

Fast Synthesis of Fast Collections

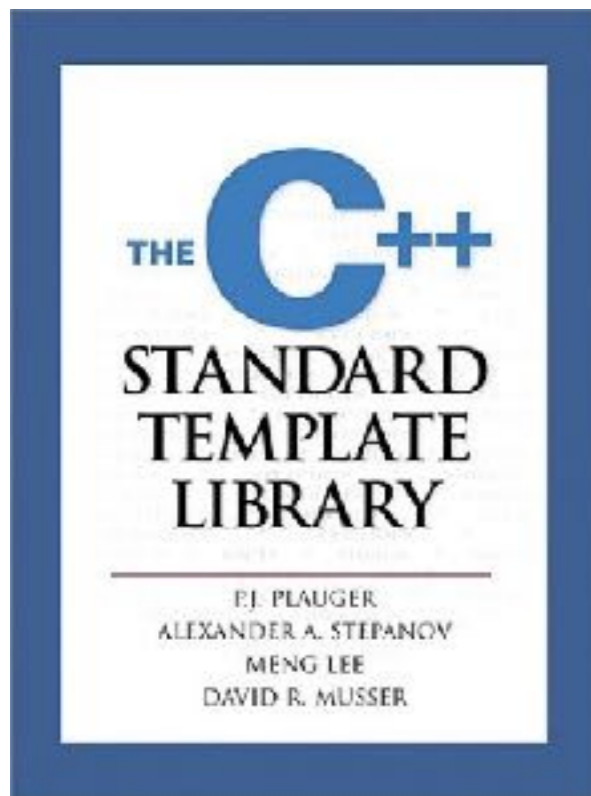
Calvin Loncaric

Emina Torlak

Michael D. Ernst

University of Washington

Data structures are everywhere



ORACLE Java SE Documentation

Oracle Technology Network Software Downloads Doc

The Collections Framework

The collections framework is a unified architecture for representing and manipulating objects. It enables interoperability among unrelated classes by defining a set of interfaces and abstract classes that implement these interfaces and abstract classes.

8.3. `collections` — Container datatypes

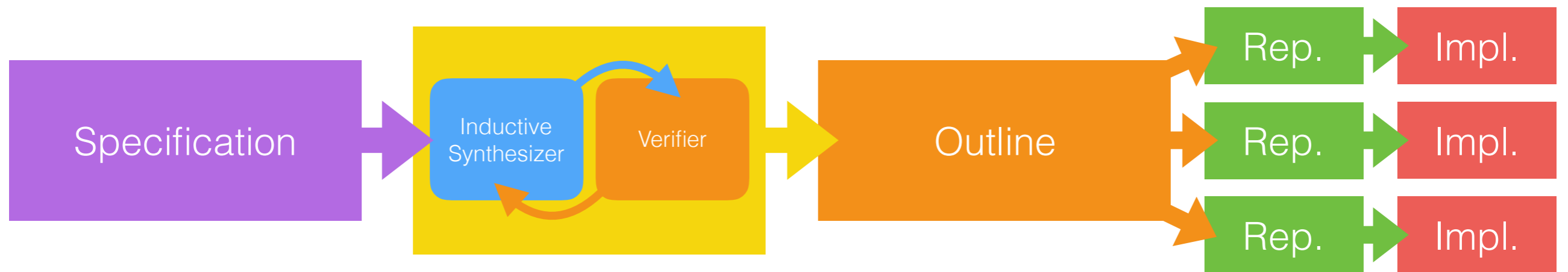
Source code: [Lib/collections/__init__.py](#)

This module implements specialized container datatypes provided by Python's general purpose built-in containers, `dict`, `list`, `set`

Lists, maps, and sets solve many problems

What if I need a custom data structure?

