Interprocedural Heap Analysis using Access Graphs and Value Contexts

Dissertation

Submitted in partial fulfillment of the requirements
for the degree of

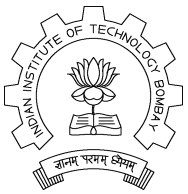
Master of Technology

by

Rohan Padhye
Roll No: 113050017

under the guidance of

Prof. Uday Khedker



Department of Computer Science and Engineering
Indian Institute of Technology Bombay

2013

Acknowledgement

I am extremely grateful to my advisor Prof. Uday Khedker for his invaluable guidance and relentless support throughout the duration of my project. I would like to thank Prof. Alan Mycroft from the University of Cambridge for providing interesting insights into the abstractions and proofs presented in this dissertation. I would also like to thank all the members of the GCC Resource Center at IIT Bombay for the regular enlightening discussions that helped shape several portions of this work. Finally, I would thank my parents for their continuous support and positive encouragement which strongly motivated me to give my absolute best to this project.

Rohan Padhye

July 5, 2013

Approval Sheet

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Rohan Padhye (Roll No: 113050017)

July 5, 2013

Abstract

Emperical studies have shown that interprocedural heap liveness analysis can be used to significantly improve garbage collection, thereby reducing memory consumption of programs. The main difficulty in achieving this goal is the unbounded nature of the heap, which makes it notoriously difficult to statically analyze using finite abstractions. A recent technique known as heap reference analysis can discover sets of live access paths using bounded access graphs, which can be used to `null`ify dead references or augment the garbage collector for improved precision. However, this analysis requires information about possibly aliased access paths for correctness.

Classical approaches to heap alias analysis provide sound but imprecise results, due to extensive summarizations performed for keeping the representation finite. This imprecision introduces a large amount of redundant live access paths if used by the liveness analysis directly, thereby diminishing the gains of liveness-based GC.

This project makes the following major contributions: (1) a liveness-driven heap abstraction is developed which is as expressible as required by the access graphs and thus can be used to answer alias queries precisely, (2) a technique called "dynamic heap pruning" is introduced, which uses the resulting access graphs to perform memory optimizations at run-time via a debugger API, (3) a generic interprocedural analysis framework is implemented for Soot, which is a popular toolkit for Java program analysis and (4) experimental results are provided for an instance of this framework that performs a flow and context-sensitive points-to analysis to resolve virtual method invocations and constructs the call graph on-the-fly; the resulting call graphs have been found to be much more precise than those that Soot provides by default.

Contents

Chapter 1

Introduction

Heap-allocated storage is a common feature of almost every programming environment used today. The characteristic feature of such dynamic memory is that allocation can be done on-demand during program execution. This feature also results in a very important problem for the run time system, that of memory mangement.

Some programming environments such as the C language leave memory management entirely to the programmer, relying on explicit deallocation of heap-allocated objects when no longer required. However, this can lead to *memory leaks* if a programmer does not free memory that is no longer required and loses all pointers to such an object. Many other environments such as the Java platform as well as most scripting languages provide automatic garbage collection. Although the periodic collection cycles induce a performance penalty, the programmer is relieved from worrying about the hassles of memory management.

While there exist several strategies for garbage collection, almost all of them use the concept of reachability to distinguish garbage from "live" objects [11]. In this definition, an object is live if it is pointed-to by a local or global variable or a field of a live object. That is, as long as a path of references exist from a named variable to a particular object, it will not be garbage collected. However, this definition does not capture the true notion of "liveness", which is whether an object will be used in the future. Thus, unnecessary memory is still held by the program if there are references to an object which is not needed. The time between the last use of an object and the point at which it is freed is known as *GC drag time*.

Static program analysis can potentially determine the set of live objects at a every program point and use that information in the following ways:

1. Warn the author of a C-like program about possible memory leaks.

2. Insert `null` assignments to dead references in a Java-like program, potentially reducing the GC drag time.

3. Augment a garbage collector to free all objects that will never be used in the future, even if references exist to them.

Although it is undecidable in general to precisely predict the last use of an object, a fixed-point data flow analysis can safely over-approximate the set of live objects, relying on reachability-based garbage collection as a fall-back for those objects that are spuriously marked live.

Emperical studies show that the precision of garbage collection will improve significantly from an interprocedural heap liveness analysis [9, 23], resulting in up to 39% reduction in peak memory utilzation for some standard benchmarks. Unfortunately, existing liveness analysis algorithms work only for named variables (stack + globals), while ignoring the heap or conservatively summarizing its liveness. The authors of [9] identify the key problem to be able to determine perfect pointer aliases within the heap, which this work tries to address. The difficulty with statically analyzing the heap arises due to the unbounded nature of its size, resulting from recursive data structures. Any static analysis that aims to discover properties of heap data must use a bounded approximation.

This project builds upon *heap reference analysis* [15], which is a backward data flow analysis that computes the (potentially infinite) set of live access paths at each program point by using bounded representations known as access graphs. References are `null`ified where possible, thereby reducing drag time. This analysis requires *may-alias* information for soundness, and *must-alias* information for improved precision. Although the original paper describes how to use the alias information, it does not present an algorithm to discover the aliases themselves.

In this project, we develop a technique for answering alias queries precisely by building an abstract points-to graph of the heap in a forward data flow analysis. The proposed approach is more precise than existing methods, in that the heap abstraction is designed to be as expressive as required by the heap liveness analysis, while still maintaing a finite bound. Thus, the two analyses are inter-dependent, similar to [14].

Another issue with performing precise heap liveness analysis is related to propagating information across procedure calls. Emperical results have shown that for the liveness analysis to be useful, it *must* incorporate inter-procedural program flow [9]. We have developed a generic inter-procedural analysis framework for performing heap analyses in Java using the concept of value contexts.

The main original contributions in this project are as follows:

1. A proposed liveness-driven heap abstraction for precise alias analysis, along with data flow equations for the liveness-driven heap analysis.

2. A generic access graph library implemented in Java for use by any analysis based on the bounded access graph representation.

3. A generic inter-procedural analysis framework using the concept of value contexts.

4. A technique for dynamically pruning heap references at run-time using the Java Debug Interface.

5. An implementation of points-to analysis in Soot [25] using the inter-procedural framework that builds context-sensitive call graphs on-the-fly (with experimental results).

The rest of this dissertation is organized as follows: Chapter 2 gives a quick recap of heap liveness analysis using access graphs and motivates the need for a precise alias analysis as well as a context-sensitive inter-procedural analysis framework. Chapter 3 details the proposed liveness-driven abstract heap representation and develops the required data flow analysis. Chapter 5 describes the concept of inter-procedural analysis using value contexts with an algorithm and example. Chapter 6 enumerates the implemented components of this project. Related work that is similar in approach or has similar goals is outlined in Chapter 7. Chapter 8 concludes the dissertation along with remarks on the current state of implementation and suggested future work.

Chapter 2

Background and Motivation

2.1 Heap Liveness Analysis using Access Graphs

The static analysis technique of [15] addresses the issue of heap liveness by looking not at the liveness of individual objects, but of access paths. For example, if the access path $x \rightarrow l \rightarrow r$ is live at a program point, then the pointers x, $x \rightarrow l$ and $x \rightarrow l \rightarrow r$ are likely to be dereferenced in the future, and thus the objects pointed to by them are live. However, as the number of live access paths at a program point can be potentially infinite (due to the presence of loops or recursion), the analysis models liveness in the form of *access graphs*. This bounded representation is equivalent to a pattern expressed in a regular language, which encompasses a potentially infinite number of access paths.

Figure 2.1 (a) shows a program that traverses a binary tree. The `while` loop traverses the left child nodes of the tree zero or more number of times starting at the root, after which the right child node is dereferenced once and its data is printed. The live access graph at the statement S_2 is shown in Figure 2.1 (b). This graph expresses the fact that the access pattern $x\ (\rightarrow l)* \rightarrow r$ will be dereferenced in the future[1]. If this information can be communicated to a garbage collector, then it will mark only the objects accessible by this access graph as live, while freeing the other nodes of the tree. This is shown in Figure 2.1 (c). Notice that the amount of garbage collected at S_2 is significantly more than a traditional reachability-based strategy in which the whole tree will be retained in memory as long as there exists a pointer to the root node.

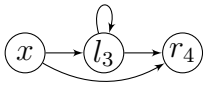
Definition 1. An **access graph** is a connected directed graph which is either empty (denoted by ε) or has a single root node (without in-edges) along with zero or more other nodes having unique labels of the form n_i where n is a field name and i is a program point.

Definition 2. A **live map** L is a mapping of variables to their access graphs. Thus, the access graph of variable x in L is given by $L(x)$. In graphical representations, live maps are shown by displaying only non-empty access graphs, whose root nodes are labelled by the name of the variable to which the access graph belongs, as shown in Figure 2.1 (b).

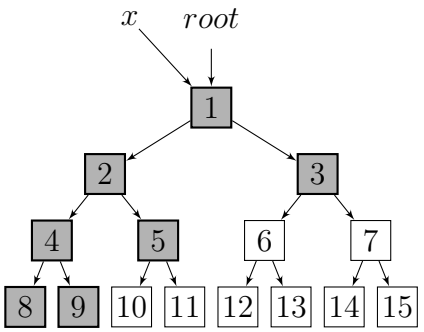
[1]The suffixes in the labels of the access graph nodes are used to distinguish between accesses at distinct program points, which helps in distinguishing loops from sequential accesses of the same field.

```
S1: x = root;
S2: while (x.val > M):
S3:    x = x.l;
S4: x = x.r;
S5: print x.val;
S6: exit;
```

(a) A tree traversing program.



(b) Access graph for x at S_2.



(c) The binary tree in the heap at S_2. Filled nodes are live objects.

Figure 2.1: Example usage of heap liveness graphs. If garbage collection is performed at statement S_2, then all nodes not accessible by traversing the access graph can be deleted.

Definition 3. A node in an access graph is called an **accessor**. The notation x/n_i is used to uniquely identify the accessor node having label n_i in the access graph of x. The symbol x is overloaded to refer to the root node of the access graph of x.

Definition 4. The **access pattern** of an accessor node is the regular expression for the language which starts with the name of its variable and is followed by a sequence of field names that are accepted by a finite state automaton having the same nodes as its access graph, with the root node as the *start* state, the accessor node as the *final* state and the state transitions as the edges in the access graph labelled by the field name of their target node. An access pattern is *live* if at least one object that is accessible along an access path expressed by the pattern is live.

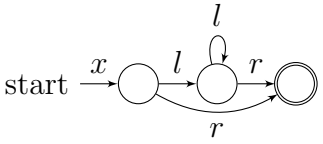
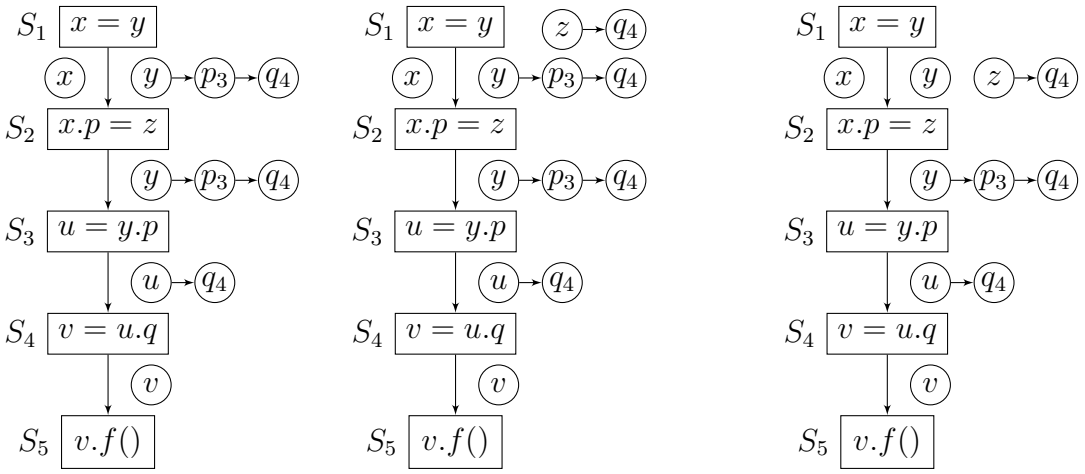


Figure 2.2: Finite state automaton for access pattern of x/r_4 in access graph of Figure 2.1 (b). The pattern is the regular expression $x\ (\rightarrow l)* \rightarrow r$.

