Roadmap

• Program Analyses we deal with
• Tight field bounds and their computation
• Distributing program analyses
• Some experimental data
• Conclusions
SAT-Based Bounded Bug Finding and Symbolic Execution
SAT-Based Bounded Bug Finding

• Given a program with a contract (pre and post conditions, representation invariants, variant functions, etc.), we want to automatically detect the existence of faults.

• We automatically generate an input exposing the failure.
TACO: Translation of Annotated Code

The process

Translation of spec formula to propositional formula

\{Q\}

\{R\}

Translation of code to propositional formula

\(F_Q\)

\(F_P\)

\(F_R\)

\(F_Q\) \&\& \(F_P\) \&\& \! \(F_R\)

SAT-solver

\(\text{val}\)

Input generator

\( i \)
TACO: Translation of Annotated Code: Demo

Symbolic Execution
Symbolic Execution

Goals

• Automated test input generation
• Automated bug detection
• Quite scalable for certain domains (primitive types using simple constraints)
• Very active research topic when applied to complex objects (data structures).
Symbolic Execution

An example

```java
public int min3(int i, int j, int k){
    int output = 0;
    if (i <= j && i <= k)
        output = i;
    else
        if (i <= j || k <= j)
            output = k;
        else
            output = j;
    return output;
}
```
Symbolic Execution

An example

```java
public int min3(int i, int j, int k) {
    int output = 0;
    if (i <= j && i <= k)
        output = i;
    else
        if (i <= j || k <= j)
            output = k;
        else
            output = j;
    return output;
}
```

(i=i0, j=j0, k=k0, true)
Symbolic Execution

An example

```java
public int min3(int i, int j, int k)
{
    int output = 0;
    if (i <= j && i <= k)
        output = i;
    else
        if (i <= j || k <= j)
            output = k;
        else
            output = j;
    return output;
}
```

(i=i0, j=j0, k=k0, output=0, true)
Symbolic Execution

An example

```java
public int min3(int i, int j, int k){
    int output = 0;
    if (i <= j && i <= k)
        output = i;
    else
        if (i <= j || k <= j)
            output = k;
        else
            output = j;
    return output;
}
```

(i=i0, j=j0, k=k0, output=0, i0>j0 || i0>k0)
Symbolic Execution

An example

```java
public int min3(int i, int j, int k){
    int output = 0;
    if (i <= j && i <= k)
        output = i;
    else
        if (i <= j || k <= j)
            output = k;
        else
            output = j;
    return output;
}
```

(i=i0, j=j0, k=k0, output=0, (i0>j0 || i0>k0) && (i0<=j0 || k0<=j0))
Symbolic Execution

An example

```java
public int min3(int i, int j, int k){
    int output = 0;
    if (i <= j && i <= k)
        output = i;
    else
        if (i <= j || k <= j)
            output = k;
        else
            output = j;
    return output;
}
```

(i=i0, j=j0, k=k0, output=k0, (i0>j0 || i0>k0) && (i0<=j0 || k0<=j0))
Symbolic Execution

An example

public int min3(int i, int j, int k) {
    int output = 0;
    if (i <= j && i <= k)
        output = i;
    else
        if (i <= j || k <= j)
            output = k;
        else
            output = j;
    return output;
}
Symbolic Execution
For Dynamically Allocated Structures

- Khurshid, Pasareanu and Visser proposed Lazy Initialization [TACAS’03].

