Fast Synthesis of Fast Collections

Calvin Loncaric
Emina Torlak
Michael D. Ernst

University of Washington
Data structures are everywhere
Data structures are everywhere
Data structures are everywhere
Data structures are everywhere
Data structures are everywhere

Lists, maps, and sets solve many problems
Data structures are everywhere

Lists, maps, and sets solve many problems

What if I need a custom data structure?
Cozy synthesizes collections

Outline


Cozy synthesizes collections

- Correct by construction
Cozy synthesizes collections

- Correct by construction
- Specifications orders-of-magnitude shorter than implementations, synthesized in < 90 seconds
Cozy synthesizes collections

- Correct by construction
- Specifications orders-of-magnitude shorter than implementations, synthesized in < 90 seconds
- Equivalent performance to human-written code
Myria Analytics Storage

Request 1

Request 2

time
Myria Analytics Storage

Request 1

Request 2

time
Myria Analytics Storage

Request 1

Request 2

time
Myria Analytics Storage

Request 1

Request 2

Goal: efficient retrieval of entries for a particular request ID in a particular timespan
Myria Analytics Storage

class AnalyticsLog {

    void log(Entry e)

    Iterator<Entry> getEntries(
        int queryId,
        int subqueryId,
        int fragmentId,
        long start,
        long end)

}
Myria Analytics Storage

Insert an entry into the data structure

class AnalyticsLog {
    void log(Entry e)

    Iterator<Entry> getEntries(
        int queryId,
        int subqueryId,
        int fragmentId,
        long start,
        long end)
}

5
class AnalyticsLog {
    void log(Entry e)
    Iterator<Entry> getEntries(
        int queryId,
        int subqueryId,
        int fragmentId,
        long start,
        long end)
}
Myria Analytics Storage

**Specification:**

**Entry** has:
- queryId : Int,
- subqueryId : Int,
- fragmentId : Int,
- start, end : Long,

... 

**getEntries:** all e where
- e.queryId = queryId and
- e.subqueryId = subqueryId and
- e.fragmentId = fragmentId and
- e.end >= start and
- e.start <= end

```java
class AnalyticsLog {
    void log(Entry e)
    
    Iterator<Entry> getEntries(
        int queryId, 
        int subqueryId, 
        int fragmentId, 
        long start, 
        long end
    )
```

```java
}
```
Cozy synthesizes collections

**Specification:**

Entry has:
- `field1 : Type1`,
- `field2 : Type2`,
- ...

retrieveA: all e where condition

retrieveB: all e where condition

```java
class Structure {
    void add(Entry e)
    void remove(Entry e)
    void update(Entry e, ...)

    Iterator<Entry> retrieveA(...)  
    Iterator<Entry> retrieveB(...)  
}
```
Trivial Solution

\[ \text{retrive: all } e \text{ where } P(e, \text{ input}) \]
Trivial Solution

`retrieve`: all $e$ where $P(e, \text{input})$

```java
List<Entry> data;

Iterator<Entry> retrieve(input) {
    for e in data:
        if P(e, input):
            yield e
}
```
Trivial Solution

```
List<Entry> data;

Iterator<Entry> retrieve(input) {
  for e in data:
    if P(e, input):
      yield e
}
```

There has to be a better way!
In the quest for a good solution, the search space of “all possible programs” is simply too large.
In the quest for a good solution, the search space of “all possible programs” is simply too large.
Entry has:
- field1, field2, ...
- retrieveA: all e where condition
- retrieveB: all e where condition

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

Iterator retrieveA(...)  
Iterator retrieveB(...)
Entry has:
- field1, field2, ...
- retrieveA: all e where condition
- retrieveB: all e where condition

**Specification** → **Outline**

Specific enough to describe asymptotic performance

General enough to encode a data structure succinctly

**Outline** → **Implementation**

**Implementation**

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

Iterator retrieveA(...)
Iterator retrieveB(...)

