1 2

Escape Analysis for Stack Allocation in Java

Eun-Sun Cho[1], **Kwangkeun Yi**[2]

[1] `eschough@ropas.kaist.ac.kr`
Dept. of Information & Computer Engineering
Ajou University
San 5 Wonchon-dong Paldal-gu
Suwon Kyunggi-do, Korea

[2] `kwang@ropas.kaist.ac.kr`
Research on Program Analysis System (ROPAS)
Dept. of Electrical Engineering & Computer Science
Korean Advanced Institute of Science & Technology
373-1 Kusong-dong Yusong-gu
Taejon, Korea

Abstract We suggest an escaping analysis for Java programs, to identify stack-allocatable objects. This analysis considers an object escaping if it is used after the deactivation of the method that created it. For each object, this analysis records the interprocedural movement of the method of its creation. An object is considered escaping the method that created it, if the method of its creation has been already deactivated. Our approach is different from prior works, in that special cares of non-local variables need not be taken. This enables us to handle some cases that are missed in the previous approach. The whole analysis is done in a single phase.

1 Introduction

Garbage collecting objects in Java makes memory management easier for the programmer, but it is time consuming – stack allocation may be an alternative [1,3,5]. This article presents a static analysis to identify the objects that never escape the method of their creation.

---

* This work is done while the first author was associated in ROPAS (http://ropas.kaist.ac.kr), KAIST in 1999 – 2000.
** This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology, and by Brain Korea 21 project.

P ::=3D $^*C$
C ::=3D `class` c `ext` c {`var`* x M^*}
M ::=3D `method` $m(x)$=3De

c *class name*
m *method name*
x *variable name*

e ::=3D x
| $head.x$
| x :=3D e
| $head.x$:=3D e
| `new` c
| e ; e
| `this`
| `if` e `then` e `else` e
| $e.m(e)$
$head$ ::=3D $head.x$
| `this`
| `x`

Fig. 1 Syntax of the language

The analysis is based on the fact that an object escapes the method that created it, if and only if it is used after the method is deactivated[4]. To identify such cases, we record for each object its interprocedural behaviors of the method that created it. At each expression using the object, the analysis checks if its record contains the sign for the deactivation of the method of its creation. If so, the object is deemed escaping the method; otherwise, it can be allocated in the method stack. This approach is quite different from conventional analyses, which consider objects escaping as long as they are assigned to non-local variables, irrespective of whether they are actually used or not.

The analysis is done in a single-phase, based on abstract interpretation[2]. We consider a core of Java for presentation simplicity. The abstract syntax is shown in Figure 1. We assume one formal parameter for each method, one field variable for each object and no arrays. We do not consider loops, which can be simulated by recursions. Neither do we consider local variables since they make no important effects on our analysis. And '`this`' cannot be abbreviated in the expressions for field access and method invocations upon it.

Except the case that `finalize` methods destroy the objects in sequence, no ordering on the destruction of objects is assumed, such as the one forced by run-time garbage collection.

2 Analysis

To safely estimate the information of interprocedural behavior, we build an abstract interpreter. Iterative evaluation of the interpretation function reaches to a fixed point that is our analysis result.

At the creation of each object, we start recording the interprocedural movements of the method that created it by encoding the movements into seven types of abstract entities: $\hat{\mathcal{P}}$ =3ff $\{\perp, \epsilon, u, u^{+}udd^{+}, ud, ud^{+}\}$. For the object `o`, let the method that created it be `m`; ϵ indicates that no activation/deactivation of `m` has happened; u ($u$pward movement of the stack pointer) means that an instance of `m` is deactivated (returned); uu^+ and dd^+ respectively means that multiple instances of `m` has been deactivated and activated; ud represents one call after one return from `m`; ud^+ stands for one or more activation after one or more deactivation of `m`. The matched pairs of activation and deactivation of the same instance of `m` is ignored, which is useless in determining if it is "outside of `m` or not". The four interprocedural movements u, uu^+, ud, and ud^+ of `m` for object `o` imply that that the object *may-escape* `m`. These seven abstract symbols are proved to cover all the cases of interprocedural movement of `m` after the creation of `o`.

