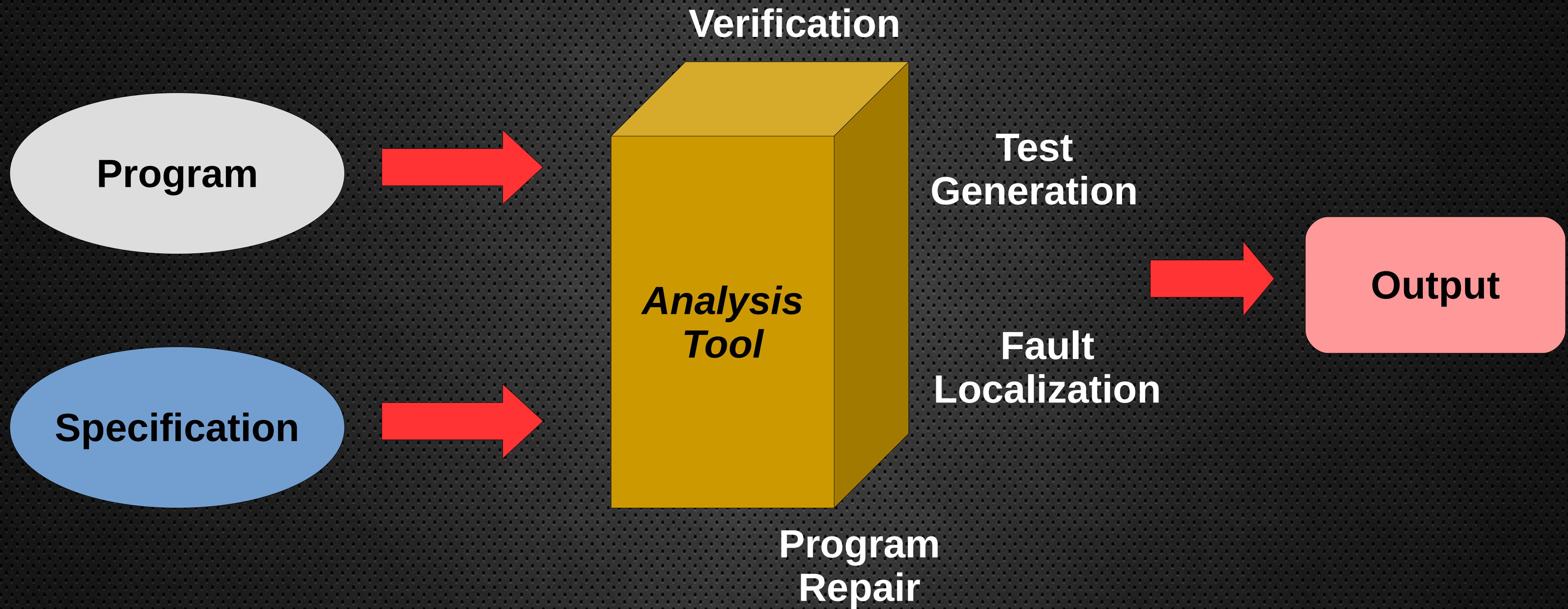
An Evolutionary Approach to Translate Operational Specifications into Declarative Specifications

Facundo Molina – César Cornejo – Renzo Degiovanni – Germán Regis – Pablo Castro – Nazareno Aguirre – Marcelo Frias

Department of Computer Science – FCEFQyN
National University of Río Cuarto
Argentina

