# Declassification: Transforming Java Programs to Remove Intermediate Classes

Bernadette Power B Sc

Submitted in fulfillment of the requirement for the degree of Master of Science in Computer Applications

Dublin City University

Supervisor Dr G W Hamilton

School of Computer Applications

February 2003

# Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of degree of master of science is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: Bernadette Power

ID No.: 98970569

Date: 06-February-2003

# Table of Contents

# Table of Figures

# Abstract

Computer applications are increasingly being written in object-oriented languages like Java and C++ Object-oriented programming encourages the use of small methods and classes However, this style of programming introduces much overhead as each method call results in a dynamic dispatch and each field access becomes a pointer dereference to the heap allocated object Many of the classes in these programs are included to provide structure rather than to act as reusable code, and can therefore be regarded as intermediate We have therefore developed an optimisation technique, called *declassification*, which will transform Java programs into equivalent programs from which these intermediate classes have been removed

The optimisation technique developed involves two phases, analysis and transformation The analysis involves the identification of intermediate classes for removal A suitable class is defined to be a class which is used exactly once within a program Such classes are identified by this analysis The subsequent transformation involves eliminating these intermediate classes from the program This involves inlining the fields and methods of each intermediate class within the enclosing class which uses it

In theory, declassification reduces the number of classes which are instantiated and used in a program during its execution This should reduce the overhead of object creation and maintenance as child objects are no longer created, and it should also reduce the number of field accesses and dynamic dispatches required by a program to execute An important feature of the declassification technique, as opposed to other similar techniques, is that it guarantees there will be no increase in code size An empirical study was conducted on a number of reasonable-sized Java programs and it was found that very few suitable classes were identified for inlining The results showed that the declassification technique had a small influence on the memory consumption and a negligible influence on the run-time performance of these programs It is therefore concluded that the declassification technique was not successful in optimizing the test programs but further extensions to this technique combined with an intrinsically object-oriented set of test programs could greatly improve its success

# Chapter 1 Introduction

Computer programming has undergone phenomenal growth in recent years Software applications are increasingly being written in object-oriented languages like Java [Gosling & Joy & Steele, 1996] and C++ [Ellis & Stroustrup, 1990], because they offer simple, uniform, abstract programming models This programming model provides the benefits of increased flexibility, maintainability, extendibility and reusability Object-oriented programs have the significant disadvantage of being difficult to analyse and reason about This makes the task of optimizing the software much more difficult A number of features associated with object-oriented languages contribute to this problem

## 1.1 Features of object-oriented languages

One of the principle features of object-oriented languages is the use of inheritance There are two types of inheritance, single and multiple-inheritance Single inheritance means each class can inherit from only a single class, multiple-inheritance means each class can inherit from one or more classes Programmers are encouraged to design and write software which is built from a hierarchy of classes Class libraries, for example, are created which contain a collection of base classes These classes can be extended by client applications for use in their applications [Johnson, 1992] Another example where a hierarchy of classes is required is building unspecialized data structures to be generic and reusable rather than building custom optimized structures A drawback of designing and writing software in this way is it increases the chances that it is built with a deeply layered inheritance structure This in turn increases the difficulty of analyzing the software

Another powerful feature of object-oriented software is polymorphism Polymorphism is the ability of an entity to become attached to objects of various possible types The key benefits of polymorphism are that it makes objects more independent of each other and it allows new objects to be added with minimal changes to existing objects A significant disadvantage of this is that dynamic dispatching is required to locate the relevant methods at run-time Dynamic dispatching is required because the type of a receiver is not known until run-time This makes the control flow of the program more difficult to follow and substantially increases the complexity of analyzing the program Data polymorphism is also facilitated and this too adds to the complexity Dynamic class loading is another feature which increases the difficulty of analysis

These features of object-oriented languages not only make it difficult to carry out precise and detailed analysis of the code but they also result in expensive overheads, which slow down the execution of the program Object-oriented programming encourages the use of many objects and methods when designing software [Calder & Grunwald & Zorn, 1994] In many programs, sophisticated structures are assembled using a large number of composite objects Creating a large number of objects results in many objects being created on the heap This has two significant disadvantages, time is required to create and maintain the heap objects, and time is necessary to access the fields and methods of each object Each access to an object's field becomes a pointer dereference to the heap allocated object This puts pressure on the memory subsystem which adversely affects the run-time performance of the program

The widespread use of polymorphism and the fact that many methods are encouraged when designing software results in a substantial decrease in the performance of object oriented software This is because it is necessary to bind these methods to their calling objects by dynamic dispatch at run-time This is a major source of overhead for two reasons, one is as a result of the direct cost of method lookup, and the second is the indirect cost incurred as a result of the loss of the opportunity to inline methods and carry out other optimization techniques The widespread use of objects and methods are not the only issues negatively affecting the run-time performance of object-oriented software Other features of object-oriented languages such as thread synchronization and exception handling also contribute to this problem

Consequently, the run-time performance of object-oriented languages like Java are behind the most popular non object-oriented languages today, even with just-in-time compilation technology There is an obvious need for more aggressive optimizing techniques The effectiveness of optimization is decreased by the difficulty in obtaining adequate and precise analysis information A wide range of optimization techniques have been researched and developed for both functional and object-oriented languages

## 1.2 The Java Language

Java is an object-oriented language and has become very popular for software development It has the following characteristics, statically typed, single inheritance, dynamic class loading and late-binding Statically typed means that every object in Java has a well-defined type that is known at compile-time Java, however, is fully run-time typed as well which means the run-time system keeps track of all objects and makes it possible to determine their types and relationships during execution It is therefore possible to use completely new kinds of dynamically loaded objects with a level of type safety Dynamic class loading means that new classes can be dynamically loaded at run-time Late-binding means that a message is dynamically bound to the receiving object at run-time These powerful properties of the language come with significant overheads to the memory subsystem and run-time performance as discussed earlier

The Java language is a portable language The source code is firstly compiled into bytecodes This byte code can then be executed on any computer architecture running a Java virtual machine This ability to compile a program once and run it on many different types of machine is very important Some optimization techniques analyse the Java source code and obtain sufficient information to carry out different types of optimizations Other techniques focus on analyzing and optimizing the Java bytecodes In section 1 1, we described some of the characteristics of object-oriented languages that increase the difficulty of optimizing programs The Java language has other features that add to this complexity

They are as follows

1 The exception mechanism in Java presents obstacles to compiler optimization When an exception occurs within a Java program, all instructions prior to the exception should have executed and all instructions after should not Optimization techniques often involve moving code within a program The exception mechanism will severely restrict the freedom of code movement The problem is exacerbated by the fact that most Java instructions can cause an exception A number of approaches have been put forward to mitigate this problem, but it is still an area of on-going research,

2 The Java source code is compiled into Java bytecode, which is subsequently run on a Java Virtual Machine (JVM) [Lindholm & Yellin 1996] There is a high

level of abstraction associated with JVM bytecode This is a salient problem for compilers as it hides many machine dependent optimization opportunities,

3 The Java language is designed to be safe This is ensured by the security features that are built into the language For example, each class is created with a certain set of access rights associated with its fields and methods These access rights give the programmer the ability to declare the visibility of the fields and methods as being one of the following, private, public, protected and default A class can be declared as being public, default, abstract and final The use of access rights and the other security features provided by the language ensures its safety This safety hinders optimization opportunities because any changes made to the program by the optimization technique should not change or weaken the security of the program

## 1.3 Garbage Collection Systems

It is estimated that CPU speeds have increased by 60% per year for the past two decades [Chilimbi & Hill & Larus, 1999A] The time required to access the main memory of the computer has only decreased by 10% per year The unfortunate consequence of this has been an ever-increasing gap between the performance of the CPU and memory subsystems It is therefore vital that the computer's heap and other memory resources are used efficiently to improve the run-time performance of a program Object-oriented programs create objects which are allocated within the heap, during the execution of a program The more objects that are created, the more heap space which must be designated to store them These programs have been criticized for the amount of heap space they require when executing and consequently, the issue of heap space is becoming a serious problem In these situations it is paramount that the objects in the heap space which are no longer needed are detected and removed '*Garbage collector*' is the term used to describe a system which will automatically search for, find and reclaim any heap allocated objects which are no longer required for the execution of a program These objects are referred to as '*garbage*'

Some object-oriented languages, such as Java and Smalltalk, do not have explicit memory management operations The programmer is prevented from including explicit memory deallocation statements within a program The programmer is therefore dependent on the garbage collection system to find and release any unused storage cells Object-oriented languages which have explicit memory management operations are dependent on the programmer to deallocate the garbage objects, when they are no longer needed Failure to do so could put unnecessary pressure on the memory subsystem These languages can also benefit from an automatic garbage collection system, which can be used to remove garbage objects from the heap space that the programmer has not explicitly deallocated This reclaimed heap space can then be used to accommodate any new objects which are created

The main aim of the run-time garbage collection system is to automatically search for, find and deallocate any *garbage* in the program's heap space during execution When an object is allocated on the heap, used, and then the reference to the object is no longer needed, the garbage collection system should detect this and deallocate the memory for the object This space will be used in the future when allocating new objects This process is performed not by the compiler, but by the runtime system Much research has and is taking place into developing and improving run-time garbage collection techniques Three of the most popular types of run-time garbage collection schemes are mark and sweep, reference counting and, copying These schemes increase the amount of heap space that is available to the running program Performing garbage collection on a program may introduce both performance overheads and extra implementation complexity There has been much research as a result into garbage collection algorithms spanning a wide range of precision, and it should be noted that determining when objects are no longer needed is a very difficult task

A second important benefit of a run-time garbage collection system is that it frees the programmer from the responsibility of explicitly releasing objects in his/her program when he/she no longer needs them Programmers do not have to waste their time even thinking about the issue when an automated tool has the full responsibility for finding and releasing garbage in a program He/she is free to concentrate on other important issues which need to be addressed when designing and writing application software Thirdly, it can sometimes be very difficult for a programmer to know when to write the statement to release the object(s) at the critical moment necessary Incorrectly placed assignment statements could result in unintended and invalid results during program execution

The ability of a garbage collection system to achieve absolute precision in the detection of garbage cells in a program is impossible and is equivalent to the halting problem

There are a number of techniques used by each of the garbage collection systems Each technique has varying degrees of sophistication and complexity, but the benefit of the technique must be judged on its ability to detect and reclaim garbage cells, increase heap memory and improve the speed performance of the program

It would be detrimental to the running of a program if an object which was being used, was identified as garbage by a garbage collector and destroyed This would result in the garbage collector arbitrarily affecting the correct execution of programs Therefore garbage collectors should use safe approximations as to what cells are no longer required by a program This is considered a conservative garbage collector If a conservative garbage collector is overly conservative then this could result in the retention of large amounts of garbage

## 1.4 Uniform verses Non-Uniform techniques

The mechanisms associated with software development using some object-oriented languages like Java promote a uniform approach, while the mechanisms associated with other object-oriented languages like C++ promote a non-uniform approach Automatic garbage collection is one of these mechanisms which promote a uniform approach to software development in Java Automatic garbage collection mechanisms can have a negative effect on the run-time performance of a program Other object-oriented languages like C++ have explicit memory management operations This means that they do not suffer from the overhead of implementing an automatic garbage collector There are, however, significant benefits associated with automatic garbage collection mechanisms as highlighted earlier and these are thought to out-weigh the disadvantage of the run-time overhead

Object-oriented languages such as C++ have a number of language features that improve the performance of the language One of these is the ability to declare an object or method to be 'inlined' Object inlining means a programmer can declare the fields of an object to be either objects or pointers to objects This feature gives the programmer the ability to explicitly inline objects within other objects In doing so you group related objects which can be allocated and deallocated together A consequence of this is a reduction in the number of memory dereferences necessary during program execution C++ also permits a programmer to explicitly declare that a method is to be inlined The method specified as inlinable can then be expanded *inline* by the compiler at each point in the program in which it is invoked In doing so, the run-time performance of the program should be improved because you have eliminated the number of dynamic dispatches Explicit inlining requires changes to be made to the code structure, which modifies the sharing semantics of a program This burdens the programmer with the responsibility of deciding which objects and methods should be inlined Automatic inlining frees the programmer from the responsibility of having to explicitly inline objects and allows him/her to program in a uniform object model

Another language feature that improves run-time performance is allowing the programmer to explicitly declare if a method is virtual or not A method that has been declared as being virtual, can be overridden by methods created in subclasses A non-virtual method cannot be overridden in this way The compiler can use this information to implement direct procedure calls There are disadvantages associated with this technique These include the fact that in some situations it may be difficult to decide if a method should be virtual or non-virtual Another reason is it restricts the extendibility and reusability of the classes when they are declared as non-virtual It also forces the programmer to write software in a non-uniform way

## 1.5 Optimizing Memory Usage

The functionality and complexity of software has increased over the past number of years and is continuing to grow This often results in a greater requirement for heap space by software programs Object-oriented programming encourages the building of programs which resemble the structure of the original problem to be solved The purported benefit of this type of programming is software which is easier to read and understand, which in turn should lead to software which is easier to debug and maintain However, this leads to greater inefficiency in memory usage and decreases the run-time performance of object-oriented software

Compile-time garbage collection and compile-time garbage avoidance techniques are optimization techniques which can reduce the run-time memory requirements of programs The compile-time garbage collection technique does not actually collect garbage cells during the compilation of a program The goal of compile-time garbage collection is to analyse a program during compilation to determine any cells that are no longer required after a particular point for the evaluation of the program This program is then annotated to identify the detected cells as garbage The garbage cells are subsequently collected automatically at run-time These cells will be available for further use during the execution of the program once they have been collected Therefore, compile-time garbage collection should be regarded as compile-time optimization of memory usage Because the garbage collection itself does not actually take place at compile-time, the term compile-time garbage collection is misleading However this term has been used for this kind of optimization in the past, so it is used again here This technique reduces the amount of heap space required by a program during execution

The compile-time garbage avoidance technique has a different approach to that of the compile-time garbage collection system Its aim is to analyse a program's source code and to detect and carry out changes to the source code These changes should transform the source code and reduce the amount of heap space it allocates and uses during its execution These optimization techniques have been successful in improving the run-time performance of programs Object inlining is an important compile-time garbage avoidance technique and it is used to optimize object-oriented programs It is estimated, for example, in [Dolby & Chien, 2000] that the object inlining technique improves the run-time performance of programs by approximately 14% Less research has taken place into compile-time garbage collection techniques for object-oriented languages A number of them are presented in section 2 2 and some have had encouraging results For example the technique in [Gay & Steensgaard, 1998] established that speed improvements of up to 11% were achievable

## 1.6 Other optimization techniques

There are many other optimization techniques developed besides the techniques which one would class as optimizing memory usage The following are a list of popular and widely used optimization techniques, which can be applied to different types of languages

1 Dead code elimination – This is the deletion of code within a program, which will never be executed An example of dead code elimination is finding a method in a class that is never called in the program This can be safely eliminated from the program,

2 Local common subexpression elimination – This is a prevalent and successful technique which is used to eliminate redundant re-compilations within a program Value numbering is the algorithm [Simpson, 1996] and [Briggs *et al* , 1996] used to achieve this optimization This algorithm involves numbering all variables in the program Two variables are given the same number only if they have the same value Two expressions will get the same number if they have identical structure and the variables used in each expression have the same value Once the numbering has been completed it is easy to identify common sub-expressions,

3 Register & stack allocation – Carefully allocating variables to storage locations within a program during program execution can enhance the performance of the program Local variables which can be stored on the stack, for example, eliminate the need for a store and subsequent load from memory and thereby improves the performance,

4 PeepHole optimizations – The performance of a program can be enhanced by replacing complex computations with simpler ones, that compute the same result,

5 Constant/copy propagation - Constant propagation involves analyzing a program to identify where constants are used It is then possible to substitute each constant with its value This aids the analysis of the program because more information is available on the variables which use these constants Copy propagation is similar to constant propagation It involves substituting a variable with a value instead of a constant

There are many optimization techniques which are specifically designed for object-oriented languages Their success at improving the run-time performance and/or heap usage has resulted in their use becoming prevalent Three of the more popular are presented in this thesis, improved memory usage, partially redundancy elimination (PRE) and elimination of dynamic dispatches These techniques are presented to illustrate some of the other methods which can be used to optimize object-oriented software

## 1.7 The Declassification Technique

In this thesis we present the declassification technique, a novel optimization technique designed for the Java language  The inception of the declassification technique was motivated by the success of the higher-order deforestation algorithm proposed in [Hamilton, 1996]  This optimization algorithm is a compile-time garbage avoidance technique  It has the ability to eliminate intermediate data structures from higher-order functional programs  By removing the intermediate data structures, the performance of the program should be improved and the heap space required by the program reduced  It was also influenced by the fact that in the C++ language, a programmer can declare an object or method as being inlined  There are a number of salient advantages associated with inlining  This motivated our research in finding a way in which to automatically inline classes

The central aim of the declassification technique is to reduce the number of *top-level* classes that are instantiated and used in a program at run-time  A top-level class is a Java class which is not an inner class  Inner classes were introduced in Java 1 1 1 and there are four types, static member classes, member classes, local classes and anonymous classes  Other references to inner classes may differ, for example in some books the term inner class refers to member classes, local classes and anonymous classes but not static member classes  Throughout this thesis top-level classes will be referred to as classes and a distinction will only be made between top-level and inner classes when extensions to the declassification technique are discussed in chapters 6 and 7

The declassification technique analyses a program to identify suitable classes for inlining  A suitable class is a class which is used *exactly once* within the program  A usage count is associated with each inlinable class and the number of its uses are calculated by analyzing how the class is instantiated and manipulated within the program  A field which is declared in the program of the inlinable class type is considered a use of the class  A field of the inlinable classes superclass type is also counted if it is used to store an instance of the inlinable class  Local variables, method parameters and method return types of the inlinable class type are counted  Similarly, if a local variable, method parameter or method return type is declared in such a way that it enabled an instance of the inlinable class to be stored there, the usage count of the inlinable class is incremented  Any anonymous object of the inlinable class type is counted Each superclass of the inlinable class is also counted as a use of the inlinable class The process of identifying a suitable inlinable class is explained in detail in section 4 3 1 Declassification means inlining each inlinable class which has a usage count of one within the class which uses it, which is referred to as the 'enclosing' class Any references to the instance of the inlined class are changed to reference its inlined fields and methods within the enclosing class object The declassification technique can then eliminate the inlined classes declaration from the program source code

Inlining as an optimization technique is a known algorithm The optimization technique presented in [Dolby, 1997], for example, inlines the fields and methods of an object within a container object This is discussed in detail in section 3 2 1 The important contributions made by the declassification technique are the presentation of a new analysis algorithm for deciding when inlining should take place and the fact that it inlines classes within container classes not objects within container objects The declassification technique involves a source code to source code transformation The optimized program code can then be compiled and run as normal Although the declassification technique has been designed and implemented to transform program source code, it is possible in theory to apply the analysis and transformation algorithms to bytecode directly We investigate the potential of the declassification technique by evaluating its effect on a number of reasonable sized programs

A number of benefits are associated with the declassification optimization technique These include

1 Eliminating the need to create 'intermediate' classes from a program This should reduce the pressure on the memory subsystem as fewer objects need to be created and maintained,
2 The fields of the inlinable class become local to the enclosing class This reduces the number of memory dereferences as these fields can now be accessed directly,
3 The methods of the inlinable class also become local to the enclosing class This eliminates the necessity of a subroutine call to access the classes methods This will in turn reduce the number of dynamically dispatched messages required by the program, which should improve the run-time performance of the program,

4 The inlinable classes methods are inlined within their enclosing class This exposes the body of each method to further optimization opportunities in the context of the original invocation,

5 An important benefit associated with this technique is that it guarantees that there will not be any increase in the code size of a program following its optimization This is an important issue for Java because one of the main reasons it became so popular was as a result of its suitability for writing software for the internet Any enlargement in code size would increase the time required to download Java programs from the Internet and increase the disk space required to store them Other optimization techniques such as [Dolby, 1997] cannot give such a guarantee

## 1.8 Summary

A lot of research is taking place into the optimization of object-oriented languages because of the popularity of object-oriented software development and the considerable overheads associated with the execution of object-oriented software The features which make object-oriented languages, specifically the Java language, difficult to analyse were discussed in section 1 1 The Java language is a popular language, it allows you to program in a uniform abstract way Its characteristics were explored in section 1 2 The Java language has inbuilt run-time garbage collection facilities The benefits and drawbacks of such a system were discussed in section 1 3

A number of different optimization techniques were introduced in section 1 5 including compile-time garbage collection and compile-time garbage avoidance techniques An overview of the declassification technique was presented in section 1 7 Its central aim is to identify and inline suitable classes It is a fully automatic optimization technique for the Java language It does not burden the programmer with the responsibility of explicitly stating which classes should be inlined

In chapter 2, we introduce compile-time garbage collection, explaining the three main techniques used, compile-time garbage marking, explicit deallocation and destructive allocation Chapter 3 presents a wide range of compile-time garbage avoidance techniques for both functional and object-oriented languages The benefits and problems associated with the garbage avoidance techniques are discussed The analysis and transformation algorithms used to carry out the declassification technique are presented in chapters 4 and 5 The declassification technique was used to optimize a number of reasonably sized programs and the results of this empirical study are presented in chapter 6 Finally, chapter 7 analyses the results from the empirical study Further possible extensions to the declassification technique are also discussed

# Chapter 2 Compile – Time Garbage Collection

In this chapter we give an overview of some compile-time garbage collection techniques The goal of *compile-time garbage collection* is the detection of garbage cells during the compilation of a program and annotating the program to allow these cells to be collected at run-time The program is searched and the relevant cells are identified This program is then annotated to highlight the detected cells as garbage and they can be subsequently reclaimed for use in allocating new objects This chapter is divided into three sections, section 2 1 discusses the compile-time garbage collection techniques which have been researched and developed for functional languages, section 2 2 discusses the techniques for object-oriented languages The benefits of compile-time garbage collection are discussed in section 2 3 We look at the techniques which are used to optimize functional languages because this research is more mature and lessons can be learned from these techniques which can and are applied to object-oriented languages

## 2.1 Functional languages

Three of the main techniques used to perform compile-time garbage collection on functional languages are

1 Compile-time garbage marking,
2 Explicit deallocation,
3 Destructive allocation

Programs are annotated for each of the three methods of compile-time garbage collection using information obtained by static analysis Static analysis is the analysis of programs to determine properties of programs without actually executing them Static analysis can involve collecting information about the definition and uses of cells in a program Relationships between cells can be traced to determine how they affect each other The control flow of the program can be analysed to try to determine the sequence of events that could take place during the execution of the program

