

CLASS INVARIANT SHAPE ANALYSIS

by

Cindy Rubio

A Thesis submitted in
Partial Fulfillment of the
Requirements for the Degree of

Master of Science
in Computer Science

at

The University of Wisconsin-Milwaukee

December 2004

CLASS INVARIANT SHAPE ANALYSIS

by

Cindy Rubio

A Thesis Submitted in
Partial Fulfillment of the
Requirements for the Degree of

Master of Science
in Computer Science

at

The University of Wisconsin-Milwaukee

December 2004

Major Professor

Date

Graduate School Approval

Date

ABSTRACT

CLASS INVARIANT SHAPE ANALYSIS

by

Cindy Rubio

The University of Wisconsin-Milwaukee, 2004
Under the Supervision of Dr. Adam B. Webber

Class invariants can be used for compiler optimization, verification and program understanding. Finding class invariants is not a trivial task. In this thesis work we focus on the identification of class invariants that describe the shapes of linked data structures manipulated by the program (an unannotated Java program). We combine two techniques in order to achieve our goal: shape analysis and DC invariant analysis. We implement the analysis and experiment with some small programs. We find that the results obtained are the most of the time accurate except when dirty-called methods are present. Dirty-called methods introduce a serious problem in the analysis, so we discuss a couple of alternatives to handle them.

Major Professor

Date

TABLE OF CONTENTS

SECTION	PAGE
1. Introduction	1
2. Problem Description	3
3. Review of Class Invariant and Shape Analysis literature.....	4
3.1 Overview of Class Invariants	4
3.2 Disciplined Clean Invariants Definitions	5
3.3 DC Invariant Analysis.....	6
3.4 Overview of Shape Analysis	8
3.4.1 Shape Analysis.....	8
3.4.2 Static Shape Graphs.....	10
3.4.3 Shape Nodes.....	11
3.4.4 Summarization of a Shape Node.....	13
3.4.5 Materialization of a Shape Node.....	13
3.4.6 Transfer functions	14
3.5 Related Work.....	16
3.5.1 Class invariants	16
3.5.2 Shape analysis.....	17
4. Class Invariant Shape Analysis	19
4.1 Description	19
4.2 Implementation details	20
4.2.1 Existing components	20
4.2.2 New components.....	21
4.2.3 Method shape analysis.....	21
4.2.4 Class invariant shape analysis.....	22
4.3 Problems faced during implementation	24
5. Examples.....	27
5.1 Class IntStack	27
5.1.1 Java source code.....	27
5.1.2 Control flow graphs, opcodes and transformers	28
5.1.3 Class invariant shape analysis.....	31
5.2 Class IntList1	48
5.3 Class IntList2.....	50
5.4 Example involving dirty-called methods	51
6. Future work	53
7. Conclusion.....	54
References.....	55

LIST OF FIGURES

FIGURE	PAGE
Figure 1: Nonstatic DC invariant analysis	7
Figure 2: Graphical representation of shape descriptors	8
Figure 3: A shape node representing a node in a singly linked list, pointed to by variables x and y and pointing to null	12
Figure 4: (A) Control flow graph for method <i>push</i> . (B) Control flow graph for method <i>push</i> after inlining	29
Figure 5: Shape graphs corresponding to the analysis of method <i>push</i>	33
Figure 6: Assertion after analyzing method <i>push</i>	34
Figure 7: Shape graphs corresponding to the analysis of method <i>pop</i>	35
Figure 8: The transformer in method <i>peek</i> does not introduce anything to an empty shape graph	35
Figure 9: The new class assertion, a set containing one shape graph.....	36
Figure 10: Shape graphs corresponding to the analysis of method <i>push</i>	37
Figure 11: Final assertion after analyzing method <i>push</i> during the second iteration.....	37
Figure 12: Shape graphs corresponding to the analysis of method <i>pop</i> in the second iteration.....	38
Figure 13: Final assertion after analyzing method <i>pop</i> during the second iteration	39
Figure 14: The shape graph produced by method <i>peek</i> in the second iteration	39
Figure 15: Assertion after analyzing method <i>peek</i> during the second iteration.....	40
Figure 16: Class assertion with graphs from abrupt termination	40
Figure 17: Class assertion without graphs from abrupt termination. (A) First graph in the set. (B) Second graph in the set.....	40
Figure 18: Shape graphs corresponding to the analysis of the method <i>push</i> in the third iteration.....	41

Figure 19: Final assertion after analyzing method <i>push</i> . (A) First graph in the set. (B) Second graph in the set.....	42
Figure 20: Shape graphs corresponding to the analysis of the method <i>pop</i> in the third iteration.....	43
Figure 21: Final assertion after analyzing method <i>pop</i> during the third iteration.....	43
Figure 22: The shape graph produced by method <i>peek</i> in the third iteration	44
Figure 23: Final assertion after analyzing method <i>peek</i> during the third iteration	44
Figure 24: Class assertion reaching after third iteration. (A) First graph in the set. (B) Second graph in the set. (C) Third graph in the set.....	44
Figure 25: Shape graphs corresponding to the analysis of the method <i>push</i> in the fourth iteration	45
Figure 26: Final assertion after analyzing method <i>push</i> . (A) First graph in the set. (B) Second graph in the set.....	45
Figure 27: Shape graphs corresponding to the analysis of the method <i>pop</i> in the fourth iteration.....	46
Figure 28: Final assertion after analyzing method <i>pop</i> . (A) First graph in the set. (B) Second graph in the set.....	46
Figure 29: The shape graph produced by method <i>peek</i> in the fourth iteration	47
Figure 30: Final assertion after analyzing method <i>peek</i> during the fourth iteration	47
Figure 31: New class assertion after the fourth iteration.....	47
Figure 32: Class assertion after the fourth iteration	50
Figure 33: Class assertion after the fourth iteration in class <code>IntList2</code>	51
Figure 34: Weakest assertion in which there is only one variable	52

LIST OF TABLES

TABLE	PAGE
Table 1: Opcodes and Transformers for constructor <i>IntStack</i>	30
Table 2: Opcodes and Transformers for method <i>push</i>	30
Table 3: Opcodes and Transformers for method <i>pop</i>	30
Table 4: Opcodes and Transformers for method <i>getTail</i>	30
Table 5: Opcodes and Transformers for method <i>peek</i>	30
Table 6: Opcodes and Transformers for constructor <i>IntList1</i>	49
Table 7: Opcodes and Transformers for method <i>reverse</i>	49
Table 8: Opcodes and Transformers for method <i>setTail</i>	41

ACKNOWLEDGMENTS

I would like to thank all of those who helped me.

I specially want to thank:

Dr. Adam B. Webber for his guidance, time, support and endless optimism.

Lidia Bonilla for the valuable discussions.

Dr. John T. Boyland and Dr. Ethan V. Munson for their useful comments.

Professors and staff of the Department of Electrical Engineering and Computer Science at the University of Wisconsin-Milwaukee for their support during my studies.

CONACYT - Consejo Nacional de Ciencia y Tecnologia (National Council of Science and Technology) for believing on me and choosing me among hundreds of students in the country to be a recipient of a Postgraduate Scholarship, Mexico, 2002-2004.

SEP - Secretaria de Educacion Publica (Secretary of Public Education) for giving me their support through the Complemento Postgraduate Scholarship, Mexico, 2003-2004.

My parents Victor and Minerva, and my sister Minerva for their love, patience and unconditional support.

1. INTRODUCTION

Apparently similar but extremely different definitions of *class invariant* can be found in the literature. The reason is that *class invariants* can be seen from different points of view. *Class invariants* are treated normatively the most of the time [9], i.e. they are used to prescribe a norm or standard. For example, the programmer identifies that certain value is never null or is in within a specific range and introduces an assertion to be checked by the system at run-time that will indicate a bug in the program whenever it evaluates to false.

Among the programming languages that allow the specification of class invariants by using assertions are Eiffel [4], Anna [3] and JML [2]. Some projects using JML [2] such as ESC/Java [13], LOOP [14] and Daikon [15] have as goal to help the programmer to find bugs in the program by attempting proofs of the assertions provided by him [9]. All these projects attempt to verify what the programmer has identified as a class invariant, so nothing is to be done about class invariants missed or just ignored by the programmer.

Our purpose is to identify class invariants for use in compiler optimization. As it can be seen, it may not be enough to rely on the programmer to specify the class invariants: first because the programmer should not be forced to specify class invariants whose only purpose is to help the compiler to generate better code and second, because even when the programmer is willing to specify them, he may not be aware of their existence. Therefore, we need an alternative way to identify them.

The approach is to directly analyze the code to identify class invariants. So rather than looking at class invariants as assertions specified by the programmer that establish a

certain behavior of the program, we see them as properties to be extracted from the code that will describe the behavior of the program.

For example, consider a class *Stack* that implements the standard stack operations (push, pop, peek and isEmpty), using a linked list in the obvious way. Our analysis can examine such a class and determine a variety of class-invariant properties. One class-invariant property of such a *Stack* is that the linked list is always acyclic. Another is that the nodes of the linked list are always unshared.

An existing technique called shape analysis can identify properties of linked data structures. [5, 6, 7, 8] show how to perform intraprocedural shape analysis. As mentioned in [17], scaling the shape analysis algorithm to analyze full programs and not just single procedures is difficult. Interprocedural shape analysis has been attempted on C [16] and Java [18] programs. Some of these attempts follow the k-call string approach, which is both imprecise and expensive according to [16]. An existing DC invariant analysis can be used to reason with class-invariant assertions. It has been applied to some unary and binary analyses, but it has never before been used on assertions about the shapes of linked data structures.

Our research combines these two techniques and achieves results that have not been achieved before: the identification of class-invariant properties of linked data structures in unannotated Java code using DC invariant analysis.

The next section presents formally the problem to be studied in this thesis. Section 3 provides an overview of class invariants and shape analysis, including the work done in both areas. Class invariant shape analysis is presented in Section 4. Section 5 includes

three examples to which class invariant shape analysis was applied and the results obtained. Future work is presented in Section 6 followed by the Conclusion in Section 7.

2. PROBLEM DESCRIPTION

Properties of linked data structures such as acyclicity, disjointness and sharedness can be used for compiler optimization. Identifying these properties is not trivial. The shape analysis technique can reveal such properties, however it is difficult to scale to analyze full programs. The DC invariant analysis can be used to find class invariants, however it has never been applied to the analysis of linked data structures.

This thesis has as its goal to design and implement a static program analysis technique to identify class-invariant properties of the linked data structures manipulated in a program by combining the DC invariant analysis and the local shape analysis technique in unannotated Java programs. The results could be used in a variety of applications such as program optimization, understanding, verification and debugging among others. The results could also be used to assess the concept of DC class invariants, which has never been applied to this kind of analysis.

3. REVIEW OF CLASS INVARIANT AND SHAPE ANALYSIS LITERATURE

In this review we start by presenting in section 3.1 a general discussion about class invariants. Section 3.2 introduces the category of class-invariants we will work with in this thesis: DC class invariants. The DC class invariant analysis is presented in section 3.3. An overview of the shape analysis technique follows in section 3.4. Finally, there is a section discussing some related work in both areas: class invariants and shape analysis.

3.1 Overview of Class Invariants

As mentioned earlier, many different definitions of class invariant can be found in the literature and that is because they can be seen from different points of view. A simple but inadequate definition might be:

A class invariant is a property that is true of all objects of a given class at all times.

[9] presents a complete discussion of this definition. There are many questions we need to ask ourselves. First we need to decide what we mean *by a property*. Are they unary or binary properties? Do we just refer to the values of the fields of an object? Are we referring to properties of linked data structures? Second, the definition says that a class invariant is a property that is true of *all objects*. Should it mean all objects that occur in any execution of a given program or all objects that occur in any execution of any program that contains the class? Third, what does it mean to say of *a given class*? How about inheritance? And last, when does an invariant start being true for an object? Is it possible for the class invariant to be temporary broken in the middle of the execution of a method? We need to answer each of these questions in order to define what we mean by class invariant.

The next section answers all of these questions for a particular category of class invariants: DC invariants. Section 3.3 introduces the DC analysis technique.

3.2 Disciplined Clean Invariants Definitions

Disciplined clean invariants [21] are a category of class invariants that are reasonably tractable for static analysis. Before introducing the definition of DC invariant, it is necessary to define disciplined and undisciplined fields, tight and leaky constructors, as well as clean-called and dirty called methods.

Definition 1

A disciplined nonstatic field of a class is one that is written only in constructors or in synchronized nonstatic methods of the class, and only as a field of this (the self object). All other nonstatic fields are undisciplined.

Definition 2

A tight constructor is one that does not store a reference to the object under construction (except on the operand stack), either directly or by passing it to a method, which stores it. All other constructors are leaky.

