

# Heap Space Analysis for Java Bytecode

**Elvira Albert<sup>1</sup>, Samir Genaim<sup>2</sup>,  
Miguel Gómez-Zamalloa<sup>1</sup>**

(1) Complutense University of Madrid (Spain)

(2) Technical University of Madrid (Spain)

THE 2007 INTERNATIONAL SYMPOSIUM ON MEMORY MANAGEMENT  
(ISMM'07)

Montreal, October 22, 2007

# Introduction

## Cost Analysis

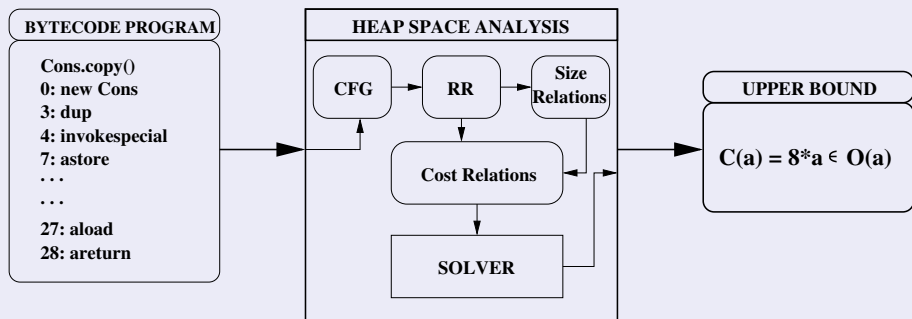
- Two important key features of a program are:
  - ▶ correctness
  - ▶ **efficiency**, i.e., the cost of program execution in terms of:
    - ★ time
    - ★ **memory**
- *Cost Analysis* is the automatic study of *program efficiency*.
- Cost Analysis has been studied for:
  - ▶ Declarative programming languages
  - ▶ High-level imperative programming languages
- Recently, we developed a *Cost Analysis framework for Java Bytecode*:
  - ▶ For mobile code, we do not have access to source code
  - ▶ We can use Cost Analysis to **accept/reject mobile code**
  - ▶ Java Bytecode (JB):
    - ★ widely used, specially in mobile systems
    - ★ security features, platform independent, ...

# Introduction

## Heap Space Analysis

- Our Cost Analysis statically generates *cost relations*:
  - ▶ They define the cost of a program as a function of its input data size.
  - ▶ They are *parametric* w.r.t. a *cost model*.
- First, we develop a cost model s.t.:
  - ▶ It defines the cost of memory allocation instructions (e.g., `new` and `newarray`) in terms of the number of heap units they consume.
  - ▶ The remaining bytecode instructions do not add any cost.
- We generate heap space cost relations which are used to infer *upper bounds* on the heap space usage of a method.
- In a second step, we refine this cost model to consider the effect of *garbage collection*.
  - ▶ We rely on *Escape Analysis*.
  - ▶ We infer upper bounds on the *active heap space* upon exit from methods (i.e. heap space that might not be garbage collected).

# General Overview



- This is the main aim of our HSA.
- In this paper, we focus on how the cost relations are obtained.
- In the following we show how our HSA works step by step through our running example.

## Java Source Code

```
abstract class List {
    abstract List copy();
}

class Nil extends List {
    List copy() {
        return this;
    }
}

class Cons extends List {
    int elem;
    List next;

    List copy(){
        Cons aux = new Cons();
        aux.elem = this.elem;
        aux.next = this.next.copy();
        return aux;
    }
}
```

## Java Bytecode

```
Cons.copy();
0:  new Cons
3:  dup
4:  invokespecial Cons.<init>
7:  astore_1
8:  aload_1
9:  aload_0
10: getfield Cons.elem
13: putfield Cons.elem
16: aload_1
17: aload_0
18: getfield Cons.next
21: invokevirtual List.copy
24: putfield Cons.next
27: aload_1
28: areturn
```

## Java Source Code

```

abstract class List {
    abstract List copy();
}

class Nil extends List {
    List copy() {
        return this;
    }
}

class Cons extends List {
    int elem;
    List next;

    List copy(){
        Cons aux = new Cons();
        aux.elem = this.elem;
        aux.next = this.next.copy();
        return aux;
    }
}