- An object attribute is initialized at the time its value is accessed. Up tp that moment, the attribute value is kept symbolic.
Lazy Initialization (LI)

An example

class Node {
    int elem;
    Node next;

    \requires Acyclic
    Node sortFirstTwo() {
        if (next != null)
            if (elem > next.elem) {
                Node t = next;
                next = t.next;
                t.next = this;
                return t;
            }
        return this;
    }
}
Lazy Initialization (LI)

An example

class Node {
    int elem;
    Node next;

    \requires Acyclic
    Node sortFirstTwo() {
        if (next != null) {
            if (elem > next.elem) {
                Node t = next;
                next = t.next;
                t.next = this;
                return t;
            }
        }
        return this;
    }
}
Lazy Initialization (LI)

An example

class Node {
    int elem;
    Node next;

    \requires Acyclic
    Node sortFirstTwo() {
        if (next != null) {
            if (elem > next.elem) {
                Node t = next;
                next = t.next;
                t.next = this;
                return t;
            }
        }
        return this;
    }
}
Lazy Initialization (LI)

An example

class Node {
    int elem;
    Node next;

    \requires Acyclic
    Node sortFirstTwo() {
        if (next != null) {
            if (elem > next.elem) {
                Node t = next;
                next = t.next;
                t.next = this;
                return t;
            }
        }
        return this;
    }
}
Lazy Initialization (LI)

An example

class Node {
    int elem;
    Node next;

    \requires Acyclic
    Node sortFirstTwo() {
        if (next != null) {
            if (elem > next.elem) {
                Node t = next;
                next = t.next;
                t.next = this;
                return t;
            }
        }
        return this;
    }
}
Lazy Initialization (LI)

An example

class Node {
    int elem;
    Node next;
    
    \requires Acyclic
    Node sortFirstTwo() {
        if (next != null) {
            if (elem > next.elem) {
                Node t = next;
                next = t.next;
                t.next = this;
                return t;
            }
        }
        return this;
    }
}

An example
Lazy Initialization (LI)

An example

class Node {
    int elem;
    Node next;

    \requires Acyclic
    Node sortFirstTwo() {
        if (next != null) {
            if (elem > next.elem) {
                Node t = next;
                next = t.next;
                t.next = this;
                return t;
            }
        }
        return this;
    }
}
Lazy Initialization (LI)

An example

class Node {
    int elem;
    Node next;
}

\texttt{\textbackslash requires Acyclic}
Node sortFirstTwo() {
    if (next != null)
        if (elem > next.elem) {
            Node t = next;
            next = t.next;
            t.next = this;
            return t;
        }
    return this;
}

\texttt{N0.next != null}
Lazy Initialization (LI)

An example

class Node {
    int elem;
    Node next;

    \requires Acyclic
    Node sortFirstTwo() {
        if (next != null) {
            if (elem > next.elem) {
                Node t = next;
                next = t.next;
                t.next = this;
                return t;
            }
        }
        return this;
    }
}
Lazy Initialization (LI)

An example

class Node {
    int elem;
    Node next;
}

\Requires Acyclic
Node sortFirstTwo() {
    if (next != null)
        if (elem > next.elem) {
            Node t = next;
            next = t.next;
            t.next = this;
            return t;
        }
    return this;
}
class Node {
    int elem;
    Node next;
}

\texttt{\textbackslash requires Acyclic}
Node sortFirstTwo() {
    if (next != null)
        if (elem > next.elem) {
            Node t = next;
            next = t.next;
            t.next = this;
            return t;
        }
    return this;
}
Lazy Initialization (LI)

An example

class Node {
    int elem;
    Node next;

    \requires Acyclic
    Node sortFirstTwo() {
        if (next != null)
            if (elem > next.elem) {
                Node t = next;
                next = t.next;
                t.next = this;
                return t;
            }
        return this;
    }
}
Lazy Initialization (LI)

An example

class Node {
    int elem;
    Node next;

    \texttt{requires} Acyclic
    Node sortFirstTwo() {
        if (next != null)
            if (elem > next.elem) {
                Node t = next;
                \texttt{next} = t.\texttt{next};
                t.\texttt{next} = this;
                return t;
            }
        return this;
    }
}
Lazy Initialization (LI)
An example

class Node {
    int elem;
    Node next;
}
\requires Acyclic
Node sortFirstTwo() {
    if (next != null)
        if (elem > next.elem) {
            Node t = next;
            next = t.next;
            t.next = this;
            return t;
        }
    return this;
}
Lazy Initialization (LI)

An example

class Node {
    int elem;
    Node next;
}

\|requires Acyclic
Node sortFirstTwo() {
    if (next != null)
        if (elem > next.elem) {
            Node t = next;
            next = t.next;
            t.next = this;
            return t;
        }
    return this;
}

N0

N0.next != null

N0.next != null && E0>E1

An example
Lazy Initialization (LI)

Remarks

• Lazy Initialization does not generate isomorphic heaps.