Definition 3

A clean-called nonstatic method is one that is not called, either directly or transitively, from any constructor of the class, or from any nonstatic method of the class at any point at which any disciplined nonstatic field has been altered. It is either explicitly synchronized, or else called only from other clean-called nonstatic methods of the class. All other nonstatic methods are dirty-called.

Definition 4

A nonstatic Disciplined Clean invariant is a relation involving only constants and/or the disciplined fields of a class that is always true on entry in clean-called nonstatic methods of the class.

If *constructors* are substituted for *class initialization code* in the previous definitions then we will get the definitions for disciplined and undisciplined static fields, clean-called and dirty-called static methods, and static DC invariants. Nonstatic DC invariants and static DC invariants need to be classified as two different categories of DC

invariants when dealing with multi-threaded programs since static methods and nonstatic methods synchronize on different locks; for single-threaded programs they can be considered as a single category.

3.3 DC Invariant Analysis

The DC invariant analysis algorithm is a consequence of Theorem 1. Theorem 2 may be used for the analysis of static DC invariants.

Theorem 1

The disciplined nonstatic fields of this on entry in any clean-called nonstatic method of a class must be as they were on exit from some constructor or nonstatic method of the class, excluding the case of abrupt termination in tight constructors.

Theorem 2

The disciplined static fields of a class on entry in any clean-called static method of the class must be as they were on exit from class initialization code or from some static method of the class.

Nonstatic DC invariant analysis is illustrated in Figure 1.

$$\begin{aligned}
C &= \text{a class} \\
A &= \text{Any domain of field assertions for the disciplined nonstatic fields of } C, \text{ ordered by implication} \\
a_{\text{default}} \in A &= \text{The strongest assertion in } A \text{ that applies to the fields as initialized using the Java default initializers} \\
\forall m \in C, \\
J_m: A \rightarrow A &= \text{Monotonic method analysis function.} \\
&\forall a \in A, \text{ if } a \text{ is true of } \textit{this} \text{ on entry in } m, \text{ then } J_m(a) \text{ is true of } \textit{this} \text{ on exit from } m \\
J_m &= N_m \sqcup A_m \\
N_m &= \text{Monotonic method analysis function for the normal termination} \\
A_m &= \text{Monotonic method analysis function for the abrupt termination} \\
B &= \sqcup \text{ Tight constructors } x \quad N_x(a_{\text{default}}) \quad (1) \\
C &= \sqcup \text{ Leaky constructors } x \quad J_x(a_{\text{default}}) \quad (2) \\
D &= \sqcup \text{ Nonstatic dirty-called methods } x \quad J_x(\top) \quad (3) \\
a_{\text{inv}} &= b \sqcup c \sqcup d \quad \sqcup \text{ Nonstatic clean-called methods } x \quad J_x(a_{\text{inv}}) \quad (4) \\
K(a) &= A \quad \sqcup \text{ Nonstatic clean-called methods } x \quad J_x(a) \quad (5) \\
A_0 &= b \sqcup c \sqcup d \quad (6) \\
a_{i+1} &= K(a_i) \quad (7)
\end{aligned}$$

Figure 1: Nonstatic DC invariant analysis

3.4 Overview of Shape Analysis

3.4.1 Shape analysis

Shape analysis is a static program analysis technique attempting to determine properties of the heap contents [1]. The goal of Shape analysis is to give, for each program point, a set of finite shape graphs that describe the possible shapes that the heap-allocated data structures manipulated by the program might have at that specific point in the program. Since shapes describe the contents of the heap, they are also known as shape descriptors.

Before mentioning examples of what a shape descriptor might indicate and the properties that this analysis technique may reveal, it is important to note that Shape analysis is exclusively interested in the heap-allocated data structures manipulated by the program and in the pointers into the heap from the stack, global memory, or from cells in the heap. Shape analysis is not interested in any non-address values computed by the program.

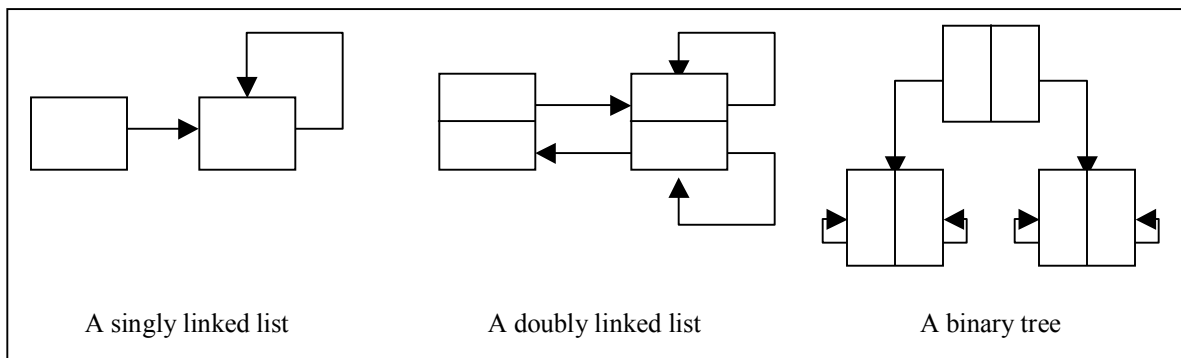


Figure 2: Graphical representation of shape descriptors.

As shown in Figure 2, a shape descriptor could indicate whether a singly linked list, doubly linked list, binary tree or any other data structure is contained in the heap.

Shape analysis computes a set of such shape graphs for every program point: a finite set representing the infinitely many possible configurations of the heap at that point. Each set of shape descriptors or shape graphs that could arise at each point in the program can then be analyzed in the search for properties such as cyclicity/acyclicity, jointness/disjointness, sharing/unsharing, reachability/unreachability, etc.

Knowing whether one or several properties hold for all the possible shape graphs at each point in the program is valuable information that could be further used in program debugging, optimization, verification and understanding. [6] gives us some examples of how this information could be used:

- a) **Debugging.** If a pointer variable or pointer component of a heap cell may contain *null* then it would be valuable debugging information to know it at the entry of any statement attempting to dereference this pointer. Another example is if we know that a heap cell may be *shared*. The heap cell might be explicitly deallocated more than once and might leave the store manager in an inconsistent state. Likewise, if we know that a heap cell is never shared, it can be deallocated as soon as the last pointer to it does not exist.
- b) **Optimization.** If we know that a heap cell will never be pointed by two distinct pointer variables, then the program dependence information can be improved. Another example is if we know that a heap cell is unreachable then it can be garbage collected. If we know that two data structures are *disjoint*, then they may be processed in parallel by different processors and may be stored in different memories.
- c) **Verification and Understanding.** Knowing the properties that hold in all the sets of possible shape graphs that could arise at a specific point in the program could

definitely help us to understand the behavior of the program as well as to verify that it meets its specification.

d) Reference counting is a very simple, non-compacting garbage collection technique in which the heap maintenance is spread throughout the program execution rather than suspending the program when the garbage collector runs [12]. Identifying that a linked data structure may be *cyclic* can help us to avoid using reference counting since one of the technique's major problems is that it cannot garbage collect circularly linked data structures.

We will not discuss the structural operational semantics of shape analysis here. For more information please refer to [5, 7].

3.4.2 Static Shape Graphs

As mentioned earlier, the Shape Analysis technique will produce a set of shape graphs for each point in the program. What is a shape graph? A static shape graph is a finite, labeled, directed graph that approximates the actual or concrete stores that can arise at a specific point in execution [7]. A shape graph consists of a set of shape nodes, each of them representing a concrete cell (we will see later that there exists a special kind of shape node that may represent multiple heap cells).

An important property of each static shape graph in a program is that they must have a bounded size, unlike the data structures that programs manipulate and whose size is in general unbounded. How is it possible to achieve a bounded size for the shape graphs when the data structures that the shape graphs are describing may not have a bounded size? A special kind of shape node called a summary node makes it possible to

achieve a bounded size for the shape graph by allowing the representation of multiple heap cells (not pointed to by any pointer variable or heap cell at that given program point) by the same shape node.

Another property of the shape graphs is that their description of the heap contents must be conservative, i.e. every concrete store that can occur at a given program point is represented by one of the shape graphs in the set computed for that point.

3.4.3 Shape Nodes

Shape graphs are abstractions of memory and they consist of one or more shape nodes. Each shape node consists of one or more pointers to other shape nodes depending on the data structure being represented. For example, if the program is manipulating a singly linked list, a shape graph will attempt to describe it and each shape node in the graph will have a single pointer to another shape node. Each shape node would have two pointers to shape nodes if the data structure being represented is a doubly linked list, and so forth.

A shape node cannot be thought of as a representation of a fixed partition in memory since a shape node does not contain information about the concrete location. A shape node instead keeps aliasing-configuration information, i.e. the set of variables pointing to the node at a particular time. This set of variables pointing to the node can be used as the node's name or identifier, which will be unique since a pointer variable or heap cell cannot point to two different locations at the same time. A shape node pointed to by at least one pointer variable or heap cell represents a unique cell in any given concrete or actual store.

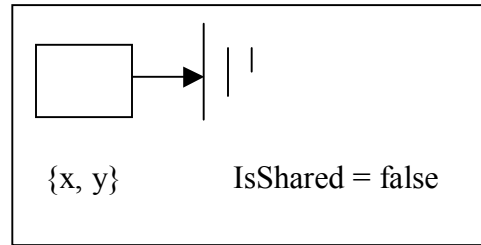


Figure 3: A Shape Node representing a node in a singly linked list, pointed to by variables x and y and pointing to null.

As mentioned earlier, in contrast to concrete stores, a shape graph must have a bounded size and this is achieved by having a summary node. The summary node, instead of representing a single concrete cell (as general shape nodes do), will represent multiple cells of a single concrete store that are not pointed to by any pointer variable or heap cell. The summary node has as name the empty set, which means that none of the cells represented by the node is pointed to by a pointer variable or heap cell.

Each shape node also has a Boolean flag *is-shared* associated with it. The flag helps us to distinguish between shared and unshared shape nodes. Whenever this flag is true, it means that the heap cell (or heap cells in the case of the summary node) represented by the node may be the target of pointers emanating from two or more distinct cell fields. This flag is especially useful for summary nodes. Summary nodes always point to themselves, because the cells they represent may be linked to each other. But if the summary node is unshared, it still tells us something useful about the shape of the linked structure: the heap cells it represents are unshared, and therefore cannot be linked circularly.

3.4.4 Summarization of a shape node

The name of a shape node in a shape graph consists of the set of the variables that point to the shape node at that time. Whenever a pointer variable or heap cell no longer points to a shape node, its name is removed from the name of that shape node. This means that whenever a shape node is not pointed to by any pointer variable or heap cell, its name becomes the empty set, which is the same as the name of the summary node. As soon as the name of a shape node becomes the empty set, that shape node is merged with the summary node. This process is known as summarizing the shape node. It is important to note that all cells pointing to the summarized node will now point to the summary node. Likewise, the summary node will point to all the nodes that the summarized node was pointing to.

3.4.5 Materialization of a shape node

When a pointer variable or cell is redirected to a concrete cell already represented by a shape node that is not the summary node in a shape graph, its name is just added to the shape node's name. However when a shape node other than the summary node does not already represent that concrete cell, we need to "unsummarize" a shape node from the summary node and add the name of the variable or cell pointing to it to its name. This process is known as materializing a shape node and allows the static shape graph to cover all possible configurations of storage. This materialized node will point to the summary node.

3.4.6 Transfer functions

The transfer functions to be used in the shape analysis follow. Please refer to [5] for a more formal description and examples of how shape graphs are transformed by each function.

Transfer function for Boolean tests. Boolean tests do not modify the heap; therefore they do not have any impact on the shape graph.

Transfer function for $x = nil$. If x is not present in any shape node in the graph, then the graph is unmodified; otherwise x is removed from the node it was pointing to. The node x was pointing to will be summarized if x was the only variable pointing to it because its name will become the empty set after removing x .

Transfer function for $x = y$. If x is equal to y , then the graph remains the same. If x is not equal to y , then the first step is to remove x from the node it was pointing to. The second step is to add x to the set of the shape node to which y points to (i.e. the shape node containing y in its name set), if any. It is important to note that even when x and y will point to the same cell, the cell will not be considered as shared. The reason is that the sharing information only records sharing in the heap and not sharing via state [5]. In other words, only cells pointed to by more than one cell field will be considered to be shared.

