Preliminary Report on Context-Sensitive Escape Analysis

Aditya Agrawal
CS20S026

CONTENTS

Preliminary Report on Context-Sensitive Escape Analysis

***Abstract*—Escape Analysis is a static analysis that determines whether the lifetime of data may exceed its static scope. The analysis performed is inter-procedural, context-sensitive and flow sensitive analysis which takes alias analysis as an intermediary step. The goal of escape analysis is achieved by performing two interdependent analysis – one forward and the other backward. Escape Analysis can be applied to stack allocation and synchronization elimination.**

I. Introduction

Most of the programming languages currently use a garbage collector to make memory management easier for the programmer. The programmer is not responsible for the de-allocation of objects and is done automatically by the garbage collector. Since Java is designed to be a safe programming language, it cannot rely on the programmer to correctly deallocate the objects from the memory. Hence, Java also provides a Garbage Collector which is used by the JVM for freeing up memory (in the heap) when objects are no longer being used. However, there a lot of overheads involved in garbage collection like managing the Eden Space, the survivor space with operations such as mark sweep and compact. Hence one solution is to go with stack allocation. However, it is only possible to stack-allocate the data if its lifetime does not exceed the static scope or the scope that can be defined in compile time (without executing the program). Therefore, the goal of escape analysis is to determine precisely what objects can be stack-allocated. Let us see an example to understand it better.

```
//LinkedList.java
class LinkedList{
  LinkedList next;
  int value;
  LinkedList(LinkedList next,int newVal){
    this.next = next;
    this.value = newVal;
  }
  LinkedList run(){
    LinkedList next = new LinkedList(null,1);
    LinkedList head = new LinkedList(next,2);
    return head.next;
  }
}
```

In the given program, the class LinkedList is used to build a linked list data structure with the next node and the corresponding integer value. The goal of escape analysis says that an object o can be stack allocated on a method m if and only if it is not reachable after the end of m. By reachability meaning it is not reachable from the parameters as well as the result of m nor from any static fields. In the run method of the given program, the object head.next is reachable from the result of run so it cannot be stack allocated. However, the variable head has its scope limited to the method run so the head variable can be stack allocated. By using escape analysis, we can also eliminate synchronization elimination. If escape analysis claims that the object o does not escape from m, o is also local to the current thread. The algorithm presented in the paper uses alias analysis as an intermediary step to prove correctness and also to handle any local variables aliased with escaping objects of method m such as parameters, static fields and the result of m(return value). As a result, we do not need to do synchronization when calling a synchronized method on object o. This is a useful optimization since synchronization is a costly operation in the JDK. Synchronization elimination could be applied to multi-threaded languages and also single threaded languages where libraries use synchronization to ensure thread-safety in all cases.

II. Algorithm

A. *Notation*

i ∈ Ind = N	Parameter Indices
ρ, ρ' ∈ Env = Method → Ind → Val	Environment for params
ρ_S, ρ'_S ∈ Env = Method → Val	Environment for the result
L, Ls ∈ LocVar = V → Val	Abstract Local Variables
E, Es ∈ Method → LocVar	Escape Analyses

B. *Lattice Representation*

An object is said to escape from method m only if it is reachable from the parameters or the result of m or from static fields. If an object escapes from m, it will not be stack allocated in m. A part of a data structure may escape while another part doesnt. Like, in the given program, LinkedList.run(), the head variable does not escape the method although head.next does. So, the output of the algorithm should contain what parts of the data structure does the method escapes. So, in the analysis, set of access paths have been chosen to represent the parts of objects. An Access Path = (Field ∪ N)* denotes the field that we are accessing or the nth element of an array n ∪ N. A given field can be accessed by (C,f,t) where C is the class name, f is the field name and t is the type of field to be returned. The escaping part of an object is called the escape context of the object. Thereby, escape contexts for this analysis are sets of paths: Ctx = P(Path) where they are ordered by inclusion. Therefore, Ctx is a complete lattice. The analysis E computes for each variable v, its escape context which is nothing but the part of v that escapes represented by a set of access paths. A newly allocated object can be stack-allocated if the object itself does not escape or the ∈ path is not in the escape context of the variable. The goal of the algorithm is to compute for each access path, the set of other access paths of data of scope larger than the analyzed method (parameters,

