On transforming Java-like programs into memory-predictable code

Diego Garbervetsky

Joint work with Sergio Yovine, Víctor Braberman, Martín Rouaux and Alejandro Taboada
RTSJ (Real time specification for Java)

- Scoped memory management
  - Dynamic memory organized in **regions** associated to particular scopes
    - Methods, threads, etc.

- Advantages
  - Better time predictability
    - (compared with non RTGCs)
  - More controlled object allocation and deallocation
    - Useful for memory consumption predictability
Drawback: Developer responsible of deciding in which region will place each object
  - Respect scoping rules
  - Difficult, error prone.

Some candidate solutions
  - Enforce scoping rules
    - Type system, static or dynamic checking
  - Automatic region synthesis
    - Approximating object’s lifetime by profiling
    - Escape analysis

Still a problem....
  - RTSJ requires developers to specify the **region size**
Our Goal:
- Inferring quantitative memory requirements

This work:
- The integration of techniques to:
  - Help identifying memory regions for a scoped memory manager
    - Optionally transform Java code to scoped-based memory
  - Compute region sizes
  - Compute application’s dynamic memory requirements
Overview

Java Application (bytecode or src) → Dynamic Memory utilization analysis → Region size inference → Memory region inference → Region-based Java code generation → Region based Java code

Local Invariant Generation

Local Invariants → Method's dynamic memory utilization

Method's region size certificates → Memory requirements inference

Region-based Java code generation → Region based Java code

Escape Analysis

Memory region inference

Region Based API

Method Consumption
A (simple) software component
Computes some aggregate information
- Out of a list of temperatures informed by an array of sensors.

<table>
<thead>
<tr>
<th>Time</th>
<th>10.00</th>
<th>10.01</th>
<th>...</th>
<th>23.59</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>10</td>
<td>20</td>
<td>...</td>
<td>30</td>
</tr>
<tr>
<td>S2</td>
<td>-5</td>
<td>10</td>
<td>...</td>
<td>10</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A (simple) software component
Computes some aggregate information
- Out of a list of temperatures informed by an array of sensors.
Running Example