Cozy synthesizes collections

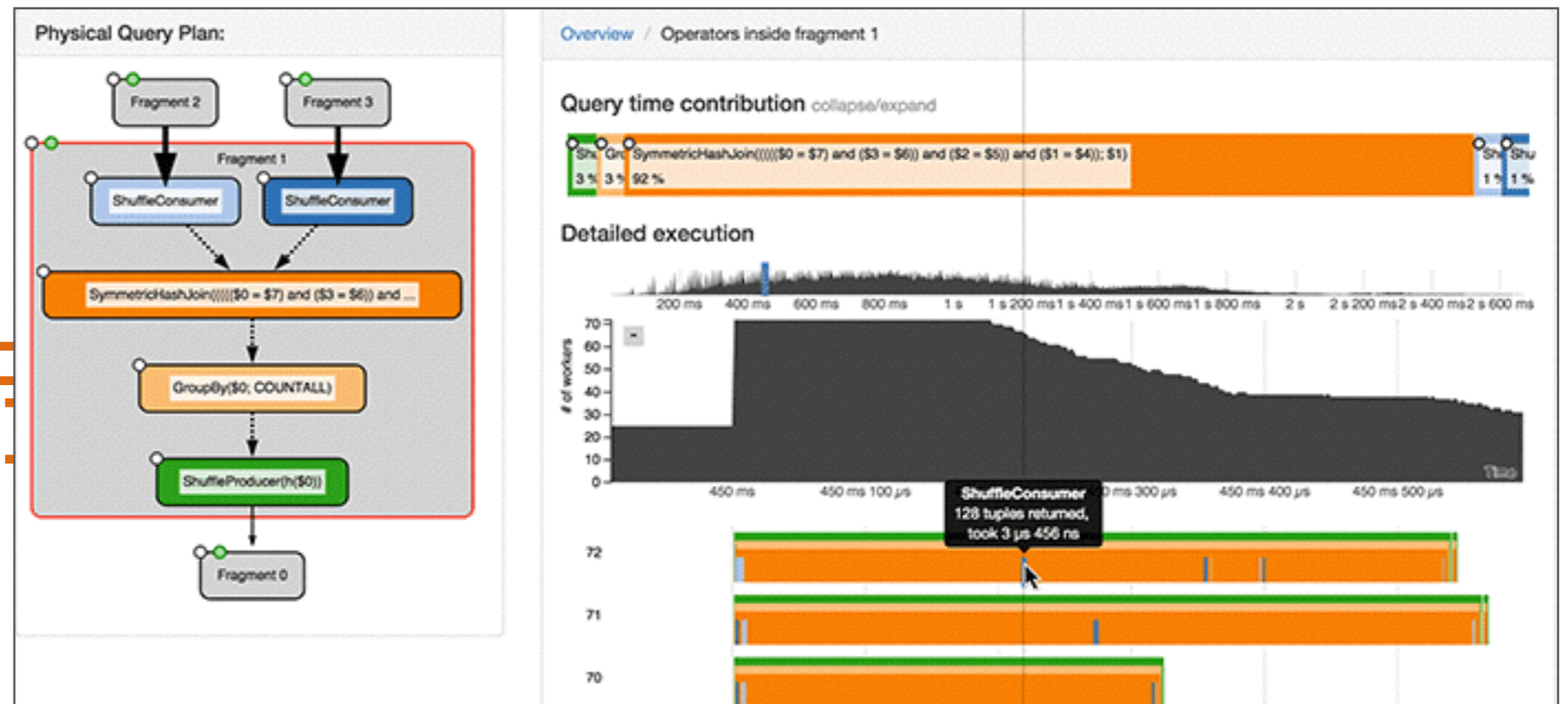


- Correct by construction
- Specifications orders-of-magnitude shorter than implementations, synthesized in < 90 seconds
- Equivalent performance to human-written code

Myria Analytics Storage

Request 1

Request 2



Request 2 in a particular timespan

time

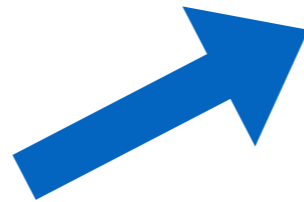
Myria Analytics Storage

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)  
}
```

Retrieve entries



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

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

Cozy synthesizes collections

Specification:

Entry has:

field1 : **Type1**,
field2 : **Type2**,
...

retrieveA: all e where
condition

retrieveB: all e where
condition



```
class Structure {
```

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

```
Iterator<Entry> retrieveA(...)  
Iterator<Entry> retrieveB(...)
```

```
}
```

Trivial Solution

retrieve: all e where
P(e, input)

