Heap Space Analysis for Java Bytecode

Elvira Albert\textsuperscript{1}, Samir Genaim\textsuperscript{2},
Miguel Gómez-Zamalloa\textsuperscript{1}

(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

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).
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: invokevirtual List.copy
13:  putfield Cons.next
16:  astore_1
17:  aload_0
18:  getfield Cons.next
21:  invokevirtual List.copy
24:  putfield Cons.next
27:  astore_1
28:  areturn
Running Example

Source code, bytecode and upper bound

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() {  // Upper bound in closed form: 
        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:
  \[ C_{\text{Cons}}^{\text{copy}}(a) = 8 \times a \in O(a) \] 

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** ≡ basic blocks which contain sequences of **non-branching bytecode instructions**.
- **Edges** ≡ 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

Cons:copy

Block^Cons_0

0:  new Cons
3:  dup
4:  invoke 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
resolve_virtual(List,copy)

Block^Cons_1

21: invoke Cons:copy

Block^Cons_2

21: invoke Nil.copy

guard(instanceof(Nil))

24: putfield Cons.next
27: aload_1
28: areturn

Block^Cons_3

guard(instanceof(Cons))

Nil:copy

0:  aload_0
1:  areturn

Block^Nil_0
### Building the RR

- Set of *guarded rules* of the form \( \langle \text{head} \leftarrow \text{guard}, \text{body} \rangle \).
- 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).
Running Example

Step II: From CFG to Recursive Representation

The Recursive Representation

Nil.copy(this, r) ←
aload(this, s_0),
areturn(s_0, r).

Cons.copy(t, r) ←
new(Cons, s_0),
dup(s_0, s_1)),
BC(Block^{Cons}_0)
...
(Cons.copy_1(t, n, ...); Cons.copy_2(t, n, ...)).

Cons.copy_1(this, next, ...) ←
resolve_virtual(List.copy)
guard(instanceof(next, Nil)),
Nil.copy(next),
Cons.copy_3(this, ...).

Cons.copy_2(this, next, ...) ←
resolve_virtual(List.copy)
guard(instanceof(next, Cons)),
Cons.copy(next),
Cons.copy_3(this, ...).

Cons.copy_3(this, ...) ←
resolve_virtual(List.copy)
putfield(Cons.next, s_0, s_1),
aload(aux, s'_0)),
BC(Block^{Cons}_3)
areturn(s'_0, r).
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. \( \text{iadd}(s_0, s_1, s'_0) \rightarrow s'_0 = s_0 + s_1 \)).
- 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. \( \text{iadd}(s_0, s_1, s'_0) \rightarrow s'_0 = s_0 + s_1 \)).
- Various measures may be considered (e.g. \( \text{path length} \) for pointers).
- Then, a fix-point is computed.

The Recursive Representation + Size Relations

\[
\begin{align*}
\text{Nil.copy}(\text{this}, r) & \leftarrow \\
& \text{BC}(\text{Block}_{\text{Nil}}^0).
\end{align*}
\]

\[
\begin{align*}
\text{Cons.copy}(\text{this}, r) & \leftarrow \\
& \text{BC}(\text{Block}_{\text{Cons}}^0), (\text{Cons.copy}_1(\text{this}, \text{next}, \ldots); \text{Cons.copy}_2(\text{this}, \text{next}, \ldots)).
\end{align*}
\]

\[
\begin{align*}
\text{Cons.copy}_1(\text{this}, \text{next}, \ldots) & \leftarrow \\
& \text{guard}(\text{instanceof}(\text{next, Nil})), \text{Nil.copy}(\text{next}), \text{Cons.copy}_3(\text{this}, \ldots).
\end{align*}
\]

\[
\begin{align*}
\text{Cons.copy}_2(\text{this}, \text{next}, \ldots) & \leftarrow \\
& \text{guard}(\text{instanceof}(\text{next, Cons})), \text{Cons.copy}(\text{next}), \text{Cons.copy}_3(\text{this}, \ldots).
\end{align*}
\]

\[
\begin{align*}
\text{Cons.copy}_3(\text{this}, \ldots) & \leftarrow \\
& \text{BC}(\text{Block}_{\text{Cons}}^3).
\end{align*}
\]
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. \( \text{iadd}(s_0, s_1, s'_0) \rightarrow s'_0 = s_0 + s_1 \)).
- Various measures may be considered (e.g. path length for pointers).
- Then, a fix-point is computed.

