NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

POSTGRADUATE STUDIES PROGRAM

MASTER THESIS

Declarative Whole-Program Escape Analysis for Java

Georgios Balatsouras

Supervisor: **Yannis Smaragdakis**, Associate Professor NKUA

ATHENS

MARCH 2012

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δηλωτική Ανάλυση Διαφυγής Πλήρους Προγράμματος για Java

Γεώργιος Μπαλατσούρας

Επιβλέπων: **Γιάννης Σμαραγδάκης**, Αναπληρωτής Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΜΑΡΤΙΟΣ 2012

MASTER THESIS

Declarative Whole-Program Escape Analysis for Java

Georgios Balatsouras

RN: M1057

SUPERVISOR:

Yannis Smaragdakis, Associate Professor NKUA

THESIS COMMITTEE:

Yannis Smaragdakis, Associate Professor NKUA
Panos Rondogiannis, Associate Professor NKUA

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δηλωτική Ανάλυση Διαφυγής Πλήρους Προγράμματος για Java

Γεώργιος Μπαλατσούρας

ΑΜ: Μ1057

Περίληψη

Ένα αντικείμενο θεωρείται ότι 'διαφεύγει' όταν η διάρκεια ζωής του μπορεί να είναι μεγαλύτερη από αυτή της μεθόδου που το δημιούργησε. Παρουσιάζουμε μία δηλωτική ανάλυση πλήρους προγράμματος για τη Java, γραμμένη στην γλώσσα Datalog, που υπολογίζει μία υπερεκτίμηση των αντικειμένων του σωρού που μπορούν να διαφύγουν. Η ανάλυσή μας είναι μέρος του Doop framework [3], το οποίο περιλαμβάνει μία ανάλυση δεικτών για ένα σύνολο από υποστηριζόμενους τύπους συμφραζομένων.

Η ανάλυσή μας μπόρεσε να ανιχνεύσει ότι, κατά μέσο όρο, το 60.66% των εντολών δημιουργίας αντικειμένων σε κώδικα εφαρμογής και το 57.43% στο σύνολο του κώδικα (συμπεριλαμβάνοντας και τις βιβλιοθήκες) της DaCapo benchmark suite δεν μπορούν να διαφύγουν, και άρα μπορούν με ασφάλεια να δημιουργηθούν στη στοίβα έναντι του σωρού.

Η κεντρική ιδέα είναι ότι για να διαφύγει ένα αντικείμενο θα πρέπει να είναι προσβάσιμο είτε από κάποιο στατικό πεδίο, είτε από κάποιο exception, ή από μία τοπική μεταβλητή κάποιας μεθόδου που μπορεί να καλέσει άμεσα την μέθοδο που δημιούργησε το εν λόγω αντικείμενο. Η ανάλυση χρειάστηκε περίπου 100 γραμμές κώδικα Datalog, το οποίο είναι ενδεικτικό των δυνατοτήτων της δηλωτικής προσέγγισης για στατική ανάλυση προγραμμάτων, η οποία μας επιτρέπει να ορίσουμε μόνο *πότε* διαφεύγει ένα αντικείμενο και όχι *πως* να τα υπολογίσουμε. Αυτό οδήγησε σε μία εκφραστική και μεστή υλοποίηση.

Τρέχοντας την ανάλυση για διαφορετικές επιλογές του τύπου των συμφραζομένων διαπιστώσαμε ότι η επιλογή αυτή δεν επηρεάζει ιδιαίτερα την ακρίβεια, αλλά είναι, ωστόσο, άκρως σημαντική για τον χρόνο εκτέλεσης.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Ανάλυση Διαφυγής Αντικειμένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Datalog, ανάλυση πλήρους προγράμματος, Java, ασφαλής δημοσίευση αντικειμένων, προσβασιμότητα αντικειμένων, συμφραζόμενα

Abstract

An object escapes when it can outlive the method that allocated it. We present a declarative whole-program escape analysis for Java, written entirely in Datalog, that over-approximates the escaped objects in a program. Our analysis is a part of the Doop framework [3] that provides it with a points-to analysis for a number of possible types of context.

Our analysis was able to identify 60.66% of the application heap allocation sites and 57.43% of all the allocation sites (i.e., including library code), by average, of the DaCapo benchmark programs as non-escaping, and thus safe candidates to be allocated on the stack.

The main intuition is that an object escapes if it is reachable through a static field, an exception, or a local variable of an immediate caller of the method that created it. The escape analysis required just about 100 lines of Datalog code, which clearly demonstrates the potency of the declarative approach for static analysis. This allowed us to focus on the definition of the escaped objects and leave their computation to the underlying Datalog engine, which resulted in a concise and expressive representation.

By running the escape analysis for a variety of possible contexts, we found that the choice of context has little effect on precision but is crucial for the execution time overhead. Specifically, the call-site-sensitive analyses take a long time to complete since they do not adequately prune the search space of object-to-object pointers when computing object reachability.

SUBJECT AREA: Escape Analysis

KEYWORDS: Datalog, whole-program analysis, Java, safe publication, object reachability, context-sensitivity

Acknowledgements

I am grateful to my supervisor, Prof. Yannis Smaragdakis, whose expertise, guidance, and patience were decisive for this work. His valuable insights and vast knowledge on the field of static analysis were crucial throughout the research for and writing of this thesis.

I would like to thank my colleagues, Katia Papakonstantinopoulou, Michael Sioutis, Panagiotis Liakos, and Yiannis Giannakopoulos for all their help, interest, and advice. Lastly, I am deeply indebted to my parents for their support.

Athens, March 30, 2012

Contents

List of Figures

List of Tables

Preface

This report is my master thesis for the conclusion of my postgraduate studies at the Department of Informatics & Telecommunications, University of Athens. It was developed as a part of the PADECL Project for the University of Athens, while conducting research with Prof. Yannis Smaragdakis on advanced program analysis using declarative languages.

This work is built on top of the Doop framework for pointer analysis using Datalog, and aims to demonstrate the capabilities of the Datalog language in expressing a variety of static analyses for bug-detection and identification of unsafe coding idioms.

Athens, March 30, 2012

Chapter 1

Introduction

An object is deemed to *escape* when it can outlive the method that allocated it. Escape analysis should not be confused with *thread-escape* analysis, where a heap object is said to escape when it can be accessed from more than one threads. In that sense, an object *escapes* from its original *thread of execution*, whereas in the traditional escape analysis an object escapes from the method where it was created.