```

## Java Bytecode

```

Cons.copy();
0:  new Cons
3:  dup
4:  invokespecial Cons.<init>
7:  astore_1
8:  aload_1
9:  aload_0
10: getfield Cons.elem
13: putfield Cons.elem
16: aload_1
17: aload_0
18: getfield Cons.next
21: invokevirtual List.copy
24: putfield Cons.next
27: aload_1
28: areturn

```

## What do we want to achieve?

- Upper bound in closed form:

$$C_{copy}^{Cons}(a) = 8 * a \in O(a)$$

where  $a$  represents the length of the list.

## Java Bytecode

```
Cons.copy();  
0:  new Cons  
3:  dup  
4:  invokespecial Cons.<init>  
7:  astore_1  
8:  aload_1  
9:  aload_0  
10: getfield Cons.elem  
13: putfield Cons.elem  
16: aload_1  
17: aload_0  
18: getfield Cons.next  
21: invokevirtual List.copy  
24: putfield Cons.next  
27: aload_1  
28: areturn  
  
Nil.copy();  
0:  aload_0  
1:  areturn
```

## Building the CFG

- The structure of the JB program is recovered by building a **Control Flow Graph (CFG)**.
- Nodes  $\equiv$  basic blocks which contain sequences of **non-branching bytecode instructions**.
- Edges  $\equiv$  possible flows originated from **branching instructions**.
  - ▶ Conditional jumps
  - ▶ Exceptions
  - ▶ **Virtual method invocations**

## Java Bytecode

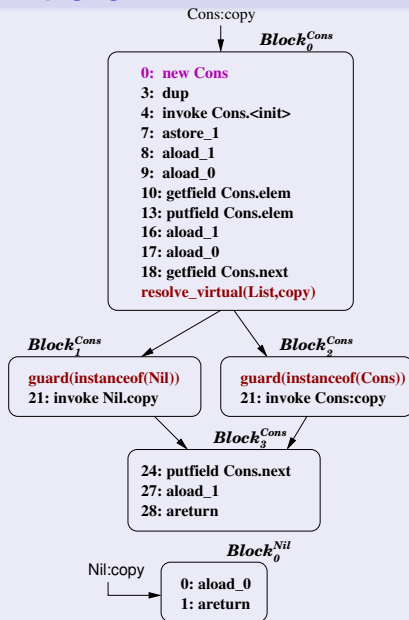
```

Cons.copy();
0:  new Cons
3:  dup
4:  invokespecial Cons.<init>
7:  astore_1
8:  aload_1
9:  aload_0
10: getfield Cons.elem
13: putfield Cons.elem
16: aload_1
17: aload_0
18: getfield Cons.next
21: invokevirtual List.copy
24: putfield Cons.next
27: aload_1
28: areturn

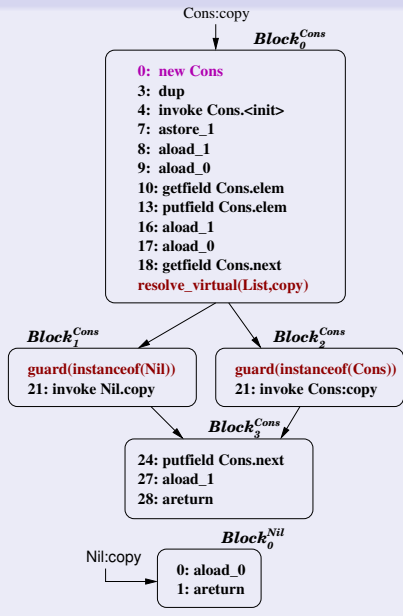
Nil.copy();
0:  aload_0
1:  areturn

