Data Structure Synthesis

Calvin Loncaric

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2018

Reading Committee:

Michael D. Ernst, Chair

Emina Torlak

Alvin Cheung

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Data Structure Synthesis

Calvin Loncaric

Chair of the Supervisory Committee:
Professor Michael D. Ernst
Paul G. Allen School of Computer Science and Engineering

Data structures are integral to software. Many programs require custom application-specific data structures more complicated than those found in standard libraries. Implementing and maintaining application-specific data structures is tedious and error-prone.

This work presents Cozy, a novel tool that synthesizes efficient implementations of application-specific data structures from high-level specifications. Cozy handles a wider range of data structures than previous work, including structures that track multiple related collections and have complex retrieval methods involving sums, counts, minimums, and maximums.

Cozy iteratively discovers good data structures using alternating steps of *query synthesis* and *state maintenance*. 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 state maintenance 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.

Cozy has three goals: to reduce programmer effort, to produce bug-free code, and to match the performance of handwritten code. We have evaluated Cozy in four large case studies, demonstrating that it meets the goals. Using Cozy requires an order of magnitude fewer lines of code than manual implementation, results in fewer bugs, and matches the performance of handwritten code.

Finally, we have used Cozy as an *automatic incrementalizer*. An incremental algorithm can update its output efficiently in response to small changes to its input. Replacing batch-style algorithms with incremental versions yields incredible speedups when the input data changes frequently. Cozy is well-suited to the problem since incrementalization is simply the task of finding the right data structure to track state between changes to the input. By re-framing the incremental computation task as a data structure specification we can produce efficient incremental versions in more situations than previous work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

My work was only possible with a great deal of help and support—more than I can list here.

Thank you Mike; your boundless enthusiasm made our time together a joy. You often understood my own ideas better than I, and I hope I can live up to your incredible talent.

Thank you Emina and Daniel; this project would never have started without your early input, nor continued without your ideas.

Thank you Alvin and Zach and Ras and all the other PLSE professors; I feel privileged for the years spent learning from you.

Thank you Eric, Stuart, Doug, James, Pavel, Martin, and all the other UW students; I would never have made progress without the brainstorming and research ideas and feedback.

Thank you Satish, Cole, Manu, and Colin; the work we did together helped me become an effective researcher. I still consider our project one of the slickest I ever worked on.

Thank you Haoming, David, and especially Andrew; the software only functions as well as it does because I had such talented collaborators.

Finally, and most importantly, thank you to my friends and family for everything you do. Because of you I am a happy human—and would be no matter what I worked on.

Chapter 1

INTRODUCTION

Software is having an incredible impact on the world. With large codebases numbering tens or hundreds of millions of lines of code, it is imperative that we develop techniques to implement and maintain incredibly complex projects.

Program synthesis offers to address the challenge. With synthesis, developers write short, high-level specifications describing *what* task to perform, and a computer figures out *how* to perform the task. Synthesis has been successfully applied to algorithm discovery, automation for end-users, teaching, programming assistance, program understanding, optimization, and other domains [41].

Recent research seeks to automatically synthesize data structure implementations from high-level specifications, thus ensuring correctness and run-time efficiency with minimum programmer effort [46, 47, 65]. 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.

This work introduces a new program synthesis technique to implement complex data structures from their specifications. Since many software modules are simply data structures in disguise—they have constructors, private state, and methods to query and update their state—a data structure synthesizer has huge potential to help developers implement complex systems quickly and correctly.

Synthesis helps with both implementation and maintenance. Since specifications are usually shorter than their corresponding implementations, they can be easier to get correct initially. Reducing the maintenance burden is just as important; the vast majority of development time is maintenance. Synthesis helps with maintenance since adjusting a specification to fit new requirements is easier than adjusting a very large implementation.

1.1 Overview of Cozy

The ideas in this document are implemented in a tool called Cozy [65, 64, 24]. Cozy synthesizes implementations for complex data structures from high-level specifications. This section describes what it is like to write programs with Cozy's help; later chapters describe the ideas and insights that comprise Cozy's internal implementation.

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. Concretely, given the specification

```
state data : List⟨Int⟩
query contains_zero() = (0 ∈ data)
op add(x : Int) = data.add(x)
```

Cozy will produce the efficient implementation

```
state zero_in_data : Bool
query contains_zero() = zero_in_data
op add(x : Int) =
  let _var20 := (zero_in_data ∨ (0 = x))
  zero_in_data := _var20
```

which can be directly translated into efficient Java or C++.

While simple, this example illustrates Cozy's power. The specification is self-documenting: each operation states exactly what it computes about the data or how it modifies the data. For this reason, the specification is easy to manually validate. The efficient implementation is not as easy to manually validate, since the correctness of $contains\_zero$ is dependent on how add alters the state. That is to say, the implementation has an extra invariant to maintain: $zero\_in\_data$ is true if and only if 0 has ever been passed as a parameter to add. The value of simple abstract specifications has been known for a long time [58], and is still pervasive today in the distinction between logical and physical layout in database systems.

The example also demonstrates Cozy's potential to improve performance. Since the implementation does not need every number in the list in order to implement $contains\_zero$, Cozy has

decided not to store the list at all. The resulting implementation has $O(1)$ performance and space usage—a savings that is only possible by completely rewriting the representation and introducing a new invariant.

1.2 Organization of This Document

Chapters 2 and 3 describe how Cozy works, starting from the high level algorithm and insights that make the tool possible (Chapter 2) and then diving into the query synthesizer that makes it effective. (Chapter 3). The next two chapters show Cozy's effectiveness at replacing handwritten data structures (Chapter 4) and automatically incrementalizing algorithms (Chapter 5). The final chapters describe Cozy in the context of prior literature (Chapter 6) and conclude (Chapter 7).

Chapter 2

ITERATIVE DISCOVERY OF GOOD DATA STRUCTURES

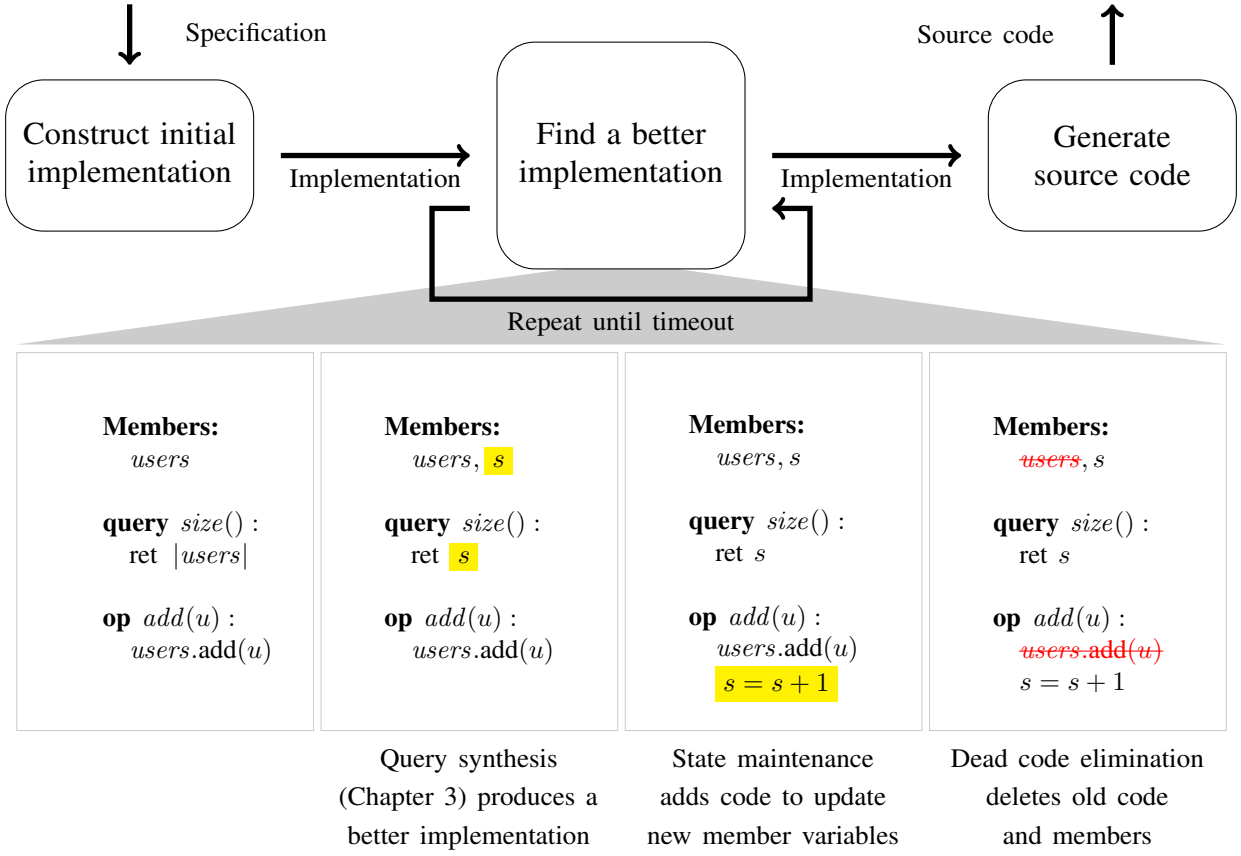


Figure 2.1: Architecture of Cozy. Each iteration through the loop performs query synthesis, state maintenance, and dead code elimination. Figure 2.2 shows an example input specification, and Figures 2.3 and 2.5 show the corresponding output.

To synthesize an efficient implementation for an abstract data type, any data structure synthesizer must solve two interrelated problems: picking a good representation and picking a good algorithm

that uses the new representation. Furthermore, the new representation needs to be properly maintained so that it stays in sync with the original representation. Cozy solves the first two problems in one unified procedure called "query synthesis" and the final problem in a separate step called "state maintenance" (previously referred to as incrementalization [64]). The state maintenance step offloads all of the hard work to query synthesis, reducing these interrelated problems to one powerful procedure.

Choosing the representation and algorithm together makes Cozy more capable than previous techniques. The older tools AutoAdmin [19, 6] and RelC [46, 47] tackle this problem in a different way. They explore many possible representations and use a query planner to determine which representations are good ones. This approach is difficult to extend very far: even trivial representations can discard information, making query planning extremely difficult. An older version of Cozy [65] worked in the opposite order, by implementing an algorithm and then attempting to infer a representation that makes the algorithm valid. This greatly restricts the space of possible algorithms. In particular, the older version of Cozy did not support conditionals, multiple collections, or reduce operators like sum or min.

In previous work, implementations for update methods were either hard-coded [65] or derived using a set of hand-written rules [46]. In Cozy, update methods are synthesized rather than hard-coded. 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 a *state maintenance* step that ensures the new representation is kept up-to-date when an update method is called. Crucially, the state maintenance step can produce specifications for new query operations to help implement the update procedure. Cozy thus uses the query synthesizer to implement both pure query operations and imperative updates. Our technique is agnostic to the exact implementation of the query synthesizer; Chapter 3 gives a detailed explanation of the concrete choices we made for Cozy. Since state maintenance offloads much of its work to query

synthesis, having a powerful query synthesis procedure makes Cozy much more effective.

The query synthesizer and state maintenance 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:

$$\mathcal{C}(S) = \sum_{x \in S} 1 \, .$$

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

$$\mathcal{C}(S') = \mathcal{C}(S - \{e\}) = \sum_{x \in (S - \{e\})} 1 \, .$$

Concretization functions are a restricted form of abstraction relation [58]. In practice, this restriction is not harmful; researchers have long noted that many invariants come in the form $E = f(x_1, ..., x_n)$ [74].

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 state maintenance 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: $\mathcal{C}(S') - \mathcal{C}(S)$. 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.

2.1 Example: The Openfire Roster

This section illustrates how Cozy behaves on a real-world example. The chat server Openfire [50] 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 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 2.1. It takes as input an executable specification of the data structure (Section 2.1.1), constructs an initial implementation (Section 2.1.2), and then iteratively improves its implementation using alternating steps of query synthesis (Section 2.1.3) and state maintenance (Section 2.1.4). A dead code elimination step (Section 2.1.5) prunes dead code as synthesis progresses.

2.1.1 Specification

Figure 2.2 shows a complete Cozy specification for part of the Openfire contact management data structure. It would be used as the input to Figure 2.1. 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 2.2 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. This example has been simplified; our experiments (Chapter 4) use a full

abstract state

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

query definition

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

update definition

```
// Update operation definition
op join(u : User, g : Group):
  assume u in users
  assume g in groups
  assume (u, g) not in members
  members.add((u, g))
```

Figure 2.2: A simplified Cozy specification for the Openfire roster data structure.

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 2.3) that runs in $O(g)$ time, where g is the maximum number of groups that any one user is a member of.

2.1.2 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 2.2, Cozy's initial implementation has the variables v_1, v_2, and v_3 and trivial concretization functions:

concrete state

```
// Users who are in 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)>
```

implementation

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

Figure 2.3: New implementation of `visible` after several query synthesis steps. Cozy does not produce the comments, which we added for clarity. Since state maintenance 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.

$$\mathcal{C}_{v1}(\texttt{users}, \texttt{groups}, \texttt{members}) = \texttt{users}$$
$$\mathcal{C}_{v2}(\texttt{users}, \texttt{groups}, \texttt{members}) = \texttt{groups}$$
$$\mathcal{C}_{v3}(\texttt{users}, \texttt{groups}, \texttt{members}) = \texttt{members}$$

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 in v1
  assume u2 in v1
  return (exists [ g | g ← v2,
    (u1, g) in v3 and (
      g.visibility == Everyone or
      (u2, g) in 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.

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

C_s3(users, groups, members) =
  MakeMap_λm.true members

C_v1(users, groups, members) = users
C_v2(users, groups, members) = groups
C_v3(users, groups, members) = members
```

Figure 2.4: Concretization functions for the synthesized implementation of `visible` in Figure 2.3.

2.1.3 Query Synthesis

Cozy synthesizes an implementation by iteratively finding improvements to the data structure. The query synthesis step in Figure 2.1 makes an improvement to some non-deterministically chosen query operation on the data structure. Chapter 3 discusses how Cozy makes the choice and the improvement.

Figure 2.3 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 2.3, 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 $\mathcal{C}_{s1}$,

```
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 in 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 in users
  assume g in groups
  assume (u, g) not in members
  return [ k | k ← MapKeys(s1) ∪ MapKeys(s1′), s1[k] != s1′[k] ]

// Compute a new value at key k in s1 when user u joins group g.
private query new_value_for_key_s1(k, u : User, g : Group):
  assume u in users
  assume g in groups
  assume (u, g) not in members
  assume s1[k] != s1′[k]
  return s1′[k]

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

(b)

Figure 2.5: (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: $\mathcal{C}_{\texttt{s1}}(\texttt{users}', \texttt{groups}', \texttt{members}')$.

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, the expression `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 2.4 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 state maintenance.

2.1.4 State Maintenance

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 state maintenance 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 2.4.