Escape analysis can be used as a compiler optimization, where heap allocations for non-escaping objects can be converted to stack allocations. This is a common optimization in modern Java compilers [9], but is computed with simple lightweight *dataflow* analysis instead of the more heavy whole-program analysis approach. The first reason for this is the dynamic nature of the Java language itself. For one thing, in Java, classes and interfaces are loaded, linked, and initialized dynamically [13, Chapter 5]. The use of (method-based) jit-compilation makes it impossible to determine when will each method be eventually compiled (which usually happens when a certain per method *usage threshold* is reached [17]), or to reason about the corresponding state of the code that has been loaded thus far at each such point. Therefore, any static analysis based on incomplete knowledge may be rendered useless, and its results invalidated, at a point later on. Furthermore, the increase in precision when using whole-program analysis instead of a more simplistic approach (like dataflow analysis) is not yet considered significant enough to justify the additional complexity. Nevertheless, the performance gain has not been studied in practice and the actual increase in precision may change this conviction.

Escape analysis can also be used in the context of bug-detection. The most prominent case is no other than *safe publication* [18]. To publish an object safely one has to ensure that the `this` reference is not allowed to escape during construction, or else an object that is not yet fully constructed may become prematurely accessible externally. In such a case, the constructor would fail to preserve any invariants that would be elsewise enforced by the time the constructor returned.

While in the context of dynamic compilation, whole-program analysis cannot be easily employed (e.g., for stack-allocation optimizations), in tools such as IDE debuggers, where it is reasonable to assume that the entire codebase is available, a whole-program escape analysis would make perfect sense. In bug-detection, precision is of the utmost importance, and thus whole-program escape analysis is a very good candidate due to its high accuracy that minimizes false positives of escaped objects.

In summary, this work makes the following contributions:

- We show that we can succinctly express an escape analysis in Datalog. Our analysis is built on top of the Doop framework [3] that provides a collection of points-to analyses with varying contexts. Client analyses, like our own, make use of its macro-based system to decouple the choice of context from analysis code. Therefore, our escape analysis can run with any possible context supported by Doop and extended with any future client analysis for this framework.

- We test our analysis with the DaCapo benchmark programs to compare its scalability and precision to earlier work on this field. Our analysis was able to identify 60.66% of the application heap allocation sites and 57.43% of all the allocation sites (i.e., including library code), by average, as non-escaping, and thus safe candidates to be allocated on the stack. This unusually high precision should be attributed to whole-program analysis that enables the identification of difficult cases of stack-allocatable objects and also dispenses with oversimplifications that are mostly found in incremental approaches having to deal with incomplete program knowledge dynamically.

- We test various choices of context for escape analysis and compare them in terms of precision and time overhead. We find that there is little effect on precision but significant correlation between the relative time overhead and the choice of context. Specifically, the call-site-sensitive analyses do not perform well, timewise, since they do not adequately prune the search space of object-to-object pointers when computing object reachability.

- We discuss a technique that builds on the escape analysis to identify cases of unsafe publication.

The rest of the thesis is organized as follows: in Chapter 2 we give a background of points-to analysis in Datalog using the Doop framework. In Chapter 3, we present the escape analysis written in Datalog. Chapter 4 discusses a technique that builds on the escape analysis and identifies cases of unsafe publication. In Chapter 5, we present the evaluation of our analysis by testing it on the DaCapo benchmark suite, and compare several choices of context and their effect on performance. We describe related work in Chapter 6 and conclusions in Chapter 7.

Chapter 2

Background

Our escape analysis uses the Doop framework [3], which provides a collection of points-to analysis (e.g., context insensitive, call-site sensitive, object sensitive, type sensitive). However, client code that is built upon any such analysis, as in our case, can be entirely oblivious to the exact choice of context (which is specified at runtime) and expressed using a generic API.

2.1 Points-To Analysis in Datalog

Doop's primary defining feature is the use of Datalog for its analyses. Architecturally, however, an important aspect of Doop's performance is that it employs an *explicit* representation of relations (i.e., all tuples of a relation are represented as an explicit table, as in a database), instead of using Binary Decision Diagrams (BDDs), which have often been considered necessary for scalable points-to analysis [25, 24, 12, 11].

Doop uses a commercial Datalog engine, developed by LogicBlox Inc. This version of Datalog allows "stratified negation", that is, negated clauses, as long as the negation is not part of a recursive cycle. It also allows specifying that some relations are functions, that is, the variable space is partitioned into domain and range variables, and there is only one range value for each unique combination of values in domain variables.

Datalog is a great fit for the domain of program analysis and, as a consequence, has been extensively used both for low-level [19, 10, 25] and for high-level [5, 7] analyses. The essence of Datalog is its ability to define recursive relations. Mutual recursion is the source of all complexity in program analysis. For a standard example, the logic for computing a callgraph depends on having points-to information for pointer expressions, which, in turn, requires a callgraph. We can easily see such recursive definitions in points-to analysis alone. Consider, for instance, two relations, `AssignHeapAllocation(?heap, ?var)` and `Assign(?to, ?from)`. (We follow the Doop convention of capitalizing the first letter of relation names, while writing variable names in lower case and prefixing them with a question-mark.) The former relation represents all occurrences in the Java program of an instruction

“$a = newA();$” where a heap object is allocated and assigned to a variable. That is, a preprocessing step takes a Java program (in Doop this is in intermediate, bytecode, form) as input and produces the relation contents. A static abstraction of the heap object is captured in variable `?heap`—it can be concretely represented as, for example, a fully qualified class name and the allocation’s bytecode instruction index. Similarly, relation `Assign` contains an entry for each assignment between two Java program (reference) variables.

The mapping between the input Java program and the input relations is straightforward and purely syntactic. After this step, a simple pointer analysis can be expressed entirely in Datalog as a transitive closure computation:

```
1 VarPointsTo(?heap, ?var) <- AssignHeapAllocation(?heap, ?var).
2 VarPointsTo(?heap, ?to) <- Assign(?to, ?from), VarPointsTo(?heap, ?from).
```

The Datalog program consists of a series of *rules* that are used to establish facts about derived relations (such as `VarPointsTo`, which is the points-to relation, i.e., it links every program variable, `?var`, with every heap object abstraction, `?heap`, it can point to) from a conjunction of previously established facts. In the LB-Datalog syntax, the left arrow symbol (`<-`) separates the inferred fact (i.e., the *head* of the rule) from the previously established facts (i.e., the *body* of the rule).

For instance, line 2 above says that if, for some values of `?from`, `?to`, and `?heap`, `Assign(?to,?from)` and `VarPointsTo(?heap,?from)` are both true, then it can be inferred that `VarPointsTo(?heap,?to)` is true. Note the base case of the computation above (line 1), as well as the recursion in the definition of `VarPointsTo` (line 2).