- A (simple) software component
- Computes some aggregate information
  - Out of a list of temperatures informed by an array of sensors.

```
<table>
<thead>
<tr>
<th>Time</th>
<th>10.00</th>
<th>10.01</th>
<th>...</th>
<th>23.59</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>10</td>
<td>20</td>
<td>...</td>
<td>30</td>
</tr>
<tr>
<td>S2</td>
<td>-5</td>
<td>10</td>
<td>...</td>
<td>10</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Diagram:
- `main`
- `Process(sl)`
- `AVG-Sensor(sl)`
- `AVG-Time(SL,t)`
- `AVG-List(l)`
- `Other components`
process(SensorList sl, int t) {
    avg_Sensor(sl);
    avg_Time(sl, t);
    ...
}

List avg_Sensor(SensorList sl) {
    List LR = new List();
    Iterator itSL = sl.iterator();
    for (; itSL.hasNext();)
    {
        S = (Sensor) itSL.next();
        float avg = avg_List(S.records);
        LR.add(avg);
    }
    return LR;
}

List avg_Time(SensorList sl, int t) {
    List LR = new List();
    for (int k=1; k<=t; k++) {
        List L = new List();
        Iterator itSL = sl.iterator();
        for (; itSL.hasNext();)
        {
            S = (Sensor) itSL.next();
            L.add(s.records[k]);
        }
    }
    return LR;
}

float avg_List(List L) {
    int c = 0, s = 0;
    Iterator itL = L.iterator();
    for (; itL.hasNext();)
    {        val = (Value) itL.next();
        CO cObj = new CO(val);
        if (cObj.validate())
        {
            s+=val.value();
            c++;
        }
    }
    if (c>0) return s/c
    else return -1;
}
```java
float avg_List(List L) {
    int c = 0, s = 0;
    Iterator itL = L.iterator();
    for (; itL.hasNext();)
        val = (Value) itL.next();
        CO cObj = new CO(val);
        if (cObj.validate()) {
            s += val.value();
            c++;
        }
    if (c > 0) return s / c
    else return -1;
}
```

- How many objects are requested by `avg_List(L)`?
  - `1 + L.size()`

- How much is requested?
  - `T(Iterator) + L.size() * T(CO)`
How many objects are requested by `process(SL, t)`?

- \(2Nt + Nd + 2N + 4T + 3\) where \(N = SL.size() \&\& d = S_0.records.size()\).
- Not trivial!
Key idea: Counting visits to statements that allocate memory

\[
\begin{align*}
\text{for } (i=0; i<n; i++) \\
\quad \text{for } (j=0; j<i; j++) \\
\quad \quad \text{• new } C() \\
\end{align*}
\]

\(\phi \equiv \{0 \leq i < n, 0 \leq j < i\} \): a set of constraints describing a iteration space

- Dynamic Memory request \(\approx\) number of visits to \texttt{new} statements
- \(\approx\) number of possible variable assignments at its control location
- \(\approx\) number of integer solutions of a predicate constraining variable assignments at its control location (i.e. an \textit{invariant})

For linear invariants, \# of integer solutions = \# of integer points = \textit{Ehrhart polynomial}

\[
\text{size}(C) \times \left( \frac{1}{2} n^2 + \frac{1}{2} n \right)
\]
Computing totAlloc

- The creation site identified by `process.avg_Sensor.avg_List.cObj`.
- `Inv:= 1 <= j <= N \&\& L.size()= d \&\& 1<= h <= N` (N=sl.size())