**Intractable**

void synthesis algorithm
Outlines

Plans for retrieving entries
Outlines

Plans for retrieving entries

• **All** ( )
Outlines

Plans for retrieving entries

• All ()

• HashLookup ( outline, field = var )
Outlines

Plans for retrieving entries

• **All** ( )

• **HashLookup** ( **outline**, **field = var** )

• **BinarySearch** ( **outline**, **field > var** )
Outlines

Plans for retrieving entries

• **All** ( )

• **HashLookup** ( outline, field = var )

• **BinarySearch** ( outline, field > var )

• **Concat** ( outline$_{1}$, outline$_{2}$ )
Outlines

Plans for retrieving entries

• **All**( )

• **HashLookup**(  outline, field = var  )

• **BinarySearch**(  outline, field > var  )

• **Concat**(  outline₁, outline₂  )

• **Filter**(  outline, predicate  )
Outlines → Implementations

Outlines → Implementations

- Specification
- Inductive Synthesizer
- Verifier
- Outline
- Rep.
- Impl.
- Rep.
- Impl.
- Rep.
- Impl.
Outlines → Implementations

HashLookup (All(), e.queryId = q )

```
class Structure {
    Iterator<Entry> retrieve(q) {
        ...
    }
}
```
class Structure {
    Iterator<Entry> retrieve(q) {
        ...
    }
}

HashLookup (All(),
    e.queryId = q )
class Structure {

T data;

Iterator<Entry> retrieve(q) {
    ...
}

}
HashLookup (data, e.queryId = q)

```java
class Structure {
    T data;
    Iterator<Entry> retrieve(q) {
        ...
    }
}
```
HashLookup (data, e.queryId = q)

```java
class Structure {
    T data;
    Iterator<Entry> retrieve(q) {
        ...
    }
}
```
Outlines → Implementations

```
class Structure {
  T data;
  Iterator<Entry> retrieve(q) {
    ...
  }
}
```
class Structure {
    HMap<K,V> data;

    Iterator<Entry> retrieve(q) {
        ...
    }
}
Outlines → Implementations

```java
class Structure {
    HMap<K, V> data;

    Iterator<Entry> retrieve(q) {
        ...}
}
```

HashLookup (data, e.queryId = q)
Outlines → Implementations

```
class Structure {
    HMap<K, V> data;

    Iterator<Entry> retrieve(q) { ... }
}
```

```
HashLookup (data, e.queryId = q)
```
class Structure {
    HMap<int, V> data;
    Iterator<Entry> retrieve(q) {
        ...
    }
}
HashLookup(
data,
e.queryId = q)

class Structure {
    HMap<int, V> data;
    Iterator<Entry> retrieve(q) {
        ...
    }
}

Outlines → Implementations
HashLookup (data, e.queryId = q)

class Structure {
    HMap<int, V> data;
    Iterator<Entry> retrieve(q) {
        ...
    }
}

HashLookup (data, e.queryId = q )

class Structure {
  HMap<int, V> data;
  Iterator<Entry> retrieve(q) {
    ...
  }
}

V = ArrayList<Entry>
HashLookup (data, e.queryId = q)

```java
class Structure {
    HMap<int, V> data;
    Iterator<Entry> retrieve(q) {
        ... }
}

V = ArrayList<Entry>
V = LinkedList<Entry>
```
Outlines → Implementations

```java
class Structure {
    HMap<int, V> data;
    Iterator<Entry> retrieve(q) {
        ...
    }
}

HashLookup (data, e.queryId = q)
```
class Structure {
    HMap<int,V> data;
    Iterator<Entry> retrieve(q)
    {
        v = data.get(q);
        return v.iterator();
    }
}
HashLookup (data, e.queryId = q)

class Structure
{
    HMap<int,V> data;

    Iterator<Entry> retrieve(q)
    {
        v = data.get(q);
        return v.iterator();
    }
}

Outlines → Implementations

add, remove, update
Specication $\rightarrow$ Outline
Specification → Outline

Specification → Outline

CEGIS

candidate

Inductive Synthesizer

Verifier

counterexample
- or -
certification of correctness

retrieve: all e where e.queryId = q and …
CEGIS

Inductive Synthesizer

Verifier

candidate

counterexample

retrieve: all e where e.queryId = q and ...

Remembers all examples; only reasons about examples collected thus far.
Specification → Outline

Remembers all examples; only reasons about examples collected thus far.