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 updates that make small changes to the concrete state in response to small changes to the abstract state.

To update the concrete state, Cozy rephrases the update procedure as a set of queries that compute what changes should take place, plus a simple hard-coded snippet that applies those computed changes. A previous approach applied this same idea to synthesize remove operations [46], 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 2.5a 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 2.3). However, the new

code uses two fresh query operations `altered_keys_s1` and `new_value_for_key_s1` (Figure 2.5b) 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

```
users′ = users
groups′ = groups
members′ = members ∪ {(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 2.5b 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 state maintenance.

2.1.5 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.

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.

2.2 Cozy's Language

Figure 2.6 shows Cozy's core specification language. All input specifications are desugared to this core language (Figure 2.7). Cozy's output language is a superset of its input language that includes additional constructs for maps:

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

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.

2.3 State Maintenance in Detail

The state maintenance step illustrated by example in Section 2.1.4 lies at the heart of Cozy's core algorithm. This section explains state maintenance in greater detail.

After query synthesis picks a new representation for the data, the state maintenance step restores proper functioning by adding code to keep that representation up-to-date as the data structure changes. Cozy's $maintain$ procedure (Figure 2.8) accomplishes that goal by leveraging the existing query synthesis procedure.

For example, Cozy may find itself in this state before $maintain$ is called:

$spec$	::=	$name$: $s_1, s_2, ...$ invariant e $m_1, m_2, ...$	specifications
s	::=	$x : \tau$	abstract state
τ	::=	Int | Bool | String	basic types
	|	Enum $\{case_1, case_2, ...\}$	enumerations
	|	$\langle \tau_1, \tau_2, ... \rangle$	tuples
	|	$\{f_1 : \tau_1, f_1 : \tau_2, ...\}$	records
	|	Bag$\langle \tau \rangle$	bags (multisets)
	|	List$\langle \tau \rangle$	lists
m	::=	query $q(args...)$: assume e; return e;	queries
	|	op $u(args...)$: 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
	|	T | F | 0 | 1 | ...	literals
	|	$e == e \mid e < e \mid ...$	comparisons
	|	$e \wedge e \mid e \vee e \mid \neg e$	bool operations
	|	$e \, ? \, e \, : \, e$	conditionals
	|	$e + e \mid e - e$	arithmetic
	|	$(e, e, ...) \mid e.n$	tuples
	|	$\{f : e, f : e, ...\} \mid e.f$	records
	|	$\emptyset \mid \{e\} \mid e \cup e \mid e - e$	bag operations
	|	Map$_f$ e | Filter$_f$ e	map and filter
	|	FlatMap$_f$ e	map union
	|	First e	first element
	|	$\Sigma \, e$	sum
	|	Distinct e	remove duplicates
	|	ArgMin$_f$ e | ArgMax$_f$ e	min and max
f	::=	$\lambda x.e$	lambda abstraction

Figure 2.6: Core specification language *spec*.

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

Figure 2.7: 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.

```
state ints : Bag<Int>
state sum  : Int
query getSum():
  return sum
op addInt(i : Int):
  ints.add(i)
  ??
```

To produce code for the `??` hole, *maintain* consults the rules in Figure 2.8 and determines that the hole `??` should be replaced by $\texttt{sum} := \texttt{sum} + q(\texttt{i})$, where q is a new query operation. The new query q must compute the change to `sum` as a result of calling `addInt` with any integer `i`. If `sum` has the concretization function $\mathcal{C}_{\texttt{sum}}(\texttt{ints}) = \Sigma\ \texttt{ints}$, then the definition of q is given by

$$\mathcal{C}_{\texttt{sum}}(\texttt{ints} \cup \{\texttt{i}\}) - \mathcal{C}_{\texttt{sum}}(\texttt{ints})\ .$$

In code, after inlining $\mathcal{C}_{\texttt{sum}}$, q gets the definition

```
query q(i : Int):
  Σ (ints ∪ {i}) - Σ ints
```

In `join` from Section 2.1.4, Cozy updated `s1` using the code

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

which calls the new queries `altered_keys_s1` and `new_value_for_key_s1`. Since `s1` has a map type, Cozy uses the *state maintenance sketch* shown in Figure 2.8 for maps. A state maintenance sketch

$maintain(x, \mathcal{C}_x)$:
Input: old abstract state σ and new abstract state σ'
Output: code to update concrete state x

Type	Update Sketch	New Queries
Int	$x \coloneqq x + q(...)$	$q(...) = \mathcal{C}_x(\sigma') - \mathcal{C}_x(\sigma)$

Bag	for $elem \in q_1(...)$:
	  x.del($elem$)
	for $elem \in q_2(...)$:
	  x.add($elem$)
		$q_1(...) = \mathcal{C}_x(\sigma) - \mathcal{C}_x(\sigma')$
		$q_2(...) = \mathcal{C}_x(\sigma') - \mathcal{C}_x(\sigma)$

Map	for $k \in q(...)$:
	  $maintain($
	  $x[k]$,
	  $\lambda\sigma.\mathcal{C}_x(\sigma)[k])$
		$q(...) = \{k \mid$
		  $k \in$ MapKeys$(\mathcal{C}_x(\sigma)) \cup$
		  MapKeys$(\mathcal{C}_x(\sigma'))$,
		  $\mathcal{C}_x(\sigma)[k] \neq \mathcal{C}_x(\sigma')[k]\}$

other	$x \coloneqq q(...)$	$q(...) = \mathcal{C}_x(\sigma')$

Figure 2.8: Rules for $maintain(x, \mathcal{C}_x)$. $\mathcal{C}_x$ is the concretization function for x. To update a map-type variable, $maintain$ is called recursively to determine how to update the value at each changed key.

is a small snippet of imperative code called an *update statement* that updates the variable. A state maintenance sketch may require new query operations in order to function. In the case of maps, the 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 2.1.4, the new value for `s1[k]` is simply `true`. In practice, new queries generated by $maintain$ often have short and efficient implementations.

2.3.1 Statement Ordering

In addition to applying a state maintenance sketch for each member variable, Cozy also needs to determine what order to execute the update statements. Each statement may call queries, and each

query may read the data structure state—even if that state is affected by other statements.

A trivial illustration of this is a simple "toggle" data structure:

```
state b : Bool
query read():
  return b
op flip():
  b = not b
```

After several rounds of query synthesis and state maintenance, Cozy may wish to store both `b` and `not b` on the data structure:

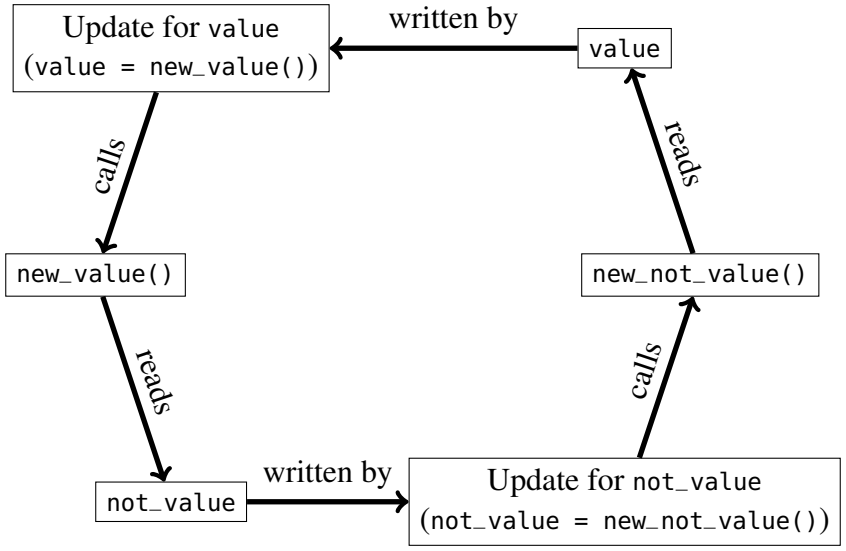
```
state     value : Bool  // = b
state not_value : Bool  // = not b
query read():
  return value
op flip():
  value = new_value()         // maintenance sketch for booleans
  not_value = new_not_value() // maintenance sketch for booleans
query new_value():
  return not_value
query new_not_value():
  return value
```

However, Cozy's new implementation is incorrect! The first statement in `flip` updates `value`, which is then read by `new_not_value` on the following line.

The toggle data structure illustrates the *read after write* problem: all of the queries introduced during state maintains (Figure 2.8) are phrased in terms of the pre-state of the data structure, before any mutation has taken place. When Cozy introduced `new_not_value` it was assuming the code would be run before the write to `value`, and when it introduced `new_value` it was assuming that the code would be run before the write to `not_value`.

In fact, there is no ordering of the two update statements that fixes the problem. To resolve the apparent conflict, Cozy needs to introduce additional temporary variables to preserve the assumption that each query is executed before the state that it reads is altered.

To resolve the problem, Cozy first constructs a graph of the call/read/write relationships among the statements, queries, and data structure members:



In the generated code, no update is allowed to read a variable that has already been written—in other words, Cozy needs a topological sorting of this graph to put statements in the correct order. In cases like the toggle example, the graph may have cycles that prevent a topological sorting.

Cozy can break a "calls" edge in the graph by introducing a new temporary variable at the top of the `flip` method that holds the result of the call. Breaking the minimum number of "calls" edges to make the graph acyclic—and thus enable topological sort—is an instance of the minimum feedback arc set problem, which is known to be NP-complete [52]. Fortunately, instances of this problem tend to be small in practice, so Cozy uses an exact algorithm to solve the problem optimally. The optimal algorithm has never been a bottleneck. In the future Cozy might solve a weighted instance of this problem to prefer holding temporaries that occupy fewer bytes in memory, but it does not do so currently.

There will always be a way to make the graph acyclic by breaking "calls" edges. Because of the structure of the $maintain$ procedure, the graph is truly tripartite: all outgoing edges from update statements are "calls" edges to queries, all outgoing edges from queries are "reads" edges to members, and all outgoing edges from members are "written by" edges to update statements. There are no other kinds of edges in the graph.

In the `flip` example, Cozy has two "calls" edges to break. It arbitrarily breaks the "calls" edge to `new_not_value`, resulting in the correct code

```
op flip():
  let tmp = new_not_value()
  value = new_value()
  not_value = tmp
```

2.4 Termination

The query synthesis procedure (Chapter 3) 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 (Chapter 4) used a fixed timeout of three hours for synthesis.

Chapter 3

QUERY SYNTHESIS

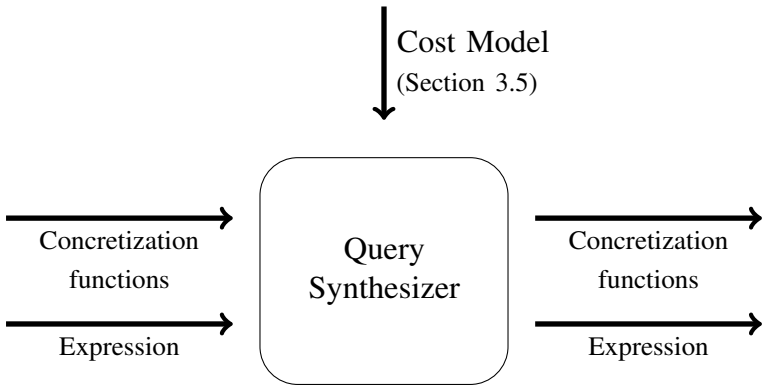


Figure 3.1: Overview of Cozy's query synthesizer

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 (Section 3.5) defines "better." Whenever a thread discovers a better implementation: (1) That implementation is immediately passed through the state maintenance 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 [95, 102, 9], but Cozy employs several novel improvements. We describe the core algorithm first, followed by our extensions.

3.1 Enumerative Synthesis Background

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 [26]) 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 [102, 65], an optimization that skips most expressions in the search space. The skipping is done safely so that Cozy never 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 adds four novel additions to the core enumerative synthesis algorithm: *representation packing*, *diversity injection*, *higher-order expression construction*, *cost optimization*, and *subexpression replacement*, Additionally, since verification is undecidable for our specification language, Cozy uses *bounded verification* instead of full functional verification. Bounded verification is also used by other synthesis tools [93].

3.2 Representation Packing

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*. Each state expression becomes a new member with the state expression as its concretization function. Query expressions are evaluated at run time when 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:

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

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. These two possibilities correspond respectively to the solutions:

```
state x : Int // = Σ Map_{λx.1} S

query q():
  return x
```

```
state x : Set // = S

query q():
  return Σ Map_{λx.1} x
```

The first is a data structure that stores the sum as a member and returns it on-demand, and the second is a data structure that stores the set S and computes the sum 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 state maintenance.

In rare cases Cozy can back itself into a corner since its cost model does not "look ahead" to see the difficulty of state maintenance given the data structure's operations. We did not observe this to be a major problem in our case studies.

Expressions that contain query arguments may not be tagged as state expressions, since those

values will not be available until the query is executed. Packed expressions have three well-formedness conditions:

- Abstract state variables must be tagged as state expressions. This enforces that any abstract state used by the expression has a corresponding member on the data structure.

- Query arguments must be tagged as query expressions. Since the arguments are not available until run time, they cannot be stored as members on the data structure.

- There may not be state expressions within state expressions; the expression

$$\underbrace{\underbrace{x}_{\text{state}} + 1}_{\text{state}}$$

  is redundant and should simply be

$$\underbrace{x + 1}_{\text{state}} .$$

The well-formedness conditions can be repaired for any expression by stripping out all state tags and then attaching a state tag to every abstract state variable. In fact, this is exactly what Cozy does to construct an initial implementation (Section 2.1.2). During synthesis, however, Cozy does not use such a heavy-handed approach since altering the tags can drastically affect the estimated cost of an expression (Section 3.5). Instead, all of Cozy's procedures that operate on expressions take care to preserve their well-formedness.

3.3 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 2.3 requires 45 nodes.

Map Introduction

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

Cleaners

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

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

Relevant Subset

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

Instantiation

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

Figure 3.2: Cozy's diversity rules.

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

Cozy uses the five diversity rules shown in Figure 3.2. 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.

3.4 Higher-Order Expression Construction

Higher-order expressions are those containing anonymous lambda functions, such as

$$\underset{\lambda x.x+1}{\text{Map}}\ \ \mathit{list}\ .$$

While Cozy's core syntax only allows lambda functions in certain locations (Figure 2.6), their presence still presents a problem for enumerative synthesis. Since Cozy builds expressions from smallest to largest, how can it ever visit the expression x without knowing in advance that there will be a lambda function with x as a formal parameter? Even if it knew that x will appear, the counterexamples returned by the verifier will not have values for x, so how should x be differentiated from other expressions?

To address the problem, Cozy employs a mix of bottom-up and top-down enumeration, exploiting the fact that lambda functions only appear in certain locations. Figure 3.3 illustrates the algorithm.

The highlighted line illustrates the solution: when Cozy needs to construct a lambda node to fill in a hole in an expression tree (*e.g.* a Map or Filter node), it introduces a fresh variable and *instantiates* all the known examples with values for that variable. Because lambda nodes only appear in certain locations, Cozy always knows at instantiation time what collection holds all possible values for the fresh variable. By evaluating that collection on all known examples, Cozy constructs a new, extended set of examples with bindings for the fresh variable.

Concretely, when enumerating bodies for the expression

$$\underset{\lambda x.\ ?}{\text{Map}}\ \ \mathit{list}$$

```
def enumerate(vars, size, examples) :
    // vars : a set of expressions (variable nodes)
    // size : a natural number
    // examples : a set of example inputs (variable-to-value mappings)

    // base case
    if size = 0 :
        for v ∈ vars :
            yield(v)
        return

    // build unary operations
    for e ∈ enumerate(vars, size − 1, examples) :
        if typeof(e) = INT : yield(−e)
        if typeof(e) = BOOL : yield(¬e)
        ...