• Constraints must be adapted so that they can be evaluated on partially symbolic heaps.
Symbolic Execution
For Dynamically Allocated Structures
Symbolic Execution
For Dynamically Allocated Structures
Symbolic Execution
For Dynamically Allocated Structures
Symbolic Execution
For Dynamically Allocated Structures
NASA’s Symbolic Pathfinder Demo
Tight Field Bounds
Intuition: propositional variables encode the relationship between objects in the memory heap, i.e., for each triple (O1, f, O2), where O1, O2 are objects and f is a field, propositional variable \( V(O1,f,O2) \) is true iff \( O1.f = O2 \).
If we use a scope of 3 nodes N0, N1, N2 in a data structure with a single field f, we will have variables:

\[ V(N0,f,N0), \ V(N0,f,N1), \ V(N0,f,N2), \ V(N1,f,N0), \ V(N1,f,N1), \ V(N1,f,N2), \ V(N2,f,N0), \ V(N2,f,N1), \ V(N2,f,N2) \]
The role of the SAT-Solver

The solver, while exploring the valuations for these variables, actually searches for a memory heap in which the program violates its contract.

\[ V(N0,f,N0), \ V(N0,f,N1), \ V(N0,f,N2), \ V(N1,f,N0), \ V(N1,f,N1), \ V(N1,f,N2), \ V(N2,f,N0), \ V(N2,f,N1), \ V(N2,f,N2) \]
Tight Field Bounds (4)

Even more unfeasible valuations

If in any list instance node identifiers are chose from the head following the order $N_0, N_1, N_2,$

$V(N_0,\text{next},N_0), V(N_0,\text{next},N_1), V(N_0,\text{next},N_2),$ $V(N_1,\text{next},N_0), V(N_1,\text{next},N_1), V(N_1,\text{next},N_2),$ $V(N_2,\text{next},N_0), V(N_2,\text{next},N_1), V(N_2,\text{next},N_2)$
Tight Field Bounds (4)

Even more unfeasible valuations

If in any list instance node identifiers are chose from the head following the order N0, N1, N2,

V(N0, next, N0), V(N0, next, N1), V(N0, next, N2),
V(N1, next, N0), V(N1, next, N1), V(N1, next, N2),
V(N2, next, N0), V(N2, next, N1), V(N2, next, N2)
Tight Field Bounds (4)

Even more unfeasible valuations

If in any list instance node identifiers are chose from the head following the order N0, N1, N2, V(N0,next,N0), V(N0,next,N1), V(N0,next,N2), V(N1,next,N0), V(N1,next,N1), V(N1,next,N2), V(N2,next,N0), V(N2,next,N1), V(N2,next,N2)
Tight Field Bounds (4)

Even more unfeasible valuations

If in any list instance node identifiers are chose from the head following the order N0, N1, N2, V(N0,next,N0), V(N0,next,N1), V(N0,next,N2), V(N1,next,N0), V(N1,next,N1), V(N1,next,N2), V(N2,next,N0), V(N2,next,N1), V(N2,next,N2)
Tight Field Bounds (4)

Even more unfeasible valuations

If in any list instance node identifiers are chose from the head following the order N0, N1, N2,

\[ V(N0, next, N0), V(N0, next, N1), V(N0, next, N2), V(N1, next, N0), V(N1, next, N1), V(N1, next, N2), V(N2, next, N0), V(N2, next, N1), V(N2, next, N2) \]
Tight Field Bounds (4)

Even more unfeasible valuations

If in any list instance node identifiers are chose from the set \{N0, N1, N2, \ldots\}, we have:

- \( V(N0, next, N0) \)
- \( V(N0, next, N1) \)
- \( V(N0, next, N2) \)
- \( V(N1, next, N0) \)
- \( V(N1, next, N1) \)
- \( V(N1, next, N2) \)
- \( V(N2, next, N0) \)
- \( V(N2, next, N1) \)
- \( V(N2, next, N2) \)

Symmetry-breaking predicates
Tight Field Bounds (5)

What’s cool about unfeasible valuations?

