state expressions, since those values will not be available until the query is executed.

Symbolic Cost Model. Figure 6 shows Cozy's novel static cost model. The cost model compares state expressions based on their complexity in terms of the number of AST nodes ($cost_S$). It compares query-time expressions based on their expected run time ($cost_Q$).

Cozy represents costs as symbolic formulas involving the cardinalities of various collections. For example, the cost of performing a filter includes the cost of evaluating the predicate on every element of the collection being filtered.

To determine the ordering between two costs c_1 and c_2, Cozy first makes solver calls to establish as many facts as possible about all the cardinalities (*i.e.*, calls to *card*) in each expression. Each call to *card* can then be replaced by a fresh real-type variable. Using these assumptions, Cozy then makes more solver calls. If there are cases where c_1 is less than c_2 ($sat(c_1 < c_2)$) and no cases where c_1 is more than c_2 ($\neg sat(c_2 < c_1)$), then the expression having cost c_1 should always be preferred over the expression having cost c_2.

Diversity Injection. In practice, the enumerative synthesis algorithm may take a long time to discover good solutions, especially for languages like ours where expression size is not strongly correlated with cost (that is, larger expressions may have lower cost). When the syntax tree for the best solution is of size fifteen or twenty, standard enumerative synthesis may take many centuries to discover it! For comparison, the syntax tree for the efficient implementation of `visible` in Figure 2 requires 45 nodes.

To bias the search toward useful expressions, Cozy employs a small number of handwritten *diversity rules* that inject new expressions into the search procedure. Whenever Cozy considers a new candidate expression, it also applies these rules and considers the resulting expressions. The diversity rules do not need to be universally correct or efficient: incorrect expressions will be rejected by the verifier, and inefficient expressions will be rejected by the cost model. However, incorrect expressions are still cached to help

Map Introduction

$\text{Filter}_{\lambda x.f(x)=y}\ X \rightarrow (\text{MakeMap}_{(\lambda k.\text{Filter}_{\lambda x.f(x)=k}\ X)}\text{Map}_{\lambda x.f(x)}X)[y]$

Cleaners

$\text{Filter}_{\lambda x.P_1(x)\wedge P_2(x)}\ X \rightarrow \text{Filter}_{\lambda x.P_1(x)}\ \text{Filter}_{\lambda x.P_2(x)}\ X$

$\text{Filter}_{\lambda x.a(x)?b(x):c(x)}\ X \rightarrow \text{Filter}_{\lambda x.a(x)\wedge b(x)}X + \text{Filter}_{\lambda x.\neg a(x)\wedge c(x)}\ X$

Relevant Subset

$X, v \mid v$ is a state variable $\rightarrow \text{Filter}_{\lambda x.x\in v}X$

Instantiation

$e_1, e_2 \mid v$ is free in $e_1 \rightarrow e_1[v \mapsto e_2]$

Figure 7: Cozy's diversity rules.

build larger expressions, as they might appear as subexpressions of correct solutions later on.

Cozy uses the five diversity rules shown in Figure 7. These diversity rules are specialized to Cozy's domain and are intended to capture some intuitions human programmers might apply. "Map introduction" converts some linear-time filter operations into efficient map lookups. "Cleaners" put expressions into normal form, which helps Cozy identify potential map lookups on later iterations. The "relevant subset" rule converts a collection into the subset that is already stored on the data structure. Finally, the "instantiation" rule helps transfer insights about a variable to insights about other expressions. For example, if Cozy has discovered the expressions $x \in S$ and y, then $y \in S$ might also be important.

In practice, Cozy's enumerative search machinery does not function well without the diversity rules and vice-versa. If the diversity rules are disabled, Cozy does not find a good solution to any specification for any of our subject programs within a three hour timeout. Similarly if the diversity rules are applied without the rest of Cozy's enumerative search machinery, the search quickly runs out of new expressions and stalls without ever finding a good solution.