    // build binary operations
    for each way to pick s1, s2 : ℕ where s1 + s2 = size − 1 :
        for e1, e2 ∈ enumerate(vars, s1, examples) × enumerate(vars, s2, examples) :
            if typeof(e1) = typeof(e2) = INT : yield(e1 + e2)
            ...
        for L ∈ enumerate(vars, s1, examples) :
            if typeof(L) is a collection :
                let v = fresh_var()
                for body ∈ enumerate(vars ∪ v, s2, instantiate(examples, v, L)) :
                    yield(Map_{λv.body} L)
                    ...

def instantiate(examples, v, collection) :
    // examples : a set of example inputs (variable-to-value mappings)
    // v : a variable
    // collection : an expression
    return {ex[v ↦ val] | ex ∈ examples, val ∈ eval(collection, ex)}
```

Figure 3.3: Cozy's higher-order enumeration algorithm. The highlighted line shows how Cozy enumerates lambda expressions when not all formal parameters are known a priori. For clarity, the figure omits the caching and equivalence class deduplication logic from *enumerate*.

with examples

$$list = \{1\},$$
$$list = \{1, 1, 2\}$$

Cozy will construct the instantiated examples

$$list = \{1\} \qquad x = 1,$$
$$list = \{1, 1, 2\} \quad x = 1,$$
$$list = \{1, 1, 2\} \quad x = 2$$

The $enumerate$ procedure does not recurse infinitely for any given size since the $size$ parameter is monotonically decreasing.

3.5 Cost Optimization

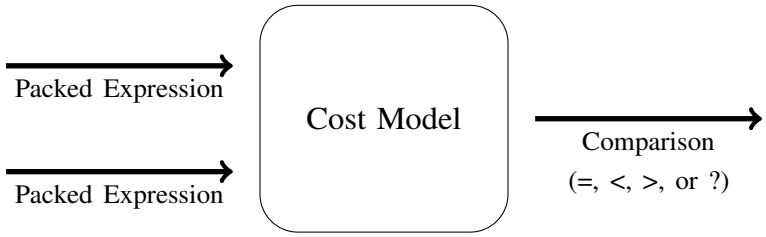


Figure 3.4: Overview of Cozy's cost model

Cozy extends the regular enumerative CEGIS algorithm with cost optimization. However, doing so for Cozy's complicated cost model is not trivial.

Figure 3.5 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 ($cost_Q$).

In a departure from previous work, 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.

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 3.5: 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.

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.

Adapting enumerative CEGIS to use a cost model requires care. A simple solution would be to always keep the lowest-cost expression for each equivalence class. This is a correct policy for

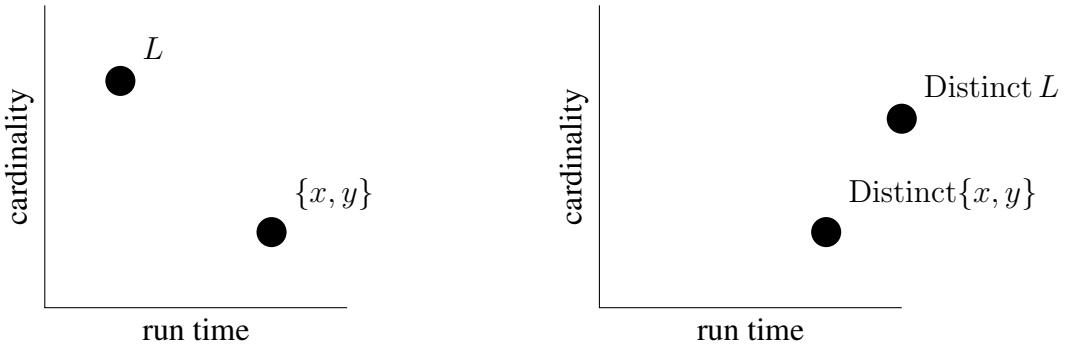


Figure 3.6: Pareto optimality in Cozy. Despite having higher run time, the expression $\{x, y\}$ cannot be discarded since it may have lower run time in larger expressions. Instead, Cozy must maintain a Pareto frontier for each equivalence class during synthesis.

cost models that exhibit *monotonicity* [65] in the sense that the cost of an expression correlates with the cost of every one of its subexpressions. Put another way, a monotonic cost model is one where replacing any subexpression with a lower-cost subexpression reduces the overall cost. Monotonicity ensures that discarding higher-cost expressions in the same equivalence class is always safe, since any expression that uses the higher-cost version as a subexpression could be improved by using the lower-cost one instead.

Unfortunately, monotonicity is difficult to achieve in general and is not true for Cozy's cost model. The cost of an expression depends not only on the estimated run times of its subexpressions, but also on the estimated cardinalities of its subexpressions. Figure 3.6 illustrates how a non-monotonic cost model foils the "keep-one" scheme proposed above. Early in synthesis the expressions L and $\{x, y\}$ may fall into the same equivalence class. The first has lower run time cost since it does not construct a new set, but its high estimated cardinality causes expressions built with it to have very high estimated run time. In this case Cozy should not discard $\{x, y\}$ or it will never visit the expression $\text{Distinct}\{x, y\}$, which may be a much more efficient implementation if it happens that $\text{Distinct}\{x, y\}$ and $\text{Distinct } L$ are semantically equivalent. Note that Cozy *can* discard $\text{Distinct } L$, since it is worse than $\text{Distinct}\{x, y\}$ along every dimension.

The fact that run time and cardinality both play a role in the cost model means that Cozy faces

a multi-objective optimization problem. To fix the "keep-one" scheme, Cozy needs to maintain a Pareto frontier of expressions in every equivalence class.

Further complicating matters, Cozy's symbolic cost model may determine that two expressions are ambiguous with respect to run time or cardinality. In this case, Cozy keeps both in its cache.

3.6 Subexpression Replacement

Unlike other applications of CEGIS, Cozy can exploit the fact that the input specifications are executable. Whenever Cozy finds any improvement to any subexpression, it immediately substitutes the old version for the better one. This allows Cozy to make improvements to the specification even when the optimal solution is out of reach because it is too large for brute-force enumeration to find.

Cozy identifies when a better version of a subexpression is found using the existing CEGIS machinery. Before synthesis starts, Cozy identifies the equivalence class of every subexpression of the specification. During synthesis, whenever an expression falls into one of those classes, Cozy compares it to the subexpressions in that class to see if it is an improvement.

As discussed in Section 3.5, Cozy's cost model has multiple objectives (cardinality and run time). Thus, Cozy tries replacements that seem to improve *either* objective, and examines the run time cost of the overall result to determine whether the replacement is worth keeping.

Even if the specifications were not executable, this technique is still applicable. Cozy could search for any legal solution and, once found, begin performing subexpression replacement.

3.7 Bounded Verification

It is undecidable to determine whether an expression in Cozy's language satisfies a specification [40]. 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* [51], 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 [10], and we found it to be true in our domain as well. Cozy's language has

$SymbolicTerm ::= \text{SMT-LIB } formula \mid \{elem_1, elem_2, ...\} \mid SymbolicTerm \to SymbolicTerm$
$elem ::= (SymbolicTerm, formula)$

$encode : (exp \times env) \to SymbolicTerm$
$encode(x, env) = env[x]$
$encode(e_1 \vee e_2, env) = \text{SMT-LIB (or } encode(e_1, env)\ encode(e_2, env))$
$encode(\emptyset, env) = \emptyset$
$encode(e_1 \cup e_2, env) = encode(e_1, env) \cup encode(e_2, env)$
$encode(\text{Filter}_f\ e, env) = \{(x, (\text{and } mask\ encode(f, env)(x)) \mid (x, mask) \in encode(e, env)\}$
$encode(\text{Map}_f\ e, env) = \{(encode(f, env)(x), mask) \mid (x, mask) \in encode(e, env)\}$
$encode(\lambda x.e, env) = \lambda y.encode(e, env[x \mapsto y])$
$encode(\Sigma e, env) = \text{SMT-LIB } (\Sigma_{(x,mask) \in encode(e)}(\text{ite } mask\ x\ 0))$
...

$reconstruct_{Type} : (SymbolicTerm \times model) \to value$
$reconstruct_{\text{Bool}}(f, m) = m[f]$
$reconstruct_{\text{Bag}\langle T \rangle}(b, m) = \{reconstruct_T(x) \mid (x, mask) \in b, reconstruct_{\text{Bool}}(mask, m)\}$
$reconstruct_{A \to B}(f, m) = \lambda x.reconstruct_B(f(encode(x, \cdot)), m)$
...

Figure 3.7: Encoding Cozy expressions to SMT-LIB [11] formulas.

limited facilities for extracting single elements out of collections, and this makes it difficult for it to special-case on the low bound we picked.

Figure 3.7 shows many of the most enlightening rules for how Cozy encodes expressions (Figure 2.6) into decidable formulas. The formulas are in the SMT-LIB language [11] used by many "satisfiability modulo theories" (SMT) solvers. Cozy uses the Z3 solver [26] to solve the encoded formulas. Integer and Boolean type expressions are simple SMT formulas. Functions are represented as functions in Cozy's implementation language (Python). Collections are represented as ordered sequences of (element, mask) pairs. Filters affect the masks while maps affect the elements. When two collections need to be merged because they are on different branches of a conditional, Cozy employs *type-driven state merging* [99].

Cozy's bounded verifier is a partial decision procedure; the encoded formulas are not equisatisfi-

able with the original. If the solver returns with a model then it is indeed a satisfying model for the original expression. However if the solver reports that the formula is unsatisfiable, it may be the case that a satisfying assignment exists with a higher verification bound. Thus "unsat" should be interpreted to mean "unknown".

There are special cases for which our bound is sufficient to prove unsatisfiability [65, 40]. If the input expression does not make use of certain primitives then the expression satisfies a convenient *small-model* property: a bound of 1 is sufficient to prove unsatisfiability. However, these simple expressions are very rare in practice. Since Cozy explores all possible expressions, most expressions will make use of some undecidable primitive.

Chapter 4

COZY IN PRACTICE

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 4.2), makes no mistakes even when human programmers do (Section 4.4), and often matches the performance of handwritten code (Section 4.3).

4.1 Case Studies

4.1.1 Openfire

Openfire [50] 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 [98]. Openfire's implicit contacts require computing information about two distinct collections (users and groups), and thus cannot be handled by any previous tool.

4.1.2 Sat4j

Sat4j [79] 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 [65]. 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.

4.1.3 ZTopo

ZTopo [105] 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 [46, 65]. 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.

4.1.4 Lucene

Lucene [96] 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. Unlike the other case studies, which were carried out by Cozy's author, the Lucene case study was done by an undergraduate student, providing some evidence that Cozy is easy to learn and use.

4.1.5 Methodology

For each of four real-world programs (Section 4.1), 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 Reducing Programmer Effort

We do not know how much time programmers spent implementing and debugging the hand-written data structures, but it was significant. Table 4.1 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 implemen-

Table 4.1: Comparison of programmer effort between handwritten and Cozy-synthesized implementations. LoC measurements do not include comments or white space.

	Hand-written			Cozy
Project	**Span**	**Commits**	**LoC**	**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

tation; 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 2.2), 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 will be easier to discover, diagnose, and correct. The specifications also serve as concise, unambiguous documentation.

4.3 Matching or Exceeding Performance

We measured the performance of the handwritten and synthesized implementations on realistic workloads. Table 4.2 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

Table 4.2: Benchmark results. All times are in seconds.

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

synthesized data structure.

Our benchmarks for ZTopo and Sat4j are the same ones used to evaluate an earlier iteration of Cozy [65]. 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.

We created a benchmark for Sat4j consisting of eleven randomly-selected input files from the 2002 Boolean satisfiability solver competition [80, 87]. 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

Table 4.3: Correctness results. Note that ZTopo has no dedicated issue tracker.

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

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.

4.4 Improving 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 4.3. The existing ZTopo data structure contains many assertions that check representation invariants at run time, *e.g.*:

```
void Cache::addToDiskLRU(Entry &e) {
  assert(!e.is_linked() && e.state == Disk);
  diskLRUSize += e.diskSize;
  diskLRU.push_back(e);
}
```

The assertions are not necessary in the synthesized replacement, since the synthesized replacement is correct by construction, and in fact, many helper methods like `addToDiskLRU` are handled entirely by the synthesized implementation.

Table 4.3 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. Chapter 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 [72]. 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 [63]. 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.

Chapter 5

INCREMENTALIZATION VIA DATA STRUCTURE SYNTHESIS

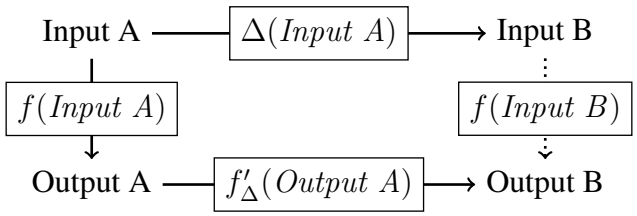


Figure 5.1: Incremental computation avoids recomputing f from scratch when the input changes according to Δ. Automatic incrementalization derives f'_Δ from f and Δ.

An incremental algorithm can update its output efficiently in response to small changes to its input. Replacing batch-style algorithms with incremental versions yields incredible speedups. However, incremental algorithms are notoriously difficult to write. Decades of research have yielded techniques to automatically derive incremental algorithms from batch-style algorithms, but these techniques still fall short even in simple situations—especially those where the change deletes information from the input.

This work revives an old idea: incrementalization is simply the task of finding the right data structure for the problem. By re-framing the incremental computation task as a data structure specification, a data structure synthesizer can produce efficient incremental algorithms. Compared to current state-of-the-art approaches, our technique can discover incremental algorithms that are asymptotically faster, with far less manual assistance.

5.1 Motivation

If an algorithm is slow or its input is large, recomputing from scratch when the input changes may be needlessly expensive. An incremental algorithm (Figure 5.1) updates the output efficiently when the input changes. When changes to the input data are much smaller than the whole input data, performing incremental updates can be much faster than recomputing from scratch. Replacing batch algorithms with incremental versions has demonstrated incredible speedups in big data [14], analytics processing [62], map-reduce jobs [71], database view maintenance [86], data flow analysis [104], graph algorithms [22], data structure design [73], compiler optimization [39], parsing [37], constraint solving [36], functional reactive programming [66], and other domains [78].

Incremental algorithms are notoriously difficult to implement because they require consistent updates to complex, optimized, inter-related, stateful data structures. A small bug can corrupt the data structure, and the correct output value will not be restored unless a full re-compute takes place. Fortunately, incremental computations have a simple specification: they must do exactly what recomputing from scratch would do. Decades of research have yielded both dynamic and static techniques to automatically derive correct incremental algorithms from their specifications, yet these techniques still fail to discover many transformations that are effective but non-obvious.

This work revives an old idea: incrementalization is simply the task of finding the correct data structure to maintain the derived state. This is not always an obvious transformation.

Data structure selection and incrementalization are closely related problems [60]: a good data structure efficiently computes values with respect to a history of incremental updates. For example, a heap structure can compute the minimum element of a set with respect to a history of incremental add and remove operations. Conversely, an incremental version of a function to find the minimum element of a set can use a heap as an efficient data structure underneath.

To illustrate, consider a service that maintains a set of users and must always be ready to find the oldest user:

$$oldest\_user(users) = \arg\max_{u \in users} age(u) \ . \tag{5.1}$$

Here are four increasingly-efficient ways to implement the service:

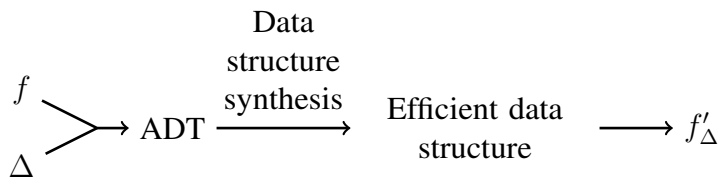


Figure 5.2: Our approach to incrementalization. Inch translates the target function and all relevant changes to the input into an abstract data type. A data structure synthesizer then creates an efficient implementation which serves as a backing data structure for f'_Δ.

1. Recompute the oldest user on every call to $oldest\_user$.

2. Cache the oldest user, but recompute it every time the set of users has changed since the last call.

3. Cache the oldest user as before, but update it efficiently when a new user is added. Only recompute when a user is deleted.

4. Cache the oldest user as before, but additionally store the users in a max-heap to efficiently handle user deletion.

Modern techniques can derive strategy 3 given the specification in Equation 5.1. However, as obvious as strategy 4 appears to any programmer who has heard of a max-heap, modern techniques cannot discover it [60, 85]. While modern techniques can discover tree-like structures to implement this service [45, 43], doing so requires the programmer to structure their computation as a reduction on a tree data structure, making the incremental algorithm highly dependent on the code in the specification and limiting interoperability with existing code.

Current state-of-the-art incrementalization techniques cannot discover auxiliary data structures unless they explicitly appear in the specification [60, 85]. In the oldest-user service, the specification mentions only the set of all users and the oldest user, not a heap.

While previous techniques treat the incrementalization problem as the task of finding a new incremental version of an algorithm, we view it as two interrelated tasks: finding both a new repre-

sentation for the input data and finding an incremental algorithm that uses the new representation. In this light, the shortcoming of previous techniques is that they do not explore many possible new representations for the data, only those provided by the programmer.

We call the new representation the *hidden state* of the incremental algorithm, since client code does not interact with it directly. To discover a good representation for the hidden state, we leverage recent work on the data structure synthesis problem [64]. Data structure synthesizers can discover new data representations to accelerate an algorithm. By re-framing the incrementalization problem as a data structure specification and leveraging a data structure synthesizer, we can produce efficient incremental algorithms in far more situations (Figure 5.2).

For example, strategy 4 can be discovered by framing Equation 5.1 as an abstract data structure:

$$\begin{aligned}
&\textbf{state } users : \text{Set}\langle\text{User}\rangle \\
&\textbf{query } \text{oldest_user}() = \arg\max_{u \in users} \text{age}(u) \\
&\textbf{op } \text{add_user}(u : \text{User}) = users.\text{add}(u) \\
&\textbf{op } \text{remove_user}(u : \text{User}) = users.\text{remove}(u)
\end{aligned}$$

Now, a data structure synthesizer can discover the expected solution. For example, Cozy [64] gives the output shown in Figure 5.3.

This paper describes Inch, a system that extends data structure synthesis to solve the incrementalization problem in general. We evaluated Inch against the current state-of-the-art.

5.2 Programming Incrementally

Consider the task of computing the latest due date over all issues in an issue tracker. In the Redmine issue tracker [101], this operation is implemented using the Ruby code

```
651  # The latest due date of an issue or version
652  def due_date
653    @due_date ||= [
654      issues.maximum('due_date'),
655      shared_versions.maximum('effective_date'),
656      Issue.fixed_version(shared_versions).maximum('due_date')
657    ].compact.max
658  end
```

Due to the line `issues.maximum('due_date')`, this computation takes linear time in the number of issues and is a known performance bottleneck in the Redmine issue tracker [100]. Redmine is not

```
state best_user : User
state users_not_empty : Bool
state h : MaxHeap⟨User⟩
state num_users : Int
query oldest_user() = best_user
op add_user(u : User) =
  let _var22 ≔ users_not_empty ? max_age(best_user, u) : u
  let _var23 ≔ (num_users + 1)
  best_user ≔ _var22
  users_not_empty ≔ True
  h.add_all(num_users, {u})
  num_users ≔ _var23
op remove_user(u : User) =
  let _var24 ≔ (u = best_user) ? h.second_max() : best_user
  let _var25 ≔ (num_users − 1)
  best_user ≔ _var24
  users_not_empty ≔ (1 < num_users)
  h.remove_all(num_users, {u})
  num_users ≔ _var25
```

Figure 5.3: Cozy's output on the oldest-user problem.

alone: out of a random sample of 140 performance issues across twelve of the most popular Ruby web applications, researchers found 18 similar issue reports [103]. Redmine was one of the web applications they investigated, but the maximum due date issue was not in their random sample.

Even though the issue was reported a year ago, this bottleneck still exists in Redmine. The user who reported the bottleneck included a patch that caches the maximum due date, but the developers rejected the patch since it incorrectly updates the cached maximum when the due date for an individual issue changes.

Caching the maximum issue due date is an example of incremental computation. The computation needs to be updated with respect to many small changes to the Redmine database, such as opening a new issue, deleting an issue, and changing an issue due date.

The difficulty that the original issue reporter had constructing a correct patch illustrates the challenge of manual incrementalization. Incrementalization requires new state and new code in many

different locations to maintain that state. Maintaining extra state complicates the program: the pure computation in the `due_date` function would disappear from Redmine altogether and be replaced by stateful operations in disparate locations that maintain the cached maximum due date. This in turn makes later maintenance harder: changing the definition of `due_date` (say, because of changes to other data structures or introduction of new variables such as `issues` and `shared_versions`) may require adjustments at all those disparate locations.

Previous approaches to automatic incrementalization are not suitable since they cannot discover useful data structures like heaps. There are both dynamic and static approaches. Dynamic approaches are based on memoization, so they incur a high memory overhead. Their behavior is highly dependent on how the programmer wrote the original code: if there is no heap in the original code, there will be no memoized heap. Previous static approaches are also dependent on how the programmer wrote the original code, since they use subexpressions of the code to determine which state to store between calls. Discovering new state that is not immediately obvious from the code has been a longstanding dream for researchers [60, 85].

There are also other practical barriers that a proposed solution needs to overcome. First, the solution should work with imperative code. While many automatic incrementalization papers target functional languages, the techniques really shine in imperative languages like Ruby where function inputs are often stateful objects. Second, the solution needs to handle pointers. Changing an issue's due date is a modification to the issue, and in Redmine the change is implemented as a write to a heap-allocated `Issue` object. Third, the solution needs to interoperate with a large body of existing code. Some previous incrementalization techniques have achieved speed by requiring the programmer to change the input types to the function being incrementalized [77, 45], which could be a difficult refactor in a large project like Redmine.

Our key insight is that incrementalizing an algorithm is ultimately the task of designing a data structure to store appropriate information between calls to the algorithm. Our approach is to ask a data structure synthesizer to decide what state to maintain and how to maintain it in order to automatically incrementalize algorithms. This is effective because the purpose of a data structure synthesizer is to discover non-obvious ways to store data. Our technique can handle complex

imperative updates—even those including pointers—and it interoperates with a project's existing types and functions.

5.3 Incrementalizing Algorithms by Synthesizing Data Structures

This section describes how Inch converts incremental computation problems into data structure specifications.

Given a pure function $f : In \to Out$ and an update function $\Delta : In \to In$, an automatic incrementalizer produces a function $f'_\Delta : Out \to Out$ such that for all inputs i,

$$f(\Delta(i)) = f'_\Delta(f(i)) \; .$$

Figure 5.1 illustrates this relationship. Intuitively, this process *differentiates* [75, 18] the function to obtain a new function describing changes to the output $f(i)$ in terms of changes to i.

Note that the update function Δ may accept inputs other than i—such as the argument u to $remove\_user$ in Section 5.1. Furthermore, Δ is often specified as an imperative update. Section 5.3.1 shows our approach in a simpler world without these concerns, while Section 5.3.2 discusses these concerns in more detail.

More concretely, consider the example from Section 5.1, the task of finding the oldest user from among a set. A concrete example of the incrementalization problem is

$$\begin{aligned} i = \text{a set of users} \quad & \{\textsc{Adam}, \textsc{Jared}, \textsc{Methuselah}\} \\ f = oldest\_user \quad & f(i) = \textsc{Methuselah} \\ \Delta = remove\_Methuselah \quad & \Delta(i) = \{\textsc{Adam}, \textsc{Jared}\} \\ f'_\Delta = oldest\_after\_removal \quad & f'_\Delta(i) = \textsc{Jared} \end{aligned}$$

As illustrated by the examples in Section 5.1 and Section 5.2, the old output $f(i)$ is often not enough to efficiently compute the new output $f(\Delta(i))$. *Hidden state* is needed. Many previous approaches [76, 1, 2, 60] including state-of-the-art approaches [45, 43, 7, 18, 53] derive the hidden state from subexpressions in f. We find this approach insufficient, and instead we aim to search for non-obvious representations for the hidden state.

5.3.1 Discovering Useful Hidden State

Our observation is that the hidden state in f'_Δ acts like a data structure. Upon seeing an input update $i' \coloneqq \Delta(i)$, the state must be updated in preparation to compute the new result $f(i')$.

The incrementalization task, then, is not simply to find an implementation of f'_Δ; it is to find a good choice for the hidden state and *also* an implementation of f'_Δ in terms of that hidden state. Previous approaches chose subexpressions of f as hidden state. This is a pragmatic choice: the subexpressions are often useful, and it reduces the problem from two difficult tasks to one. However, it is a naive approach since useful representations are often missing from the definition of f.

Fortunately, there now exist data structure synthesizers that can solve these two interrelated tasks together [46, 47, 65, 64]. Solving them together leads to better results since the two problems are so closely tied to each other; the representation should be tailored to the algorithm and the algorithm needs to effectively employ the representation. These tools are the latest in a long line of work that aims to separate the high-level description of what should be computed from the low-level details of how to compute it [83, 28, 82, 23, 90, 8].

We have reformulated the incrementalization problem as producing three functions

- $init : In \to HiddenState$,
- $update : HiddenState \to HiddenState$, and
- $query : HiddenState \to Out$

satisfying the requirements

$$\forall i, query(init(i)) = f(i) \tag{5.2}$$

$$\forall i, update(init(i)) = init(\Delta(i)) \tag{5.3}$$

This new formulation describes an abstract data type, and therefore can take advantage of powerful data structure synthesizers. Figure 5.4 shows how to create a data structure specification from f and Δ. The input type for f is the type of the private state on the data structure; since it is

Input (incrementalization problem):

$f : In \to Out$ function to be incrementalized
$\Delta : In \to In$ an update function

Output (data structure synthesis problem):