November - 2016

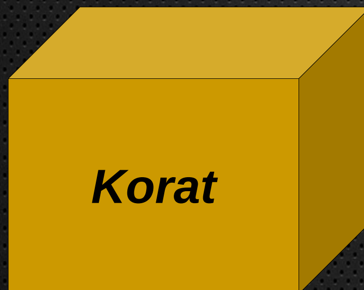
Motivation



Specification Styles

Operational

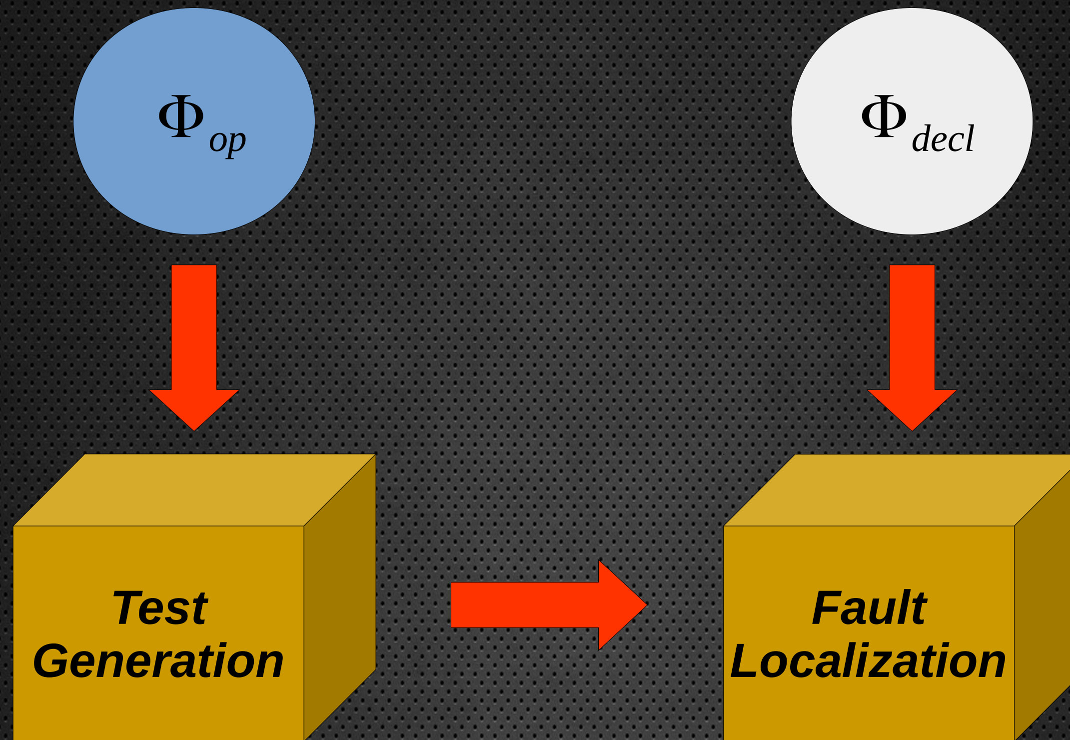
```
public boolean repOK () {
  Set<Entry> visited = new HashSet<Entry>();
  visited.add(header);
  Entry current = header;
  while (true) {
    Entry next = current.getNext();
    if (next == null) break;
    if (!visited.add(next)) return false;
    current = next;
  }
  if (visited.size() != size) return false;
  return true ;
}
```



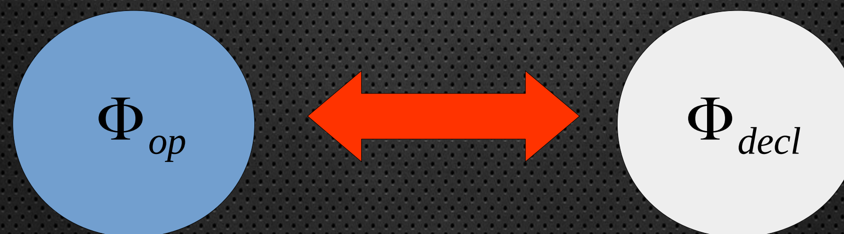
Declarative

```
pred repOK [thiz:List, header: List -> one Node+Null,
         size: List -> one Int, next: Node -> one Node + Null] {

       (all n: thiz.header.*next | n !in n.^next)
                     and
       (# thiz.header.*next = thiz.size)
}
```

Cross-usage of automated analysis tools



What if we have the specification in just one style?