If L is a live map at a program point, then the access patterns of each accessor in L are considered live indicating that they will likely be dereferenced in the future. The predicate $live_L(\alpha)$ is true if the access pattern α is live in L. Trivially, the predicate $live_L(x)$ is true if the graph of x is not empty. Note that the exact pattern α may not correspond to any accessor node in L, but it may still be live if there exists a pattern β of an accessor node which is live in L and $\alpha \cap \beta \neq \emptyset$ (i.e. there is an access path common to both patterns). The notations $pred_L(a)$ and $succ_L(a)$ may be used to denote the predecessors and successors of an accessor node a in the live map L respectively.



(a) Alias analysis is not performed.

(b) Alias analysis finds x and y *may-alias* at S_2.

(c) Alias analysis finds x and y *must-alias* at S_2.

Figure 2.3: Example depicting how *may-alias* information is needed for sound liveness analysis, and how *must-alias* information can help improve precision.

2.2 Need for Alias Analysis

Observe the program in Figure 2.3 (a), and the results of a naive liveness analysis. At the exit of statement S_2, which contains the instruction $x.p = z$, the access path $y \rightarrow p \rightarrow q$ is live. But x and z refer to the same object due to statement S_1 and hence the statement S_2 is equivalent to $y.p = z$. Hence, due to the indirect transfer of live access paths, z should be marked live at the entry to S_2 and its accessor should be suffixed with the suffix of y/p_3. However, if alias analysis is not performed then this indirect transfer is not done and z is not marked as live because the algorithm only considers the access graph of x (which is empty at the exit of S_2). A liveness-based garbage collector using the resulting access graphs may free the object referenced by z, resulting in an exception when it is dereferenced at S_4 via u.

Assuming an alias analysis was performed which conservatively provided the information that x and y may-alias (denoted $x \overset{may}{=} y$), then the effect of implicit liveness can be taken into account as seen in Figure 2.3 (b), where $z \rightarrow q$ is correctly marked live, resulting in no exception at either statement S_4 or S_5. However, the analysis fails to mark $y \rightarrow p$ as dead even though it is going to be implicitly reassigned at S_2 without any prior use. This *spurious* liveness is not unsafe, but reduces the gains of liveness-based GC.

Ideally, the alias analysis would determine that x must-alias y (denoted $x \overset{must}{=} y$), and that $y \rightarrow p \rightarrow q$ need not remain live, as the KILL to $x \rightarrow p$ ($\rightarrow *$) in statement S_2 implicitly also kills $y \rightarrow p$ ($\rightarrow *$). This is called a *strong update*. Figure 2.3 (c) shows the more precise liveness access graphs, which would result in improved garbage collection.

2.3 Issues in Interprocedural Analysis

For any heap analysis to be effective, it must be performed in an inter-procedural manner. This is because real-world programs often pass around pointers to heap-allocated objects as arguments when calling a procedure, which can modify the objects in any way.

Several approaches to inter-procedural analysis can be found in the literature. These are usually classified into context-sensitive and context-insensitive techniques. A context-insensitive analysis does not distinguish between distinct calls to a procedure. This causes the propagation of data flow values across interprocedurally invalid paths (i.e. paths in which calls and returns may not match) resulting in a loss of precision. A context-sensitive analysis restricts the propagation to valid paths and hence is more precise.

Two most general methods of precise flow and context-sensitive analysis are the *Functional* approach and the *Call Strings* approach [24]. The functional approach constructs summary flow functions for procedures by reducing compositions and meets of flow functions of individual statements to a single flow function, which is used directly in call statements. However, constructing summary flow functions may not be possible in general. The tabulation method of the functional approach overcomes this restriction by enumerating the functions as pairs of input-output data flow values for each procedure, but requires a finite lattice. The call strings method remembers calling contexts in terms of unfinished calls as call strings. However, it requires an exponentially large number of call strings. The technique of value based termination of call string construction [13] uses data flow values to restrict the combinatorial explosion of contexts and improves the efficiency significantly without any loss of precision.

A popular approach to interprocedural analysis uses the concept of graph reachability [19, 20], and is a special case of the functional approach. Formally, it requires flow functions $2^A \mapsto 2^A$ to distribute over the meet operation so that they can be decomposed into meets of flow functions $A \mapsto A$. Here A can be either a finite set D (for IFDS problems [19]) or a mapping $D \mapsto L$ (for IDE problems [20]) from a finite set D to a lattice of values L. Intuitively, A represents a node in the graph and a function $A \mapsto A$ decides the nature of the edge from the node representing the argument to the node representing the result. Flow function composition then reduces to a transitive closure of the edges resulting in paths in the graph. The advantage of this approach is the worst-case polynomial bound on the termination of the algorithm. For example, in the case of IFDS problems the worst-case complexity is $O(ED^3)$, where E is the number of edges in the program super-graph and D is the size of the data flow domain.

However, many types of heap analyses such as points-to analysis and heap liveness analysis have non-distributive flow functions. Hence, these problems cannot be encoded as instances of the IFDS/IDE framework and solved via graph reachability. For example, consider the statement `x = y.n` to be processed for points-to analysis. If we have points-to edges $y \rightarrow o_1$ and $o_1.n \rightarrow o_2$ before the statement (where o_1 and o_2 are heap objects), then it is not possible to correctly deduce that the edge $x \rightarrow o_2$ should be generated after the statement if we consider each input edge independently. The flow function for this statement is a function of the points-to graph as a whole and cannot be decomposed into independent functions of each edge and then merged to get a correct result.

Chapter 3

Abstract Heap Representations

The example given in the previous chapter required answering alias queries between two variables. In general the alias analysis needed would not be so trivial. While performing liveness analysis for a statement such as $x.n = z$, it is required to determine whether x is aliased to objects accessible by the patterns of all accessor nodes having a node labelled n_i as their succecssor (where i can have any value). For every such alias that holds, the suffix of the access graph after such a n_i node is appended to the access graph of z as per the algorithm in [15]. Thus, the requirement is an alias analysis which can answer alias queries between access patterns of live accessor nodes.

One approach is to perform alias analysis by maintaining pairs (or equivalence classes) of access graphs. However, this can generate large amounts of unnecessary alias information which is not needed.

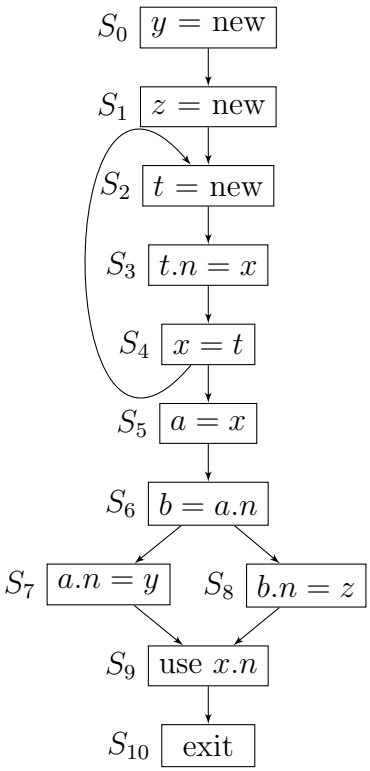
Another approach is to maintain a points-to graph of the heap at every program point. A points-to graph can be queried for aliases between access patterns by determining if there are any nodes reachable by both patterns.

Consider the program shown in Fig. 3.1 (a), which builds a linked list in a loop and then uses its second element just before exiting. The snapshot of the heap at the exit of statement S_6 of the program is shown in Fig. 3.1 (b). The following observations can be made about an ideal backwards liveness analysis:

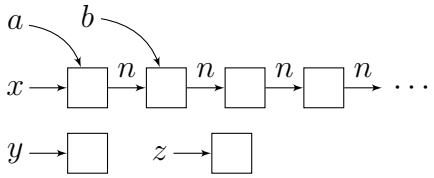
1. Due to the use of the second element of the linked list at statement S_9, the live access graph at its entry is shown in Fig. 3.1 (c). This access graph also propogates to the exit of S_7 and S_8.

2. At statement S_7, a must-alias x due to the assignment made earlier at S_5. Also, $x \rightarrow n$ is live after S_7. Thus, $a \rightarrow n$ is also implicitly live. Therefore, the variable y should be marked live at the entry to S_7, while $x \rightarrow n$ should be killed due to the strong update. The expected access graphs are shown in Fig. 3.1 (d).

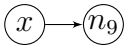
3. At statement S_8, b is aliased to $x \rightarrow n$, but since $x \rightarrow n \rightarrow n$ is not live, the variable z need not be marked live either. The expected access graphs are shown in Fig. 3.1 (e). Note that z is not used anywhere else in the program either and hence is a completely redundant object.



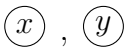
(a) Control-flow-graph of an example program.



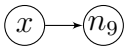
(b) Actual heap memory at the end of statement S_6.



(c) Liveness access graph at the entry of S_9 and exit of S_7 & S_8. $LVOUT_7 = LVOUT_8 = LVIN_9$.



(d) Expected value of $LVIN_7$.



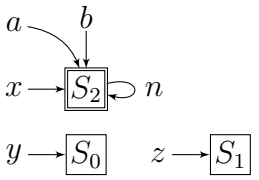
(e) Expected value of $LVIN_8$.

Figure 3.1: Snapshot of a program that constructs an arbitrary sized linked list and then uses its second element.

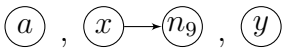
For the above steps to work correctly, an alias analysis would need to maintain a points-to graph of the heap at every program point. However, the size of the heap is unbounded due to recursive data structures such as the linked list in the example above. Thus, the alias analysis needs to use a bounded representation called an abstract heap which approximates the effect of all possible heaps at each program point. This is typically done by aggregating the information about multiple objects into a "summary node".

If, for any execution of the program, some two access paths can possibly point to the same object, then the points-to graph must have at least one node that is accessible by both these access paths to indicate a may-alias relationship. However, spurious may-alias relationships between access paths that will never point to the same object for all executions of the program result in a loss of precision. Similary, summarization of multiple objects into one abstract heap node loses the ability to answer must-alias queries, which is also a loss of precision.

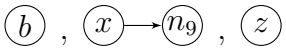
The choice of how to map abstract heap nodes into concrete objects (i.e. which nodes to merge and summarize) is a significant factor in the precision of the abstract heap representation.



(a) Traditional points-to analysis. $PTOUT_6 = PTIN_7 = PTIN_8$.



(b) Safe, but imprecise $LVIN_7$ as (a, x) considered *may* aliases.



(c) Safe, but imprecise $LVIN_8$ as (b, x) considered *may* aliases.

Figure 3.2: A points-to analysis that merges nodes based only on allocation sites is safe, but suffers from imprecision.

3.1 Merging on Common Allocation Sites

A classical solution has been to merge objects allocated at the same site to a *summary node*, by aggregating the in-edges and out-edges of its constituents [6]. The intuition is that objects allocated at the same site undergo similar transformations and hence their properties can be summarized.

Figure 3.2 shows how this technique is safe, in that aliases are preserved during merges, but imprecise, as new aliases which are not present in the original program are formed while summarizing nodes.

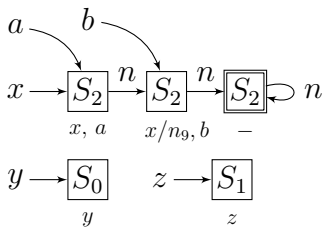
Look at the representation shown in Fig. 3.2 (a), which is the points-to graph at the exit of S_6 and thus also at the entry of S_7 and S_8. As all objects in the linked list were allocated at site S_2, they are summarized into one node in the points-to graph, shown with a double border. The objects pointed to by y and z are not summarized, and are represented with a single border and labelled with allocation sites.

This representation preserves the alias pair $a \overset{may}{=} x$, but spuriously creates the alias pair $b \overset{may}{=} x$. Note that both these aliases are may-aliases because they stem from summary nodes, which makes it difficult to guarantee if the alias holds for all objects summarized by this node. The spurious alias pair $b \overset{may}{=} x$ causes an unnecessary live variable z at the start of S_8, as shown in Fig. 3.2 (c). Further, because the summarization destroys the guarantee that x must-alias a, statement S_7 is a *weak update* and $x \rightarrow n$ is not killed, as shown in Fig. 3.2 (b).

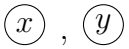
The creation of unnecessary may-aliases out of no-aliases and must-aliases reduces the precision of liveness analysis, which in turn diminishes the gains of liveness-based GC.

3.2 Distinguishing between Access Patterns

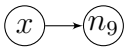
The key idea in our proposed approach is to improve the precision of points-to graphs summarized on allocation sites by distinguishing between objects that can be reached by distinct sets of live access patterns. Consider, for example, that the previous naive alias analysis was used to generate the live acccess graphs, which at the exit of S_6 are equal to the union of Figures 3.2 (b) and (c). The following observations can be made about



(a) Improved points-to analysis. $PTOUT_6 = PTIN_7 = PTIN_8$.



(b) Precise $LVIN_7$ as (a, x) *must* alias.