Transfer function for $x = y.sel$. This statement is equivalent to a sequence of three assignments: $t = y.sel$, $x = t$ and $t = nil$, where t is a temporary variable. If there is no node pointed to by x in the shape graph, then the statement will not have any effect on the graph. If there exists a node pointed to by x , then we start by removing x from that node's set name. If y does not point to any node in the graph, we are done; otherwise one of three possibilities may occur: 1) $y.sel$ is nil , 2) $y.sel$ points to a node that is not the

summary node or 3) $y.sel$ points to the summary node. If $y.sel$ is nil , then the graph will not suffer any other modification. If the second possibility arises instead, we proceed to add x to the node to which $y.sel$ points to. If $y.sel$ points to the summary node, then a node needs to be materialized and x is added to this materialized node.

Transfer function for $x.sel = nil$. First, if there is no node pointed to by x , then the statement does not have any effect on the graph. On the other hand, if there exists a node pointed to by x , then we proceed to check whether $x.sel$ points to a node. If $x.sel$ does not point to any node we are done, however if it does point to a node we need to remove the link between the node pointed to by x and the node pointed to by $x.sel$. If the node pointed to by $x.sel$ was shared, then we need to check whether this condition has changed such as in the case where there is at most one pointer left and it does not have source the summary node in which situation the node will become unshared.

Transfer function for $x.sel = y$. This statement is equivalent to $t = y$, $x.sel = t$ and $t = nil$ where t is a temporary variable. The statement does not have any effect on the graph if x does not point to any node in the graph. If there exists a node pointed to by x , then we check whether any node is pointed to by y . If there is no node pointed to by y , then we have the transformer $x.sel = nil$ discussed earlier. If there exists a node pointed to by y , we need to remove the link from $x.sel$ and the node it points to, if any. Then we proceed to add a link from the node pointed to by x to the node pointed to by y . We need to update the shared information of the node pointed to by y because it might have become shared.

Transfer function for $x.sel = y.sel$. This statement is equivalent to the sequence of assignments: $t = y.sel$, $x.sel = t$ and $t = nil$ where t is a temporary variable. As discussed earlier, if x is not pointing to any node in the graph, the statement does not have any

effect on it. If x points to a node, then we check for y . If y or $y.sel$ are nil, then we have the transformer $x.sel = nil$; otherwise we also remove any link between the node pointed to by x and the node pointed to by $x.sel$ and add a link between the node pointed to by x and the node pointed to by $y.sel$. The shared information of the node pointed to by $y.sel$ needs to be updated.

Transfer function for $x = new X$. The old binding for x needs to be removed and a new node is introduced in the graph. This new node is unshared and x is added to its set name.

3.5 Related work

3.5.1 Class invariants

- Houdini is an annotation assistant that infers suitable ESC/Java annotations for an unannotated Java program [1]. This annotation assistant can derive invariants from unannotated Java code as well. The main difference between Houdini and the work discussed in this thesis is that Houdini does not derive invariants about the shapes of the linked data structures manipulated by a program.
- An analyzer of Java bytecode developed at the University of Wisconsin – Milwaukee [11] performs DC invariant analysis to identify class invariants in unannotated Java code. It has been applied to unary analyses such as null-pointer analysis and lattice-of-signs analysis. It has also been applied to a local method analysis called relational constraint. This method analysis finds binary relations among the variables and constants of a program by constructing a table of binary relations and treating the program as a collection of constraints on tuples of relations in the table [10,11]. The class invariant shape analysis discussed in this thesis is implemented in this analyzer

as well, however the main difference between shape analysis and the other implemented analyses is that none of them reveal information related to the shapes of the linked data structures manipulated by a program as shape analysis do.

3.5.2 Shape analysis

- Checking Cleanness in Linked Lists [16] describes an algorithm called SG+R that automatically discovers memory cleanness errors in C programs statically. The algorithm consists on performing interprocedural shape analysis to the program. They use the automatic call-string approach of PAG to handle procedures and mention that this approach is inadequate since it is both imprecise and expensive. This algorithm applies shape analysis as explained in [5, 6, 7] however each statement also has a cleanness precondition (a requirement that every store occurring at this statement must satisfy).

The main differences between the SG+R algorithm and our work is that for each graph that occurs before a statement they impose conditions that guarantee that the preconditions of that statement are met by every store represented by the graph. The tool checks for cleanness preconditions on the fly, displaying error messages as soon as cleanness violations are found. We do not associate preconditions to any statement and do not search for any properties in the shape graphs during analysis until we are done. They work with C programs whereas we are performing the analysis on Java programs. Moreover, as mentioned earlier, they use the automatic call-string approach to do the interprocedural analysis while we use the DC invariant analysis technique.

- [17] presents an interprocedural shape analysis algorithm for programs manipulating linked lists. The algorithm analyzes recursive procedures more precisely than existing algorithms. They follow the approach of summarizing activation records the same way list elements are summarized. The main idea is to make the runtime stack an explicit data structure and abstract it as a linked list (the entire heap and run-time stack are represented at every program point). It handles programs manipulating linked lists written in a subset of C. We handle programs manipulating linked lists in Java and our approach using DC invariant analysis is completely different. Also, they have solved a problem still open for us: the handling of recursive procedures.
- A relational approach to interprocedural shape analysis is introduced in [19]. Procedures are considered as transformers from the entire program heap before the call, to the entire program heap after the call. Every heap-allocated object is represented at every program point; on the other hand, only the values of the local variables of the current procedure are represented, which means that irrelevant parts of the heap are summarized to a single summary node during the analysis of an invoked procedure. This approach is really similar to our approach to handle method calls, however the difference again is that we are applying the DC invariant analysis technique.
- [20] presents a non-standard semantics for Java programs in which procedures operate on local-heaps reachable from actual parameters. This algorithm yields an algorithm more efficient than those existing shape analysis algorithms, however it is not that precise for programs with many sharing patterns. The similarity with our

work is that [20] also attempts to do an interprocedural shape analysis in Java programs, but following a different approach.

- The design and implementation of a framework for implementing the algorithm discussed in [20] for single threaded Java programs is provided in [18]. We have not experimented with multi-threaded Java programs, but our DC invariant analysis supports this in theory. So that would be a significant contribution.

4. CLASS INVARIANT SHAPE ANALYSIS

4.1 Description

The DC invariant technique can be seen as an amplification of any local method analysis since class invariants are built up from the information obtained from local method analysis. Therefore, the local method analysis is an interchangeable component of the class invariant analysis.

Class invariant shape analysis is an amplification of the shape analysis technique, used exclusively in local method analysis until now. In class invariant shape analysis, rather than applying shape analysis on isolated methods in order to find local invariants, we apply shape analysis to constructors and dirty-called methods and iterative shape analysis to the clean-called methods of the class until a fixed point is reached. Class invariants may then be built up from the information obtained as a result of the method analyses.

A more detailed description of class invariant shape analysis follows. We apply shape analysis individually to tight constructors, leaky constructors and dirty-called methods. Each method has an initial assertion, which is the set of shape graphs that may

arise at the point when the method being analyzed is called. The initial assertion for constructors is the empty set of shape graphs since reference fields are initialized to null. The initial assertion for dirty-called methods must be the weakest possible assertion, representing the fact that nothing is known about the shapes in the heap to which reference fields of the class may point. The union of the graphs resulting from these shape analyses will become the initial assertion for clean-called methods.

Once the constructors and dirty-called methods are analyzed, we proceed to analyze the clean-called methods. We apply an iterative shape analysis on them until a fixed point is reached. Each clean-called method is analyzed considering its initial assertion. After each iteration, the union of the graphs resulting from each method analysis will become the new assertion. The fixed point will be reached whenever this new assertion is equal to our initial assertion.

4.2 Implementation details

4.2.1 Existing components

The class invariant shape analysis is implemented on Webber's analyzer for Java bytecode previously discussed. The analyzer performs three main tasks. First, it decompiles class files into internal flow graphs. Second, the analyzer does a whole-program load and analysis in order to classify the fields as disciplined or undisciplined fields, the constructors as tight or leaky constructors, and the methods as clean-called or dirty-called methods. Third, the analyzer has the capability of inlining leaf calls. This feature has an important impact in the class invariant shape analysis implementation.

4.2.2 New components

A general description of the work done for this thesis follows:

ShapeNode class: creates instances of shape nodes, which make up the shape graphs that describe the heap contents. Each shape node has a name that is a set containing the name of the variables pointing to the shape node at that particular time. Shape nodes have one or more links to other shape nodes depending on the data structure to be represented (singly linked list, doubly linked list, etc). Shape nodes also have an *isShared* flag that will indicate whether the shape node is shared or not.

ShapeGraph class: creates instances of shape graphs. The shape analysis technique has the goal of producing a set of shape graphs for each point in the program. Each shape graph consists of one or more shape nodes.

Assertion class: an assertion is a possibly empty set of graphs that may arise at a specific point in the program. Assertions play an important role in the class invariant analysis. For example, when applying iterative shape analysis to clean-called methods, assertions will determine when the fixed point has been reached.

Shape Analysis: this is a static class that implements the local method analysis as well as the class invariant analysis.

Copy Elimination: does copy elimination in any given control flow graph.

4.2.3 Method shape analysis

We traverse the method's control flow graph in order to perform the analysis. A worklist is used to determine the order in which the blocks in the control flow graph will

be visited. Each block contains one or more bytecodes. Each bytecode knows how to transform the shape graphs arising at that point.

The method has an initial assertion and a final assertion. The initial assertion is the set of graphs that might arise at the point in which the method is called. The initial assertion may have zero or more shape graphs and the method must be analyzed considering each of them. The final assertion is the set of graphs produced during the analysis and that reach the general exit of the control flow graph. The general exit covers both normal and abrupt termination, which can also be analyzed separately.

4.2.4 Class invariant analysis

The class is loaded and each method is decompiled. After decompilation, we do leaf calls inlining. We implemented reaching definitions in order to do copy elimination. Doing copy elimination helped us to reduce the number of resulting shape graphs by about 65%.

The class invariant analysis treats constructors and dirty-called methods differently than clean-called methods, so it is necessary to do the classification before we start the analysis. The analysis starts by analyzing individually the tight constructors. We assume an empty initial assertion. The final assertion is the set of graphs at the normal termination block. The analysis then proceeds to analyze the leaky constructors whose initial assertion is also the empty set of shape graphs. Then we analyze the dirty-called methods. Their initial assertion is the weakest possible assertion. The final assertion for leaky constructors and dirty-called methods is the set of graphs at the general exit block (that includes abrupt and normal termination).

The union of the final assertion for each constructor and dirty-called method becomes the initial assertion for the first iteration of shape analysis to clean-called methods. It is important to note that the final assertion for each method is always simplified before it is union to the other final assertions. What do we mean by simplifying an assertion? An assertion is a set of shape graphs. Of course, there are not duplicate shape graphs in the set. As mentioned earlier, in the process of analyzing a method, temporary variables may be introduced into the graphs due to the fact that a statement in the program is usually internally represented by a sequence of Java opcodes (the operation is not done in a single step). So when we talk about simplifying an assertion we mean simplifying the shape graphs in the assertion by removing all temporary variables introduced during the method analysis as well as removing all local variables of the method.

Having a simplified final assertion reduces significantly the number of shape graphs in the assertion, which makes the analysis of clean-called methods faster. Moreover, simplifying the shape graphs helps us to keep only that information we are interested in: the DC fields of the class. So the process of simplification throws away all the local information we are not interested in.

The second phase of the analysis involves analyzing the clean-called methods. Like constructors and dirty-called methods, clean-called methods also have an initial and final assertion. The initial assertion contains all the shape graphs that may arise at the point when the method is called, therefore all of them must be considered when analyzing the method. Likewise, all those new graphs produced during the analysis that reach the method's general exit are added to the method's final assertion.

The union of final assertions is performed after all clean-called methods have been analyzed. The result is the final assertion from this iteration. If this new assertion is equal to our initial assertion, that means we have reached the fixed point. Otherwise, the new assertion becomes the initial assertion and we iterate through the clean-called methods again. For efficiency, the analysis remembers the shape graphs contained in the previous initial assertion so that the clean-called methods are re-analyzed only on those newly added graphs.

Once we have reached the fixed point, that final assertion (class assertion) is the class invariant. It represents class-invariant shape information about every DC field of the class. It can be a complex collection of shape graphs, but we can use it to test for a variety of specific, simple properties. For example, if we are interested in knowing whether the data structure is acyclic, we can write a method that checks whether a shape graph is acyclic. Then we can check whether all the shape graphs in the class assertion are acyclic, if so we can conclude that it is acyclic. If one or more of the shape graphs represent a cycle in the store, then the answer will be that it is not acyclic.

We have implemented a couple of methods that use the class invariant to test for such properties. They include the previously discussed method to find out whether the data structure is acyclic, a method to check for null values, a method to check for sharedness and a method to check whether two linked lists are disjoint.