Semantics-preserving Translations

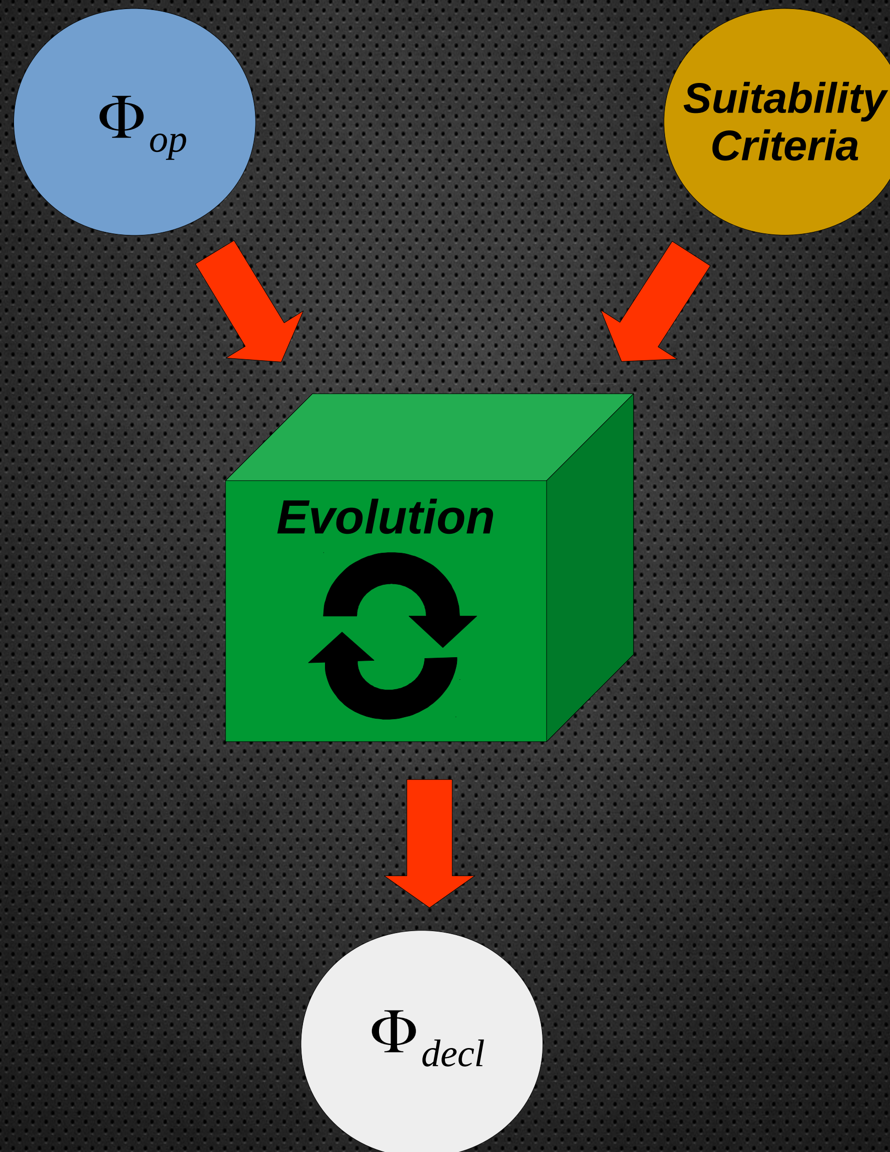
```
public boolean repOK () {
  Set<Entry> visited = new HashSet<Entry>();
  visited.add(header);
  Entry current = header;
  while (true) {
    Entry next = current.getNext();
    if (next == null) break;
    if (!visited.add(next)) return false;
    current = next;
  }
  if (visited.size() != size) return false;
  return true ;
}
```

```
pred repOK[thiz_0: List, header_0: List ->one (Node + Null),
    size_0: List ->one Int, next_0: Node ->one (Node + Null),
    result_0, result_1: boolean] {

    nodesToVisit_1 = thiz_0.size_0 and
    current_1 = thiz_0.header_0 and ((lt[thiz_0.size_0, 0] and
    result_1 = false and current_1 = current_4 and
    nodesToVisit_1 = nodesToVisit_4 ) or (not
    lt[thiz_0.size_0,0] and ((current_1 = current_4 and
    nodesToVisit_1 = nodesToVisit_4 ) or
    (gt[nodesToVisit_1, 0] and current_1 != Null and
    nodesToVisit_2 = sub[nodesToVisit_1, 1] and
    current_2 = current_1.next_0 and ((current_2 = current_4 and
    nodesToVisit_2 = nodesToVisit_4 ) or (gt[nodesToVisit_2, 0]
    and current_2 != Null and nodesToVisit_3 =
    sub[nodesToVisit_2,1] and  current_3 = current_2.next_0 and
    ((current_3 = current_4 and
    nodesToVisit_3 = nodesToVisit_4 ) or (gt[nodesToVisit_3, 0]
    and current_3 != Null and nodesToVisit_4 =
    sub[nodesToVisit_3, 1] and current_4 =
    current_3.next_0))))))
    and not (gt[nodesToVisit_4, 0] and current_4 != Null ) and
    ((eq[nodesToVisit_4, 0] and current_4 = Null and
    result_1 = true) or (not (eq[nodesToVisit_4, 0] and
    current_4 = Null) and result_1 = false))))
}
```