Variables that are forced to be false can actually be removed and be substituted by their known value (false). In this way the SAT-solver has less work to do (exponential speedup).

\[ V(N_0, \text{next}, N_0), \; V(N_0, \text{next}, N_1), \; V(N_0, \text{next}, N_2), \; V(N_1, \text{next}, N_0), \; V(N_1, \text{next}, N_1), \; V(N_1, \text{next}, N_2), \; V(N_2, \text{next}, N_0), \; V(N_2, \text{next}, N_1), \; V(N_2, \text{next}, N_2) \]
Tight Field Bounds (5)

What’s cool about unfeasible valuations?

Variables that are forced to be false can actually be removed and be substituted by their known value (false). In this way the SAT-solver has less work to do (exponential speedup).

V(N0,next,N0), V(N0,next,N1), V(N0,next,N2), V(N1,next,N0), V(N1,next,N1), V(N1,next,N2), V(N2,next,N0), V(N2,next,N1), V(N2,next,N2)
Tight Field Bounds (5)

What’s cool about unfeasible valuations?

Variables that are forced to be false can actually be removed and be substituted by their known value (false). In this way the SAT-solver has less work to do (exponential speedup).

V(N0, next, N0), V(N0, next, N1), V(N0, next, N2), V(N1, next, N0), V(N1, next, N1), V(N1, next, N2), V(N2, next, N0), V(N2, next, N1), V(N2, next, N2)
Tight Field Bounds (5)

What’s cool about unfeasible valuations?

Variables that are forced to be false can actually be removed and be substituted by their known value (false). In this way the SAT-solver has less work to do (exponential speedup).

V(N0,next,N0), V(N0,next,N1), V(N0,next,N2), V(N1,next,N0), V(N1,next,N1), V(N1,next,N2), V(N2,next,N0), V(N2,next,N1), V(N2,next,N2)
Tight Field Bounds (5)

What's cool about unfeasible valuations?

Variables that are forced to be false can actually be removed and be substituted by their known value (false). In this way the SAT-solver has less work to do (exponential speedup).

V(N0,next,N0), V(N0,next,N1), V(N0,next,N2), V(N1,next,N0), V(N1,next,N1), V(N1,next,N2), V(N2,next,N0), V(N2,next,N1), V(N2,next,N2)
Tight Field Bounds (6)

Definition

Given object identifiers N₀, ..., Nᵢ and a field f, the tight field bound tfb(f) is a minimal binary relation on N₀, ..., Nᵢ, null such that (Nᵢ, Nⱼ) in tfb(f) iff there exists a valid memory heap satisfying the symmetry-breaking predicates in which Nᵢ.f = Nⱼ (idem for null).
Tight Field Bounds (7)

Valid

Satisfying the representation invariant (AVL? Red-Black Tree? BinHeap?)
Tight Field Bounds (7)
Satisfying Symmetry-Breaking

Object identifiers in the heap follow a canonical ordering: BFS-traversal.
Rule 1: The graph’s root is labeled N0.
Rule 2: Two nodes with the same parent are labeled from left to right.
Tight Field Bounds (7)

Satisfying Symmetry-Breaking

Rule 2: Two nodes with the same parent are labeled from left to right.

```
N0
  \  /\  \\
  3  6  \\
 /\     /\     /\     \
2  4   5  3  6  4  5
```
Rule 2: Two nodes with the same parent are labeled from left to right.
Tight Field Bounds (7)

Satisfying Symmetry-Breaking

Applying once again Rule 2,
Applying once again Rule 2,
Tight Field Bounds (7)
Satisfying Symmetry-Breaking

Applying once again Rule 2,
There are two algorithms, a distributed and a sequential one. Will focus on the sequential.