The Recursive Representation + Size Relations

\[
\begin{align*}
\text{Nil.copy}(\text{this}, r) & \leftarrow \{ \text{this} = 1 \} \\
& \quad \text{BC}(\text{Block}_{\text{Nil}}^0). \\
\text{Cons.copy}(\text{this}, r) & \leftarrow \{ \text{next} = \text{this} - 1, \text{this} \geq 1, \text{next} \geq 0 \} \\
& \quad \text{BC}(\text{Block}_{\text{Cons}}^0), (\text{Cons.copy}_1(\text{this}, \text{next}, \ldots); \text{Cons.copy}_2(\text{this}, \text{next}, \ldots)). \\
\text{Cons.copy}_1(\text{this}, \text{next}, \ldots) & \leftarrow \{ \text{this} = 1 \} \\
& \quad \text{guard}(\text{instanceof}(\text{next}, \text{Nil})), \text{Nil.copy}(\text{next}), \text{Cons.copy}_3(\text{this}, \ldots). \\
\text{Cons.copy}_2(\text{this}, \text{next}, \ldots) & \leftarrow \{ \text{this} \geq 2 \} \\
& \quad \text{guard}(\text{instanceof}(\text{next}, \text{Cons})), \text{Cons.copy}(\text{next}), \text{Cons.copy}_3(\text{this}, \ldots). \\
\text{Cons.copy}_3(\text{this}, \ldots) & \leftarrow \{ \} \\
& \quad \text{BC}(\text{Block}_{\text{Cons}}^3). 
\end{align*}
\]
**The Recursive Representation + Size Relations + Slicing**

\[
\begin{align*}
\text{Nil.copy}(\text{this}) & \leftarrow \{\text{this} = 1\} \quad \text{BC}(\text{Block}_{\text{Nil}}^0).
\\
\text{Cons.copy}(\text{this}) & \leftarrow \{\text{next} = \text{this} - 1, \text{this} \geq 1, \text{next} \geq 0\}
\\ & \quad \text{BC}(\text{Block}_{\text{Cons}}^0), \ (\text{Cons.copy}_1(\text{this}, \text{next})); \ (\text{Cons.copy}_2(\text{this}, \text{next})).
\\
\text{Cons.copy}_1(\text{this}, \text{next}) & \leftarrow \{\text{this} = 1\}
\\ & \quad \text{guard}(\text{instanceof}(\text{next}, \text{Nil})), \ \text{Nil.copy}(\text{next}), \ \text{Cons.copy}_3(\text{this}).
\\
\text{Cons.copy}_2(\text{this}, \text{next}) & \leftarrow \{\text{this} \geq 2\}
\\ & \quad \text{guard}(\text{instanceof}(\text{next}, \text{Cons})), \ \text{Cons.copy}(\text{next}), \ \text{Cons.copy}_3(\text{this}).
\\
\text{Cons.copy}_3(\text{this}) & \leftarrow \{} \quad \text{BC}(\text{Block}_{\text{Cons}}^3).
\end{align*}
\]

**Cost Model for Heap Space**

\[
M_{\text{heap}}(\text{bc}) = \begin{cases} 
\text{size}(\text{Class}) & \text{if } \text{bc} = \text{new}(\text{Class}, \_ ) \\
S_{\text{PrimType}} \ast \text{L} & \text{if } \text{bc} = \text{newarray}(\text{PrimType}, \text{L}, \_ ) \\
S_{\text{ref}} \ast \text{L} & \text{if } \text{bc} = \text{anewarray}(\text{Class}, \text{L}, \_ ) \\
0 & \text{otherwise}
\end{cases}
\]
The Recursive Representation + Size Relations + Slicing

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

Cons.copy(this) ← \{next = this – 1, this \geq 1, next \geq 0\}
BC(Block^\text{Cons}_0), (Cons.copy_1(this,next); Cons.copy_2(this,next)).

Cons.copy_1(this,next) ← \{this = 1\}
guard\text{instanceof}(next,Nil), Nil.copy(next), Cons.copy_3(this).

Cons.copy_2(this,next) ← \{this \geq 2\}
guard\text{instanceof}(next,Cons), Cons.copy(next), Cons.copy_3(this).

Cons.copy_3(this) ← {} BC(Block^\text{Cons}_3).