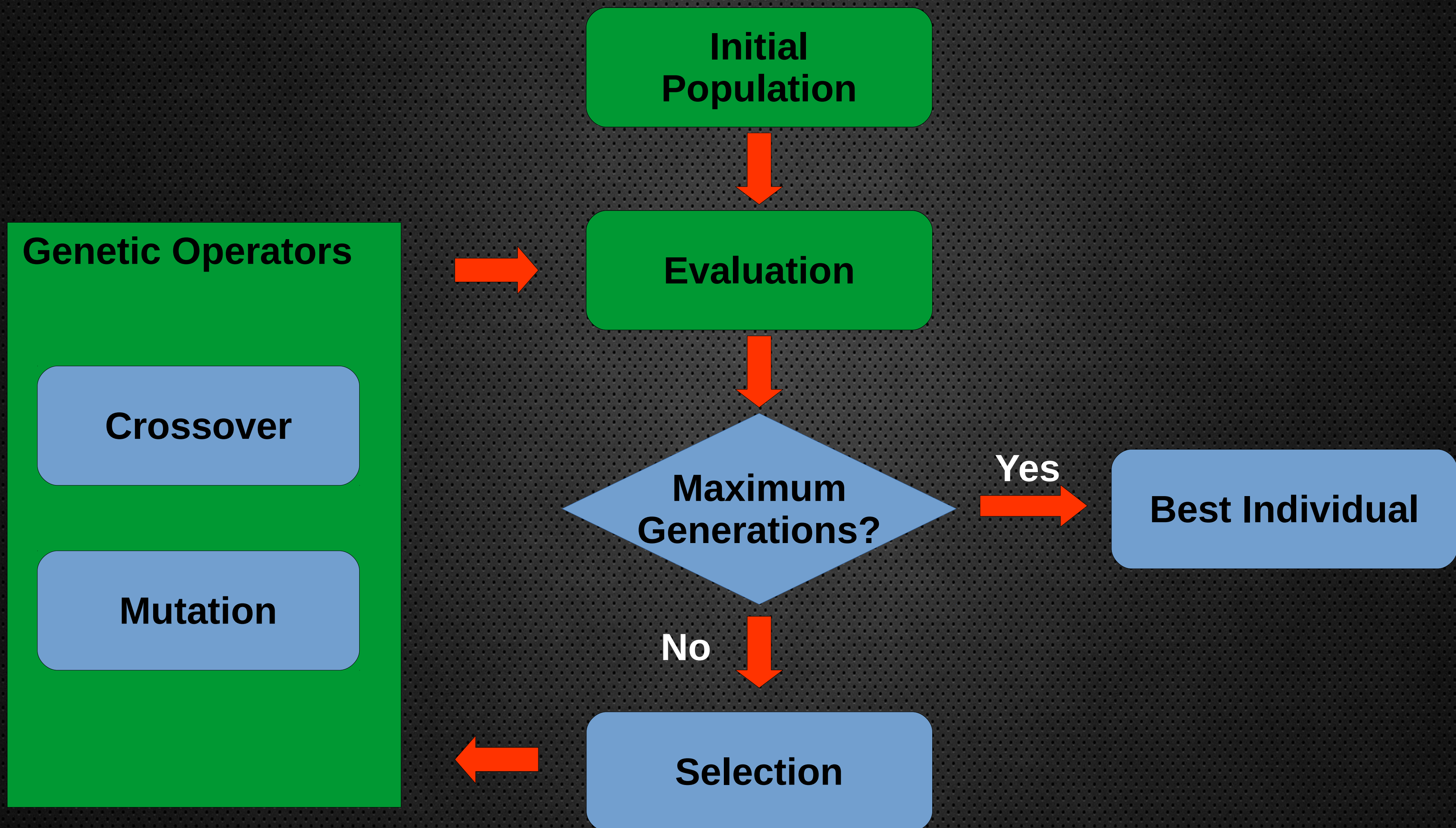
```
pred repOK [thiz:List, header: List -> one Node+Null,
    size: List -> one Int, next: Node -> one Node + Null] {

    (all n: thiz.header.*next | n !in n.^next)
                  and
    (# thiz.header.*next = thiz.size)
}
```