This record of the interprocedural behavior is based on the concept of *procedure string* in Harrison's work[4]. He proposed an escaping analyses for functional languages, to handle implicitly created objects such as cons cells and closures. Procedure string is the sequence of α^u's and α^d's to record the interprocedural behavior of running programs, where α is a function (method) name. An abstract value of a procedure string is a mapping from the function (method) names to subsets of the set $\{\epsilon, u, uu^+, d, dd^+, ud, ud^+\}$.

Note that we do not use d, which would indicate a single call of method `m` that created the object `o`. It would happen only if another instance of `m` is invoked before the existing instance of `m`, which created `o`, is deactivated. This means that the program part responsible for d must include recursive call of `m`, which would always result in dd^+ after iterative evaluation of abstract interpreter to get a fixed point. In addition, we have a new type ud, representing a single pair of deactivation and invocation. This enables us to handle the repeated but non-recursive invocations of `m`, which were abstracted in the same way as recursive invocations of `m` in [4].

An abstract memory object is identified by the `new` expression together with the record of interprocedural movement of the method of its creation. The former one ensures that for each `new` expression, at least one abstract object is identified. The latter one is useful when a `new` expression creates more than one object, as in loops or in recursive functions. $2^{O\hat{R}ef}$ in heap, environment and escaping is the set of ideals ordered by set inclusion.

The interpretation function $\hat{\mathcal{E}}$ takes a pre-state and returns a post-state, where a pre-state is an element of $\hat{Env} \times \hat{Heap} \times \hat{Escape}$, and a post-state is an element of $2^{O\hat{R}ef} \times \hat{Env} \times \hat{Heap} \times \hat{Escape}$. The first element of a post-state represents a set of abstract objects, the possible result of a given

Domains

labels for new expressions	$\beta \in B$
interprocedural behavior	$\hat{p} \in \hat{\mathcal{P}}$ =3D $\{\epsilon, u, d, uu^+, dd^+, ud, ud^+\} \cup \bot$
object reference	$\hat{r} \in O\hat{R}ef$ =3D $\bot B \times \hat{\mathcal{P}}_\bot$
heap	$\hat{h} \in H\hat{e}ap$ =3D $O\hat{R}ef \to 2^{O\hat{R}ef}$
environment	$\hat{\sigma} \in E\hat{n}v$ =3D $2^{O\hat{R}ef} \times 2^{O\hat{R}ef}$
escaping	$\hat{e} \in Esc\hat{a}pe$ =3D $\bot B \to \{\top, \bot\}$

$$\hat{\mathcal{E}} \in\ Expr \to (\hat{Env} \times \hat{Heap} \times \hat{Escape}) \to (2^{\hat{ORef}} \times \hat{Env} \times \hat{Heap} \times \hat{Escape})$$

$$\begin{aligned}\hat{\mathcal{F}} \in\ & (Expr \to (\hat{Env} \times \hat{Heap} \times \hat{Escape}) \to (2^{\hat{ORef}} \times \hat{Env} \times \hat{Heap} \times \hat{Escape})) \to \\ & (Expr \to (\hat{Env} \times \hat{Heap} \times \hat{Escape}) \to (2^{\hat{ORef}} \times \hat{Env} \times \hat{Heap} \times \hat{Escape}))\end{aligned}$$