Bounded Verification. It is undecidable to determine whether an expression in Cozy's language satisfies a specification. Thus, Cozy employs bounded verification: collection-type variables are limited to a fixed number of elements. In our experiments, we found a limit of four to be sufficient to ensure correct solutions. This may be thanks to the *small-scope hypothesis* [14], which proposes that most program bugs can be exhibited with small inputs. There is some evidence that the small scope hypothesis is true for simple programs [4], and we found it to be true in our domain as well.

3.3 Incrementalization

After query synthesis picks a new representation for the data, the incrementalization step restores proper functioning by adding code to keep that representation up-to-date as the data structure changes. Cozy's *incrementalize* procedure accomplishes that goal by leveraging the existing query synthesis procedure.

In `join` from Section 2.4, Cozy updated `s1` using the code

```
for k ∈ altered_keys_s1(u, g):
  s1[k] = new_value_for_key_s1(k, u, g)
```

Figure 8 shows the rules for Cozy's *incrementalize* procedure. Since `s1` has a map type, Cozy uses the *update sketch* shown in the figure for maps. An update sketch is a small snippet of imperative code that updates the variable. An update sketch may require new query

incrementalize(x, C_x):
Input: old abstract state σ and new abstract state σ'
Output: code to update concrete state x

Type	Update Sketch	New Queries
Int	$x \leftarrow x + q(\ldots)$	$q(\ldots) = C_x(\sigma') - C_x(\sigma)$

Bag	for $elem \in q_1(\ldots)$:
	  x.del($elem$)
	for $elem \in q_2(\ldots)$:
	  x.add($elem$)
		$q_1(\ldots) = C_x(\sigma) - C_x(\sigma')$
		$q_2(\ldots) = C_x(\sigma') - C_x(\sigma)$

Map	for $k \in q(\ldots)$:
	  incrementalize(
	  $x[k]$,
	  $\lambda\sigma.C_x(\sigma)[k]$)
		$q(\ldots) = \{k \mid$
		  $k \in \text{MapKeys}(C_x(\sigma)) \cup$
		  $\text{MapKeys}(C_x(\sigma'))$,
		  $C_x(\sigma)[k] \neq C_x(\sigma')[k]\}$

other	$x \leftarrow q(\ldots)$	$q(\ldots) = C_x(\sigma')$

Figure 8: Rules for *incrementalize*(x, C_x). C_x is the concretization function for x. To update a map-type variable, *incrementalize* is called recursively to determine how to update the value at each changed key.

operations in order to function. In the case of maps, the update sketch finds the keys whose values have changed and updates each one in the map. Cozy introduces the new query `altered_keys_s1` to compute which keys have changed.

Since the values in `s1` are booleans, Cozy uses the fallback sketch for "other" types to update each value. This rule uses a new query `new_value_for_key_s1` to compute—from scratch—a new value for `s1[k]`. As discussed in Section 2.4, the new value for `s1[k]` is simply `true`. In practice, new queries generated by *incrementalize* often have short and efficient implementations.

3.4 Dead Code Elimination

When a better query implementation is found, some state variables may go unused. The imperative operations that keep these variables up-to-date are unnecessary, as are any queries required only by those imperative operations, and so forth. The dead code elimination procedure is important; it frequently eliminates variables in this manner as better query solutions are found.

To clean up unused state and operations, Cozy uses mark-and-sweep. User-specified query operations start as roots. Any state that they use is marked as relevant, and code to update that state is also marked. Queries used by the update code are then marked, and so on until fixed point. Finally any unmarked state, queries, or update code can be safely removed.

3.5 Termination

The query synthesis procedure (Section 3.2) has no formal termination guarantees, and as a result, neither does Cozy itself. But since the input specification is executable, Cozy always has a correct solution and the synthesis process can be stopped at any time. Our experiments used a fixed timeout of three hours for synthesis.

4 EVALUATION

Cozy has three goals: to reduce programmer effort, to produce bug-free code, and to match the performance of handwritten code. We found that using Cozy requires an order of magnitude fewer lines of code than manual implementation (Section 4.3), makes no mistakes even when human programmers do (Section 4.4), and often matches the performance of handwritten code (Section 4.5).