The results of the static analysis are used by each technique to add the necessary annotations to a program Type inference is an example of a static analysis technique which is used to obtain information about the object types which are used in a program This information is obtained by the analysis of the program structure and physical data layouts, to distinguish between different data types which occur in a program Examples of type inference schemes which can be used are described in [Baker-Finch, 1992], [Wright & Baker-Finch, 1993] and [Smetsers *et al*, *1993*] Other static analysis techniques, which can be used are described in [Mycroft, 1981], [Hudak, 1987], [Jones & Le Metayer, 1989], [Hamilton & Jones, 1990], [Hamilton, 1992] and [Hamilton, 1998]

### 2.1.1 Compile-Time Garbage Marking

This technique involves marking those cells in a program which will become garbage after their first use These cells will subsequently be freed and made available for further allocations Static analysis is used to obtain information about cell usage in a program and the program is then annotated for compile-time garbage marking A usage counter could be associated with each cell to determine the number of times that cell has been used A cell, which has a usage count of no more than 1, will be tagged to indicate that it will become garbage after it has been used During the execution of the program, the run-time garbage collector will automatically collect these tagged/marked cells after their first use These cells will be added to a run-time free list and their memory space will be available for further allocations

[Hughes, 1991] describes how a strict higher-order functional language can be optimized by a compile-time garbage marking technique Static analysis is used to determine properties about a program During the compilation of a program it is annotated to indicate that at certain points during its execution the store-cells can be collected as garbage [Hughes, 1991] introduces two static analysis techniques to obtain information which will allow the programs to be optimized for compile-time garbage marking, generation analysis and inheritance analysis The *generation analysis* technique identifies the sharing information among values within expressions It can tell which values within a particular list are unshared when evaluation of the expression finishes *Inheritance analysis* is the second technique and it identifies which values of a list used within a function are required for the evaluation of the result of the function

The sharing information obtained by the two techniques is very important It is used to identify values which are no longer needed after a particular point in a program These values can therefore be marked and considered as garbage A reference semantics is provided to show that the store related optimizations of compile-time garbage marking are correct By 'correct' we mean that an unoptimized and an optimized program should compute exactly the same result, that is, they are equivalent The denotational semantics of the language is therefore augmented with denotational store semantics This serves as a reference for the correctness of the analysis and optimizations

In [Hamilton, 1995] it is shown how a program written in a first-order lazy functional language could be annotated for compile-time garbage marking The static analysis technique used to obtain properties about a program is called *usage counting analysis* and is responsible for detecting and counting the number of times each value is used in a program Usage counting values are then abstracted to usage patterns to allow usage counts to be determined at compile-time The aim of this analysis is to be able to determine at particular points in a program the maximum number of times a value will be used in the future evaluation of the program This information is invaluable in identifying cells within a program, which should be tagged to indicate that they will become garbage after their first use It is shown how a program optimized in this way can be proven correct A reference semantics is provided in order to show that the usage counting store related optimizations for compile-time garbage marking are correct

This technique is similar to that of [Hughes, 1991], as they both use static analysis information to tag cells in a program, to indicate that they can be collected as garbage at certain points during program execution

### 2.1.2 Explicit Deallocation

Using this technique, cells in a program are analysed to determine if they will always become garbage at a particular point in that program  The program is then annotated to indicate that the cell can always be deallocated at this particular point  This method ensures that cells are explicitly  deallocated immediately after becoming garbage  Explicit deallocation does not involve marking cells in a program and then checking each cell during its execution to see if it is marked, as is done for compile-time garbage marking  However, it does require that cells which are explicitly deallocated are added to a free list  Static analysis is again used to obtain information about cell usage in a program and it is used to annotate programs for explicit deallocation

In [Hamilton, 1995], it is shown how a program written in a first-order lazy functional language could be annotated for compile-time explicit deallocation  Usage counting analysis is again used to identify the particular points in a program where cells are no longer needed and can be explicity deallocted  This usage counting analysis is similar to the analysis used for compile-time garbage collection  This technique is also similar in the way the standard semantics of the language are augmented with store semantics  These are used as a reference against which the usage counting store related optimizations can be proved correct

An explicit deallocation technique is considered in [Mohnen, 1995], which analyses the data structures found in functions written in a first-order functional language  Each function is analysed to detect data structures in the function which will become garbage after a particular point in the program  This data structure can then be subsequently deallocated and made available for further allocations  This optimization technique focuses entirely on the data structures of the arguments to a function and it has the ability to handle arbitrary data structures  Its aim is to obtain enough information to be able to determine if the data structure becomes obsolete during the execution of the function

An abstract interpretation, which exploits the special structure of the underlying functional language was developed in [Mohnen, 1995]  This interpretation will make it possible to determine if the heap cells of the arguments are inherited in the function result  An argument is inherited if it is needed for the further evaluation of the program and is therefore not considered garbage  The inheritance information gathered by the abstract interpretation associates an abstract domain with the heap cells of the function arguments The abstract domain can be inferred directly from the types of the arguments This abstract domain is a finite partially ordered set, and the efficiency of the technique is improved because it does not contain all possible argument combinations It should be noted that if one cell of a data structure is inherited, all cells of that data structure are considered inherited The cells associated with the arguments of the function which are considered garbage on termination of that function, can be safely deallocated The correctness of the abstract interpretation is considered in this paper and it proves that a program, which has been modified by this technique is correct Using information gathered as a result of the abstract interpretation, commands can be inserted into the program to explicity deallocate the appropriate cells

The [Hughes, 1991] approach to static analysis has close similarities with [Mohnen, 1995] The [Mohnen, 1995] technique however, has the ability to handle arbitrary data structures or structures containing functional parameters Garbage is detected in [Hughes, 1991] if heap cells are not inherited, [Mohnen, 1995] does not have this restriction

There are two ways in which the deallocation of the cells can take place within a program

1 The cell could be deallocated immediately after it becomes garbage This will have the disadvantage of frequent interruptions to the actual computation,
2 The second approach delays collection of garbage cells until the end of the corresponding function call It has the advantage of efficiency as more deallocation can be performed at the same time However, it has the disadvantage of delaying the collection of garbage cells

### 2.1.3 Destructive Allocation

This technique involves reusing garbage cells directly within a program A deallocation function is not required since garbage cells are reused directly within a program rather than being added to a free list Static analysis is again used to obtain information about cell usage in a program and it is used to annotate programs for destructive allocation The analysis must determine that a cell will no longer be needed for the evaluation of a program and could be explicitly deallocated This cell can then be destructively allocated at some later point in the program These cells are not deallocated by the run-time garbage collector and added to a free list for further allocation The cells are reallocated directly within the program

[Hughes, 1991] describes how a strict higher-order functional language can be optimized by a compile-time destructive allocation technique Static analysis is again used to obtain properties about a program without actually executing it It is shown how programs are annotated to allow a deallocation followed by an allocation to be coalesced to give a destructive allocation instead Generation analysis which has been described earlier in compile-time garbage marking, and a destruction function analysis are the two techniques used to obtain the necessary information

The *destruction function* analysis technique in [Hughes, 1991] investigates the arguments to a function, to identify any of these arguments or part of the arguments, which could be reused within the function body A free cell does not have to be allocated if we can reuse an existing argument value This is essential information for the destructive allocation technique Similarly, as in the case of compile-time garbage marking, [Hughes, 1991] provides a reference semantics to show that the store related optimization of compile-time destructive allocation are correct

In [Hamilton, 1995], it is shown how a program written in a first-order lazy functional language could be annotated for compile-time destructive allocation Usage counting analysis is again used to obtain information about cell usage in a program and it is used to annotate programs for destructive allocation It is shown how a program optimized in this way can be proven correct Similarly, as in the cases of compile-time garbage marking and explicit allocation, the standard semantics of the language is augmented with store semantics This is used as a reference against which the usage counting store related optimizations can be proved correct

[Mohnen, 1995] considers using the technique previously presented for explicit allocation to implement a destructive allocation compile-time garbage collection technique Consider the scenario where a deallocation is followed immediately by an allocation In this situation a deallocation is not required since the garbage cells can be reused directly within the program, rather than being added to a free list However, [Mohnen, 1995] does not implement a destructive allocation technique because of a major disadvantage associated with it The complexity of the optimization technique would have to be increased in order to handle destructive updates  It is estimated that the gains associated with destructive allocation are far out-weighed by the increased complexity

## 2.1.4 Comparison of the different functional techniques

### 2.1.4.1 Compile-time garbage marking

Compile-time garbage marking requires extra storage per cell to indicate whether or not the cell is marked  The extra storage required may be more than the storage which is saved by using this technique  A second problem associated with this technique is that extra time is required during the execution of the program to check each cell to see if it is marked  This overhead in execution speed could potentially swamp out the benefits  Another disadvantage is the necessity of maintaining a run-time free list, to which the marked cells are added  This restricts the run-time garbage collector to a scheme which uses a free list  Because of the overall disadvantages associated with compile-time garbage marking, it is unlikely to be suitable for practical use

### 2.1.4.2 Explict Deallocation

This technique does not require storage space to be allocated to associate a usage counter with each cell in a program  Additional memory space is therefore not required by this technique  Expensive time is not wasted during the execution of the program checking each cell to see if it is marked, which must be done to implement the compile-time garbage marking technique

Explict deallocation requires that cells, which are explicitly deallocated, are added to a free list  The run-time garbage collection system used must therefore use a free list when it is allocating storage  This restricts the type of run-time garbage collection system which can be chosen, to a scheme like mark and sweep  Run-time garbage collectors which do not use a free list are considered to be more efficient garbage collectors  As a result, the majority of garbage collectors which are currently used for functional languages are of the copying type  The techniques of compile-time garbage marking and explict deallocation are therefore of limited use

There is also a problem of conflicts between this technique and the run-time garbage collector A cell, which has been explicitly deallocated might still be considered to be live by the run-time garbage collection system There are instances where compile-time garbage marking can be used to collect a garbage cell within a program and explicit deallocation is unable to The reverse of this situation is also true, there are situations where explicit deallocation can be used and compile-time garbage marking cannot

### 2 1 4.3 Destructive Allocation

This technique has the benefit of not requiring a free list since garbage cells are reused directly within a program We are therefore not restricted in the type of run-time garbage collection scheme used

Destructive allocation does not have the same overheads associated with compile-time garbage marking These overheads are the extra memory space necessary to store the usage counter and the overhead on execution time caused by checking each cell to see if it is marked However, there are instances where compile-time garbage marking can be used to collect a garbage cell or the cell can be explicitly deallocated within a program, but the destructive allocation technique cannot be performed This technique is therefore less applicable than the other techniques The complexity of the algorithm to implement a destructive allocation technique has been identified as a significant disadvantage It is estimated that the gains associated with this technique are over-shadowed by the increased complexity

## 2.2 Object-Oriented Languages

Not a lot of research has been done in the area of compile-time garbage collection techniques for object-oriented languages, research has instead concentrated on compile-time garbage avoidance and a myriad of other optimization techniques However, a number of algorithms have been devised for compile-time garbage marking and explicit deallocation techniques and are presented in this section Some of the techniques have identified methods for compacting memory or allocating objects on the stack

### 2.2.1 Compile-Time Garbage Marking

[Diwan *et al*, 1992] explores how information gathered during the compilation of a statically typed language can be used to support the compaction of memory by a run-time garbage collector [Diwan *et al*, 1992] describes a technique which is used to build stack maps for the Modula-3 language Modula-3 is an object-oriented statically typed language and a salient feature of this language is the fact that a stack map is not generated for every instruction, but is restricted to the garbage collection points A garbage collection point is a place in the program where a collection might occur The stack map data structure which is used in this technique, is an example of a tagless system and it has been used by other compile-time garbage collection techniques

It is imperative that the garbage collector obtains the necessary information on the different data types contained within a program It is stated that tracking the location of pointers in global variables is trivial Locating variables on the stack and in registers is more difficult and it is emphasized that there is particular difficulty associated with derived pointers A derived pointer's value is created as a result of pointer arithmetic and there are difficulties associated with accurately identifying the base values for each derived value, at particular points in the program There are three different tables constructed to store the necessary information on stack, register and derived pointers A study was conducted in [Diwan *et al*, 1992] and it was estimated that the table sizes when compressed reduced to 16% of the optimized code size It is possible to store and extract information from the tables quickly, which is important It was also established as a result of this study that the technique had no effect on the optimized code produced

[Agesen *et al* , 1998] considers two techniques which are used to increase the accuracy of the root set, type precision and liveness analysis A root set is the set of local and global variables in a program Some of these variables could be used to reference objects that are local or global The aim of these techniques is to reduce the root set, that is to establish what cells in memory are no longer being referenced and can be released To achieve absolute precision of the root set is not possible because there could be pointers to cells within memory, where it is unable to determine whether the cell is garbage or not This is because it does not have enough definite information about the cell and in this situation it should consider the cell to be live

The first algorithm in [Agesen *et al* , 1998] considers the type of all local area variables in a program This information is gathered and stored during the compilation of the program in order to increase the precision of the part of the root set resulting from local variables This technique was developed using the Java programming language where local variables are stored in slots in stack frames It could be adapted for use with other object-oriented languages This technique generates a stack map data structure to store the relevant information The generation of the stack map would be relatively trivial if it was not for the fact that the JVM allows one exception to the Gosling property [Lindholm & Yellin 1996] The Gosling property states that the stack and registers must always *look the same* whenever a JVM instruction is executed For example, the type of each stack element and local variable at any point in a program should not depend on the path taken to reach that static program point The exception to the Gosling property is the JSR subroutine It makes it very difficult to carry out type analysis on a program, as it may be unable to determine the exact type of a particular variable in a JSR subroutine A solution was found by adding additional information to the stack map and splitting conflicting variables

The second technique in [Agesen *et al* , 1998] uses intra-procedural live variable analysis to identify the local variables within the stack, which are reachable from the root set We consider a variable to be live if it holds a value that may be needed in the future execution of the program Liveness analysis is not a new research area and has been used in the past to reduce the root set by identifying dead references, which are garbage cells Live variable analysis information is added to the stack map and it is estimated that the cost of generating a live-precise stack map is approximately 50% greater than the cost of generating a type-precise stack map

## 2.2.2 Explicit Deallocation

[Gay & Steensgaard, 1998] describes a technique which allocates objects on the stack rather than the heap The algorithm will try to calculate the lifetimes of each object created in a program, and any object with a known limited lifetime can be created on the stack The memory associated with these objects is automatically reclaimed when the stack frames are reclaimed as functions return There are two important benefits associated with allocating objects on the stack instead of the heap, an increase in the amount of heap memory available during the execution of the program and a reduction in the memory management activities, which should subsequently lead to an increase in the execution speed of the program

Two analysis techniques are introduced which are used to obtain the necessary information to allow it to identify the stackable objects, escape analysis and loop analysis The *escape analysis* technique conservatively estimates the references to the objects in a program An object cannot be stack allocated in a method's stack frame if a reference to the object escapes from that stack frame The analysis considers a reference to an object to have escaped if the object is returned from the method or if another object stores a reference to this object *Loop analysis* is the second technique and it is primarily concerned with the control flow of a program This is necessary to evaluate whether assignments are made to variables in a loop dependent manner

Information is gathered on fresh methods and variables A *fresh method* is a method which will return an *unaliased* object of a certain type An object is considered unaliased when there are no other references to this object A fresh variable is one which is assigned a new object or an object from a fresh method The stackable variables are computed using this information and the results of the escape analysis and the loop analysis An empirical study in [Gay & Steensgaard, 1998] estimated that approximately 10-20% of objects created in a program could be allocated on the stack instead of in heap memory

[McDowell, 1998] presents a hypothesis that a compiler can identify a significant number of object allocations that can be changed from being allocated on the heap to being allocated on the stack This study was carried out on the Java language

A Java virtual machine was instrumented to count the number of potentially stackable objects at runtime, in a small collection of Java programs JDK1 0 2 from JavaSoft was the virtual machine used This version of the Java virtual machine does not have a Java Native Interface (JNI) specification and therefore cannot accurately track references which are passed into native methods A native method is a method which is callable from a Java program and is written in a different language The results of the analysis can be interpreted in two ways The first is a conservative analysis and it assumes that all references passed to native methods cannot be considered as a stackable object The second is the non-conservative analysis and it assumes that all references passed to native methods are stackable

The conservative analysis estimated that approximately 10% of the heap allocated objects could be allocated on the stack during the execution of the program The results of the non-conservative analysis was significantly better, one of the programs tested resulted in 56% of its objects being suitable for allocation on the stack It is proposed that further research is necessary to implement a Java virtual machine which will support the stack allocation of objects

No object-oriented destructive allocation techniques were found in the literature

## 2.2.3 Comparison of the different object-oriented techniques

### 2.2.3.1 Compile-time garbage marking

The techniques presented in [Diwan *et al*, 1992] and [Agesen *et al*, 1998] are both tagless systems An alternative to this approach would be to use tags, this would involve tagging all objects allocated in the stack The overhead of this technique is expensive because stack frames are created and destroyed regularly during the compilation of a program Diwan's technique [Diwan *et al*, 1992] is used to obtain information on a wider range of cell storage than Agesen's technique [Agesen *et al* ], which only concentrates on obtaining information about variables stored in the stack frame [Diwan *et al*, 1992] and [Agesen *et al*, 1998] do not generate stack maps for every instruction in the program They are instead generated at particular points within the program

An empirical study was carried out in [Agesen *et al*, 1998] and it was found that there was on average an 11% reduction in the amount of memory space required by the local variable root set as a result of the liveness analysis technique This technique therefore appears to offer minimal benefits over type-precise analysis However, it should be noted that it is important in reducing the possibility of surprisingly large volumes of garbage being retained [Diwan *et al*, 1992] did not carry out an empirical study, however, they state that because the overhead of implementing the stack map is small, the application of this technique should be of practical use

The complexity of the technique in [Diwan *et al*, 1992] is increased as a result of the problem in clearly identifying type information about derived pointers Two solutions are explored, path variable scheme and path splitting Both solutions have overhead associated with them Path splitting increases the code size of the program and the path variable scheme increases the size of the derived tables and there is also the problem with indirect references in machines with complicated addressing modes The path variable scheme is the method used by [Diwan *et al*, 1992] because it is considered to be simpler and more straightforward It is stated that the problem with indirect references should not occur for a load/store architecture

A problem could arise in the type precision technique [Agesen *et al*, 1998] as a result of rewriting instructions because an instruction's position and length could change

This could result in a method exceeding the maximum method size To resolve conflicts occurring in the type precision technique, variables are split This will result in an increase in the number of variables created within a program This could exceed the limit on the number of local variables allowed within a method which is imposed by the byte code instruction set The above two problems are in practice extremely unlikely to occur

[Diwan *et al*, 1992] and [Agesen *et al*, 1998] are compile-time garbage marking techniques and they suffer from similar drawbacks to the ones highlighted in reference to functional compile-time garbage marking techniques

### 2.2.3 2 Explicit Deallocation

The empirical study [Gay & Steensgaard, 1998] established that speed improvements of up to 11% for medium sized programs were achievable However, there are a number of salient speed variations where a negative performance results This cannot be attributed to stack allocation, as some of the programs where there is a considerable negative performance carried out very little stack allocation It is therefore unclear as to why these speed variations occur between the benchmark programs

The analysis algorithm in [Gay & Steensgaard, 1998] can be accomplished in linear time and is considered to be simple and fast However, the escape analysis has been criticized for being too conservative The performance of the stack allocating technique could be enhanced if the quality of the escape analysis was improved For example, the analysis does not track objects which are thrown by exceptions, it considers them to be live and necessary for the execution of the program

The work in [McDowell, 1998] does not implement a compile-time explicit deallocation technique, but carries out a theoretical study to quantify the benefits of allocating objects on the stack [McDowell, 1998]'s conservative analysis estimates that approximately 10% of objects could be allocated on the stack instead of the heap It is noted, however, that as the overhead of garbage collection increases, this technique could become more attractive It is impractical to make a comparison between the results of stack allocating objects in [Gay & Steensgaard, 1998] because McDowell's study did not consider the possibility of stack allocating objects returned by 'fresh methods'

The explicit deallocation technique presented in [Gay & Steensgaard, 1998] suffers from similar disadvantages, as described in section 2 1 4 2 For example, the type of run-time garbage collector is restricted because the reclaimed objects must be added to a run-time free list

## 2.3 Benefits of Compile-Time Garbage Collection techniques

The main benefit of these techniques is the increase in the amount of available memory during the execution of a program as a result of the garbage cells that have been detected and deallocated An empirical study was carried out on a number of programs in [Mohnen, 1995] which shows clearly that compile-time explicit deallocation is worth the effort

One of the important benefits is that programs can be written which may be far from optimal in their use of memory, but are much easier to read and understand These programs can then be analysed to search for and detect any garbage cells by one of the compile-time garbage collection techniques

In some object-oriented languages, it is the responsibility of the programmer to know when to write the assignment statements to release the object(s) Incorrectly placed deallocation statements could result in unexpected and invalid results A compile-time garbage collection system will automatically find and deallocate garbage cells

There should be a reduction in the amount of garbage which will be found by the run-time garbage collection system This should result in speed improvements as time is not wasted searching for and detecting the unshared memory cells during the execution of a program However, this is not always the case as the overhead involved in carrying out one of the compile-time garbage collection techniques may outweigh any increase in speed due to reducing the overhead of run-time garbage collections This is particularly pertinent in the case of compile-time garbage marking as it is shown to slow down the execution speed of a program

Some program bugs can be highlighted as a result of different program optimization techniques The liveness analysis in [Agesen *et al*, 1998] is therefore not unique in having the ability to expose or hide bugs within a program These bugs are exposed when it identifies and omits dead variable references within a program It is therefore imperative that an optimization technique does not interfere with the results of a program execution, because the outcome of a program must not be dependent on the optimization techniques used

Improving the locality of objects stored in the heap and making the allocation of new objects faster are important benefits The stack allocation of objects presented in [Gay & Steensgaard, 1998] is an example of this This is because objects that have short lifetimes are allocated on the stack and are destroyed automatically when the stack frames are reclaimed Objects that have longer lifetimes can be allocated in heap memory and this avoids the fragmentation of heap memory by short lived objects [Diwan *et al*, 1992] and [Agesen *et al*, 1998] also present techniques that lead to improved memory locality

## 2.4 Summary

In this chapter, we have introduced the area of compile-time garbage collection, explaining the three most popular types of garbage collection schemes, compile-time garbage marking, explicit deallocation and destructive allocation We have presented a number of different algorithms in each of the garbage collection schemes for the functional and object-oriented languages In chapter 3, we introduce compile-time garbage avoidance techniques for both functional and object-oriented languages We discuss how this approach differs from those which we have seen in this chapter

# Chapter 3 Compile-Time Garbage Avoidance

In this chapter we consider compile-time garbage avoidance techniques *Compile-time garbage avoidance* has a different approach to the resolution of the 'garbage problem' in programs Its main aim is to reduce the amount of garbage created during the execution of the program In order to achieve this, the source code of the program is transformed in some way and a number of different techniques are used to accomplish this The transformed program should be more efficient in terms of memory space used because the amount of heap space required by the program to execute has been reduced The amount of time spent allocating and deallocating these intermediate data structures has also been reduced The reduction in the amount of heap space used by a program after it has been transformed depends on the particular program and the type of compile-time garbage avoidance technique used