- `Count(Inv,N,t) = Nd`
- `S(Inv,N,t) = Nd * Size(CO)`

1- Identify allocation sites
The creation site identified by `process.avg_Sensor.avg_List.cObj`:

- Inv: $1 \leq j \leq N \land L.size() = d \land 1 \leq h \leq N$ ($N = sl.size()$)
- Count(Inv, N, t) = $Nd$
- $S(Inv, N, t) = Nd \times \text{Size(CO)}$

2- Compute a global invariant for the allocation site
Computing `totAlloc`

- The creation site identified by `process.avg_Sensor.avg_List.cObj`:
  - Inv := \(1 \leq j \leq N \land \text{L.size()} = d \land 1 \leq h \leq N\) (N=sl.size())
  - Count(Inv,N,t) = Nd
  - S(Inv,N,t) = Nd * Size(CO)

- Count # of solutions of the global invariant
Computing totAlloc

- The creation site identified by `process.avg_Sensor.avg_List.cObj`:
  - Inv: $1 \leq j \leq N \land \text{L.size()} = d \land 1 \leq h \leq N$ (N=sl.size())
  - Count(Inv,N,t) = Nd
  - S(Inv,N,t) = Nd * Size(CO)

4- Adapt expression according to the type
**Computing totAlloc**

\[ \text{totAlloc}(\text{Process}) = 2Nt + Nd + 2N + 4T + 3 \]

- process(avg_Sensor.LR = 1
- process(avg_Sensor.itSL = 1
- process(avg_Sensor.LRe = N
- process(avg_Sensor.avg_List.itSL = N
- process(avg_Sensor.avg_List.cObj = Nd
- process(avg_Time.LR = 1
- process(avg_Time.L = t
- process(avg_Time.itSL = t
- process(avg_Time.Le = tN
- process(avg_Time.LRe = t
- process(avg_Time.avg_List.itSL = t
- process(avg_Time.avg_List.cObj = tN

**List avg_Sensor(SensorList sl)**
- LR::List() -> true
- itSL::Iterator -> true
- LRe::Float -> 1 <= j <= sl.size()

**List avg_Time(SensorList sl, int t)**
- LR::List() -> true
- L::Float -> 1 <= k <= t
- itSL::Iterator -> 1 <= k <= t
- Le::Float -> 1 <= k <= t
  & 1 <= h <= sl.size()
- LRe::Float -> 1 <= k <= t

**float avg_List(List L)**
- itL::Iterator -> true
- cObj::CO -> 1<= h <= L.size()
Main components

1) Finds out creation sites
2) Generate global invariants
3) Count the number of solutions (visits)
Main components

1) Ensures that variables concerning visits to statements are considered in the counting (to ensure soundness)
2) Try to compute a minimal set of variables (to filter out irrelevant variables, better precision)
Inferring regions

Our tool

1. Automatic region inference of m-regions
   - Using escape analysis
2. Translation to region based bytecode
   - RC (regions library)
   - RTSJ
   - JikesVM
Region synthesis performed by Escape analysis

process(SensorList sl, int t)

List avg_Sensor(SensorList sl)
  - LR::List() -> true
  - itSL::Iterator -> true
  - LRe::Float -> 1 <= j <= sl.size()

List avg_Time(SensorList sl, int t)
  - LR::List() -> true
  - L::Float -> 1 <= k <= t
  - itSL::Iterator -> 1 <= k <= t
  - Le::Float -> 1 <= k <= t
  - LRe::Float -> 1 <= k <= t

float avg_List(List L)
  - itL::Iterator -> true
  - cObj::CO -> 1 <= h <= L.size()
Computing region sizes (using \texttt{memCap})

\texttt{memcap(m) is totAlloc(m) applied only to captured allocations per regions}

\texttt{process(SensorList sl , int t)}

\texttt{List avg\_Sensor(SensorList sl)}
- \texttt{LR::List()} -> true
- \texttt{itSL::Iterator} -> true