```
List<Entry> data;  
Iterator<Entry> re  
    for e in data:  
        if P(e, input):  
            yield e  
}
```

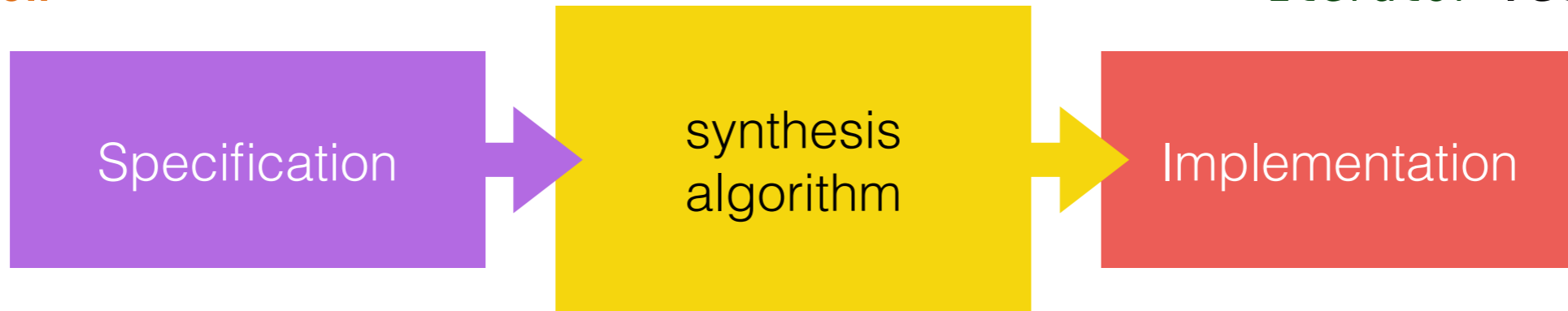
There has to be a
better way!

Entry has:
field1, **field2**, ...
retrieveA: all e where
condition
retrieveB: all e where
condition

Intractable

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

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

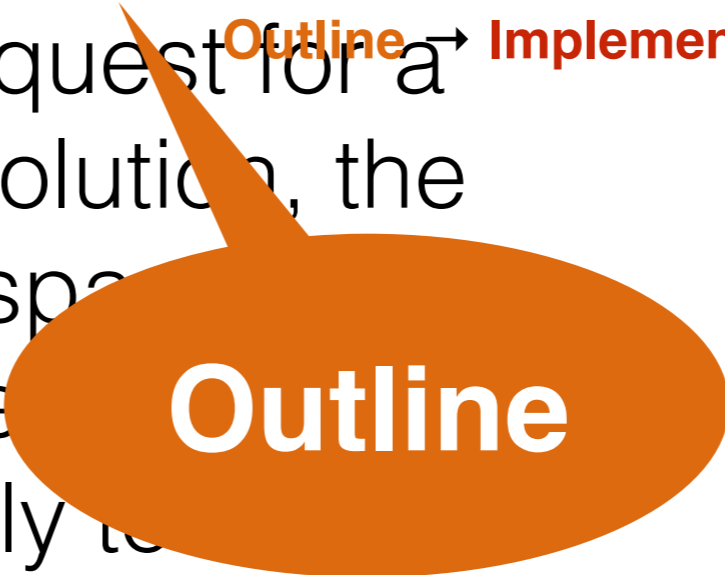


Tractable

Tractable

Specification → Outline → Implementation

In the quest for a good solution, the search space is simply too large to explore. The search space is **specific** enough to describe asymptotic performance, but **general** enough to encode a data structure successfully.

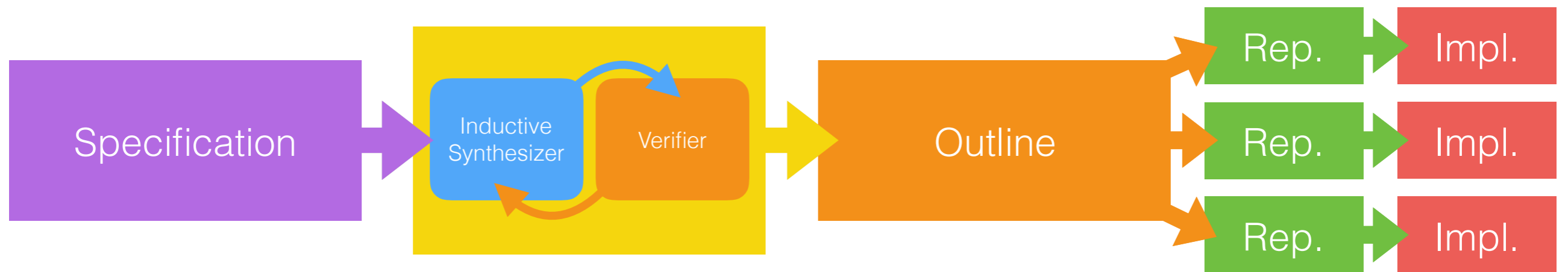


Outlines

Plans for retrieving entries

- **All** ()
- **HashLookup** (`outline`, field = var)
- **BinarySearch** (`outline`, field > var)
- **Concat** (`outline1`, `outline2`)
- **Filter** (`outline`, predicate)

Outlines → Implementations



Outlines → Implementations

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

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

Outlines → Implementations

```
HashLookup (
  data,
  e.queryId = q )
```

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

Outlines → Implementations

```
HashLookup (  
  data,  
  e.queryId = q )
```



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

Outlines → Implementations

```
class Structure {
```

```
HashLookup (  
  data,  
  e.queryId = q )
```

```
  HMap<int, V> data;
```

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