The declarativeness of Datalog makes it attractive for specifying complex program analysis algorithms. Particularly important is the ability to specify recursive definitions—program analysis is fundamentally an amalgam of mutually recursive tasks. For instance, Doop uses mutually recursive definitions of points-to analysis and call-graph construction.

The key for a precise points-to analysis is context-sensitivity, which consists of qualifying program variables (and possibly object abstractions—in which case the context-sensitive analysis is said to also have a *context-sensitive heap*) with context information: the analysis collapses information (e.g., “what objects this method argument can point to”) over all possible executions that result in the same context, while separating all information for different contexts. Object-sensitivity and call-site-sensitivity are the main flavors of context sensitivity in modern points-to analyses. They differ in the contexts of a context, as well as in when contexts are created and updated. To gain insight into the aforementioned variations of context sensitivity, the interested reader is referred to [21, 14]. Here we will not yet concern ourselves with such differences—at this point, it suffices to know that a context-sensitive analysis qualifies its computed facts with extra information.

Context-sensitive analysis in Doop is, to a large extent, similar to the above context-insensitive logic. The main changes are due to the introduction of Datalog variables representing contexts for variables (and, in the case of a context-sensitive heap, also objects), in the analyzed program. For an illustrative example, the following two rules handle method calls as implicit assignments from the actual parameters of a method to the formal parameters, in a 1-context-sensitive analysis with a context-*insensitive* heap.
(This code is the same for both object-sensitivity and call-site-sensitivity.)

```
Assign(?calleeCtx, ?formal, ?callerCtx, ?actual) <-
  CallGraphEdge(?callerCtx, ?invocation, ?calleeCtx, ?method),
  FormalParam[?index, ?method] = ?formal,
  ActualParam[?index, ?invocation] = ?actual.

VarPointsTo(?heap, ?toCtx, ?to) <-
  Assign(?toCtx, ?to, ?fromCtx, ?from),
  VarPointsTo(?heap, ?fromCtx, ?from).
```

(Note that some of the above relations are functions, in which case the functional notation "`Relation[?domainvar] = ?val`" is used instead of the traditional relational notation, "`Relation(?domainvar, ?val)`". Semantically the two are equivalent, only the execution engine enforces the functional constraint and produces an error if a computation causes a function to have multiple range values for the same domain value.)

The example shows how a derived `Assign` relation (unlike the input relation `Assign` in the earlier basic example) is computed, based on the call-graph information, and then used in deriving a context-sensitive `VarPointsTo` relation.

For deeper contexts, one needs to add extra variables, since pure Datalog does not allow constructors and therefore cannot support value combination. Doop employs a macro system to hide the number of context elements so that such variations do not pollute the main analysis, as well as any client (like in our case, the escape analysis), logic.

```
//  If S is an ordinary (nonarray) class, then:
//     o If T is a class type, then S must be the
//       same class as T, or a subclass of T.
CheckCast(?s, ?s) <- ClassType(?s).
CheckCast(?s, ?t) <- Subclass(?t, ?s).
...
//     o If T is an array type TC[], that is, an array of components
//       of type TC, then one of the following must be true:
//          + TC and SC are the same primitive type
CheckCast(?s, ?t) <-
  ArrayType(?s), ArrayType(?t),
  ComponentType(?s, ?sc), ComponentType(?t, ?sc), PrimitiveType(?sc).

//          + TC and SC are reference types (2.4.6), and type SC can be
//            cast to TC by recursive application of these rules.
CheckCast(?s, ?t) <-
  ComponentType(?s, ?sc), ComponentType(?t, ?tc),
  ReferenceType(?sc), ReferenceType(?tc), CheckCast(?sc, ?tc).
```

Figure 2.1: Excerpt of Datalog code for Java cast checking, together with Java Language Specification text in comments. The rules are quite faithful to the specification.

Generally, the declarative nature of Doop often allows for very concise specifications of analyses. In [3], Martin Bravenboer and Yannis Smaragdakis demonstrate a striking example of the logic for the Java cast checking—i.e., the answer to the question “can type A be cast to type B?”. The Datalog rules are almost an exact transcription of the Java Language Specification. A small excerpt, with the Java Language Specification text included in comments, can be seen in Figure 2.1.

Chapter 3

Over-Approximating Escaped Objects

In this chapter we formalize the concept of escaped objects and describe the static analysis that computes them (or rather, over-approximates them).

3.1 The Immediate Callers Conjecture

Informally, an object *escapes* when it can “outlive” the method that allocated it (Figure 3.1). It is easier to formalize the notion of non-escaping (i.e., stack-allocatable) objects.

Definition 3.1.1. An object o allocated in method m *does not escape* if there is no possible execution of the program in which o may be contained in the set of *live objects* after the instance of m that allocated o has returned.

The set of live objects (at time t) is a common term in garbage collection, where it is used to denote the objects that are (transitively) reachable by pointer relationships from the *root set* (at time t). The root set in Java consists of the static variables, and the local variables in the activation stack. After garbage collection, the objects not contained in this set may safely be discarded.

Also notice that by m, we refer to the specific instance that allocated this particular object. There may be other instances of m higher up the call stack, that may have, in turn, allocated different objects with the same allocation instruction. However, in the context of

```
/* Object may escape, if it can outlive the method that allocated it */

MayEscape(?obj) -> HeapAllocationRef(?obj).

MayEscape(?obj) <-
  MayOutlive(?obj, ?inmethod), AssignHeapAllocation(?obj, _, ?inmethod).
```

Figure 3.1: Escape Definition

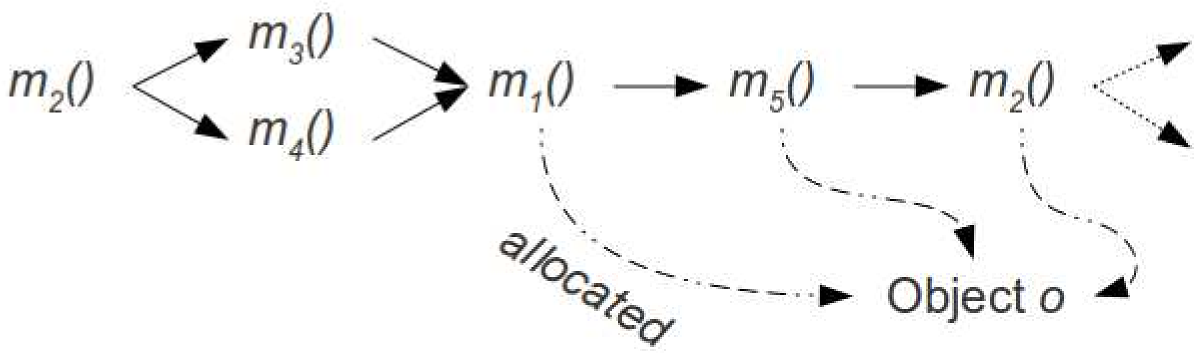


Figure 3.2: Cycle in which m_2 is both a *caller* and a *callee* of m_1, and thus may reach the object o even though it does not escape.

static analysis, we abstract heap objects via their allocation sites (and possibly some analysis-specific heap context) and compute escape information accordingly. Thus, we produce an over-approximation of escaping *heap allocation sites*. Our analysis is sound in the sense that any object allocated at a non-escaping allocation site will definitely not be in the set of live objects since the method call that performed its allocation has returned.

