Analysis for Object Inlining in Java

Peeter Laud
Universität des Saarlandes

supported by the Esprit LTR project #28198
JAVA and CoSy Technology for Embedded Systems (JOSES)

Problem description

- Uniform representation of data:
  $\Rightarrow$ Pointer to the heap, pointing to actual data.
- Aggregated representation gives
  - less pointer dereferences;
  - less objects;
  - keeps (possibly) related objects together in memory.

  But has different semantics!

We want: automatically detect aggregation possibilities.
$\Rightarrow$ Must be semantics-preserving!!!

Type of analysis results

A map $\mathcal{I} : \langle \text{creation point}, \text{field} \rangle \mapsto \{\text{“inlined”}, \text{“not inlined”}\}$

$\mathcal{I}(c, f)$: is field f of objects created at c inlined or not?

Program transformation actually needs a lot of other information, too.

All derivable from $\mathcal{I}$ and type inference:

$\mathcal{T} : \langle \text{creation point}, \text{field} \rangle \mapsto \{\text{creation point}, \text{creation point}, \ldots\}$.

$(o$ created at c, o' created at c', $o.f = o') \Rightarrow c' \in \mathcal{T}(c, f)$

Transformation

1. Calculate the layouts of fused objects. Replace `new`-statements.
   - Size of an atomic or non-inlined field — fixed beforehand.
   - Size of an inlined field — size of the largest object inlined to it.
   - Size of an object — sum of the sizes of its fields.
2. Change some statements `x.f = y` to deep copies.
   - Iff in all creation points c of `x`, $\mathcal{I}(c, \mathtt{f}) =$ "inlined"
3. Adjust the accesses of objects to account for changed layout.
   - "pointer to an object" $\Longrightarrow$ "pointer to a field of an object"

Correctness conditions

- No cycles of the form $c_1 \xrightarrow{\text{inlined}} c_2 \xrightarrow{\text{inlined}} \cdots \xrightarrow{\text{inlined}} c_n \xrightarrow{\text{inlined}} c_1$.
- No type incompatibilities. Fit inside Java type system.
  - "pointer to an object" and "pointer to a field of an object" are incompatible
- No changes in aliasing patterns.

$\mathcal{I}$ must be such, that these conditions hold.

Structure of the remaining talk

- Two parts of the analysis.
  - Ensuring the preservation of sharing patterns.
    - Heap analysis, weaker than alias analysis.
    - Takes a rather abstract view of the heap.
  - Ensuring the compatibility of types (“representation analysis”).
- Preliminary results.

Heap analysis

View on heap: Heap is a set of objects, an object is a set of fields.

We allow a field `f` to be inlined, if

- All definitions of `y` at statement `x.f = y` are at **`new`**-statements.
  - Use-definition chains.
  - Aliasing of reference variables only.
- At `x.f = y`, the object pointed to by `x.f` is dead.
  - We don't let any other fields point to an inlined object.
  - For live variables `z`, $\mathcal{T}(\mathtt{z}) \cap \mathcal{T}(\mathtt{x.f}) = \emptyset$ must hold.

Heap analysis

Inlining may create new aliases.

- Statement `x.f = y`, field `g` of `y` could be inlined.
- If a live variable points to `y.g`:
  - Either `x.f` or `y.g` cannot be inlined.
  - We introduce a constraint system, add such constraints to it.
  - Objective function — inline as many fields as possible.

Representation analysis

- We want to find the possible types of variables.

  Onto which fields of which objects do the variables point?

- Use data flow analysis to propagate them through the program.

- Generate constraints stating the compatibility of those types.

  Generally, inlining fields makes types less compatible.

Both heap and representation analysis consist of bit-vector analyses.

Evaluation (very preliminary)

benchmark	source	size of `.class`-files	analysis time	total fields	inlined fields
compress	SPEC	77 KB	408 s	26	19
SciMark		16 KB	80 s	4	3
machineSim	ashes	27 KB	98 s	24	23
matrix	ashes	3 KB	6 s	4	4
Sets1	JGL	29 KB	108 s	10	5
Stack1	JGL	23 KB	38 s	2	2
PriorityQueue1	JGL	28 KB	55 s	8	5

Conclusion

- We have presented an analysis for object inlining:
  - more precise than previous analyses.
  - still scalable.
- More thorough evaluation remains to be done.
  - implementation may be tunable.
  - dynamic counts are more interesting.