4.1 Methodology

For each of four real-world programs (Section 4.2), we

(1) identified an important, complex, handwritten data structure,
(2) manually wrote a Cozy specification,
(3) allowed Cozy a three-hour timeout to synthesize a new implementation, and
(4) replaced the original data structure by the synthesized one.

Replacing handwritten code with Cozy-synthesized code required some light refactoring in each program. For example, some programmers intertwine data structure code with I/O code. We disentangled these, because Cozy does not synthesize I/O code. This refactoring was only necessary because these projects did not use Cozy from day one. Furthermore, we believe it results in better code style and easier-to-understand abstractions.

We ran our experiments on a machine with 96 cores and 512 Gb of memory. Cozy spawns one thread for each query in the specification and runs fastest on a machine with at least that many cores, but does not require it. The Openfire specification, our largest, has 12 query operations, thus requiring 12 cores for fastest operation. Memory usage steadily climbs the longer Cozy runs; we have observed it reach 32 Gb in the worst case.

The three hour synthesis time does not slow down the edit-compile-test cycle. Since Cozy specifications are executable, they can be immediately translated into usable but inefficient code. Developers can code and test against the slow version to gain confidence in their specification before running the full synthesizer. We made use of this feature while writing specifications in our evaluation.

4.2 Subject Programs

ZTopo [32] is a topological map viewer implemented in C++. Its cache of map tiles asynchronously loads map tiles over the network and caches them on disk or in memory. The cache enables any other part of the program to query for information about a given map tile. ZTopo was also a target for previous data structure synthesis work [11, 16]. Cozy is also able to synthesize two parts of the cache that previous work could not. First, Cozy can synthesize the code that accounts for the total disk and memory usage of cached map tiles. Second, Cozy synthesizes a key operation to look up a single element by its unique identifier. Previous tools implemented this operation inefficiently by checking whether a computed collection of results contained a single element or not.

Sat4j [17] is a Boolean satisfiability solver implemented in Java. Its variable store tracks, among other things, when a guess was last made about a variable's value and whether any listeners are watching that variable's state. Sat4j was also a target for previous data structure synthesis work [16]. As with ZTopo, Cozy's synthesized implementation of the Sat4j data structure is a closer match to the original than previous tools, requiring less wrapper code.

Table 1: Programmer effort. LoC measurements do not include comments or whitespace.

Project	Hand-written Span	Commits	LoC	Cozy LoC
ZTopo	1 week	15	1024	41
Sat4j	8 years	22	195	42
Openfire	10 years	47	1992	157
Lucene	13 years	20	68	36

Openfire [13] is a large, scalable IRC server implemented in Java. Its in-memory contact manager is extremely complex. Users' contacts can be either explicit (added by users manually) or implicit (present due to users' group memberships). Furthermore, the contact manager must keep its state in sync with the underlying database as users and groups are created, modified, and deleted. This logic has been a frequent source of bugs [30]. Openfire's implicit contacts require computing information about two distinct collections (users and groups), and thus cannot be handled by any previous tool.

Lucene [29] is a search engine back end implemented in Java. Lucene uses a custom data structure that consumes a stream of words and aggregates key statistics about them. The data structure has an `add` method that is called once for each token instead of getting the tokens as one big list. The logic for handling each token is tricky since the data structure needs to to be queryable between calls to its `add` method. Cozy helps avoid the logic in the `add` method by having a clean specification that describes the abstract state as a bag of tokens and descriptions of the queries that matter.

4.3 Programmer Effort

We do not know how much time programmers spent implementing and debugging the hand-written data structures, but it was significant. Table 1 shows the size of each implementation, in non-comment non-blank lines of code. It also reports how many commits contributed to the current version of the data structure implementation, and across how much time those commits were made. The long time periods are because Sat4j, Openfire, and Lucene are established projects and still undergoing active maintenance. In all three, however, bug fixes have been made to the data structure in the last five commits, indicating that full functional correctness has been difficult to achieve.