```
V = ArrayList<Entry>
```

```
V = LinkedList<Entry>
```

Outlines → Implementations

HashLookup (
 data,
 e.queryId = q)

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


Outlines → Implementations

**add,
remove,
update**

`class Structure`

`HMap<int, V> data;`

`Iterator<Entry>
retrieve(q)`

`{`

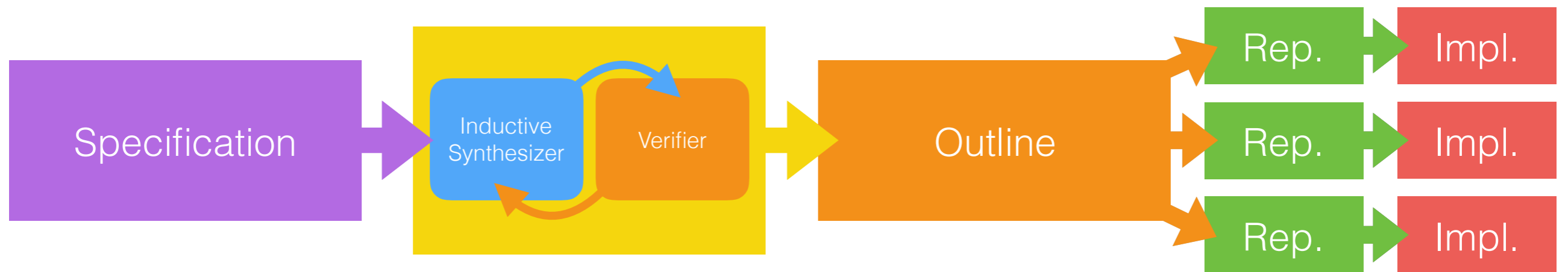
`v = data.get(q);
return v.iterator();`

`}`

HashLookup (
`data,`
`e.queryId = q`)



Specification → Outline

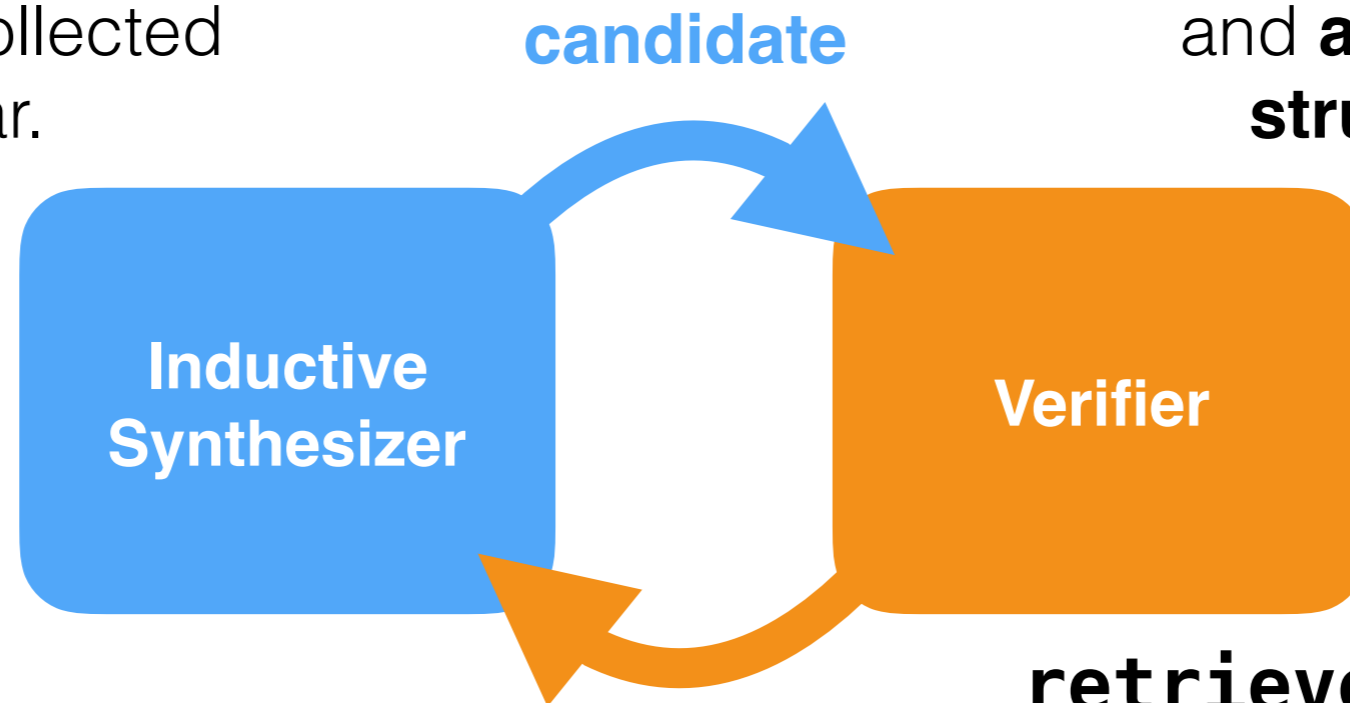


Specification → Outline

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

CEGIS

Must ensure the outline is correct for **all possible inputs** and **all possible data structure states**.



counterexample
- or -
certification of correctness