4.3 Problems faced during implementation

The main problem was how to associate each statement in the program with the corresponding graph transformer. The smallest unit in the analyzer is the opcode and a

single statement in the program usually consists of more than one. We tried three different approaches to solve the problem. These approaches are discussed below.

The first approach was to identify the group of opcodes that corresponded to each statement in the program and whenever that group was found, apply a specific transformer to the shape graph. The problem arose when groups of opcodes were split into two or more blocks in the control flow graph. It was hard to group the opcodes in this situation. Difficult but not impossible, this approach was still too particular and seemed to fail in more complex programs.

In the second approach, we tried to keep a list of transformers in each block. These transformers were those corresponding to the opcodes in that specific block. We found the same problem as in the first approach when attempting to group the opcodes. Even when the opcodes were successfully grouped, it was not clear what to do with those transformers belonging to more than one block. One solution was to merge or split blocks in the control flow graph as necessary, but that did not work either.

The third approach was to treat each opcode as the smallest unit (as they are) and associate a transformer with each of them. As mentioned earlier, a statement in a program usually consists of several opcodes. The problem we faced was that temporary variables are used in the opcodes before we get the final result, so applying the transformers did not produce the expected shape graphs since we were not handling those temporary variables correctly. We then came up with the idea of giving special names to these temporary variables so that we could introduce them into the shape graph and later be able to remove them from the graph. These temporary variables allowed us to transform

the shape graphs correctly. Removing them from the shape graphs helped to reduce the number of different graphs obtained at the end of the analysis.

Another problem we faced was related to iterating through loops. In our first approach, we were able to specify how many times to iterate, but of course it was unrealistic. Then we successfully applied the standard techniques of forward data flow analysis. (This aspect of shape analysis is not explicit in [7], but is clarified in [5]) Each block in the control flow graph has an initial set of shape graphs, which is the union of sets of shape graphs from the blocks we are coming from. Whenever the sets do not change, we stop iterating.

The last problem arose when we attempted to analyze a method containing calls to other methods. The approach we follow is to map the actual parameters to the formal parameters and take the current set of shape graphs as the initial set for the method being called. Once this method is analyzed, then the final set of shape graphs is returned and the analysis of the caller method can continue. This approach seems to work on our implementation, however we would prefer to inline any method calls to have a more general solution that could even be a contribution to the program analyzer and be used for other program analysis in the future. Leaf calls inlining has already been added to the analyzer to support this project. For non-leaf calls we still follow the approach we just discussed.

5. EXAMPLES

5.1 Class IntStack

Our first example illustrates the class invariant shape analysis of a class IntStack. IntStack is a stack of integers implemented using a singly linked list. As this is our first example, the analysis description will be very detailed. At the end of the class invariant analysis we will conclude whether any class invariants were identified.

5.1.1 Source code

Below the Java source code for classes IntStack and ConsCell.

```
public class IntStack {
    private ConsCell top; // top of the stack or null

    /**
     * Construct a new Stack given its first ConsCell.
     * @param s the first ConsCell in the Stack or null
     */
    public IntStack() {
        this.top = null;
    }

    /**
     * Pushes i onto the top of the Stack
     * @param i the integer to be added onto the Stack.
     */
    public void push(int i) {
        top = new ConsCell(i, top);
    }

    /**
     * Returns the object at the top of the Stack and removes it.
     */
    public ConsCell pop() {
        ConsCell e = null;
        if (top != null) {
            e = top;
            top = top.getTail();
        }
        return e;
    }

    /**
     * Returns the object at the top of the Stack without removing it.
     */
    public ConsCell peek() {
        return top;
    }
}
```

```

/**
 * A ConsCell is an element in a linked list of
 * ints.
 */
public class ConsCell {
    private int head; // the first item in the list
    private ConsCell tail; // rest of the list or null

    /**
     * Construct a new ConsCell given its head and tail.
     * @param head the int contents of this cell
     * @param tail the next ConsCell in the list or null
     */
    public ConsCell(int head, ConsCell tail) {
        this.head = head;
        this.tail = tail;
    }

    /**
     * Accessor for the head of this ConsCell.
     * @return the int contents of this cell
     */
    public int getHead() {
        return head;
    }

    /**
     * Accessor for the tail of this ConsCell.
     * @return the next ConsCell in the list or null
     */
    public ConsCell getTail() {
        return tail;
    }

    /**
     * Mutator for the head of this ConsCell.
     * @param head the new int in this ConsCell
     */
    public void setHead(int head) {
        this.head = head;
    }

    /**
     * Mutator for the tail of this ConsCell.
     * @param tail the new tail for this ConsCell
     */
    public void setTail(ConsCell tail) {
        this.tail = tail;
    }
}

```

5.1.2 Control flow graphs, opcodes and transformers

The analyzer decompiles Java class files into internal control flow graphs. Each control flow graph consists of a group of blocks where each block contains one or more Java opcodes. The control flow graph corresponding to the method *push* is illustrated in

Figure 4. A crossed out Java opcode represents an opcode that is removed after copy elimination.

```
/**
 * Pushes i onto the top of the Stack
 * @param i the integer to be added onto the Stack.
 */
public void push(int i) {
    top = new ConsCell(i, top);
}
```

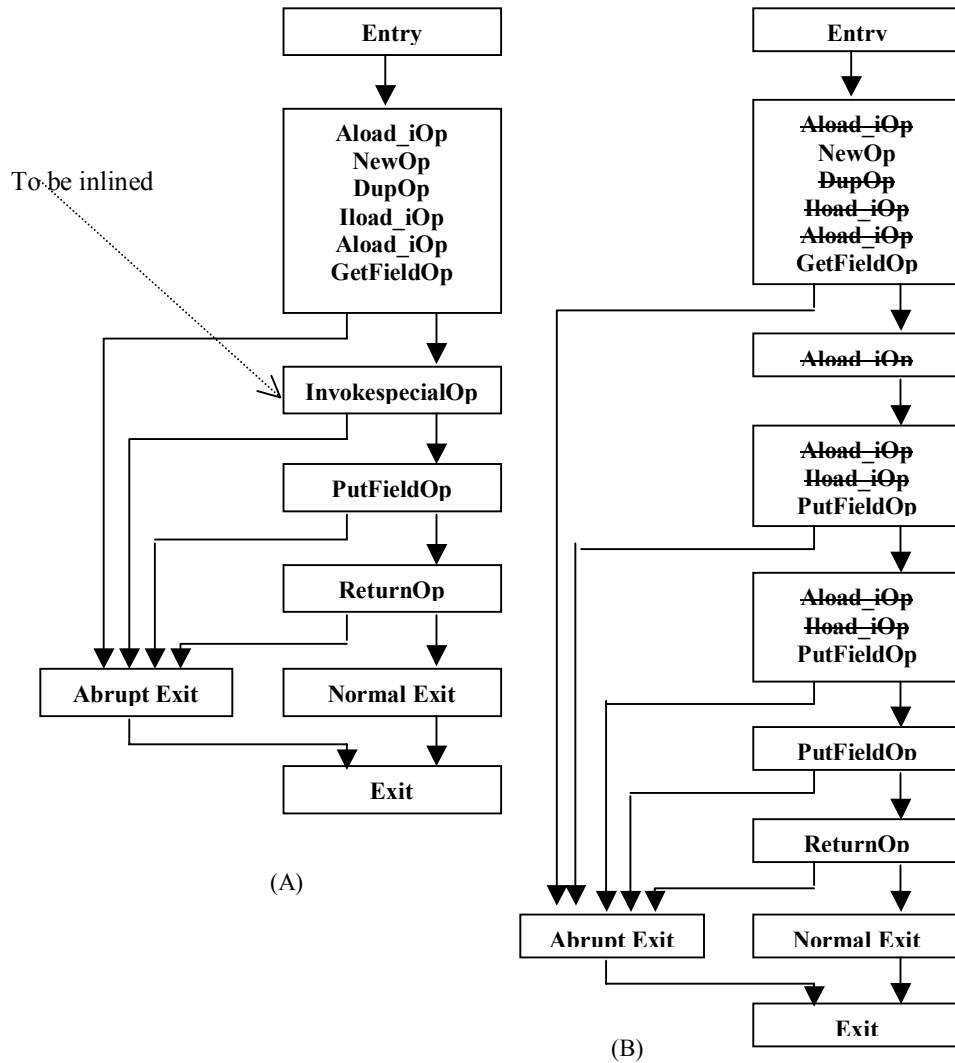


Figure 4: (A) Control flow graph for method push. (B) Control flow graph for method push after inlining.

It is important to note that in our implementation, the name *local0* refers to the field *this*. The name *local1* refers to the field *top*. Figures 1, 2, 3, 4 and 5 contain the opcodes and transformers for each method in class *IntStack*.

Opcode	Transformer
PutFieldOp	local0.top = null
ReturnOp	None

Table 1: Opcodes and Transformers for constructor *IntStack*.

Opcode	Transformer
NewOp	_t1
GetFieldOp	_t2 = local0.top
PutFieldOp	_t1.head = local1
PutFieldOp	_t1.tail = _t2
PutFieldOp	local0.top = _t1
ReturnOp	None

Table 2: Opcodes and Transformers for method *push*.



Opcode	Transformer		
Astore_iOp	local1 = null		
GetFieldOp	None		
IfNullOp	None		
GetFieldOp	_t1 = local0.top		
Astore_iOp	local1 = _t1		
GetFieldOp	_t2 = local0.top		_t3 = _t2.tail
InvokeVirtualOp	_t2 = _t2.getTail()		_t2 = _t3
PutFieldOp	local0.top = _t2		
ReturnOp	None		

Table 3: Opcodes and Transformers for method *pop*.

Opcode	Transformer
GetFieldOp	_t3 = _t2.tail
ReturnOp	None

Table 4: Opcodes and Transformers for method *getTail*.

Opcode	Transformer
GetFieldOp	_t1 = local0.top
ReturnOp	None

Table 5: Opcodes and Transformers for method *peek*.

5.1.3 Class invariant shape analysis

We will assume the following classification of fields and methods for simplicity.

Later we will show an example in which a dirty-called method is present

DC fields:	1 (top)
Tight constructors:	1
Leaky constructors:	0
Dirty-called methods:	0
Clean-called methods:	3 (push, pop and peek)

PHASE 1. Analyzing tight constructors, leaky constructors and dirty-called methods.

We assume an empty class assertion at the entry to each constructor since it is the strongest assertion. An empty set of shape graphs is the strongest assertion because it means that no shapes in the store are pointed to by the variables of interest. This is a correct representation because the default initialization for reference fields in Java is null.

This example does not contain any dirty-called methods, however it is important to mention that the initial assertion for them would be the weakest assertion, i.e. the set of shape graphs representing all possible configurations of the heap contents. This is still an open problem when there are multiple variables in the set and multiple references to an object since the set of shape graphs representing all possible configurations would be very large, at least exponential in the number of variables.

Each constructor and dirty-called method is analyzed independently. A possibly not empty assertion results from the analysis of each them. The union of these resulting assertions will become the new class assertion to be used when analyzing clean-called methods.

- Analyzing tight constructor:

Initial assertion: empty set of shape graphs.

Transformer(s): (See table 1)

local0.top = null -> the set of shape graphs remains empty.

Final assertion: empty set of shape graphs.

There are no other tight constructors or any leaky constructors and dirty-called methods to analyze, so we proceed to union the final assertion from each analyzed constructor and dirty-called method (in this case we only have a final assertion from the tight constructor), which results in an empty set of shape graphs.

PHASE 2. Analyzing clean-called methods.

We do an iterative shape analysis to clean-called methods. The class assertion is updated at the end of each iteration by performing the union of all the final assertions produced by the analysis of each clean-called method and the class assertion itself. We keep iterating until the resulting class assertion is equal to the class assertion at the beginning of the iteration.

At this point in this example, some graphs will start being introduced to the set. Throughout this and other examples, we will use the following notation:

- A rectangle node models the abstract representation of a cell in the heap. In this example, the rectangle is divided into two parts to denote the fact that node being represented consists of two fields: an integer and a reference to another node. Each node is identified by a name, which is the set of variables pointing to it. [5, 6, 7, 8] use an arrow pointing to a node per each variable pointing to it besides including their names in its name set. We omit these arrows for the sake of simplicity.
- The summary node will be represented the same way ‘normal’ shape nodes are, however we will identify it by \emptyset , which denotes the empty set. The summary node

represents one or more shape nodes, which are not pointed to by any variables at that time. The nodes represented by the summary node still have as many fields as ‘normal’ shape nodes do. However it is important to note that since the summary node may represent several shape nodes at the same time, even when the shape nodes represented by it have a single link to another shape node, more than one link may emanate from the summary node.