```
state hidden_state : In
constructor(input : In) =
  hidden_state := input
query q() =
  f(hidden_state)
op d() =
  hidden_state := Δ(hidden_state)
```

Figure 5.4: Converting an incremental computation problem into a data structure specification. A data structure synthesizer can solve the latter problem, including selecting a better representation for the hidden state.

private, the data structure synthesizer is free to replace this type with a better one. The computation f becomes a query operation (*i.e.* one that does not modify the private state), and the update Δ becomes an update operation (*i.e.* one that *does* modify the private state).

5.3.2 Handling Imperative Mutations

The formalism above treats the update Δ as a pure function of the old input. It is usually more natural to specify Δ as an imperative procedure that may take some arguments and modifies some program state. Fortunately, imperative procedures can be rewritten as pure ones.

Our specification language (Figure 5.5) allows Δ to be written as a procedure that takes arguments $\vec{a}$ and may modify any of the private state variables or modify the values of pointers. The procedure can have multiple statements as well as if-conditions, but no loops. Given such a procedure, our tool needs to reason about how the procedure affects the state to enforce Equation 5.3.

To do this, we introduce "pre-state inference" $(s; \text{ret } e) \approx e'$ shown in Figure 5.6. The inference relates a statement s and an expression e to an expression e' such that evaluating e' before executing s gives the same result as evaluating e after executing s. The rules are implicitly quantified over all

$Inch\_spec$	::=	$extern_1, extern_2, ...,$	
		$f, \Delta_1, \Delta_2, ...$	
$extern$	::=	type id = Native "$string$"	
	|	extern $id(args...)$ = "$string''$	
f	::=	τ $id(args...)$:	pure functions
		assume e;	
		return e;	
Δ	::=	void $id(args...)$:	updates
		assume e;	
		$stmt$;	
$stmt$	::=	...	
	|	$*e := e$	pointer assignment
e	::=	...	
	|	$*e$	pointer dereference

Figure 5.5: Inch's specification language, an extension of Cozy's core language (Figure 2.6). Heap and map operations are not part of the specification language, but they are in Inch's output language.

environments; we might have written the rule $\Gamma \vdash (s; \text{ret } e) \approx e'$, but since the rules do not interact with the environment we opted for a more concise form in the inference rules.

The inference is syntax-directed on the structure of s, making it easy to compute. It is akin to weakest precondition inference [29], since it infers an expression describing e in the pre-state of s.

We use pre-state inference to convert imperative mutations into pure functions. However, applying this inference to Equation 5.3 requires a more pedantic view of the formula: pre-state inference examines the structure of both the statement s and the expression e, meaning that we need the syntax trees for Δ, $init$, and $update$ in order to use it.

Supposing that Δ and $init$ are function-type expressions and i is a fresh variable, Equation 5.3 might be better written

$$\forall \Gamma, [\Gamma \downarrow update(init(i))] = [\Gamma \downarrow init(\Delta(i))] \tag{5.4}$$

where $[\Gamma \downarrow e]$ denotes evaluating expression e in environment Γ and the formulas $update(init(i))$ and $init(\Delta(i))$ denote the construction of call-nodes in a syntax tree, not mathematical function calls.

If instead Δ_s is a procedure (a syntax tree that takes arguments and runs a statement), using pre-state inference and Equation 5.4, then using fresh variables $\vec{a}$ and i we can convert the requirement into

$$\forall \Gamma \forall e', (\Delta_s(\vec{a}); \text{ret } init(i)) \approx e') \implies$$
$$[\Gamma \downarrow update(init(i))] = [\Gamma \downarrow e'])$$

meaning that if e' is an expression to compute the value of the hidden state after executing Δ_s, then evaluating the expression $update(init(i))$ should be equivalent to evaluating e'. The update operator d in Figure 5.4 becomes

$$\textbf{op } d(\vec{a}) =$$
$$hidden\_state \coloneqq e'$$

where $(\Delta_s; \text{ret } init(i)) \approx e')$.

The definition of pre-state inference is slightly subtle. The rule for ASSIGN-VAR examines the structure of e to perform capture-avoiding substitution. Similarly, the rule for ASSIGN-PTR examines the structure of e to replace all pointer dereferences with conditionals: if the pointer is the one being written to, then the dereference produces the new value, otherwise it produces the old value.

The rule for SEQ appears flipped, but is in fact correct. Pre-state inference is a backwards analysis. Consider how the procedure $x \coloneqq 1;\ x \coloneqq 2$ modifies the expression x. We expect the inference to tell us that the final output is 2, since the second write to x overwrites the first. The backwards SEQ rule works out as follows:

$$\frac{(x \coloneqq 2; \text{ret } x) \approx x[x \mapsto 2] \qquad (x \coloneqq 1; \text{ret } 2) \approx 2[x \mapsto 1]}{(x \coloneqq 1;\ x \coloneqq 2; \text{ret } x) \approx 2}$$

Using pre-state inference allows our tool to easily handle complex imperative mutations.

5.4 Extensions to Cozy

Inch uses Cozy [64] as its data structure synthesis engine. We found that it needed extensions in order to handle our new application of it.

$$\frac{}{(x \coloneqq y; \text{ret } e) \approx e[x \mapsto y]}\textsc{Assign-Var}$$

$$\frac{}{(*p \coloneqq x; \text{ret } e) \approx e[*\phi \mapsto (\phi = p)\ ?\ x : *\phi]}\textsc{Assign-Ptr}$$

$$\frac{(*R \coloneqq \{id_0 : R.id_0, id_1 : R.id_1, ..., id : x, ...\}; \text{ret } e) \approx e'}{(R.id \coloneqq x; \text{ret } e) \approx e'}\textsc{Assign-Field}$$

$$\frac{(L \coloneqq L \cup \{x\}; \text{ret } e) \approx e'}{(L.add(x); \text{ret } e) \approx e'}\textsc{Insert} \qquad \frac{(L \coloneqq L - \{x\}; \text{ret } e) \approx e'}{(L.remove(x); \text{ret } e) \approx e'}\textsc{Delete}$$

$$\frac{(s_2; \text{ret } e) \approx e' \qquad (s_1; \text{ret } e') \approx e''}{(s_1; s_2; \text{ret } e) \approx e''}\textsc{Seq} \qquad \frac{(s_1; \text{ret } e) \approx e_1 \qquad (s_2; \text{ret } e) \approx e_2}{(\text{if } c \text{ then } s_1 \text{ else } s_2; \text{ret } e) \approx c\ ?\ e_1 : e_2}\textsc{If}$$

Figure 5.6: Rules for reasoning about imperative code. Given an imperative statement s and an expression e, the pre-state inference $(s; \text{ret } e) \approx e'$ means that executing s and then evaluating e is the same as evaluating e' before executing s.

Because it is open source, Cozy can be extended. We extended it with heaps (Section 5.4.1) and rewrite rules that improve its performance (Section 5.4.2).

5.4.1 Heaps

Cozy creates a new data structure by combining known primitives. Cozy's built-in list of primitives includes multisets and hash maps, but does not contain heaps. One of Cozy's main advantages is that it can be extended with additional primitives. We added heaps, enabling Cozy to use a heap as a component (a field) of a synthesized data structure. Adding a new primitive to Cozy requires (1) the interface: new types and operation signatures, (2) semantics for those operations, (3) cost estimation rules for those operations, (4) state maintenance sketches for the new types, and (5) compilation from the new operations to to Java and C++.

Types and Pure Operations The Heap type and pure operations are:

$$\begin{aligned}
\tau &::= \text{MinHeap}\langle\tau\rangle \mid \text{MaxHeap}\langle\tau\rangle \\
e &::= \text{MakeMinHeap}_{priority}\ e \mid \text{MakeMaxHeap}_{priority}\ e \\
&\quad \mid \text{HeapElems}\ e \mid \text{HeapPeek}\ e \mid \text{HeapPeek2}\ e
\end{aligned}$$

MakeMinHeap and MakeMaxHeap construct a heap out of the elements of e ordered according to some key function $priority$. For instance, constructing a heap of all users arranged by age would be done with $\text{MakeMinHeap}_{age}\ users$ or $\text{MakeMaxHeap}_{age}\ users$.

Heaps can be queried for their elements (HeapElems), minimum or maximum (HeapPeek), and second-minimum or second-maximum (HeapPeek2). HeapPeek2 allows Cozy to compute what the minimum of a min-heap would be after popping off the old minimum—but without doing the mutation. Like HeapPeek, the HeapPeek2 primitive is also $O(1)$ to compute.

HeapPeek2 is necessary because Cozy's synthesizer only reads from the pre-state version of the data structure during updates. This restriction makes Cozy more effective by keeping the search space small. It is documented in the Cozy implementation, but not in previous papers on Cozy [65, 64]. However, it means that extensions like ours need additional primitives like HeapPeek2 so that in the pre-state Cozy can compute values that could otherwise be computed in the post-state. As discussed in Section 5.3.2, our pre-state inference forms expressions in terms of the pre-state.

We also added imperative statements to modify heaps, and they are discussed in "State Maintenance" below.

Semantics Enumerative CEGIS requires a *verifier* that checks whether a candidate expression is correct. To use heaps, Cozy needs semantics for the pure heap primitives in the language of the verifier. Cozy uses Z3 [26] as its verifier, and Z3 accepts input in the SMT-LIB [11] language.

Instead of converting the heap primitives directly to SMT-LIB, we exploited the fact that Cozy's verifier already supports sets, tuples, $\arg\min$, and $\arg\max$. The new heap operations we added can be easily lowered into those primitives. Note that this lowering is only for the verifier; when Cozy

outputs Java or C++ code, it will use efficient native operations, not these lowered equivalents.

To lower a heap expression (denoted $[\![e]\!]$), each heap is replaced by a set of $\langle element, key \rangle$ tuples:

$$\begin{aligned}
[\![\text{MakeMinHeap}_{priority}\ e]\!] &= \{\langle x, [\![f]\!](x)\rangle \mid x \in [\![e]\!]\} \\
[\![\text{HeapPeek}\ e]\!] &= fst\left(\operatorname*{arg\,min}_{x \in [\![e]\!]} snd(x)\right) \\
[\![\text{HeapPeek2}\ e]\!] &= fst\left(\operatorname*{arg\,min}_{x \in [\![\text{HeapElems}\ e - \{\text{HeapPeek}\ e\}]\!]} snd(x)\right) \\
[\![\text{HeapElems}\ e]\!] &= \{fst(p) \mid p \in [\![e]\!]\}
\end{aligned}$$

The rules for max-heaps are symmetric; they use "$\arg\max$" instead of "$\arg\min$".

Cost Estimation Cozy's cost model contains a function $cost_Q$ that estimates the run time of an operation in terms of the cardinalities of different collections. We extended it for heaps.

Heap creation is an expensive operation:

$$cost_Q(\text{MakeMinHeap}_{priority}\ e) = cost_Q(e) + cost_Q(priority(x)) \times |e|\ .$$

where x is a fresh variable. HeapPeek, HeapPeek2, and HeapElems are constant-time operations, so their costs are each $1 + cost_Q(h)$, where h is the cost of the heap. Note that HeapElems is constant-time, despite returning a potentially very large set. Collections are returned by reference, and HeapElems simply upcasts the heap to a generic iterable type. In general, Cozy's cost model does not penalize expressions that return lots of elements; it penalizes the work done to find those elements. Operations like HeapElems or simply returning a variable that stores a big collection are very cheap, while operations like Filter or Σ that need to iterate over a collection are expensive.

State Maintenance Cozy's core synthesis code only works on pure expressions. While CEGIS can also be used to synthesize imperative statements [93], Cozy does not do so. Instead, it synthesizes imperative code by framing all imperative code as two steps: first a pure computation that determines

what change to apply, and second a hard-coded snippet called a *state maintenance sketch* that applies the change.

Cozy requires a state maintenance sketch for every possible type. The sketch can include both pure computations that will be optimized using CEGIS and also imperative statements that will be included in the output implementation.

To build a state maintenance sketch for heaps, we introduced three new kinds of statements to Cozy:

$$stm ::= h.add\_all(e, e) \mid h.remove\_all(e, e) \mid h.update(e, e)$$

Note that we do not need to provide Cozy with semantics for the statements add_all, $remove\_all$, and $update$. Cozy does not use synthesis to improve statements, and correct usage of these statements depends on the correctness of our state maintenance sketch.

Suppose that a synthesized data structure uses a heap as one of its fields h. Cozy has an invariant for h of the form

$$h = \text{MakeMinHeap}_{age}\, users\ .$$

Wherever a method on the data structure would change the heap's abstract state (its contents) from h to h', Cozy introduces state maintenance code to the body of the method preserve the invariant. Our state maintenance sketch for heaps asks Cozy to produce code to compute

$$\begin{aligned}
&toRemove \coloneqq \text{HeapElems}(h) - \text{HeapElems}(h') \\
&toAdd \coloneqq \text{HeapElems}(h') - \text{HeapElems}(h) \\
&toUpdate \coloneqq \{\langle x, h'.priority(e)\rangle \mid \\
&\qquad x \in \text{HeapElems}(h') \cap \text{HeapElems}(h), \\
&\qquad h.priority(x) \neq h'.priority(x)\} \\
&len_1 \coloneqq |\text{HeapElems}(h)| \\
&len_2 \coloneqq |\text{HeapElems}(h)| - |toRemove|
\end{aligned}$$

and then has Cozy insert the code

$$\begin{aligned}
&h.\text{remove_all}(len_1, toRemove) \\
&h.\text{add_all}(len_2, toAdd) \\
&\textbf{for}\ elem, newKey \in toUpdate : \\
&\quad h.\text{update}(elem, newKey)
\end{aligned}$$

to the method to maintain the heap.

As written, the pure computations in green are not efficient to compute. Instead, Cozy uses enumerative synthesis to find an efficient implementation for each one. In practice, most will have a very simple implementation, for instance a singleton set. Our state maintenance rules need only give the specification for these pure computations; Cozy's synthesis engine does the rest.

As discussed in Section 5.4.1, all pure computations are in terms of the pre-state before any updates have taken place.

There are two oddities in our state maintenance sketch: the strange $h.priority(e)$ syntax and the variables len_1 and len_2.

Note the syntax $h.priority(e)$ and $h'.priority(e)$ in the pure computations. These are not true AST nodes; instead, we must run an analysis to construct an expression for the priority function of h and h' given Cozy's current abstraction relation. This is a necessary wart since Cozy does not support true first-class functions, and so we cannot introduce a proper expression node with semantics for this operation.

Computing the set of elements whose keys changed is necessary to preserve correctness; the behavior of the priority function for a heap is allowed to change during an update method, since it may contain free variables. For instance, the expression

$$\text{MakeMaxHeap}_{\lambda u.(mode == \text{MAX})?age(u):-age(u)}\, users$$

could be used to construct a heap that is a max-heap when $mode = \text{MAX}$ and a min-heap otherwise. While there is seldom a use case for this, we saw no reason to restrict Cozy's search space by forbidding free variables in priority functions. Cozy similarly allows free variables in other functions, such as filter predicates.

The len_1 and len_2 variables are an optimization. At run time, a heap needs to know how many elements it contains to correctly update itself. Introducing len_1 and len_2 allows Cozy to decide how best to store or compute the count. In particular, it allows Cozy to share the count between multiple heaps or collections that have the same cardinality.

Compilation Our generated code for heaps does not use the native heap types in Java or C++. In generated code, Cozy's "heap" data structures are smarter than regular heaps: every heap has a map from elements to nodes in order to make its removals and updates faster. This is necessary since Cozy heaps need to support removal of any arbitrary element, not just the current minimum or maximum.

5.4.2 Fusing Enumerative and Rule-Based Search

We modified Cozy's brute-force synthesis procedure to leverage additional insights in the form of handwritten rewrite rules (Table 5.1). Cozy now applies these rules on every expression it enumerates, providing a convenient way to shortcut the search and discover good solutions more quickly. We found that these rules provide an enormous speedup (Section 5.5.2). This section describes the rules and how we modified Cozy to use them.