(c) Precise $LVIN_8$ as (b, x) do not alias.

Figure 3.3: The improved points-to analysis distinguishes between live access patterns, thus answering alias queries precisely.

the heap at that point, which is shown in Fig. 3.1 (b):

1. The first element of the list is accessible by the live access expressions x and a.
2. The second element of the list is accessible by the live access expressions b and $x \to n$.
3. The remaining objects of the list are not accessible by any live access expression.
4. The objects pointed to by y and z are accessible by the expressions y and z, respectively.

In our abstraction, nodes allocated at a particular site are merged *only if they are accessible by the same set of access patterns*. Thus, the third and subsequent elements of the linked list are merged into a summary node, while the other objects remain distinguished because they have unique access expressions. The resulting points-to graph can be seen in Figure 3.3 (a). Every node is additionally annoted below by the accessors whose access patterns reach it[1].

Such a representation allows precise answering of alias queries - we can now say that a *must* alias x and b definitely does not alias x. If we use this information to compute liveness access graphs again, then their precision can be improved. Figure 3.3 (b) shows the precise liveness graphs at S_7, in which $x \to n$ was killed due to a must-alias strong update, while Figure 3.3 (c) shows the precise access graphs at S_8, where z is not unnecessarily marked live because the variables x and b are known not to alias. These graphs are identical to the expectations shown previously in Figures 3.1 (d) and (e).

The advantage of this method is that the object z, which is found not to be live anywhere in the program, can be garbage collected very early even though there is a pointer to it. Similarly, if the left branch is taken (i.e. S_7), then the second and subsequent elements of the linked list can be freed, or if the right branch is taken (i.e. S_9), then the third element onwards can be freed along with the object pointed to by y.

[1]We use "accessors" and "access patterns" interchangibly when distinguishing nodes in this representation. This is allowed because if two nodes have equal sets of accessors, they must also have equal sets of access patterns.

Thus, our proposed approach is superior in terms of precision for alias analysis, because our abstract heap graphs are as expressible as required by the live access patterns, but finite at the same time. The points-to analysis depends on liveness access graphs, while accurate liveness analysis depends on alias information derived from the points-to graph. Thus, these analyses are inter-dependent.

Our approach is to perform alternate rounds of points-to analysis and liveness analysis similar to [14]. However, while the liveness-based pointer-analysis of [14] starts with an empty set of points-to relationships and progressively increases it as live variables are discovered, our approach works in the other direction; we start with a conservative classical alias analysis and progressively refine it using liveness graphs to improve precision. The following steps are to be performed:

1. Initialize the liveness information to the empty map at every program point.

2. Perform a forward points-to analysis on the heap, constraining the growth of recursive data structures by using the proposed abstract heap representation that merges nodes allocated at the same site only if they have the same set of accessors[2].

3. Perform a backward liveness analysis on the heap using access graphs. The abstract heap constructed in the previous step can be used to answer alias-related queries for statements such as $x.p = z$.

4. Repeat from step 2 until some stopping criteria is satisfied.

Note that with the current formulation it is necessary to perform the forward points-to and backward liveness analyses in distinct consecutive phases (each pair of phases is called a round). Interleaving the two does not generate useful results because a change in the liveness access graph needs to be propogated all the way up to the point of construction of a data structure so that it starts out as a precise representation; merely updating the points-to graphs of the local region cannot "open up" already summarized nodes.

An important point of discussion is when to terminate the alternate forward and backward phases. As mentioned earlier, unlike the rounds of liveness-based pointer analysis [14] which needs to reach a fixed point to get a correct result, our method ensures sound results for the points-to heap graph at every round – the refinement is purely for improving precision. Intuitively, the most precise results possible with this method seem to result when a fixed point is reached (that is, neither the liveness nor points-to information changes). However, we shall show in Section 4.4 that when considering only may-aliases, two rounds are sufficient to get the most precise result possible with this method.

We next formalize our proposed abstract heap representation which is called an *accessor relationship graph* because it indicates the connections between objects reachable by some set of accessors.

[2]The classical allocation-site based summarization [6] is a special case of our approach when no liveness information is available.

Chapter 4

Liveness-Driven Heap Analysis

4.1 Accessor Relationship Graph

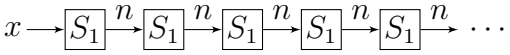
Let:

- V be the set of root variables (stack + globals).
- F be the set of possible field names.
- S be the set of statements in the program being analyzed.
- $M \subseteq S$ be the set of statements at which memory allocations occur.
- $R \subseteq S$ be the set of statements at which a reference field of an object is accessed.
- $A \subseteq V \cup (V \times F \times R)$ be the set of possible nodes in the access graphs. Each node is of the form x or x/n_i where $x \in V, n \in F, i \in R$. Thus, $|A| = |V| + |V| \times |R|$. [1]
- $H \subseteq M \times 2^A$ be the set of possible nodes in the proposed abstract heap graph. Each node has an allocation site and a set of accessors which can be accessed by the functions $alloc : H \to M$ and $accessors : H \to 2^A$ respectively.

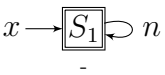
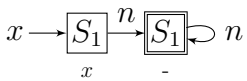
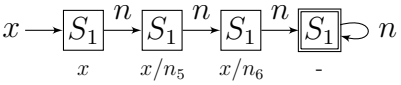
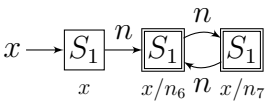
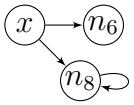
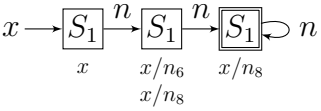
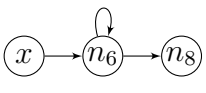
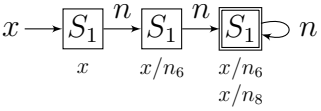
Definition 5. An **accessor relationship graph** (ARG) is a triple $\langle E_v, E_f, summary \rangle$, where:

- $E_v \subseteq V \times H$ is a set of edges representing root variable references.
- $E_f \subseteq H \times F \times H$ is a set of edges representing object field references.
- $summary : H \to \{true, false\}$ specifies whether a node represents a summary of multiple objects instead of a single unique location.

[1] As we consider a maximum of only one field access of the form $y = x.n$ in a given statement, the maximum number of nodes of the form n_i in an access graph of x is only $|R|$ and not $|F| \times |R|$.

Program	Concrete Heap
0: do { 1: t = new Node(); 2: t.n = x; 3: x = t; 4: } while (...);	

(a) A linked-list constructing program and the concrete heap at the end of the loop.

No.	Program	Access Graph	Abstract Heap Graph
1.	5: EXIT	-	
2.	5: use x; 6: EXIT		
3.	5: x = x.n; 6: x = x.n; 7: use x; 8: EXIT		
4.	5: while(...) { 6: x = x.n; 7: x = x.n; 8: } 9: use x; 10: EXIT		
5.	5: if(...) 6: x = x.n; 7: else do { 8: x = x.n; 9: } while(...); 10: use x; 11: EXIT		
6.	5: do { 6: x = x.n; 7: } while(...); 8: x = x.n; 9: use x; 10: EXIT		

(b) Accessor relationship graphs for various endings to the program.

Figure 4.1: Examples showing how different access graphs for a linked list result in different abstract heap graphs.

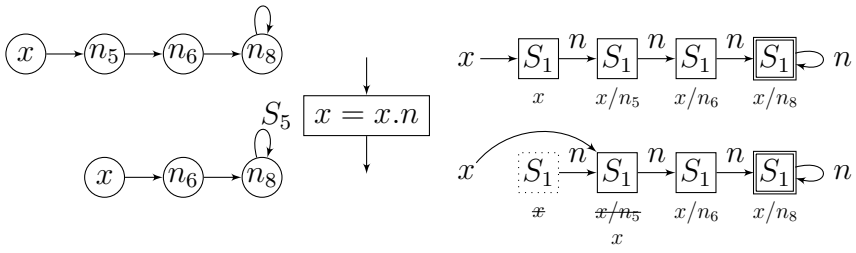


Figure 4.2: An example demonstrating normalization of an ARG.

Figure 4.1 (a) shows a program that constructs a linked list, whose elements are allocated at statement S_1. The real heap is shown in the second column.

Figure 4.1 (b) demonstrates various accessor relationship graphs for six different endings to the program. In each example, the second column is the program from statement S_5 onwards, the third column is the live access graph at S_5, and the fourth column shows the ARG computed from a pass after liveness analysis, at program point S_5. Node labels indicate allocation sites. Accessors are listed below the node. Summary nodes are marked with a double border.

In the first three examples, the liveness access graphs correspond to only one object per accessor, and thus these are distinguished while the rest of the list (no accessors) are merged to a summary node.

In the fourth example, the acesssor x/n_6 has a pattern $x \rightarrow n\ (\rightarrow n \rightarrow n)*$ while the accessor x/n_7 has the pattern $x \rightarrow n \rightarrow n\ (\rightarrow n \rightarrow n)*$. Thus, they represent accesses of every alternate node. The ARG shows exactly such a structure, where both the even offset and odd offset objects are summarized.

In the fifth example, the second node of the linked list is accessible by both the pattern of x/n_6 and x/n_8. The rest of the list is accessible only by x/n_8. The ARG provides exactly this information: all nodes from the second onwards are accessible by the pattern of x/n_8, while only the second by the pattern of x/n_6.

The last example is somewhat non-intuitive, as the structure of the access graph does not visually resemble that of the accessor relationship graph like it did in the previous examples. However, the information it represents is correct, in that the pattern of x/n_6 (which is $x \rightarrow n\ (\rightarrow n)*$) reaches the second and subsequent nodes of the linked list, while the pattern of x/n_8: (which is $x \rightarrow n \rightarrow n\ (\rightarrow n)*$) reaches the third and subsequent nodes of the list.

We now define some properties of the accessor relationship graph and its nodes which will be useful in data flow analysis.

4.2 Properties and Operations

During data flow analysis, the statements in a basic block may cause modifications in the edges of the abstract heap graph. Objects that were previously accessible by some set of access patterns will now be accessible by some other set of patterns. Hence, the

accessor relationship graphs will become inconsistent. Thus, nodes in the graph must be substituted with the nodes having the correct set of accessors to maintain consistency. Also, the deletion of some edges may result in some nodes being unreachable from any root variables. The redundant edges between unreachable nodes should be removed in order to maintain a compact representation.

For example, in Figure 4.2, the effect of statement S_5 removes the edge from x to the head of the list, and adds an edge from x to the second element of the list. Now, the annotations are inconsistent because the first node is not accessible by the expression x, and the second node is not accessible by x/n_5, but instead by the expression x. Further, because there is no path of edges from a root variable to the head of the list, it is an unreachable node, and hence can be removed (shown in the figure using dotted lines).

Another case where inconsistencies may arise is on a branch-node in the CFG, where points-to information is directly propogated from the branched node to its successors, but the liveness access graphs at the successor may be different. Hence, we subject both node and edge flow functions to a normalization function that ensures consistency and reachability. The properties and operations are formally defined below.

Reachability

Definition 6. A node k is said to be **reachable** within an accessor relationship graph $X = \langle E_v, E_f, summary \rangle$ if it is referenced by a root variable, or a field reference of a reachable node. Thus the reachability predicate $\rho_X(k)$ is defined as:

$$\rho_X(k) \Leftrightarrow (\exists x{\in}V : \langle x, k \rangle \in E_v) \vee (\exists h{\in}H : \langle h, *, k \rangle \in E_f \wedge \rho_X(h))$$

Consistency

Definition 7. An accessor relationship graph $X = \langle E_v, E_f, summary \rangle$ is **consistent** with respect to a live map L if every node is accessible by each of its accessors' access patterns. Thus, the following must hold true:

- $\forall k{\in}H \; \forall x{\in}V \; (x \in accessors(k)) \Leftrightarrow \langle x, k \rangle \in E_v \wedge live_L(x)$
- $\forall k{\in}H \; \forall x{\in}V \; \forall n{\in}F \; (x/n_i \in accessors(k)) \Leftrightarrow$
  $\exists a{\in}A \; \exists h{\in}H \; (a \in pred_L(x/n_i) \wedge \langle h, n, k \rangle \in E_f \wedge a \in accessors(h))$

Normalization

Definition 8. An accessor relationship graph is **normalized** with respect to a live map L if it is consistent with respect to L and all its edges are between reachable nodes only.