The Cozy specifications are an order of magnitude shorter than the manual implementations. Most of our time was spent reverse-engineering to understand the undocumented existing implementation; once we understood it, writing the specification was quick. For example, writing, integrating, and testing the ZTopo and Sat4j specifications took less than a day each. The Openfire roster manager was more challenging because we had to first formalize the implicit contacts function, a task the developers never carried out. We already understood the Cozy specification language (Section 3.1), but we believe that a programmer could learn it more quickly than it took us to reverse-engineer any one of the programs.

Because the specifications are shorter, simpler, and more abstract, they are much easier to understand. Programmers writing specifications are therefore less likely to make mistakes, and mistakes

Table 2: Correctness results. ZTopo has no dedicated issue tracker.

Project	Issues	New defects found
ZTopo	n/a	No
Sat4j	7	No
Openfire	25	Yes
Lucene	1	No

Table 3: Performance results. All times are in seconds.

Project	Time (orig.)	Time (Cozy)
ZTopo	5	5
Sat4j	53	61
Openfire	16	15
Lucene	9	9

will be easier to discover, diagnose, and correct. The specifications also serve as concise, unambiguous documentation.

4.4 Correctness

Cozy might produce an incorrect data structure because of its use of bounded verification. We also might have made an error when writing the specification. To check the correctness of the Cozy-synthesized data structures, we ensured that all tests in each project still pass. ZTopo, Openfire, and Lucene have no tests that cover the data structure we replaced. For these projects we verified that our synthesized data structure behaves identically to the original implementation during execution of the benchmarks we used in Section 4.5.

Table 2 lists how many data-structure-related issues in each project's respective issue tracker might have been prevented by Cozy. Most issues relate to defective update code putting the data structure in a bad state. Cozy is perfectly positioned to prevent those defects: changes to a data structure's abstract state are much easier to specify than the code that updates an optimized representation. We now discuss some of these issues.

Sat4j's variable metadata storage has suffered both performance and functional correctness issues in the past that Cozy avoids. Today Sat4j has a test suite that achieves 89% statement coverage on the data structure we replaced, and Cozy's synthesized implementation passes all tests.

Of Sat4j's seven reported issues, five relate to update code. Sat4j's data structure includes several arrays of data that grow exponentially as entries are added, and the logic to grow them and keep the capacity information up-to-date proved tricky to get right. The data structure also supports a `reset()` method to clear all of its internal state, but developers did not properly revise its implementation when they introduced new state variables. Cozy can prevent these kinds of problems since the programmer does not need to maintain the concrete representation.

Openfire, having a more complex data structure, has been even more difficult to get right. Section 2 presented only a simplified portion of the Openfire roster manager specification. The full specification has additional rules and visibility modes for groups. In particular, a user u_1 is visible to a user u_2 if any one of four different conditions are met: (1) the users have added each other as explicit contacts, (2) u_1 is in a group with visibility set to `Everyone`, (3) both users share a group with visibility set to `OnlyGroup`, or (4) u_1 is in a group g_A with visibility set to `OnlyGroup` and u_2 is a member of a group g_B configured to have visibility onto g_A.

This definition gives rise to two kinds of roster items: explicit items due to condition 1 and implicit items due to conditions 2–4. The manually written implementation makes a trade-off: all explicit items plus implicit items due to conditions 2 and 3 are held as concrete objects in memory, but implicit items due to condition 4 are constructed on-demand to save memory. Developers had to write a large amount of code to keep the implicit contacts correct when groups change visibility or when group membership changes. That code has been a frequent source of defects, and still has open issues. For example, one issue still open at time of writing reports that when administrators delete a user without first manually removing her from all of her groups, she remains in other users' contact lists.[1] Other issues were caused by the stored state of the roster getting out-of-sync with the abstract state of the roster. By contrast, a Cozy programmer does not need to write the update code; Cozy discovers its own data representation and determines how to update it in response to changes.

Additionally, we discovered multiple new failures while replacing the original implementation.[2] For example, the original implementation makes it possible to create a situation in which two users see different views of the roster: according to one user, both are visible to each other, while according to another, there is only a one-way visibility. The synthesized implementation does not suffer from these problems. We do not know how many source code defects contribute to the observed failures.