It should also be noted that research has revealed that compile-time garbage collection and compile-time garbage avoidance can be complementary [Hamilton, 1995] A program which has been transformed using a compile-time garbage avoidance technique, could be further optimized to reduce the heap space requirements, by applying a compile-time garbage collection technique Much research has taken place into developing and improving compile-time garbage avoidance techniques This chapter is divided into three main sections, section 3 1 discusses the compile-time garbage avoidance techniques which have been researched and developed for functional languages, section 3 2 discusses the compile-time garbage avoidance techniques for object-oriented languages Section 3 3 presents other techniques that are used to optimize object oriented languages

## 3.1 Functional Languages

Much research has taken place into developing compile-time garbage avoidance techniques for functional languages and a number of algorithms have been developed One of the reasons for the interest in functional languages is because of their suitability for transformation [Burstall & Darlington, 1977] researched and developed an unfold/fold transformation system which is the basis for a substantial number of optimizing techniques It consists of a number of rules which are applied to a set of equations and can be used to transform almost any first-order or higher-order program

### 3.1.1 First-order Languages

#### 3 1.1 1 Listlessness Algorithms

The technique in [Wadler, 1984] removes intermediate lists from programs that have been written in a functional language and use lazy evaluation A program compiled with lazy evaluation will not evaluate any expression unless its value is required by some other part of the computation Intermediate lists were identified as being a cause of inefficiency during program execution A program that produces intermediate lists will require more memory space to store these lists There could also be an increase in execution time as these lists must be allocated, traversed and finally deallocated when no longer required

The technique presented in [Wadler, 1984] can automatically transform a program to improve its efficiency by removing all intermediate lists It can only be applied to programs which can be lazily evaluated in a bounded space, excluding the space occupied by the input and output of the program The domain of programs to which this technique can be applied, is therefore very limited It cannot be used, for example, to transform a program which processes tree structures using a stack of pointers as this requires unbounded internal storage The technique presented in [Wadler, 1984] uses a listless transformer which is partly responsible for calculating the storage requirements of the program and thereby estimating its suitability for transformation This is important because applying this technique to a program which cannot be evaluated in a constant bounded space could result in an infinite loop

Further research was carried out in [Wadler, 1985], in the area of listless programming It was evident that more research was required to enhance the transformation developed, to widen the domain of programs to which the technique could be applied There are many programs which cannot be transformed by the technique described in [Wadler, 1984] An example of a program which cannot be made listless is a program which requires two traversals of the input data Other programs which cannot be transformed include programs with tree data structures

In general there are few complete programs which are listless, that is the whole program can be automatically transformed to eliminate all intermediate lists There may be sections of the program which can be made listless, but other sections which cannot The technique presented in [Wadler, 1985] has the capability to transform parts of a program The complexity of the overall task may be reduced as a result of breaking the complex problem into a number of smaller manageable programs which are easier to construct and reuse All the sub-programs which make up the overall program may not be suitable to the technique described in [Wadler, 1984], therefore the program as a whole cannot be transformed The technique described in [Wadler, 1985] supports modularity, as it allows you to transform a sub-program to remove intermediate lists and combine this program with the other parts of the program which are not listless The types of program which can be optimized by [Wadler, 1985]'s algorithm is therefore widened

### 3 1.1 2 Deforestation Techniques

[Wadler, 1990] describes a transformation technique that is used to eliminate the intermediate trees produced in a program that has been written in a first-order lazy functional language This algorithm is referred to as *deforestation* because it detects and eliminates any intermediate tree structures which are produced in the program The program is transformed by this deforestation technique and is based on the unfold/fold strategy of [Burstall & Darlington, 1977]

The deforestation algorithm in [Wadler, 1990] can only be applied to a term which has a function definition in a given syntactic form This algorithm is presented in three steps The first is the 'pure' treeless form A term is treeless with respect to the function definition if it is linear and every argument of a function and every selector of a case term is a variable A term is considered linear if a variable does not appear in the term more than once The restriction that every term is linear guarantees that certain program transformations do not result in a more inefficient program

The input to the deforestation algorithm is therefore a linear term in which the function definitions are treeless The output will be a treeless term, which has eliminated all intermediate tree data structures This 'pure' treeless form is very restrictive for most practical uses and as a result it was extended The first extension allows the use of 'blazing' which is the marking of data structures of a certain type to indicate where intermediate values can remain The terms are blazed either with a $\oplus$ or $\ominus$ mark, which are assigned solely on the basis of type The terms which are blazed with a $\oplus$ will be transformed by the deforestation algorithm to eliminate intermediate data structures while the terms which are blazed with a $\ominus$ can be extracted and transformed independently The programs which we can input to this blazed deforestation algorithm are therefore less restrictive The second extension to the deforestation algorithm is to allow some higher-order functions, by treating them as macros The deforestation algorithm and its two extensions are proven to terminate in [Wadler, 1990]

[Chin, 1990] and [Chin, 1991] present a generalised deforestation algorithm which can be used to eliminate intermediate data structures from all programs which have been written in a first-order functional language Blazing is used in this technique and is similar to the blazed deforestation algorithm [Wadler, 1990] A function's arguments must satisfy an extended treeless form (e-treeless form) Any part of the function arguments which are not in this form, are blazed $\ominus$ and extracted

An enhancement in [Chin, 1990] to the deforestation algorithm was researched and developed This enhancement will allow programs from the complete higher-order language to be transformed, the technique is called 'higher-order removal' This algorithm can be proved to terminate if a well-typed higher-order program is input for transformation It has the ability to eliminate most higher-order expressions from the program, which facilities the removal of intermediate data structures A proof of the termination of this deforestation algorithm is presented in [Chin, 1990], which proves that the algorithm will terminate if only e-treeless functions are used in the expressions to be transformed

[Hamilton & Jones, 1991A] & [Hamilton & Jones, 1991B] also extend the deforestation technique presented in [Wadler, 1990] This extended deforestation algorithm can be applied to all programs which have been written in a first-order functional language This algorithm will detect and remove intermediate structures from a program and it is achieved by performing static analysis to determine which data structures can be eliminated

Two static analysis techniques, inheritance analysis and creation analysis are performed to obtain properties about the program without actually executing it *Inheritance* *analysis* is a backward analysis method It is used to determine if values of a structure which are used by a term, are definitely needed by the result of the overall term of which it is a part This structure is considered inherited if it is needed by the result and is therefore not an intermediate structure The second static analysis method is *creation analysis* This is a forward analysis technique, that is used to determine if a term constructs a structure Inheritance analysis is then used to establish whether this structure is inherited to the result, and if not it is called a constructed intermediate structure

The extended deforestation algorithm performs these static analysis techniques on all terms to be transformed Any constructed intermediate structures which are identified are blazed ⊕ and can be eliminated by the algorithm Any inherited intermediate structures are blazed ⊖ and are extracted in a similar manner to the blazed deforestation algorithm described by [Wadler, 1990] The algorithm shows that there is a finite number of terms encountered during transformation This bound is pertinent in proving that the algorithm will terminate

Further research was carried out by Chin in [Chin, 1992], where he extends the deforestation algorithm presented in [Chin, 1990] and [Chin, 1991] [Chin, 1992] presents a 'safe' deforestation algorithm which has the ability to transform a complete program and it outlines the necessary criteria to conservatively determine when this is possible The most salient feature of this algorithm is its ability to prevent the transformation of a program entering an infinite loop This was a serious problem with the previous generalised algorithm which could under certain circumstances enter an infinite loop during transformation The transformed program will be at least as efficient as the original program and should be more efficient when its intermediate data structures are removed

This algorithm eliminates intermediate data structures from all first-order functional languages and from most higher-order programs These transformations are carried out safely by annotating any unsafe expressions and extracting them before transformation takes place The result of the algorithm are expressions which have a form which is known as extended-treeless or e-treeless This generalised deforestation algorithm can therefore be proven to terminate

### 3.1.2 Higher-Order Languages

The higher-order deforestation algorithm can be used to transform both higher-order languages and first-order languages which are a subset of it

#### 3.1.2.1 Deforestation algorithms

A higher-order deforestation algorithm is presented in [Marlow & Wadler, 1992] This algorithm will remove intermediate data structures from programs which are written in a higher-order language An integral part of this algorithm is a set of transformation rules which convert a given expression into a higher-order treeless expression The transformation consists of three mutually recursive functions There are short-comings associated with this technique however, a higher-order treeless form is not defined for the algorithm and a proof of termination is not given

The deforestation algorithm provides the use of let-expressions, which can be used to indicate in the source language where intermediate structures should be allowed to appear It is expounded that there is increased flexibility and convenience associated with using let-expressions instead of the use of blazing Any intermediate structures as a result, which cannot be removed by deforestation can be present in the input program by the explicit use of a let-expression It is therefore relatively easy to generate treeless expressions from a non-treeless expression by adding appropriate let-expressions For example a let-expression can be used to protect any non-linear arguments in a program However, there is no mechanism which will accurately determinate where let-expressions should be placed within a program, to guarantee that a more efficient program will result after transformation

[Hamilton, 1995B] and [Hamilton, 1996] also present a higher-order deforestation algorithm A higher-order treeless form of expression is defined, similar to the blazed treeless form defined in [Wadler, 1990] There are two conditions in which an expression can be blazed Firstly, function arguments and case selectors which are not variables are blazed $\ominus$ Secondly, all non-linear variables within higher-order treeless expressions must be blazed $\ominus$ at their binding occurrence This means that these intermediate structures will not be removed by the deforestation algorithm and must be transformed separately All other intermediate structures within the program will be removed

A set of transformation rules are used to convert a given expression into a higher-order treeless expression This higher-order treeless form of expression is an easily recognized form of expression and any function definition can easily be generalised so that it is in this form The two major contributions made in [Hamilton, 1995B] and [Hamilton, 1996] are, a higher-order treeless form is defined for the algorithm and a proof of termination is given

Marlow presents a deforestation algorithm in [Marlow, 1996] which is fundamentally based on previous research done in [Marlow & Wadler, 1992] He was encouraged to revert his attention to his previous work following the publication of Hamilton's deforestation algorithm [Hamilton, 1995B] Marlow's algorithm like other deforestation algorithms is based on the [Burstall & Darlington, 1977] unfold/fold transformation system This thesis shows how deforestation can be performed for arbitrary higher-order functional programs and it specifies the conditions that must be satisfied

A higher-order treeless form is defined in [Marlow, 1996] for the deforestation algorithm and he also provides a proof of termination The cut-elimination principles of logic have been merged with the simple first-order deforestation algorithm to obtain a new higher-order deforestation algorithm Any sub-terms which are considered dangerous within a program, are transformed into let-expressions This algorithm is similar to the deforestation algorithm in [Hamilton, 1995B] and [Hamilton, 1996]

A higher-order deforestation technique is presented in [Seidl & Sørensen, 1997] The major contribution of this paper is it ensures the termination of the higher-order deforestation algorithm for a wider class of programs A control-flow analysis is performed on the program, and a set of integer constraints are collected This analysis is necessary in order to gather the vital information needed to detect dangerous sub-terms within the program A sub-term is considered dangerous if there is a risk that the program will enter an infinite loop and as a result not terminate, when carrying out the deforestation technique

The pnnciples behind this technique are well-known and a detailed description of the termination analysis technique used in the deforestation algorithm is given in [Seidl, 1996] The unfolding of function calls is carried out, except in the case where it will affect the termination of the deforestation algorithm In this situation, folding is introduced to avoid the repeated unfolding of the same expression The analysis is powerful enough to be able to determine where unfolding is not safe within a program

[Seidl, 1996] was unable to resolve a problem with constructor functions and, as a result, there was a necessity to restrict higher-order programs A program which has a constructor function with a functional argument could result in the transformation of the program looping indefinitely A solution was identified but requires that constructor functions cannot have functional arguments A mitigating argument is provided which states that using constructor functions with functional arguments may not be popular when writing programs It also states that it is only in some circumstances that the deforestation algorithm will enter an infinite loop when processing a program with such constructor functions

### 3.1.3 Comparison of functional language techniques

The definition of treeless form is simpler and more straightforward in the technique presented in [Wadler, 1990] than the definition of listless form presented in [Wadler, 1984] and [Wadler, 1985] The range of programs to which [Wadler, 1990] can be applied are more general, as it will transform programs that use intermediate trees and they do not have to evaluate in a constant bounded space but may use the space bounded by the depth of the tree

There are two extensions to the original deforestation algorithm in [Wadler, 1990], the blazed treeless form and higher-order macro technique The blazed treeless form extends the type of program which can be optimized The higher-order macro technique can only be applied to higher-order functions which have first-order recursion This will restrict the usability of these higher-order functions It is also more difficult for a programmer to see where intermediate structures will be eliminated, due to the loss of transparency when macros are used The deforestation technique in [Wadler, 1990] is considered to be restrictive and limited as a result

The algorithms presented in [Chin, 1990], [Chin, 1991] and [Hamilton & Jones, 1991A] are not as restrictive as the algorithm presented in [Wadler, 1990] as they can eliminate intermediate data structures from *all* programs which are written in a first-order functional language Programs which can be transformed by [Hamilton & Jones, 1991A] and [Hamilton & Jones, 1991B] algorithm must be linear Any non-linear function arguments must be blazed $\ominus$ and extracted before the program transformation takes place The linear property is essential to ensure that there is no loss of efficiency in program execution after deforestation This is more restrictive than other non-linear algorithms such as [Chin, 1990], [Chin, 1991] and [Marlow & Wadler, 1992] These algorithms, however, are unable to guarantee the improved efficiency of a program after transformation This is a significant drawback and could result in a slower program following transformation as duplication of expressions could occur

The technique in [Hamilton & Jones, 1991A] could be considered to be more intuitive and straightforward than the generalised deforestation algorithm presented in [Chin, 1990] and [Chin, 1991] This is because it is not necessary to transform function definitions into a treeless form before carrying out the transformation and special consideration does not have to be given to recursive functions [Chin, 1992] presents a deforestation algorithm which is proven to be safe This was not done previously An important benefit of this algorithm is its ability to eliminate more intermediate structures than [Chin, 1990] and [Chin, 1991]

The 'higher-order removal' extension in [Chin, 1990] and [Chin, 1991] requires a separate process to remove the higher-order features from a program before deforestation can begin The techniques in [Hamilton, 1996] and [Marlow, 1996] do not This 'higher-order removal' extension could result in redundant intermediate data structures remaining in the program and increases the complexity of the deforestation algorithm Chin's algorithm also has the major drawback that it cannot be applied to all higher-order expressions The techniques in [Hamilton, 1996] and [Marlow, 1996], in comparison, can be applied to all programs and are considered a more efficient algorithm for higher-order program deforestation Chin's algorithm may result in the transformed program having a considerable increase in code size

[Marlow, 1996]'s contributions on higher-order deforestation are very similar to the notion of higher-order treelessness and the termination proof made by [Hamilton, 1995B] and [Hamilton, 1996] The algorithm in [Hamilton, 1995B] and [Hamilton, 1996] is considered to be a less complicated and a more intuitive process than the other techniques used to eliminate intermediate structures such as [Marlow & Wadler, 1992], [Marlow, 1996] and [Seidl & Sørensen, 1997]

[Hamilton, 1996] and [Seidl & Sørensen, 1997] present proofs which will guarantee the termination of the deforestation techniques Transparency is a property associated with the algorithms presented in [Hamilton, 1996] and [Marlow, 1996] This is vital in achieving predictable optimization results and facilitating the changing of a source program for further enhancements The importance of transparency is that it is easy for a programmer to see where in the source program intermediate data structures will be eliminated

The deforestation algorithm in [Seidl & Sørensen, 1997] is considered to be very complex and it may be difficult for a programmer to identify where intermediate data structures will be eliminated The analysis which must be accomplished to implement this technique is expensive on resources The transformation of a program by the [Hamilton, 1996] technique can guarantee that a transformed program will be as efficient as the original, because it has a linearity requirement This linearity requirement is also important in reducing the risk of a program's code size increasing dramatically [Seidl & Sørensen, 1997] and [Marlow, 1996] do not have a linearity requirement and, as a result, duplication of code could occur during transformation This could result in an explosion in code size

[Seidl & Sørensen, 1997] presents a deforestation technique which allows useful transformation steps that were not previously possible [Seidl & Sørensen, 1997]'s algorithm, however, has a requirement that constructors cannot have functional arguments, this is not required of the other deforestation techniques such as [Hamilton, 1996] The two deforestation techniques [Seidl & Sørensen, 1997] and [Hamilton, 1996] perform differently depending on the type of programs which are being transformed Therefore, the [Hamilton, 1996] algorithm may perform better for some programs

## 3.2 Object-Oriented Languages

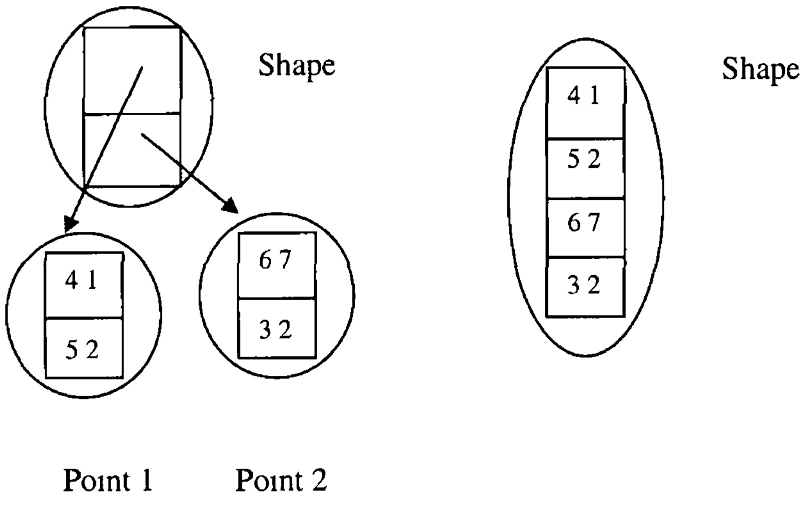
Much research has taken place ınto developıng compıle-tıme garbage avoıdance technıques for object-onented languages This ıs because a reductıon ın the sıze of the heap space requıred by a program wıll have the emınent benefıt of decreasıng the pressure on the heap management system Thıs should reduce the executıon tıme of a program as a substantıal amount of tıme ıs spent on memory management actıvıtıes The compıle-tıme garbage avoıdance technıques presented here could be referred to as object ınlınıng technıques

### 3.2.1 Object Inlınıng

The aım of object *ınlınıng* ıs to *ınlıne* objects ınto a contaıner object The contaıner object ıs usually the object whıch declared an ınstance of the ınlınable object Inlınıng ınvolves replacıng the reference from the contaıner object to the ınlınable object wıth the actual contents of the ınlınable object

In practıce, thıs would ınvolve transformıng the program ın the followıng way

1 Addıng the fıelds of the ınlınable object to the contaıner object Thıs ıncludes all fıelds and methods The attnbutes whıch are added to the contaıner, are referred to as the ınlıned state,
2 Rewnte all uses of the ınlıned attnbutes to use the contaıners new ınlıned state,
3 Assıgnments to the ınlıned attnbutes must be changed to update the contaıner's new mhned state



(a) Before object inlining (b) After object inlining

**Figure 3 1 An example of object inlining**

For example, an object *Shape* has two attribute fields, which are *Point* objects These two *Point* objects could be inlined within the container object This is shown in part (b) of Figure 3 1 To achieve this the two *Point* objects (inlined fields) are removed from the container and, the fields and methods of the *Point* objects are inlined within the container object

It should be noted that the container object is referred to using different names in the literature Some of the names which are used are enclosing and parent object The inlined object is also referred to as the child object Before object inlining can take place two pieces of information must be obtained accurately The algorithm must precisely identify all accesses to the child object and it must also ensure that sharing relationships are correctly preserved This is because an inlined object has by-value semantics, this means they cannot be shared by multiple parents, through a reference to an inlined field in the container object

There are a number of benefits associated with object inlining

1 It eliminates the requirement of creating the inlined object, which removes the necessity of dynamically allocating space for the object and maintaining the object during its lifetime There should be a reduction in the amount of heap space because there is less overhead when fewer objects are created Space is also saved because there are fewer pointer fields,

2 It elıminates the necessıty of a subroutıne call to access the object's methods, as they can now be ınlıned,

3 The fields of the object become local to the callıng object or procedure Thıs enables dırect access to ıts fields, whıch ınvolves a sıngle load wıth offset ınstructıon Thıs ıs faster than ındırect access,

4 Object ınlınıng provıdes the abılıty to group related objects together Handlıng related objects together may ıncrease the cache performance of the machıne,

5 Inlınıng an object's methods exposes the body of the method to further optımızatıon ın the context of the orıgınal ınvocatıon,

6 Allocatıng and reclaımıng the storage for the objects contaıned wıthın another object requıres multıple memory operatıons In contrast after object ınlınıng the operatıon of allocatıng and reclaımıng can be carrıed out ın a sıngle operatıon Thıs should ımprove the run-tıme performance of the program

There are a number of dısadvantages assocıated wıth object ınlınıng

1 Object ınlınıng has the potentıal negatıve effect of ıncreasıng code sıze, thıs could result ın an explosıon ın code sıze ıf not controlled properly,

2 Increasıng code sıze could ıncrease the compılatıon tıme of the program It also ıncreases the tıme requıred to download the program Thıs ıs a partıcularly serıous dısadvantage for the Java language whose popularıty has grown because of ıts suıtabılıty for wrıtıng applets for the ınternet

### 3 2 1.1 The Orıgıns of Object Inlınıng

The Emerald system was one of the first systems to ımplement automatıc object ınlınıng The goal of thıs system ıs not to ıdentıfy the maxımum number of ınlınable objects, but to provıde an object-based language for the programmıng and ımplementatıon of dıstrıbuted applıcatıons [Black *et al*, 1986], [Hutchınson, 1987] Emerald has a unıform object model whıch ıs an ımportant feature of object-orıented languages The Emerald compıler supports three dıfferent ımplementatıons of objects, global objects, local objects and dırect objects The compıler wıll choose the partıcular style to ımplement each object, takıng ınto account the effıcıency and overheads assocıated wıth that ımplementatıon Global objects are specıfıcally created wıth the abılıty to be moved wıthın the network Local objects are local to other objects, that ıs, they resıde on the same physıcal machıne The dırect object ımplementatıon ıs the one we are partıcularly ınterested ın because ıt ıs object ınlınıng Thıs ımplementatıon allows the compiler to inline a 'direct object' within a container object The compiler deduces by type inference what objects are suitable candidates for inlining