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

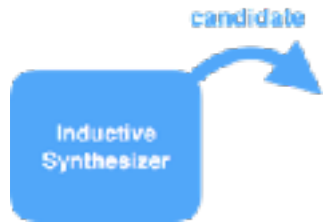
$$\forall I \forall S, out = \{ e \mid e \in S \wedge P(I, e) \}$$

Cost Model



Cozy prefers outlines with lower cost

Inductive

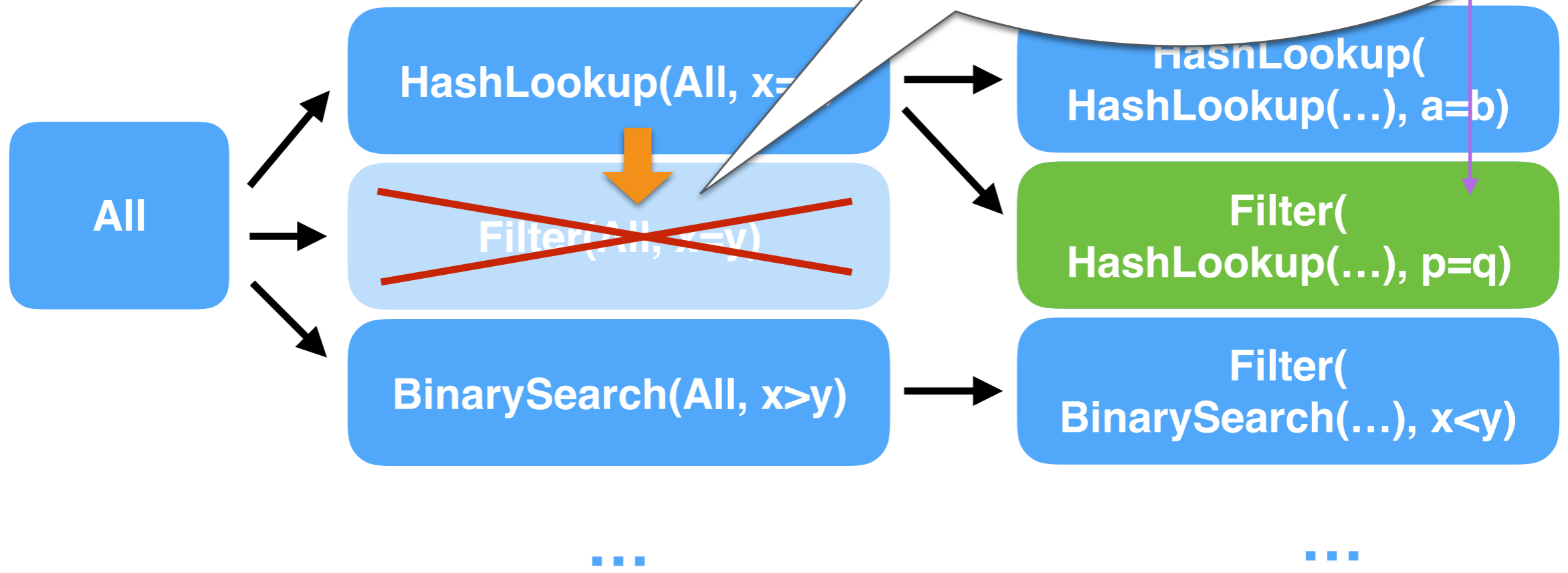


Enum

size 1

size 2

Concat (**HashLookup**(...), ...) VS Concat (**Filter**(...), ...)





Outline Verification

Specification

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

subqueryId : Int,
...

P

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

Hashed set

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

e.queryId = q



**representative
predicate *Q***

e.queryId = q



Outline Verification

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