$$\begin{array}{ll}
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ x\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & =3D \quad \langle\ snd(\hat{\sigma}), \hat{\sigma}, \hat{h}, \hat{e}\ \rangle \\
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ \mathtt{this}\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & =3D \quad \langle\ \sigma(\mathtt{this}), \hat{\sigma}, \hat{h}, \hat{e}\ \rangle \\
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ head.x\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & =3D\ let\ \langle\ \hat{rs}, \hat{\sigma}, \hat{h}, \hat{e_1}\ \rangle =3D \hat{\mathcal{E}}[\![\ head\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle \\
 & \quad in\ \langle\ ReadH\ \hat{h}\ \hat{rs}, \hat{\sigma}, changeE\ \hat{e_1}\ \hat{rs}\ \rangle \\
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ x{:}{=}3De\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & =3D\ let\ \langle \hat{rs_1}, \hat{\sigma_1}, \hat{h_1}, \hat{e_1}\ \rangle =3D \hat{\mathcal{E}}[\![\ e\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle \\
 & \quad in\ \langle\ \hat{rs}_1, \langle\ fst(\hat{\sigma_1}), snd(\hat{\sigma_2}) \sqcup \hat{rs}_1\ \rangle, \hat{h_1}, \hat{e_1}\ \rangle \\
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ head.x{:}{=}3De\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & =3D\ let\ \langle \hat{rs_1}, \hat{\sigma}, \hat{h}, \hat{e_1}\ \rangle =3D \hat{\mathcal{E}}[\![\ head\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle \\
 & \qquad \langle\ \hat{rs}_2, \hat{\sigma_2}, \hat{h_2}, \hat{e_2}\ \rangle =3D \hat{\mathcal{E}}[\![\ e\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e_1}\ \rangle \\
 & \quad in\ \langle\ \hat{rs}_2, \hat{\sigma_2}, WriteH\ \hat{h_2}\ \hat{rs}_1\ \hat{rs}_2, \\
 & \qquad\qquad changeE\ \hat{e_2}\ \hat{rs}_1\ \rangle \\
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ \mathtt{new}_\beta\ c\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & =3D\ let\ \hat{r} =3D \langle\ \beta, \epsilon\ \rangle\ \ \hat{rs} =3D \{\hat{r}\} \\
 & \quad in\ \langle\ \hat{rs}, \sigma, WriteH\ \hat{h}\ \hat{rs}\ \{InitField(c)\}, \hat{e}\ \rangle \\
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ e_1.m(e_2)\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & =3D\ let\ \langle \hat{rs_1}, \hat{\sigma_1}, \hat{h_1}, \hat{e_1}\ \rangle =3D \hat{\mathcal{E}}[\![\ e_1\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle \\
 & \qquad \langle\ \hat{rs}_2, \hat{\sigma_2}, \hat{h_2}, \hat{e_2}\ \rangle =3D \hat{\mathcal{E}}[\![\ e_2\ ]\!]\langle\ \hat{\sigma_1}, \hat{h_1}, \hat{e_1}\ \rangle \\
 & \quad in\ \begin{cases} \langle\ \hat{rs}_1, \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & if\ \hat{rs}_1 =3D \bot \\ \bigsqcup_{pairwise}\ \{\ \langle\ \hat{UpRs}\ \hat{rs}_3\ \alpha, \sigma_2, \hat{UpH}\ \hat{h_3}\ \alpha, \hat{e_3}\ \rangle\ | & \\ \quad \langle\ \hat{rs}_3, \hat{\sigma_3}, \hat{h_3}, \hat{e_3}\ \rangle =3D \hat{\mathcal{E}}[\![\ e\ ]\!] & \\ \qquad \langle\ \hat{DownEnv}\ \langle\ \{\hat{r}\}, \hat{rs}_2\ \rangle\ \alpha, \hat{DownH}\ \hat{h_2}\ \alpha, & \\ \qquad changeE\ (changeE\ \hat{e_2}\ \hat{rs}_2)\{\hat{r}\}\ \rangle & \\ \quad \wedge\ \lambda_\alpha x.e =3D \hat{Method}(snd(\hat{r}), m) \wedge\ \hat{r} \in \hat{rs}_1\ \} & \\ & otherwise \end{cases} \\
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ e_1;\ e_2\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & =3D\ let \qquad \langle \hat{rs_1}, \hat{\sigma_1}, \hat{h_1}, \hat{e_1}\ \rangle =3D \hat{\mathcal{E}}[\![\ e_1\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle \\
 & \quad in\ \hat{\mathcal{E}}\ [\![\ e_2\ ]\!]\langle\ \hat{\sigma_1}, \hat{h_1}, \hat{e_1}\ \rangle \\
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ \mathtt{if}\ e_1\ e_2\ e_3\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & =3D\ let \qquad \langle \hat{rs_1}, \hat{\sigma_1}, \hat{h_1}, \hat{e_1}\ \rangle =3D \hat{\mathcal{E}}[\![\ e_1\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle \\
 & \quad in\ \hat{\mathcal{E}}\ [\![\ e_2\ ]\!]\langle\ \hat{\sigma_1}, \hat{h_1}, changeE\ \hat{e_1}\ \hat{rs}_1\ \rangle \\
 & \qquad\quad \bigsqcup \hat{\mathcal{E}}\ [\![\ e_3\ ]\!]\langle\ \hat{\sigma_1}, \hat{h_1}, changeE\ \hat{e_1}\ \hat{rs}_1\ \rangle
\end{array}$$