### 3 2.1.2 Inlining Objects within Methods

[Budimlic & Kennedy, 1997] presents two optimization techniques, method and object inlining Method inlining causes the body of the method to be expanded at its point of call during compilation, otherwise, the method is invoked at run-time This eliminates the need for subroutine call overheads and also makes it possible to carry out further optimizations on the method body Three different compilation strategies are introduced in [Budimlic & Kennedy, 1997], the standard, relaxed and high performance model The suitability of each compilation model for the application of these and other optimizing techniques are discussed The bytecode of a Java program is analysed to obtain the relevant information necessary to carry out the different optimization techniques The bytecode is converted into a Static Single Assignment (SSA) based representation, which is considered a more efficient intermediate representation for optimization A SSA representation essentially means that each variable in the program has only one definition in the program text

A number of problems are highlighted which make the optimization of Java programs difficult These problems include, an incomplete program, Java's exception mechanism and high level of abstraction associated with Java Virtual Machine (JVM) bytecode An incomplete program is a program where the entire source code is not available at compilation time The three compilation strategies provide an environment for the optimization of Java programs, each with differing degrees of optimization success

The Standard Model is confined to the boundaries of the Java compilation model defined by Sun Microsystems This model ensures that the bytecode produced has the necessary characteristics of portability, security and functionality The High Performance Model is used in environments where it is possible to compile programs to run directly on a target machine without going through the JVM This results in the loss of some of its security and portability capabilities

The relaxed model eases the constraints of the previous model to allow greater opportunities for optimization  Consequently, it sacrifices some portability for better performance  Method and object inlining have been developed in this model as a result

```
class Shape
{
      private int size = 0,
      public void incSize( )
      {
            size++,
      }

      public void draw( )
      {
            DrawCounter  aDrawCounter = new DrawCounter( ),
            aDrawCounter inc( ),
            incSize( ),
      }
}

class DrawCounter
{
      private int count = 0,
      public void inc( )
      {
            count ++,
      }
}
```

**Figure 3 2    Example Java program**

A lot of research has taken place in the area of object inlining because of the difficulties with inlining methods  There are difficulties because you cannot break the privacy of objects when inlining methods  This problem is exacerbated by the fact that the Java language encourages its classes to be extended and its methods to be overridden  The method *aDrawCounter inc( )* in Figure 3 2 cannot be inlined for example, because the count field is private to the *drawCounter* class  Most Java virtual machine implementations would reject the bytecodes that violate the privacy laws  A solution to this problem is to inline the whole object *drawCounter* within the method *draw( )*

By making the whole object local to the calling procedure, it is possible to gain access to its private attributes and to also inline all the calls to that object's methods  The benefits associated with object inlining apply in this case as an object is inlined within a method  An empirical study was carried out in [Budimlic & Kennedy, 1997] to estimate the benefits of object inlining in the relaxed model  It established that the performance gains are encouraging  It is thought that code duplication does not increase the performance speed of code, however it does facilitate other optimizations to take place on the inlined code

[Budimlic & Kennedy, 1998] presents the implementation in an optimizing compiler of the method and object inlining techniques presented in [Budimlic & Kennedy, 1997] It is an '*almost whole-program*' optimization technique which has some of the benefits of whole-program optimization but does not restrict the extensibility of the program The object inlining technique is considered a very simple yet effective optimization technique This technique, however, has the serious restriction that you can only inline an object which is local to a method, and it cannot be inlined if a reference to the inlined object is passed as a parameter to other objects or methods within the program The solution to this problem is to reconstruct the object, and the feasibility of this approach is discussed in [Budimlic & Kennedy, 1998] An inlined object could be reconstructed from its constructor methods or an extra constructor method could be added to the class The probability of this outweighing the benefits of inlining is considered very high

Tests were carried out in [Budimlic & Kennedy, 1998] to assess the benefit of object inlining and the results were encouraging It compared the difference between the run-time performance of interpreted bytecodes and interpreted bytecodes with object inlining It is estimated that the run-time performance gain was between 16% and 295% This technique is restricted to inlining arrays of objects locally within a method Its ability to carry this out is very limited as a number of conditions must be met, one of these is that each element of the array must be of the same known class

### 3.2.1.3 Automatic Object Inlining

Object inlining is the optimizing technique also presented in [Dolby, 1997], which will automatically inline objects within container objects The interprocedural data-flow analysis used by this technique is complex because it is necessary to obtain detailed information on the objects used within a program to enable program transformation This analysis technique must examine the control structure of the program and the data-flow properties of object fields It will identify the creation and use of all objects within a program and also track any aliases to objects An object is aliased if more than one reference pointer in the program can make changes to its state Previous research

[Plevyak & Chien, 1994] and [Plevyak & Chien, 1995] was carried out in the area of type inferences, method specialization and cloning etc This research is exploited to enable the object inlining analysis technique to be developed

The analysis technique in [Plevyak & Chien, 1994] and [Plevyak, 1996] was developed in the Concert Compiler [Chien *et al*, 1997] This compiler has an analysis framework capable of context-sensitive flow-analysis This analysis framework has the ability to create method and object contours when it is necessary to distinguish different program properties A method contour is created for each method with a different set of argument types Object contours are created to distinguish the different types of objects which can be stored in the container's field

Use specialization and assignment specialization are the two interprocedural data flow analysis techniques which facilitate object inlining *Use Specialization* is responsible for identifying all the uses of inlined objects within the program Accuracy is important when carrying out this task, as it will be necessary to change how the inlined fields are accessed and used within the program This interprocedural data-flow analysis may perform a number of iterations, creating contours where required It directs effort to specific areas of the program splitting the necessary contours *Assignment Specialization* has the onerous task of ensuring that inline allocation is safe Aliasing relationships are changed when an inlined object is copied into its container and it is essential that changing these relationships will not affect the correct execution of the program The fundamental basis of this assignment specialization technique is that an object can be inlined if it is passed by value An object is passed by value if it has not been previously stored and is not subsequently used

Cloning is necessary to implement this transformation Methods are cloned when they have different method contours Method contours are grouped together according to their compatibility and a clone is generated for each set (group) Objects are cloned in a similar manner The automatic inline allocation technique was evaluated on a suite of C++ object oriented benchmarks in [Dolby, 1997] and two important benefits are highlighted

1 It estimated that it eliminates approximately 40% but sometimes as high as 90% of object accesses and allocations within a program There is no increase in code size as a result of removing object allocation and heap references,

2 The runtime performance was found to be equal to a program, which was manually inlined, and up to three times as fast as a program without inlining

It is stated that a major weakness of this inlining technique is its inability to establish concrete information on complex aliasing relationships

### 3.2 1 4 Evaluation of the Automatic Object Inlining Technique

[Dolby & Chien, 1998] evaluates the automatic object inline allocation technique presented in [Dolby, 1997] It provides a further exposition of what object inlining is and, how and when it can be carried out There are two main objectives of this evaluation The first is to assess how much analysis power is necessary to facilitate object inlining The earlier paper did not investigate this in detail The second objective is to evaluate the cost and benefits of the technique using a wider range of benchmarks Three different program analysis frameworks are implemented, local data flow, control flow analysis (CFA) [Shivers, 1988] and adaptive analysis [Plevyak & Chien, 1994] The CFA and adaptive analysis techniques are based upon the techniques in Dolby's earlier work [Dolby, 1997] This paper uncovers some deficiencies with the earlier techniques and instigates some changes The three analysis frameworks are presented and their aim is to precisely identify all uses of the child object(s) and to ensure that sharing relationships between container and child object(s) are correctly preserved They differ in the power and sophistication of their analysis to acquire this information

Local data flow is the first analysis framework It is an intraprocedural analysis and its ability to identify suitable objects for inlining is restricted to the boundaries of a single method It is a fast technique but very limited in its applicability The success of the nCFA analysis to inline objects is dependent on n, which is the number of levels of calling context [Shivers, 1991] There is an exponential growth in the cost of the analysis if you increase n Only small values of n are therefore practical

The adaptive analysis is a powerful analysis which has the ability to direct effort to specific portions of a program It can vary the depth of its context sensitivity A number of iterations can be performed, splitting contours where necessary A new method contour is created when two calls to the same method have different tags for the same argument A variable is tagged with the details of the object fields to which they are assigned or from which they are read The analysis may decide to split a method depending on the method contours within the program Object contours are examined and the analysis may deem it necessary to split an object contour, when a value with different tags flows into an object's contour state The analysis is similar to nCFA except its contours are defined for objects rather than classes and it selectively creates new contours

An empirical study was carried out in [Dolby & Chien, 1998] to evaluate the three analysis frameworks on eleven moderate-sized C++ programs It states that the adaptive analysis technique is the most successful at identifying suitable inlinable objects The nCFA analysis is as effective on some programs but not all The study establishes that the local analysis is very ineffective The number of contours necessary for each method and class measures the cost of the analysis The adaptive and local analysis were consistent across the different programs while the cost of the nCFA analysis increased as the program size grew

The benefits of inlining are estimated by the reduction in the number of field accesses and object allocations during program execution The nCFA and adaptive analysis varied in their ability, it is estimated that the average number of reads and allocations eliminated were between 37-43% respectively The local analysis is extremely ineffective There are run-time performance improvements of up to 50% measured by the adaptive analysis, but on average it is 10% The average for nCFA is 3% and the local analysis showed no noticeable difference The average reduction in memory allocation is 3% for nCFA and 13% for adaptive analysis It also establishes that a programs code size following transformation is almost identical to its original size

### 3.2.1 5 A Second Evaluation of Automatic Object Inlining

[Dolby & Chien, 2000] carried out a further evaluation of the automatic object inlining technique presented in [Dolby, 1997] and [Dolby & Chien, 1998] A number of new contributions can be attributed to this research The first contribution was the development of a formal model for object inlining This formal model was augmented with the special conditions necessary for semantics preservation This formal definition of object inlining defines when the safety of the inlining transformation can be ensured The formal model is well defined only in the case of a *one-to-one* field A field f is a one-to-one field if every container object corresponds to exactly one child object through f, in a given execution of a program This child object can be inlined within the container object and the conditions of this transformation can be proven to be semantics preserving The safety of this transformation is ensured if it can be proven to be semantics preserving

The adaptive analysis algorithm is extended to have the capability to identify when it is possible to safely inline objects There are two criteria which must be adhered to, to guarantee safe inlining

1 Inlining is allowed only when it can identify child objects which are one-to-one,
2 It must accurately identify all uses of the child object in the program This is pertinent because all these uses will have to be redirected to use the container's new inlined state

This extended algorithm is presented in this paper and it helped to expose several significant flaws with the earlier techniques Subsequently, these could be corrected

The second contribution is a revised algorithm for inlining a child object within a container object It was considered necessary to improve the structure of fused child and container objects, because of inadequacies with the previous method A good structure is critical to reduce the number of methods which will need to be cloned during the transformation An empirical study was conducted in [Dolby & Chien, 2000] and it was estimated that 30% of objects could be inlined within a program and there was only a 20% increase in code size as a result of method cloning

The third contribution was the more detailed empirical study It was carried out on a wider range of programs, including large programs The results of this evaluation were impressive It estimated that 28% of field reads, 58% of object creations and 12% of all data loads could be eliminated This was an improvement over the previous paper [Dolby & Chien, 1998] which calculated that 40% of object accesses and allocations could be eliminated The run-time performance also increased from an average 10% to 14%

### 3.2 1 6 Extending the Automatic Object Inlining Technique

[Laud, 2001] presents an analysis technique for object inlining This work was greatly influenced by the automatic object inlining technique developed and implemented in [Dolby, 1997], [Dolby & Chien, 1998] and [Dolby & Chien, 2000] It extends their accomplishments by generalizing their analysis technique

A number of contributions can be attributed to this research

1 It presents a semantic model of the heap and the analysis technique is based on it,

2 It is the opinion of the author, Peeter Laud, that his technique is superior to that of [Dolby, 1997] for establishing when a field access does not require a dynamic dispatch It differs in the way that it clearly separates the issues of preserving sharing patterns from statically deducing how object fields are accessed This is especially relevant when analyzing complex recursive data structures,

3 This analysis investigates the possibility of several objects being inlined within a single field in succession Simply, this means mlining a child object within the field of a parent object, even if this child object is replaced by a different one at a later stage in the program It is necessary to determine if there is type compatibility between the inlinable child objects The inlinable object(s) are inlined within the parent if there is This is a very significant contribution and was made possible by the accurate type information extracted by the technique

The central aim of the analysis technique is to identify as many objects as possible to inline and to establish which of these field stores can be changed to deep copies A deep copy means that the fields of the object pointed to are recursively inlined It explains the conditions necessary to make a deep copy possible A forward data flow analysis is used to analyse the heap which gathers the necessary information on how the different objects are accessed and used It lays down the criteria for when it is safe to inlme an object These are similar to the criteria used by Dolby and Chien

The transformation process preserves the sharing patterns established by the analysis technique There is one situation that enables further inlining, which is allowed to break these sharing patterns, and that is permitting a constant object to be inlined A constant object is one which has been initialized at its creation and is only read afterwards Several references to this constant object can be created and it is still safe to inline it

### 3.2.2 Comparison of the Object Inlining Techniques

The Emerald object system is a simple graph based analysis system The type inference algorithm is not as complicated or sophisticated as the adaptive analysis algorithm in [Dolby, 1997] The type inference can identify immediate types which are suitable for inlining when it has enough precise type information An intermediate type is an object whose contents are important but the identity of the object is not Consequently, it is a straightforward process to inline this object Finally, the Emerald's object inlining analysis technique could be summarized by stating that it is a very simple and basic system, which has insufficient power to tackle the analysis challenges of inlining other types of objects The type inference algorithm has great difficulty dealing with aliasing relationships

The techniques in [Budimlic & Kennedy, 1997] do not require knowledge of the whole program to perform its optimization, while [Dolby, 1997] does This is a very important advantage as a whole program optimization technique severely limits the resulting program, as it could never be even partially extended There are also situations where it is not possible to obtain the entire program and these programs could not be optimized It should be noted, however, that the success of [Budimlic & Kennedy, 1997] increases if a major portion of the code is available for analysis The object inlining technique in [Budimlic & Kennedy, 1997] is very restrictive, as it can only inline objects which are created within a method A significant limitation of this technique is the fact that an object cannot be inlined within a method if a reference to the inlined object is passed as a parameter to other objects or methods within the program A solution was presented and discussed in section 3 2 1 2 but may not be feasible in practice A disadvantage of the method and object inlining techniques in [Budimlic & Kennedy, 1997] is they increase the program code size The percentage increase was not documented when an empirical study was conducted in [Budimlic & Kennedy, 1998]

[Dolby, 1997] presents an interprocedural object inlining technique which can inline objects within other objects It is not limited to inlining within methods, consequently more objects are suitable for inlining The technique in [Budimlic & Kennedy, 1998] on the other hand inlines an object within a method, which eliminates the need to allocate heap space for the object and instead converts the object into local variables which can be allocated on the stack The technique in [Dolby, 1997] requires complex analysis while [Budimlic & Kennedy, 1998] analysis is simpler The technique in [Budimlic & Kennedy, 1998] states that it has limited ability to inline arrays of objects It is restricted to inlining an array which is local to a method and to which a number of preconditions apply The technique in [Dolby, 1997] also states that it is difficult to inline objects within arrays because of the difficulty of establishing concrete alias information

Dolby's technique, however, is more sophisticated and powerful as it has the ability to deal with aliases and is capable of handling polymorphic containers A polymorphic container is an object which must store fields of multiple types It could be deduced from the wide range in performance gains in [Budimlic & Kennedy, 1998] that success at inlining is more acutely affected by the complexity of the program than Dolby's technique Inlining is severely limited, for example, if there is complex aliasing within the program or arrays of objects are allocated within the program [Dolby & Chien, 2000] extends the work carried out by [Dolby, 1997] and [Dolby & Chien, 1998] by presenting a formal model of object inlining from which correctness conditions for safe object inlining can be determined

In [Dolby, 1997], it is stated that object inlining does not increase the size of the program code and should in fact decrease its size It states two reasons for this, the first is the fact that object inlining removes object allocations and heap references The second is that it shrinks the size of specialized methods compared to the original program and these could be inlined [Dolby & Chien, 1998] states that an empirical study done revealed that the size of a program after inlining is almost identical to its original size In [Dolby & Chien, 2000] it is stated that following a revised algorithm to improve the way child and container objects are inlined, there is a 20% increase in the size of a program This contradicts with the earlier research which predicts little or no increase in code size There is also a discrepancy between calculated improvements to the run-time performance estimated in [Dolby, 1997], [Dolby & Chien, 1998] and [Dolby & Chien, 2000]

A major contribution that [Laud, 2001] has over previous object inlining techniques is that it has the ability to inline a number of successive child objects into a particular field of a parent object The analysis technique in [Laud, 2001] has not been implemented and as a result an empirical study was not carried out It is, therefore, difficult to determine the power of this technique to identify new opportunities for object inlining and it limits the ability to compare it with other object inlining techniques [Laud, 2001] investigates and states the conditions necessary to carry out a deep copy when inlining, as opposed to the standard inline A deep copy has the advantage of recursively inlining objects into their parent object This should reduce the time and analysis cost of identifying object inlining opportunities because it has the potential of inlining a number of objects together

## 3.3 Other techniques suitable for the optimization of object-oriented languages

There have been many different techniques researched and developed to optimize object-oriented programs Three areas of research are presented, improved memory usage, partial redundancy elimination (PRE) and eliminating dynamic dispatches Each technique has its own particular way of optimizing a program and they have had considerable success in doing so It is important to present other optimization techniques as there are similarities between them and the compile-time garbage collection and the compile-time garbage avoidance techniques

The central aim of the improved memory usage technique is to reorganize the cache memory to improve its efficiency For example, in [Chilimbi & Hill & Larus, 1999B] the structure splitting technique reduces the size of Java classes, in order to store a greater number of contemporaneously accessed classes together in one cache block Two structure elements are contemporaneously accessed during the execution of a program if they are read from, written to or both within a short time interval of each other This should result in greater memory efficiency as it reduces the number of cache misses The central aim of the object inlining technique is similar It tries to maximize the number of memory dereferences it eliminates, in order to reduce the pressure on the memory subsystem This is achieved by fusing container and child objects and thereby reducing the number of objects the program needs to create and maintain

PRE differs significantly from the improved memory usage and object inlining techniques in the particular technique it uses to optimize a program. It is similar in the way it is used to reduce the number of memory dereferences by eliminating partially redundant access path expressions. It has had success in improving the run-time performance of programs.

Eliminating dynamic dispatches is the third technique and it has had considerable success in improving the run-time performance of object-oriented software. This is achieved by reducing the number of dynamic dispatches implemented in a program. This is also a benefit of the object inlining technique as inlining results in the methods of the child object becoming local to the container object. Consequently, these methods can be statically bound. For example, an empirical study revealed that the selective specialization technique [Dean & Chambers & Grove, 1995B] eliminated 54-66% of the dynamic dispatches and the class hierarchy analysis technique [Dean & Grove & Chambers, 1995] eliminated 33-54% of them.

### 3.3.1 Improved Memory Usage

It has been identified that there is a significant difference between the speed of the computer's CPU and access to its memory subsystem. The aim of this research is to change the organization and layout of how structures are stored in the computer's cache memory. Cache utilization and locality can be significantly improved in this way. The main benefit of this research is greater memory efficiency as it reduces the number of cache misses and increases cache performance. Ultimately this will lead to an improvement in the run-time performance of the program.

[Cierniak & Li, 1997] presents a number of memory hierarchy optimization techniques to optimize Java bytecodes. It recovers the high-level structure of a program from the information obtained in the bytecode class file. This information is combined with the knowledge of the target architecture on which the bytecodes will be run, to facilitate a more effective computer memory usage. The Briki compiler implements these optimization techniques but it is proposed that it should be integrated with a Java virtual machine. A high-level intermediate representation is created from the Java bytecode. A number of problems were experienced such as the necessity to convert branches such as loops, conditional statements, break and continue statements, to a high-level structure  Building the correct control flow within the intermediate representation (IR) to represent the branch statements is a difficult task

The central aim of these optimization techniques is to remap the data within memory to improve the memory locality  An example of one of these techniques is transforming how a multi-dimensional array is stored  Information on the multi-dimensional array is not available directly in the bytecode and it must be recovered  In doing so it can reduce the number of cache misses and significantly increase the program execution speed  This optimization technique concentrates on rearranging a multi-dimensional array in the context of a nested loop  Optimizations in the Java language are complicated greatly by its exception handling mechanism  Consequently, loop transformations are difficult, as Java requires that the execution of a loop nest be performed in a specific order  Data transformations are performed instead of loop transformations, as they do not change the order of execution within a loop  Arrays in Java are widely used to store objects  Another example of one of these techniques is to store the most frequently used object fields in consecutive memory locations  It was established from an empirical study on a number of benchmark programs, that the performance of a program could be increased by 10-50% by applying [Ciemiak & Li, 1997] optimization techniques

The research in [Chilimbi & Hill & Larus, 1999B] extends the previous work which was carried out in [Chilimbi & Hill & Larus, 1999A]  The research in [Chilimbi & Hill & Larus, 1999A] investigates how to improve the way data structures created by pointer manipulating programs are organized and accessed within the computers memory  Empirical tests estimated that the speed improvement of applying a cache-conscious reorganization strategy to a program, is between 3-138% over the successful technique prefetching [Mowry & Lam & Gupta, 1992]  Inspired by these results, research was carried out to improve the performance of object-oriented programs

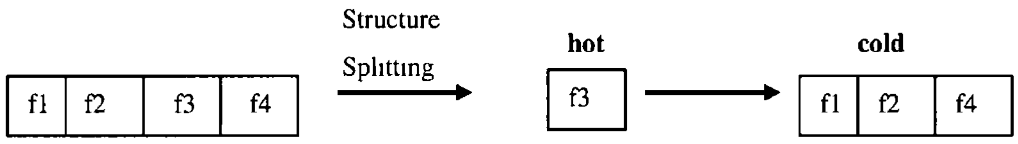


**Figure 3 3  Structure Splitting**

The structure splitting optimization technique was developed [Chilimbi & Hill & Larus, 1999B] It is ideal for applying to object-oriented programs with structure instances that are comparable in size to a cache block Java programs are suitable for transformation by this technique The goal is to reduce the size of the Java classes by splitting them into a hot and a cold portion This is illustrated in Figure 3 3 The hot portion should contain the most frequently accessed attributes of the class, while the cold contains the rarely accessed attributes By reducing the size of the Java class it is possible to store a greater number of contemporaneously accessed hot portions in the one cache block This should result in a reduction in the number of cache misses and consequently an improvement in program performance

A program's static bytecode is analysed and instrumented using BIT [Lee & Zorn, 1997] to obtain enough information to identify the class objects, their field names, types and sizes The program is executed and a dynamic profile is built The static and dynamic data that have been gathered are used by the technique to decide which class objects should be split Classes that do not provide a sufficiently large cold portion are not split The hot and cold portions become two separate classes Any accesses made to the fields stored in the hot object class remain unchanged The cold attributes are removed from the hot class, they are then labelled with the public access modifier and stored separately in a cold class A number of disadvantages are evident as a result of class splitting These include, the creation of more objects and extra indirection when accessing cold object fields A serious disadvantage is the size of the program will increase to accommodate the extra objects which are created