- The output is inappropriate
- The efficiency of tools sometimes are very dependent on how specifications are written

An Evolutionary Algorithm for Learning Declarative Specifications



Anatomy of an Evolutionary Algorithm



Population individuals

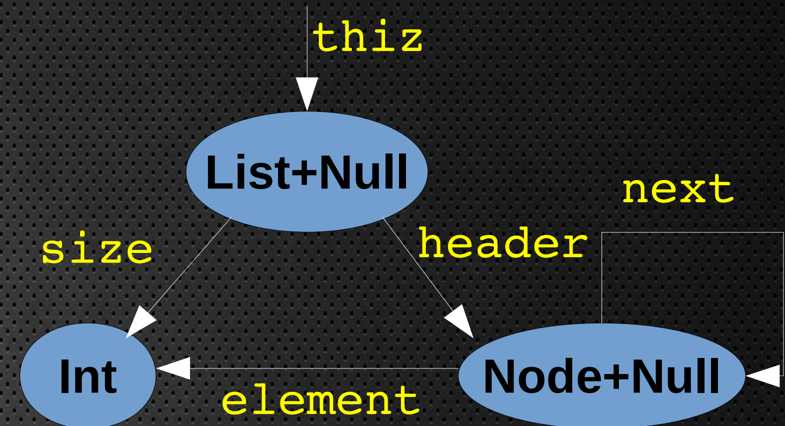
- Each chromosome (individual) will represent a candidate specification
  - A chromosome is a vector of genes $C = [\, g_0 \mid g_1 \mid g_2 \mid g_3 \,]$
  - A gene can be
    - a boolean constant `true`
    - an atomic formula `thiz.header != Null`
    - a quantified formula `all n : thiz.header.*next : n != Null`
- The specification represented by a chromosome is the *conjunction* of its genes

$$spec(C) = g_0 \wedge g_1 \wedge g_2 \wedge g_3$$

The initial Chromosomes

- **Create a type graph from the data structure definition:**

```
thiz: List
header: List → one Node+Null
next: Node → one Node+Null
size: List → one Int
element: Node → one Int
```



- **Create a set of evaluable expressions considering a *scope*:**

```
thiz
thiz.size
thiz.header
thiz.header.element
thiz.header.next
thiz.header.next.element
thiz.header.next.next
```

```
thiz.header != Null
thiz.size = 2
thiz.header.next != Null
```

```
thiz.header.*next
thiz.header.*next.element
```

```
all n : thiz.header.*next : n != Null
```