Conjecture 3.1.1. In the absence of exceptions and static variables, an object o allocated in method m_1 escapes only if there exists a method m_2 that may directly call m_1 and (transitively) reach o through its local variables.

That is, if an object is not reachable by any of the direct callers of the method that allocated it, it will not be reachable by any other indirect callers (i.e., all the methods that may constitute the call stack up to m_1) and therefore will not escape. [1] (Notice that other methods, such as those that can be called by m_1, may normally reach o even if it does not escape.)

[1]There is one exception to the reachability of indirect callers. If the call-graph contains cycles, it is possible that an object will be reachable by some of the indirect callers that may also be called by m_1 (and thus form a cycle) and unreachable by all of its direct callers. Even so, the instances that may indeed reach o will be lower on the call stack than the m_1 that allocated o, and thus, o will not escape. Such an example is depicted in Figure 3.2.

3.2 Object Reachability

In order to compute the escaped objects, we must first compute object reachability. Figure 3.3 corresponds to this part of the analysis. An object `?fromObj` points directly to another object `?toObj`, either through some field (lines 9-11), or through some array index (lines 6-7) in case `?fromObj` is of an array type. The `ArrayIndexPointsTo` and `InstanceFieldPointsTo` relations are precomputed by DOOP during the points-to analysis.

For example, an instruction "$a.f = b;$" will cause DOOP to produce the fact "`InstanceFieldPointsTo(HeapAbstraction(?toObj), ?field, HeapAbstraction(?fromObj))`", if variables "a" and "b" may point to heap objects `?fromObj` and `?toObj` respectively, and `?field` represents the field "f". The `HeapAbstraction` macro augments a heap object with context. The "`ArrayIndexPointsTo(HeapAbstraction(?toObj), HeapAbstraction(?fromObj))`" fact will be produced respectively for the instruction "$a[i] = b;$". Notice that, in the latter case, the produced fact will not contain any information about i (unlike the field case where f was recorded) since the analysis is array-insensitive.

Computing the transitive closure of object reachability is quite straightforward (lines 15-19). The compound term `ObjectMayReach(?fromObj, ?toObj)` denotes that there is a finite sequence of fields and array indices by which `?fromObj` may reach `?toObj`.

The `AnyHeapAbstraction` macro (lines 7, 11) states that we dispose of any heap context available at this point when computing the `ObjecPointsTo` relation. The transitive closure is then computed context-insensitively. Even so, the context sensitivity has played its part, since it reduces the size of the `ArrayIndexPointsTo` and `InstanceFieldPointsTo` relations with respect to the context-insensitive case. As it will later be evident in chapter 5, the size of `InstanceFieldPointsTo` is the bottleneck of the escape analysis, and therefore context-sensitivity, even in this limited form, is essential for the efficiency of the escape analysis. On the other hand, the full context-sensitive computation of object-reachability (that is, the retaining of heap context for the `ObjectMayReach` relation in a manner similar to the `ArrayIndexPointsTo` relation) would be prohibitive for any heap-sensitive analyses. We believe that this design choice achieves a good tradeoff between precision and performance.

3.3 Static Fields

Apart from local variables in the activation stack (i.e., the per-thread *Java virtual machine stack* that adds and removes frames each time a method is invoked and completes respec-

```
/* Object points-to another object */

ObjectPointsTo(?fromObj, ?toObj) ->
  HeapAllocationRef(?fromObj), HeapAllocationRef(?toObj).

ObjectPointsTo(?fromObj, ?toObj) <-
  ArrayIndexPointsTo(AnyHeapAbstraction(?toObj), AnyHeapAbstraction(?fromObj)).

ObjectPointsTo(?fromObj, ?toObj) <-
  InstanceFieldPointsTo(
     AnyHeapAbstraction(?toObj), _, AnyHeapAbstraction(?fromObj)).

/* Transitive Closure for object reachability */

ObjectMayReach(?fromObj, ?toObj) <-
  ObjectPointsTo(?fromObj, ?toObj).

ObjectMayReach(?fromObj, ?toObj) <-
  ObjectPointsTo(?fromObj, ?interm), ObjectMayReach(?interm, ?toObj).
```

Figure 3.3: Object Reachability

tively), the root set also contains the static variables. Therefore, any heap object reachable from such a variable at a given time must be included in the set of live objects. Since we lack flow-sensitivity, we choose to simply mark any object that can be reached by any static variable at any given time as escaping.

In Java, static variables can only be defined inside class/interface declarations. The Doop framework represents static variables as field signatures and defines that "`StaticFieldPointsTo(HeapAbstraction(?obj), ?field)`" holds if the static field represented by `?field` can point to the heap-sensitive object `?obj`. From that point, the manner of computing the objects reachable through static fields is a simple transitive closure computation on the aforementioned `ObjectPointsTo` relation (Figure 3.4).

3.4 Handling of Exceptions

Exceptions are the only way to transfer control from a method m_1 to a method m_2 that has not called m_1 directly, but instead has reached it through a non-empty sequence of intermediate method calls. In that case, the only part of m_1's state that is accessible at the point when the

```
/* Objects reachable through static field */

ReachableThroughStaticField(?obj) -> HeapAllocationRef(?obj).

ReachableThroughStaticField(?obj) <-
  StaticFieldPointsTo(AnyHeapAbstraction(?obj), _).

ReachableThroughStaticField(?toObj) <-
  ReachableThroughStaticField(?fromObj), ObjectPointsTo(?fromObj, ?toObj).
```

Figure 3.4: Objects reachable through static fields

```
/* Objects reachable by thrown exception */

ReachableByException(?obj) -> HeapAllocationRef(?obj).

ReachableByException(?obj) <-
  ThrowPointsTo(AnyHeapAbstraction(?obj), AnyContext(_)).

ReachableByException(?toObj) <-
  ReachableByException(?fromObj), ObjectPointsTo(?fromObj, ?toObj).
```

Figure 3.5: Objects reachable by an exception

exception is caught in m_2 is the thrown exception and anything reachable from it.