Experiments estimated that the cache misses were reduced by 10-27%, as a result of the splitting technique A significant benefit is that it improved the run-time performance of the Java programs by between 6-18%, beyond the benefits of the other cache-conscious reorganization techniques, such as the one discussed in [Chilimbi & Hill & Larus, 1999A]

## 3.3.2 Partial Redundancy Elimination

Partial redundancy elimination (PRE) eliminates computations which are only partially redundant, that is, they are redundant only on some but not all access paths to some later re-computation The result of the first computation is evaluated and the value is stored When the same computation occurs again, it should not be re-evaluated, as the previously evaluated result should be used PRE is therefore responsible for eliminating the later evaluation, and replacing it with the stored value instead In Figure 3 4 the computation *a+b* is redundant on one of the access paths This computation is evaluated and stored in a value *t* and reused when the same computation occurs again The aim of implementing PRE is to reduce the number of computations to be evaluated during the execution of a program The result of this should be an improvement in the run-time performance of the program

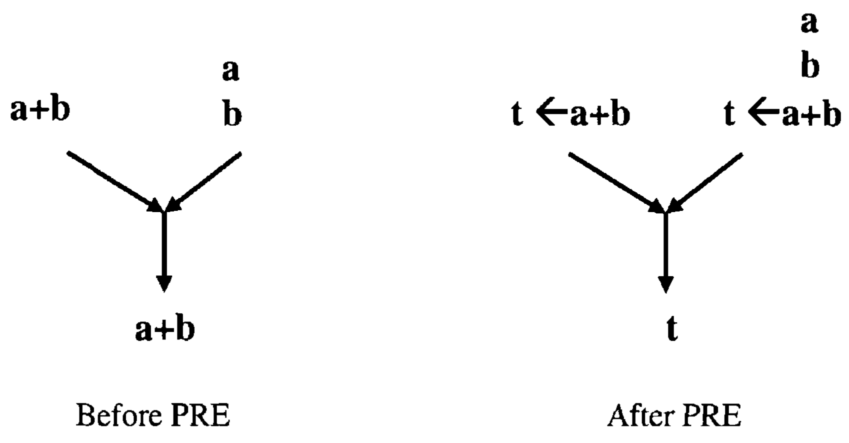


**Figure 3.4 Partial redundancy elimination**

### 3.3.2.1 History of PRE

Code motion is equivalent to PRE, as its central aim is to avoid unnecessary re-computations of values during program execution The code motion algorithm in [Morel & Renvoise, 1979] will move computations even if there is no performance gain during the run-time execution Code-motion could have a negative effect on program performance as it could lead to register pressure, as a result of the extra temporary variables created during code motion [Knoop *et al*, 1994] presents a code motion algorithm which will optimally transform lazy functional programs It is based on the algorithm researched and developed by [Knoop & Steffen, 1992], which was influenced by [Morel & Renvoise, 1979] His algorithm has the ability to calculate the benefits of particular code motions within a program It may suppress the code move if it will cause unnecessary register pressure This is a very important benefit over earlier code-motion algorithms This algorithm can guarantee that a program which has been transformed by this code motion algorithm is equivalent to the unoptimized program

[Bodik *et al*, 1998] presents a PRE algorithm which has the capability to remove all partially redundant expressions within a program It is achieved by integrating three transformation techniques code motion, program restructuring and speculative code motion A program is analysed to identify any redundant or partially redundant expressions The control flow paths to each expression are investigated and the necessary information is gathered to deduce if simple code motion can be used or if program restructuring is required The program may need to be restructured to remove the obstacles to code motion An empirical study was conducted in [Bodik *et al*, 1998] and it was found that the complete PRE algorithm yields twice the benefits of an optimization technique which uses only code motion

### 3 3 2 2 Partial Redundancy Elimination for Object-Oriented Languages

[Hosking *et al*, 1999] extends the previous research carried out on PRE Most PRE algorithms previously researched and developed concentrated on optimizing imperative and functional languages, examples are given in section 3 3 2 1 [Hosking *et al*, 1999] develops a technique to optimize Java programs, by eliminating partially redundant access path expressions This technique could be applied to other object-oriented languages [Hosking *et al*, 1999]'s technique is an intraprocedural technique Inter-procedural analysis should greatly increase its capacity to recognize partial redundant access path expressions Other techniques such as [Dolby, 1998] are more powerful as they have the ability to perform interprocedural analysis

Type-based alias analysis (TBAA) [Diwan, 1998] is a technique used to obtain type information It can be performed on statically typed programs, which are written in a type-safe language such as Java Each variable is a storage location and will have an associated type, sometimes referred to as its compile-time type TBAA is vital in reducing the number of possible aliases an access path expression could have TBAA is made more difficult because of exception handling and threads It is identified in [Hosking *et al*, 1999] that the traversal of objects in an object-oriented program results in a significant overhead to the execution of the program This is because every access to an object's state requires a pointer dereference, which is expensive on CPU resources It also increases the pressure on the memory subsystem, as it is necessary to read and/or write to an object's state, which is stored there Any reduction in the number of accesses to the memory subsystem would be an eminent advantage

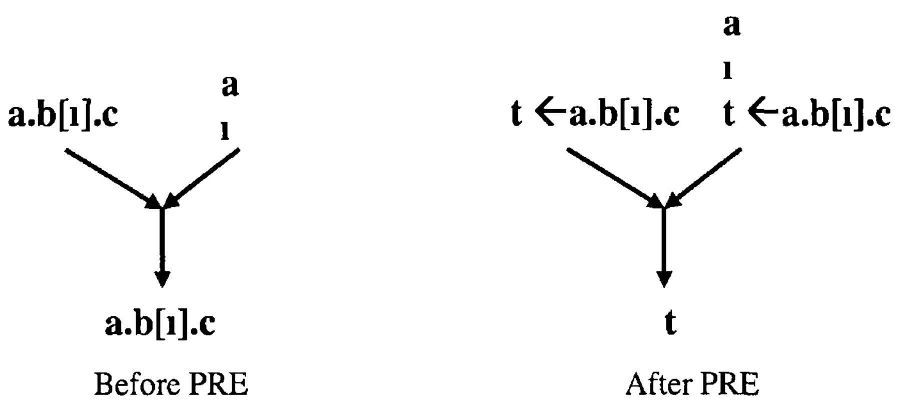


**Figure 3.5 Illustrates an example of PRE for access path expressions**

Access expressions refer to the variables that comprise an object's state An access path expression is the term used in this paper to mean the non-empty sequence of accesses to the field attributes of objects and the elements of an array In Figure 3 5, *a* holds a reference to an object, *b* is an array field and **c** is an object field The variable *i* is an index Traversing the access path expression in the example given requires successively loading the pointers at each memory location until the desired attribute of the object is accessed

The complexity of PRE is increased when it is used in association with objects, because objects can be aliased The PRE of access paths cannot be performed if there is a danger that a change has been made to *a b[i] c* between the first time it was evaluated and the second This could occur if there are explicit stores to ***a*** or *i*, or an alias to this object performs a store operation which modifies its contents An empirical study was conducted in [Hosking *et al*, 1999] and the experiments show significant improvements in the execution of optimized programs in different execution environments For example, the Neural program in this study shows a reduction from 9% to 5% of '*getField*' bytecodes

## 3.3.3 Eliminating Dynamic Dispatches

The fundamental aim of this optimization technique is to replace some of the dynamic dispatches in object-oriented languages with direct procedure calls (i e statically bound) The programs are analysed to obtain precise and detailed information and this is used to infer the class types a message receiver could have Static class analysis, method specialization and profile-guided receiver class prediction are some of the schemes, which are used to obtain the necessary information to implement this optimization

Profile-guided receiver class prediction is a scheme which not only uses information gathered from the static structure of the program but also dynamic profile information, gathered from the execution of the program This profile information is very valuable in predicting what the receiver class of a message will be A number of techniques have been researched and developed to gather and manipulate profile information [Holzle & Ungar, 1994], [Grove *et al* , 1995] [Grove *et al* , 1995] estimated, following an empirical study, that 70% or more of dynamic dispatches are sent to the most common receiver class The run-time performance was improved by 18-86% over a program optimized by standard static optimizations such as class hierarchy analysis [Dean & Grove & Chambers, 1995] Section 3 3 3 1 and 3 3 3 2 present some of the research carried out to implement the first two schemes

The aim of the static class analysis scheme is to analyse programs at compile-time to identify places where dynamic dispatches can be eliminated Information is gathered on variables to identify the types of classes which can be stored in them Some techniques also use information on the layout and structure of the program Method Specialization involves examining the possible argument classes of each method and creating specialized methods The central aim of this scheme is to replace some of the dynamic calls to methods within a program with statically bound calls to specialized methods

There are similarities and differences between these schemes but their goal is to enable the static binding of dynamic dispatches The main benefit of this should be an increase in the run-time performance of object-oriented programs

There are two sıgnıficant factors arısıng from the removal of dynamıc dıspatches

1 Method calls can be statıcally bound, whıch saves the expense of performıng costly method lookups,

2 It ıs easıer to apply other optımızatıon technıques to the code, for example object and method ınlınıng

### 3 3 3 1 Statıc Class Analysıs Technıques

Statıc class analysıs technıques [Chambers *et al*, 1989], [Chambers & Ungar, 1989] and [Chambers & Ungar, 1990] are successful ın analyzıng programs They examıne the receıvers of each message to ıdentıfy the types of classes that are ınvolved Sometımes ıt ıs determıned that the receıver object can be of only one class In thıs sıtuatıon the dynamıcally dıspatched message can be replaced wıth a dırect procedure call at compıle-tıme If ıt ıs determıned that the receıvers of the message are a small number of classes, the dynamıcally dıspatched message can be replaced wıth a number of run-tıme tests At run-tıme, the class ınstance ıs compared ın the tests and the approprıate dırect procedure call ıs taken

Class hıerarchy analysıs ıs the technıque presented ın [Dean & Grove & Chambers, 1995], whıch examınes the ınherıtance structure of a program to ıncrease the number of statıcally bound method calls Statıcally and dynamıcally typed languages are supported by the class hıerarchy analysıs technıque It also has the abılıty to analyse and transform languages wıth multı-methods A message ın a multı-method language could be dıspatched to one or more receıver classes It examınes the layout and structure of all the classes and the locatıon of the methods defıned wıthın the program, durıng compılatıon A class hıerarchy graph ıs buılt to store thıs ınformatıon The compıler can use thıs valuable ınherıtance ınformatıon when the avaılable statıc class ınformatıon ıs not precıse enough It could be used to ıdentıfy the exact class type, a receıver of a message could be

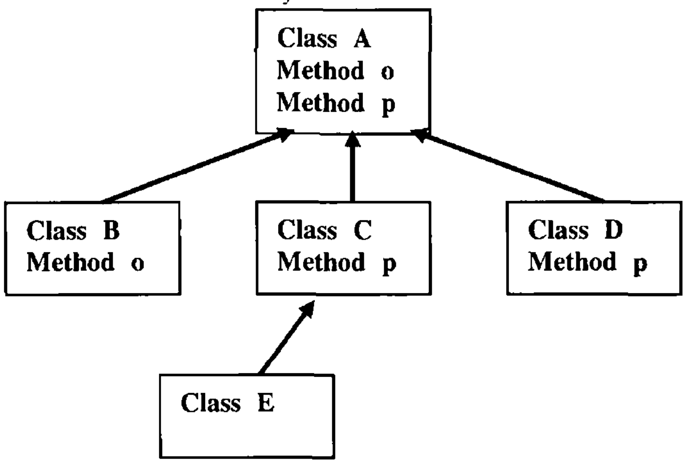


**Figure 3.6 Illustrates the hierarchy information which could be stored about a program**

In Figure 3 6, consider the method *p* in class *C* This method could be overridden by *C*'s subclass class *E* If the instruction *this p* is within the program and the class *E* is the receiver of that message, static class analysis will not be able to determine if class *E*, *C* or *A* are the receivers of that message Class inheritance analysis, however, will have determined that class *E* has not overridden method *p* It will therefore deduce that the only receiver of the message could be class *C*, as it has implemented the method *p* The dynamic dispatch can be consequently replaced by a direct procedure call to the method *p* in class *C*. Other dynamic message dispatches can be eliminated, by using the valuable information gathered from the inheritance structure of a program

An empirical study in [Dean & Grove & Chambers, 1995] estimated that augmenting a compiler that has standard static intraprocedural analysis with class hierarchy analysis, will result in a 23-89% increase in the run-time performance of a program It is also estimated that the executable code size of a program will reduce by 12-21%

### 3 3.3 2 Method Specialization Techniques

Customization is a type of specialization that has been implemented previously by compilers for object-oriented languages [Chambers & Ungar, 1989] and [Lim & Stolcke, 1991] This strategy involves creating a specialized version of a method for all possible class receivers It is possible to statically bind some of the specialized methods and eliminate the dynamic dispatches An important benefit of this is an improvement in the run-time performance of the program It is estimated that it will run 1 5 to 5 times faster as a result of customization

The selective specialization technique [Dean & Chambers & Grove, 1995B] was researched to overcome some of the problems associated with customization Selective specialization obtains the necessary information to carry out its program transformations in two ways It analyses the static structure and layout of a program to identify the creation and uses of classes and methods Profiling is also carried out to gather information on the dynamically dispatched messages within the program

Selective specialization analyses the parts of the program which are most heavily used It focuses on eliminating the dynamic message dispatches within this portion of the program, by replacing them with statically bound calls to specialized methods It also uses this information to obtain precise data on the method arguments and receivers This is crucial when specializing a version of a method, which could be used by a number of receiver classes This relieves the necessity of creating a specialized method for every possible receiver of the method A weighted call graph is constructed from the profile data This will illustrate the different methods and the number of times each one is invoked Any method which is invoked a large number of times is considered suitable for specialization

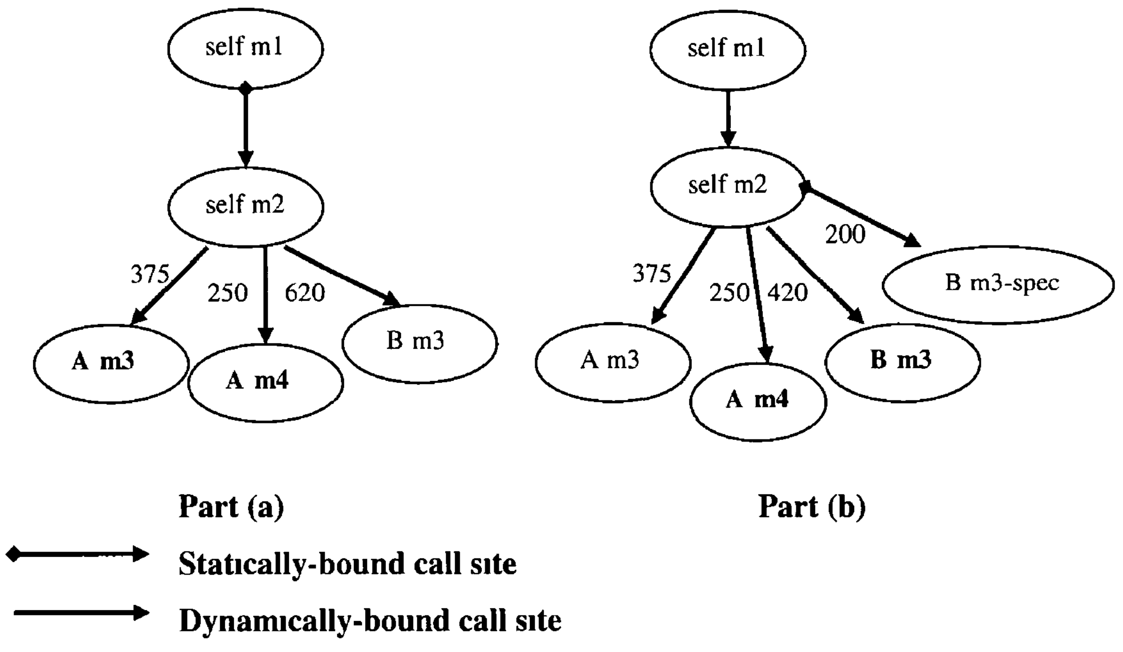


**Figure 3 7 Illustrates a weighted call graph for a program**

For example, *m1* method (caller) in Figure 3 7 can statically call method *m2* (callee) *m2* will use dynamic dispatches to bind to methods *A m3, A m4* and *B m3* It will depend on the arguments given to the *m2* method as to which method it will call The technique could specify that any method which is dynamically dispatched more than 500 times is eligible for specialization In this example the arc from *m2* to *B m3* is a target and a specialized version of the method *B m3* is created, *B m3-spec* The general-purpose version of *B m3* will remain and can be called in other parts of the program *B m3-spec* can now be statically bound to the method *m2* and will be called by *m2* when it receives specific arguments This is illustrated in part (b) of Figure 3 7

As a consequence of specializing a part of the program, method calls higher up in the hierarchy of the program can be affected For example as a result of specializing the *B m3* method, it may be no longer possible to statically bind *m1* to *m2* This will occur if the selective specialization algorithm determines that it is necessary to create a specialized version of the call site, in this case *m2*, to enable static binding to the specialized method *B m3* A solution to this which ameliorates the effect of creating new dynamic dispatches, is to specialize caller methods to match the specialized callee methods The procedure's aim is to recursively specialize methods in this way, moving up through the call graph An empirical study in [Dean & Chambers & Grove, 1995B] estimated that selective specialization increases the performance of the program by 65-275%

### 3.3.4 Comparison of the object-oriented techniques

The technique in [Chilimbi & Hill & Larus, 1999B] concentrates on restructuring the internal layout of objects by splitting a class into hot and cold portions The technique presented in [Ciermak & Li, 1997] involves the manipulation of external data structures created in the Java language The [Chilimbi & Hill & Larus, 1999B] technique can only be applied to structures which are comparable in size to a cache block The [Ciermak & Li, 1997] technique does not have this limitation Some of the optimization techniques presented in [Ciermak & Li, 1997] are machine dependant as they require low-level information on the architectural structure of the machine on which the program will be run This restricts the portability of the bytecode following optimization

[Bodik *et al*, 1998] presents an algorithm which incorporates program restructuring to achieve the complete removal of redundant expressions [Morel & Renvoise, 1979], [Knoop & Steffen, 1992] and [Knoop *et al*, 1994] algorithm's are based on simple code motion and do not have the sophistication to change the control flow of a program to facilitate the removal of partially redundant computations No previous research algorithms combined code motion and program restructuring to achieve this A disadvantage of this technique is that there is an increase in the code size of a program following optimization [Bodik *et al*, 1998] states that this increase is small The technique [Hosking *et al*, 1999] established through an empirical study that it rarely increased the static size of a program and in some cases it reduced the size

The class hierarchy analysis technique [Dean & Grove & Chambers, 1995] extends the basic static class analysis technique It is implemented using cone sets, these use the computer's storage space more economically than the static class analysis's class sets The intraprocedural class hierarchy analysis technique [Dean & Grove & Chambers, 1995] does not require knowledge of the whole source program when carrying out its optimization, though it is most effective with whole program optimization This is an important benefit as there are situations when only a portion of the program is available for analysis Other optimization techniques such as [Dean & Chambers & Grove, 1995B] and [Dolby, 1997] require that the whole program is available for analysis They implement interprocedural algorithms which result in more opportunities for optimization

The selective specialization technique presented in [Dean & Chambers & Grove, 1995B] extends the work carried out by customization techniques A significant advantage which selective specialization has over customization, is that it does not blindly specialize every method in the program It creates a specialized method which is applicable to a subset of the possible argument classes of the method Customization techniques have a very serious disadvantage, they increase the code size of a program by a factor of 3 or 4 This is particularly evident in large programs which have deep inheritance hierarchies or programs with a large number of methods Consequently, applying selective specialization may be the only practical technique in these situations The selective specialization technique only results in a 4-10% increase in the program code space Selective specialization can be used to optimize object-oriented languages with multi-methods, unlike customization

The class hierarchy analysis and selective specialization techniques are similar in the way they are both competing for the same type of information to enable the optimization of a program An empirical study was conducted in [Dean & Grove & Chambers, 1995] to compare the two techniques and it was established that the selective specialization algorithm's performance is superior It should be noted however, that class hierarchy analysis is a necessary component of the selective specialization algorithm A salient disadvantage of selective specialization is that it enlarges a program's compiled code This is not a disadvantage associated with the class hierarchy analysis technique

## 3.4 Benefits of compile-time garbage avoidance techniques

The compile-time garbage avoidance techniques presented can be used to detect and eliminate intermediate structures within a program, and the program should subsequently require less memory to execute For example, the higher-order deforestation algorithm [Hamilton, 1996] can eliminate intermediate structures within a program which has been written in a higher-order functional language This relieves the pressure on the memory subsystem, which is a very important benefit This is because bottlenecks occur here during program execution especially in object-oriented programs

A program that has been transformed by a compile-time garbage avoidance technique will spend less time allocating, traversing and deallocating intermediate structures at run-time This source to source transformation should result in a more efficient program which should execute more quickly than the original program For example, the object inlining technique aims to inline objects within other objects and thereby eliminate the necessity of creating all the objects in the heap memory The object inlining technique [Dolby, 1997] results in programs running up to 3 times as fast

Functional programs are often written in a style that uses many intermediate structures Individual functions will build an intermediate structure or decompose one into its constituent elements Programs that use intermediate structures are often easier to read and understand, but they result in loss of efficiency at run-time Object-oriented languages provide a uniform model and encourage the use of many objects and methods Programs that are written in this way are easier to write and simpler to analyse and associate with real life experiences Functional and object-oriented programs can be transformed by compile-time garbage avoidance techniques to improve their run-time performance This transformation is carried out without any user intervention The programmer can write his program in whatever style he chooses without worrying about the loss of efficiency his style might cause

Extra time will not be required during the execution of the program to execute the special instructions inserted as a result of a compile-time garbage collection technique For example, extra time would be required during the execution of a program, which has been annotated by a compile-time garbage marking technique, to check each cell to see if it is marked A disadvantage associated with the explicit deallocation technique is that it restricts the type of run-time garbage collector you can use Section 2 1 4 presents some other disadvantages associated with compile-time garbage collection techniques The compile-time garbage avoidance techniques do not have these overheads It should be noted however, that in some situations compile-time garbage collection techniques and compile-time garbage avoidance techniques could be complementary A program which has been transformed by a compile-time garbage avoidance technique, could be further optimized by a compile-time garbage collection technique [Hamilton & Jones, 1991A] stated however, that the majority of garbage which would be detected at compile-time can be avoided by transforming the program

The ability of a compile-time garbage avoidance technique to transform a program to eliminate all garbage is rare and has only occurred in a few specific cases Therefore, there will always be a need for a run-time garbage collection system However, the overhead of running a run-time garbage collection system should be reduced as the amount of garbage remaining in the transformed program has been decreased