Genetic Operators

Crossover

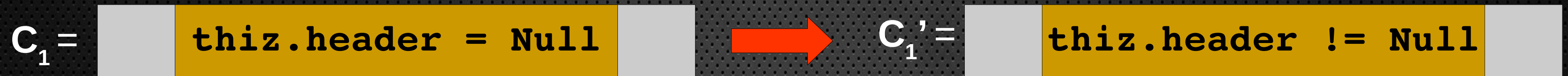
$C_1 =$	α_1	β_1
$C_2 =$	α_2	β_2

$C_1' =$	α_1	β_2
$C_2' =$	α_2	β_1

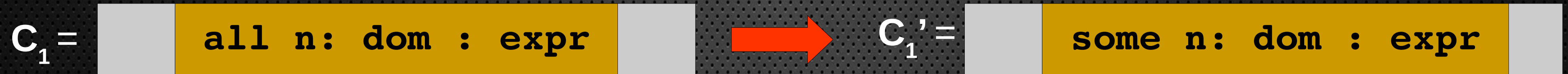
Mutation

- set to `true`

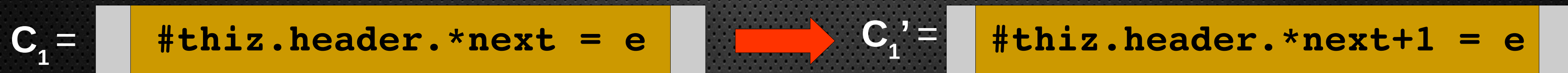
- operator replacement



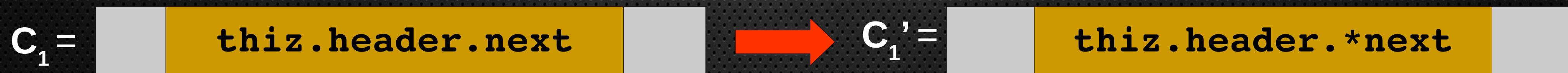
- quantifier replacement



- integer addition/substraction

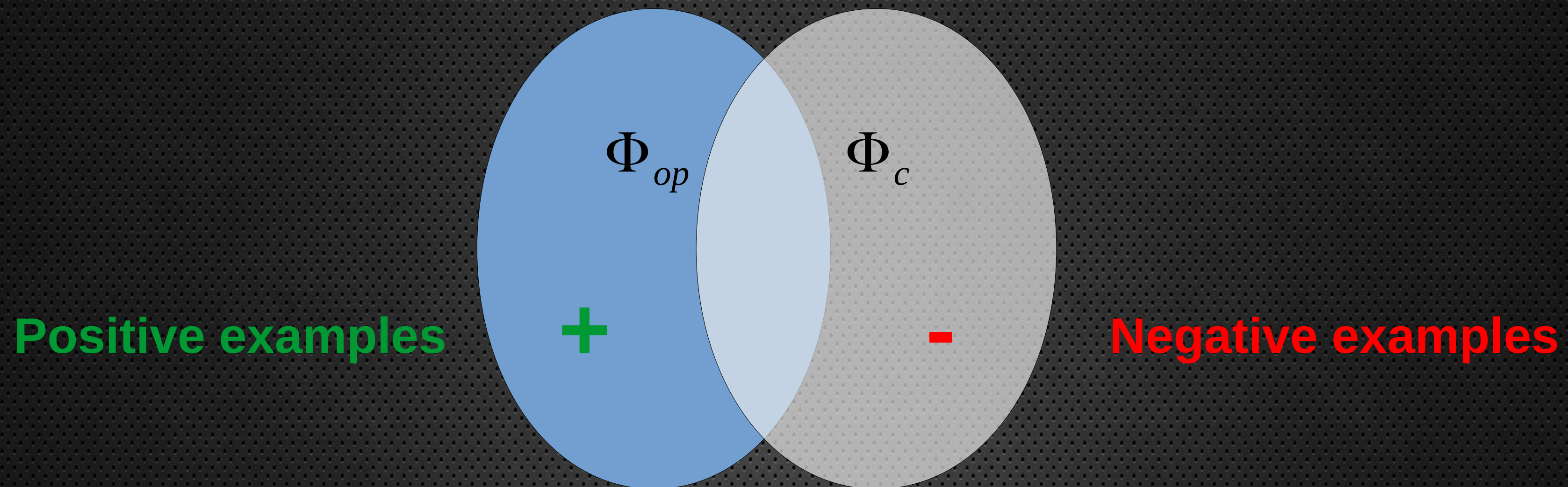


- closure operator insertion/elimination

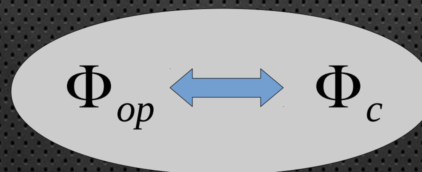


Fitness Function

- **For a chromosome *c*, assess how close the chromosome specification to the desired specification is:**



- **The fitness value of *c* is computed by counting the examples(+ and -) that do not satisfy**



