Free-Me

A Static Analysis for Automatic Individual Object Reclamation

Samuel Z. Guyer, Kathryn S. McKinley, Daniel Frampton

presented by Stephanie Stroka
University of Salzburg

January 13, 2011

Outline

State of the Art

Manual Memory Management

- Can be more efficient
- Dangling pointers
- Double free
- Reachable and Unreachable memory leaks

Automatic Memory Management

- Stop-the-World/Concurrent/Incremental
- No dangling pointer bugs
- No double free bugs
- Reachable memory leaks

Free-Me idea

Why not just combine them?

- Combine benefits of both systems
- Discard disadvantages of both systems
- Reclaim memory quickly
- Reduce programmer effort

Related work

- Stack allocation, Escape Analysis
- Region Allocation

Free-Me idea

Goals

- Identify points in the program where memory can be discarded
- Allow handling of factory methods
- Discard memory immediately
- Reduce number of GC cycles

Code example

```
public void parse(InputStream stream) {
  while(...) {
    String idName = stream.readToken();
    Identifier id = symbolTable.lookup(idName);
    if(id == null) {
      id = new Identifier(idName);
      symbolTable.add(idName, id);
    }
    computeOn(id);
  }
}
```

Code example

```
public void parse(InputStream stream) {
  while(...) {
    String idName = stream.readToken();
    Identifier id = symbolTable.lookup(idName);
    if(id == null) {
      id = new Identifier(idName);
      symbolTable.add(idName, id);
    }
    else {
      // idName is no longer used
      free(idName);
    }
    computeOn(id);
  }
}
```

Compiler Analysis

The two kinds of points-to analysis

- Flow-insensitive pointer analysis
  $\Rightarrow$ To identify allocation nodes and factory methods
- Flow-sensitive liveness analysis
  $\Rightarrow$ To inserting calls to `free()`

Flow-insensitive pointer analysis

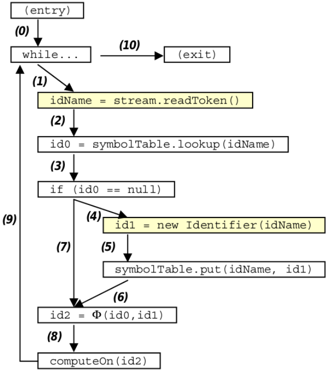

Figure 1: Single Static Assignment (SSA)

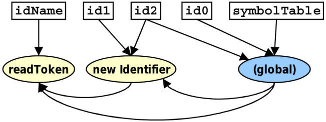

Figure 2: Connectivity Graph

Analyzing assignments

```
Object object = new Object();
Field field_1 = new Field();
object.field = field_1;
Field field_2 = object.field;
```

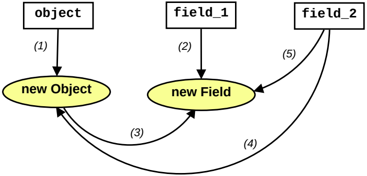

Figure 3: Connectivity Graph

Assignment	Points-to set
$v1 = v2;$	$PtsTo(v1) \cup = PtsTo(v2)$
$v = Cls.f;$	$PtsTo(v) \cup = \{N_G\}$[1]
$Cls.f = v;$	$PtsTo(N_G) \cup = PtsTo(v)$
$v1.f = v2;$	$\forall n \in PtsTo(v1)$ $PtsTo(n) \cup = PtsTo(v2)$
$v1 = v2.f;$	$PtsTo(v1) \cup = PtsTo * (v2)$[2]

Table 1: Rules for assignments

[1] N_G: Node for all globals

[2] $PtsTo*$: Transitive closure of points-to

Procedure summary

Method summaries aim to...

- summarize the intra-method connectivities
- keep record of passed parameters in callee-methods
- identify "hot" methods
  - Methods with allocation calls
  - Methods with factory calls

NODES	PROCEDURE SUMMARY
${N_{pj}}^{1} \in PtsTo*(p_i)$	record entry (p_i, p_j)
${N_{lj}}^{2} \in PtsTo*(p_i)$	record entry $(p_i, *p_j)$
$N_{pj} \in PtsTo*(N_G)$	record entry $(global, p_j)$
$N_{pj} \in PtsTo*(return)$	record entry $(return, p_j)$
$PtsTo*(return) \subset {N_A}^{3}$	record *method is a factory*