A prerequisite to understanding some of the rules is a recap of Cozy's "packed" representation for expressions (Section 3.2), illustrated in Figure 5.7. The packed representation is an implementation trick that allows Cozy to shoehorn both a new representation and an algorithm into a single expression, allowing it to use a simpler CEGIS procedure. Every expression in Cozy is tagged as a state expression or a query expression. State expressions implicitly represent new pieces of state that track their values.

For instance, the top expression in Figure 5.7 indicates that Cozy should store the sum of the $ints$ collection as a data structure member and should add one to it at run time. The bottom expression indicates that Cozy should store the $ints$ collection and the constant value 1 as members on the data structure, and at run time it should sum the $ints$ and add the stored constant to the result.

Cozy's static cost model works on packed expressions and uses the tags to determine the cost. Between the two expressions in Figure 5.7, the static cost model would prefer the first since it is $O(1)$ while the second is $O(|ints|)$.

Some of the rewrite rules in Table 5.1 depend on—or affect—the tags on packed expressions.

For instance, the rule

$$x \in \underbrace{ys}_{\text{state}} \rightarrow \underbrace{histogram(ys)}_{\text{state}}[x] > 0$$

says that if ys will be stored on the data structure and we need to be able to test containment ($x \in ys$), it would be better to store a histogram of ys that maps each element of ys to its cardinality in the collection, and do the containment check using a map lookup.

The first three rules in Table 5.1 are powerful rewrite schemes that can trigger on many different expressions. For instance, the first rule says that any expression being computed at run time should instead be stored on the data structure. This transformation is subject to well-formedness requirements on packed expressions: the state part of the expression is not allowed to mention method arguments that are only available at run time [64]. All other rewrite rules are subject to the same requirement.

When we introduced heaps to Cozy we also introduced a rewrite rule to help discover when they are useful:

$$\begin{aligned}\arg\min_f(xs - \{x\}) \rightarrow (x == \arg\min_f xs \\ ? \ HeapPeek2(MakeMinHeap(xs)) \\ : \arg\min_f xs)\end{aligned}$$

The rule says that the minimum of a collection after an element is removed is either the second minimum (if the minimum was removed) or the old minimum. Expressions like $xs - \{x\}$ are very common outputs from pre-state inference (Figure 5.6), making this rule quite useful.

The rewrite rules do not need to be efficient or even correct! For example, the rule

$$\Sigma(xs - \{x\}) \rightarrow \Sigma(xs) - x$$

is not universally correct: if $x \notin xs$, then $\Sigma(xs - \{x\}) = \Sigma(xs)$. This behavior is fine because Cozy uses an SMT solver to verify each possible solution against the entire specification before output, discarding incorrect solutions. As an optimization, CEGIS algorithms like Cozy's also do

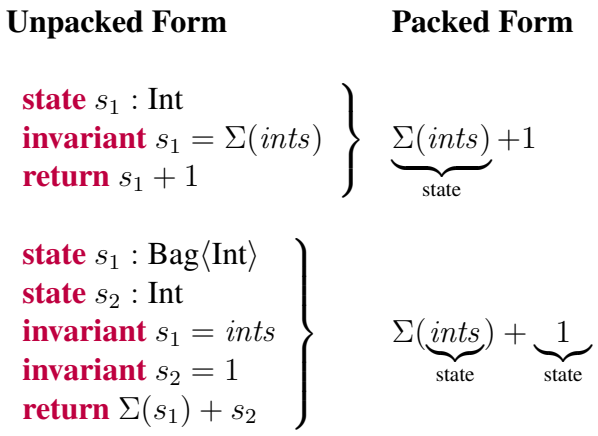


Figure 5.7: Two examples of expression packing in Cozy. While the two expressions on the right are syntactically identical, different tags (the blue color) cause them to have different meanings in the final data structure.

testing to weed out incorrect solutions more quickly and to group expressions by what value they compute [102, 9]. Cozy retains only the most efficient known expression that computes a given value.

5.4.3 Interoperability

A data structure specification may need to refer to types and functions that already exist in a project.

We extended Cozy with the ability to declare the existence of "external" types and functions. These external entities are *uninterpreted*: Cozy does not know anything about how the types or functions behave.

For example, the issues in Redmine can be declared as

type Date = **Native** "Date"

type Issue = **Native** "Issue"

extern $getDueDate(i$: Issue$)$: Date = "$\{i\}.due\_date$"

Cozy assumes that instances of an external type can be compared to each other. Cozy also assumes that external functions are pure and fast to evaluate. It is the programmer's responsibility

Table 5.1: New rewrite rules that we added to Cozy.

Trigger		**Rewrite**
x	$\rightarrow$	$\underbrace{x}_{\text{state}}$
$f(cond\ ?\ x : y)$	$\rightarrow$	$cond\ ?\ f(x) : f(y)$
$f(\underbrace{m}_{\text{state}}[x])$	$\rightarrow$	$\underbrace{m'}_{\text{state}}[x]$
$L[0]$	$\rightarrow$	$\text{First}\ L$
$\arg\min_f \emptyset$	$\rightarrow$	$\mathit{undefined}$
$\arg\min_f \{x\}$	$\rightarrow$	x
$\arg\min_f (xs \cup ys)$	$\rightarrow$	$\exists xs\ ?\ \arg\min_f(\{\arg\min_f xs\} \cup ys)\ :\ \arg\min_f ys$
$\arg\min_f (xs - \{x\})$	$\rightarrow$	$(x == \arg\min_f xs\ ?\ \mathit{HeapPeek2}(\mathit{MakeMinHeap}(xs))\ :\ \arg\min_f xs)$
$\arg\min_f (\text{Distinct}\ xs)$	$\rightarrow$	$\arg\min_f xs$
$\text{True}\ ?\ x : y$	$\rightarrow$	x
$\text{False}\ ?\ x : y$	$\rightarrow$	y
$\{..., f_i : x_i, ...\}.f_i$	$\rightarrow$	x_i
$x \in \emptyset$	$\rightarrow$	False
$x \in \{y\}$	$\rightarrow$	$x == y$
$x \in \underbrace{ys}_{\text{state}}$	$\rightarrow$	$\underbrace{\mathit{histogram}(ys)}_{\text{state}}[x] > 0$
$x \in (xs \cup ys)$	$\rightarrow$	$x \in xs \vee x \in ys$
$x \in (\underbrace{ys}_{\text{state}} - \{z\})$	$\rightarrow$	$(x == z)\ ?\ (\underbrace{\mathit{histogram}(ys)}_{\text{state}}[x] > 1) : x \in \underbrace{ys}_{\text{state}}$
$x \in (ys - zs)$	$\rightarrow$	$\mathit{histogram}(ys)[x] > \mathit{histogram}(zs)[x]$
$x \in (\text{Filter}_f\, ys)$	$\rightarrow$	$f(x) \wedge x \in ys$
$x \in (\text{Map}_f\, ys)$	$\rightarrow$	$\exists y \in ys, f(y) == x$
$\Sigma\{x\}$	$\rightarrow$	x
$\Sigma(xs \cup ys)$	$\rightarrow$	$\Sigma xs + \Sigma ys$
$\Sigma(xs - \{x\})$	$\rightarrow$	$\Sigma xs - x$
$\Sigma(xs - ys)$	$\rightarrow$	$\Sigma xs - \Sigma\text{Filter}_{\lambda y.\, y \notin xs}\, ys$
$\exists x \in \emptyset, P(x)$	$\rightarrow$	False
$\exists x \in \{y\}, P(x)$	$\rightarrow$	$P(y)$
$\exists x \in (xs \cup ys), P(x)$	$\rightarrow$	$(\exists x \in xs, P(x)) \vee (\exists x \in ys, P(x))$
$\exists x \in (xs - ys), P(x)$	$\rightarrow$	$(\exists x \in xs, P(x)) \wedge \neg(\exists x \in ys, P(x))$
$\exists x \in \text{Map}_f xs, P(x)$	$\rightarrow$	$\exists y \in xs, P(f(y))$
$\exists x \in \text{Filter}_f xs, P(x)$	$\rightarrow$	$\exists y \in xs, f(y) \wedge P(y)$
$\exists x \in xs, x \notin ys$	$\rightarrow$	$\lvert xs\rvert > \lvert ys\rvert$
$\exists x \in xs, P(x) \vee Q(x)$	$\rightarrow$	$(\exists x \in xs, P(x)) \vee (\exists x \in xs, Q(x))$

Table 5.1: New rewrite rules that we added to Cozy (continued).

$\text{First}(L - \{x\})$	$\rightarrow$	$x == L[0]\ ?\ L[1] : \text{First}\ L$
$\text{First}\,\text{Map}_f xs$	$\rightarrow$	$\exists xs\ ?\ f(\text{First}\ xs) : \mathit{undefined}$
$\text{First}(xs \cup ys)$	$\rightarrow$	$\exists xs\ ?\ (\text{First}\ xs) : (\text{First}\ ys)$
$\text{Distinct}\,\emptyset$	$\rightarrow$	$\emptyset$
$\text{Distinct}\{x\}$	$\rightarrow$	$\{x\}$
$\text{Distinct}(xs \cup ys)$	$\rightarrow$	$\text{Distinct}\ xs \cup \text{Filter}_{\lambda y.y \notin \text{Distinct}\ xs}\ \text{Distinct}\ ys$
$\text{Distinct}(xs - ys)$	$\rightarrow$	$\text{Filter}_{\lambda x.x \notin \text{Distinct}\ ys}\ \text{Distinct}\ xs$
$\text{Distinct}\,\text{Filter}_f xs$	$\rightarrow$	$\text{Filter}_f\ \text{Distinct}\ xs$
$\text{Filter}_{\lambda x.\text{True}} xs$	$\rightarrow$	xs
$\text{Filter}_{\lambda x.\text{False}} xs$	$\rightarrow$	$\emptyset$
$\text{Filter}_f \emptyset$	$\rightarrow$	$\emptyset$
$\text{Filter}_f \{x\}$	$\rightarrow$	$f(x)\ ?\ \{x\} : \emptyset$
$\text{Filter}_f (xs \cup ys)$	$\rightarrow$	$(\text{Filter}_f xs) \cup (\text{Filter}_f ys)$
$\text{Filter}_f (xs - ys)$	$\rightarrow$	$(\text{Filter}_f xs) - (\text{Filter}_f ys)$
$\text{Filter}_{\lambda x.x==y} \underbrace{xs}_{\text{state}}$	$\rightarrow$	$y \in xs\ ?\ \{y\} : \emptyset$
$\text{Filter}_{\lambda x.f(x)==g(y)} \underbrace{xs}_{\text{state}}$	$\rightarrow$	$\underbrace{m}_{\text{state}}[g(y)]$
$\text{Filter}_{\lambda x.P(x) \vee Q(x)} xs$	$\rightarrow$	$(\text{Filter}_P xs) \cup (\text{Filter}_{\lambda x.\neg P(x) \vee Q(x)} xs)$
$\text{Filter}_{\lambda x.P(x) \wedge Q(x)} xs$	$\rightarrow$	$\text{Filter}_P \text{Filter}_Q xs$
$\text{Map}_{\lambda x.x} xs$	$\rightarrow$	xs
$\text{Map}_f \emptyset$	$\rightarrow$	$\emptyset$
$\text{Map}_f \{x\}$	$\rightarrow$	$\{f(x)\}$
$\text{Map}_f (xs \cup ys)$	$\rightarrow$	$(\text{Map}_f xs) \cup (\text{Map}_f ys)$
$\text{Map}_f (xs - ys)$	$\rightarrow$	$(\text{Map}_f xs) - (\text{Map}_f ys)$

to ensure that these assumptions hold.

5.5 Evaluation

5.5.1 Benchmarks

To permit direct comparison to previous work, we reimplemented the set of functions that several previous papers examine [5, 57, 42, 44, 45, 43].

Each benchmark in Table 5.2 repeatedly computes a map or fold operation over a list, with side effects performed in between map/fold operations. “Remove/replace” changes allow removal of list elements and replacement of removed elements. “Swap” changes allow cutting the list into two

Table 5.2: Benchmarks from [45]. Adapton's numbers are from the original paper [45]; we did not recompute them.

		Speedup		Memory	
f	Δ	Adapton_T	Inch	Adapton_T	Inch
Filter[0]	r/r	12.8×	843.3×	+170%	+34%
Filter	r/r	2.0×	2.4×	+910%	+54%
Filter	swp	2.0×	2.5×	+910%	-38%
Map[0]	r/r	7.8×	2386.1×	+170%	+45%
Map	r/r	2.2×	3.1×	+590%	+78%
Map	swp	2.4×	3.4×	+590%	-56%
Sum	r/r	1640.0×	2308.9×	+810%	+4%
Sum	swp	501.0×	5266.3×	+810%	-63%
Min	r/r	2020.0×	1386.5×	+770%	+10%
Min	swp	472.0×	5427.7×	+810%	-63%

Table 5.3: Asymptotic performance on the benchmarks.

		Query		Update	
f	Δ	Baseline	Inch	Baseline	Inch
Filter[0]	r/r	n	1	n	n
Filter	r/r	n	1	n	n
Filter	swp	n	1	n	n
Map[0]	r/r	n	1	n	1
Map	r/r	n	1	n	n
Map	swp	n	1	n	n
Sum	r/r	n	1	n	1
Sum	swp	n	1	n	1
Min	r/r	n	1	n	$\log n$
Min	swp	n	1	n	1

sub-lists at a given cut point and exchanging the two sub-lists.

These workloads were also used to evaluate Adapton [45], a state-of-the-art library that helps programmers incrementalize their programs. We chose to compare against Adapton since it is recently published, outperforms many earlier incrementalization approaches, and has an easily reproducible benchmark methodology. While an improved version of Adapton called Nominal Adapton was later published [43], we did not compare against Nominal Adapton since it is not a fully automatic solution. Nominal Adapton requires much more insight from the programmer in the form of "names" attached to computations.

Methodology. To permit direct comparison, we adopt the Adapton methodology without change. For each benchmark, we wrote a specification indicating the function to be computed and the allowed changes to the data structure. We ran Inch with a timeout of 15 minutes to synthesize a C++ implementation. (Cozy is an "any time" algorithm that runs for a set amount of time.) We also implemented, in C++, a baseline that recomputes from scratch every time the function is called. The methodology computes time and memory relative to the baseline.

We used Apple LLVM version 9.1.0 (clang-902.0.39.2) to compile the resulting programs. We ran each one 5 times, measuring median user CPU time and peak memory usage (as reported by GNU Time [97] version 1.9) across any run.

Table 5.2 also gives results from the Adapton evaluation [45] for comparison. While Inch and Adapton target different languages, the benchmarks and workloads are functionally identical, so the *relative* speedups and memory overheads can be directly compared.

Results. Table 5.2 shows that Inch's solution is always faster than recomputing from scratch. Inch's speedup always surpasses Adapton's. Inch's memory overhead (median ~10%) is significantly lower than Adapton's high overheads (median ~800%).

Some memory overhead is expected since any incremental version of the algorithm needs to store additional indexing data to make the query and update fast. In almost half the cases, however, Inch achieves a decrease in memory by eliminating the underlying list. For instance, the swapping

change does not affect the sum of the list, so for "Sum(swp)" Inch computes the sum once on $init$, discards the list, and turns $update$ into a no-op.

One strength of our approach is that a powerful data structure synthesizer like Inch can specialize itself to the structure of each problem. For instance, we noticed that the remove/replace benchmarks do not allow two removals in a row—updates must alternate between remove and replace. We wrote the specification to reflect this, and Inch exploited the remove/replace restriction in the Map[0] benchmark. On that benchmark, the only "remove" operations that matter are the ones that remove the first element of the list. All others do not affect the output. Therefore, Inch simply stores a Boolean indicating whether the first element was removed. If multiple removals were possible in a row, Inch would synthesize a more complicated implementation since a client could remove, say, all of the first five elements of the list.

Concrete speedup numbers are not particularly informative. Since both Inch and Adapton make asymptotic improvements over the baseline, the numbers can be made arbitrarily large or small by changing the size of the benchmark data. Therefore, Table 5.3 gives the asymptotic complexity of each implementation, in big-O notation where n is the size of the list. The table compares the baseline to the solution found by Inch. While Inch does not find an optimal solution in all cases, it does heavily optimize the query part of the problem and performs no worse than the baseline. In practice its solutions are very effective.

Unlike Adapton, Inch is able to accept input in many different forms and completely transform it in the conversion to hidden state. As a result, algorithms can be specified with more natural types and better interoperate with existing code. Adapton's implementation of the benchmarks, for instance, does not interoperate with ML's standard list library.

5.5.2 Effect of Rewrite Rules

We added many new rewrite rules to Cozy (Section 5.4.2). To evaluate their effect, we compared Inch's performance on the benchmarks with Cozy in three different modes: enumerative synthesis only, rewrites only, and both. We ran Cozy with timeouts varying up to three hours and measured total speedup on the benchmarks for each timeout.

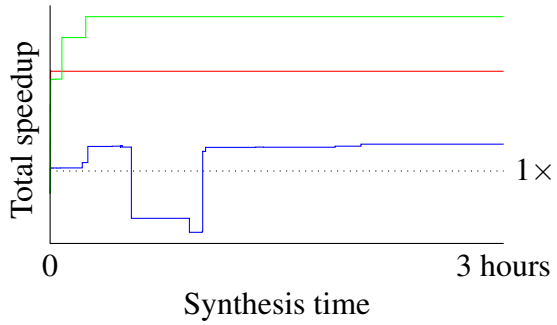


Figure 5.8: The effect of our new rewrite rules (Section 5.4.2): total speedup for the whole benchmark suite in three different configurations. The blue line is enumeration only without the rewrite rules. The red line is rewrite rules only, without enumerative synthesis. The green line is Cozy, which does both.

Figure 5.8 shows that Cozy performs best with both rewrite rules and enumerative synthesis. With rewrite rules only, Cozy quickly discovers a good solution for each benchmark but then stops finding improvements. With both enumerative synthesis and rewrite rules, Cozy is able to continue finding non-intuitive improvements.

Cozy performs poorly without the rewrite rules, even with three hours to synthesize a data structure. In principle Cozy will eventually discover solutions as good as the ones found by rewrites, but in practice it does not do so in a reasonable amount of time. Interestingly, with enumeration only, Cozy does worse than the baseline for a while. This is because Cozy's cost model is applied locally on each method, and sometimes improvements to one method change the data representation and make the global run time worse. Our rewrite rules help insulate from this problem by short-cutting past bad solutions.

Chapter 6

RELATED WORK

Several bodies of work relate to the approach outlined in this document: 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.

6.1 Data Structure Synthesis

Data structure synthesis has been an active area of research for many years. One class of techniques stems from Earley's work on iterator inversion [31], in which high-level set comprehensions can be rewritten automatically to use more efficient data structures. Subsequent work generalized these ideas [35, 75]. 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 [83, 81, 82, 17]. 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 [84].