Definition 9. The **pattern mapping** function $\pi_{X,L} : H \rightarrow 2^A$ for a heap graph $X = \langle E_v, E_f, summary \rangle$ and live map L gives the new set of accessors for each node of X to maintain consistency with L:

$$\begin{aligned} \pi_{X,L}(k) = &\{x \mid live_L(x) \wedge \langle x, k \rangle \in E_v)\} \cup \\ &\{x/n_i \mid a \in pred_L(x/n_i) \wedge \langle h, n, k \rangle \in E_f \wedge a \in \pi_{X,L}(h)\} \end{aligned}$$

Definition 10. The **node substitution function** $\sigma_{X,L} : H \to H$ is a mapping of heap graph nodes of X to nodes with same allocation site and consistent access patterns with respect to L:

$$\sigma_{X,L}(k) = k' \Leftrightarrow (alloc(k) = alloc(k') \wedge \pi_{X,L}(k) = accessors(k'))$$

Definition 11. Given an accessor relationship graph $X = \langle E_v, E_f, summary \rangle$ and a live map L, the **normalization function** is defined as $\Theta(X, L) = Y$, where $Y = \langle E'_v, E'_f, summary' \rangle$ such that:

- $E'_v = \{\langle x, k' \rangle \mid \langle x, k \rangle \in E_v \wedge \rho_X(k) \wedge k' = \sigma_{X,L}(k)\}$
- $E'_f = \{\langle h', n, k' \rangle \mid \langle h, n, k \rangle \in E_v \wedge \rho_X(h) \wedge \rho_X(k) \wedge h' = \sigma_{X,L}(h) \wedge k' = \sigma_{X,L}(k)\}$
- $summary'(m) = \begin{cases} true & \exists h, k \colon h \neq k \wedge \sigma_{X,L}(h) = \sigma_{X,L}(k) = m \\ summary(m) & \text{otherwise} \end{cases}$

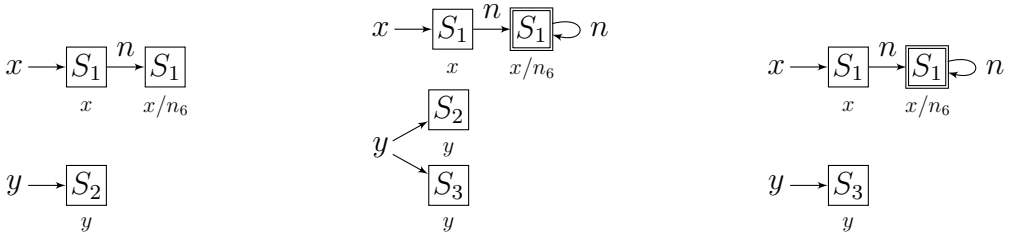
Theorem 1. *If $X = \langle E_v, E_f, summary \rangle$ is an accessor relationship graph and L is a live map, then $\Theta(X, L) = Y$ is an accessor relationship graph that is normalized with respect to L.*

Proof. As each node in the heap graph Y is the result of the substitution function σ, its accessors are given by the pattern mapping function π, which, by definition, ensures satisfaction of the consistency condition. Also, the normalization function only accepts the subset of edges whose nodes are reachable, satisfying the reachability condition. Thus, Y is normalized with respect to L by Definition 8. □

Querying for Aliases

The accessor relationship graph representation makes it very easy to determine aliases between live access patterns (i.e accessor nodes). Simply put, the patterns of two accessors α and β are may-aliased if there exists a reachable node which is accessible by both α and β. The condition for must-alias is much stricter, in that every node that is accessible by α must be accessible by β and vice versa, and none of these nodes must be a summary node.

- $may\_alias_X(\alpha, \beta) \Leftrightarrow \exists k \in reachable(X) \colon \{\alpha, \beta\} \subseteq accessors(k)$
- $must\_alias_X(\alpha, \beta) \Leftrightarrow \forall k \in reachable(X) \colon$
  $(\alpha \in accessors(k) \Rightarrow (\beta \in accessors(k) \wedge \neg summary(k))) \wedge$
  $(\beta \in accessors(k) \Rightarrow (\alpha \in accessors(k) \wedge \neg summary(k)))$



(a) X_1 (b) X_2 (c) X_3

Figure 4.3: Three heap graphs such that $X_2 \sqsubset X_1$, $X_2 \sqsubset X_3$ and also $X_2 = X_1 \sqcap X_3$.

4.3 Data Flow Analysis

Lattice Representation

We now define a partial order on the set of accessor relationship graphs and a meet operation for determining the greatest lower bound. Intuitively, if one heap graph is a subset of another, then the second graph includes all may-aliases of the first plus some more, while some must-aliases in the first graph might be may-aliases in the second graph, and hence the second is weaker (less precise).

Definition 12. Given two accessor relationship graphs $X = \langle E_v, E_f, summary \rangle$ and $Y = \langle E'_v, E'_f, summary' \rangle$, they are **partially ordered** $X \sqsupseteq Y$, if and only if:

- $E_v \subseteq E'_v$
- $E_f \subseteq E'_f$
- $\forall k \in H : summary(k) \Rightarrow summary'(k)$

Thus, X is more precise than Y. Conversely, Y is said to be more general than (an approximation of) X, denoted $Y \sqsubseteq X$.

For example see Figure 4.3 in which X_2 is strictly weaker than X_1 because it has extra edges (from x/n_6 to itself, and from y to the object allocated at S_3), and because a non-summary node in X_1 is summarized in X_2.

We now define a meet operation which determines the greatest lower bound of two accessor relationship graphs (i.e. given X and Y, the meet is the strongest Z such that $X \sqsubseteq Z$ and $Y \sqsubseteq Z$).

Definition 13. Given two accessor relationship graphs $X = \langle E_v, E_f, summary \rangle$ and $Y = \langle E'_v, E'_f, summary' \rangle$, the **meet operation** is $X \sqcap Y = Z$, where $Z = \langle E''_v, E''_f, summary'' \rangle$ such that:

- $E''_v = E_v \cup E'_v$
- $E''_f = E_f \cup E'_f$
- $\forall k \in H : summary''(k) = summary(k) \vee summary'(k)$

Stmt	$ModSet$	GEN_n	$KILL_n$
$x = null$	E_v	$\emptyset$	$\{\langle x, k\rangle \mid \langle x, k\rangle \in E_v\}$
$x = new$	E_v	$\{\langle x, k\rangle \mid k = \langle n, \{x\}\rangle\}$	$\{\langle x, k\rangle \mid \langle x, k\rangle \in E_v\}$
$x = y$	E_v	$\{\langle x, k\rangle \mid \langle y, k\rangle \in E_v\}$	$\{\langle x, k\rangle \mid \langle x, k\rangle \in E_v\}$
$x = y.q$	E_v	$\{\langle x, k\rangle \mid \langle y, h\rangle \in E_v, \langle h, q, k\rangle \in E_f\}$	$\{\langle x, k\rangle \mid \langle x, k\rangle \in E_v\}$
$x.q = y$	E_f	$\{\langle h, q, k\rangle \mid \langle x, h\rangle \in E_v, \langle y, k\rangle \in E_v\}$	$\{\langle h, q, k\rangle \mid \forall g \neq h\colon \langle x, g\rangle \notin E_v,$ $\langle x, h\rangle \in E_v, summary(h) = false\}$

Table 4.1: Flow functions for the liveness-driven heap analysis

For example see Figure 4.3 in which X_2 is the meet of X_1 and X_3 because it is an aggregation of the constituent edges, and a node $(S_1, x/n_6)$ is summarized because it was a summary in one of the operands, i.e. X_3. Also, the variable y which pointed to a unique object in the operands, now points to either of the two objects in the meet.

It is easy to see that the partial order is reflexive, antisymmetric and transitive, while the meet operation is idempotent, commutative and associative. Also, the lattice is bounded in that:

- $\top = \langle \emptyset,\ \emptyset,\ \lambda k.false \rangle$
- $\bot = \langle V \times H,\ H \times F \times H,\ \lambda k.true \rangle$

Also, as the sets V, H and F are finite, the lattice satisfies the descending chain condition.

Data Flow Equations

For each control-flow graph node n, the data flow equations for the forward points-to analysis pass are:

$$PTIN_n = \bigsqcap_{p \in pred(b)} \Theta(PTOUT_p, LVIN_b)$$

$$PTOUT_n = \Theta(f_n(PTIN_n), LVOUT_b)$$

Notice that the function $\Theta(X, L)$ is applied on the accessor relationship graph of every predecessor before performing the meet operation (in case the liveness information is different along the CFG edges) at $PTIN_n$, and also at $PTOUT_n$ after the statement-specific flow function f_n (in case the statment caused modifications to the edges of the accessor relationship graph).

For a given accessor relationship graph $X = \langle E_v, E_f, summary \rangle$ the flow functions are defined as:

$$f_n(X) = \begin{cases} \langle (E_v - KILL_n) \cup GEN_n, E_f, summary \rangle & \text{if } ModSet = E_v, \\ \langle E_v, (E_f - KILL_n) \cup GEN_n, summary \rangle & \text{if } ModSet = E_f \end{cases}$$

Where the GEN and $KILL$ sets for different types of statements are described in Table 4.1. $ModSet$ determines whether the set of root-variable edges or object-field edges are modified. The convergence of data flow analysis follows from the compositional monotonicity of statement flow functions with the normalization function.

Thus we have now defined an abstract heap representation that is capable of distinguishing between objects accessible by different access patterns, and developed a lattice-based data flow analysis which captures the points-to information of heap objects using this representation. The next section addresses the issue of how many rounds of alternate liveness and points-to analysis to perform in order to get the most precise liveness solution.

4.4 Precision of Liveness Analysis

In the following discussion we use S to refer to the set of all statements in the program (also known as program points) and AP to refer to the (infinite) set of all access paths that can be constructed by the variables and fields used in the program being analyzed.

Definition 14. Let $\hat{L}$ be the result of liveness analysis, which can be defined as a function of the form $S \times AP \rightarrow \{true, false\}$. That is, $\hat{L}$ determines if an arbitrary access path is live at a given program point. The actual function $\hat{L}$ can be derived from the access graphs resulting from heap reference analysis.

Definition 15. Let $\hat{P}$ be the result of points-to analysis, which can be defined as a function of the form $S \times AP \times AP \rightarrow \{true, false\}$. That is, $\hat{P}$ determines if a pair of access paths may be aliased at a given program point. The actual function $\hat{P}$ can be derived from the accessor relationship graphs resulting from heap points-to analysis.

Definition 16. Heap liveness analysis (HLA) can formally be defined as a function which takes the results of a points-to analysis (for answering alias queries) and returns the results of liveness analysis. Hence, we get $HLA : \hat{P} \rightarrow \hat{L}$.

Definition 17. Liveness-driven heap points-to analysis (PTA) can be formally defined as a function which takes the results of liveness analysis (for normalizing the accessor relationship graphs) and returns the results of the points-to analysis. Hence, we get $PTA : \hat{L} \rightarrow \hat{P}$.

Now, we are in a position to formally specify the system of alternate liveness and points-to analysis. The analysis is specified by the following equations:

$$\hat{L}_0 = \lambda s \lambda a.false \tag{4.1}$$

$$\forall i \geq 0 : \hat{P}_i = PTA(\hat{L}_i) \tag{4.2}$$

$$\forall i \geq 0 : \hat{L}_{i+1} = HLA(\hat{P}_i) \tag{4.3}$$

The above equations indicate that the first round of analysis is a points-to analysis which uses no liveness information ($\hat{L}_0$). After this, each round of liveness analysis and

points-to analysis uses the results of the previous round of points-to analysis and liveness analysis respectively.

Hence, we get a sequence of results $\hat{L}_0\hat{P}_0\hat{L}_1\hat{P}_1\hat{L}_2\hat{P}_2\hat{L}_3\cdots$. Of these, $\hat{L}_0$ is not sound as it is the initial assumption but all other results are sound as they are the results of sound data flow analysis. The question we would like to answer in this section is : *How many rounds need to be performed to get a precise liveness result?* In order to define precision we first need to define an ordering between results of analysis.

Definition 18. $\hat{L}_i \subseteq \hat{L}_j$ iff $\forall s \in S, \forall a \in AP : \hat{L}_i(s,a) \Rightarrow \hat{L}_j(s,a)$. That is, if an access path is live at a given point in $\hat{L}_i$, then it must be live at the same point in $\hat{L}_j$. Thus, $\hat{L}_i$ is as or more precise than $\hat{L}_j$ as it contains equal or fewer live access paths respectively.

Definition 19. $\hat{P}_i \subseteq \hat{P}_j$ iff $\forall s \in S, \forall a \in AP, \forall b \in AP : \hat{P}_i(s,a,b) \Rightarrow \hat{P}_j(s,a,b)$. That is, if a pair of access paths may alias at a given point in $\hat{P}_i$, then they are also may-aliased at the same point in $\hat{P}_j$. Thus, $\hat{P}_i$ is as or more precise than $\hat{P}_j$ as it contains equal or fewer aliased access paths respectively.