Table 2: Records for project summary

[1] Nodes for targets in parameters

[2] Parameter "inner" nodes

[3] Allocation nodes

Flow-sensitive pointer analysis / liveness analysis

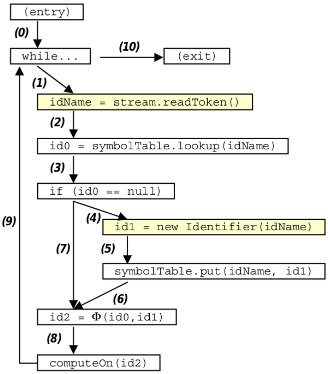

Figure 4: Single Static Assignment (SSA)

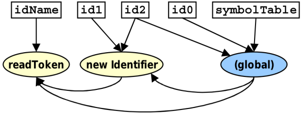

Figure 5: Connectivity Graph

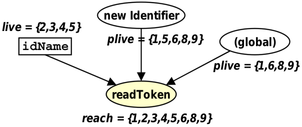

Figure 6: Liveness of `idName`

Result

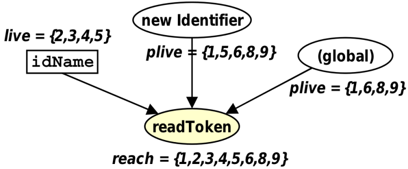

Figure 7: Liveness of `idName`

Result of liveness analysis on `idName`

→ `readToken` is reachable from program points {1,2,3,4,5,6,8,9}, BUT not from program point 7

Free placement

Where to put `free()`?

- Place `free()` as soon as possible
- Avoid excessive calls to `free()`
- Use temporary variables for every object that will be freed

```
// Wrong:
object = new Object();
object = o.field;
free(object);
```

```
// Correct:
object = new Object();
tmp0 = object;
object = o.field;
free(tmp0);
tmp0 = null;
```

Runtime Support

Implementations of `free()` for different allocators

- Size-segregated free-list implementation
- Bump-Pointer implementation

Implementations of `free()` for different collectors

- Mark-Sweep
- Reference Counting
- Copying Collector
- Generational Garbage Collector

Lazy Free-List

Lazy Free-List

- Supports *k* size-segregated *free*-list
- Incremental re-usage of memory
- Less free-list creations, memory tracing and GC cycles

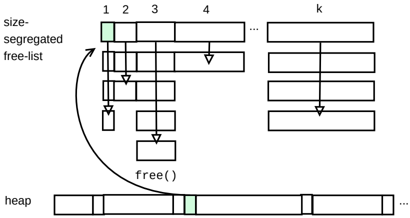

Figure 8: `free()` on segregated free-list

Bump-Pointer Allocation

Bump-Pointer Allocation

- Three implementations:
  - **Unbump**: Last allocated, first deallocated
  - **Unbump Region**: Memorize nearest, reclaimed region and unbump
  - **Unreserve**: Diminish reserved copying memory
- Slow memory fill-up, less GC cycles
- Smaller reserved region for copy process

Bump-Pointer Allocation

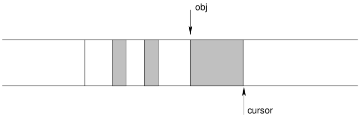

Figure 9: Unbump (1)

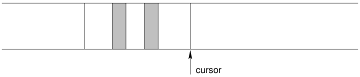

Figure 10: Unbump (2)

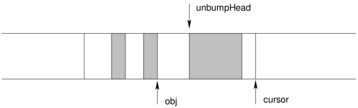

Figure 11: Unbump Region (1)

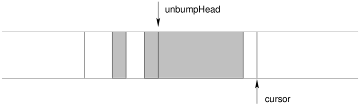

Figure 12: Unbump Region (2)

Optimization Techniques

Three steps

- Analyze Java standard class libraries during JikesRVM boot
- Pre-compute method summaries offline
- Pre-compile hot methods

$\Rightarrow$ almost doubles the compile time :-(

Results

Is Free-Me really saving memory?!

- Benchmarks: SPECjvm98, pseudojbb, SPECjbb2000, DaCapo
- On average, Free-Me frees **32%** of memory
- Max: **81%** of memory savings
- Compared to stack allocation: **+7%**
- Without conditional frees: **+11%**

Results

Is Free-Me really saving time?!

- Mark-Sweep Collector:
  - **5% - 50%**
  - Improves temporal locality and reduces allocator work
- Generational Collector:
  - Avg: No effect on GC time :-(
  - BUT: Improves collector time

Conclusion

Conclusion

- Analysis identifies a large fraction of short-lived objects
- Analysis is not effective on
  - large data structures
  - containers classes
  - conditional factories
- Provides incremental collection of garbage
- Works well on MS, but not on a Generational Collector