```

## The CFG



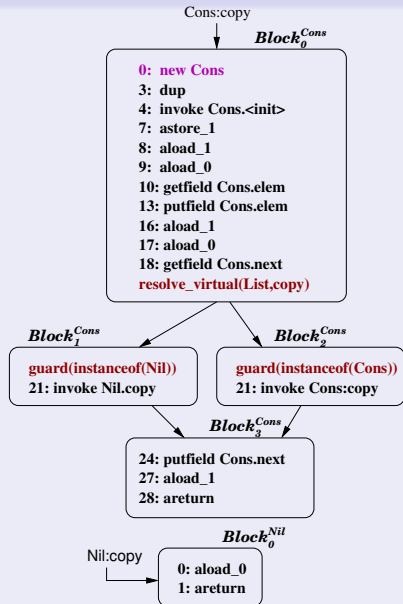
## The CFG



## Building the RR

- Set of *guarded rules* of the form  $\langle head \leftarrow guard, body \rangle$ .
- Every form of iteration is transformed into recursion.
- Stack positions are flattened and visible in the rules together with the local variables.
- Some (unnecessary) variables may be eliminated (Slicing step).

## The CFG



## The Recursive Representation

```

Nil.copy(this, r) ←
  aload(this, s0),
  areturn(s0, r). } BC(Block0Nil)

```

```

Cons.copy(t, r) ←
  new(Cons, s0),
  dup(s0, s1), } BC(Block0Cons)
  ...,
  (Cons.copy1(t, n, ...) ; Cons.copy2(t, n, ...)).

```

```

Cons.copy1(this, next, ...) ←
  guard(instanceof(next, Nil)),
  Nil.copy(next),
  Cons.copy3(this, ...).

```

```

Cons.copy2(this, next, ...) ←
  guard(instanceof(next, Cons)),
  Cons.copy(next),
  Cons.copy3(this, ...).

```

```

Cons.copy3(this, ...) ←
  putfld(C.next, s0, s1),
  aload(aux, s'0),
  areturn(s'0, r). } BC(Block3Cons)

```

## Step III: The Size Analysis

- For each program rule, it infers relations between the variables in the head and the calls occurring in the body.
- Bytecode instructions are abstracted into the linear constraints they impose on their arguments (e.g.  $iadd(s_0, s_1, s'_0) \rightarrow s'_0 = s_0 + s_1$ ).
- Various measures may be considered (e.g. *path length* for pointers).
- Then, a **fix-point** is computed.

## Step III: The Size Analysis

- For each program rule, it infers relations between the variables in the head and the calls occurring in the body.
- Bytecode instructions are abstracted into the linear constraints they impose on their arguments (e.g.  $iadd(s_0, s_1, s'_0) \rightarrow s'_0 = s_0 + s_1$ ).
- Various measures may be considered (e.g. *path length* for pointers).
- Then, a **fix-point** is computed.

### The Recursive Representation + Size Relations

```

Nil.copy(this, r) ←
  BC(Block0Nil).

Cons.copy(this, r) ←
  BC(Block0Cons), (Cons.copy1(this, next, ...); Cons.copy2(this, next, ...)).

Cons.copy1(this, next, ...) ←
  guard(instanceof(next, Nil)), Nil.copy(next), Cons.copy3(this, ...).

Cons.copy2(this, next, ...) ←
  guard(instanceof(next, Cons)), Cons.copy(next), Cons.copy3(this, ...).

Cons.copy3(this, ...) ←
  BC(Block3Cons).
  
```

## Step III: The Size Analysis

- For each program rule, it infers relations between the variables in the head and the calls occurring in the body.
- Bytecode instructions are abstracted into the linear constraints they impose on their arguments (e.g.  $iadd(s_0, s_1, s'_0) \rightarrow s'_0 = s_0 + s_1$ ).
- Various measures may be considered (e.g. *path length* for pointers).
- Then, a **fix-point** is computed.