In order to reason about the relative precision of different rounds we need to state two lemmas. These lemmas are stated without proof but they are intuitive and should be easy to accept.

Lemma 1. $\forall i,j \geq 0 : \hat{L}_i \subseteq \hat{L}_j \Rightarrow \hat{P}_i \supseteq \hat{P}_j$

Lemma 2. $\forall i,j \geq 0 : \hat{P}_i \subseteq \hat{P}_j \Rightarrow \hat{L}_{i+1} \subseteq \hat{L}_{j+1}$

Lemma 1 suggests that the availability of less liveness information results in imprecise results of alias analysis. This is easy to see because the liveness-driven heap abstraction (also known as accessor relationship graph) distinguishes abstract heap nodes with distinct sets of accessors. Hence, more the number of live access patterns, lesser is the summarization in the abstract heap and thus fewer spurious aliases will be created.

Lemma 2 suggests that a more precise alias analyis results in more precise liveness analysis. This is very easy to see as the number of access graph nodes indirectly generated by a strong update statement of the form $x.n = z$ is directly proportional to the number of accessor nodes aliased with x.

Theorem 2. *The results of the second round of heap liveness analysis is the most precise result which is also sound. That is,* $\forall k > 0 : \hat{L}_2 \subseteq \hat{L}_k$.

Proof.

$$\begin{aligned}
\text{Equation 4.1 and Definition 18} &\Rightarrow \forall k \geq 0 : \hat{L}_0 \subseteq \hat{L}_k && (4.4)\\
\text{Equation 4.4 and Lemma 1} &\Rightarrow \forall k \geq 0 : \hat{P}_0 \supseteq \hat{P}_k && (4.5)\\
\text{Equation 4.5 and Lemma 2} &\Rightarrow \forall k \geq 0 : \hat{L}_1 \supseteq \hat{L}_{k+1} && (4.6)\\
\text{Equation 4.6 and Lemma 1} &\Rightarrow \forall k \geq 0 : \hat{P}_1 \subseteq \hat{P}_{k+1} && (4.7)\\
\text{Equation 4.7 and Lemma 2} &\Rightarrow \forall k \geq 0 : \hat{L}_2 \subseteq \hat{L}_{k+2} && (4.8)\\
\text{Equation 4.6 and 4.8} &\Rightarrow \forall k > 0 : \hat{L}_2 \subseteq \hat{L}_k && (4.9)
\end{aligned}$$

□

Chapter 5

Interprocedural Analysis using Value Contexts

For performing the proposed heap analysis in an interprocedural manner, we adapt the functional approach by combining the classical tabulation method [24] and the technique of value-based termination of call strings [13]. Both approaches revolve around the same key idea: if two or more calls to a procedure p have the same the data flow value (say x) at the entry of p, then all of them will have an identical data flow value (say y) at the exit of p. The tabulation method uses this idea to enumerate flow functions in terms of pairs of input-output values (x, y) whereas the modified call strings method uses it to partition call strings based on input values, reducing the number of call strings significantly. In each case a procedure needs to be analyzed only once[1] for an input value x.

The two methods lead to an important conclusion: Using data flow values as contexts of analysis can avoid re-analysis of procedure bodies. We make this idea explicit by defining a *value context* $X = \langle \mathit{method}, \mathit{entryValue} \rangle$, where *entryValue* is the data flow value at the entry to a procedure *method*. Additionally, we define a mapping $\mathit{exitValue}(X)$ which gives the data flow value at the exit of *method*. As data flow analysis is an iterative process, this mapping may change over time (although it will follow a descending chain in the lattice). The new value is propagated to all callers of *method* if and when this mapping changes. With this arrangement, intraprocedural analysis can be performed for each value context independently, handling flow functions in the usual way; only procedure calls need special treatment.

Although the number of value contexts created per procedure is theoretically proportional to the size of the lattice in the worst-case, we have found that in practice the number of distinct data flow values reaching each procedure is often very small. This is especially true for heap-based analyses that use bounded abstractions, due to the locality of references in recursive paths.

[1] In some cases in which the mapping changes, the modified call strings method can avoid re-analysis of the entire procedure.

5.1 Algorithm

Figure 5.1 provides the overall algorithm for performing interprocedural analysis using value contexts. Line 1 declares three globals: a set of contexts that have been created, a transition table mapping a context and call site of a caller method to a target context at the called method and a work-list of context-parametrized control-flow graph nodes whose flow function has to be processed.

The procedure initContext (lines 2-11) initializes a new context with a given method and entry value. The exit value is initialized to the $\top$ element. IN/OUT values at all nodes in the method body are also initialized to $\top$, with the exception of the method's entry node, whose IN value is initialized to the context's entry value. All nodes of this context are added to the work-list.

The doAnalysis procedure (lines 12-51) first creates a value context for the `main` method with some boundary information (BI). Then, data flow analysis is performed using the traditional work-list method, but distinguishing between nodes of different contexts.

A node is removed from the work-list and its IN value is set to the meet of the OUT values of its predecessors (lines 16-21). For nodes without a method call, the OUT value is computed using the normal flow function (line 37). For call nodes, parameter passing is handled by a call-entry flow function that takes as input the IN value at the node, and the result of which is used as the entry value at the callee context (lines 24-26). The transition from caller context and call-site to callee context is also recorded (line 27). If a context with the target method and computed entry value has not been previously created, then it is initialized now (line 34). Otherwise, the exit value of the target context is used as the input to a call-exit flow function, to handle returned values. A separate call-local flow function takes as input the IN value at the call node, and propagates information about local variables. The results of these two functions are merged into the OUT value of the call node (lines 29-32).

Once a node is processed, its successors are added to the work-list if its OUT value has changed in this iteration (lines 39-43). If the node is the exit of its procedure (lines 44-49), then the exit value of its context is set and all its callers are re-added to the work-list.

The termination of the algorithm follows from the monotonicity of flow functions and the finiteness of the lattice (which bounds the descending chain as well as the number of value contexts).

This algorithm assumed single entry/exit nodes per procedure as well as single targets at method calls. However, it can easily be extended to handle multiple entry/exit points per procedure as well as virtual method calls by merging data flow values across these multiple paths. It can also be easily adapted for backward data flow analyses.

```
1: global contexts, transitions, worklist
2: procedure initContext(X)
3:     add(contexts, X)
4:     Set exitValue(X) ← ⊤
5:     Let m ← method(X)
6:     for all nodes n in the body of m do
7:         add(worklist, ⟨X, n⟩)
8:         Set in(X, n) ← ⊤ and out(X, n) ← ⊤
9:     end for
10:    Set in(X, entryNode(m)) ← entryValue(X)
11: end procedure
12: procedure doAnalysis
13:    initContext(⟨main, BI⟩)
14:    while worklist is not empty do
15:        Let ⟨X, n⟩ ← removeNext(worklist)
16:        if n is not the entry node then
17:            Set in(X, n) ← ⊤
18:            for all predecessors p of n do
19:                Set in(X, n) ← in(X, n) ⊓ out(X, p)
20:            end for
21:        end if
22:        Let a ← in(X, n)
23:        if n contains a method call then
24:            Let m ← targetMethod(n)
25:            Let x ← callEntryFlowFunction(X, m, n, a)
26:            Let X′ ← ⟨m, x⟩                          ▷ x is the entry value at m
27:            Add an edge ⟨X, n⟩ → X′ to transitions
28:            if X′ ∈ contexts then
29:                Let y ← exitValue(X′)
30:                Let b1 ← callExitFlowFunction(X, m, n, y)
31:                Let b2 ← callLocalFlowFunction(X, n, a)
32:                Set out(X, n) ← b1 ⊓ b2
33:            else
34:                initContext(X′)
35:            end if
36:        else
37:            Set out(X, n) ← normalFlowFunction(X, n, a)
38:        end if
39:        if out(X, n) has changed then
40:            for all successors s of n do
41:                add(worklist, ⟨X, s⟩)
42:            end for
43:        end if
44:        if n is the exit node then
45:            Set exitValue(X) ← out(X, n)
46:            for all edges ⟨X′, c⟩ → X in transitions do
47:                add(worklist, ⟨X′, c⟩)
48:            end for
49:        end if
50:    end while
51: end procedure
```

Figure 5.1: Algorithm for performing inter-procedural analysis using value contexts.

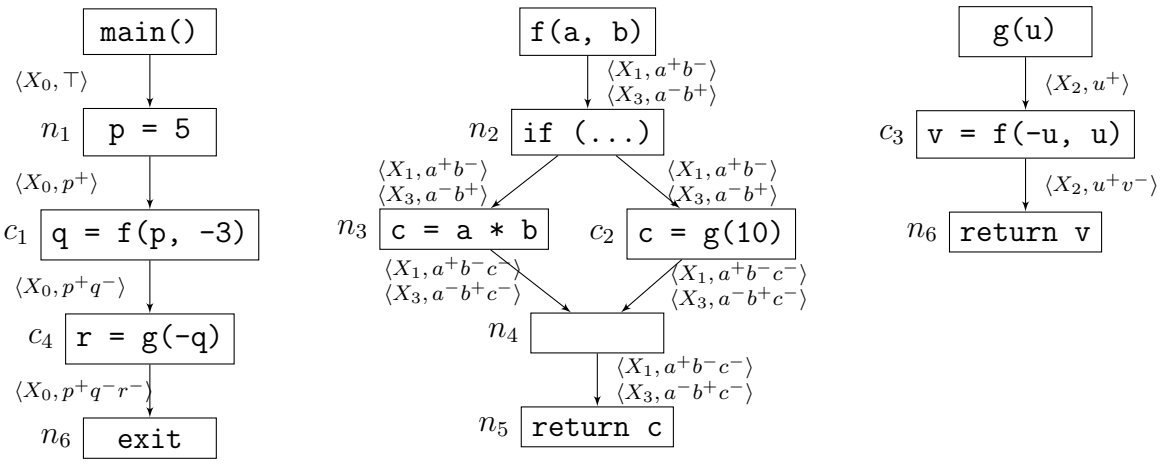


Figure 5.2: An example of interprocedural sign analysis using value contexts on a program with mutually recursive procedures. The control-flow graphs are annotated with context-sensitive data flow values.

5.2 Example

Consider the program in Figure 5.2, for which we wish to perform a simplified *sign analysis*, to determine whether a scalar local variable is negative, positive or zero. The call from `main` to `f` at c_1 will only return when the mutual recursion of `f` and `g` terminates, which happens along the program path $n_2 n_3 n_4 n_5$. Notice that the arguments to `f` at call-site c_3 are always of opposite signs, causing the value of variable c to be negative after every execution of n_3 in this context. Thus, `f` and hence `g` always returns a negative value.

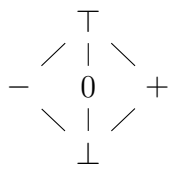


Figure 5.3: Lattice of data flow values for sign analysis.

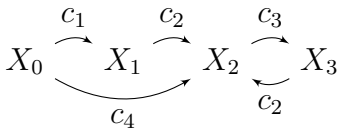
To compute this result using the algorithm described above, we use data flow values that are elements of the lattice in Figure 5.3, where $\top$ indicates an uninitialized variable and $\bot$ is the conservative assumption. We use superscripts to map variables to a sign or $\bot$, and omit uninitialized variables.

At the start of the program no variables are initialized and hence the analysis starts with the initial value context $X_0 = \langle \texttt{main}, \top \rangle$. For work-list removal, we will use lexicographical ordering of contexts (newer first) before nodes (reverse post-order).

The flow function of $\langle X_0, n_1 \rangle$ is processed first, which makes p positive (written as p^+). The next node picked from the work-list is c_1, whose call-entry flow function passes one positive and one negative argument to parameters a and b of procedure `f` respectively.

Context	Proc.	Entry	Exit
X_0	`main`	$\top$	$p^+q^-r^-$
X_1	`f`	a^+b^-	$a^+b^-c^-$
X_2	`g`	u^+	u^+v^-
X_3	`f`	a^-b^+	$a^-b^+c^-$

(a) Value contexts with entry/exit values



(b) Context-transition diagram

Figure 5.4: Resulting value contexts and their transitions for the sign analysis example.

Thus, a new value context $X_1 = \langle \texttt{f}, a^+b^- \rangle$ is created and the transition $\langle X_0, c_1 \rangle \rightarrow X_1$ is recorded.

Analysis proceeds by processing $\langle X_1, n_2 \rangle$ and then $\langle X_1, c_2 \rangle$, which creates a new value context $X_2 = \langle \texttt{g}, u^+ \rangle$ due to the positive argument. The transition $(X_1, c_2) \rightarrow X_2$ is recorded. When $\langle X_2, c_3 \rangle$ is processed, the arguments to `f` are found to be negative and positive respectively, creating a new value context $X_3 = \langle \texttt{f}, a^-b^+ \rangle$ and a transition $(X_2, c_3) \rightarrow X_3$.