Even Lucene's small data structure has been a source of defects. Overlapping words caused some of its internal statistics to become corrupted because the original developers did not foresee this possibility. Our Cozy implementation handles this case gracefully; the natural way to specify Lucene's operations does not have the defect.

4.5 Performance

We measured the performance of the handwritten and synthesized implementations on realistic workloads. Table 3 reports the wall-clock time required to run each benchmark to completion. The benchmarks are end-to-end, and include application behavior in addition to the data structure itself; the resulting time, therefore, represents the overall effect on each program from using the synthesized data structure.

Our benchmarks for ZTopo and Sat4j are the same ones used to evaluate an earlier iteration of Cozy [16]. The ZTopo benchmark is a log of recorded application usage that we replay. The Cozy-synthesized implementation of the ZTopo tile cache matches the performance of the existing implementation almost exactly. The handwritten and synthesized implementations are conceptually identical: both store map tiles in linked lists grouped by tile type. The dominant factor affecting performance is the speed of finding tiles by unique ID, which both implementations do using a hash table.

[1] https://issues.igniterealtime.org/browse/OF-1121
[2] https://community.igniterealtime.org/thread/60317

Sat4j's benchmark suite consists of eleven randomly-selected input files from the 2002 boolean satisfiability solver competition [18, 22]. The synthesized data structure for Sat4j under-performs the existing implementation. The handwritten code exploits some facts about the data that Cozy does not know: in Sat4j, variable IDs can be used as indexes into an array since they always fall between zero and a known maximum bound. This interacts poorly with Cozy's total semantics for map lookups. At code generation time, Cozy must insert safety checks at every map lookup. In Sat4j those safety checks are unnecessary and harm performance substantially.

Our benchmark for Openfire is a replayed sequence of actions against its admin panel that offers direct access to the internal roster data structure, where users, groups, and explicit contacts can be modified. The synthesized structure improves performance slightly. There are several contributing factors, but the dominant one is that the synthesized data structure can avoid a number of expensive internal representation checks. To improve correctness, the handwritten implementation will often clean up its own state, which imposes some overhead. By generating correct code, Cozy avoids these internal checks.

Our benchmark for Lucene is a series of operations on artificial data. Cozy's synthesized data structure for Lucene is very similar to the manually written one, leading to identical performance.

5 RELATED WORK

The data structure synthesis problem dates to the 1970s and *iterator inversion*, a technique for constructing data structures to accelerate iterative operations [9, 10]. Our syntax for queries is similar to that found in Earley's work, although our techniques are substantially more powerful. Iterator inversion required handwritten rewrite rules, while Cozy's exhaustive search discovers complex transformations unaided.

The developers of the SETL language took a different approach by splitting it into a *pure* language and a *representation sub-language*. The sub-language specifies what structures to use when running pure code [8, 19, 20]. More recently, researchers have investigated dynamic techniques to achieve the same effect [21]. Beyond simply choosing better existing implementations of an interface, Cozy can implement more complex interfaces that require composing data structure representations.

Modern program synthesis techniques have been applied to low-level data structure code [23, 27]. These techniques can help to write pointer and array manipulations but, unlike our work, require the programmer to choose a data representation in advance.

More recently, researchers have made headway on synthesizing complete data structures. RelC [11] constructs data structure implementations that track subsets of a collection. It was later extended to produce safe concurrent data structures [12]. An earlier version of Cozy [16] used a custom "outline language" to describe data structure implementations and was able to synthesize data structures with richer specifications than RelC. By generalizing to arbitrary expressions and concretization functions, Cozy can now synthesize a far wider class of data structures, including the data structures for Openfire and Lucene that require multiple related collections and aggregation operators. To gain this expressiveness we have given up decidability, relying instead on bounded verification.