- \texttt{LRe::Float} -> 1 <= \( j \) <= sl.size()

\texttt{List avg\_Time(SensorList sl , int t)}
- \texttt{LR::List()} -> true
- \texttt{L::Float} -> 1 <= \( k \) <= t
- \texttt{itSL::Iterator} -> 1 <= \( k \) <= t
- \texttt{Le::Float} -> 1 <= \( k \) <= t
  \&\& 1 <= \( h \) <= sl.size()
- \texttt{LRe::Float} -> 1 <= \( k \) <= t

\texttt{float avg\_List(List L)}
- \texttt{itL::Iterator} -> true
- \texttt{cObj::CO} -> 1\( <= h \) <= L.size()

Call from \texttt{avg\_Sensor}: 1 <= \( j \) <= N \&\& L.size() = d

Call from \texttt{avg\_Time}: 1 <= \( k \) <= t \&\& L.size() = N

\texttt{Process-region = N+T+2}
- \texttt{avg\_Sensor.LR} = 1
- \texttt{avg\_Sensor.LRe} = N
- \texttt{avg\_Time.LR} = 1
- \texttt{avg\_Time.LRe} = T

\texttt{avg\_Sensor-region = Nd + N + 1}
- \texttt{avg\_Sensor.itSL} = 1
- \texttt{avg\_Sensor.avg\_List.itSL} = N
- \texttt{avg\_Sensor.avg\_List.cObj} = Nd

\texttt{avg\_Time-region =2tN +3T}
- \texttt{avg\_Time.L} = t
- \texttt{avg\_Time.itSL} = t
- \texttt{avg\_Time.Le} = tN
- \texttt{avg\_Time.avg\_List.itSL} = t
- \texttt{avg\_Time.avg\_List.cObj} = tN
Approach: Leverage on region based memory manager to model consumption evolution (allocation – collection)

- We use **m-regions**: one region per method

- **When & Where** allocation/collection occurs:
  - created at the beginning of method
  - destroyed at the end

- **How much** memory is allocated/deallocated in each region:
  - \( \text{memCap}(m) \geq \) actual region size of m for any call context

- **How much** memory is allocated in outer regions:
  - \( \text{memEsc}(m) \geq \) actual memory that is allocated in callers regions
Computing memory requirements

- `avg_time` is called after `avg_sensor` finished its execution
- Region for `avg_sensor` is released before creating region for `avg_time`
- They do not occupy memory at the same time!

\[
\text{Peak}_\text{process} = N + t + 2 + \max(Nd + N + 1, 2tN + 3t)
\]

\[
\text{mem}^{\text{cst.m}}_{\text{mua}} = \max \text{size}^{\text{cst.m}}_{\text{mua}} + \max_{(m,l,m_i) \in \mathcal{E}_{\text{mua}}} \text{mem}^{\text{cst.m.l.m}_i}_{\text{mua}}
\]

(details about how to compute peak computation available in paper)
Refining memory regions

- Escape analysis over approximates object lifetime (to be safe)
  - It may impact of memory regions
- JScoper: A tool for Region edition and visualization
  - Call graph visualization
  - Region edition
  - Interfacing with Escape Analysis
  - Region-based code generation
  - Region-based memory manager simulator
    - Debugging
    - Analysis of case studies
Refining regions

process(SensorList sl, int t)

List avg_Sensor(SensorList sl)
- LR::List() -> true
- itSL::Iterator -> true
- LRe::Float -> 1 <= j <= sl.size()

List avg_Time(SensorList sl, int t)
- LR::List() -> true
- L::Float -> 1 <= k <= t
- itSL::Iterator -> 1 <= k <= t
- Le::Float -> 1 <= k <= t
  && 1 <= h <= sl.size()
- LRe::Float -> 1 <= k <= t

float avg_List(List L)
- itL::Iterator -> true
- cObj::CO -> 1 <= h <= L.size()
Region sizes on refined Regions