P(pc)	Forward Analysis L_S
entry to m	$L_S = P_E$
v1 = v2 or v1 = (t)v2	$L_S(v1) = L_S(v2)$
v1 = ϕ(v2,v3)	$L_S(v1) = L_S(v2) \cup L_S(v3)$
v1 = v2.f	$L_S(v1) = f^{-1}.L_S(v2)$
v1.f = v2	NA
v = C.f	$L_S(v) = T_E[f]$
C.f = v	NA
v1 = v2[v3]	$L_S(v1) = N^{-1}.L_S(v2)$
v1[v2] = v3	NA
v = new C/new t[w]	$L_S = L(v)$
v = null	L_S
if(...) goto a	NA
w = v_0.m'(v1,....,vn)	$L_S(w) = p'_S(m')$ o $(L(w),L_S(v_0),...,L_S(v_n))$
return v	$p_S(m)$ ¿= $L_S(v)$

P(pc)	Backward Analysis L
entry to m	$\forall \cup [0,j], p(m)(i) >= L(p_i)$
v1 = v2 or v1 = (t)v2	$L(v_2) >= L(v_1)$
v1 = ϕ(v2,v3)	$L(v_2) >= L(v_1), L(v_3) >= L(v_1)$
v1 = v2.f	$L(v_2) >= f.L(v_1)$
v1.f = v2	$L(v_2) >= f^(- 1).L_S(v1), L(v_1) >= f.L_S(v_2)$
v = C.f	NA
C.f = v	$L(v) >= T_E[f]$
v1 = v2[v3]	$L(v_2) >= N.L(v_1)$
v1[v2] = v3	$L(v_3) >= N^(- 1).L_S(v1), L(v_1) >= N.L_S(v3)$
v = new C/new t[w]	NA
v = null	NA
if(...) goto a	NA
w = v_0.m'(v1,....,vn)	$L(v_i) >= p'(m')(i)o (L(w), L_s(v_0), ...L_S(v_n)), i \cup [0,n]$
return v	$L(v) >= first_E$

Table 1 : Equations for deriving the escape context

result of the method and static fields). There are two basic operations on escape contexts:-

- constructions f.c where f ∈ Field and c ∈ Context : If c is a context associated with field f of an object o, f.c yields the context associated with o.
- f^{-1}.c is a restriction: if c is the context associated with an object o, f^{-1}.c yields the context of the field o.f.

For example, in the given example, program the escape context c(run) = {head.next}. The fields associated with head which are escaping can be found by $head^{-1}$.c = {next}.

C. Escape Analysis

To compute the escape information, the paper uses a bidirectional propagation algorithm. E is called the backward analysis whereas Es is a forward analysis. The analysis E and Es depend on each other. The backward propagation starts from the results and moves up to the parameters where as the forward analysis starts from the parameters and goes down to the results. Given the code snippet below,

```
//BackwardAnalysisExample
method m(v0){
  v0 = new ...;
  v1 = v2.f;
  return v1;
}
```

The backward analysis algorithm would first mark v1(which is the result of m) as escaping. Then when the corresponding assignment statement is encountered, the corresponding field f of v2 is also marked as escaping if v1 is marked as escaping. But, consider the assignment to v0. The backward analysis algorithm would have had no idea then that the v0 is the parameter of the method which would escape. So, we need a forward propagation algorithm for this case. The abstract values are context transformers in the sense functions from contexts to contexts. Knowing the contexts associated with the parameters with the help of forward analysis $c_0...c_j$ and with the result by backward analysis c_{-1} of the current method, they yield the context associated with the corresponding concrete value $\phi(c_{-1}...c_j)$. Therefore $Val_E = Ctx_E \rightarrow Ctx_E$. The this pointer is treated as a parameter to the method. The greatest element of the lattice of abstract values is $T_E[f] = (c_{-1},...,c_j) \rightarrow$ Path. The abstract value of the result is $first_E = c_{-1}$. If the formal parameters of m are $p_0,...,p_j$, the abstract values for the parameters are $P_E(p_i) = c_i$.

The escape analysis E_S and E output for each method, each local variable v of m, its corresponding escape abstract value E(m)(v) or E_S(m)(v). The environment p ∈ Env_E yields for each method the escape information for its parameters by the analysis E. Each parameter is represented by an index i ∈ Ind. The environment p_S ∈ Env_{ES} yields for each method the escape information for its result by the backward propagation. The analysis E and E_S yield the escape information for each variable. The E and E_S change according to the rules given in Table 1.