The work-list now picks nodes of context X_3, and when $\langle X_3, c_2 \rangle$ is processed, the entry value at `g` is u^+, for which a value context already exists – namely X_2. The transition $\langle X_3, c_2 \rangle \rightarrow X_2$ is recorded. The exit value of X_2 is at the moment $\top$ because its exit node has not been processed. Hence, the call-exit flow function determines the returned value to be uninitialzed and the OUT of $\langle X_3, c_2 \rangle$ gets the value a^-b^+. The next node to be processed is $\langle X_3, n_3 \rangle$, whose flow function computes the sign of c to be negative as it is the product of a negative and positive value. The IN value at $\langle X_3, n_4 \rangle$ is $(a^-b^+c^- \sqcap a^-b^+) = a^-b^+c^-$. Thus, the sign of the returned variable c is found to be negative. As n_4 is the exit node of procedure `f`, the callers of X_3 are looked up in the transition table and added to the work-list.

The only caller $\langle X_2, c_3 \rangle$ is now re-processed, this time resulting in a hit for an existing target context X_3. The exit value of X_3 being $a^-b^+c^-$, the returned variable v gets a negative sign, which propagates to the exit node n_6. The callers of X_2, namely $\langle X_1, c_2 \rangle$ and $\langle X_3, c_2 \rangle$, are re-added to the work-list.

$\langle X_3, c_2 \rangle$ is processed next, and this time the correct exit value of target context X_2, which is u^+v^-, is used and the OUT of $\langle X_3, c_2 \rangle$ is set to $a^-b^+c^-$. When its successor $\langle X_3, n_4 \rangle$ is subsequently processed, the OUT value does not change and hence no more nodes of X_3 are added to the work-list. Analysis continues with nodes of X_1 on the work-list, such as $\langle X_1, c_2 \rangle$ and $\langle X_1, n_3 \rangle$. The sign of c is determined to be negative and this propagates to the end of the procedure. When exit node $\langle X_1, n_5 \rangle$ is processed, the caller of X_1, namely $\langle X_0, c_1 \rangle$, is re-added to the work-list. Now, when this node is processed, q is found to be negative.

Value-based contexts are not only useful in terminating the analysis of recursive procedures, as shown above, but also as a simple *cache* table for distinct call sites. For example, when $\langle X_0, c_4 \rangle$ is processed, the positive argument results in a hit for X_2, and thus its exit value is simply re-used to determine that r is negative.

Figure 5.4 shows the resulting value contexts for the program and the transitions

between contexts at call-sites.

A context-insensitive analysis would have merged signs of a and b across all calls to `f` and would have resulted in a $\perp$ value for the signs of c, v, q and r. Our context-sensitive method ensures a precise data flow solution even in the presence of recursion.

Notice that the flow function for n_3 is non-distributive since $f_{n_3}(a^+b^-) \sqcap f_{n_3}(a^-b^+) = a^+b^-c^- \sqcap a^-b^+c^- = a^\perp b^\perp c^-$ but $f_{n_3}(a^+b^- \sqcap a^-b^+) = f_{n_3}(a^\perp b^\perp) = a^\perp b^\perp c^\perp$. Hence this problem does not fit in the IFDS/IDE framework, but such flow functions do not pose a problem to our algorithm.

Another advantage of using *value contexts* as opposed to the classical tabulation method [24] or the modified call-strings method using value-based termination [13] is that it is very easy to implement the alternate system of liveness and points-to analysis that is proposed in this project for heap data and which is described in [14] for C-like pointers. The trick is to use a pre-generated call graph (with functions as nodes and call-sites as edges) as input for the first round of analysis, and to use the context-transition graph of the preceeding round (with value contexts as nodes and call-sites as edges) as input to the subsequent rounds. That is, the latter rounds will perform analysis on value-context bodies instead of method bodies. Effectively, the latter rounds will be analyzing a program where each method of the original program is cloned for every distinct data flow value of the previous round that reaches it. This is equivalent to using the tabulation or modified call-strings method by augmenting the data flow values to include a unique reference to a context of the preceeding round.

Chapter 6

Implementation

With the goal of implementing a complete liveness-based GC solution, several components have been implemented in this project, namely:

1. A generic access graph library in Java for use by any access graph-based analysis such as heap reference analysis [15] or the liveness-driven heap points-to analysis proposed earlier.

2. A generic inter-procedural analysis framework using the concept of value contexts which can be used for any data flow analysis in Java.

3. A novel approach to using access graphs generated by an analysis called *dynamic heap pruning* which uses the Java Debug Interface.

4. A Soot-based points-to analysis using the inter-procedural framework that builds precise context-sensitive call graphs on-the-fly.

6.1 Generic Access Graph Library

Access graphs were originally introduced in [15] and briefly described in Chapter 2. Access graphs are principally used as data flow values in the backward liveness analysis. Access graphs are also used by our proposed liveness-driven heap points-to analysis, and they may even be useful in the context of shape analysis as will be discussed in Chapter 8. Hence, the first step toward implementation is a generic access graph library that can be used by any client analysis.

The access graph library has been implemented in Java and uses generic types so that it can be used with any analysis toolkit or intermediate representation. The core classes are shown in Figure 6.1. The generic types are `V` for variable, `F` for field and `S` for statement. To put this in perspective, a Soot-based implementation would use the concrete types `soot.Local`, `soot.SootField` and `soot.Unit` respectively.

As the figure shows, the central class is `AccessGraph` which contains nothing more than a collection of edges stored as a map from an access graph node to a set of target

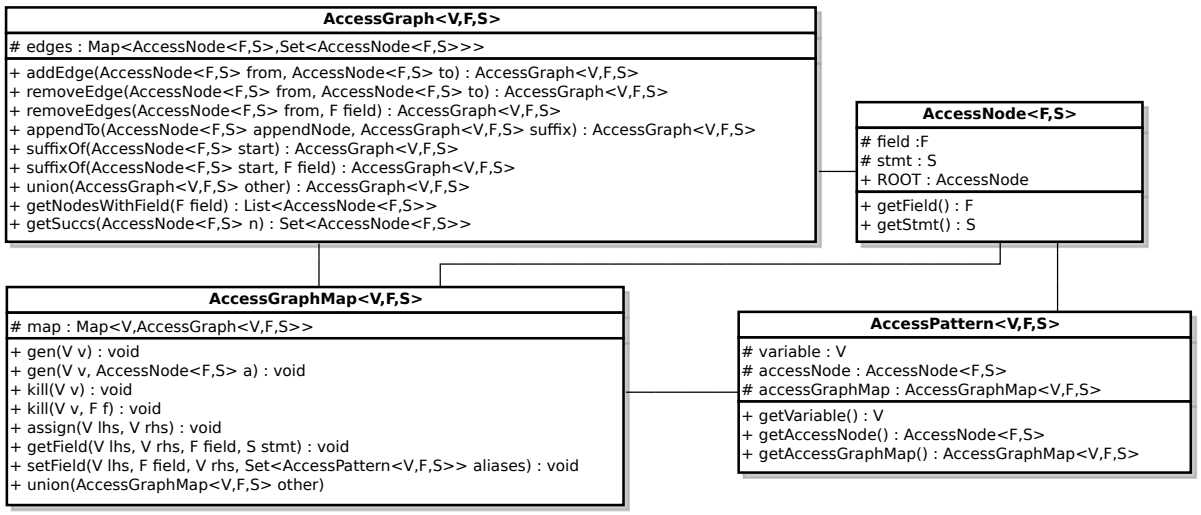


Figure 6.1: Class diagram of the generic access graph library.

nodes. Each access graph node is a pair containing a field and a statement suffix which is encapuslated by the class `AccessNode`. The `AccessGraphMap` class stores a map from variables to access graphs, and represents a live map used in data flow analysis. The `AccessPattern` class is used to uniquely refer to an access graph node of a variable in a given live map which, theoretically, can be used to deduce the finite automata and regular expression called the access pattern. In the implementation however, we simply use this triple of variable, access graph node and access graph map instead of actually building the automaton.

Objects of class `AccessGraph` are immutable. Hence, every operation on it (such as adding or removing edges or appending a suffix to a particular node) results in a new access graph. This design decision was taken so that multiple live maps can refer to the same object if the access graph of a variable is the same in both the maps. Since this is a very common case (as flow functions change access graphs of only one variable at a time), the immutable design helps improve performance.

On the other hand, objects of class `AccessGraphMap` are mutable in that operations such as `gen` and `kill` modify the map itself. This design decision was taken because data flow frameworks such as Soot and VASCO (described in the next section) typically assume mutable objects as data flow values. Flow functions often perform multiple operations on data flow values (such as removing a suffix of one access graph and appending it to another) and hence using immutable structures here would waste a lot of time in object allocation and garbage collection. Thus, users of the access graph library should take care to understand these concepts before using the classes.

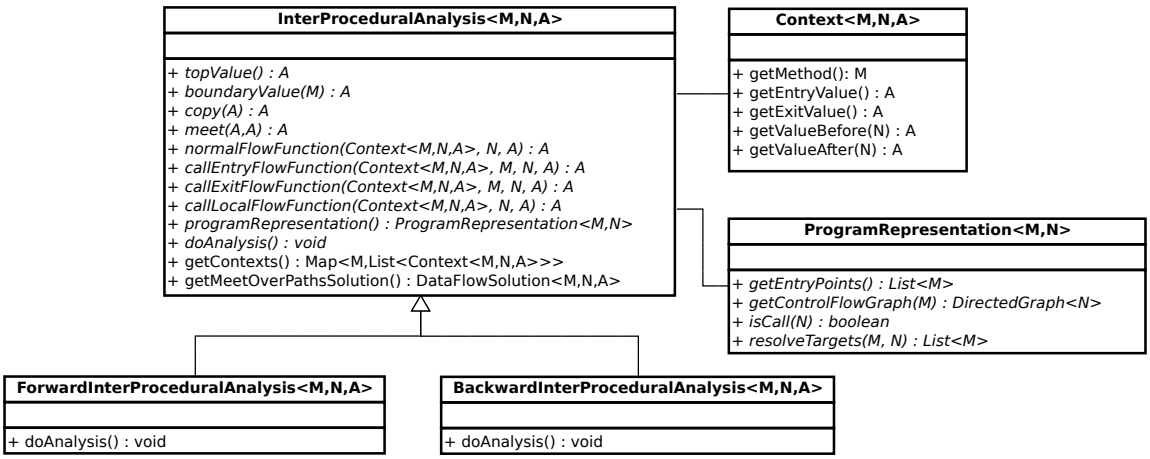


Figure 6.2: Class diagram of VASCO - the generic interprocedural analysis framework.

6.2 Generic Interprocedural Analysis Framework

Our platform of choice for implementing this project was Soot [25], which is a popular framework for analyzing Java programs. Although Soot traditionally lacked an interprocedural analysis framework, a recent effort by Bodden [5] has allowed integration with a graph-reachability solver called HEROS. Unforunately, as described in Section 2.3, heap analyses do not fit into the IFDS/IDE framework due to the presence of non-distributive flow functions.

Hence, we decided to build our own framework using the concept of value contexts as described in Chapter 5. This approach allows (1) full context-sensitivty even in the presence of recursion and non-distributive flow functions and (2) a context-sensitive data flow solution which is not possible in HEROS and is central to the concept of dynamic heap pruning which is explained in the next section.

The inter-procedural framework is named VASCO, which is an acronym for VAlue-Sensitive Contexts. It consists of a handful of core classes as shown in Figure 6.2. The use of generic types makes the framework agnostic to any particular toolkit or IR. The classes are parameterized by three types: `M` represents the type of a method, `N` represents a node in the control flow graph and `A` is the type of data flow value used by the client analysis. The framework can be naturally instantiated for Soot using the type parameters `soot.SootMethod` and `soot.Unit` for `M` and `N` respectively.

Clients using this framework would extend either `ForwardInterProceduralAnalysis` or `BackwardInterProceduralAnalysis`, both of which are subclasses of an abstract class `InterProceduralAnalysis`. The abstract methods `topValue`, `boundaryValue`, `copy` and `meet` provide a hook for client analyses to express initial lattice values and basic operations on them. The major functionality of the client analysis would be present in the `*FlowFunction` methods, whose roles were explained in Chapter 5. Additionally, clients are expected to provide a `ProgramRepresentation` object, which specifies program entry points (for which boundary values are to be defined) and resolves virtual calls.

Our framework ships with default program representations for Soot's Jimple IR. The launch point of the analysis is the `doAnalysis` method, which is implemented as per the algorithm from Figure 5.1 in the directional sub-classes.