- As [5], we will use a double rectangle to represent the fact that a node’s is-shared flag is set to true.
- If the reference field of a node does not point to any other node, it means it is null.

ITERATION 1

- Analyzing method *push*.

The initial assertion is the class assertion resulting from the analysis of constructors and dirty-called methods: an empty set of shape graphs. Table 2 has a list of transformers for the method *push*. Figure 5 shows how each of these transformers modifies the shape graph. Figure 6 shows the final assertion.

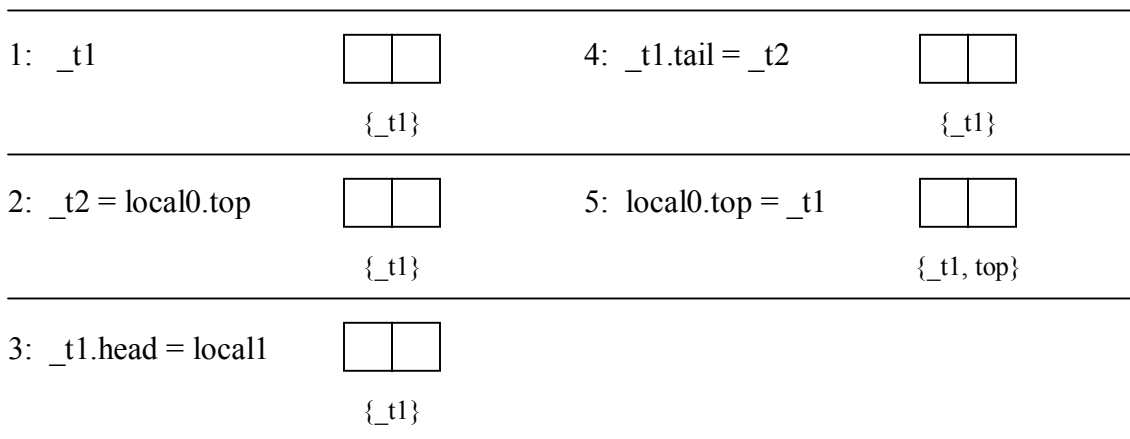


Figure 5: Shape graphs corresponding to the analysis of method *push*.

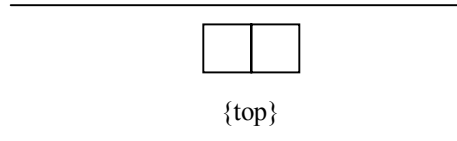


Figure 6: Assertion after analyzing method *push*.

If we look at the control flow graph for method *push*, the opcodes corresponding to the first and second transformer are in the same block while each of the other opcodes corresponding to the rest of the transformers are found in different blocks. It is important to note that all the blocks containing these opcodes are connected to the abrupt termination block. This means that all the graphs except the one generated by the first transformer will reach the general exit block. Why does the final assertion only contain one graph? Shouldn't it contain four graphs then? It should contain two graphs in any case because the assertion is a set of shape graphs, so there are no duplicate graphs. The second, third and fourth graphs are identical, so the set just contains it once and the fifth graph. But the final assertion does not even contain two shape graphs?

We are interested in finding class invariants, so the information we are interested in is the information related to the DC fields, the rest of the information can be dropped. Here is where we do some simplification in order to avoid having graphs that seem to be different because of temporary or local variables but that provide exactly the same information about the DC fields.

So, if we remove the temporary name `_t1` from the second, third or fourth shape graphs (it will be once in the set), the node is not pointed to by any variable anymore and since this node is not pointed to by `top` (our DC field) or points to a node to which `top` is pointing to, we can get rid of it because it does not represent any useful information to us. When removing `_t1` from the fifth graph, we get a node pointed to by `top` and we keep it.

Then we place the graphs resulting after the name removals (there is only one) in a set and that set becomes our final assertion. Figure 6 shows this final assertion containing only one graph.

- **Analyzing method *pop*.**

The initial assertion is an empty set of shape graphs since that is still our class invariant. Table 3 shows the transformers for the method *pop*. Figure 7 shows how no one of this transformers introduce anything new to the empty graph, therefore the final assertion is still a empty set of shape graphs.

1: local1 = null	5: _t3 = _t2.tail
2: _t1 = local0.top	6: _t2 = _t3
3: local1 = _t1	7: local0.top = _t2
4: _t2 = local0.top	

Figure 7: Shape graphs corresponding to the analysis of method *pop*.

- **Analyzing method *peek*.**

The initial assertion is the empty set of shape graphs. The method *peek* as a single transformer, which does not introduce anything new to an empty graph, so the final assertion is also an empty set of shape graphs.

_t1 = local0.top

Figure 8: The transformer in method *peek* does not introduce anything to an empty shape graph.

- Union of final assertions and class assertion.

Class assertion = class assertion \sqcup final assertion from method push \sqcup final assertion from method pop \sqcup final assertion from method peek =

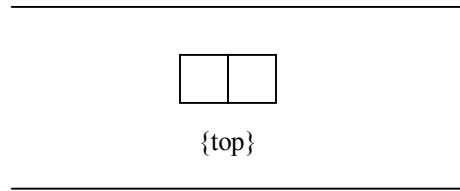


Figure 9: The new class assertion, a set containing one shape graph.

The new class assertion in Figure 9 and the class assertion at the beginning of this iteration (which was the empty set of shape graphs) are not equal, so we iterate through the clean-called methods again. This new class assertion says that the stack may have one element and it is pointed to by *top*.

ITERATION 2

- Analyzing method *push*.

The initial assertion is the class assertion in Figure 9. Figure 10 contains the graphs produced during the analysis of the method *push* when the initial assertion consists of one shape graph with a node pointed to by *top*.

We are analyzing the method *push* again. As explained during the first iteration, all the shape graphs generated during the analysis except the one generated by the first transformer will reach the general exit block. The final assertion contains initially three graphs: one representing the graphs produced by the second and third transformers, which are identical and other two graphs resulting from the fourth and fifth transformers.

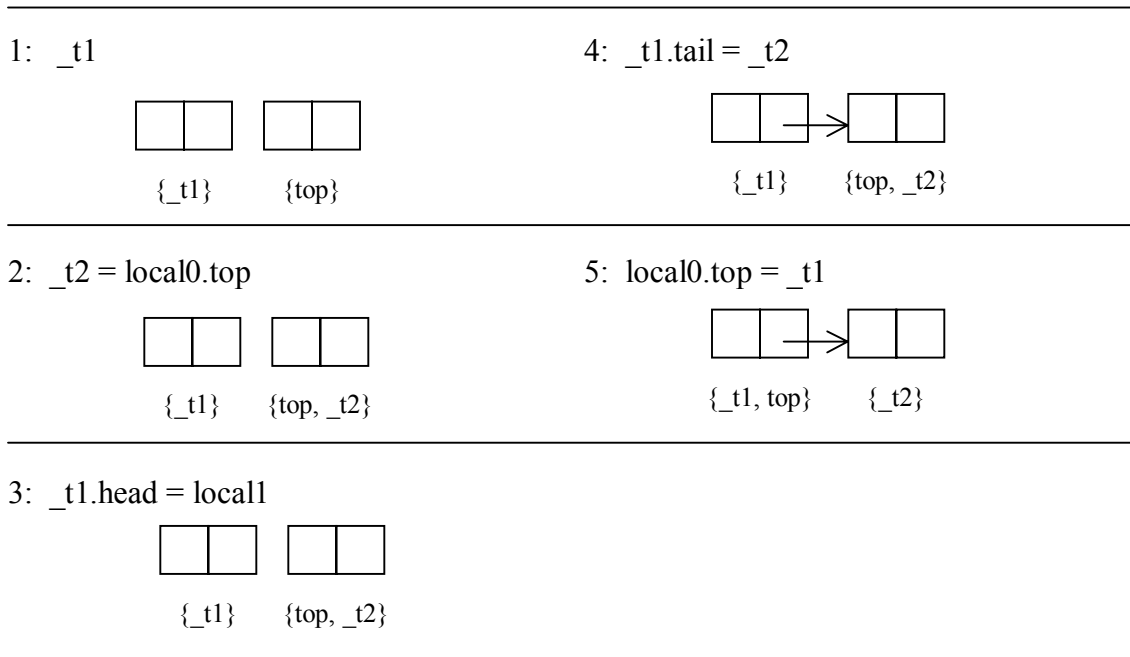


Figure 10: Shape graphs corresponding to the analysis of method *push*.

Removing the temporary `_t1` and `_t2` from the shape graph corresponding to the second transformer we get a shape graph containing a single node pointed to by *top*. From the fourth graph we get a node pointed to by nothing so it gets summarized into the summary node. The summary node will point to the node pointed to by *top*. The fifth graph results in a shape node pointed to by *top*, which points to some node (the one previously pointed to by *top*) that has just been summarized into the summary node since no variable points to it anymore. This graph tells us that the stack may have two elements where *top* points to the first one in the stack (the one at the top).

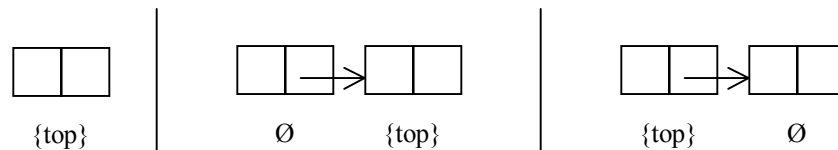


Figure 11: Final assertion after analyzing method *push* during the second iteration.

As it can be seen, the final assertion adds some new information: the first and second graphs in Figure 11 illustrate the possible shapes of heap in the case of abrupt termination. The third graph shows the shape of the heap when the method has a normal termination. For the sake of simplicity, we will not introduce the graphs reaching the exit block through the abrupt termination block in this example, however it is important to note that we do keep them in the implementation of the analysis and we get the same class invariants identified at the end of the analysis.

- **Analyzing method *pop*.**

Figure 9 shows the initial assertion, which is the set containing a single shape graph that consists of a shape node pointed to by *top*. Figure 12 contains the shape graphs produced during the analysis of the method *pop*.

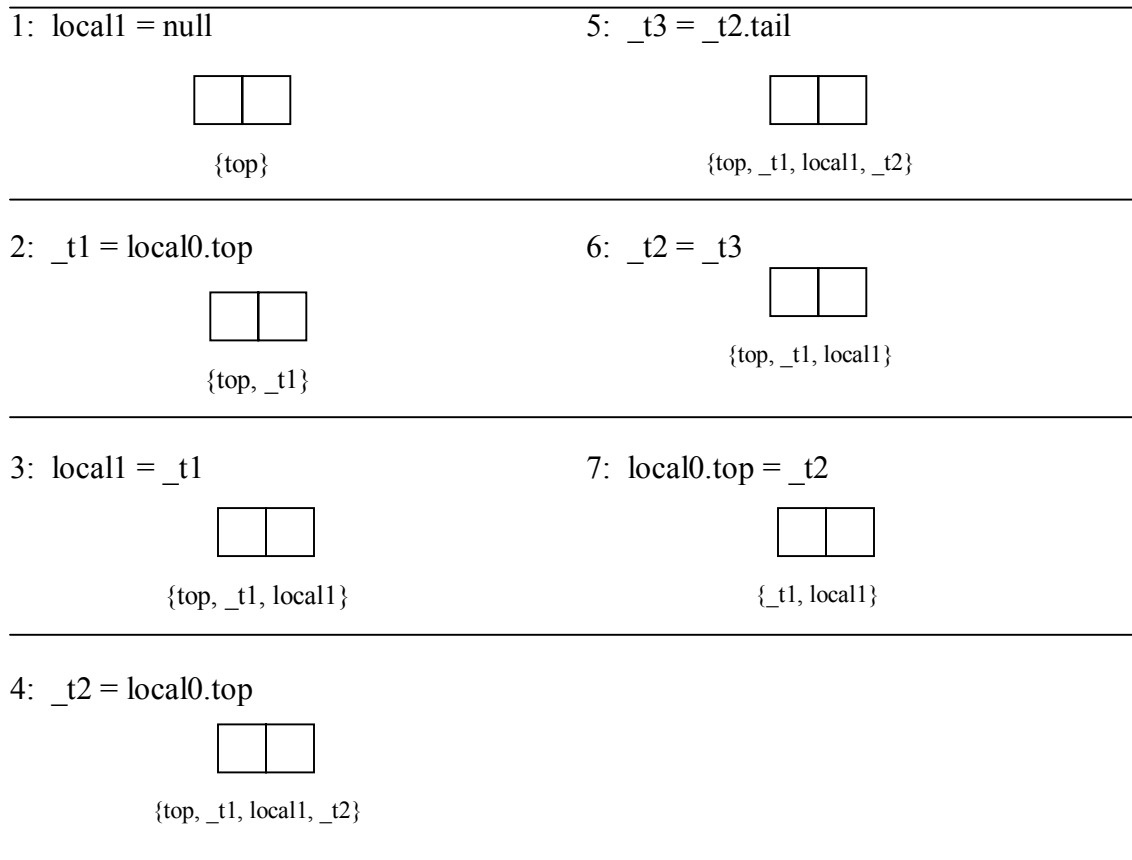


Figure 12: Shape graphs corresponding to the analysis of method *pop* in the second iteration.