CEGIS

Inductive Synthesizer

Verifier

counterexample - or - certification of correctness

retrieve: all e where e.queryId = q and …

∀I ∀S, out = { e | e ∈ S ∧ P(I, e) }
Cost Model

Filter ( All(),
         e.queryId = q )

HashLookup ( All(),
             e.queryId = q )
Cost Model

\[ O(1) \]

Filter (All(), e.queryId = q )

HashLookup (All(), e.queryId = q )
Cost Model

Filter (All(), e.queryId = q)

HashLookup (All(), e.queryId = q)

$O(n)$

$O(1)$
Cost Model

\[ O(n) \]

Filter ( 
\[ \text{All()}, \quad e.\text{queryId} = q \] )

\[ O(1) \]

HashLookup ( 
\[ \text{All()}, \quad e.\text{queryId} = q \] )

\[ O(1) \]
Cost Model

Filter ( All(), e.queryId = q )

HashLookup ( All(), e.queryId = q )

\( O(n) \)

\( O(1) \)

\( O(1) \)

\( O(1) \)
Cost Model

$O(n)$

$O(1)$

Filter (All(), e.queryId = q )

$O(1)$

HashLookup (All(), e.queryId = q )

$O(1)$

$O(1)$
Cost Model

$O(n)$

$O(1)$

Filter ($\text{All}()$, $e.\text{queryId} = q$)

$O(1)$

HashLookup ($\text{All}()$, $e.\text{queryId} = q$)

Cozy prefers outlines with lower cost
Inductive Synthesis
Enumerative search
Inductive Synthesis

Enumerative search

size 1

All
Inductive Synthesis

Enumerative search

size 1

size 2

All

HashLookup(All, x=y)

Filter(All, x=y)

BinarySearch(All, x>y)

...
Inductive Synthesis

Enumerative search

size 1

size 2

All

HashLookup(All, x=y)

Filter(All, x=y)

BinarySearch(All, x>y)

...
Inductive Synthesis

Enumerative Search

$\text{size 1}$

$\text{size 2}$

All

$\text{HashLookup(All, x=y)}$

$\text{BinarySearch(All, x>y)}$

$\ldots$

$\text{Concat(HashLookup(...), \ldots)}$

vs

$\text{Concat(Filter(...), \ldots)}$
Inductive Synthesis
Enumerative search

size 1

All

size 2

HashLookup(All, x=y)
BinarySearch(All, x>y)

size 3

HashLookup(HashLookup(...), a=b)
Filter(HashLookup(...), p=q)
Filter(BinarySearch(...), x<y)
Inductive Synthesis
Enumerative search

size 1

All

size 2

BinarySearch(All, x>y)

Filter(All, x=y)

size 3

Filter(BinarySearch(…), x<y)

Filter(HashLookup(…), p=q)

HashLookup(HashLookup(…), a=b)

correct on all current examples
Outline Verification

**Specification:**

Entry has:
- `queryId : Int`,
- `subqueryId : Int`,
  ...

\[ P \]

\text{retrieve: all } e \text{ where } e.queryId = q \text{ and } ...
Outline Verification

Specification:

\[ \{ e \mid e \in S \land P(I, e) \} \]

subqueryId : Int,
...

\( P \) retrieve: all e where e.queryId = q and ...
Outline Verification

**Specification:**

\[
\{ e \mid e \in S \land P(I, e) \}
\]

subqueryId : Int,
...

**retrieve:** all e where e.queryId = q and ...

\[
\text{HashLookup}(\text{All}(),
\ e.\text{queryId} = q)
\]
Outline Verification

**Specification:**

\[ \{ e \mid e \in S \land P(I, e) \} \]

subqueryId : Int,
...

**retrieve:** all e where
  e.queryId = q and ...

HashLookup(
  All(),
  e.queryId = q)

representative predicate \(Q\)

\[ e.queryId = q \]
Outline Verification

Specification:

\{ e \mid e \in S \land P(I, e) \}

retrieve: all e where
  e.queryId = q and ...

Hashl selection:

\{ e \mid e \in S \land Q(I, e) \}

e.queryId = q

representative predicate \( Q \)
Outline Verification

$$\{ e \mid e \in S \land P(I,e) \} \ \overset{?}{=} \ \{ e \mid e \in S \land Q(I,e) \}$$
Outline Verification

\[ \{ e \mid e \in S \land P(I, e) \} \equiv \{ e \mid e \in S \land Q(I, e) \} \]

yes if and only if for all \( I, e \):
\[ P(I, e) = Q(I, e) \]
Outline Verification

\[ \{ e \mid e \in S \land P(I, e) \} \equiv \{ e \mid e \in S \land Q(I, e) \} \]

yes if and only if for all \( I, e \):
\[ P(I, e) = Q(I, e) \]

equivalence can be checked with an SMT solver
Evaluation
Evaluation

• Improve correctness
Evaluation

• Improve correctness
• Save programmer effort
Evaluation

- Improve correctness
- Save programmer effort
- Match performance
Evaluation

• Improve correctness ✓
• Save programmer effort ✓
• Match performance ✓
Case studies
Case studies