The `Context` class encapsulates information about a value context. Every context is associated with a method, an entry value and an exit value, each of which can be retrieved using the corresponding *getter* methods. The `getValueBefore` and `getValueAfter` methods return data flow values for a context just before and after a node respectively. This is the recommended way for accessing the results of the analysis in a context-sensitive manner. A mapping of methods to a list of all its contexts is available through the `getContexts` method of the `InterProceduralAnalysis` class. Alternatively, `getMeetOverValidPathsSolution` can be used to obtain a solution that is computed by merging data flow results across all contexts of each method. The `DataFlowSolution` class (not shown in the figure) simply provides `getValueBefore` and `getValueAfter` methods to access the resulting solution.

6.3 Dynamic Heap Pruning using the Java Debug Interface

We have so far focused on data flow analysis techniques with the end goal of determining precise liveness information for heap allocated memory using access graphs. This section discusses the problem of using the resulting access graphs effectively in order to improve memory usage of running programs.

The original use of access graphs as given in [15] was to `null`ify dead references by inserting assignments such as `x = null` or `z.p.q = null` at appropriate places in the program, where the access paths are no longer live. This strategy has several issues associated with it. Firstly, care must be taken to ensure that these new statements do not raise exceptions due to dereferencing fields of null pointers. Hence, an *availability* and and *anticipability* analysis of access paths has to be performed prior to inserting these assignments. This reduces the number of access paths that can be effectively nullified. Further, the availability and anticipability analyses themselves require alias information, which if imprecise, results in further loss of nullification opportunities. Secondly, every such assignment that is added into the program code adds to the cost of execution. In some cases, the same field reference may be nullified more than once due to aliased links, further increasing the code size and execution cost.

An alternative solution would be to communicate the access graphs to an augmented garbage collector. Instead of starting at root variables and transitively marking all reachable objects as live, the garbage collector can only traverse the heap using the access graphs as a kind of state machine, and collect all objects that are not accessible by any live access patterns. This approach avoids the problems from the static nullification method because any references found to be `null` during the heap traversal can be safely ignored without worrying about raising exceptions. Also, aliasing information is perfect at run-time. However, the main problem with this solution is maintaining consistency

between the domain of access graphs and the run-time objects in the heap. It may be difficult to associate the named local variables and fields that are used by the access graphs with memory addresses and field offsets in the run-time system. Additionally, any program optimizations that are performed after the completion of liveness analysis (such as just-in-time optimizations which are common in many virtual machines) may change the program structure and render the liveness information unsound.

We present a novel solution called **dynamic heap pruning** that lies in-between the above two techniques. Firstly, to avoid the problem of raising exceptions when nullifying references, we shall perform the nullification at run-time only and without inserting any new statements in the original program. Secondly, in order to cleanly associate the domain of named locals and fields with run-time objects we shall use the **Java Debug Interface** (JDI), which allows debuggers to communicate with a running virtual machine for examining and possibly modifying objects in the heap. The problem of JIT optimizations is elegantly solved by the Java Virtual Machine (JVM) which can perform *dynamic deoptimization* [10] without a significant performance penalty when a debugger-like agent wishes to examine a running program. The procedure for dynamic heap pruning is as follows:

1. The Pruning Agent (PA), which is the entity that performs dynamic heap pruning, pauses a running program in a JVM using the debugger API. The program should be paused at such a point for which live access graph maps are available. In our current implementation, we trap all calls to a special static method `hra.dhp.DynamicHeapPruning#pruneNow()`. However, this can be engineered in different ways (e.g. periodic pruning, or pruning when memory usage exceeds a particular threshold).

2. The PA examines the call stack of each paused thread, and loads the correct access graphs for each stack frame. For the activation record at the top of the stack, the current Program Counter (PC) is the paused program point. For all other stack frames, the *return address* in the activation record of the next frame gives the program point at which the caller method is paused. Also, for each frame, the sequence of return addresses from the bottom of the stack to the frame in question gives a *call string* which can be used to find the exact value context from the context transition table in the results of the inter-procedural liveness analysis. Once the value context is determined for a stack frame, and the paused program point is known, the access graphs for that frame can be correctly loaded.

3. For each local in each stack frame, the corresponding access graph that has been loaded is used to traverse the heap, starting from the object referenced by the local variable. The PA labels each object by the set of accessors whose access patterns can be used to access them. The traversal of the heap starting from a root local and following field references is possible using the Java Debug Interface. Once all stack frames are processed in this manner, all live objects should be labelled with the complete set of accessors that can be used to reach them.

4. For each live object, the PA examines all its field references. If any accessor that reaches this object has an outgoing edge to this field in its own access graph, then this field reference is live. If none of the accessors of this object have an outgoing edge in their access graph to the field in question, then it is dead and hence the field reference is set to `null`. The manipulation of field references at run-time is also possible using the Java Debug Interface. This act of run-time nullification of dead links gives the technique of dynamic heap pruning its name.

5. When all objects are processed, the program is resumed again, and the next round of garbage collection will collect all unreachable objects as usual. Due to the pruning of dead links, some objects may become unreachable and hence will be collected automatically. Thus, liveness-based garbage collection is performed indirectly.

6.4 Points-To Analysis for Call Graph Construction

As mentioned earlier, the target platform for implementing this project was Soot [25]. Our first attempt at running a whole-program analysis using out inter-procedural framework turned out to be infeasible due to a large number of interprocedural paths that were the result of an imprecise underlying call graph.

The SPARK engine [16] in Soot uses a flow and context insensitive pointer analysis on the whole program to build the call graph, thus making conservative assumptions for the targets of virtual calls in methods that are commonly used such as those in the Java library. For example, it is not uncommon to find call sites in library methods with 5 or more targets, most of which will not be traversed in a given context. Some call sites can even be found with more than 250 targets! This is common with calls to virtual methods defined in `java.lang.Object`, such as `hashCode()` or `equals()`.

When performing whole-program data flow analysis, the use of an imprecise call graph hampers both efficiency, due to an exponential blow-up of spurious paths, and precision, due to the meet over paths that are actually interprocedurally invalid, thereby diminishing the gains from context-sensitivity.

Soot provides a context-sensitive call graph builder called PADDLE [17], but this framework can only perform k-limited call-site or object-sensitive analysis, and that too in a flow-insensitive manner. We were unable to use PADDLE with our framework directly because at the moment it not clear to us how the k-suffix contexts of PADDLE would map to our value-contexts.

We have implemented a flow and context-sensitive points-to analysis using our interprocedural framework to build a call graph on-the-fly. This analysis is both a demonstration of the use of our framework as well as a proposed solution for better call graphs intended for use by other interprocedural analyses.

The data flow value used in our analysis is a points-to graph in which nodes are allocation sites of objects. We maintain two types of edges: $x \rightarrow m$ indicates that the root variable x may point to objects allocated at site m, and $m.f \rightarrow n$ indicates that objects allocated at site m may reference objects allocated at site n along the field f.

Benchmark	Time	Methods (M)		Contexts (X)		X/M		Clean	
		Total	App.	Total	App.	Total	App.	Total	App.
`compress`	1.15s	367	54	1,550	70	4.22	1.30	50	47
`jess`	140.8s	690	328	17,280	9,397	25.04	28.65	34	30
`db`	2.19s	420	56	2,456	159	5.85	2.84	62	46
`mpegaudio`	4.51s	565	245	2,397	705	4.24	2.88	50	47
`jack`	89.33s	721	288	7,534	2,548	10.45	8.85	273	270
`antlr`	697.4s	1,406	798	30,043	21,599	21.37	27.07	769	727
`chart`	242.3s	1,799	598	16,880	4,326	9.38	7.23	458	423

Table 6.1: Results of points-to analysis using our framework. "App." refers to data for application classes only.

Flow functions add or remove edges when processing assignment statements involving reference variables. Nodes that become unreachable from root variables are removed. Type consistency is maintained by propagating only valid casts.

The points-to graphs at each statement only maintain objects reachable from variables that are local to the method containing the statement. At call statements, we simulate assignment of arguments to locals of the called method, as well as the assignment of returned values to a local of the caller method. For static fields (and objects reachable from them) we maintain a global flow-insensitive points-to graph. For statements involving static loads/stores we operate on a temporary union of local and global graphs. The call graph is constructed on-the-fly by resolving virtual method targets using type information of receiver objects.

Points-to information cannot be precise for objects returned by native methods, and for objects shared between multiple threads (as our analysis is flow-sensitive). Thus, we introduce the concept of a *summary node*, which represents statically unpredictable points-to information and is denoted by the symbol $\perp$. For soundness, we must conservatively propagate this effect to variables and fields that involve assignments to summary nodes. The rules for summarization along different types of assignment statements are as follows:

Statement	Rule used in the flow function
$x = y$	If $y \to \perp$, then set $x \to \perp$
$x.f = y$	If $y \to \perp$, then $\forall o : x \to o$, set $o.f \to \perp$
$x = y.f$	If $y \to \perp$ or $\exists o : y \to o$ and $o.f \to \perp$, then set $x \to \perp$
$x = p(a_1, a_2, ...)$	If p is unknown, then set $x \to \perp$, and $\forall o : a_i \to o$, $\forall f \in fields(o)$ set $o.f \to \perp$

The last rule is drastically conservative; for soundness we must assume that a call to an unknown procedure may modify the fields of arguments in any manner, and return any object. An important discussion would be on what constitutes an *unknown* procedure. Native methods primarily fall into this category. In addition, if p is a virtual method

invoked on a reference variable y and if $y \to \perp$, then we cannot determine precisely what the target for p will be. Hence, we consider this call site as a *default* site, and do not enter the procedure, assuming worst-case behaviour for its arguments and returned values. A client analysis using the resulting call graph with our framework can choose to do one of two things when encountering a *default* call site: (1) assume worst case behaviour for its arguments (eg. in liveness analysis, assume that all arguments and objects reachable from them are live) and carry on to the next statement, or (2) fall-back onto Soot's default call graph and follow the targets it gives.

A related approach partitions a call graph into calls from application classes and library classes [2]. Our call graph is partitioned into call sites that we can precisely resolve to one or more valid targets, and those that cannot due to statically unpredictable factors.

Table 6.1 lists the results of points-to analysis performed on seven benchmarks. The experiments were carried out on an Intel Core i7-960 with 19.6 GB of RAM running Ubuntu 12.04 (64-bit) and JDK version 1.6.0_27. Our single-threaded analysis used only one core.

The first two columns contain the names of the benchmarks; five of which are the single-threaded programs from the SPEC JVM98 suite [1], while the last two are from the DaCapo suite [4] version 2006-10-MR2. The third column contains the time required to perform our analysis, which ranged from a few seconds to a few minutes. The fourth and fifth columns contain the number of methods analyzed (total and application methods respectively). The next two columns contain the number of value-contexts created, with the average number of contexts per method in the subsequent two columns. It can be seen that the number of distinct data flow values reaching a method is not very large in practice. As our analysis ignores paths with method invocations on null pointers, it was inappropriate for other benchmarks in the DaCapo suite when using stub classes to simulate the suite's reflective boot process.

The use of *default* sites in our call graph has two consequences: (1) the total number of analyzed methods may be less than the total number of reachable methods and (2) methods reachable from *default* call sites (computed using SPARK's call graph) cannot be soundly optimized by a client analysis that jumps over these sites. The last column lists the number of *clean* methods which are not reachable from *default* sites and hence can be soundly optimized. In all but two cases, the majority of application methods are clean.

In order to highlight the benefits of using the resulting call graph, just listing the number of edges or call-sites alone is not appropriate, as our call graph is context-sensitive. We have thus computed the number distinct paths in the call graph, starting from the entry point, which are listed in Table 6.2. As the total number of call graph paths is possibly infinite (due to recursion), we have counted paths of a fixed length length k, for $1 \leq k \leq 10$. For each benchmark, we have counted these paths using call graphs constructed by our Flow and Context-sensitive Pointer Analysis (FCPA) as well as SPARK, and noted the difference as percentage savings ($\Delta\%$) from using our context-sensitive call graph. The option `implicit-entry` was set to `false` for SPARK.