If we only considered the very last graph produced, then after removing `_t1` and `local1` no variable would be pointing to the node and we could end up with an empty final assertion. This makes sense since we are popping off an element from the stack and our initial assertion contains a single graph that has a single shape node pointed to by `top`, which means that the stack has at most one element, so popping it off means that the stack does not have any element any more and the shape graph is empty. However, the method includes the various paths of abrupt termination (even when these paths are not taken in any possible execution). Through these paths, other graphs besides the last one reach the general exit block. After removing temporary and local variables from those graphs we get as final assertion a set containing a shape graph. This graph reflects the fact that the stack may still contain that one element when popping it off has failed because of possible abrupt termination.

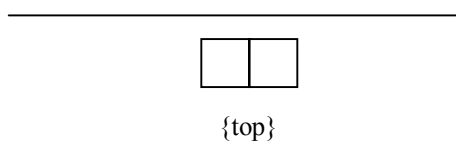


Figure 13: Assertion after analyzing method *pop* during the second iteration.

- **Analyzing method *peek*.**

The initial assertion is still the set containing a single shape graph whose only shape node is pointed to by `top`. Figure 14 shows the shape graphs produced during the analysis of the method *peek*.

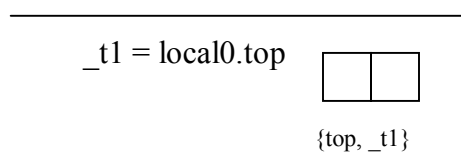


Figure 14: The shape graph produced by method *peek* in the second iteration.

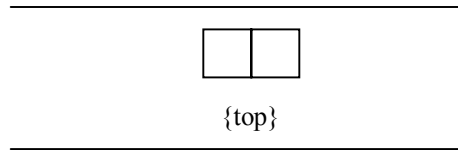


Figure 15: Assertion after analyzing method *peek* during the second iteration.

- Union of final assertions and class assertion.

The union of the class assertion, final assertion from method *push*, final assertion from method *pop* and final assertion from method *peek* is shown in Figure 16.

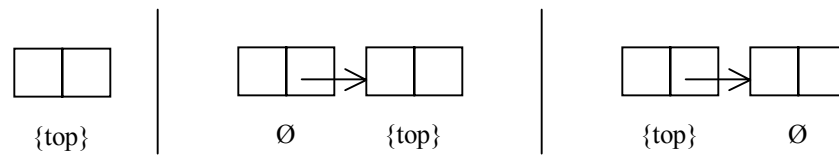


Figure 16: Class assertion with graphs from abrupt termination.

As mentioned before, in this example we will only consider those graphs that reach the exit block from the normal termination. The first graph reaches the exit block in the analysis of the method *peek* so we keep it. The second graph reaches the exit block through the abrupt termination block in the analysis of the method *push*, so we do not consider it. The third graph reaches the general exit block in the analysis of the method *push*, so we keep it. Figure 17 shows the class assertion that we will use.

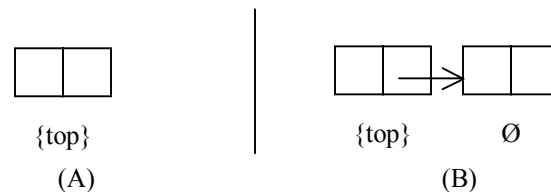


Figure 17: Class assertion without graphs from abrupt termination. (A) First graph in the set. (B) Second graph in the set.

The class assertion in Figure 17 is different to the initial class assertion in Figure 9, so we iterate through clean-called methods again.

ITERATION 3

The initial assertion in this iteration includes the two graphs presented in Figure 17, labeled as (A) and (B). We need to analyze each clean-called method considering both graphs. As you might have noticed, we have already analyzed the clean-called methods with graph (A) in iteration 2, so we will not show the shape graphs produced in these analyses again, however the final assertion for each clean-called method in this iteration will be the union of the final assertion when the graph (B) is at entry and the final assertion we obtained in iteration 2 when the graph (A) was at entry.

- Analyzing method *push*.

Figure 18 contains the shape graphs generated when graph (B) is at entry.

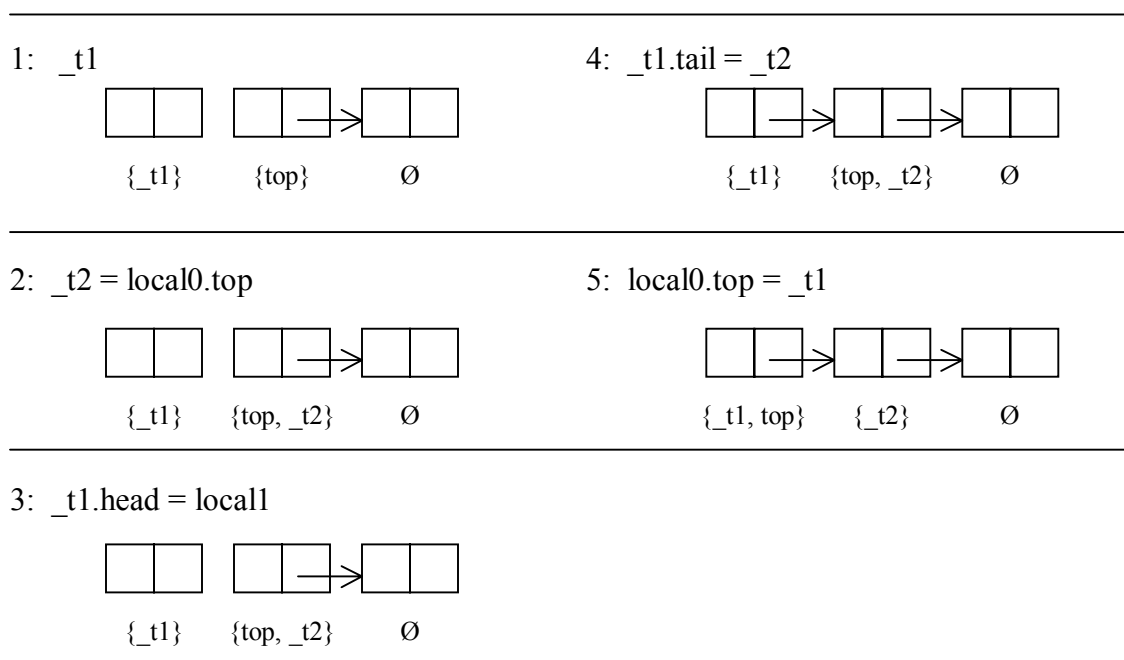


Figure 18: Shape graphs corresponding to the analysis of method *push* in the third iteration.

Here we analyze again the method *push*. Graph (b) tells us that the stack at entry to the method has two elements, the one in the front being pointed to by *top*. From

now on we will not take into account shape graphs that reach the general exit through the abrupt termination block, in order to make this example easier to illustrate, but it is important to note that including these graphs would not change any conclusions at the end of the analysis.

The final assertion will contain the last shape graph produced during the analysis of the method *push*. Now we need to remove temporary names and local variables. After removing *_t1* and *_t2* from the graph, the first node will only be pointed to by *top* whereas the second node will no longer be pointed by any variable. This second node will be summarized into the summary node yielding the graph (B) in Figure y. The graph (A) in Figure y comes from the final assertion when graph (A) is at the entry of the method.

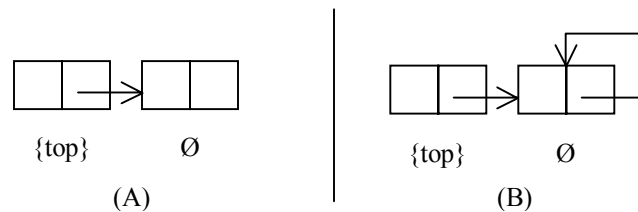


Figure 19: Final assertion after analyzing method *push*. (A) First graph in the set. (B) Second graph in the set.

- Analyzing method *pop*.

Here is the analysis when the graph (B) in Figure 17 is at the entry to the method. Figure y contains the final assertion (the final assertion when graph (A) is on entry is the empty set).

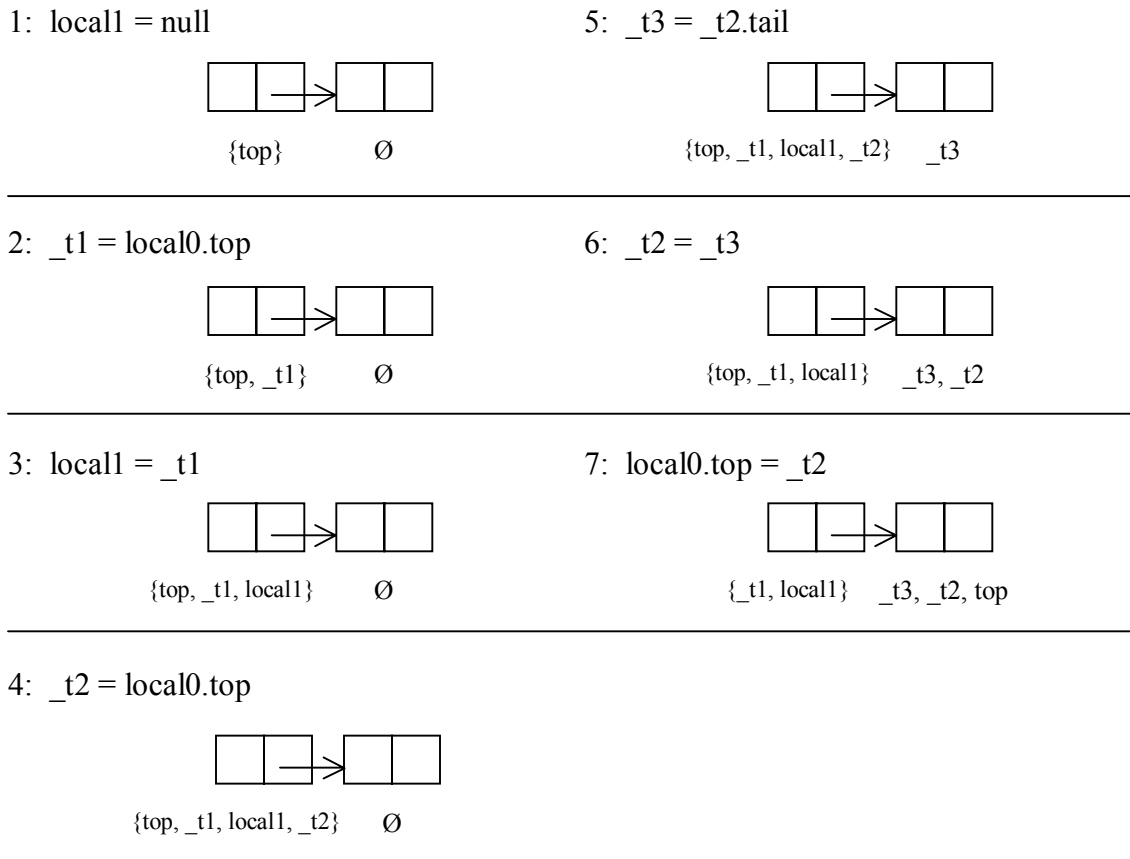


Figure 20: Shape graphs corresponding to the analysis of method *pop* in the third iteration.

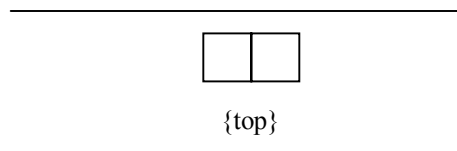


Figure 21: Final assertion after analyzing method *pop* during the third iteration.

- Analyzing method *peek*.

Figure 22 contains the shape graph produced during the analysis of method *peek* when graph (B) in Figure 17 is on entry. Figure 23 shows the final assertion, which includes those graphs obtained from the analysis of the method when the graph (A) in Figure 17 is at entry.

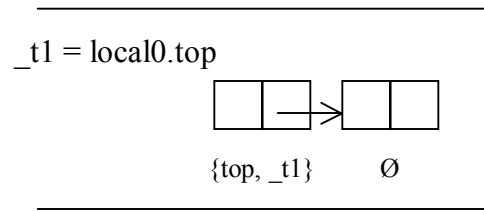


Figure 22: The shape graph produced by method *peek* in the third iteration.