Cost Relations

<table>
<thead>
<tr>
<th>Heap Space Cost Equations</th>
<th>Size relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{C}^\text{Nil}_{\text{copy}}(a) ) = 0</td>
<td>{a=1}</td>
</tr>
<tr>
<td>( \text{C}^\text{Cons}_{\text{copy}}(a) ) = \text{C}_0(a)</td>
<td></td>
</tr>
<tr>
<td>( \text{C}_0(a) ) = size(Cons) + \text{CC}_0(a,b)</td>
<td>{a \geq 1, b \geq 0, a=b+1}</td>
</tr>
<tr>
<td>( \text{CC}_0(a,b) ) = { \begin{array}{ll} \text{C}_1(a,b) &amp; \hat{b} \in \text{Nil} \ \text{C}_2(a,b) &amp; \hat{b} \in \text{Cons} \end{array} )</td>
<td></td>
</tr>
<tr>
<td>( \text{C}<em>1(a,b) ) = \text{C}^\text{Nil}</em>{\text{copy}}(b) + \text{C}_3(a)</td>
<td>{a=1}</td>
</tr>
<tr>
<td>( \text{C}<em>2(a,b) ) = \text{C}^\text{Cons}</em>{\text{copy}}(b) + \text{C}_3(a)</td>
<td>{a \geq 2}</td>
</tr>
<tr>
<td>( \text{C}_3(a) ) = 0</td>
<td></td>
</tr>
</tbody>
</table>
Running Example

Step IV: From RR + Size rels. to Heap Space Cost Equations

The Recursive Representation + Size Relations + Slicing

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

\text{Cons.copy}(\text{this}) \leftarrow \{ \text{next} = \text{this} - 1, \text{this} \geq 1, \text{next} \geq 0 \}
\quad \text{BC(} \text{Block}^\text{Cons}_0 \text{)}, (\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 \}
\quad \text{guard(} \text{instanceof}(\text{next}, \text{Nil}) \text{)}, \text{Nil.copy}(\text{next}), \text{Cons.copy}_3(\text{this}).

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

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

Simplified Cost Relations

<table>
<thead>
<tr>
<th>Equation</th>
<th>Size relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C^{\text{Nil}}_{\text{copy}}(a)$</td>
<td>$a=1$</td>
</tr>
<tr>
<td>$C^{\text{Cons}}_{\text{copy}}(a)$</td>
<td>$a=2$</td>
</tr>
<tr>
<td>$C^{\text{Cons}}_{\text{copy}}(a)$</td>
<td>$8 + C^{\text{Cons}}_{\text{copy}}(b)$</td>
</tr>
</tbody>
</table>

Closed Form

$$C^{\text{Cons}}_{\text{copy}}(a) = 8 \times (a - 1) \in O(a)$$
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;
}

//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

$C_{map}^{Nil}(a) = 0$

$C_{map}^{Cons}(a) = gc(4) + ngc(8)$

$C_{map}^{Cons}(a) = gc(4) + ngc(8) + C_{map}^{Cons}(b)$

\{a=1\}

\{a=2\}

\{a \geq 3, b \geq 1, a = b+1\}
Example

Java source code

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

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

//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

\[ C_{\text{map}}^{\text{Nil}}(a) = 0 \quad \{a=1\} \]
\[ C_{\text{map}}^{\text{Cons}}(a) = gc(4) + ngc(8) \quad \{a=2\} \]
\[ C_{\text{map}}^{\text{Cons}}(a) = gc(4) + ngc(8) + C_{\text{map}}^{\text{Cons}}(b) \quad \{a \geq 3, b \geq 1, a = b+1\} \]