?
=

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

yes **if and only if** for all I, e :
 $P(I, e) = Q(I, e)$

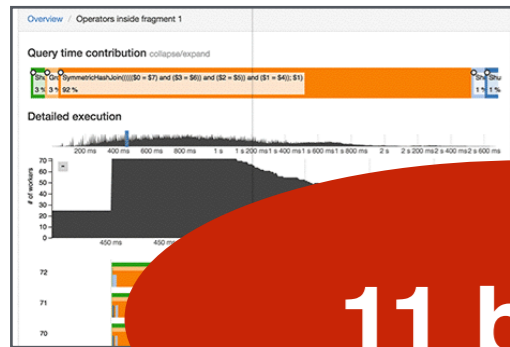
**equivalence can be checked
with an SMT solver**

Evaluation

- Improve correctness ✓
- Save programmer effort ✓
- Match performance ✓

Case studies

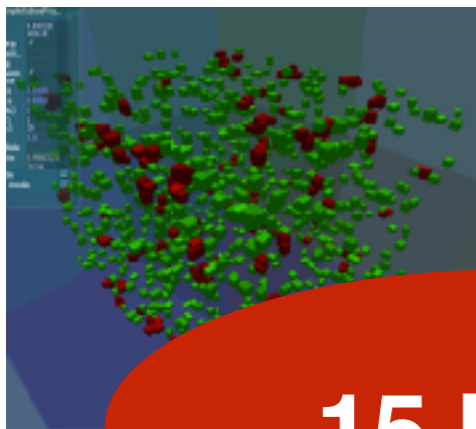
- **Myria:** analytics



Analytics data indexed by
operation and by

11 bugs

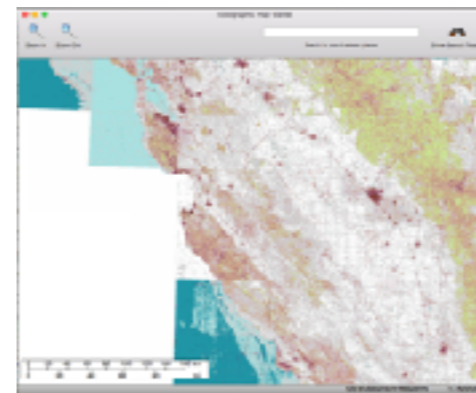
- **Bullet:** volume tree



Stores axis-aligned bounding boxes for fast collision detection

15 bugs

- **ZTopo:** tile cache



Tracks map tiles in a least-recently-used cache

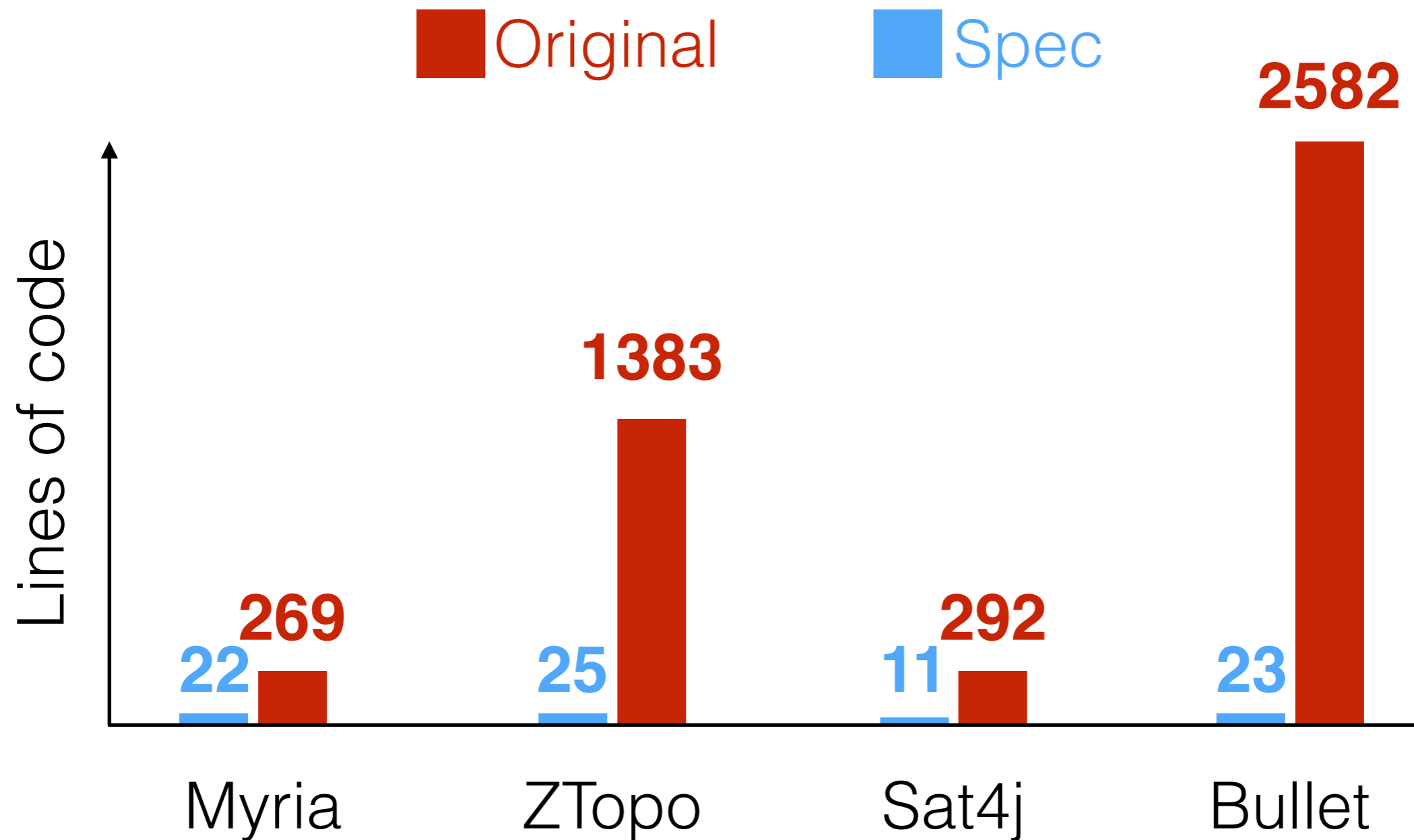
- **Sat4j:** variable metadata