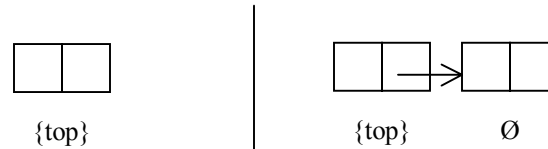


Figure 23: Final assertion after analyzing method *peek* during the third iteration.

- Union of final assertions and class assertion.

The union of the class assertion, final assertion from method *push*, final assertion from method *pop* and final assertion from method *peek* is shown in Figure 24.

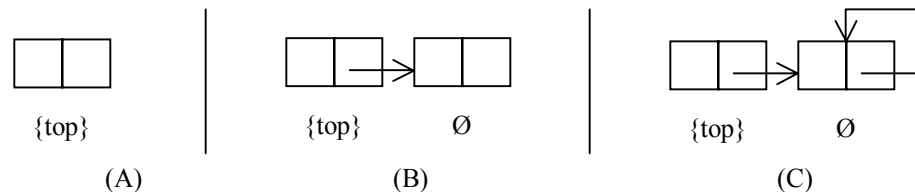


Figure 24: Class assertion reaching after the third iteration. (A) First graph in the set. (B) Second graph in the set. (C) Third graph in the set.

Since this new class assertion is not equal to the class assertion at the beginning of this iteration (the graph C has been introduced in this iteration) we need to iterate again through the clean-called methods.

ITERATION 4

In this iteration, we only show the analysis of the clean-called methods when the graph (C) in Figure 24 is on entry. The final assertion is the union of the final assertion from the analysis when the graph (C) is on entry and the final assertions when (A) and

(B) are the graphs on entry to the method (this information is taken from previous iterations).

- **Analyzing method *push*.**

Figure 25 contains the shape graphs generated when graph (B) is at entry.

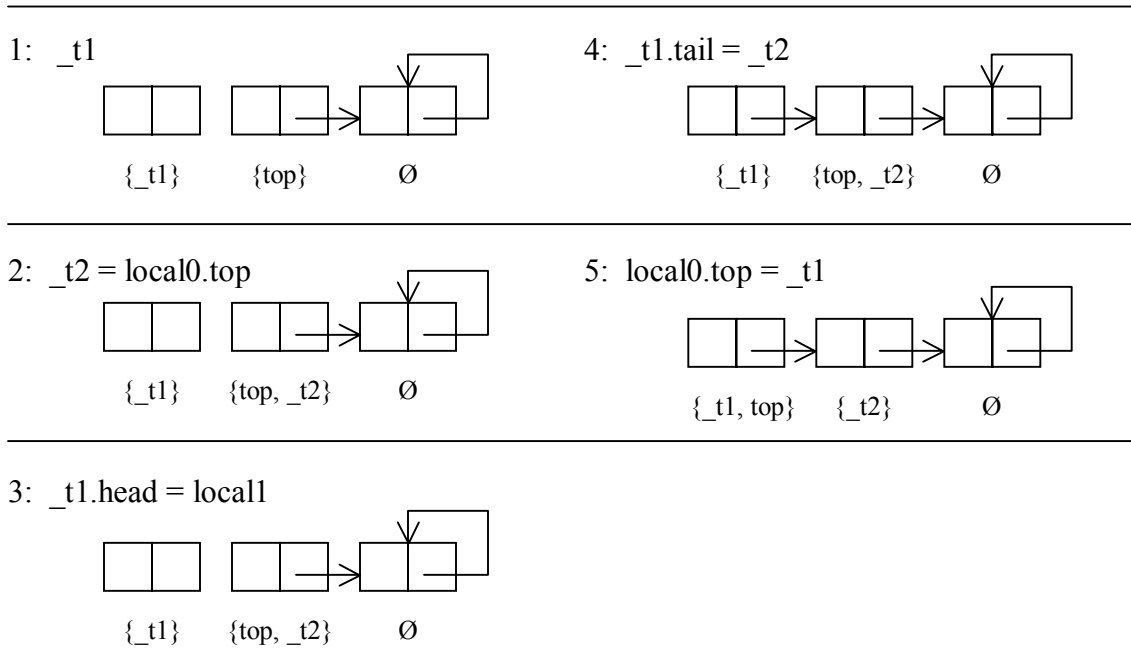


Figure 25: Shape graphs corresponding to the analysis of method *push* in the fourth iteration.

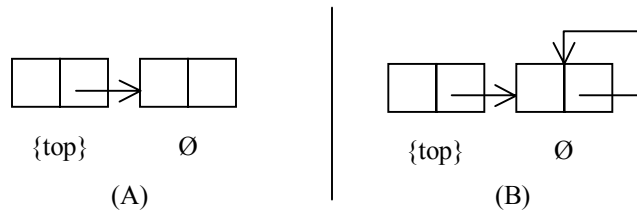


Figure 26: Final assertion after analyzing method *push*. (A) First graph in the set. (B) Second graph in the set.

- **Analyzing method *pop*.**

Figure 27 shows the shape graphs produced during the analysis when the graph (C) in Figure 24 is at the entry to the method. Figure 28 contains the final assertion.

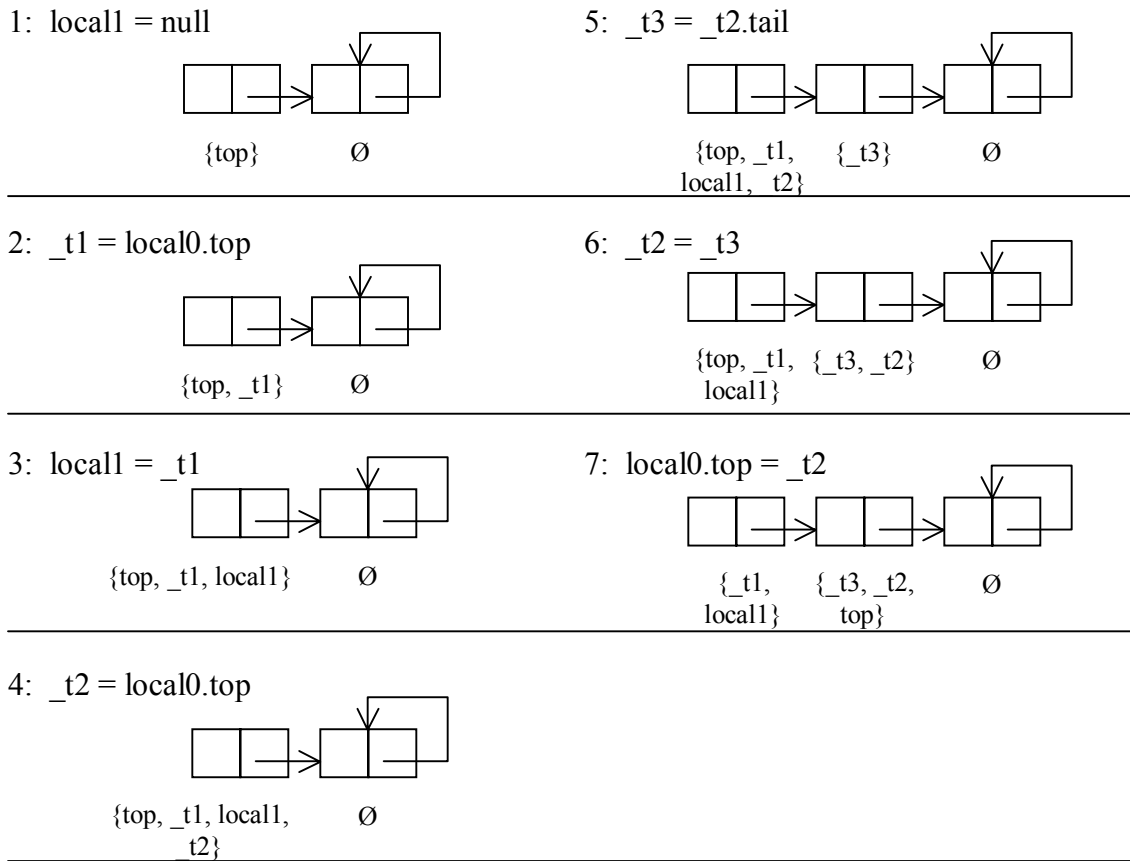


Figure 27: Shape graphs corresponding to the analysis of method *pop* in the fourth iteration.

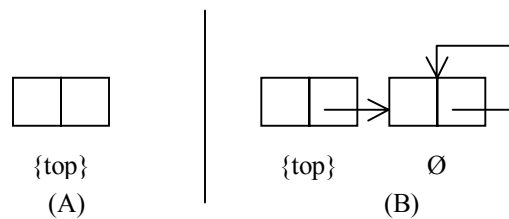


Figure 28: Final assertion after analyzing method *pop*. (A) First graph in the set. (B) Second graph in the set.

- Analyzing method *peek*.

Figure 29 contains the shape graph produced during the analysis of method *peek* when graph (C) in Figure 24 is on entry. Figure 30 shows the final assertion, which includes those graphs obtained from the analysis of the method when the graphs (A) and (B) in Figure 24 are on entry.

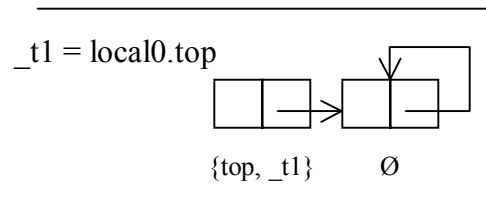


Figure 29: The shape graph produced by method *peek* in the fourth iteration.

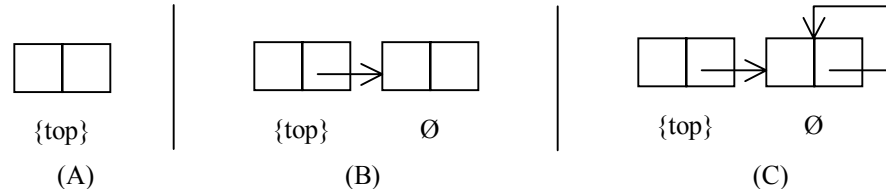


Figure 30: Final assertion after analyzing method *peek* during the fourth iteration.

Figure 30 shows the final assertion when the method *peek* is analyzed with graphs (A), (B) and (C) in the class assertion for this iteration in Figure 24.

- Union of final assertions and class assertion.

Class assertion = class assertion \sqcup final assertion from method *push* \sqcup final assertion from method *pop* \sqcup final assertion from method *peek* =

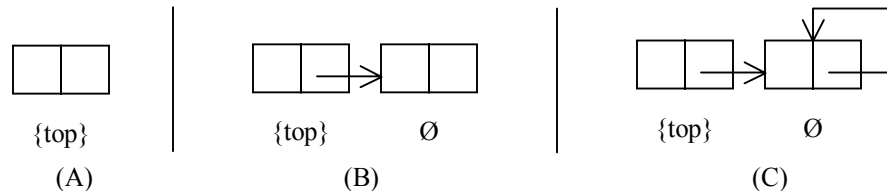


Figure 31: New class assertion after the fourth iteration.

The new class assertion and the class assertion at the beginning of this iteration are equal, so have reached the fixed point and we no longer need to iterate through the clean-called methods.

Now we can proceed to analyze the graphs in the final class assertion to search for any property we are interested in. By analyzing graphs (A), (B) and (C) in the final class

assertion we can conclude that the `IntStack` class has two class invariants: the singly linked list representing the stack internally is acyclic and its nodes are unshared. The fact that the nodes are unshared can be seen because there are no nodes represented by a double rectangle in the graphs, which would mean sharedness in our notation. A node becomes shared when more than one cell field points to it. As you might have noticed, this was not the case for any of the nodes in the shape graphs produced during the analysis.

5.2 Class `IntList1`

The class `IntList1` represents a singly linked list. In order to make this example clear and short, it only includes two methods: *add* and *reverse*. The method *add* adds a node to the front of the list. The method *reverse* reverses the list. We have borrowed the example of using a method *reverse* from [5,7] in order to illustrate that the results of our implementation are identical to those achieved by [7] and at the same time to show our contribution by applying shape analysis to the whole class in order to find class invariants.

5.2.1 Java source code

```
public class IntList1 {
    private ConsCell start;

    /**
     * Constructs a new empty IntList1.
     */
    public IntList1() {
        this.start = null;
    }
}
```



```

/**
 * Adds i onto the front of the IntList1.
 * @param i the integer to be added onto the IntList1.
 */
public void add(int i) {
    start = new ConsCell(i, start);
}

/**
 * Reverses the IntList1.
 */
public void reverse() {
    ConsCell y = null;
    ConsCell t = null;
    while (start != null) {
        t = y;
        y = start;
        start = start.getTail();
        y.setTail(t);
    }
    t = null;
    start = y;
}
}

```

Opcode	Transformer
PutFieldOp	local0.start = null
ReturnOp	None

Table 6: Opcodes and Transformers for constructor *IntList1*.