The handling of this case is analogous to that of Section 3.3, where the basis of the recursion is now the relation `ThrowPointsTo` (Figure 3.5). That is, we treat everything that can be reached by any exception as escaping. In Doop, "`ThrowPointsTo(HeapAbstraction(?exc), Context(?ctx, ?meth))`" holds if method `?meth` (with context `?ctx`) can throw an exception `?exc` (which is in fact an ordinary heap-sensitive object of the appropriate exception type).

3.5 Putting Everything Together

Figure 3.6 presents the main body of the code that constitutes the escape analysis.

```
/* Context-insensitive direct function calls */

Calls(?m1, ?m2) -> MethodSignatureRef(?m1), MethodSignatureRef(?m2).
Calls(_, ?m) -> Reachable(?m).

Calls(?fromMethod, ?toMethod) <-
  CallGraphEdge(AnyContext(?invocation), AnyContext(?toMethod)),
  Instruction:Method[?invocation] = ?fromMethod,
  Reachable(?fromMethod).

/* Method may reference object */

MethodsThatMayReference(?obj, ?meth) ->
  MethodSignatureRef(?meth), HeapAllocationRef(?obj).

MethodsThatMayReference(?obj, ?meth) <-
  VarPointsTo(AnyHeapAbstraction(?obj), AnyContext(?var)),
  Var:DeclaringMethod(?var, ?meth),
  Reachable(?meth).

/* Object may outlive a method */

MayOutlive(?obj, ?method) ->
  HeapAllocationRef(?obj), MethodSignatureRef(?method).

lang:derivationType[‘MayOutlive] = "Derived".

MayOutlive(?obj, ?method) <-
  ReachableThroughStaticField(?obj), Reachable(?method).

MayOutlive(?obj, ?method) <-
  ReachableByException(?obj), Reachable(?method).

MayOutlive(?obj, ?callee) <-
  Calls(?caller, ?callee), MethodsThatMayReference(?obj, ?caller).

MayOutlive(?toObj, ?callee) <-
  Calls(?caller, ?callee),
  MethodsThatMayReference(?fromObj, ?caller),
  ObjectMayReach(?fromObj, ?toObj).
```

Figure 3.6: Putting everything together

3.5.1 Synthesis

The `MethodsThatMayReference` relation contains an `(?obj, ?method)` pair if method `?meth` defines a variable `?var` which may point to object `?obj` (Figure 3.6, lines 13-19). The `Calls` relation (lines 3-9) represents a context-insensitive method-to-method call graph edge.

We are now able to define the `MayOutlive` relation (lines 23-40) to bring it all together. The `MayOutlive` relation is never computed exhaustively (as stated in line 26), but is instead inlined in `MayEscape` (Figure 3.1). Therefore, its arguments (e.g., `?toObj`, `?callee`) will be bound to a heap object and to the method that allocated it.

Lines 28-29 and 31-32 relate to Section 3.3 and Section 3.4 respectively. Lines 34-40 correspond to Conjecture 3.1.1, by stating that an object escapes if it is directly (lines 34-35) or indirectly (37-40) reachable by an immediate caller of the method that allocated it.

3.5.2 Optimizations

The use of the `Reachable` relation (lines 9, 19, 29, 32) is an optimization that limits the escape analysis to the reachable methods. Therefore, the objects that may escape are a subset of the reachable objects (i.e., the objects allocated in reachable methods). The constraint in line 4 acts as a simple sanity check.

Appendix A contains the entire code of the escape analysis presented in this chapter, including some additional optimizations specific to the LogicBlox engine [3] that are out of the scope of this thesis.

The fact that, in about 100 lines of code, we were able to express a powerful escape analysis shows how expressive Datalog is, and how appropriate an environment for developing static analyses. Furthermore, the design of Doop, with its variety of supporting contexts and its macro-based API, facilitates the generilization of any such client analysis. The escape analysis may be added on top of any points-to analysis, with no need for code changes.

Chapter 4

Safe Publication

A caveat for using escape analysis to identify pathological cases of unsafe construction is that we cannot use references to the original object that is being created (and to which the constructor's `this` reference points to), since that object will probably "leak" to a left-value of an assignment (with a `new` command as its right-value that implicitly calls a constructor). This problem can be circumvented by introducing an artificial per-constructor object and an additional `VarPointsTo` edge to it from the constructor's `this` reference. In this way we can insulate the via-constructor escaping from ordinary external instance creation commands by simply examining the lifetime of this auxiliary object. If the analysis reports that such an object escapes, then the corresponding constructor is unsafe, since the only way for the object to outlive it is by escaping while under construction.

In Figure 4.1 for instance, the class `SafelyConstructed` does not let the `this` reference escape during construction. This can be detected by creating an auxiliary heap object `HeapObject:Guard` and a `VarPointsTo` edge from the constructor's `this` variable to it. We then have to check only that this particular object, or any other auxiliary object for this class's constructors, does not escape (and indeed it doesn't) in order to characterize the class as safely constructed.

In Figure 4.2 the `this` reference gets written to a static field during construction. Thus, the auxiliary object escapes (Section 3.3) and unsafe construction is detected. If the constructor was supposed to enforce the invariant that the `msg` field is always uppercase, it now fails to do so by prematurely publishing the object under construction in a static field (that may be read too soon by another thread for example).

```java
public class SafelyConstructed {
  private final String msg;

  /** Constructor */
  public SafelyConstructed(String message) {
    //! this -> {HeapObject:1, HeapObject:Guard}
    init(message);
  }

  /** Helper method that performs some basic initialization */
  private final void init(msg) {
    //! this -> {HeapObject:1, HeapObject:Guard}
    this.msg = msg.toUpperCase();
  }

  public void main(String[] args) {
    Object obj = new SafelyConstructed("I’m safe"); //! obj -> HeapObject:1
  }
}
```

Figure 4.1: Safe Construction with `VarPointsTo` information in comments

```java
public class EscapingUnderConstruction {
  private String msg;
  public static EscapingUnderConstruction instance;

  /** Constructor */
  public EscapingUnderConstruction(String message) {
    this.msg = message;
    init();
  }

  /** Helper method that performs some basic initialization */
  private final void init() {
    instance = this; //! auxiliary object escapes
    this.msg = msg.toUpperCase();
  }
}
```

Figure 4.2: Unsafe Construction

Chapter 5

Experimental Results

This chapter presents the evaluation of the escape analysis on a well-known benchmark suite, and the experimental results.