- If \( \forall H, gc(H) = 0 \) and \( ngc(H) = H \) then \[ C_{\text{map}}^{\text{Cons}}(a) = 8 * (a - 1) \]
- If \( \forall H, gc(H) = H \) and \( ngc(H) = H \) then \[ C_{\text{map}}^{\text{Cons}}(a) = 12 * (a - 1) \]
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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size</th>
<th>RR</th>
<th>Size An.</th>
<th>Cost</th>
<th>Total</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListInt</td>
<td>0.86</td>
<td>24</td>
<td>53</td>
<td>7</td>
<td>83</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Results</td>
<td>1.31</td>
<td>83</td>
<td>275</td>
<td>15</td>
<td>374</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>BSTInt</td>
<td>0.48</td>
<td>37</td>
<td>113</td>
<td>5</td>
<td>156</td>
<td>$O(2^n)$</td>
</tr>
<tr>
<td>List</td>
<td>1.79</td>
<td>71</td>
<td>207</td>
<td>16</td>
<td>293</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Queue</td>
<td>1.93</td>
<td>219</td>
<td>570</td>
<td>24</td>
<td>813</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Stack</td>
<td>1.38</td>
<td>89</td>
<td>643</td>
<td>17</td>
<td>749</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>BST</td>
<td>1.43</td>
<td>97</td>
<td>238</td>
<td>14</td>
<td>349</td>
<td>$O(2^n)$</td>
</tr>
<tr>
<td>Scoreboard</td>
<td>0.65</td>
<td>280</td>
<td>1539</td>
<td>12</td>
<td>1830</td>
<td>$O(a^2 * b)$</td>
</tr>
<tr>
<td>MultiBST</td>
<td>2.35</td>
<td>166</td>
<td>510</td>
<td>34</td>
<td>709</td>
<td>$O(n * 2^n)$</td>
</tr>
</tbody>
</table>
Another example: Dealing with Complex Multi-dimen. Arrays

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

```java
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][][[]; // Start of for loop
        for (int i = 1;i <= a;i++) {
            scores[i-1] = new Score[i][]; // Start of for loop for j
            for (int j = 0;j < (i-1);j++) {
                scores[i-1][j] = new Score[b]; // Start of for loop for k
                for (int k = 0;k < b;k++)
                    scores[i-1][j][k]=new Score();  // End of for loop for k
            }
        }
    }
}
```

```
      T1 | T2 | T3 | T4
-----|----|----|----
 T1  |    |    |    |
 T2  |    |    |    |
 T3  |    |    |    |
 T4  |    |    |    |
```

Another example: Dealing with Complex Multi-dimen. Arrays

Java source code

```java
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][][[]; // Start of for loop
        for (int i = 1;i <= a;i++) {
            scores[i-1] = new Score[i][]; // Start of for loop for j
            for (int j = 0;j < (i-1);j++) {
                scores[i-1][j] = new Score[b]; // Start of for loop for k
                for (int k = 0;k < b;k++)
                    scores[i-1][j][k]=new Score();  // End of for loop for k
            }
        }
    }
}
```

```
      T1 | T2 | T3 | T4
-----|----|----|----
 T1  |    |    |    |
 T2  |    |    |    |
 T3  |    |    |    |
 T4  |    |    |    |
```

M. Gómez-Zamalloa (UCM)  Heap Space Analysis for Java Bytecode  Montreal, October 22, 2007  13 / 14
Another example: Dealing with Complex Multi-dimen. Arrays

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

```java
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];
                scores[i-1][j][k] = new Score();
            }
        }
    }
}
```

Heap Space Cost Equations

<table>
<thead>
<tr>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C_{\text{init}}(a, b))</td>
<td>1st dim (a \times S_{\text{ref}} + C_1(a, b, 1)) ({i \leq a, d = i+1})</td>
</tr>
<tr>
<td>(C_1(a, b, i))</td>
<td>2nd dim (i \times S_{\text{ref}} + C_2(b, i, 0) + C_1(a, b, d)) ({i &gt; a})</td>
</tr>
<tr>
<td>(C_1(a, b, i))</td>
<td>3rd dim 0 ({j &lt; (i-1), d = j+1})</td>
</tr>
<tr>
<td>(C_2(b, i, j))</td>
<td>(b \times S_{\text{ref}} + C_3(b, 0) + C_2(b, i, d)) ({j \geq (i-1)})</td>
</tr>
<tr>
<td>(C_3(b, k))</td>
<td>(2 \times S_{\text{int}} + C_3(b, c)) ({k &lt; b, c = k+1})</td>
</tr>
<tr>
<td>(C_3(b, k))</td>
<td>0 ({k \geq b})</td>
</tr>
</tbody>
</table>
Another example: Dealing with Complex Multi-dimen. Arrays

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

\[ C_{\text{init}}(a, b) \leq ((2 \times S_{\text{int}} \times b) + b \times S_{\text{ref}}) \times a + a \times S_{\text{ref}} \times a + a \times S_{\text{ref}} \in O(b \times a^2). \]
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.