Depth k =		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
compress	FCPA	2	5	7	20	55	263	614	2,225	21,138	202,071
	SPARK	2	5	9	22	57	273	1,237	23,426	545,836	12,052,089
	Δ%	**0**	**0**	**22.2**	**9.09**	**3.51**	**3.66**	**50.36**	**90.50**	**96.13**	**98.32**
jess	FCPA	2	5	7	30	127	470	4,932	75,112	970,044	15,052,927
	SPARK	2	5	9	32	149	924	24,224	367,690	8,591,000	196,801,775
	Δ%	**0**	**0**	**22.2**	**6.25**	**14.77**	**49.13**	**79.64**	**79.57**	**88.71**	**92.35**
db	FCPA	2	5	11	46	258	1,791	21,426	215,465	2,687,625	42,842,761
	SPARK	2	5	13	48	443	4,726	71,907	860,851	13,231,026	245,964,733
	Δ%	**0**	**0**	**15.4**	**4.17**	**41.76**	**62.10**	**70.20**	**74.97**	**79.69**	**82.58**
mpegaudio	FCPA	2	14	42	113	804	11,286	129,807	1,772,945	27,959,747	496,420,128
	SPARK	2	16	46	118	834	15,844	250,096	4,453,608	87,096,135	1,811,902,298
	Δ%	**0**	**12**	**8.7**	**4.24**	**3.60**	**28.77**	**48.10**	**60.19**	**67.90**	**72.60**
jack	FCPA	2	18	106	1,560	22,652	235,948	2,897,687	45,480,593	835,791,756	17,285,586,592
	SPARK	2	18	106	1,577	27,201	356,867	5,583,858	104,211,833	2,136,873,586	46,356,206,503
	Δ%	**0**	**0**	**0**	**1.08**	**16.72**	**33.88**	**48.11**	**56.36**	**60.89**	**62.71**
antlr	FCPA	6	24	202	560	1,651	4,669	18,953	110,228	975,090	11,935,918
	SPARK	6	24	206	569	1,669	9,337	107,012	1,669,247	27,670,645	468,973,725
	Δ%	**0**	**0**	**1.9**	**1.58**	**1.08**	**49.99**	**82.29**	**93.40**	**96.48**	**97.45**
chart	FCPA	6	24	217	696	2,109	9,778	45,010	517,682	7,796,424	164,476,462
	SPARK	6	24	219	714	2,199	20,171	306,396	7,676,266	192,839,216	4,996,310,985
	Δ%	**0**	**0**	**0.9**	**2.52**	**4.09**	**51.52**	**85.31**	**93.26**	**95.96**	**96.71**

Table 6.2: Number of k-length call graph paths for various benchmarks using SPARK and FCPA (Flow and Context-sensitive Pointer Analysis).

The savings can be clearly observed for $k > 5$. For $k = 10$, SPARK's call graph contains more than 96% spurious paths for three of the benchmarks, and 62-92% for the remaining. The gap only widens for larger values of k (for which the number of paths was too large to compute in some cases).

Client analyses using our interprocedural framework can be configured to use our context-sensitive call graphs which avoid these spurious paths, hence enabling efficient and precise solutions.

Chapter 7

Related Work

The potential benefits of heap liveness have been studied before using emperical methods [9, 23]. Heap reference analysis [15] is one of the approaches to achieving this goal, and this project builds upon its access graph representation. The need for a heap alias analysis was identified, but most work on pointer analysis has focused only on stack variables [8], with heap modelling falling into a class of work known as "shape analysis".

The classical approach to modelling the heap by partitioning on allocation sites was first introduced in [6]. This approach suffered from problems such as irreversible summarization or sharing of nodes. The concept of "materialization" was introduced in [21], in which shape graph nodes were distinguished by the set of variables that pointed-to them. The improved approach was much better at preserving shape invariants for destructive programs, such as whether an acyclic list remained an acyclic list after a reversal algorithm. However, their objective was mainly to detect topological properties of heap structures, and did not directly address alias analysis.

The concept of performing points-to analysis only on live variables was introduced in [14], in which alternate rounds of liveness and pointer analysis are performed until a fixed point is reached. Thus, the amount of information discovered is kept to a minimum. In this method, the points-to and liveness sets start with an optimistic assumption and increase in size on each round. Our approach progresses in the reverse direction. We start with a simple pessimistic heap model and use liveness information to improve the precision of the shape analysis, which in turn refines liveness. Bootstrapping [12] is another approach to performing multiple rounds of pointer analysis which improves precision in each round. In this method, the initial rounds are fast flow-insensitive analyses, and the subsequent flow and context-sensitive analyses are performed only on the set discovered by the previous rounds. In our approach, we stick to flow-sensitive analysis only.

The work that probably comes closest to ours is [3], in which a phased bi-directional shape analysis is performed using three-valued logic [22]. In this approach, the abstract heap is built in the forward pass, while the backward pass marks live objects or references on the heap graph itself. However, their analysis does not seem to be field sensitive (to capture complex access patterns such as Figure 2.1) and works well only for structures such as lists and trees in which summarized objects can be unambigiously separated when needed (unlike objects that are pointed-to by more than one field reference).

Most of these papers on heap analysis avoid discussing the problem of interprocedural program flow. Traditionally, interprocedural analysis has been divided into context-sensitive and context-insensitive techniques. Emperical studies have shown that for object-oriented programs, context-sensitive approaches are significantly more precise in the case of points-to analysis [18]. However, this study only compared different types of context-sensitivity for a flow-insensitive pointer analysis. Also, all the types of context-sensitive approaches that were studied (e.g. call-site sensitive, object sensitive, etc) sacrificed precision with the hopes of efficiency by limited the context information to some limit k, which was either 1 or 2. Possibly the best-known work on precise flow and context-sensitive points-to analysis is [7], which used a functional approach. However, their work was restricted to named stack locations and did not model the heap, which is the problem that we have addressed head-on.

The implementation platform we chose was Soot [25], which has been used for hundreds of client analyses and is very popular. However, Soot traditionally lacked an interprocedural data flow analysis framework. Recently, it has become possible to use Soot with a graph reachability-based solver called HEROS [5] which performs interprocedural analysis for IFDS/IDE problems. However, as points-to analysis and heap reference analysis both use non-distributive flow functions, they cannot be encoded as IFDS/IDE problems. VASCO, our interprocedural framework that uses value contexts, overcomes this restriction and can hence be used as a general framework for performing context-sensitive data flow analysis using Soot.

Chapter 8

Conclusion & Future Work

We have addressed the problem of heap alias analysis with the end goal of developing a totally liveness-based garabage collection mechanism. The Accessor Relationship Graph, which is our proposed heap abstraction, allows for distinguishing between abstract heap nodes which would have traditionally been summarized, while still maintaing a finite size. The precision of the heap abstraction is dependent on the access graphs between which aliasing queries need to be answered. Hence, our solution is an inter-dependent liveness and points-to analysis.

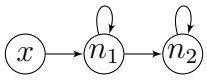
We have also implemented several critical components of the liveness-based GC system, including an access graph library, an interprocedural data flow analysis framework and a pruning agent based on our novel Dynamic Heap Pruning technique. The only piece missing in the whole system is the interprocedural heap liveness analysis which can use the access graph library and interprocedural framework to produce live access graphs for the pruning agent, resulting in a complete liveness-based GC solution.

8.1 Status of Implementation

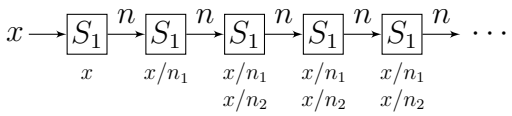
The access graph library that has been implemented is complete and its API has been documented. It has even been used in a test pilot heap reference analysis, and hence has been validated.

The Dynamic Heap Pruning implementation using the Java Debug Interface is also complete and tested. However, the testing was done on micro-benchmarks for which heap liveness analysis was performed without alias information. Until a complete liveness analysis implementation is available, the pruning agent cannot be tested on large benchmarks.

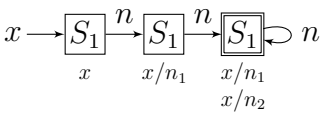
The interprocedural data flow analysis framework (VASCO) is also complete and has been released as an open source project (https://github.com/rohanpadhye/vasco). The framework ships with default program representations for Soot (such as using a context-insensitive or context-sensitive call graph) and example analyses (such as copy constant propagation). The framework API has been thoroughly documented and published.



(a) Artifical access graph for determining cycles in a linked list.



(b) A concrete linked list annotated with accessors that reach it.



(c) The corresponding abstract list in the accessor relationship graph.

Figure 8.1: Issues with performing shape analysis using artifical access graphs.

The points-to analysis that has been developed using this framework has been evaluated for its performance in constructing call graphs on-the-fly as reported in Section 6.4. Although the resulting call graphs for many micro-benchmarks have been manually verified, an automated validation system is required to prove correctness for large programs. A suggested approach for this can be to use the Java Debug Interface to trap every method call and return during program execution, and ensure that the dynamic call strings (deducable from the state of the activation stack) conform to the analyzed call graph. A prototype has been implemented to this extent but it is not fully usable due to some problems inherent in the JVM specification. In particular, the main thread often executes class loading and package resolution routines which pollutes the activation stack for the purpose of validation.

8.2 Shape Analysis using Access Graphs

It has not escaped our notice that the abstract heap representation that we have proposed immediately suggests a possible demand-driven mechanism for examining relationships between specific sets of access patterns, regardless of true liveness. For example, instead of performing a whole-program backward liveness analysis starting from the program exit, one could artifically introduce access graphs at some program point and propagate them up to the constructor of a data structure whose properties one wants to observe. The subsequent round of forward points-to analysis will distinguish between exactly those objects whose access patterns were of interest. This could be very useful in the context of performing shape analysis.

However, some fundamental issues still persist due to the regular nature of access patterns. Consider, for example, a situation in which we want to determine whether a data structure that is supposedly a linked list with head x, is actually an acyclic data structure. To verify that this list does not have cycles, we can query for an alias relationship between the accessors x/n_1 and x/n_2 of the access graph in Figure 8.1 (a). Intutively, the accessor x/n_1 represents any node in the linked list while the accessor x/n_2 represents any node that is strictly after the one referred to by x/n_1. Hence, if these two

accessors are aliased, then the linked list may contain cycles.

However, this intuition fails to materialize because the access pattern of x/n_1 (which is $x.n(.n)*$) is a superset of the access pattern of x/n_2 (which is $x.n.n(.n)*$). Figure 8.1 (b) shows the concrete linked list which is annotated with the accessors that reach it, resulting in the abstract heap representation (i.e. accessor relationship graph) of Figure 8.1 (c). As the figures show, the third and subsequent nodes in the linked list are accessible by both x/n_1 and x/n_2 and hence the may-alias relationship between these accessors seems to hold true, even when the linked list is indeed acyclic.

The problem seems to occur because our representation cannot remember state, such as the rule that if x/n_1 points to some object a, then x/n_2 must point only to objects *after* a. This could be rooted in the fact that access graphs describe a regular language. Perhaps a more stateful grammar would be appropriate for solving this problem. This appears to be an important and very interesting direction for extending the contributions of this project in the future.

References

[1] http://www.spec.org/jvm98. Accessed: April 3, 2013.

[2] Karim Ali and Ondřej Lhoták. Application-only call graph construction. In *Proceedings of the 26th European conference on Object-Oriented Programming*, ECOOP'12, 2012.

[3] Gilad Arnold, Roman Manevich, Mooly Sagiv, and Ran Shaham. Combining shape analyses by intersecting abstractions. In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006*, volume 3855, pages 33–48, 2006. http://www.odysci.com/article/1010112988884260.

[4] S.M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, October 2006.

[5] Eric Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, SOAP '12, 2012.

[6] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 296–310, New York, NY, USA, 1990. ACM.

[7] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 242–256, New York, NY, USA, 1994. ACM.

[8] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[9] Martin Hirzel, Amer Diwan, and Johannes Henkel. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Trans. Program. Lang. Syst.*, 24(6):593–624, November 2002.

[10] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM.

[11] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[12] Vineet Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 249–259, New York, NY, USA, 2008. ACM.

[13] Uday P. Khedker and Bageshri Karkare. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: resurrecting the classical call strings method. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, CC'08/ETAPS'08, 2008.

[14] Uday P. Khedker, Alan Mycroft, and Prashant Singh Rawat. Liveness-based pointer analysis. In *Proceedings of the 16th International Static Analysis Symposium*, SAS 2012, pages 265–282, 2012.

[15] Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *ACM Trans. Program. Lang. Syst.*, 30(1), November 2007.

[16] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using SPARK. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, 2003.

[17] 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), October 2008.

[18] 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):3:1–3:53, October 2008.

[19] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, 1995.

[20] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1-2), October 1996.

[21] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 16–31, New York, NY, USA, 1996. ACM.

[22] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 105–118, New York, NY, USA, 1999. ACM.

[23] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Estimating the impact of heap liveness information on space consumption in java. In *Proceedings of the 3rd international symposium on Memory management*, ISMM '02, pages 64–75, New York, NY, USA, 2002. ACM.

[24] M. Sharir and A Pnueli. Two approaches to interprocedural dataflow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., 1981.

[25] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, 1999.

Publications

[1] Rohan Padhye and Uday P. Khedker. Interprocedural data flow analysis in Soot using value contexts. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, SOAP '13, pages 31–36, New York, NY, USA, 2013. ACM.