5.1 Setup

We use a 64-bit machine with a quad-core Xeon E5530 2.4GHz CPU (only one thread was active at a time). The machine has 24GB of RAM.

We analyzed the DaCapo benchmark programs, v.2006-10-MR2, with JDK 1.4. These benchmarks are the largest in the literature on context-sensitive points-to analysis. We concentrated on a subset of the DaCapo benchmarks, namely the *antlr*, *chart*, *eclipse*, *luindex*, and *pmd*, all of which can be successfully analyzed by the DOOP framework with reflection-analysis enabled.

5.2 Evaluation

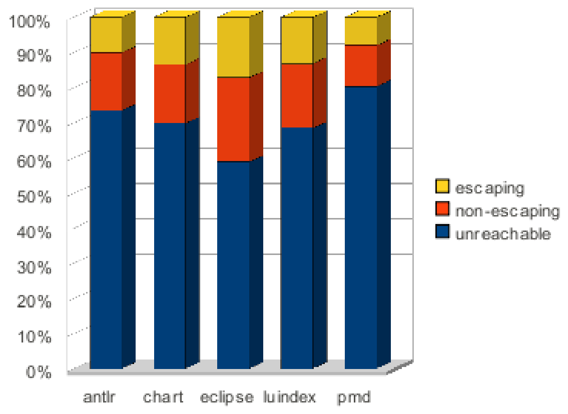
Table 5.1 presents the time overhead and the number of escaped objects reported by our analysis for each benchmark. The "time overhead" is the additional time required by the escape analysis, whereas the "total time" is the sum of the escape analysis time plus the time of the basic analysis as performed by DOOP.

The rest is the total heap allocations ("allocations"), the heap allocations in reachable code ("reachable"), and the escaping heap allocations ("escaping"). The "stack-allocatable" percentage is computed as the fraction of the reachable allocations that do not escape (Section 3.5.2). Moreover, we also measure the escaped/reachable objects in application code (i.e., not including library code).

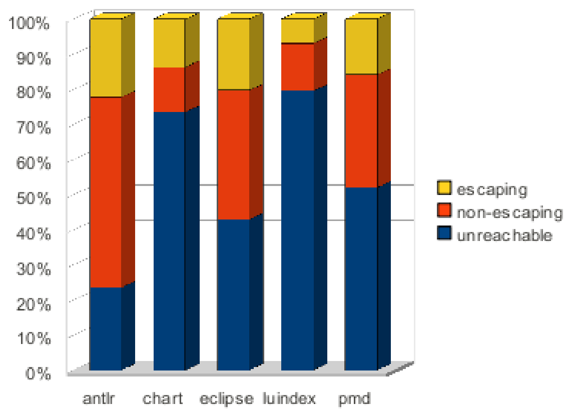
The execution time overhead is significant (8-24%) but anticipated since it involves the semi-expensive computation of the transitive closure of object reachability (which may be useful in other contexts as well in the future).

		1-object-sensitive				
	benchmark	antlr	chart	eclipse	luindex	pmd
	total time	179.45s	427.39s	269.03s	86.87s	165.14s
	time overhead	28.61s	50.41s	62.19s	13.27s	30.33s
	time overhead (%)	15.94%	11.79%	23.12%	15.28%	18.37%
app + lib	allocations	41155	48694	24414	24481	45551
	reachable	10880	14695	10046	7707	8945
	escaping	4118	6544	4193	3218	3607
	stack-allocatable	62.15%	55.47%	58.26%	58.25%	59.68%
app only	allocations	4990	6106	4166	3052	3856
	reachable	3815	1613	2377	623	1851
	escaping	1108	835	842	213	609
	stack-allocatable	70.96%	48.23%	64.58%	65.81%	67.10%
		2-type-sensitive+heap				
	benchmark	antlr	chart	eclipse	luindex	pmd
	total time	170.46s	278.66s	449.60s	104.22s	166.36s
	time overhead	39.44s	42.87s	39.36s	18.91s	33.61s
	time overhead (%)	23.14%	15.38%	8.75%	18.14%	20.20%
app + lib	allocations	41155	48694	24414	24481	45551
	reachable	10791	14538	9740	7591	8753
	escaping	4676	6691	4279	3321	3721
	stack-allocatable	56.67%	53.98%	56.07%	56.25%	57.49%
app only	allocations	4990	6106	4166	3052	3856
	reachable	3813	1610	2355	593	1744
	escaping	1576	862	956	228	631
	stack-allocatable	58.67%	46.46%	59.41%	61.55%	63.82%

Table 5.1: Escaping Objects and Analysis Time



(a) Application + Library Code



(b) Application Code Only

Figure 5.1: Allocations in 1-Object-Sensitive Analyses

	total time	time overhead		escaping		escaping (app)	
1obj	179.45s	28.61s	(15.94%)	4118	(37.85%)	1108	(29.04%)
1obj+H	521.35s	45.82s	(8.79%)	4115	(37.88%)	1108	(29.04%)
1call+H	1161.55s	740.07s	(63.71%)	4108	(37.88%)	1133	(29.70%)
2obj	1142.00s	67.36s	(5.90%)	4116	(37.83%)	1108	(29.04%)
2type+1H	170.46s	39.44s	(23.14%)	4676	(43.33%)	1576	(41.33%)
2full+1H	231.71s	24.90s	(10.75%)	4102	(38.03%)	1108	(29.06%)
2obj+H	370.65s	46.36s	(12.51%)	4103	(38.02%)	1108	(29.04%)
1type1obj+1H	198.50s	24.55s	(12.37%)	4102	(38.02%)	1108	(29.06%)

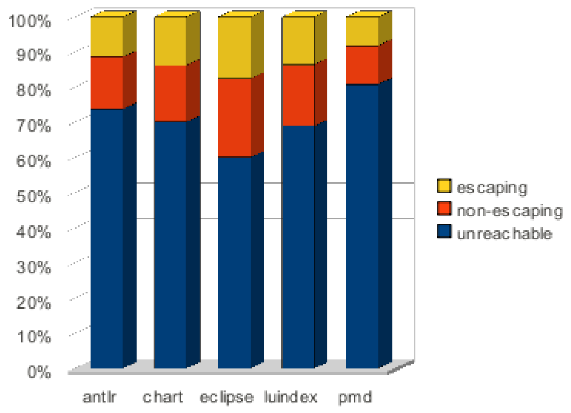
Table 5.2: Precision and Execution Time for Antlr

Figure 5.1, and Figure 5.2 depict the 1-object-sensitive and 2-type-sensitive+heap allocation results respectively, in percent stack charts. The percentage of the reachable objects that can be safely allocated in the stack ranges from 46% to 70%. The fraction of non-escaping objects is almost always higher in application code, with an average of 60.66% as opposed to the 57.43% average when including library code.