### The Recursive Representation + Size Relations

`Nil.copy(this, r) ← {this = 1}`  
`BC(Block0Nil).`

`Cons.copy(this, r) ← {next = this - 1, this ≥ 1, next ≥ 0}`  
`BC(Block0Cons), (Cons.copy1(this, next, ...); Cons.copy2(this, next, ...)).`

`Cons.copy1(this, next, ...) ← {this = 1}`  
`guard(instanceof(next, Nil)), Nil.copy(next), Cons.copy3(this, ...).`

`Cons.copy2(this, next, ...) ← {this ≥ 2}`  
`guard(instanceof(next, Cons)), Cons.copy(next), Cons.copy3(this, ...).`

`Cons.copy3(this, ...) ← {}`  
`BC(Block3Cons).`

## The Recursive Representation + Size Relations + Slicing

$\text{Nil.copy}(\text{this}) \leftarrow \{\text{this} = 1\} \text{BC}(\text{Block}_0^{\text{Nil}}).$

$\text{Cons.copy}(\text{this}) \leftarrow \{\text{next} = \text{this} - 1, \text{this} \geq 1, \text{next} \geq 0\}$   
 $\text{BC}(\text{Block}_0^{\text{Cons}}), (\text{Cons.copy}_1(\text{this}, \text{next}); \text{Cons.copy}_2(\text{this}, \text{next})).$

$\text{Cons.copy}_1(\text{this}, \text{next}) \leftarrow \{\text{this} = 1\}$   
 $\text{guard}(\text{instanceof}(\text{next}, \text{Nil})), \text{Nil.copy}(\text{next}), \text{Cons.copy}_3(\text{this}).$

$\text{Cons.copy}_2(\text{this}, \text{next}) \leftarrow \{\text{this} \geq 2\}$   
 $\text{guard}(\text{instanceof}(\text{next}, \text{Cons})), \text{Cons.copy}(\text{next}), \text{Cons.copy}_3(\text{this}).$

$\text{Cons.copy}_3(\text{this}) \leftarrow \{\} \text{BC}(\text{Block}_3^{\text{Cons}}).$

## Cost Model for Heap Space

$$\mathcal{M}_{\text{heap}}(\text{bc}) = \begin{cases} \text{size}(\text{Class}) & \text{if } \text{bc} = \text{new}(\text{Class}, -) \\ S_{\text{PrimType}} * L & \text{if } \text{bc} = \text{newarray}(\text{PrimType}, L, -) \\ S_{\text{ref}} * L & \text{if } \text{bc} = \text{anewarray}(\text{Class}, L, -) \\ 0 & \text{otherwise} \end{cases}$$

## The Recursive Representation + Size Relations + Slicing

$$\text{Nil.copy}(\text{this}) \leftarrow \{\text{this} = 1\} \text{BC}(\text{Block}_0^{\text{Nil}}).$$

$$\text{Cons.copy}(\text{this}) \leftarrow \{\text{next} = \text{this} - 1, \text{this} \geq 1, \text{next} \geq 0\} \\ \text{BC}(\text{Block}_0^{\text{Cons}}), (\text{Cons.copy}_1(\text{this}, \text{next}); \text{Cons.copy}_2(\text{this}, \text{next})).$$

$$\text{Cons.copy}_1(\text{this}, \text{next}) \leftarrow \{\text{this} = 1\} \\ \text{guard}(\text{instanceof}(\text{next}, \text{Nil})), \text{Nil.copy}(\text{next}), \text{Cons.copy}_3(\text{this}).$$

$$\text{Cons.copy}_2(\text{this}, \text{next}) \leftarrow \{\text{this} \geq 2\} \\ \text{guard}(\text{instanceof}(\text{next}, \text{Cons})), \text{Cons.copy}(\text{next}), \text{Cons.copy}_3(\text{this}).$$

$$\text{Cons.copy}_3(\text{this}) \leftarrow \{\} \text{BC}(\text{Block}_3^{\text{Cons}}).$$

## Cost Relations

Heap Space Cost Equations		Size relations
$C_{copy}^{\text{Nil}}(a)$	$= 0$	$\{a=1\}$
$C_{copy}^{\text{Cons}}(a)$	$= C_0(a)$	
$C_0(a)$	$= \text{size}(\text{Cons}) + CC_0(a, b)$	$\{a \geq 1, b \geq 0, a = b + 1\}$
$CC_0(a, b)$	$= \begin{cases} C_1(a, b) & \hat{b} \in \text{Nil} \\ C_2(a, b) & \hat{b} \in \text{Cons} \end{cases}$	
$C_1(a, b)$	$= C_{copy}^{\text{Nil}}(b) + C_3(a)$	$\{a=1\}$
$C_2(a, b)$	$= C_{copy}^{\text{Cons}}(b) + C_3(a)$	$\{a \geq 2\}$
$C_3(a)$	$= 0$	