Intuition: Incrementally ask the SAT-solver for valid heaps, each one contributing new pairs to the bound, until the solver says “UNSAT”.
Tight Field Bounds (9)

Computing Tight Field Bounds: Example

Consider AVL trees with up to 4 nodes, and fields left and right.

left = {}
Tight Field Bounds (9)

Computing Tight Field Bounds: Example

Consider AVL trees with up to 4 nodes, and fields `left` and `right`.

```
left = {}
```

```
N0 5
N1 3
```
Tight Field Bounds (9)

Computing Tight Field Bounds: Example

Consider AVL trees with up to 4 nodes, and fields left and right.

\[ \text{left} = \{(N0,N1)\} \]
Tight Field Bounds (9)

Computing Tight Field Bounds: Example

Consider AVL trees with up to 4 nodes, and fields \texttt{left} and \texttt{right}.

\[
\text{left} = \{(N0,N1)\}
\]
Tight Field Bounds (9)

Computing Tight Field Bounds: Example

Consider AVL trees with up to 4 nodes, and fields left and right.

\[ \text{left} = \{(N0,N1)\} \]
Tight Field Bounds (9)

Computing Tight Field Bounds: Example

Consider AVL trees with up to 4 nodes, and fields left and right.

\[ \text{left} = \{(N0,N1), (N1,N3)\} \]
Tight Field Bounds (9)

Computing Tight Field Bounds: Example

Consider AVL trees with up to 4 nodes, and fields left and right.

left = {(N0,N1), (N1,N3)}
Tight Field Bounds (9)

Computing Tight Field Bounds: Example

Consider AVL trees with up to 4 nodes, and fields left and right.

\[ \text{left} = \{(N0,N1), (N1,N3)\} \]
Tight Field Bounds (9)

Computing Tight Field Bounds: Example

Consider AVL trees with up to 4 nodes, and fields left and right.

$\text{left} = \{(N0, N1), (N1, N3), (N2, N3)\}$
Tight Field Bounds in Bug Finding

Demo

- All variables that represent pairs of nodes that are not in the bound, are set to false on translation.

- In this way, the SAT-problem has fewer propositional variables.
Tight Field Bounds in Symbolic Execution

Intuition

• Pairs that are outside the tight field bounds are not used as options during lazy initialization.

• In this way, there are fewer cases to consider.

Tight Field Bounds in Symbolic Execution

Intuition

N0 → null, N1
N1 → null, N3
N2 → null, N3
N3 → null

LI

Uleft
Tight Field Bounds in Symbolic Execution

Intuition

N0 ➞ null, N1
N1 ➞ null, N3
N2 ➞ null, N3
N3 ➞ null
Tight Field Bounds in Symbolic Execution

Intuition

N0 → null, N1
N1 → null, N3
N2 → null, N3
N3 → null

LI

Uleft
Tight Field Bounds in Symbolic Execution

Intuition

LI

N0 → null, N1
N1 → null, N3
N2 → null, N3
N3 → null

Uleft
Intuition

Tight Field Bounds in Symbolic Execution

LI

N0 → null, N1
N1 → null, N3
N2 → null, N3
N3 → null

Uleft
Tight Field Bounds in Symbolic Execution

Intuition

BLI

N0 → null, N1
N1 → null, N3
N2 → null, N3
N3 → null

Uleft
Tight Field Bounds in Symbolic Execution

Intuition

N0 → null, N1
N1 → null, N3
N2 → null, N3
N3 → null
Tight Field Bounds in Symbolic Execution

Intuition

BLI

N0 → null, N1
N1 → null, N3
N2 → null, N3
N3 → null

Uleft
Distributed Analysis by Refinement of Tight Field Bounds
### Bounds: A Running Example