Fig. 2 Definition of abstract evaluation function

expression. Each pre-state and post-state describes the current heap, the current environment and escaping information. A heap maps each object to its possible field values, which are again objects. *head.x* refers the heap; *head.x*:=3ffe and $\mathtt{new}_\beta c$ change the heaps. An environment consists of the pairs of sets of objects for the variable `this` and those for the parameter x. It is accessed by the expressions x and `this`, and modified by x:=3ffe. Modifying `this` is impossible.

The method is invoked with a new environment that consists of the receiver and the parameter as its `this` and `x`; the previous environment is restored after the method is returned. The evaluation of method invocation updates the record of the interprocedural behavior for the objects related. This assigns new identifiers to the objects, since the record is a part of the object identifier. To hide the complexity of this operation, auxiliary functions prefixed with *Up* and *Down* are introduced. They also adjust the heap and the environment to the new object identifiers. The definitions of auxiliary functions are in the appendix.

The result of the analysis is collected in $\hat{e}$, a mapping from each `new` expression to $\{\top, \bot\}$, where $\top$ means *may-escaping* and $\bot$ means *never-escaping*. At the beginning of the analysis, the $\bot$ value is assumed for every `new` expression. The value is updated to $\top$ whenever one of the objects created by the `new` expression is possibly used. Here, "using" an object means that the object is accessed. Thus, $changeE$, updating operation for $\hat{e}$, is applied whenever the objects in heap are referenced: reading/writing the objects, passing the objects to the methods as actual parameters are passed, and so on. Note that, during the abstract evaluation the escaping value is only increasing from $\bot$ to $\top$, which makes the iterative application of the abstract monotonic interpreter (collecting the escaping information) reach a fixed point within a finite time.

The correctness of our analysis is shown by the following theorem. $fix\, \mathcal{F}[\![\, expr \,]\!]$ is a concrete semantics of the language. We abuse the terms 'pre-state' and 'post-state' in the theorem for the ones before the abstraction.

Theorem 1 *For any expression expr,*

$$Abs_{post\text{-}state} \circ fix\ \mathcal{F}[\![\ expr\ ]\!] \sqsubseteq fix\hat{\mathcal{F}}[\![\ expr\ ]\!] \circ Abs_{pre\text{-}state}$$

3 Related Works

Unlike our analysis, previous escaping analyses for Java[1,3,5] consider an object escaping the method that created it, as soon as it is assigned to global variables, parameters, and/or return variables. Thus, they miss the objects that are assigned to such variables, but not actually used outside the methods of their creation. As far as we know, this analysis, considering actual using points of objects, is the only one to cover such cases.

Blanchet[1] proposed a type-based and two-phase (a backward phase and a forward phase) escaping analysis. Like our work, he build an abstract interpretation and defined more than one abstract objects for each `new` expression.

4 Conclusions

This paper introduces a single phase escape analysis based on abstract interpretation to determine if an object is stack-allocatable. By recording the interprocedural movement of the method of creation for each object, this analysis considers an object stack allocatable as long as it is not actually used after the method of its creation is deactivated. We are now concentrating on the cases with the unknown methods, and implementing the analysis to obtain experimental result.

Appendix : Auxiliary functions

$$changeE =3D\ \lambda\hat{e}.\lambda\hat{rs}\begin{cases} \perp & if\ \hat{e} =3D \perp \\ \lambda\beta.\begin{Bmatrix} \hat{e}(\beta) \sqcup \top\ if\ \hat{p} \in \{u, uu^+, ud, ud^+\} \\ \quad for\ some\ \langle\ \beta, \hat{p}\ \rangle \in \hat{rs} \\ \hat{e}(\beta) \quad otherwise \end{Bmatrix} & otherwise \end{cases}$$

$$ReadH =3D \lambda\hat{h}.\lambda\hat{rs}.\bigsqcup_{\hat{r}\in\hat{rs}} \hat{h}(\hat{r}),\ WriteH =3D \lambda\hat{h}.\lambda\hat{rs}_1.\lambda\hat{rs}_2.\lambda\hat{r}.\begin{cases} \perp & if\ \hat{r} =3D \perp \\ \hat{h}(\hat{r}) & if\ \hat{r} \notin \hat{rs}_1 \\ \hat{h}(\hat{r}) \sqcup \hat{rs}_2 & otherwise \end{cases}$$