## The Recursive Representation + Size Relations + Slicing

$$\text{Nil.copy}(\text{this}) \leftarrow \{\text{this} = 1\} \text{BC}(\text{Block}_0^{\text{Nil}}).$$

$$\text{Cons.copy}(\text{this}) \leftarrow \{\text{next} = \text{this} - 1, \text{this} \geq 1, \text{next} \geq 0\} \\ \text{BC}(\text{Block}_0^{\text{Cons}}), (\text{Cons.copy}_1(\text{this}, \text{next}); \text{Cons.copy}_2(\text{this}, \text{next})).$$

$$\text{Cons.copy}_1(\text{this}, \text{next}) \leftarrow \{\text{this} = 1\} \\ \text{guard}(\text{instanceof}(\text{next}, \text{Nil})), \text{Nil.copy}(\text{next}), \text{Cons.copy}_3(\text{this}).$$

$$\text{Cons.copy}_2(\text{this}, \text{next}) \leftarrow \{\text{this} \geq 2\} \\ \text{guard}(\text{instanceof}(\text{next}, \text{Cons})), \text{Cons.copy}(\text{next}), \text{Cons.copy}_3(\text{this}).$$

$$\text{Cons.copy}_3(\text{this}) \leftarrow \{\} \text{BC}(\text{Block}_3^{\text{Cons}}).$$

## Simplified Cost Relations

Equation	Size relations
$C_{copy}^{\text{Nil}}(a) = 0$	$\{a=1\}$
$C_{copy}^{\text{Cons}}(a) = 8$	$\{a=2\}$
$C_{copy}^{\text{Cons}}(a) = 8 + C_{copy}^{\text{Cons}}(b)$	$\{a \geq 3, b \geq 1, a = b + 1\}$

## Closed Form

$$C_{copy}^{\text{Cons}}(a) = 8 * (a - 1) \in O(a)$$

# Active Heap Space with Garbage Collection

- We propose a refinement of our cost model to consider the effect of *garbage collection*.
- We will have *safe annotations* for the heap space that will be garbage collected upon exit.
- This is done by relying on *Escape Analysis*:
  - ▶ It aims to determine which objects will never outlive the method in which they are created (*local* objects).
  - ▶ Escape Analysis will annotate which allocation instructions are *local*.
  - ▶ Then we define a refined cost model with the corresponding annotations.
- The annotated cost relations are used to infer upper bounds on the *active heap space* upon exit from a method.

# Example

## Java source code

```

//class List
abstract List map(Func o);

//class Nil
List map(Func o)
  return this;
}

List map(Func o) { //class Cons
  List tail = this.next.map(o);
  Cons head = new Cons();
  head.next = tail;
  head.elem =
    o.f(new Integer(this.elem));
  return head;
}

```

## Annotated Cost Relations

$$\begin{array}{l|l}
 C_{map}^{Nil}(a) = 0 & \{a=1\} \\
 C_{map}^{Cons}(a) = gc(4) + ngc(8) & \{a=2\} \\
 C_{map}^{Cons}(a) = gc(4) + ngc(8) + C_{map}^{Cons}(b) & \{a \geq 3, b \geq 1, a = b + 1\}
 \end{array}$$

# Example

## Java source code

```
//class List
abstract List map(Func o);

//class Nil
List map(Func o)
  return this;
}

List map(Func o) { //class Cons
  List tail = this.next.map(o);
  Cons head = new Cons();
  head.next = tail;
  head.elem =
    o.f(new Integer(this.elem));
  return head;
}
```

## Annotated Cost Relations

$$\begin{array}{l|l}
 C_{map}^{Nil}(a) = 0 & \{a=1\} \\
 C_{map}^{Cons}(a) = gc(4) + ngc(8) & \{a=2\} \\
 C_{map}^{Cons}(a) = gc(4) + ngc(8) + C_{map}^{Cons}(b) & \{a \geq 3, b \geq 1, a = b + 1\}
 \end{array}$$

• If  $\forall H, gc(H) = 0$  and  $ngc(H) = H$  then  $C_{map}^{Cons}(a) = 8 * (a - 1)$

• If  $\forall H, gc(H) = H$  and  $ngc(H) = H$  then  $C_{map}^{Cons}(a) = 12 * (a - 1)$

# Experiments

- We have implemented a prototype analyzer.
- We still have not incorporated an escape analysis.
- We support the full instructions set of sequential JB.
- Plenty of room for optimization, mainly in the size analysis phase.