```
process(SensorList sl, int t)
```

```
List avg_Sensor(SensorList sl)
- LR::List() -> true
- itSL::Iterator -> true
- LRe::Float -> 1 <= j <= sl.size()
```

```
List avg_Time(SensorList sl, int t)
- LR::List() -> true
- L::Float -> 1 <= k <= t
- itSL::Iterator -> 1 <= k <= t
- Le::Float -> 1 <= k <= t
  && 1 <= h <= sl.size()
- LRe::Float -> 1 <= k <= t
```

```
float avg_List(List L)
- itL::Iterator -> true
- cObj::CO -> 1<= h <= L.size()
```

```
Process-region = N + t + 2
avg_Sensor.LR,
avg_Sensor.LRe
avg_Time.LR
avg_Time.LRe
```

```
avg_Sensor-region = 1
avg_Sensor.itSL
avg_Time-region = tN + 2t
avg_Time.L
avg_Time.itSL,
avg_Time.Le
```

```
avg_List-region = L.size() + 1
avg_List.itSL
avg_List.cObj
```
**Computing peak (revisited)**

- **avg_list** is called from different methods within loops
  - Regions for **avg_list** are created and destroyed several times

\[
\text{peak}^\uparrow(\sigma_0, m_0) = \max \sum \text{size}(r_k(\sigma))
\]

The consumption of all regions sharing the same calling context can be approximated by choosing an expression representing **the largest region size for that calling context**.
Approximating regions sizes

\[
\text{maxrsize}(\pi.m, mo)(P_{mo}) = \text{Maximize } rsize(m) \text{ subject to } I_\pi(P_{mo}, P_m, W)
\]

- **Avg_List-region expressed in terms of their parameters**
  - \(\text{memcap(avg LIST)} = \text{L.size() + 1}\)

- **Maximum according to calling context and in terms of process parameters**
  - \(\text{maxrsize}_{\text{process avg_sensor avg_list}}(N, d, t) = d + 1\)
    
    *Inv*: \(1 \leq j \leq N \&\& \text{L.size()} = d\)

  - \(\text{maxrsize}_{\text{process avg_time avg_list}}(N, d, t) = N + 1\)
    
    *Inv*: \(1 \leq k \leq t \&\& \text{L.size()} = N\)

Since region sizes can be polynomials, this lead to a non-linear optimization problem!
Non linear maximization problem solved using an approach based on Bernstein basis over polyhedral domains (Clauss et al. 2004-2009)

- Enables bounding a polynomial over a parametric domain given as a set of linear restraints
- Yields a set of candidate polynomials
Computing memory requirements

- \(\text{maxRsize}^{\text{process}}(N,t) = N + t + 2\)
- \(\text{maxRsize}^{\text{process . avg_sensor}}(N,t) = 1\)
- \(\text{maxRsize}^{\text{process . avg_time}}(N,t) = tN + 2t\)
- \(\text{maxRsize}^{\text{process . avg_sensor.avg_list}}(N,t) = d + 1\)
- \(\text{maxRsize}^{\text{process . avg_time.avg_list}}(N,t) = N + 1\)

Peak\_process(N,t,d) = N + t + 2 + 
+ \max(1 + d + 1, tN + 2t + N + 1) 
= N + t + 3 + \max(1 + d + 1, tN + 2t + N)

\[\text{mem}_{\text{cst.m}} = \text{maxRsize}_{\text{mua}}^{\text{cst.m}} + \max_{(m,l,m_i) \in \mathcal{E}_{\text{mua}}} \text{mem}_{\text{cst.m}.l.m_i}^{\text{mua}}\]
Comparing consumption using regions

Region stack evolution for \( N = 2, d = 20, t = 4 \)

Predicted memory requirements for 1, 3, and 4 regions.
\( N = 10..100, d = 20, t = 4 \)
We get some good results analyzing Jolden
- We used to check the accuracy of bounds

We analyzed some case studies:
- A fragment of a transaction processing component of a banking application
- A simplified fragment of a Collision Detector (CDx)

We analyzed the complete approach:
- generating and editing memory regions
- computing regions sizes and memory requirements.
Some lessons learnt

- For loop intensive programs the tool performs very well
  - Too conservative for recursive methods or data structures whose values affect future memory allocations
    - For instance in CDx we need to add a parameter denoting the maximum number of iterations required to process a data structure (processed in a recursive method)

- Imprecision came mainly from:
  - Escape analysis
  - Program invariants
  - Inductive variables analysis

- Global approach make the analysis and tuning of results a hard task
  - Need better tool support to deal with invariants at source level
  - Sometimes it would be necessary to provide bounds manually
    - Recursion, non analyzable methods, easy to understand but with a non-linear invariant
Conclusions

- A set of techniques for assisting the generation of region-based code with parametric upper bounds of memory consumption
  - Support for transforming java programs into region-based programs
  - Inference of parametric specifications of region sizes
  - Inference of parametric certificates of dynamic memory requirements

Future Work:

- Overcome restrictions on the input
  - Dealing with recursive methods and data structures
  - Other memory management mechanism / multithreading.
  - More complex data structures

- Improve precision
  - Not only polynomial consumption
  - Finer grained regions
  - Get better invariants

- Usability / Scalability:
  - Compositional analysis: Use of method summaries, allow annotations (consumption/aliasing).
    - Scalability: Usability; Modularity, deal with analyzable code.
  - Integration with other tools/techniques: JML / Spec#, Type Systems
  - Reduce dependency on program invariants

- Consider a concrete manager: fragmentation, administration overhead, etc.
<table>
<thead>
<tr>
<th>Author</th>
<th>Year</th>
<th>Language</th>
<th>Expressions</th>
<th>Memory Manager</th>
<th>Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hofmann &amp; Jost</td>
<td>2003</td>
<td>Functional</td>
<td>Linear</td>
<td>Explicit</td>
<td>No</td>
</tr>
<tr>
<td>Lui &amp; Unnikrishnan</td>
<td>2003</td>
<td>Functional</td>
<td>Recursive functions</td>
<td>Ref. Counting</td>
<td>Add-hoc (Lists)</td>
</tr>
<tr>
<td>Chin et al</td>
<td>2005</td>
<td>Java like</td>
<td>Linear (Pressburger): Checking</td>
<td>Explicit</td>
<td>Jolden</td>
</tr>
<tr>
<td>Chin et al</td>
<td>2008</td>
<td>Bytecode</td>
<td>Linear: Inference</td>
<td>Explicit</td>
<td>SciMark, MyBench</td>
</tr>
<tr>
<td>Albert et al (2)</td>
<td>2007</td>
<td>Bytecode</td>
<td>Recurrence equations</td>
<td>No &amp;&amp; Esc Analysis</td>
<td>No</td>
</tr>
<tr>
<td>Albert et al</td>
<td>2009</td>
<td>Bytecode</td>
<td>Closed forms</td>
<td>Esc Analysis</td>
<td>Jolden</td>
</tr>
</tbody>
</table>