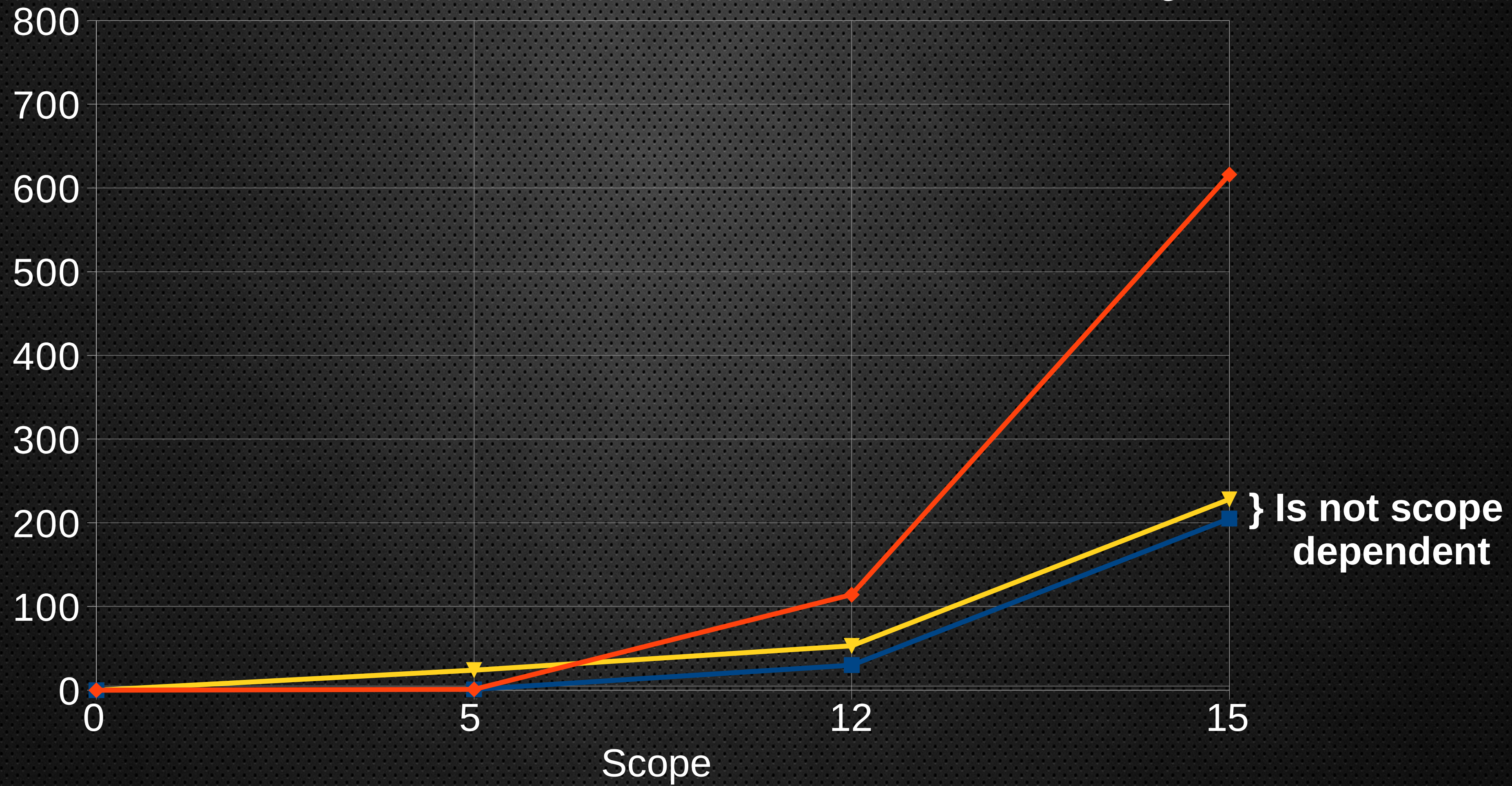
- The fewer the examples, the better
- The smaller the specification, the better

Evaluation

RQ1 – Is it efficient and worthwhile?