Benchmark	Size	RR	Size An.	Cost	Total	Complexity	
ListInt	0.86	24	53	7	83	$O(n)$	$n \equiv$ list length
Results	1.31	83	275	15	374	$O(1)$	–
BSTInt	0.48	37	113	5	156	$O(2^n)$	$n \equiv$ tree depth
List	1.79	71	207	16	293	$O(n)$	$n \equiv$ list length
Queue	1.93	219	570	24	813	$O(n)$	$n \equiv$ queue length
Stack	1.38	89	643	17	749	$O(n)$	$n \equiv$ stack length
BST	1.43	97	238	14	349	$O(2^n)$	$n \equiv$ tree depth
Scoreboard	0.65	280	1539	12	1830	$O(a^2 * b)$	$\{a, b\} \equiv$ input args.
MultiBST	2.35	166	510	34	709	$O(n * 2^n)$	$n \equiv$ tree depth

## Java source code

```

class Score{
    private int gt1, gt2;
    public Score() {
        gt1 = 0;
        gt2 = 0;
    }
}
class Scoreboard{
    private Score[][][] scores;

    public Scoreboard(int a,int b) {
        scores = new Score[a][][];
        for (int i = 1;i <= a;i++) {
            scores[i-1] = new Score[i][];
            for (int j = 0;j < (i-1);j++) {
                scores[i-1][j] = new Score[b];
                for (int k = 0;k < b;k++)
                    scores[i-1][j][k]=new Score();
            }
        }
    }
}

```



## Java source code

```

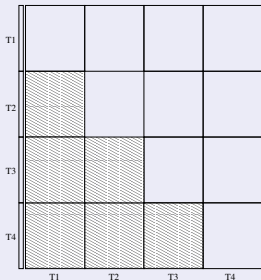
class Score{
    private int gt1, gt2;
    public Score() {
        gt1 = 0;
        gt2 = 0;
    }
class Scoreboard{
    private Score[][][] scores;

```

```

    public Scoreboard(int a,int b) {
        scores = new Score[a][][];
        for (int i = 1;i <= a;i++) {
            scores[i-1] = new Score[i][];
            for (int j = 0;j < (i-1);j++) {
                scores[i-1][j] = new Score[b];
                for (int k = 0;k < b;k++)
                    scores[i-1][j][k]=new Score();
            }
        }
    }
}

```



## Java source code

```

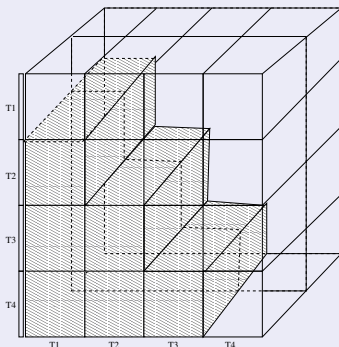
class Score{
    private int gt1, gt2;
    public Score() {
        gt1 = 0;
        gt2 = 0;
    }
class Scoreboard{
    private Score[][][] scores;

```

```

    public Scoreboard(int a,int b) {
        scores = new Score[a][][];
        for (int i = 1;i <= a;i++) {
            scores[i-1] = new Score[i][];
            for (int j = 0;j < (i-1);j++) {
                scores[i-1][j] = new Score[b];
                for (int k = 0;k < b;k++)
                    scores[i-1][j][k]=new Score();
            }
        }
    }
}