The E_S is a forward analysis algorithm so it is defined when a variable v is defined and used when v is used. Similarly since E is a backward analysis algorithm, it is defined when v is used and used when v is defined. It propagates information from the uses to definition of v and ends when the begin of the method is encountered and the information is propagated to the environment ρ. Similarly, the forward analysis ends when the return statement is encountered and the information is propagated to ρ_S.

On entry to method m, the abstract values of the parameters are initialized for the forward analysis algorithm. So, the escape context of any parameter p_i is denoted as $E(m)(p_i) = c_i$. Therefore $L_S(p_i) = P_E(p_i)$. Now, we describe the rules given in Table 1. For local variable assignments, we have already seen an example above to understand how the backward analysis would try to propagate the escape information.

When reading a field by $v_1 = v_2.f$, the contents of v_1 is reachable as soon as the field f of v_2 would be reachable. Hence, the escape information of v_1 is computed by restricting the information of v_2 to field f in the analysis E_S (E-store). For E, if a part of v1 escapes, the same part of the field of v_2 escapes.

When writing into a field $v_1.f = v_2$, the part of field f of v_1 that escapes is the part of v_2 that escapes. If the field f of v_1 escapes, then v_2 may escape and vice versa.

For a virtual method call m' where m' = (C,m",t) we consider all the methods that can be called from the subclass of C. Therefore, a class hierarchy analysis and corresponding call graph can give information of which method(s) to call. The escape information for all the calls can be defined in terms of the environments : $\rho'(C, m'', t)(i) =$

$\cup foreverysubclassC' ofC\rho(C', m'', t)(i) and \rho'_S(C, m'', t)(i) = \cup foreverysubclassC' ofC\rho_S(C', m'', t)(i)$
On a return from method m, the escape information for the result of m is updated in environment ρ_S. As we know, the escape context of the result is c_{-1}.

The analysis algorithm works in several passes, as follows:- For each method m, of it has not been analyzed yet, search the call graph of methods that may be called from m and which have not been analyzed yet, and compute the strongly connected components of this graph.

For each strongly connected component,

- Build the equations for each method
- Transform the code into SSA Format
- Create unknowns L1(m)(v) and L1s(m)(v) for each variable v. Emit the equations of the escape analysis algorithm given in Table 1 for each instruction.
- Solve the equations using an iterative fixpoint solver.
- Prepare the post-transformation by building the structure giving for each allocation its escape information.

III. Example

We will use the example program as given in the Introduction Section. The corresponding table is given below for the method LinkedList constructor. :-

Analysis E_S	Program Instruction	Analysis E
$L_S(p_i) = P_E(p_i)$	LinkedList(LinkedList n,int newVal){	
	this.next = n;	$L(n) >= next^{-1}.L_S(this), L(this) >= next.L_S(n)$
	this.value = newVal;	$L(newVal) >= value^{-1}.L_S(this)$
	}	$L(this) >= value.L_S(newVal)$

The left column provides equations for the forward analysis part while the right column provides equations for the backward analysis part.
The middle column is the analyzed instruction. Abstract values for this method have three parameters which are the escape contexts of this, n and newVal. There is no c_{-1} here because the method doesn't return anything. So the abstract values are of the form $(c_0, c_1, c_2) \rightarrow \phi(c_0, ..., c_2)$ which represent the concrete values. We say that an object o store-escapes m when if we store an object o' in o, o' escapes. Analysis E_S determines a superset of the set of objects that store-escape. At the beginning of the method, the local variables are initialized with the parameters of m so the corresponding escape information is P_E.
The right column denote the backward propagtion pass. The first line $L(n) >= next^{-1}.L_S(this$, means that if the next field of this store escapes, then n escapes. The second inequation $L(this) >= next.L_S(l)$ means that if n store escapes, then next field of this escapes. We obtain then the following equations:- E(m)(n) = $(c_0, c_1, c_2) \rightarrow next^{-1}.c_0$. and E(m)(this) ¿= $(c_0, c_1, c_2) \rightarrow next.c_1$. The analysis, therefore yields
$E(m)(this)(c_0, c_1, c_2) = next.c_1 \cup value.c2$
$E(m)(l)(c_0, c_1, c_2) = next^{-1}.c_0$.

IV. Extending the Analysis to Handle Concurrency

The given analysis works with single threaded programs but some problems may arise due to concurrent multi-threaded programs. This report summarizes the learnings involved so far and focuses on finding methods to extend it in terms of concurrent programs.