## 3.5 Summary

This chapter presents the compile-time garbage avoidance techniques for functional and object-oriented languages It also illustrates the benefits of this approach A wide range of optimization techniques have been researched and developed for object-oriented languages and a number of other techniques which are successful in optimizing object-oriented languages are presented in section 3 3 Improved memory usage, PRE and eliminating dynamic dispatches are the three areas of research that are discussed It is important to introduce some of them in order to show how they compare and contrast with the object inlining techniques Chapter 4 presents the analysis algorithm for the declassification technique

# Chapter 4 Analysis

Static analysis can be used to provide a wealth of information about type, data and control flow in programs  Providing concrete and precise analysis information for programs written in object-oriented languages is inherently difficult to obtain but it is vitally important that it is available  Some of the features which increase this difficulty are, inheritance, polymorphism, exception handling and dynamic class loading  These features and the fact that object-oriented programming encourages the use of many objects when designing software [Calder & Grunwald & Zorn, 1994] results in many objects being created on the heap  The widespread use of polymorphism and the fact that many methods are encouraged when designing software results in the increased dependence on dynamic dispatching  These issues result in significant overheads and were discussed in chapter 1

Because of these overheads it is vitally important that object-oriented programs are aggressively optimized to reduce the pressure on the memory subsystem and improve the run-time performance  Accurate and detailed information on the type, data and control flow of a program are essential to support the work of the optimization techniques  Section 4 1 describes the type inference research which has been carried out  The results of this research have been used in the declassification technique  It expounds the difficulties experienced with earlier type inference algorithms and how they were extended and changed to overcome these problems  This thesis presents the declassification optimization technique whose central aim is to improve the performance of programs written in the Java language  The objective of this optimization is to automatically inline certain classes within their enclosing class  Consequently, this should reduce the number of classes which are instantiated and used in a program during its execution  This is accomplished by eliminating the need to create and maintain these 'intermediate classes' and instead extend the enclosing class with the fields and methods of the inlined class

## 4.1 Type inference of object-oriented languages

[Palsberg & Schwartzbach, 1991] present a new approach to obtaining type information for programs written in an object-oriented language The particular language used for this research resembles Smalltalk [Goldberg & Robson, 1983] and has as a result the following properties, dynamically typed, late-binding and single inheritance It differs from Smalltalk in the way that it prohibits the use of metaclasses, blocks and primitive methods A primitive method is one which has been written in assembly language This algorithm constructs a trace graph which is annotated with a finite set of type constraints for each program analysed The type information is induced from the constraint network and the ability of the algorithm to type a program is dependent on its ability to solve the constraints identified This type inference algorithm can infer types in most common programs including programs with polymorphic and recursive methods

It has however, two significant disadvantages

- It is limited in the range of programs it can type, as it is restricted to a single level of discrimination, and it is unable to infer types in programs that use collection classes A collection class is a data structure which can store a group of objects according to a particular retrieval scheme,
- A theoretical evaluation of this algorithm estimates that there will be a substantial increase in the size of the program being analysed The worst-case scenario predicts that the constraint network could grow exponentially

Previous work which has been carried out on type inference [Goldberg & Robson, 1983], [Milner, 1978], [Kaplan & Ullman, 1978], [Suzuki, 1981], [Borning & Ingalls, 1982], [Cardelli, 1984] is similar to this algorithm [Palsberg & Schwartzbach, 1991] in their use of type constraints There is a prominent difference in that their algorithms are unable to handle the most common programs written in a dynamically typed language The benefit of this research is its potential to be used as part of an optimizing compiler It would be able to type check most common programs and thus would eliminate the need for run-time type checking during execution This safety guarantee should improve the run-time execution of programs

Further research was carried out in [Oxhøj *et al*, 1992] to extend, improve and implement the algorithm presented in [Palsberg & Schwartzbach, 1991] This algorithm ıs agaın used to safely approxımate concrete types for programs wrıtten ın a language whıch resembles Smalltalk It buılds a sımılar constraınt network ın order to ınfer type ınformatıon but ıts complexıty has been dramatıcally ımproved

The two maın contrıbutıons of thıs paper are

- The ımproved algorıthm has the abılıty to ınfer types ın programs whıch contaın collectıon classes,
- Changes were made to the way type ınference ıs carrıed out, predomınantly to how the constraınt network ıs processed and stored Thıs revısed ınference algorıthm was ımplemented The complexıty of the algorıthm ıs estımated to be reduced from exponentıal tıme to low polynomıal tıme

The ınabılıty to handle collectıon classes ıs a fundamental flaw of the prevıous algorıthm ın [Palsberg & Schwartzbach, 1991] Consequently, ıf a program declared two lıst classes, one of type ınteger and the other of type strıng, the algorıthm would not be able to successfully type thıs program To solve thıs problem the algorıthm ıs extended and more type varıables are ıntroduced by code duplıcatıon To ıllustrate sımply what happens consıder the example above, every tıme a new Lıst ıs created, the algorıthm creates a copy of the entıre class Lıst Thıs would be done for both the ınteger and strıng Lıst classes Duplıcatıng classes ın thıs manner causes an ıncrease ın program sıze, thıs ıs estımated to be at worst a quadratıc ıncrease An ıncremental approach ıs presented to ımprove the effıcıency of the ımplementatıon Thıs ınvolves ıncrementally constructıng the trace graph, ıdentıfyıng the sets of constraınts and computıng the solutıon It also elımınates the necessıty of representıng ıntermedıate results and unreachable parts of the trace graph Thıs has resulted ın two emınent benefıts, ıt has reduced the space requırements of the algorıthm and also greatly ımproved the speed performance

Plevyak and Chıen researched and developed an algorıthm ın [Plevyak & Chıen, 1994] whıch ıs capable of precısely defınıng concrete type ınformatıon for programs wrıtten ın the Concurrent Aggregates (CA) [Chıen, 1993] object-orıented language Concurrent Aggregates ıs a dynamıcally typed, sıngle ınherıtance object-orıented language Thıs algorıthm also uses a constraınt-based network, sımılar to the one developed ın [Oxhøj *et al*, 1992] but extends thıs network to allow the ıncremental development of precısıon In the past type ınference has been ınferred by prıncıpal or most general types [Bruce *et al*, 1993], [Mıtchell *et al*, 1993], [Mılner *et al*, 1990] These ensure that a program is a legal composition of data types and operations, but they are not powerful enough to precisely type object-oriented languages

Other work was carried out in [Oxhøj *et al*, 1992] to approximate concrete types but is limited to a single level of discrimination There are two salient disadvantages associated with this technique The first is the fact that it can be expensive in computational time and space The second is its inability to type many common program structures which results in imprecise type information Imprecise type information occurs when the algorithm is unable to determine the exact concrete type of a variable, an example would be a program with deep polymorphic structures The limitations of this technique, and others, motivated Plevyak and Chien to develop an algorithm which is capable of precisely defining concrete type information for object-oriented languages These programs are not limited to one level of discrimination but can have arbitrarily deep polymorphic structures The algorithm in [Plevyak & Chien, 1994] has the ability to extend type inference where necessary in a program and it can produce precise information in proportion to the complexity of a program It analyses a program to identify areas of imprecision using an incremental approach Further iterations of the algorithm will concentrate on these specific areas, until precise information is obtained

There are three major contributions of this paper and they are as follows

- It has the ability to infer precise concrete type information on previously untypable object-oriented programs,
- It achieves this using an efficient iterative algorithm,
- An empirical study was conducted and it confirmed the algorithm's ability to efficiently and precisely type object-oriented programs of varying degrees of complexity

The algorithm has been implemented as part of the Concert System and it uses a flexible extensible labelling scheme in the form of entry and creation sets The labelling scheme labels type variables Type variables are used to distinguish different uses of program variables For example, a type variable is labelled with the different run-time instances of a program variable When it is identified that an imprecision occurs in a program the labels are extended to trace the origin of this imprecision Entry sets are specifically designed to deal with functional polymorphism Functional polymorphism refers to a function which can operate on arguments with a variety of types In Figure 4 1 *calculateArea( )* can operate on both integer and float number types Entry sets summarize the different calling environments of methods in a program This environment is impacted by the type of arguments passed to a method and the type of the object which is dispatching the method

```
Square mySquare = new Square( ),
mySquare calculateArea(3, 9),              (a)
mySquare calculateArea(3 4, 5 7),          (b)
```

**Figure 4 1  Polymorphic Function**

Creation sets are specifically designed to deal with data polymorphism This is the ability of a variable to hold a number of objects of different concrete types It also includes container polymorphism which is the ability of an object to hold other objects of different concrete types Creation sets summarize the different types of run-time objects created at particular creation points within a program A creation point is the program statement and execution environment where an object was created Type variables are labelled with entry and creation sets where appropriate

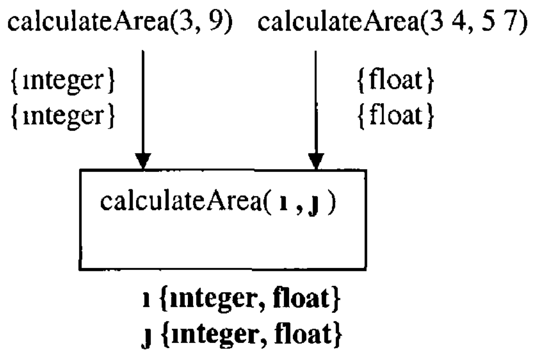


**Figure 4 2  Entry set for the method calculateArea( )**

Entry and creation sets provide the invaluable concrete type information needed by the declassification optimization technique Figure 4 2 illustrates the entry set for the method *calculateArea( )* The entry set has two different concrete types {integer and float} because the function is passed two different parameters

```
(new filledSquare(1) ) getA( ),          (a)
(new filledSquare(1 1) ) getA( ),        (b)
```

**Figure 4 3  Polymorphic Container**

Figure 4 3 illustrates an example of where creation sets are necessary A creation set is created at the creation points (a) and (b) to specify the type of objects which should be instantiated as a result of the call to the method *filledSquare( )* This type inference algorithm is extended to include splitting Splitting is used to increase the precision Entry and creation sets can be split when the algorithm locates an area of imprecision in the program Each split introduces more type variables, which has the potential of eliminating imprecisions from the inferred type Deciding where the best place to split a set, is a very important decision The precise and concrete type information needed by the declassification technique is available in the entry and creation sets and splitting is not necessary Splitting would be necessary if you were implementing a cloning optimization technique Cloning is a technique which makes new copies of a method to enable the replacement of a dynamic dispatch with a statically bound method call

A salient advantage of the algorithm in [Plevyak & Chien, 1994] is its ability to handle deeply nested polymorphic methods and data structures and recursive versions of each Previous type inference algorithms such as [Bruce *et al*, 1993], [Mitchell *et al*, 1993], [Milner *et al*, 1990] and [Oxhøj *et al*, 1992] had difficulty practically typing such structures The constraint-based type inference algorithm presented in [Oxhøj *et al*, 1992] is safe The algorithm in [Plevyak & Chien, 1994] is also safe because it uses a constraint-based network It refines the analysis by splitting and summarizing type variables within the constraints of the network but it does not change the values of the constraint network It can ensure that the analysis will terminate because there are only a finite number of iterations performed A recursive program will also terminate because a limit is put on the amount of analysis carried out It is vital that the algorithm is guaranteed to terminate because it ensures that it will never fall into an infinite loop when analyzing programs for optimization

The empirical study which was conducted in [Plevyak & Chien, 1994] proved that this algorithm provides more precise and detailed type information than the previous algorithms mentioned This incremental type inference algorithm does so efficiently without unnecessarily splitting type variables

## 4.2 Ways in which this information can be used

The availability of concrete type, data and control flow information will provide essential information to facilitate certain safety checks to be carried out during compilation For example, it makes it possible for the compiler to carry out type checking which will detect errors in the program code The programmer could then be informed of these possible problems Another reason and one which could be just as important as the safety guarantee, is that concrete type information can be used for program optimization For example, the results of the type inference could ensure that a number of type-checks are unnecessary and can therefore be removed This should improve the run-time performance of the program

Method cloning is another optimization technique in [Plevyak, 1996] which can make valuable use of the concrete type information available on the methods defined and used in a program Cloning involves making new copies of a method for different invocation contexts It would examine, for example, the entry set information available in [Plevyak & Chien, 1994] and clone methods to eliminate dynamic dispatches in a program A test was carried out to estimate the numbers of clones necessary to eliminate dynamically dispatched messages in [Plevyak & Chien, 1994] It was estimated that creating between 1 5 and 2 5 as many methods, would eliminate the majority of dynamic dispatches Other optimization techniques are also facilitated such as global constant propagation and dead code elimination

## 4.3 Analysis for the Declassification technique

The central aim of the declassification technique is to identify suitable classes for inlining A suitable class is a class which is used *exactly once* within the program The fields and methods of each inlinable class are inlined within its enclosing class The declaration of the inlined class can now be removed from the source code Consequently, this technique will change the hierarchical structure of the program by eliminating these inlinable classes The enclosing class will now create an instance of the inlinable classes superclass

It is essential for the correct operation of the declassification technique that exact and definite information is available on the usages of the different classes within the program This requires a very sophisticated type inference algorithm as Java programs like other object-oriented programs are difficult to reason about This is because of object-oriented properties like inheritance and dynamically bound methods that make the analysis process very complex The context-sensitive flow analysis algorithm in [Plevyak & Chien, 1994] provides the necessary type information to allow us to apply the declassification technique This algorithm is efficient as it allocates additional effort incrementally to the areas of the program where imprecise results are obtained The algorithm was written to provide concrete type information for programs written in the Concurrent Aggregates language Although it has been written for a specific language, the algorithm is general enough that it could be easily extended to facilitate other object-oriented languages

The algorithm has not been extended to facilitate the Java language, as a result, the analysis of a Java program by Plevyak & Chien's algorithm is applied by hand The algorithm determines for each variable the set of classes to which it may be instantiated at run-time It provides information on different types of variables, these include fields of objects, local variables, method parameters and return types This information is then used to determine how many possible usages of each class there may be within a program

### 4.3.1 Analysis Process

The objective of this optimization is to reduce the number of classes which are instantiated and used in a program during its execution This optimization technique is divided into two parts, analysis and transformation The job the analysis has to fulfill in order to achieve this objective is to identify classes which are used exactly once within the program Each such class is a suitable candidate for declassification By declassification we mean inlining the fields and methods of the class into its enclosing class It should be noted that the enclosing class is the only class to use the inlinable class The enclosing class could *instantiate* this class one or more times within the program even though it uses it exactly once It should be remembered however, that each instance of the enclosing class is associated with only one instance of the inlinable class at any point in the program execution This situation is illustrated in Figure 4 13

The declaration of the inlined class can then be removed from the program as it is no longer needed By inlining the class in this way we are eliminating the need to create and maintain these 'intermediate classes' and instead we are extending the size of their enclosing class The analysis examines all top-level classes in the program to establish their suitability for inlining

To aid the exposition of this technique, consider the example below

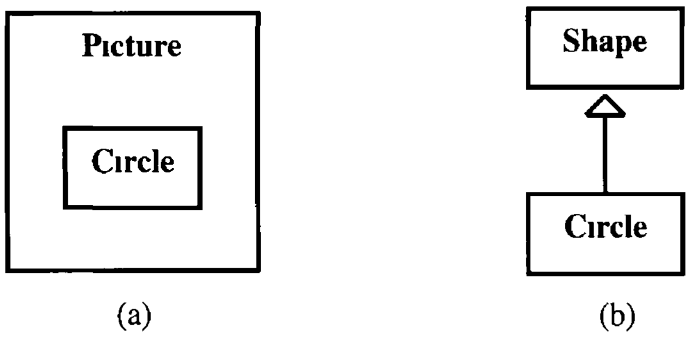


**Figure 4 4 The Picture and Circle classes**

The *Picture* class has a field which stores an instance of the *Circle* class The *Circle* class extends the superclass *Shape* This is illustrated in Figure 4 4 (a) and (b) The *Circle* class is a potential candidate for inlining, if it is established by the analysis algorithm that this class is used once in the program The *Picture* class is the enclosing class which uses it The *Circle* classes fields and methods are inlined within the enclosing class *Picture* The class declaration of the *Circle* class is then eliminated from the source program, as it is no longer needed An instance of the *Circle* classes superclass is created in the enclosing class

Figure 4 5 illustrates the example of the *Picture* and *Circle* classes in a simple Java program There are four classes in the program, *Picture*, *Circle*, *Square* and *Shape* The *Circle* and *Square* classes are subclasses of the *Shape* class Each class has its own associated fields and methods Again the *Circle* is used exactly once by the *Picture* class and is a suitable class for inlining The *Square* class is not a suitable class because it is used twice by the *Picture* class The fields *radius* and *area* will be added to the *Picture* class The methods *getArea( )* and *printDetails( )* will also be added to the *Picture* class An instance of the superclass *Shape* will be created and stored in the variable *myCircle*.

```
class Picture
{
      private double area,
      private Circle myCircle  = new Circle( ),
      private Square redSquare = new Square( ),
      private Square blueSquare = new Square( ),

      void initialPictureCircle( )
      {
            myCircle radius = 2,
            myCircle colour = "Brown",
      }

      double getArea( )
      {
            return area,
      }
}

class Shape
{
      private float XCoOrdinate,
      private float YCoOrdinate,
      String  colour,
}

class Circle extends Shape
{
      float radius = 1 0,
      private double area = 0,

      double getArea( )
      {
            if (area == 0)
                  area = Math PI * (radius * radius),
            return area,
      }

      void printDetails( )
      {
            System out println("Radius of circle is  " + radius),
            System out println("Colour of circle is  " + colour),
      }
}

class Square extends Shape
{
      private float width = 1 0,
      private double area = 0,

      double getArea( )
      {
            if (area == 0)
                  area = width * width,
            return area,
      }
}
```

**Figure 4 5  Program code**

### 4.3.2 Intermediate Representation

Following the analysis of the Java program by applying the algorithm in [Plevyak & Chien, 1994] by hand, precise and concrete type information will be available It will determine for each variable the set of classes to which it may be instantiated Type data can be extracted from these sets and combined with our analysis to identify suitable classes for declassification Our analysis algorithm is used to gather information on the class structure within a Java program It is necessary to gather detailed and exact information on the different aspects of a class declaration Some of the details which are gathered include, class name, super classes, interfaces used, fields and methods, and, member classes A Java parser [JavaCC, 1999] has been augmented with Java statements which gather this information on classes, as the program is parsed

The central objective of the declassification technique is to identify classes which are used once within the program The type information in the sets created by the algorithm in [Plevyak & Chien, 1994] is interrogated and is supplemented with information gathered on the layout and structure of the classes in the Java program The combined information is interpreted and a count is made of the number of times each class is used in the program The pseudocode for the analysis algorithm is illustrated in Figure 4 6, it specifies the criteria by which usage counts are calculated for all top-level classes A class with a count of one has been determined by the analysis as having only one use within the program This class is deemed an 'intermediate class' and should be removed from the program The main program class is not a potential class for inlining and will not be chosen by the analysis algorithm

```
Initialize all usage counts to zero

Start with the main program class
Count
Begin
    Add 1 to the usage count of current class
    If current class not marked
    Begin
        Mark current class
        For each class in the set determined for each
            field perform count
        For each class in the set determined for each
            local variable perform count
        For each class in the set determined for each
            method parameter perform count
        For each class in the set determined for each
            method return type perform count
        For each class in the set determined for each
            anonymous object perform count
        For each superclass perform count
    End
End
```

**Figure 4 6 The pseudocode for the analysis algorithm**

This analysis algorithm is O(n) for a program of size n The length of the analysis algorithm can therefore be simply calculated by the size of the program and the degree of program complexity does not have to be taken into account It is of paramount importance that the declassification technique is safe Its safety is assured by the fact that a class is only inlined within another class if there is only one use of that class in the program This guarantees that there are no alias relationships with the inlined class in the program, as such an alias would be considered a use of the class Inlining is greatly complicated by alias relationships and it is more difficult to guarantee that such a program transformation will not result in invalid results Consequently, the declassification technique does not have to deal with aliases and its analysis and transformation is significantly simplified The analysis algorithm could therefore be considered to be simple and straightforward as there are no complex calculations required to deal with alias relationships The issue of semantic relationships and aliases are discussed in the next section

## 4.3.3 Sharing Semantics

It is essential that the declassification technique's analysis and transformation algorithm does not change the sharing semantics of the program to such an extent that the optimized program is not equivalent to the unoptimized program It is vital that the sharing semantics of the inlinable class are changed to use the enclosing classes new inlined state correctly

A class *e* is suitable for inlining because it has exactly one use within the program Its enclosing class *r* has declared this use, which is the use of the class in a field *f* This field *f* is used to store an instance of the class *e*

It is essential to prove that the analysis algorithm upholds the following statement

> The only operation of storing a reference to the inlinable object *e* is when it is stored in its enclosing object *r f*, that is, *r f* = *e*

The statement is true because

There are no other variables in the program which can store a reference to the object *e* A variable could be a local variable, class field, method parameter or return type This is correct because the analysis algorithm checks and counts these variables in all the classes within the program

```
class Picture
{
      private Circle blueCircle  = new Circle( ),       {Circle}
      private Square redSquare = new Square( ),         {Square}
      private Square yellowSquare = new Square( ),      {Square}
}

class Circle extends Shape                              {Shape}
{

}

class Square extends Shape                              {Shape}
{
        .
}
```

**Figure 4 7 Program code**

The algorithm in [Plevyak & Chien, 1994] determines for each variable the set of classes to which it may be instantiated at run-time The analysis algorithm interrogates the type information available in the sets and combines it with information on the class structure of the program From these sources it is able to determine the usage counts of each class within the program Figure 4 7 illustrates the type information in sets which would be available after running Plevyak and Chien's algorithm The *Circle* and *Square* classes are subclasses of the *Shape* class This is the case for all the other examples given in section 4 3 3 It is determined that the variable *blueCircle* is instantiated to a *Circle* object at run-time A usage count of two is calculated for the *Shape* class because it is the superclass for both the *Circle* and *Square* classes There are no other uses made of this class in the program shown in Figure 4 7 The class *Circle* is used exactly once by the *Picture* class and the *Square* is used twice

| Class name | Shape | Picture | Circle | Square |
|---|---|---|---|---|
| Usage count | 2 | 1 | 1 | 2 |

**Figure 4.8** **Usage Counts**

The usage count of the *Circle* class is one because there is a field in the *Picture* class of this type The usage count of the *Square* is two because there are two fields of type *Square* created within the *Picture* class The usage count for the program in Figure 4 7 is illustrated in Figure 4 8 The *Picture* class is not considered a suitable class for inlining even though it has a usage count of one because it is the main program class Following the analysis we are definite that there is just one reference to the inlinable class *Circle* within the program We are therefore assured that the field *blueCircle* of the enclosing class *Picture* is the only reference to the instantiation of the class *Circle* and that there are no aliases to this class object