RelC and earlier versions of Cozy had a tuning step that used a user-supplied benchmark to make low-level optimizations. Cozy no longer has this step. Its effectiveness was never fully evaluated and our powerful symbolic cost model now fills the role. Some data structures that Cozy originally supported have also been dropped. These were not necessary for the case studies we explored, but we plan to reimplement them to extend Cozy's applicability.

Cozy's high-level algorithm resembles *programming by refinement* (PBR), in which programs are produced by manual iterative modifications to an initial specification. Unlike PBR tools such as KIDS [24], Designware [25], and Fiat [7], each refinement iteration that Cozy makes may bear little resemblance to the implementation before it. This is because Cozy enumerates possible solutions in a fixed order rather than transforming the input specification. Furthermore, Cozy requires no manual effort beyond writing a specification. The cost of this simplicity is that Cozy cannot produce many of the more complicated algorithms derived by PBR systems. However, Cozy can automate parts of the job, specifically the "finite differencing" and "data type refinement" tasks [24].

The transformations that Cozy performs are akin to the *index selection* and *view maintenance* problems in database systems. Index selection is the task of choosing useful indexes to speed up desired queries. AutoAdmin [1, 5] solves the problem by enumerating many possible indexes and using a query planner to decide which work best. As a result, AutoAdmin is limited by the set of optimization rules available to the query planner.

View maintenance is the problem of keeping an index or materialized view up-to-date as the data changes. Materialized views are similar to Cozy's concretization functions: they can be computed from the original state of the database. DBToaster [2] implements a very efficient view maintenance system. More recently, the same team has worked on generalizing these ideas to collections, including nested collections [15]. While it is possible to augment Cozy with these techniques, Cozy's enumerative synthesizer generally discovers those same solutions without the need for manual rewrite rules.

6 CONCLUSION

Cozy is effective because incrementalization allows it to implement both pure and imperative operations using only a query synthesizer. A high-quality cost function and diversity injection make the query synthesizer powerful and practical. As a result, Cozy does not need clever analyses or transformation rules. Our case studies demonstrate that data structure synthesis can improve software development time, correctness, and efficiency.

Acknowledgments

David Grant assisted with the Cozy implementation and the Lucene case study. This material is based upon work supported by the United States Air Force under Contract No. FA8750-15-C-0010, and on research sponsored by Air Force Research Laboratory and DARPA under agreement number FA8750-16-2-0032. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. This work is supported in part by NSF grant CCF-1651225, and the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA).

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 496–505. http://dl.acm.org/citation.cfm?id=645926.671701

[2] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *Proceedings of the VLDB Endowment* 5, 10 (June 2012), 968–979. DOI:http://dx.doi.org/10.14778/2336664.2336670

[3] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *Formal Methods in Computer-Aided Design (FMCAD '13)*. IEEE, 1–8. http://ieeexplore.ieee.org/document/6679385/

[4] Alexandr Andoni, Dumitru Daniliuc, and Sarfraz Khurshid. 2003. *Evaluating the "Small Scope Hypothesis"*. Technical Report. MIT.

[5] Surajit Chaudhuri and Vivek R. Narasayya. 1997. 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)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 146–155. http://dl.acm.org/citation.cfm?id=645923.673646

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. 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/ETAPS '08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. DOI:http://dx.doi.org/10.1007/978-3-540-78800-3_24

[7] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 689–700. DOI:http://dx.doi.org/10.1145/2676726.2677006

[8] Robert B. K. Dewar, Arthur Grand, Ssu-Cheng Liu, Jacob T. Schwartz, and Edmond Schonberg. 1979. Programming by Refinement, As Exemplified by the SETL Representation Sublanguage. *ACM Transactions on Programming Languages and Systems* 1, 1 (Jan. 1979), 27–49. DOI:http://dx.doi.org/10.1145/357062.357064

[9] Jay Earley. 1973. Relational Level Data Structures for Programming Languages. *Acta Informatica* 2, 4 (Dec. 1973), 293–309. DOI:http://dx.doi.org/10.1007/BF00289502

[10] Jay Earley. 1975. High Level Iterators and a Method for Automatically Designing Data Structure Representation. *Computer Languages* 1, 4 (Jan. 1975), 321–342. DOI:http://dx.doi.org/10.1016/0096-0551(75)90019-3

[11] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. 2011. Data Representation Synthesis. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 38–49. DOI:http://dx.doi.org/10.1145/1993498.1993504

[12] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. 2012. Concurrent Data Representation Synthesis. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 417–428. DOI:http://dx.doi.org/10.1145/2254064.2254114

[13] Ignite Realtime. 2016. Openfire real time collaboration server. (2016). https://www.igniterealtime.org/projects/openfire/ (Retrieved March 28, 2017).