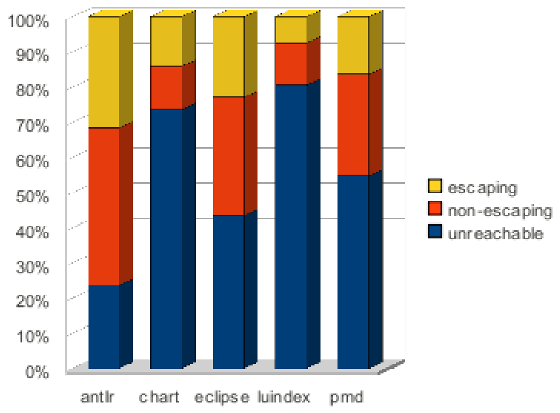
Generally speaking, the results are quite encouraging when contrasted to earlier escape analysis literature [4, 26, 1]. The gain in precision comes from the use of whole-program analysis that enables the identification of more difficult non-escaping object cases.

Table 5.2 presents the precision and execution time overhead on the *anltr* benchmark, for several types of analyses (with different types of contexts) supported by DOOP.

There was an upper bound for the execution time of the escape analysis equal to two times



(a) Application + Library Code



(b) Application Code Only

Figure 5.2: Allocations in 2-Type-Sensitive+Heap Analyses

that of the basic analysis, after which the escape analysis was terminated before completion. That happened only on the context-insensitive and 1-call-site-sensitive analyses.

The reason for these timeouts is that the execution time of the escape analysis is dominated by the computation of object reachability, which in turn depends heavily on the size of the `InstanceFieldPointsTo` relation (since the size of `ArrayIndexPointsTo` is relatively small in most cases). Therefore, call-site-sensitive and context-insensitive analyses are bad candidates for escape analysis since they produce a large `InstanceFieldPointsTo` relation (which also explains the large overhead of the 1call+H analysis).

That is why context-sensitivity is critical (as noted in Section 3.2), even if contexts are discarded on a later stage when computing the transitive closure of object reachability. It prunes the size of `InstanceFieldPointsTo` early on, as to allow a fast transitive closure computation.

As for precision, the choice of context does not seem that important. With the exception of the 2type+1H analysis, there is small variation on the percentage of escaped objects on the range of 37.83-38.03% for application plus library code, and 29.04-29.70% for just application code.

Chapter 6

Related Work

Escape analysis for Java has been well studied in the past [4, 26, 1, 2, 6, 23, 8], often in conjunction with thread-escape analysis and synchronization elimination [25, 20, 22, 16, 15].

However, none of the analyses listed above are expressed in Datalog (with the exception of static race and deadlock detection that contain only a *thread-escape* analysis [16, 15]) but are instead formalized using dataflow algorithms (e.g., [4]) or similar means, and are often summary-based.

By using Datalog for static whole-program escape analysis we are able to obtain an expressive, concise, and scalable algorithm, while our analysis can be easily employed by any other part of the Doop framework and extended in any possible way with little or no effort. On the other hand, more ad hoc solutions like earlier work on this field lack this generality and extensibility.

The analysis in [4] is based on connection graphs that represent the points-to information but can be easier summarized to avoid recomputing the escape information when a method is called in different escape contexts. John Whaley and Martin Rinard present a combined pointer and escape analysis algorithm for object-oriented programs, designed to analyze arbitrary parts of complete or incomplete programs, obtaining complete information for objects that do not escape the analyzed parts [26]. Blanchet uses integers to represent type heights that encode how an object of one type can have references to other objects or is a subtype of another object [1]. The escaping part of an object is represented by the height of its type. He proposes a two-phase (a backward phase and a forward phase) flow-insensitive analysis for computing escape information. He uses escape analysis for both stack allocation and synchronization elimination. In contrast, our work achieves better results, by average, with much smaller variation for different benchmark programs than any of the aforementioned techniques [4, 26, 1]. The great divergence in the reported results of earlier work may imply that the benchmarks used for evaluation were either small or not clearly representative of the Java language.

The constraint-based, flow-insensitive, context-sensitive analysis of Bogda and Hölzle is essentially a whole-program analysis, but is applied to synchronization elimination rather than stack allocation [2].

Gay et al. provide an algorithm that is linear in the size of the program plus the size of the static call graph [6]. They focus on speed rather than precision and thereby assume that any reference assigned to a field escapes (and thus fail to identify cases of stack allocation of objects referenced in fields of other stack allocated objects).

Frédéric Vivien and Martin Rinard have followed an incremental approach instead of whole-program analysis, which concentrates only on the parts of the program that may deliver useful results [23]. Another scalar approach that combines an intraprocedural and an interprocedural analysis, well-suited to the needs of a dynamic compiler which lacks a concrete view of the complete program, is presented by Thomas Kotzmann and Hanspeter Mössenböck [8].

While incremental approaches may indeed be more appropriate in the context of dynamic compilation, they suffer from imprecision due to their incomplete knowledge that leads to several oversimplifications (e.g., anything assigned to a field or passed to a method escapes). This may be unavoidable for a JVM that has to perform the loading and linking of classes dynamically. In other cases, however, such as in debugging tools where the accuracy of the reported results is of the utmost importance and the entire code is almost always available, whole-program analysis is much more promising.

Chapter 7

Conclusions

By using Datalog we were able to succinctly express a declarative whole-program escape analysis for Java, that was able to identify 60.66% of the application heap allocation sites and 57.43% of all the allocation sites (i.e., including library code), by average, of the DaCapo benchmark programs as non-escaping, and thus safe candidates to be allocated on the stack.

The escape analysis, in its final optimized version, required just about 100 lines of Datalog code, which clearly demonstrates the potency of the declarative approach for static analysis. This allowed us to focus on the definition of the escaped objects and leave their computation to the underlying Datalog engine, which resulted in a concise and expressive representation.

The escape analysis was built on top of the Doop framework [3] which allowed the convenient decoupling of the choice of context from the escape analysis code. By analyzing *antlr* for a variety of possible contexts, we found that there is little effect on precision but significant correlation between the relative time overhead and the choice of context. Specifically, the call-site-sensitive analyses do not perform well, timewise, since they do not adequately prune the search space of object-to-object pointers when computing object reachability.