The semantics of the Java language complicate the analysis which calculates the usage counts of classes Consider the following situations

1 A variable is declared which is the same type as the super class of a class *e* An instance of the class *e* is stored in the variable by casting to its superclass type It is essential that this variable is counted as an instance of the class *e* An example of this situation is illustrated in Figure 4 9 An instance of the class *Circle* is created and stored in a variable *blueCircle* This instance is then cast to its superclass instance and stored in a variable *aShape*

```
class Picture
{
      private Circle blueCircle  = new Circle( ),      {Circle}
      private Shape aShape = blueCircle,               {Circle}

      .
}
```

**Figure 4.9** **Program code**

The type inference algorithm in [Plevyak & Chien, 1994] is able to deal with this situation It will infer that the variable *aShape* in Figure 4 9 is used to reference a *Circle* object The usage count of the *Circle* class will therefore be two because there are two usages of the class *Circle*, *blueCircle* and *aShape* The *aShape* variable is an alias to the *Circle* object and no aliases are allowed if transformation is to take place All references to the *Circle* object through its superclass are counted

2 Interfaces complicate the identification of the usage counts of inlinable classes If an inlinable class implements an interface then variables could be declared of the interface type to store an instance of that class An example is illustrated in Figure 4 10 The *Circle* and *Square* classes implement the interface *drawable* A variable classicDraw is declared of type *drawable* and an instance of the class *Circle* is stored there An instance of *Square* is also stored in the interface variable impressDraw because this class implements this interface also The type inference algorithm in [Plevyak & Chien, 1994] is able to deal with interfaces To adequately deal with them requires a powerful algorithm because other classes in the program could implement the same interface

```
class Picture
{
      private drawable classicDraw = new Circle( ),     {Circle}
      private drawable impressDraw = new Square( ),     {Square}
}

class Circle implements drawable
{
        .
}

class Square implements drawable
{
        .
}
```

**Figure 4 10** **Program code**

The type inference algorithm infers the type of classes which are stored in the interface variables and the sets are filled accordingly The algorithm determines that there is one usage count for the *Circle* and *Square* classes By using this algorithm it simplifies the process of dealing with interfaces

3 Method parameters are counted as a use of the inlinable class This is because it would complicate the declassification technique and a number of limitations would have to be placed on the inlining of classes which are passed as a parameter The program in Figure 4 11 would have a usage count of one for the inlinable class *Circle* if you did not count parameters Transforming such a program could result in a compile-time error if there was another method in the enclosing class *Picture* or one of its superclasses with the same name, number and type of parameters as the transformed method *draw(Shape)* This is illustrated in the transformed program in Figure 4 11 Return types are counted for similar reasons as parameters An example of a limitation that would have to be placed on this transformation process is that a method with polymorphic return types could not be inlined

```
class Picture
{
    private Circle myCircle,

    void draw(Circle aCir)                    void draw(Shape aShape)
    {                                         {
        float temp = aCir radius,
        String aCol    = aCir colour,         }
          :
    }
}

Transforms to:
class Picture
{
    private Shape myCircle,

    void draw(Shape aCir)                     void draw(Shape aShape)
    {                                         {
        float temp = radius,
        String aCol = aCir colour,            }

    }
}
```

**Figure 4.11 Program code before and after transformation**

4 Other situations which complicate the process of counting the number of possible usages are collection classes and arrays A collection class stores a group of objects according to a particular retrieval scheme A hashtable and vector class are an example A hashtable is like a dictionary because it stores and retrieves elements with key values Hashtables operate on elements of type object A vector is a dynamic array which can store different kinds of objects It is as a result, very difficult to determine what the exact type of the object is, when it is being stored or retrieved from these collection classes Arrays can also be difficult to analyse Take for example, an array of type object which can store different types of objects An instance of the object *Circle* and *Shape* could be cast to their superclass object and stored in this array It is stated in [Plevyak & Chien, 1994] that the type inference algorithm is unable to deal with programs which store a variety of types in a single array Consequently, the declassification analysis algorithm also has this limitation and is unable to deal with arrays and collection classes which are used in this way

Consequently, the statement $r f = e$ is only operation storing a reference to the inlinable object $e$ in the program if the analysis algorithm determines that the usage count of a class $e$ is one This ensures that the field $f$ of the class $r$ is the only variable which can store a reference to the inlinable class $e$ There are as a result, no alias relationships to the class $e$ Without this confirmation, there could be a number of alias relationships to the object $e$, which could distort the sharing semantics of the program Alias relationships would greatly complicate the declassification process It would be difficult to inline an object $e$ into the enclosing class $r$ if another variable $g$ had a reference to this object We would have to change the sharing semantics to inline the class $e$ and the variable $g$ would also have to be changed Any alias to this variable $g$ would also have to be tracked and changed In this way a chain of aliases could be created and it is very complicated to infer by type inference what aliases there are and how they should be changed

```
class Picture
{

     private Circle blueCircle  =  new Circle( ),         {Circle}
     private Circle redCircle    =  blueCircle,           {Circle}
     private Shape aShape        =  blueCircle,           {Circle}

}
```

**Figure 4 12 Program code**

Figure 4 12 illustrates an example of a program where there are a number of alias relationships to the object *blueCircle* Inlining the *Circle* class would be very difficult as the sharing semantics of these aliases would have to be changed The process of type inference is greatly simplified if it is established that there is only one reference to the inlinable class object Inlining these classes into their enclosing class will require the sharing semantics to be changed but it will not distort them with the danger of leaving the program in an incorrect state

[Dolby & Chien, 2000] presents a formal model of object inlining which has the ability to identify *one-to-one* fields This model proves that the process of inlining a one-to-one field is semantics preserving This ensures that the inlinmg is safe and that the program is correct following the transformation A one-to-one field consists of a container and inlinable object (r,e) Two conditions must be adhered to in order to qualify for one-to-one field status

a The only operation of storing a reference of the object *e* in the program, is *r f=e*,

b The only operation of storing a reference where *r f* stands on the left-hand side is *r f = e*

The first condition has already been discussed in relation to the declassification technique It verified that field *f* of an enclosing object *r* is the only variable to store a reference of the object *e*, if *e* has been chosen for declassification The second condition specifies that the only operation of storing a reference, where the enclosing object *r f* stands on the left hand side, is *r f = e* This condition ensures that no other child object is stored in the field *f* of the enclosing object *r* This condition is *not* enforced because the declassification technique permits the field of a container object to be instantiated to one or more child objects This is illustrated in Figure 4 13

```
class Picture
{
      private Circle aCircle,

      encloseMethod( )
      {
            aCircle = new Circle( ),
            double firstArea = aCircle getArea( ),

            aCircle = new Circle( "Yellow",4 5),
            double secondArea = aCircle getArea( ),

      }


}

class Circle extends Shape
{
      float radius = 1 0,
      private double area = 0,

      {
            setXCoOrdinate(9 2),
            setYCoOrdinate(15 7),
      }



      Circle ( String aColour, float aRadius)
      {
            super(aColour),
            radius = aRadius,
      }

}
```

**Figure 4 13 Program code**

An instance of the *Circle* class is created and stored in the *aCircle* field Another instance of the inlinable class *Circle* is created and stored in this field at a later state in the program Two different instances are consecutively stored in the same field of the enclosing class There is, however, only *one* use of the *Circle* class through the *aCircle* reference The transformation will result in the fields and methods of the *Circle* class being inlined within the *Picture* class A declaration of the superclass *Shape* is declared in place of the *Circle* class

The sharing semantics of the program are not distorted following the transformation phase This is because the field *aCircle* is the only variable which stores a reference to the *Circle* object This condition is still true even though the field *aCircle* stores two instances of the class in succession Inlining this class will change the semantics of the program to reference the enclosing classes new inlined state but the optimized program is still equivalent to the unoptimized program The formal model in [Dolby & Chien, 2000] as a result, cannot be applied to the declassification technique to prove the correctness of its transformation The declassification technique can, therefore, transform a program which has consecutive instances of the same class stored in the same variable There is a limitation on this transformation, the variable which is used to store the instance of the inlinable class cannot be used to store an instance of any other class To inline the inlinable class in this situation could result in the distortion of the program The results of the algorithm in [Plevyak & Chien, 1994] can be used to establish this

## 4.4 Summary

In this chapter we have outlined the analysis phase of the declassification technique The algorithm in [Plevyak & Chien, 1994] provides us with concrete and precise type information on programs with many levels of polymorphism in functions and data structures The way the analysis algorithm uses this type information to identify suitable classes for declassification and how the sharing semantics are affected by this optimization technique are discussed A number of sample programs are given in order to demonstrate how the analysis algorithm gathers the information it requires, to identify suitable classes for inlining Chapter 5 discusses how the source program is transformed to inline these intermediate classes into their enclosing class

# Chapter 5 Transformation

The analysis algorithm will identify suitable classes for inlining The transformation algorithm will make the necessary changes to transform the source code to eliminate these '*inlinable*' classes In discussing the transformation, the following terms are used, the inlined class variable is the field which is used to store the instance of the inlinable class within the enclosing class The inlined state constitutes the fields and methods being added to the enclosing class

The transformation involves inlining the fields and methods of each inlinable class within the enclosing class that has declared the instance The declaration of the inlined class instance is removed and replaced with a declaration of a variable of the inlinable classes superclass type The layout of the enclosing class is restructured in this way and further changes are necessary to ensure that all uses of the inlined fields and methods are redirected to the enclosing classes' new inlined state The transformed source code could then be further transformed by other optimization techniques There are a wide variety of different optimization techniques suitable for object-oriented programs, some of these are outlined in chapter 3

## 5.1 Transformation Algorithm

Changes are necessary to the fields and methods of the inlinable and enclosing classes This is because the sharing semantics between the two classes has now changed

### 5.1.1 Fields

It may be necessary to change a field's name in the inlinable class if there is a name clash A name clash occurs when an inlinable field has the same name as one of the following

1 One of the fields in the enclosing class,
2 One of the fields in any of the superclasses of the enclosing class,

3 One of the local variables in an enclosing class method,

4 One of the field constants of any of the interfaces the enclosing class or any of its superclasses implement

A compiler error would occur if an inlined field clashes with a field in the enclosing class The incorrect execution of the program could result if it clashes with a field in the enclosing classes' superclass Another error could occur if the inlined field clashed with one of the local variables in an enclosing class method This is because local variables take precedence over fields A reference to the inlinable classes field could after transformation reference a local variable if their names clashed Finally, a clash with a field constant of one of the enclosing classes' interfaces would result in the inlined field taking precedent over it This would cause the incorrect execution of the program The resolution of the clash is achieved by concatenating each clashing inlinable field name with a randomly generated number An inlinable field name that does not clash is left unchanged

The changed and the unchanged fields of the inlinable class are then inlined within the enclosing class A number of changes are necessary to both the inlinable and enclosing class to account for the new inlined fields If a field name has changed it is essential that changes are made to the field initializers and methods of the inlinable class If a field name has not changed, no changes are necessary to the inlinable class to account for that particular field

Changes are necessary to the enclosing class to reference the new fields in its inlined state, even if the field names remain the same The instantiation of the inlined class is removed and a declaration of a variable of its superclass type is created in its place Any references to fields belonging to this superclass can remain unchanged Any references to fields not belonging to the superclass must be changed to reference the enclosing classes' new inlined state instead The position in the enclosing class where the inlined fields are placed is important They must be inlined immediately after the declaration of the inlined class variable This is important for two reasons Firstly, a field(s) of the inlinable class could be used to initialize a field(s) of the enclosing class It is vital, therefore, that this inlinable field(s) is declared before the initialization takes place Secondly, a field(s) of the enclosing class could be used to initialize an inlinable classes field(s) and if so, it must be declared before the initialization A compile-time error could occur if the inlinable classes fields are not inlined in the correct position The position where the methods are placed is not important

### 5.1.2 Methods

Similarly, it is necessary to change the name of a method in the inlinable class if there is a name clash A name clash occurs when an inlinable method has the same name and number of parameters as one of the following

1 One of the methods in the enclosing class,
2 One of the methods in any of the enclosing classes' superclasses

The ability of passing object references to methods complicates the task of identifying when two methods clash A method parameter could be used to store references to classes that are subclasses of it For example, the classes *Circle* and *Square* are subclasses of the class *Shape* A reference to *Circle* or *Square* could be passed to a method with a parameter of type *Shape* Figure 5 1 (a) illustrates two methods which clash Figure 5 1 (b) illustrates two methods which do not clash because the number of parameters differ

```
void draw(Object aShape),

void draw(Shape aShape),
```

(a) A clash occurs

```
void draw(Object aShape),

void draw(Object aShape, Shape aShape),
```

(b) A clash does not occur

**Figure 5.1 Method Clashes**

To inline a method which has the same name and number of parameters could result in the incorrect execution of the program Any clash that is identified is resolved by changing the inlinable classes' method name This is done by concatenating each method name with a randomly generated number The references in the inlinable class must be changed to reference any new method names No changes are necessary to the inlinable class if the method name remains the same The enclosing class must be again changed to reference its new inlined state

It is necessary to check if the new field or method name clashes The new name is again compared with the enclosing class under the criteria specified above for name clashes If a clash occurs a new randomly generated number is concatenated with the field or method name This new name is again checked A conservative approach has been taken to deal with the problem of field and method clashes No effort, for example, is made to determine if the inlinable classes field that clashes with a field in the superclass will result in a reference to the field of the superclass being wrongly directed to the new field Approaching the problem in this way reduces the amount of analysis required to transform the program

### 5.1.3 Transformation Example

Figure 5 2 illustrates the example given in Figure 4 4 after transformation The *Circle* class was identified as a suitable class for inlining The fields and methods of this class are inlined within the *Picture* class Figure 5 2 (a) illustrates the aggregate association between the *Picture* and *Shape* classes, following the transformation The *Circle* class has been eliminated and the inheritance structure of the *Shape* class is illustrated in Figure 5 2 (b)

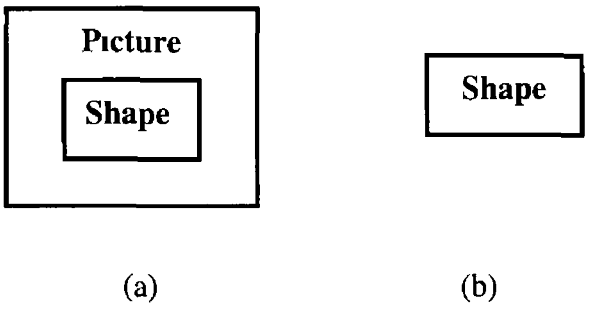


**Figure 5.2 The Picture and Shape classes**

The *Picture* class now creates an instance of the *Circle* classes' superclass *Shape* Figure 5 3 illustrates the example program given in Figure 4 5 following the transformation

```
class Picture
{

      private double area;
      private Shape myCircle;

      {
            circle35223( );
      }

      float radius;
      private double area45143;
      private Square redSquare   = new Square( );
      private Square blueSquare = new Square( );

      void initialPictureCircle( )
      {
            radius = 2;
            myCircle.colour = "Brown";
      }

      double getArea( )
      {
            return area;
      }

      double getArea67454( )
      {
            if (area45143 == 0)
                  area45143 = Math.PI * (radius * radius);
            return area45143;
      }

      void printDetails( )
      {
            System.out.println("Radius of circle is  " + radius);
            System.out.println("Colour of circle is  " +
                                                myCircle.colour);
      }

      circle35223( )
      {
            myCircle = new Shape( );
            radius = 1.0;
            area45143=0;
      }

}

class Shape
{
      private float XCoOrdinate;
      private float YCoOrdinate;
      String  colour;
}
```

```
class Square extends Shape
{
      private float width = 1 0,
      private double area = 0,

      double getArea( )
      {
            if (area == 0)
                  area = width * width,
            return area,
      }

}
```

**Figure 5 3 Transformed program code**

### 5.1.4 Pseudocode for the transformation algorithm

This is illustrated in Figure 5 4 The technique assumes there is a clash of field and method names between the inlinable and enclosing classes The algorithm would be simpler if this was not the case

```
For each class with a usage count equal to 1
Begin

  For each field of the inlinable class which clashes                       Step A
    Concatenate field name with a random integer number

  For each method of the inlinable class which clashes
    Concatenate method name with a random integer number

  Add inlinable class fields to enclosing class                             Step B
  Add inlinable class methods to enclosing class

  Change inlined class variable to be of the superclass type                Step C
  Add call to inlinable classes' changed constructor method

  For each field initializer and method in enclosing class                  Step D
        Perform ReferencesChangesInEnclosingClass

  For each field initializer and method in inlinable class                  Step E
        Perform ReferencesChangesInInlinableClass
    Perform ChangeConstructorMethods                                        Step F

    Combine enclosing class finalize method with inlinable class finalize method    StepG

    Append inlinable interfaces to enclosing class interfaces               StepH

End

ReferencesChangesInEnclosingClass
Begin
    Change references to inlinable class fields and methods to reference the new inlined fields and methods
    References to the inlinable class superclass fields and methods should remain unchanged
    Change reference to inlinable superclass object to inlined class variable
End

ReferencesChangesInInlinableClass
Begin
    Change references to inlinable class fields and methods to reference the new names of the fields and
    methods
    Change references to the inlinable class superclass fields and methods by referencing the inlined class
    variable
    Change reference to inlinable superclass object to inlined class variable
End

ChangeConstructorMethods
Begin
    For each inlinable constructor method
    Begin
        Concatenate method name with a random interger number
        Instantiate the inlinable classes' superclass
        For each inlinable instance field
            Add field body to constructor method body
        For each inlinable instance block initializer
            Add block initializer body to constructor method body
    End
End
```

**Figure 5 4 The pseudocode for the transformation technique**

A detailed explanation of the pseudocode is given by explaining how the sample program in Figure 5 3 is transformed step by step

Step A There is a name clash between the field *area* in the *Circle* class and the field *area* in the enclosing class The field name is changed by concatenating it with a randomly generated number For example double *area45143* All the other inlinable field names remain unchanged

Step B There is also a name clash between the method name *getArea( )* in the *Circle* class and the method *getArea( )* in the enclosing class The method name is changed by concatenating it with a randomly generated number For example *getArea67454( )* All the other inlinable method names remain unchanged The inlinable classes' fields and methods are added to the enclosing class Some of the inlinable classes' methods are not added directly to the enclosing class but are combined with the enclosing classes' methods Examples of these are the constructor, finalize and instance block initializer methods

Step C In Figure 4 5 the inlined class variable is of type *Circle* and an instance of this class is created The transformation process changes the declaration type of the variable to its superclass *Shape* This is done to enable the enclosing class to access the fields and methods of the *Circle* classes' superclass The instantiation of the inlinable class is removed A call is placed to the inlinable classes' changed constructor method This method instantiates the superclass and initializes the inlinable classes' fields as required

Step D The enclosing class *Picture* must be changed to reference its new inlined state The changes are made to the enclosing classes' field initializers and methods References to the inlinable class *Circle* are changed to reference the inlined field and method names directly No changes are necessary to any references to the fields and methods of the superclass *Shape* Any reference to the superclass object must be changed to reference the inlined class variable *myCircle*, within the enclosing classes' inlined state

Step E Changes must also be made to the inlinable class to account for the changes in the names of the fields and methods Local variables in methods complicate the process of identifying field references Any reference to the inlinable classes' superclass object or its fields and methods must be done by referencing the superclass object stored in *myCircle*

Step F Each of the inlinable classes' constructor methods are changed to create an instance of the superclass and store it in the inlined class variable *myCircle* Each field body of the inlinable classes' instance fields are then added If a call to an instance block initializer follows, its method body is added Static field bodies or static block initializers are not added to the constructor method These additions are added to the start of each constructor method

Step G The finalize method of the inlinable class must be combined with that of the enclosing class

Step H The declassification technique inlines the methods of the inlinable class within the enclosing class Any interfaces implemented by the inlinable class are now implemented by the enclosing class The class declaration of the enclosing class must be changed by adding these interfaces to it, if it does not already implement them

## 5.2 Another example of program transformation

The program illustrated in Figure 4 5 is changed in Figure 4 13 It shows how an inlinable class can be instantiated one or more times in succession, yet the usage count of the *Circle* class is one This class is a suitable class for inlining and the transformed program is illustrated in Figure 5 5 Arguments to constructor methods of the inlinable class are handled by the transformation algorithm An argument could be passed to the constructor method in order to initialize the inlinable classes superclass or it could be used to initialize the field(s) of the inlinable class

An explanation of the steps taken to transform this program is given below

Step A The data type of the *aCircle* inlined class variable is changed to declare a field of type *Shape*

Step C A new method *Circle34525( )* was created because the Circle class does not contain a default constructor method The *Circle34525( )* method creates an instance of the *Circle* classes' superclass *Shape* and stores it in the field *aCircle* It also initializes the *Circle* classes' fields as would be done if an instance of the *Circle* class was created  This is accomplished by adding the *radius* and *area* field initializations  An instance block initializer method follows the declaration of the *area* field  Its body is inlined within the constructor method  It is placed after the *radius* and *area* field initializations

Step D The inlinable classes constructor method illustrates how the transformation algorithm handles arguments  Two arguments are passed to this constructor  The first argument *aColour* is used to instantiate the superclass  The inlinable classes field initializations and the body of the instance block initializer method are added to the constructor method as described in step C  The body of the constructor method follows  It contains a statement which assigns the inlinable classes field *radius* to the value of its second argument *aRadius*  The instantiation of the *Circle* class is transformed by making a call to this changed constructor method in step B

```
class Picture
{

    private Shape aCircle,                                   Step A
    float radius,
    private double area45143,



    encloseMethod ( )
    {
        circle34525( ),
        double firstArea = getArea67454( ),

        circle34525( "Yellow" 4 5)                           Step B
        double secondArea = getArea67454( ),
    }



    circle34525( )
    {                                                        Step C
        aCircle = new Shape( ),
        radius = 1 0,
        area45143 = 0,
        aCircle setXCoOrdinate(9 2),
        aCircle setYCoOrdinate(15 7),
    }
```

```
        circle34525(String aColour, float aRadius )             Step D
        {
            aCircle = new Shape(aColour );
            radius = 1.0;
            area45143 = 0;
            aCircle.setXCoOrdinate(9.2);
            aCircle.setYCoOrdinate(15.7);
            radius = aRadius;
        }

}
```

**Figure 5.5 Transformed program code**

## 5.3 Restrictions

It was necessary to place four restrictions on class inlining and they are as follows, method overriding, abstract parent, self inlining and reflection. It would be unsafe to allow class inlining in the above circumstances.

### 1 Method Overridding

The inlinable class could have one or more overridden methods. Dynamic binding occurs at run-time to establish which method is being called, whether it is the inlinable classes' or the superclasses' method. It is necessary, however, to establish which method is being called during the transformation process. This is because the sharing semantics have now changed. A restriction is therefore placed on class inlining to prevent a class being inlined if it has an overridden method *and* there is a reference to one of the overridden methods in either the enclosing or inlinable class.