[14] Daniel Jackson and Craig Damon. 1996. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. In *ISSTA*. 239–249. DOI:http://dx.doi.org/10.1145/229000.226322

[15] Christoph Koch, Daniel Lupei, and Val Tannen. 2016. Incremental View Maintenance For Collection Programming. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '16)*. ACM, New York, NY, USA, 75–90. DOI:http://dx.doi.org/10.1145/2902251.2902286

[16] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. 2016. Fast Synthesis of Fast Collections. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 355–368. DOI:http://dx.doi.org/10.1145/2908080.2908122

[17] Sat4j 2016. Sat4j Boolean Reasoning Library. (2016). https://www.sat4j.org (Retrieved February 3, 2016).

[18] SatCompetition2002 2002. SAT Competition 2002. (2002). http://www.satcompetition.org/2002/ (Retrieved February 3, 2016).

[19] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. 1981. An Automatic Technique for Selection of Data Representations in SETL Programs. *ACM Transactions on Programming Languages and Systems* 3, 2 (April 1981), 126–143. DOI:http://dx.doi.org/10.1145/357133.357135

[20] Jacob T. Schwartz. 1975. 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)*. ACM, New York, NY, USA, 36–40. DOI:http://dx.doi.org/10.1145/512976.512981

[21] Ohad Shacham, Martin Vechev, and Eran Yahav. 2009. Chameleon: Adaptive Selection of Collections. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 408–418. DOI:http://dx.doi.org/10.1145/1542476.1542522

[22] Laure Simon, Daniel Le Berre, and Edward A. Hirsch. 2005. The SAT2002 Competition. *Annals of Mathematics and Artificial Intelligence* 43, 1 (1 Jan. 2005), 307–342. DOI:http://dx.doi.org/10.1007/s10472-005-0424-6

[23] Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing Data Structure Manipulations from Storyboards. In *Proceedings of the 19th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 289–299. DOI:http://dx.doi.org/10.1145/2025113.2025153

[24] Douglas R. Smith. 1990. KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering* 16, 9 (Sept. 1990), 1024–1043. DOI:http://dx.doi.org/10.1109/32.58788

[25] Douglas R. Smith. 1999. Designware: Software Development by Refinement. *Electronic Notes in Theoretical Computer Science* 29 (Dec. 1999), 275 – 287. DOI:http://dx.doi.org/10.1016/S1571-0661(05)80320-2

[26] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. University of CA at Berkeley, Berkeley, CA, USA. Advisor(s) Bodík, Rastislav. AAI3353225.

[27] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. 2008. Sketching Concurrent Data Structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 136–148. DOI:http://dx.doi.org/10.1145/1375581.1375599

[28] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. *Operating Systems Review* 40, 5 (Oct. 2006), 404–415. DOI:http://dx.doi.org/10.1145/1168917.1168907

[29] The Apache Software Foundation. 2016. Apache Lucene. (2016). https://lucene.apache.org.

[30] John Toman and Dan Grossman. 2016. Staccato: A Bug Finder for Dynamic Configuration Updates. In *30th European Conference on Object-Oriented Programming (ECOOP '16)*. Schloss Dagstuhl, Dagstuhl, Germany, 24:1–24:25. DOI:http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.24

[31] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 287–296. DOI:http://dx.doi.org/10.1145/2491956.2462174

[32] ZTopo 2015. ZTopo Topographic Map Viewer. (2015). https://hawkinsp.github.io/ZTopo/ (Retrieved May 8, 2015).