Our resulting algorithm is scalable and extensible in such a way that it can, almost effortlessly, become a part of future client analyses for the Doop framework. Previous work on the field has focused on ad hoc solutions that were not able to provide the same levels of precision. We believe that in any context where it is reasonable to assume that the largest part of the codebase is available, our whole-program escape analysis is an efficient and highly-accurate candidate.

Acronyms and Abbreviations

Abbreviation	Full Name
1obj	1-object-sensitive analysis
1obj+H	1-object-sensitive+heap analysis
1call+H	1-call-site-sensitive+heap analysis
2obj	2-object-sensitive analysis
2type+1H	2-type-sensitive+heap analysis
2full+1H	2-full-object-sensitive+heap analysis
2obj+H	2-object-sensitive+heap analysis
1type1obj+1H	2-full-type-object-sensitive+heap analysis
JVM	Java Virtual Machine
JLS	Java Language Specification
LB	LogicBlox Inc.

Appendix A

Escape Analysis Code

```
#include "macros.logic"

/* Context-insensitive direct function calls */

Calls(?m1, ?m2) -> MethodSignatureRef(?m1), MethodSignatureRef(?m2).
Calls(_, ?m) -> Reachable(?m).

Calls(?fromMethod, ?toMethod) <-
  CallGraphEdge(AnyContext(?invocation), AnyContext(?toMethod)),
  Instruction:Method[?invocation] = ?fromMethod,
  Reachable(?fromMethod).

/* Method may reference object */

MethodsThatMayReference(?obj, ?method) ->
  MethodSignatureRef(?method), HeapAllocationRef(?obj).

MethodsThatMayReference(?obj, ?method) <-
  VarPointsTo(AnyHeapAbstraction(?obj), AnyContext(?var)),
  Var:DeclaringMethod(?var, ?method),
  Reachable(?method).

/* Object points-to another object */

ObjectPointsTo(?fromObj, ?toObj) ->
  HeapAllocationRef(?fromObj), HeapAllocationRef(?toObj).

ObjectPointsTo(?fromObj, ?toObj) <-
  ArrayIndexPointsTo(AnyHeapAbstraction(?toObj), AnyHeapAbstraction(?fromObj)).

ObjectPointsTo(?fromObj, ?toObj) <-
  InstanceFieldPointsTo(
    AnyHeapAbstraction(?toObj), _, AnyHeapAbstraction(?fromObj)).

/* Transitive Closure for object reachability */

ObjectMayReach(?fromObj, ?toObj) <-
  ObjectPointsTo(?fromObj, ?toObj).

ObjectMayReach(?fromObj, ?toObj) <-
  ObjectPointsTo(?fromObj, ?interm), ObjectMayReach(?interm, ?toObj).

/* Optimization */

ObjectIsPointedBy(?toObj, ?fromObj) <-
  ObjectPointsTo(?fromObj, ?toObj).

/* Objects reachable through static field */

ReachableThroughStaticField(?obj) -> HeapAllocationRef(?obj).

ReachableThroughStaticField(?obj) <-
  StaticFieldPointsTo(AnyHeapAbstraction(?obj), _).

ReachableThroughStaticField(?toObj) <-
  ReachableThroughStaticField(?fromObj), ObjectIsPointedBy(?toObj, ?fromObj).

/* Objects reachable by thrown exception */

ReachableByException(?obj) -> HeapAllocationRef(?obj).

ReachableByException(?obj) <-
  ThrowPointsTo(AnyHeapAbstraction(?obj), AnyContext(_)).

ReachableByException(?toObj) <-
  ReachableByException(?fromObj), ObjectIsPointedBy(?toObj, ?fromObj).

/* Object may outlive a method */

MayOutlive(?obj, ?method) ->
  HeapAllocationRef(?obj), MethodSignatureRef(?method).

lang:derivationType[‘MayOutlive] = "Derived".
```

```

MayOutlive(?obj, ?method) <-
  ReachableThroughStaticField(?obj), Reachable(?method).

MayOutlive(?obj, ?method) <-
  ReachableByException(?obj), Reachable(?method).

MayOutlive(?obj, ?callee) <-
  Calls(?caller, ?callee), MethodsThatMayReference(?obj, ?caller).

MayOutlive(?toObj, ?callee) <-
  Calls(?caller, ?callee),
  MethodsThatMayReference(?fromObj, ?caller),
  ObjectMayReach(?fromObj, ?toObj).

/* Object may escape, if it can outlive the method that allocated it */

MayEscape(?obj) -> HeapAllocationRef(?obj).

MayEscape(?obj) <-
  MayOutlive(?obj, ?inmethod), AssignHeapAllocation(?obj, _, ?inmethod).
```

References

[1] Bruno Blanchet. Escape analysis for object-oriented languages: application to Java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 20–34, New York, NY, USA, 1999. ACM.

[2] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 35–46, New York, NY, USA, 1999. ACM.

[3] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA '09: 24th annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 2009. ACM.

[4] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 1–19, New York, NY, USA, 1999. ACM.

[5] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *ICSE '08: Proc. of the 30th int. conf. on Software engineering*, pages 391–400, New York, NY, USA, 2008. ACM.

[6] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, pages 82–93, London, UK, UK, 2000. Springer-Verlag.

[7] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with Datalog. In *Proc. European Conf. on Object-Oriented Programming (ECOOP)*, pages 2–27. Spinger, 2006.

[8] Thomas Kotzmann and Hanspeter Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 111–120, New York, NY, USA, 2005. ACM.

[9] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot^{TM} client compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008.

[10] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *PODS '05: Proc. of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, New York, NY, USA, 2005. ACM.

[11] Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, January 2006.

[12] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):1–53, 2008.

[13] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[14] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[15] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[16] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 386–396, Washington, DC, USA, 2009. IEEE Computer Society.

[17] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot^{TM} server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[18] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. Publication and escape. In *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[19] Thomas Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 163–196. Kluwer Academic Publishers, 1994.

[20] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, PPoPP '01, pages 12–23, New York, NY, USA, 2001. ACM.

[21] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. Pick your contexts well: Understanding object-sensitivity (the making of a precise and scalable pointer analysis). In *POPL '11: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2011. ACM.

[22] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In *Proceedings of the 2nd international symposium on Memory management*, ISMM '00, pages 18–24, New York, NY, USA, 2000. ACM.

[23] Frédéric Vivien and Martin Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 35–46, New York, NY, USA, 2001. ACM.

[24] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2005.

[25] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proc. of the ACM SIGPLAN 2004 conf. on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM.

[26] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. *SIGPLAN Not.*, 34(10):187–206, October 1999.