	ϵ	u	d^+	uu^+	ud	ud^+	$\perp$
$\oplus d$	$\{d\}$	$\{ud\}$	$\{d^+\}$	$\{ud^+\}$	$\{ud^+\}$	$\{ud^+\}$	ϕ
$\oplus u$	$\{u\}$	$\{uu^+\}$	$\{\epsilon, d^+\}$	$\{uu^+\}$	$\{u\}$	$\{u, uu^+, ud^+\}$	ϕ
$\ominus d$	ϕ	ϕ	$\{\epsilon\}$	ϕ	$\{u\}$	$\{u, uu^+, ud^+\}$	ϕ
$\ominus u$	ϕ	$\{\epsilon\}$	ϕ	$\{u, uu^+\}$	ϕ	ϕ	ϕ

$$\hat{DownR} =3D\ \lambda\langle\ \beta, \hat{p}\ \rangle.\lambda\alpha.\begin{cases} \perp =3D \phi & if\ \langle\ \beta, \hat{p}\ \rangle =3D \perp \\ \{\langle\ \beta, \hat{p'}\ \rangle | \hat{p'} \in \hat{p} \oplus d\} & if\ birthmethod(\beta) =3D \alpha \\ \{\langle\ \beta, \hat{p}\ \rangle\} & \end{cases}$$

$$DownRs =3D\ \lambda\hat{rs}.\lambda\alpha.\begin{cases} \phi & if\ \hat{rs} =3D \phi \\ \{DownR\ \hat{r}\ \alpha | \hat{r} \in \hat{rs}\} & otherwise \end{cases}$$

$$\hat{DownH} =3D \lambda\hat{h}.\lambda\alpha.\lambda\langle\ \beta, \hat{p}\ \rangle.\begin{cases} \perp & if\ \hat{h} =3D \perp =3D \phi \\ \sqcup\{\hat{DownR}\ \hat{h}\langle\ \beta, \hat{p'}\ \rangle\ \alpha | \hat{p'} \in \hat{p} \ominus d\} & if\ birthmethod(\beta) =3D \alpha \\ \hat{DownR}\ \hat{h}\ \langle\ \beta, \hat{p}\ \rangle\ \alpha & otherwise \end{cases}$$

$$Down\hat{E}nv =3D\ \lambda\hat{\sigma}.\lambda\alpha.\begin{cases} \langle\ \phi, \phi\ \rangle & if\ \hat{\sigma} =3D \perp \\ \langle\ \bigsqcup\{DownR\ \hat{r}\ \alpha\ |\ \hat{r} \in fst(\hat{\sigma})\}, & \\ \quad \bigsqcup\{DownR\ \hat{r}\ \alpha\ |\ \hat{r} \in fst(\hat{\sigma})\}\ \rangle & otherwise \end{cases}$$

References

1. B. Blanchet. “Escape analysis for object oriented languages application to java”. In *Proc. of the ACM OOPSLA Conf.*, 1999.
2. P. Cousot and N. Cousot. “Abstract interpretation : a unified lattice model for static analysis of programs by construction of approximation”. In *Proc. of SIGPLAN Conf. on Principle of Programming Languages*, 1977.
3. J-D. Choi et. al. “Escape Analysis for Java”. In *Proc. of the ACM OOPSLA Conf.*, 1999.
4. W. L. Harrison. “The Interprocedural Analysis and Automatic Parallelization of scheme Programs”. *Lisp and Symbolic Computation*, 2(3):179–396, 1989.
5. J. Whaley and M. Rinard. “Compositional Pointer and Escape Analysis for Java Programs”. In *Proc. of the ACM OOPSLA Conf.*, 1999.