Specialized languages for data structure implementation have also been proposed [12, 89, 90]. 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 [46]. 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 [47], which we do not address here.

6.2 Synthesis

Our work builds on existing CEGIS techniques for synthesizing individual functions [93]. 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. λ^2 [33] 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 sub-components, the enumeration algorithm used in λ^2 is not applicable. SYNAPSE [15] 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.

6.3 Data Structure Generation

Researchers have investigated automatic data structure generation for testing [67, 16, 25, 38]. 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.

6.4 Databases and Query Planning

Cozy's expression language (Figure 2.6) can be viewed as a planning language for executing retrieval operations. 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 [19, 6], 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 [7]. 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.

6.5 Early Work

The data structure synthesis problem dates to the 1970s and *iterator inversion*, a technique for constructing data structures to accelerate iterative operations [30, 32]. 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 [83, 28, 82]. More recently, researchers have investigated dynamic techniques to achieve the same effect [84]. Beyond simply choosing better existing implementations of an interface, Cozy can implement more complex interfaces that require composing data structure representations.

6.6 Synthesis for Data Structures

Modern program synthesis techniques have been applied to low-level data structure code [94, 88]. 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 [46] constructs data structure implementations that track subsets of a collection. It was later extended to produce safe concurrent data structures [47]. An earlier version of Cozy [65] 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 re-implement them to extend Cozy's applicability.

6.7 Programming by Refinement

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 [91], Designware [92], and Fiat [27], 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 [91].

6.8 Databases and View Maintenance

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 [19, 6] 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 [7] implements a very efficient view maintenance system. More recently, the same team has worked on generalizing these ideas to collections, including nested collections [53]. 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.

6.9 Incremental Computation

There is a deep connection between data structures and incremental computation. In fact, many data structures exist solely to incrementalize certain operations; a heap, for instance, incrementalizes the computation of a minimum or maximum element of a collection with respect to insertions and removals. This relationship was noted by Paige and Koenig [75], who observed that iterator inversion is actually a special kind of incrementalization.

Assisted Techniques Our work considers aims for fully automatic incrementalization. There are also assisted techniques that, with some interaction between human and computer, achieve impressive results. Assisted techniques can be much more expressive than our system, but require much more effort from the programmer.

Memoization [13, 68, 69] is perhaps the simplest form of incremental computation. By manually altering their program to cache outputs, programmers can achieve a degree of incrementality.

Programming by refinement systems like the SETL language [28], KIDS [91], and Designware [92] make incrementalization possible. Fiat [27] is a more recent programming by refinement system for the Coq proof assistant that enables programmers to produce fully verified data structures. Fiat automates a great deal of low-level reasoning, but still relies on many refinement steps and

insights from the programmer. It has a larger output space than our tool but is not fully automatic.

CACHET [60] implements a library of incrementalization strategies and mostly-automatic techniques. However, CACHET's automated techniques only consider subexpressions of the input function as additional hidden state.

Incrementalization Libraries There are several libraries to help programmers write incremental algorithms. The most notable of these are in the family of self-adjusting computation (SAC) [2, 4, 5, 54]. If programmers write their algorithms using the SAC library instead of the standard library for their language, then the library can propagate changes from inputs to outputs by building a large dependency graph among computations at run time.

While SAC is different from our approach and requires more manual effort to port existing codebases, there are ways it can influence our work. In particular, the developers recognized the need for imperative programming [3], which our pre-state inference is meant to address, and for cost optimization [55], which Cozy handles for us.

Adapton [45, 43] is similar to other SAC libraries, but makes the dependency graph somewhat lazy. This allows the library to avoid pushing changes through the dependency graph unless they are needed by a client, making the library much more efficient in many cases. Non-monotonic self-adjusting computation [56] also improves the typical SAC approach by relaxing the restriction that recomputes happen in the same order as the original computations.

Naiad [70], a dataflow system, is a library that offers a very simple abstraction for building distributed incremental computations. Like other library-based approaches, Naiad requires the programmer to translate their task into calls to the library.

Static Incrementalization Static techniques take a batch-style program as input and produce an incremental version as output. Our approach is a static technique.

There are many static approaches that infer ways to memoize code [1, 48, 61, 76, 78, 34]. Alphonse [49] is a notable example of such a system. These systems can, by design, only re-use computations present in the input function. Another key limitation of these systems is that they

can only re-use outputs from sub-computations when the inputs are exactly the same. Flipping the operands to a commutative operator like + will incur a recompute. Our system can exploit the semantics of operations like + to avoid work in some cases.

IλC [18] is a more recent static incrementalization system. Unlike our approach, IλC uses exclusively the output of the function as hidden state. While it is able to support higher-order functions, it does not produce asymptotically good solutions for many examples because it does not even store intermediate computations of the function as hidden state.

DBToaster [7, 53], a view maintenance system for databases, statically incrementalizes functions expressed as SQL queries. DBToaster can use expressions and sub-queries from the input as additional materialized views (*i.e.* additional hidden state), but may still do work in cases like the Sum(swap) benchmark, even though no work needs to happen. We are able to discover these facts by using a CEGIS-backed synthesizer as a subroutine.

ICQ [59] handles queries in object-oriented languages like Java by converting the problem to relational logic and statically constructing a graph of relationships between collections. While capable of incrementalizing queries containing arbitrary filter predicates, it does not know the semantics of those predicates. Furthermore, the system cannot deal with queries that have reductions like minimums or sums.

Dynamic Incrementalization There has been work on automatically applying SAC to existing code [21, 20]. These are truly dynamic approaches that require no intervention from the programmer, but suffer from the high overheads of dynamic dependency graphs. However, they are able to handle more problems than our work.

Chapter 7

CONCLUSION

This thesis presents 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.

Cozy is effective because state maintenance 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.

BIBLIOGRAPHY

[1] Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 83–91. ACM, May 1996.
DOI: 10.1145/232627.232638.

[2] Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Carnegie Mellon University, 2005.

[3] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*, pages 309–322. ACM, January 2008.
DOI: 10.1145/1328438.1328476.

[4] Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. A library for self-adjusting computation. *Electronic Notes in Theoretical Computer Science*, 148(2):127–154, March 2006.
DOI: 10.1016/j.entcs.2005.11.043.

[5] Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*, pages 96–107. ACM, June 2006.
DOI: 10.1145/1133981.1133993.

[6] Sanjay Agrawal, Surajit Chaudhuri, and Vivek 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. Morgan Kaufmann, 2000.

[7] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *Proceedings of the VLDB Endowment*, 5(10):968–979, June 2012.
DOI: 10.14778/2336664.2336670.

[8] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. A framework for sparse matrix code synthesis from high-level specifications. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (SC '00)*. IEEE, November 2000.
DOI: 10.1109/SC.2000.10033.

[9] 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. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FM-CAD '13)*, pages 1–8. IEEE, October 2013.
DOI: 10.1109/FMCAD.2013.6679385.

[10] Alexandr Andoni, Dumitru Daniliuc, and Sarfraz Khurshid. Evaluating the "small scope hypothesis". Technical report, MIT, 2003.

[11] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The satisfiability modulo theories library (SMT-LIB), 2016. accessed May 18, 2018.
`http://www.smt-lib.org`.

[12] Don Batory, Vivek Singhal, and Marty Sirkin. Implementing a domain model for data structures. *International Journal of Software Engineering and Knowledge Engineering*, 2:2–3, 1992.
DOI: 10.1142/S021819409200018X.

[13] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[14] Pramod Bhatotia, Alexander Wieder, İstemi Ekin Akkuş, Rodrigo Rodrigues, and Umut A. Acar. Large-scale incremental data processing with change propagation. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing (HOTCLOUD '11)*, pages 18–18. USENIX Association, June 2011.

[15] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In *Proceedings of the 43rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages (POPL '16)*, pages 775–788. ACM, January 2016.
DOI: 10.1145/2837614.2837666.

[16] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko 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. ACM, 2002.
DOI: 10.1145/566172.566191.

[17] Jiazhen Cai, Philippe Facon, Fritz Henglein, Robert Paige, and Edmond Schonberg. Type transformation and data structure choice. Technical report, NYU, November 1988.

[18] Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pages 145–155. ACM, 2014.
DOI: 10.1145/2594291.2594304.

[19] Surajit Chaudhuri and Vivek 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. Morgan Kaufmann, 1997.

[20] Yan Chen, Joshua Dunfield, and Umut A. Acar. Type-directed automatic incrementalization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*, pages 299–310. ACM, June 2012.
DOI: 10.1145/2254064.2254100.

[21] Yan Chen, Joshua Dunfield, Matthew A. Hammer, and Umut A. Acar. Implicit self-adjusting computation for purely functional programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*, pages 129–141. ACM, September 2011.
DOI: 10.1145/2034773.2034792.

[22] Grant A. Cheston. *Incremental Algorithms in Graph Theory*. PhD thesis, University of Toronto, March 1976.

[23] Donald Cohen and Neil Campbell. Automating relational operations on data structures. *IEEE Software*, 10(3):53–60, May 1993.
DOI: 10.1109/52.210604.

[24] Cozy homepage.
`https://cozy.uwplse.org`.

[25] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of 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. ACM, 2007.
DOI: 10.1145/1287624.1287651.

[26] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS/ETAPS '08)*, pages 337–340. Springer, March 2008.
DOI: 10.1007/978-3-540-78800-3_24.

[27] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 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)*, pages 689–700. ACM, 2015.
DOI: 10.1145/2676726.2677006.

[28] Robert B. K. Dewar, Arthur Grand, Ssu-Cheng Liu, Jacob T. Schwartz, and Edmond Schonberg. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM Transactions on Programming Languages and Systems*, 1(1):27–49, January 1979. DOI: 10.1145/357062.357064.

[29] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall, March 1997.

[30] Jay Earley. Relational level data structures for programming languages. *Acta Informatica*, 2(4):293–309, December 1973. DOI: 10.1007/BF00289502.

[31] Jay Earley. High level operations in automatic programming. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 34–42. ACM, March 1974. DOI: 10.1145/800233.807043.

[32] Jay Earley. High level iterators and a method for automatically designing data structure representation. *Computer Languages*, 1(4):321–342, January 1975. DOI: 10.1016/0096-0551(75)90019-3.

[33] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*, pages 229–239. ACM, 2015. DOI: 10.1145/2737924.2737977.

[34] John Field and Tim Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*, pages 307–322. ACM, June 1990. DOI: 10.1145/91556.91679.

[35] Amelia C. Fong and Jeffrey 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. ACM, January 1976. DOI: 10.1145/800168.811544.

[36] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, January 1990. DOI: 10.1145/76372.77531.

[37] Carlo Ghezzi and Dino Mandrioli. Incremental parsing. *ACM Transactions on Programming Languages and Systems*, 1(1):58–70, January 1979. DOI: 10.1145/357062.357066.

[38] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko 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. ACM, 2010.
DOI: 10.1145/1806799.1806835.

[39] Allen Goldberg and Robert Paige. Stream processing. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP '84)*, pages 53–62. ACM, August 1984.
DOI: 10.1145/800055.802021.

[40] Shelly Grossman, Sara Cohen, Shachar Itzhaky, Noam Rinetzky, and Mooly Sagiv. Verifying equivalence of spark programs. In *Proceedings of the 29th International Conference on Computer Aided Verification, Part II (CAV '17)*, pages 282–300. Springer, July 2017.
DOI: 10.1007/978-3-319-63390-9_15.

[41] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP '10)*, pages 13–24. ACM, 2010.
DOI: 10.1145/1836089.1836091.

[42] Matthew A. Hammer, Umut A. Acar, and Yan Chen. CEAL: A C-based language for self-adjusting computation. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, pages 25–37. ACM, June 2009.
DOI: 10.1145/1542476.1542480.

[43] Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. Incremental computation with names. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '15)*, pages 748–766. ACM, October 2015.
DOI: 10.1145/2814270.2814305.

[44] Matthew A. Hammer, Georg Neis, Yan Chen, and Umut A. Acar. Self-adjusting stack machines. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*, pages 753–772. ACM, October 2011.
DOI: 10.1145/2048066.2048124.

[45] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pages 156–166. ACM, June 2014.
DOI: 10.1145/2594291.2594324.

[46] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data representation synthesis. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*, pages 38–49. ACM, June 2011.
DOI: 10.1145/1993498.1993504.

[47] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Concurrent data representation synthesis. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*, pages 417–428. ACM, 2012.
DOI: 10.1145/2254064.2254114.

[48] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*, pages 311–320. ACM, June 2000.
DOI: 10.1145/349299.349341.

[49] Roger Hoover. Alphonse: Incremental computation as a programming abstraction. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*, pages 261–272. ACM, June 1992.
DOI: 10.1145/143095.143139.

[50] Ignite Realtime. Openfire real time collaboration server, 2016. accessed March 28, 2017.
`https://www.igniterealtime.org/projects/openfire/`.

[51] Daniel Jackson and Craig Damon. Elements of style: Analyzing a software design feature with a counterexample detector. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '96)*, pages 239–249, 1996.