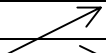
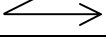
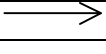
Opcode	Transformer		
Astore_iOp	local1 = null		
Astore_iOp	local2 = null		
GetFieldOp	None		
IfNullOp	None		
Astore_iOp	local2 = local1		
GetFieldOp	_t1 = local0.start		
Astore_iOp	local1 = _t1		
GetFieldOp	_t2 = local0.start		t3 = _t2.tail
InvokeVirtualOp	_t2 = _t2.getTail()		_t2 = _t3
PutFieldOp	local0.start = _t2		
InvokeVirtualOp	local1.setTail(local2)		local1.tail = local2
GotoOp	None		
Astore_iOp	local2 = null		
PutFieldOp	local0.start = local1		
ReturnOp	None		

Table 7: Opcodes and Transformers for method *reverse*.

We assume that `start` is a DC field, the constructor is a tight constructor and `add` and `reverse` are both clean-called. After the fourth iteration we reached the fixed point.

The class assertion contains the set of shape graphs shown in Figure 32.

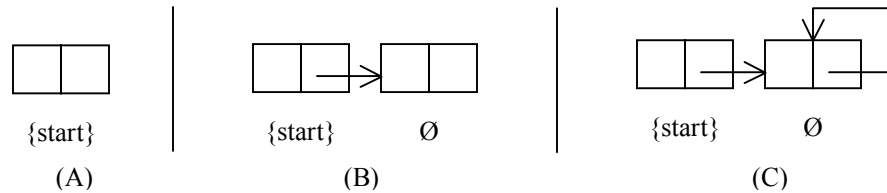


Figure 32: Class assertion after the fourth iteration.

After analyzing graphs (A), (B) and (C) we conclude that the singly linked list is acyclic and its nodes are unshared.

5.3 Class IntList2

```
public class IntList2 {
    private ConsCell start;
    /**
     * Constructs a new empty IntList2.
     */
    public IntList2() {
        this.start = null;
    }

    /**
     * Adds i onto the front of the IntList2.
     * @param i the integer to be added onto the IntList2.
     */
    public void add(int i) {
        start = new ConsCell(i, start);
    }

    /**
     * Makes the IntList2 circular.
     */
    public void make_circular() {
        if (start != null) {
            ConsCell x = start;
            while ((x.getTail() != null) && (x.getTail() != start)){
                x = x.getTail();
            }
            x.setTail(start);
        }
    }
}
```

The fixed point is reached at the fourth iteration with the set of shape graphs shown in Figure 33 being the class assertion.

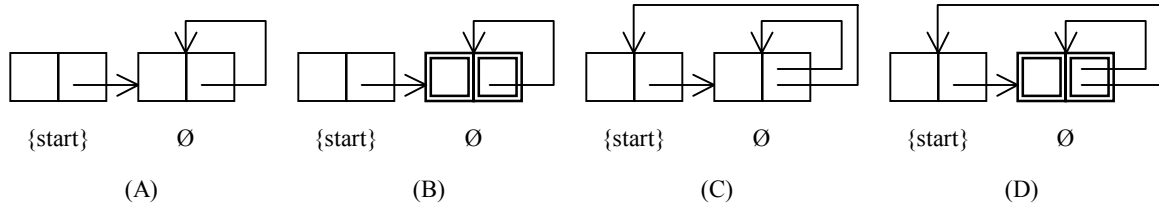


Figure 33: Class assertion after the fourth iteration in class IntList2.

After analyzing graphs (A), (B), (C) and (D) we can conclude that the singly linked list is not acyclic and its nodes are not unshared.

5.4 Example involving a dirty-called method

If the class to be analyzed contains one or more dirty-called methods, we need to construct the weakest assertion: the set of all legal shape graphs for the given set of variables. It sounds difficult to construct in general, when we have multiple variables in the set and multiple references in an object. Moreover, it would be large, at least exponential in the number of variables. This causes a serious problem with applying DC analysis to classes with dirty-called methods.

Figure 33 illustrates the graphs contained in the weakest assertion when there is a single variable in the set.

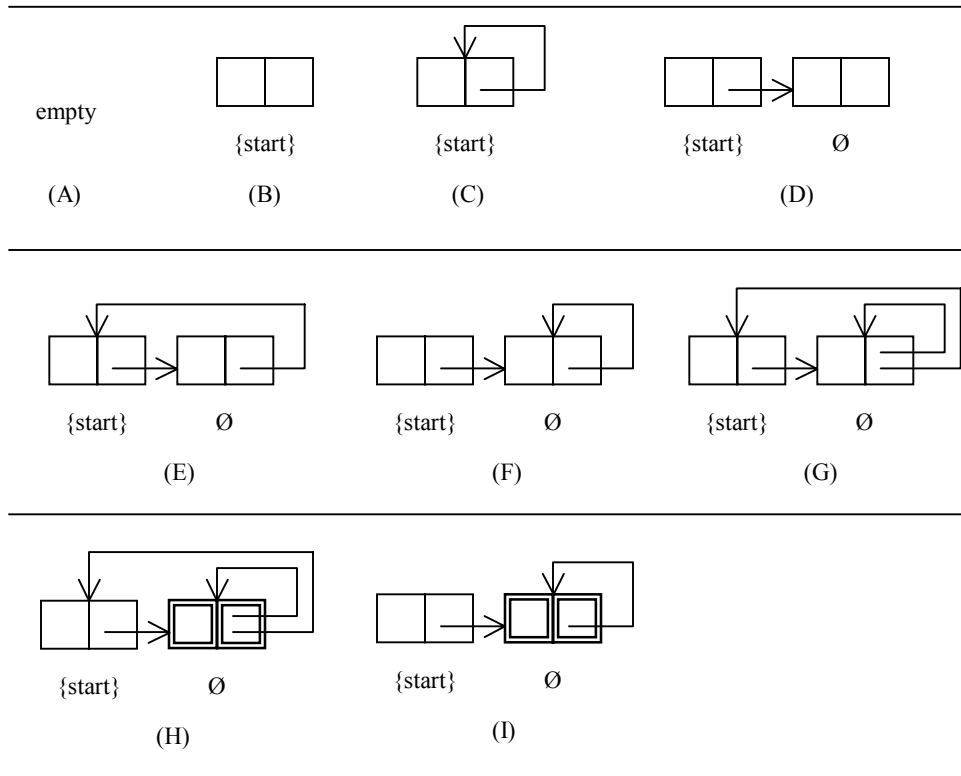


Figure 34: Weakest assertion in which there is only one variable.

It seems like it would be a good idea to look for an alternative way to handle dirty-called methods. We could check whether the dirty-called method modifies the heap contents. Another possibility would be if we know the invariants ahead of time: then we could check whether they have been broken when the dirty-called method was invoked.

6. FUTURE WORK

- In order to know whether this technique is practical we need to test the analysis on full-scale Java applications. Assess the DC idea is one of the goals in [22], so the ability to test the class invariant shape analysis on full-scale Java applications could contribute to it.
- Handling recursive methods. Our implementation cannot analyze applications with recursive methods. This is still an open problem for us. It is not an open problem in general as we saw in [17]. It is not clear to us whether the ideas discussed in [17] could be of utility when attempting to solve the problem in our implementation.
- Testing on multi-threaded programs. The DC invariant technique has the capability of dealing with multi-threaded programs. The class invariant shape analysis idea has not been tested on multi-threaded programs yet, so this would be another area of future work in our research.
- Dirty-called methods. As we showed earlier, dirty-called methods can be a serious problem when applying DC invariant analysis to classes that contain them. We have discussed some alternative solutions to the case in which we are doing shape analysis, however we need to formalize them and implement them to see whether they do contribute to resolve the problem.
- The implementation is not capable of constructing the weakest assertion. We have been injecting the graphs by hand (and only in the case when exactly one DC field exists). We need to add this feature in order to be able to perform complete test cases and obtain more reliable results.

- Extending the current implementation to handle a bigger variety of data structures that can be manipulated by the program.
- Make the analysis more efficient.

7. CONCLUSION

Class invariant shape analysis produced good results when analyzing classes not containing dirty-called methods. Dirty-called methods introduced the problem of constructing the weakest set of shape graphs, which is at least exponential in the number of variables. We conclude that it may not be worth it to use the full DC invariant analysis mechanism when searching for properties describing the shapes of the linked data structures manipulated by the program.

We definitely need to work on dirty-called methods. We believe that finding more selective ways of classifying dirty-called methods would be a significant improvement in the technique. It is currently possible for a method to be classified as dirty-called even though no shape invariant is broken at any of its points of call. In these cases, too many unnecessary graphs are introduced into the analysis, making it less precise and slower.

Analyzing full-scale Java applications would also reveal how often dirty-called methods are found, and whether it is worthwhile to work on the problem of reducing/eliminating false dirty-called methods. We also need to be able to produce the weakest assertion automatically so that we are able to measure real results.

REFERENCES

- [1] C. Flanagan, K. Rustan, and M. Leino. Houdini, an annotation assistant for ESC/Java. In: *International Symposium of Formal Methods Europe 2001: Formal Methods for Increasing Software Productivity*. Volume 2021 of Lecture Notes in Computer Science, pages 500-517. Springer, March 2001. Available at <http://gatekeeper.dec.com/pub/DEC/SRC/publications/rustan/krml100.ps>
- [2] G. T. Leavens. *JML home page*. At <http://www.cs.iastate.edu/~leavens/JML>
- [3] D. Luckham. *Programming with Specifications: An Introduction to Anna, a Language for Specifying Ada Programs*. Springer Verlag, 1991.
- [4] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991.
- [5] F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, pp.102-129, 1999.
- [6] T. Reps, M. Sagiv, and R. Wilhelm. Shape Analysis. In: *International Conference on Compiler Construction*. Springer Verlag, Lecture Notes in Computer Science, pp.1-16, no.1781, 2000.
- [7] T. Reps, M. Sagiv, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, Vol. 20, No. 1, pp. 1-50, 1998.
- [8] Y. N. Shrikant, and P. Shankar. *The Compiler Design Handbook: Optimizations & Machine code Generation*. CRC Press, pp.175-217, 2003.
- [9] A. B. Webber. What is a class invariant? In: *Proceedings of the 2001 ACM SIGPLAN - SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*, (Snowbird, Utah, June 18-19, 2001), pp. 86-89.
- [10] A. B. Webber. Program analysis using binary relations. In: *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pp. 249-260. Available at <http://www.webber-labs.com/research/pldi97>.
- [11] A. B. Webber. *Class-Invariant Assertions in Object-Oriented Programs*. NSF grant CCR-0073070.
- [12] A. Kaminsky. *Heaps and Garbage Collection*. Available at http://www.cs.rit.edu/~ark/lectures/gc/04_01_00.html

- [13] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. *Extended Static Checking*. Compaq SRC Research Report 159, 1998.
- [14] B. Jacobs and E. Poll, Java program verification at Nijmegen: developments and perspective. In: K. Futatsugi and F. Mizoguchi and N. Yonezaki (Eds.), *Software Security -- Theories and Systems (ISSS'03)*. Volume 3233 of Lecture Notes in Computer Science, pp. 134-153. Springer, 2004.
- [15] M. Ernst. The Daikon invariant detector. Available at <http://www.pag.csail.mit.edu/daikon>
- [16] N. Dor, M. Rodelh, and S. Sagiv. Checking cleanness in linked lists. In: *Proceedings of the 7th International Symposium on Static Analysis*, pp. 115-134. Springer-Verlag, 2000.
- [17] N. Rinetzky. Masters Thesis: *Interprocedural Shape Analysis*. Israel Institute of Technology, 2000.
- [18] N. Rinetzky, M. Sagiv and E. Yahav. *Interprocedural shape analysis using local heaps*. Technical Report TAU-CS-26/04.
- [19] B. Jeannot, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In: *Static Analysis Symposium*, 2004.
- [20] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. *A semantics for procedure local heaps and its abstractions*. Tech. Rep. 1, AVACS, September 2004. Available at <http://www.avacs.org>
- [21] A. B. Webber. Analysis of class invariant binary relations. Dagstuhl Seminar on Program Analysis. Hanne Riis Nielson and Mooly Sagiv, eds. Schloss Dagstuhl, Wadern, Germany. Apr. 12-16, 1999. Seminar No. 99151, Report No. 236.
- [22] L. Bonilla. *PhD Proposal*. University of Wisconsin-Milwaukee, 2003.