• **Myria**: analytics

Analytics data indexed by timespan and by request ID
Case studies

- **Myria**: analytics
  
  Analytics data indexed by timespan and by request ID

- **ZTopo**: tile cache
  
  Tracks map tiles in a least-recently-used cache
Case studies

- **Myria**: analytics
  - Analytics data indexed by timespan and by request ID

- **ZTopo**: tile cache
  - Tracks map tiles in a least-recently-used cache

- **Bullet**: volume tree
  - Stores axis-aligned bounding boxes for fast collision detection
Case studies

- **Myria**: analytics
  Analytics data indexed by timespan and by request ID

- **ZTopo**: tile cache
  Tracks map tiles in a least-recently-used cache

- **Bullet**: volume tree
  Stores axis-aligned bounding boxes for fast collision detection

- **Sat4j**: variable metadata
  Tracks information about each variable in the formula
Case studies

- **Myria**: analytics
  Analytics data indexed by timespan and by request ID

- **Bullet**: volume tree
  Stores axis-aligned bounding boxes for fast collision detection

- **ZTopo**: tile cache
  Tracks map tiles in a least-recently-used cache

- **Sat4j**: variable metadata
  Tracks information about each variable in the formula
Specifications vs. Implementations

<table>
<thead>
<tr>
<th></th>
<th>Myria</th>
<th>ZTopo</th>
<th>Sat4j</th>
<th>Bullet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code</td>
<td>22</td>
<td>25</td>
<td>11</td>
<td>23</td>
</tr>
<tr>
<td>Original</td>
<td>269</td>
<td>1,383</td>
<td>292</td>
<td>2,582</td>
</tr>
<tr>
<td>Spec</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Synthesis Time

- **Outline Synthesis**
- **Auto-Tuning**

<table>
<thead>
<tr>
<th>Software</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Myria</td>
<td>90</td>
</tr>
<tr>
<td>ZTopo</td>
<td>30</td>
</tr>
<tr>
<td>Sat4j</td>
<td>10</td>
</tr>
<tr>
<td>Bullet</td>
<td>90</td>
</tr>
</tbody>
</table>
Performance

[Red square] Original  [Blue triangle] Synthesized
Performance

Data structures are nearly identical

Original
Synthesized

ZTopo
Performance

Data structures are nearly identical

Binary search tree vs. space partitioning tree

ZTopo  Bullet
Performance

- Original
- Synthesized

Data structures are nearly identical

Binary search tree vs. space partitioning tree

ZTopo

Bullet
Performance

- Original
- Synthesized

Data structures are nearly identical

Binary search tree vs. space partitioning tree

Small overhead; performance dominated by other factors

ZTopo

Bullet

Sat4j
Performance

- **Original**
- **Synthesized**

Data structures are nearly identical

Binary search tree vs. space partitioning tree

Small overhead; performance dominated by other factors

Original implementation has worst-case linear time

---

**Charts**

- **ZTopo**
- **Bullet**
- **Sat4j**
- **Myria**
Performance

- **Original**
- **Synthesized**

Data structures are nearly identical

Binary search tree vs. space partitioning tree

Small overhead; performance dominated by other factors

Original implementation has worst-case linear time

ZTopo

Bullet

Sat4j

Myria
Related Work
Related Work

• J. Earley: “High level iterators and a method for automatically designing data structure representation” (1974)

• Hard-coded rewrite rules
Related Work

• J. Earley: “High level iterators and a method for automatically designing data structure representation” (1974)
  • Hard-coded rewrite rules

  • Enumerate possible views & indexes based on query syntax and use the planner to decide which ones to keep
Related Work

  - Hard-coded rewrite rules

  - Enumerate possible views & indexes based on query syntax and use the planner to decide which ones to keep

  - Enumerate representations and use a planner to implement retrieval operations; conjunctions of equalities only
Implementation outlines make the problem tractable

Synthesis completes < 90 seconds

Cozy generates correct code, and matches handwritten implementation performance