DOI: 10.1145/229000.226322.

[52] Richard M. Karp. *Complexity of Computer Computations: Proceedings of a Symposium on the Complexity of Computer Computations*, pages 85–103. Springer, 1972.
DOI: 10.1007/978-1-4684-2001-2_9.

[53] Christoph Koch, Daniel Lupei, and Val Tannen. Incremental view maintenance for collection programming. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '16)*, pages 75–90. ACM, 2016.
DOI: 10.1145/2902251.2902286.

[54] Ruy Ley-Wild. *Programmable Self-Adjusting Computation*. PhD thesis, Carnegie Mellon University, October 2010.

[55] Ruy Ley-Wild, Umut A. Acar, and Guy Blelloch. Non-monotonic self-adjusting computation. In *Proceedings of the 21st European Conference on Programming Languages and Systems (ESOP '12)*, pages 476–496. Springer, March 2012.
DOI: 10.1007/978-3-642-28869-2_24.

[56] Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*, pages 186–199. ACM, January 2009.
DOI: 10.1145/1480881.1480907.

[57] Ruy Ley-Wild, Matthew Fluet, and Umut A. Acar. Compiling self-adjusting programs with continuations. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*, pages 321–334. ACM, September 2008.
DOI: 10.1145/1411204.1411249.

[58] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59. ACM, March 1974.
DOI: 10.1145/800233.807045.

[59] Yanhong A. Liu, Jon Brandvein, Scott D. Stoller, and Bo Lin. Demand-driven incremental object queries. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming (PPDP '16)*, pages 228–241. ACM, September 2016.
DOI: 10.1145/2967973.2968610.

[60] Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. Strengthening invariants for efficient computation. *Science of Computer Programming*, 41(2):139–172, 2001.
DOI: 10.1016/S0167-6423(01)00003-X.

[61] Yanhong A. Liu and Tim Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–39, 1995.
DOI: 10.1016/0167-6423(94)00031-9.

[62] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. Stateful bulk processing for incremental analytics. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC '10)*, pages 51–62. ACM, June 2010.
DOI: 10.1145/1807128.1807138.

[63] Calvin Loncaric. Asymmetric subscription status between OpenFire users, February 2017. accessed February 28, 2017.
`https://discourse.igniterealtime.org/t/asymmetric-subscription-status-between-openfire-users/62041`.

[64] Calvin Loncaric, Michael D. Ernst, and Emina Torlak. Generalized data structure synthesis. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*, pages 958–968. ACM, May 2018.
DOI: 10.1145/3180155.3180211.

[65] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. Fast synthesis of fast collections. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*, pages 355–368. ACM, June 2016.
DOI: 10.1145/2908080.2908122.

[66] Ingo Maier and Martin Odersky. Higher-order reactive programming with incremental lists. In *ECOOP 2013 – Object-Oriented Programming*, pages 707–731. Springer, July 2013.
DOI: 10.1007/978-3-642-39038-8_29.

[67] Darko Marinov and Sarfraz 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 22–31. IEEE, 2001.
DOI: 10.1109/ASE.2001.989787.

[68] John McCarthy. A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*, 26:33–70, 1959.
DOI: 10.1016/S0049-237X(09)70099-0.

[69] Donald Michie. "Memo" functions and machine learning. *Nature*, 218, April 1968.

[70] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 439–455. ACM, November 2013.
DOI: 10.1145/2517349.2522738.

[71] Christopher Olston, Greg Chiou, Laukik Chitnis, Francis Liu, Yiping Han, Mattias Larsson, Andeas Neumann, Vellanki B.N. Rao, Vijayanand Sankarasubramanian, Siddharth Seth, Chao Tian, Topher ZiCornell, and Xiaodan Wang. Nova: Continuous Pig/Hadoop workflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*, pages 1081–1090. ACM, June 2011.
DOI: 10.1145/1989323.1989439.

[72] [OF-1121] fails to remove roster items when a user is deleted on the server, April 2016. accessed March 28, 2017.
https://issues.igniterealtime.org/browse/OF-1121.

[73] Mark H. Overmars. *The Design of Dynamic Data Structures*. Springer, 1983.
DOI: 10.1007/BFb0014927.

[74] Robert Paige. Programming with invariants. *IEEE Software*, 3(1):56–69, January 1986.
DOI: 10.1109/MS.1986.233070.

[75] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.
DOI: 10.1145/357172.357177.

[76] William Pugh. *Incremental Computation via Function Caching*. PhD thesis, Cornell University, 1988.

[77] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*, pages 315–328. ACM, January 1989.
DOI: 10.1145/75277.75305.

[78] G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93)*, pages 502–510. ACM, 1993.
DOI: 10.1145/158511.158710.

[79] Sat4j boolean reasoning library, 2016. accessed February 3, 2016.
https://www.sat4j.org.

[80] SAT competition 2002, 2002. accessed February 3, 2016.
http://www.satcompetition.org/2002/.

[81] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. Automatic data structure selection in SETL. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '79)*, pages 197–210. ACM, January 1979.
DOI: 10.1145/567752.567771.

[82] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. An automatic technique for selection of data representations in SETL programs. *ACM Transactions on Programming Languages and Systems*, 3(2):126–143, April 1981.
DOI: 10.1145/357133.357135.

[83] Jacob 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. ACM, 1975.
DOI: 10.1145/512976.512981.

[84] Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, pages 408–418. ACM, 2009.
DOI: 10.1145/1542476.1542522.

[85] Rohin Shah and Rastislav Bodík. Automated incrementalization through synthesis. In *Proceedings of the 1st Workshop on Incremental Computing (IC '17)*, June 2017.

[86] Oded Shmueli and Alon Itai. Maintenance of views. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*, pages 240–255. ACM, June 1984.
DOI: 10.1145/602259.602293.

[87] Laure Simon, Daniel Le Berre, and Edward A. Hirsch. The SAT2002 competition. *Annals of Mathematics and Artificial Intelligence*, 43(1):307–342, January 2005.
DOI: 10.1007/s10472-005-0424-6.

[88] Rishabh Singh and Armando 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. ACM, 2011.
DOI: 10.1145/2025113.2025153.

[89] Marty Sirkin, Don Batory, and Vivek Singhal. Software components in a data structure precompiler. In *Proceedings of the 15th International Conference on Software Engineering (ICSE '93)*, pages 437–446. IEEE, 1993.
DOI: 10.1109/ICSE.1993.346022.

[90] Yannis Smaragdakis and Don Batory. DiSTiL: A transformation library for data structures. In *USENIX Conference on Domain-Specific Languages*, pages 257–270, 1997.

[91] Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.
DOI: 10.1109/32.58788.

[92] Douglas R. Smith. Designware: Software development by refinement. *Electronic Notes in Theoretical Computer Science*, 29:275–287, December 1999.
DOI: 10.1016/S1571-0661(05)80320-2.

[93] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California at Berkeley, 2008.

[94] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching concurrent data structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*, pages 136–148. ACM, 2008.
DOI: 10.1145/1375581.1375599.

[95] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *Operating Systems Review*, 40(5):404–415, October 2006.
DOI: 10.1145/1168917.1168907.

[96] The Apache Software Foundation. Apache Lucene, 2016. accessed April 27, 2018.
`https://lucene.apache.org`.

[97] The Free Software Foundation. GNU Time, 2018. accessed June 21, 2018.
`https://www.gnu.org/software/time/`.

[98] John Toman and Dan Grossman. Staccato: A bug finder for dynamic configuration updates. In *30th European Conference on Object-Oriented Programming*, pages 24:1–24:25. Schloss Dagstuhl, July 2016.
DOI: 10.4230/LIPIcs.ECOOP.2016.24.

[99] Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pages 530–541. ACM, June 2014.
DOI: 10.1145/2594291.2594340.

[100] Redmine Issue Tracker. N + 1 queries when rendering the Gantt, August 2017. accessed August 21, 2018.
`https://www.redmine.org/issues/26691`.

[101] Redmine Issue Tracker. Overview - Redmine, 2018. accessed August 21, 2018.
`https://www.redmine.org/`.

[102] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev 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. ACM, 2013.
DOI: 10.1145/2491956.2462174.

[103] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. How not to structure your database-backed web applications: A study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*, pages 800–810. ACM, May 2018.
DOI: 10.1145/3180155.3180194.

[104] Frank Kenneth Zadeck. *Incremental Data Flow Analysis in a Structured Program Editor*. PhD thesis, Rice University, 1984.

[105] ZTopo topographic map viewer, 2015. accessed May 8, 2015.
`https://hawkinsp.github.io/ZTopo/`.

Appendix A

POLICIES AND HEURISTICS

Cozy implements an exhaustive search procedure capable of finding any data structure in its output space. Nevertheless, there are a number of policies and heuristics that we found improve performance substantially. Some of these skip certain expressions, giving up on Cozy's completeness promise but drastically improving performance and preventing bad behavior from our imperfect cost model. Others simply affect the order in which Cozy explores the space.

Some heuristics are important enough to document in the description of the algorithms they affect, and are thus described in the main text of this document. These are diversity injection (Section 3.3) and our choice of rewrite rules (Section 5.4.2). In some sense the cost model (Section 3.5) and our state maintenance sketches (Section 2.3) are heuristics since Cozy would still function with different definitions.

The remainder of these rules are not central to our approach, but are documented here for future research.

Expression Stealing When Cozy enumerates expressions in order of size (Chapter 3), it does not use the size of the syntax tree as the size metric. Before enumeration starts, Cozy "steals" all subexpressions of the specification, including the user-specified data structure invariants, synthesized concretization functions, user-specified method preconditions, and method bodies. These stolen expressions are treated as having size zero. The enumeration procedure in Figure 3.3 already supports this: the $vars$ parameter can be given any number of expressions (even non-variable expressions) to enumerate at $size = 0$. Expression stealing is useful because in practice, many subexpressions in the specification will also appear in any efficient implementation.

Aggressive Restarts Enumerative CEGIS algorithms need to restart whenever they discover a new counterexample [102], since a critical expression might have been skipped during equivalence class deduplication. In addition, Cozy restarts its synthesis algorithm every time it finds a legal improvement to the specification. While these restarts sound harmful, they interact well with expression stealing: if Cozy has done a great deal of work to brute-force an improvement, that improvement should be treated as having size zero for the next iteration.

Blind Replacements In addition to replacing subexpressions of the specification with improved versions when they are discovered (Section 3.6), Cozy also attempts "blind" replacements. In a blind replacement, the the subexpression in the specification and its candidate replacement have different behavior. Making blind replacements often uncovers surprising improvements to the specification in cases where some of the return value of the subexpression would be discarded. For instance, if Cozy is meant to improve the expression

$$\text{Distinct}\,\underset{>0}{\text{Filter}}\,\underbrace{ints}_{\text{state}}$$

it might attempt to replace $ints$ with $(\text{Distinct}\,ints)$. While those expressions have different behaviors, the replacement happens to be legal in this example, and since $|\text{Distinct}\,ints| < |ints|$, the cost model will favor the resulting expression

$$\text{Distinct}\,\underset{>0}{\text{Filter}}\,\underbrace{\text{Distinct}\,ints}_{\text{state}}\ .$$

After an aggressive restart and expression stealing, the expression

$$\underset{>0}{\text{Filter}}\,\underbrace{\text{Distinct}\,ints}_{\text{state}}$$

will have size zero and Cozy will immediately discover it as an improvement to the original expression.

Early Termination Many of the query operations that Cozy adds to the data structure (Section 2.3) have very simple implementations. As described, Cozy will simplify the bodies of these query operations to simple expressions such as "return true" or "return the singleton set x" and then continue searching forever for improvements to those simple expressions. This results in an entire thread of execution occupied on a fruitless task. To help avoid this, we stop synthesis if the current best expression is a variable, a literal, a singleton set of a variable or literal, a field read on a variable or literal, or any other constant-time unary operation on a variable or literal.

Default Values Cozy has a default value for every type (Section 2.2). Some of these are obvious; the default value for a collection should be an empty collection and the default value for a tuple should be a tuple containing the default value for each of its constituent elements. However, Cozy needs to make a choice about the default value for integers (zero or one) and for booleans (true or false). Perhaps surprisingly, this choice matters. Cozy performs better because we use zero as the default value for integers and false as the default value for booleans. The use cases that motivate these choices are determining whether a collection L contains an element x, which can be accomplished with

$$\underbrace{\left(\underset{\lambda y.\text{True}}{\text{MakeMap}\ L}\right)}_{\text{state}}[x]$$

and counting the number of occurrences of x in L, which can be accomplished with

$$\underbrace{\left(\underset{\lambda y.|\text{Filter}_{==y}\ L|}{\text{MakeMap}\ \ L}\right)}_{\text{state}}[x]\ .$$

Both of these expressions would be more complicated if our choices for the default values for integers and booleans were different, since the default values are what gets returned when $x \notin L$.

Culling Useless Expressions Since Cozy uses an enumerative algorithm in an exponential search space, it also has heuristics that skip some expressions entirely to help keep the search manageable. Since the cost model is not perfect, these heuristics also help to avoid some degenerate behavior. It may be possible to remove some of these heuristics with an improved cost model.

- No expressions may have a type that is a collection of collections, unless that type appears in the specification.

- Constant-time binary operators may not be state expressions. This prevents Cozy from storing *e.g.* the value $x + 1$ on the data structure.

- Literals may not be state expressions. They may appear in state expressions, but no state expression may be just a literal. This prevents Cozy from storing, say, the empty set as a member on the data structure.

- Nonzero integer constants may not appear in any subexpression of a state expression. This helps keep the set of state expressions small.

- $\mathrm{FlatMap}$ may not appear in any subexpression of a state expression. This prevents Cozy from storing $O(n^2)$ size collections on the data structure.

- Collection subtraction may not appear in any subexpression of a state expression. This prevents Cozy from answering element removal operations with a map whose keys are elements and whose values are the entire collection minus its corresponding key. Cozy's cost model is very fond of this particular construction, but it performs very poorly in practice.

- Map keys may not be collection types. This is because Cozy collections are mutable, and mutable map keys can be very dangerous.

- The set of keys passed to $\mathrm{MakeMap}$ must not be a singleton or empty collection. Small or empty maps are often better answered with conditional operations than by the overhead of an entire hash map in memory.

- An $\arg\min$ or $\arg\max$ expression may not be a subexpression of any state expression if some update method on the data structure could remove more than one element from the underlying collection. This works around a limitation that Cozy has no efficient way to compute, say, the third or fourth minimum element in a collection.