<table>
<thead>
<tr>
<th>Fields</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Left</strong></td>
<td>( N_0 \rightarrow N_1 + N_0 \rightarrow \text{null} + N_1 \rightarrow N_3 + N_1 \rightarrow \text{null} + N_2 \rightarrow N_3 + N_2 \rightarrow N_4 + N_2 \rightarrow \text{null} + N_3 \rightarrow \text{null} + N_4 \rightarrow \text{null} )</td>
</tr>
<tr>
<td><strong>Right</strong></td>
<td>( N_0 \rightarrow N_1 + N_0 \rightarrow N_2 + N_0 \rightarrow \text{null} + N_1 \rightarrow N_3 + N_1 \rightarrow N_4 + N_1 \rightarrow \text{null} + N_2 \rightarrow N_3 + N_2 \rightarrow N_4 + N_2 \rightarrow \text{null} + N_3 \rightarrow \text{null} + N_4 \rightarrow \text{null} )</td>
</tr>
</tbody>
</table>

Fields *left* and *right* from Red-Black tree, up to 5 nodes.
The Distribution Idea

- Fields are functions. Therefore, there is still a degree of nondeterminism that can be eliminated from the bounds.

<table>
<thead>
<tr>
<th>left in N0→N1 + N0→null + N1→N3 + N1→null + N2→N3 + N2→N4 + N2→null + N3→null + N4→null</th>
</tr>
</thead>
<tbody>
<tr>
<td>right in N0→N1 + N0→N2 + N0→null + N1→N3 + N1→N4 + N1→null + N2→N3 + N2→N4 + N2→null + N3→null + N4→null</td>
</tr>
</tbody>
</table>
How to split problems

left in NO->N1 + NO->null
  + N1->N3 + N1->null
  + N2->N3 + N2->N4 + N2->null
  + N3->null
  + N4->null
Problem Splitting

Removing all nondeterminism from nodes N0 and N1 yields 36 subproblems.
Problem Explosion

- Red-Black tree 15 nodes, number of problems removing nondeterminism from nodes $N0-N(n-1)$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>#subprob.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>36</td>
</tr>
<tr>
<td>3</td>
<td>720</td>
</tr>
<tr>
<td>4</td>
<td>14,400</td>
</tr>
<tr>
<td>5</td>
<td>604,800</td>
</tr>
<tr>
<td>6</td>
<td>33,868,800</td>
</tr>
</tbody>
</table>
Taming problem explosion

1. Do not generate those subproblems that violate the symmetry breaking axioms.

2. Mine aliasing information, and avoid subproblems that contain unwanted aliasing.

3. Check first that the class invariant is satisfied.
Taming problem explosion

<table>
<thead>
<tr>
<th>TreeSet</th>
<th>NO</th>
<th>36</th>
<th>720</th>
<th>14,400</th>
<th>604,800</th>
<th>33,868,800</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OP1</td>
<td>9</td>
<td>56</td>
<td>250</td>
<td>3,028</td>
<td>36,163</td>
</tr>
<tr>
<td></td>
<td>OP2</td>
<td>7</td>
<td>19</td>
<td>52</td>
<td>184</td>
<td>694</td>
</tr>
<tr>
<td></td>
<td>OP3</td>
<td>7</td>
<td>19</td>
<td>34</td>
<td>100</td>
<td>172</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AVLTree</th>
<th>NO</th>
<th>36</th>
<th>720</th>
<th>8,640</th>
<th>259,200</th>
<th>14,515,200</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OP1</td>
<td>9</td>
<td>56</td>
<td>105</td>
<td>716</td>
<td>11,670</td>
</tr>
<tr>
<td></td>
<td>OP2</td>
<td>7</td>
<td>19</td>
<td>34</td>
<td>94</td>
<td>334</td>
</tr>
<tr>
<td></td>
<td>OP3</td>
<td>7</td>
<td>19</td>
<td>28</td>
<td>70</td>
<td>142</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BinHeap</th>
<th>NO</th>
<th>72</th>
<th>648</th>
<th>10,368</th>
<th>259,200</th>
<th>6,480,000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OP1</td>
<td>26</td>
<td>108</td>
<td>600</td>
<td>4,204</td>
<td>17,030</td>
</tr>
<tr>
<td></td>
<td>OP2</td>
<td>13</td>
<td>36</td>
<td>111</td>
<td>370</td>
<td>1,034</td>
</tr>
<tr>
<td></td>
<td>OP3</td>
<td>7</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>17</td>
</tr>
</tbody>
</table>