Tracks information about each variable in the formula

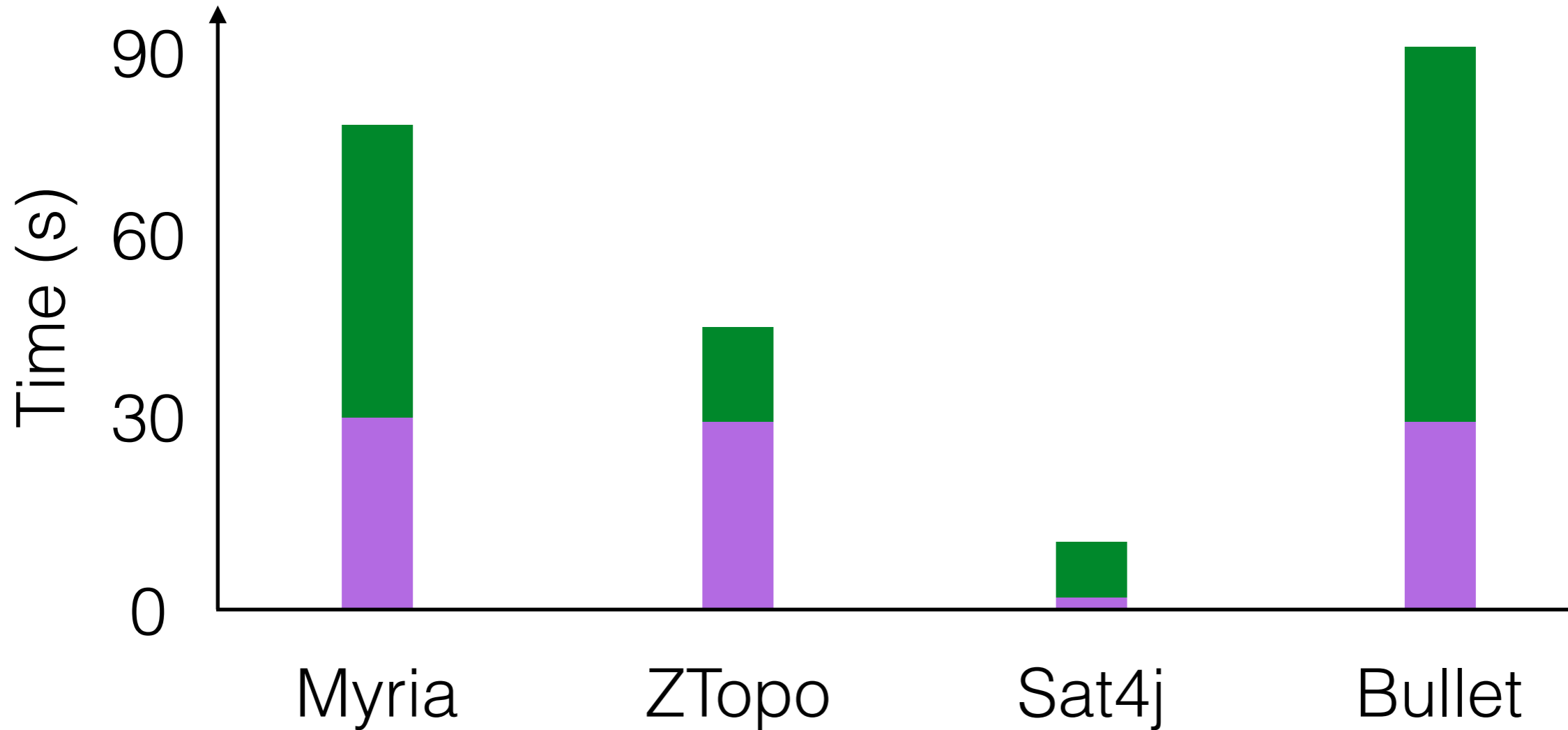
7 bugs

Specifications vs. Implementations



Synthesis Time

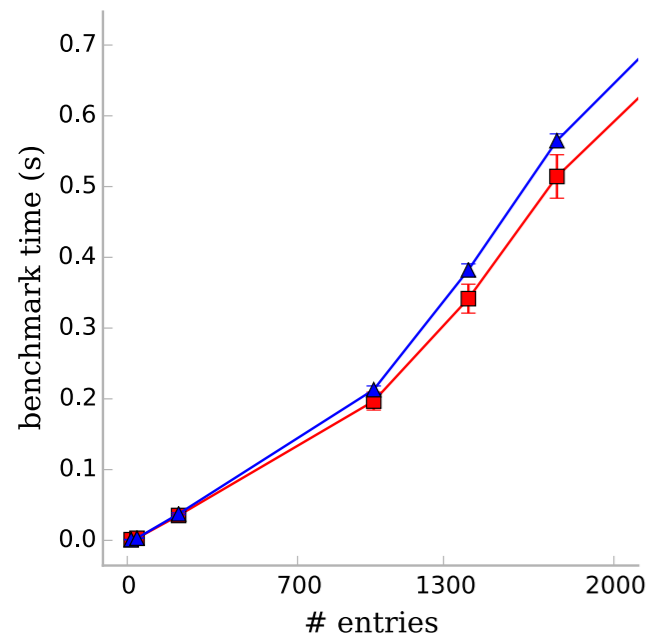
Outline Synthesis Auto-Tuning



Performance

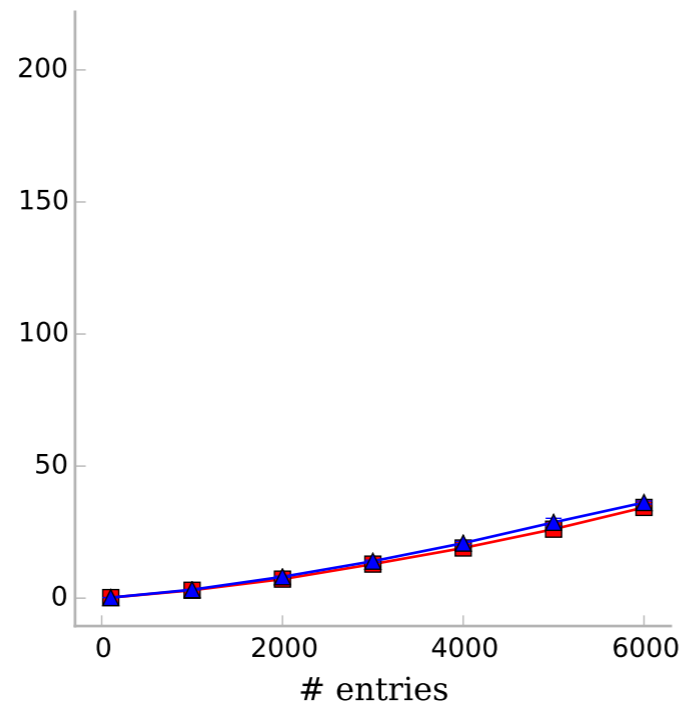
Original Synthesized

Data structures are nearly identical



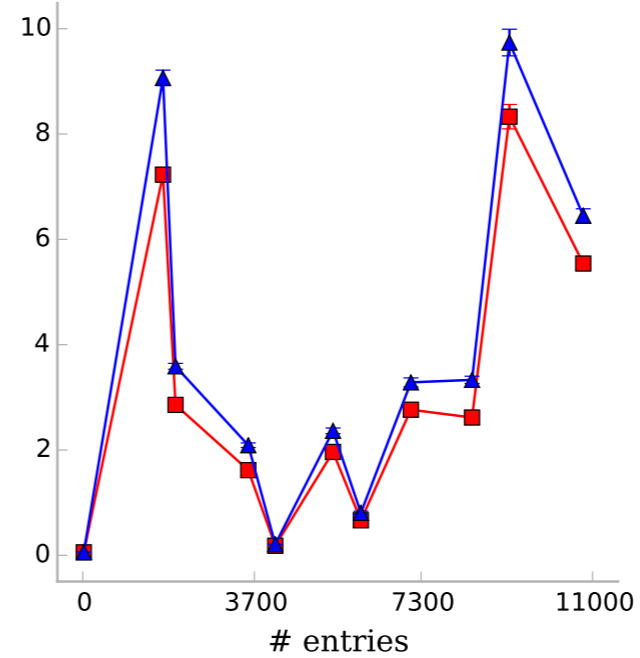
ZTopo

Binary search tree vs. space partitioning tree



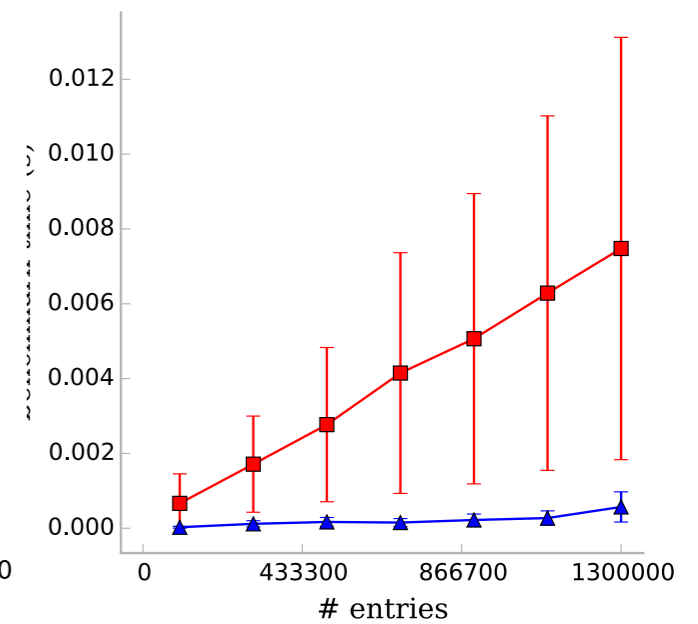
Bullet

Small overhead; performance dominated by other factors



Sat4j

Original implementation has worst-case linear time

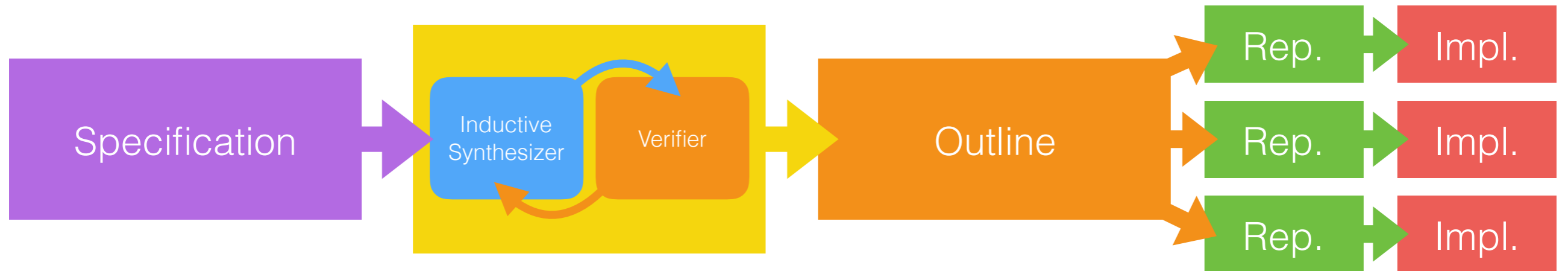


Myria

Related Work

- J. Earley: “**High level iterators and a method for automatically designing data structure representation**” (1974)
 - Hard-coded rewrite rules
- S. Agrawal et al: “**Automated selection of materialized views and indexes in sql databases**” (2000)
 - Enumerate possible views & indexes based on query syntax and use the planner to decide which ones to keep
- P. Hawkins et al: “**Data representation synthesis**” (2011)
 - Enumerate representations and use a planner to implement retrieval operations; conjunctions of equalities only

<http://cozy.uwplse.org>



- Implementation outlines make the problem tractable
- Synthesis completes < 90 seconds
- Cozy generates correct code, and matches handwritten implementation performance

Special thanks to:



Michael
Ernst



Emina
Torlak

also Haoming Liu &
Daniel Perelman