```



## Java source code

```

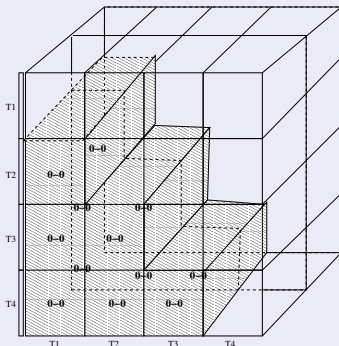
class Score{
    private int gt1, gt2;
    public Score() {
        gt1 = 0;
        gt2 = 0;
    }
class Scoreboard{
    private Score[][][] scores;

```

```

    public Scoreboard(int a,int b) {
        scores = new Score[a][][];
        for (int i = 1;i <= a;i++) {
            scores[i-1] = new Score[i][];
            for (int j = 0;j < (i-1);j++) {
                scores[i-1][j] = new Score[b];
                for (int k = 0;k < b;k++)
                    scores[i-1][j][k]=new Score();
            }
        }
    }
}

```



## Java source code

```
class Score{
  private int gt1, gt2;
  public Score() {
    gt1 = 0;
    gt2 = 0;
  }
class Scoreboard{
  private Score[][][] scores;
```

```
public Scoreboard(int a,int b) {
  scores = new Score[a][][];
  for (int i = 1;i <= a;i++) {
    scores[i-1] = new Score[i][];
    for (int j = 0;j < (i-1);j++) {
      scores[i-1][j] = new Score[b];
      for (int k = 0;k < b;k++)
        scores[i-1][j][k]=new Score();
    }
  }
}
```

## Heap Space Cost Equations

$C_{\langle \text{init} \rangle}(a, b)$	$=$	$\overbrace{a * S_{ref}}^{\text{1st dim}}$	$+ C_1(a, b, 1)$	
$C_1(a, b, i)$	$=$	$\overbrace{i * S_{ref}}^{\text{2nd dim}}$	$+ C_2(b, i, 0) + C_1(a, b, d)$	$\{i \leq a, d = i + 1\}$
$C_1(a, b, i)$	$=$	$0$		$\{i > a\}$
$C_2(b, i, j)$	$=$	$\overbrace{b * S_{ref}}^{\text{3rd dim}}$	$+ C_3(b, 0) + C_2(b, i, d)$	$\{j < (i - 1), d = j + 1\}$
$C_2(b, i, j)$	$=$	$0$		$\{j \geq (i - 1)\}$
$C_3(b, k)$	$=$	$\overbrace{2 * S_{int}}^{\text{size(Score)}}$	$+ C_3(b, c)$	$\{k < b, c = k + 1\}$
$C_3(b, k)$	$=$	$0$		$\{k \geq b\}$

## Java source code

```

class Score{
    private int gt1, gt2;
    public Score() {
        gt1 = 0;
        gt2 = 0;
    }
}
class Scoreboard{
    private Score[][][] scores;

    public Scoreboard(int a,int b) {
        scores = new Score[a][][];
        for (int i = 1;i <= a;i++) {
            scores[i-1] = new Score[i][];
            for (int j = 0;j < (i-1);j++) {
                scores[i-1][j] = new Score[b];
                for (int k = 0;k < b;k++)
                    scores[i-1][j][k]=new Score();
            }
        }
    }
}

```

## Upper Bound

$$C_{\langle \text{init} \rangle}(a, b) \leq (((2 * S_{\text{int}} * b) + b * S_{\text{ref}}) * a + a * S_{\text{ref}}) * a + a * S_{\text{ref}} \in O(b * a^2).$$

## Conclusions

- We have presented an automatic analysis of heap usage for JB.
- It generates at compile-time **cost relations** which define the heap space consumption of a program as a function of its input data size.
- Our analysis is able to infer **non-trivial bounds for complex data structures** (including polynomial and exponential complexities).

## Future Work

- On the practical side:
  - ▶ Incorporate **escape analysis** as outlined in the paper.
  - ▶ Scalability is a question of **performance vs. precision trade-off** and depends on the underlying abstract domain used by the size analysis.
- On the theoretical side:
  - ▶ Adapt our analysis to infer upper bounds on the heap usage at given program points in presence of garbage collection.
  - ▶ Analysis for inferring upper bounds on the call stack usage.