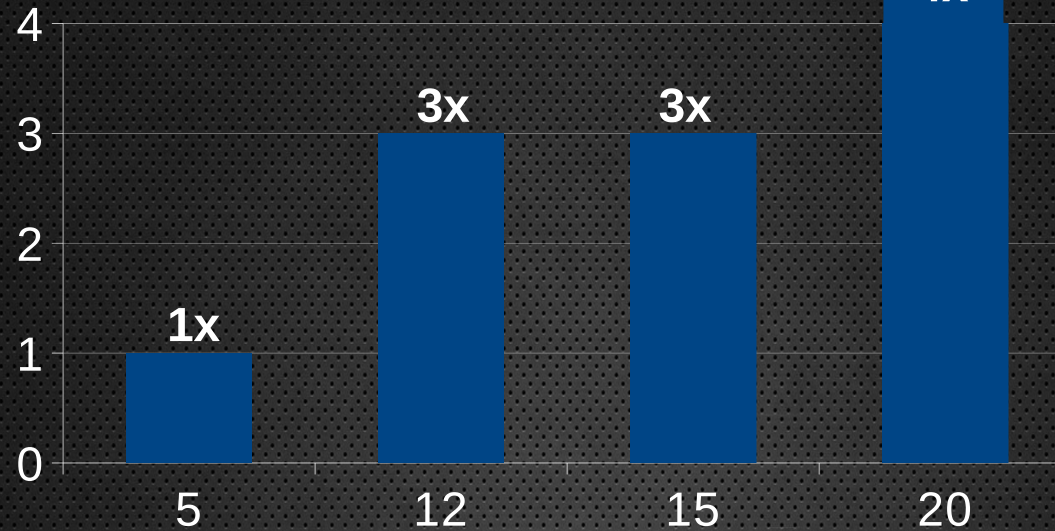
Singly Linked Sorted List – Verification in a bounded scenario

Total time (seconds) Operational Relational Relational + Average learning time



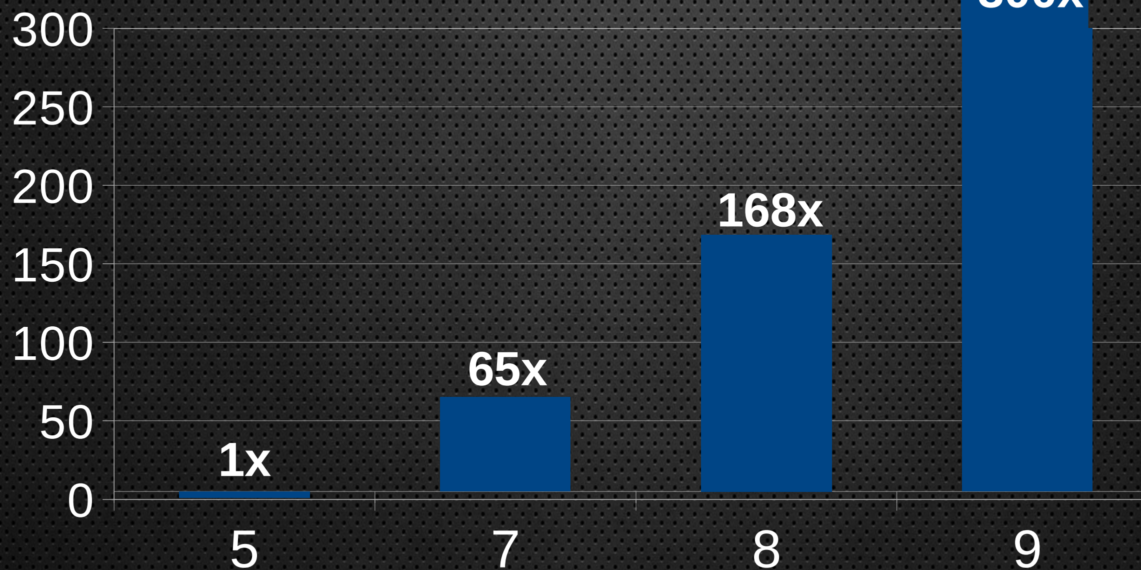
Singly Linked Sorted List – Verification in a bounded scenario

Speed up (w.r.t the operational specification)



Binary trees – Verification in a bounded scenario

Speed up (w.r.t the operational specification)



RQ2 – Is it precise ?

Data Structure	Invariant Learned ?
singly linked list	✔
sorted singly linked list	✔
circular linked list	✔
binary trees	✔
heaps	✔
directed acyclic graph (binary)	✔
red-black trees	✔

For red-black trees we are able to learn most of the expected invariant, except for the "*black height*" portion of it.

Future work

- **Generalize the approach in order to produce a richer set of specifications**
- **Analyze our approach in other kinds of programs, not just data structure representation invariants**
- **Implement cross usages of analysis tools using our algorithm**

Thank you !