15 nodes. n = 2,...,6
Experimental Results for MUCHO-TACO: Distributed Bug Finding

<table>
<thead>
<tr>
<th>TSet</th>
<th>find</th>
<th>s10</th>
<th>s12</th>
<th>s15</th>
<th>s17</th>
<th>s20</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>01:39</td>
<td>06:17</td>
<td>93:17</td>
<td>260:56</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00:58</td>
<td>00:40</td>
<td>04:31</td>
<td>16:48</td>
<td>516:28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.7x</td>
<td>9.4x</td>
<td>20.6x</td>
<td>15.5x</td>
<td>(\gg 1.1x)</td>
</tr>
<tr>
<td>insert</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>08:37</td>
<td>56:41</td>
<td>241:52</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;69.6x</td>
<td>(\gg 10.6x)</td>
<td>(\gg 2.5x)</td>
<td>(\gg)</td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td>196:58</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>13:03</td>
<td>92:29</td>
<td>TO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15.1x</td>
<td>(\gg 6.5x)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BH</th>
<th>min</th>
<th>s10</th>
<th>s12</th>
<th>s15</th>
<th>s17</th>
<th>s20</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>00:14</td>
<td>00:17</td>
<td>01:31</td>
<td>02:51</td>
<td>07:26</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00:14</td>
<td>00:36</td>
<td>01:13</td>
<td>01:27</td>
<td>03:15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.0x</td>
<td>0.4x</td>
<td>1.2x</td>
<td>2.0x</td>
<td>2.3x</td>
</tr>
<tr>
<td>decrKey</td>
<td>30:26</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td></td>
<td>04:35</td>
<td>13:48</td>
<td>20:51</td>
<td>81:05</td>
<td>236:42</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.6x</td>
<td>(\gg 43.5x)</td>
<td>(\gg 28.8x)</td>
<td>(\gg 7.4x)</td>
<td>(\gg 2.5x)</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>37:30</td>
<td>218:13</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>08:13</td>
<td>19:02</td>
<td>21:05</td>
<td>114:36</td>
<td>325:28</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.6x</td>
<td>11.4x</td>
<td>(\gg 28.4x)</td>
<td>(\gg 5.2x)</td>
<td>(\gg 1.8x)</td>
<td></td>
</tr>
<tr>
<td>extrMin</td>
<td>36:52</td>
<td>TO</td>
<td>43:33</td>
<td>176:47</td>
<td>TO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>07:01</td>
<td>26:19</td>
<td>01:46</td>
<td>04:05</td>
<td>01:50</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.2x</td>
<td>(\gg 22.8x)</td>
<td>24.6x</td>
<td>43.3x</td>
<td>(\gg 1058x)</td>
<td></td>
</tr>
</tbody>
</table>
Experimental results for Symbolic Execution

Fig. 14. Analysis time (seconds, logscale) for increasing scope, method AvlTree.dfs().
Fig. 15. Analysis time (seconds, logscale) for increasing scope, method AvlTree.insert().
Experimental results for Symbolic Execution

Fig. 27. Analysis time (seconds, logscale) for increasing scope, method BinomialHeap.extractMin().
Distributed Symbolic Execution

Demo
Distributed Symbolic Execution

Demo
Conclusions

• We showed that the same partition technique can be used in two seemingly unrelated program analyses.

• In both cases, the results were very positive.

• If you are a user of a program analysis technique, and the technique may profit from the use of tight field bounds, then a distributed version of the analysis can be obtained almost for free by reducing nondeterminism from tight field bounds.
Thanks!