A pure polymorphic method could occur in one of the superclasses of the inlinable class, such a method is abstract and as a result cannot be invoked at run-time. Consequently, there is no ambiguity when dynamic binding occurs in establishing if the inlinable classes' or the superclasses' method is being invoked. The case of a pure polymorphic method is not handled by the transformation algorithm and the restriction as listed above is enforced. It should be noted that a pure polymorphic method occurring in the immediate superclass of the inlinable class results in this superclass being abstract. The declassification technique prohibits the inlining of classes if the immediate superclass is abstract. This is explained in point 2 of this section.

**2 Abstract Parent**

To inline an inlinable class which has an immediate abstract superclass would be wrong This abstract superclass cannot be instantiated and the transformation algorithm would attempt to do so

**3. Self Inlining**

It is necessary that the declassification technique places a restriction on the inlining of a class within itself This is because it would have a recursive effect which would cause the transformation to enter an infinite loop This could happen if the program it is optimizing is using some classes within a library Self inlining could occur in a library when a class creates an instance of itself and the program does not use this class

**4. Reflection**

Reflection is not handled by the declassification technique The Class forName( ) method could be used in a program to invoke the inlined class I am assuming that the code being transformed does not use reflection

## 5.4 Visibility Modifiers

Encapsulation is an important aspect of object-oriented design Encapsulation is a technique for hiding data and behaviour within a class It seals the data and internal methods inside the 'capsule' of the class where it can be accessed only by the class itself The ability to use access modifiers allows the programmer to decide what access classes should have to each other Access modifiers can be associated with classes, fields, methods and local variables The declassification technique should not break the privacy of objects and change the visibility of the inlined classes' fields and methods within the enclosing class To weaken this visibility would be a change to how the inlined class is accessed and is considered unsafe Access modifiers are *only* considered in the context of an inlinable class being inlined within an enclosing class within the same package The transformation algorithm does not inline classes from different packages

The only access modifier that can be associated with local variables is 'final' Transforming an inlinable method with this access modifier will not change its visibility

### 5.4.1 Class Access Modifiers

We look at what access modifiers an inlinable class could have and when it is suitable to inline it within an enclosing class

1 An inlinable class with a public access modifier is safe to inline as it is visible to all classes,

2 An inlinable class with a default access modifier is only accessible to other classes within its package The enclosing class is in the same package as the inlinable class,

3 An abstract modifier is not applicable, as an instance of this class cannot be created,

4 The final access modifier can be inlined safely because the optimized code will not be extended Any changes required by the user should be made to the unoptimized program and this modified program can then be optimized

### 5.4.2 Fields and Methods

We look at what access modifiers fields and methods could have and when it is suitable to inline them within an enclosing class

1 Public fields and methods will remain public and they are accessible from any class in any package,

2 The protected modifier allows access by other classes in the package and it also allows special access permissions for subclasses Protected fields and methods are visible to subclasses of the class, even if they are defined in a different package Inlining protected fields and methods does not change the visibility because the enclosing class is in the same package as the inlinable class Theoretically, subclasses of the enclosing class could now have access to these inlined protected fields and methods This will not occur, however, because the optimized code will not be extended,

3 Fields and methods with the default access modifier means that classes within the same package can have access to them It is therefore safe to inline them within the enclosing class,

4 Any pnvate fields and methods ın the ınlınable class wıll remaın pnvate ın the enclosıng class In thıs way, these fields and methods are accessıble only wıthın the enclosıng class,

5 Statıc fields and methods belong to a class, not an ındıvıdual object These are ınlmed wıthın the enclosıng class wıth the statıc keyword modıfier,

6 The final keyword may be applıed to methods and fields Inlınıng the fields and methods wıth thıs modıfier wıll not change the way they are used and accessed,

7 Any class wıth an abstract method ıs automatıcally abstract ıtself and must be declared as such The enclosıng class would not be able to ınstantıate an abstract class Thıs access modıfier ıs therefore not applıcable

Consequently, there ıs no onus on the transformatıon algonthm to check the access modıfiers of eıther the ınlınable class or ıts member fields and methods Inlınıng a class wıth any of the above access modıfiers does not change the vısıbılıty of the fields and ınethods and ıs consıdered safe

## 5.5 Other issues

The ınlınıng of a class wıthın ıts enclosıng class elımınates the need to ınstantıate thıs ınlınable class Thıs transformatıon does not only elımınate *one* ınstance of the ınlınable class, the number of ınstantıatıons elımınated depends on the number of tımes the enclosıng class ıs ınstantıated wıthın the program The more tımes the enclosıng class ıs ınstantıated the greater the benefits wıll be to the declassıficatıon technıque

An ınlınable class can be ınlıned ıf a statıc ınstance of ıt ıs created wıthın the enclosıng class Each of the ınhnable classes fields and methods are added to the enclosıng class wıth a statıc modıfier The ınlınable classes statıc fields and methods are added unchanged The shanng semantıcs between the ınlınable and enclosıng class has not been changed after transformatıon

## 5.6 Comparison of the Declassification technique to the object inlining optimization techniques

The concepts of these object inlining techniques are worth a more in-depth exposition, as they most closely resemble our optimization technique Comparisons will be drawn between them to highlight their weaknesses and strengths I will refer collectively to the following optimization techniques as object inlining techniques [Black *et al* , 1986], [Budimlic & Kennedy, 1997], [Budimlic & Kennedy, 1998], [Dolby, 1997], [Dolby & Chien, 1998], [Dolby & Chien, 2000] and [Laud, 2001], even though some of these papers research other issues

The main objective of the declassification technique and the object inlining optimization techniques is the same, they aim to improve the performance of object-oriented software by reducing the pressure on the memory subsystem The declassification optimization technique differs in the method it uses to accomplish this Its method involves inlining a class within its enclosing class This results in changes being made to the hierarchical structure of the program because the 'intermediate' class is removed and an instance of the inlinable classes' superclass is created The class declarations of the 'intermediate' class is subsequently removed from the program The aim of the object inlining technique is to inline an object within its enclosing object Changes will also have to be made to the hierarchical structure of the program, as some objects will be inlined The class declaration of the inlinable object is not removed from the program For example, [Dolby, 1997] is an automatic technique for inlining objects within container objects Its central aim is to inline as many objects as possible within their container objects

There are similarities between the transformation phase of the declassification technique and the object inlining techniques Both involve inlining the fields and methods of the inlinable class/object into their container class/object Similar changes have to be made to how the inlined attributes are accessed and used

The Emerald object system in [Black *et al* , 1986] has a simple type inference algorithm which is not as sophisticated as the algorithm in [Plevyak & Chien, 1994] The declassification technique combines its own analysis with that of [Plevyak & Chien, 1994] to provide precise concrete type information to enable class inlining to take place

The object inlining technique in [Budimlic & Kennedy, 1997] and in [Budimlic & Kennedy, 1998] differs from the declassification technique because its central aim is to identify and inline objects which are created within a method These are referred to as local objects The declassification technique has the ability to inline top-level classes The Budimlic and Kennedy's technique is considered very limited by [Dolby, 1997] The declassification technique is also limited as it does not have the ability to inline local, anonymous or member classes

Budimlic and Kennedy's technique can be implemented on programs where the complete program is not available at compilation time This eliminates the ability to carry out interprocedural optimization The declassification technique and the technique in [Dolby, 1997] requires that the whole program is available Further work in the future could provide an extension to the declassification technique to facilitate the optimization of incomplete programs One of the solutions presented in [Budimlic & Kennedy, 1998] is as follows a method could be overridden by a method in a new subclass being introduced because it was optimized as part of an incomplete program A solution is presented which involves generating two versions of the code One version contains the code with the method inlined, the second does not Run-time checks are inserted into the execution environment to decide which version to use The declassification technique could be extended in a similar manner A serious disadvantage of this solution is that it increases the size of the program being optimized

Both the declassification technique and Dolby and Chien's automatic object inlining technique [Dolby, 1997], [Dolby & Chien, 1998], [Dolby & Chien, 2000] use Plevyak & Chien's algorithm [Plevyak & Chien, 1994] to obtain precise and concrete type information Dolby and Chien's technique is estimated to cause a 20% increase in the size of a program This issue was highlighted in section 3 2 1 5 Although they state that there is a possible 20% increase in code size, it is believed that there could be a substantially higher increase in the code size Their technique has the ability to inline an object into multiple container objects and they, therefore, cannot guarantee what the maximum increase will be This increase in code size is considered a significant flaw The declassification technique guarantees there is no increase in code size because the declaration of the class is removed when it has been inlined and it is only inlined once This is a very important advantage

The technique in [Dolby, 1997] is very complicated This complexity is necessary because it has to deal with alias relationships It is necessary to investigate intricate alias relationships because an instance of a class could be inlined within the program many times It is not necessary for the declassification technique to deal with alias relationships This is discussed in section 4 3 3 This significantly simplifies both the analysis and transformation parts of the declassification technique There is also no necessity to investigate and implement possible structures for the fused child and container objects This is necessary in Dolby's automatic object inlining technique, as a good structure is critical to reduce the number of methods needed for cloning during transformation [Dolby & Chien, 2000] discusses a revised algorithm

[Dolby & Chien, 2000] has developed a formal model for object inlining This model has the ability to identify one-to-one fields and these fields can be safely inlined which ensures the correctness of the program This formal model cannot be applied to the declassification technique because the fields chosen for inlining are not one-to-one fields We cannot, therefore, use this model to prove that the transformation is safe This issue is discussed in section 4 3 3

[Laud, 2001] extends the research on automatic object inlining in Dolby's papers He investigates the possibility of several objects being inlined within a single field in succession The declassification technique is different as it is inlining classes It has however, the ability to inline a number of instances of the same class into the same field in succession This is handled differently An explanation of how this is carried out is again given in section 4 3 3

## 5.7 Implementation of the Declassification Technique

The declassification technique is written entirely in Java It consists of 21 files and 22 classes Excluding the parser there are approximately 6100 lines of code The declassification technique is made up of two parts, the Analysis and Transformation A Java parser [JavaCC, 1999], which was augmented with extra Java code to gather the necessary information on the class structures within the program, was used by the analysis algorithm to parse the program source code It was necessary to assimilate a considerable amount of information about the classes declared in the program Examples of the type of information required are, the hierarchical structure of each class, the type and name of each field and method, the instance and static block initializer methods, the constructor methods used, the interfaces each class implements and the exceptions thrown by each class

This information was used to calculate the usage counts of all top-level classes The pseudocode for this algorithm is illustrated in Figure 4 6 It is vital that precise type information from Plevyak & Chien's algorithm [Plevyak & Chien, 1994] is used to supplement the declassification's analysis algorithm It is documented in section 4 3 that Plevyak & Chien's algorithm has not yet been written to analyse a Java program As a result, the algorithm is applied by hand and the necessary type information is added manually to increment the usage counts of the classes

The transformation algorithm is fully automatic except for its inability to deal with user interfaces, this is documented below The transformation algorithm inlines classes that have a usage count of one, within their enclosing class The pseudocode for this algorithm is illustrated in Figure 5 4 A number of difficult issues had to be dealt with to accomplish this These include

- Ensuring that there are no field or method name clashes,
- Ensuring the superclass of the inlinable class is referenced correctly after transformation,
- Any references to the inlinable classes interfaces must be transformed to ensure that no clash occurs between an interface field constant of the inlinable and enclosing classes,
- The transformation of the inlinable classes constructor methods must inline any instance field initializations and the body of any instance block initializer methods Static field initializations and static block initializers methods remain unchanged Any arguments to the constructor method and any calls to other constructor methods through *this( )* or *super( )* references must be transformed correctly,
- It is necessary to take local variables into consideration when transforming the body of a method This is because the sharing semantics of the inlined fields have changed and they could now clash with local variables in both the enclosing and inlinable class methods

The declassification technique is a prototype and as such there are a number of limitations that have not been addressed

- It cannot inline classes from different packages, this was documented in section 5 4,
- Information on class interfaces declared in the program must be entered manually,
- References to the inlinable classes static fields and methods through their package names must be transformed manually This issue could not have been addressed until packages had been dealt with,
- The majority of Java statements are being transformed but as this is a prototype there is no guarantee that every Java statement will be transformed correctly

The declassification technique was used to analyse and transform four programs in the SPEC98 [SPEC JVM, 1998] benchmark suite and two other programs These programs are collectively referred to as test programs The results of this empirical study are documented in chapter 6 The externally-observable behaviour of the test programs was examined in detail Each of the unoptimized test programs was run through the SPEC98 benchmark program 10 times They were then optimized by the declassification technique and again they were run through the SPEC98 benchmark program 10 times The benchmark program runs the test program and gathers statistics on it An example of one of the statistics is the amount of time required to execute the test program

The outputs from running the unoptimized and optimized versions of the program were examined and the results established that the externally-observable behaviour is the same For example in the test program Check, a test *testArray* resulted in the output *OK* for both the unoptimized and optimized versions of the program Another example of a type of test carried out is as follows, the declassification test program was optimized and then the optimized version of the declassification program was used to optimize an unoptimized version of the declassification program The results of these optimizations were the same It should be noted, however, that even though thorough testing was carried out, the externally-observable behaviour of the transformed programs is not guaranteed to be exactly the same

The time taken to run the declassification technique is small It takes, for example, on average 30 seconds to analyse and transform the declassification program itself The source code for the declassification technique is available from the department of Computer Applications in Dublin City University

## 5.8 Summary

The transformation algorithm involves inlining the fields and methods of each inlinable class within its enclosing class  This algorithm is presented in this chapter and two example programs are used to help expound it  The visibility modifiers were examined and it was established that they do not restrict the inlining process if the two classes are within the same package  A comparison is made between object inlining optimization techniques and the declassification technique in section 5 6  An empirical study was carried out to establish the success of the declassification technique  The results are documented in chapter 6

# Chapter 6 Evaluation

The declassification technique was evaluated by analyzing and transforming a number of reasonably sized object-oriented programs in the SPEC98 [SPEC JVM, 1998] benchmark suite SPEC98 is one of the most commonly used benchmark suites Two other medium sized Java programs were also evaluated The results of this evaluation are presented in this chapter and from these results the benefits and costs of the technique are assessed The benefits of the declassification technique are the run-time performance gains, reduced memory usage and the fact that there is no increase in code size The cost could be the fact that the performance gain as a result of the optimization is not substantial enough when measured against the additional compile-time cost of carrying out the technique A description of the test programs is given in section 6 1 The results of the analysis and transformation are given in section 6 2 and 6 3 Further possible extensions to the declassification technique are given in section 6 4 and finally, the summary is in section 6 5

## 6.1 Test Programs

The industry standard SPEC98 benchmark suite has been used to conduct this study The source code for four Java programs is available and a brief description of each is given below

**Check benchmark**

This is a simple program to test various features of the JVM to ensure that it provides a suitable environment for Java programs Two tests that it includes are array indexing beyond its bounds and creating a super class and its sub class and then accessing the public, private and protected variables and over-ridden methods

**Compress benchmark**

This benchmark uses the modified Lempel-Ziv method (LZW), which finds common substrings and replaces them with a variable size code This is deterministic and can be done on the fly

**db benchmark**

This benchmark performs multiple database functions on a memory resident database It reads in a 1 MB file which contains records with names, addresses and phone numbers of entities and a 19KB file called scr6 which contains a stream of operations to perform on the records in the file

**Raytrace benchmark**

This is a variant of _205_raytrace, a raytracer that works on a scene depicting a dinosaur, where two threads each render the scene in the input file time-test model, which is 340KB in size

Two other medium sized Java programs are used as test data, the declassification and deforestation programs described below

**Declassification (declass) program**

The declassification source program has been written entirely in Java Part of the source code includes JavaCC (Java Compiler Compiler) [JavaCC, 1999] JavaCC is a Java parser generator

**Deforestation (deforest) program**

The source code for the deforestation algorithm as proposed in [Hamilton, 1995B] and [Hamilton, 1996] This is an algorithm to remove intermediate data structures from programs written in a higher order functional language

## 6.2 Analysis

The programs listed in section 6 1 were analysed to calculate the number of suitable classes for inlining The histogram in Figure 6 1 has two bars The first illustrates the number of top-level classes in each program, the second illustrates the number of classes which are suitable for declassification in each program It is obvious from these results that there are very few top-level classes which meet the criteria for inlining

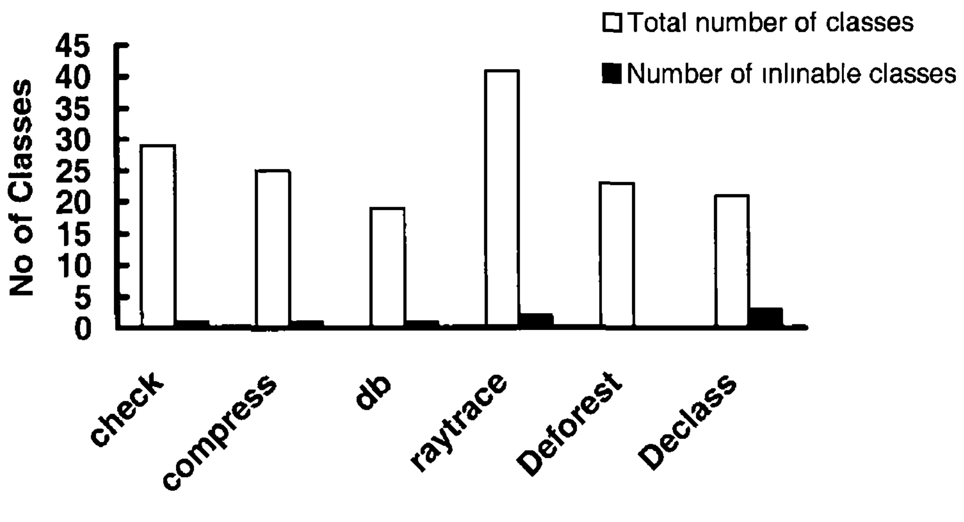


**Figure 6.1 The number of inlinable classes in each program**

Three of the test programs, check, compress and db have only one inlinable class The deforest program has no suitable classes for declassification The percentage of classes suitable for inlining in raytrace and db is 4% The declass program shows the most promising result with 14% of its classes being inlinable

## 6.3 Transformation

In order to establish the effectiveness of the declassification technique, it is necessary to compare the run-time performance and memory consumption of the unoptimized and optimized versions of each test program From these results, we can extrapolate the benefits of the declassification technique

### 6.3.1 Performance

To measure the effectiveness of the declassification technique we transformed the source code of the test programs Each optimized program was then compiled using Java 2 SDK standard edition version 1 4 The unoptimized and optimized versions of each program were measured by the benchmark program available in SPEC98 Measurements were taken on a Pentium 200 with 32 megabytes of RAM and are the average of 10 runs The percentage improvement (or disimprovement) is calculated by dividing the difference between the optimized and unoptimized measurements (eg run-time memory usage) by the original unoptimized measurement

The percentage decrease in memory consumption of the optimized programs is illustrated in Figure 6 2

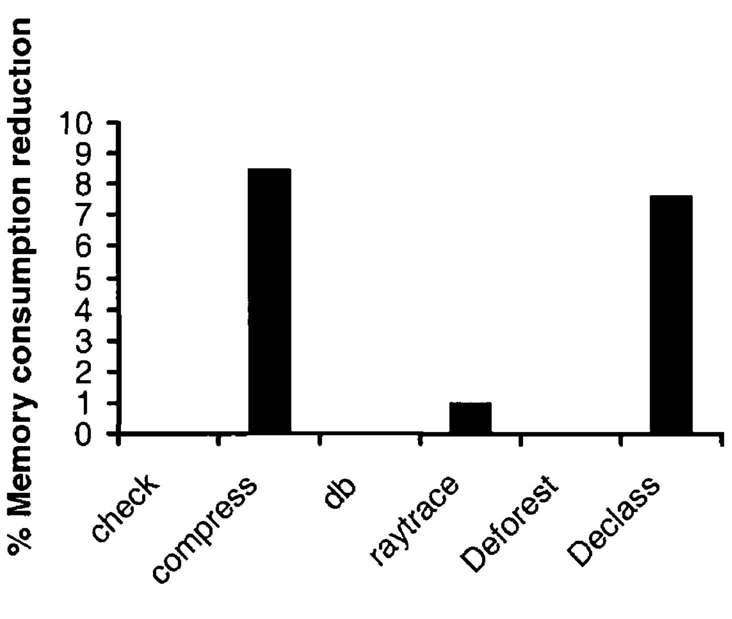


**Figure 6 2 Reduction in memory consumption of the test programs**

The programs, check, db and deforest show little or no improvement in memory consumption as a result of optimization This result is expected from the declassification of the deforest program as no optimization took place because there were no suitable classes for inlining The raytrace program shows a negligible decrease in memory consumption of 1% There are, however, significant improvements in the memory consumption of the compress and declass programs The declass program shows an 8% and the compress shows a 9% reduction in memory use

Figure 6 3 shows the percentage increase in the run-time performance of the optimized programs

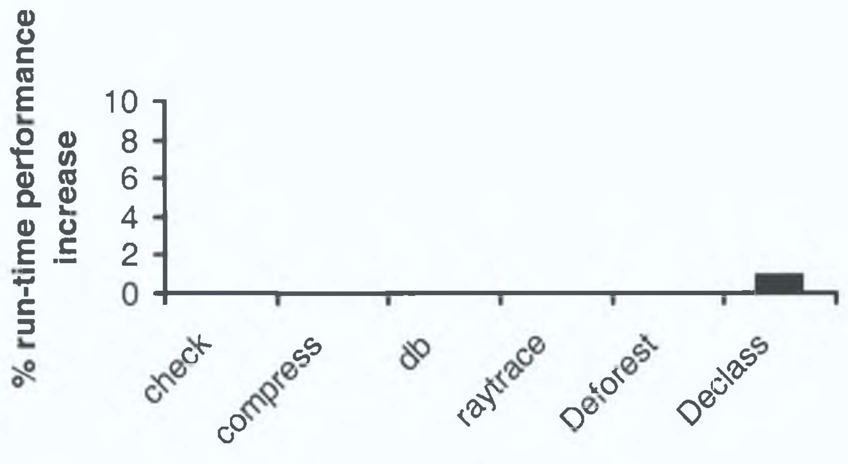


**Figure 6.3 Percentage increases in run-time performance**

The majority of programs tested; check, compress, db, raytrace and deforest show little or no increase in their run-time performances after declassification. This was expected from the deforest program as it had no suitable classes for inlining. The check, compress and db had only one class suitable for inlining. The declass program shows a very small increase even though three classes were inlined.

## 6.3.2 Program code size

A major cost of many optimization techniques is an increase in code size. This increase can often be considerable and may depend on the structure and complexity of the source program. The object inlining technique in [Dolby,98] estimates that there is a 20% increase in code size as a result of object inlining. It is thought that this increase could be substantially higher as each object could be inlined within multiple container objects. Their technique cannot guarantee what the maximum increase in code size will be. The declassification technique can guarantee that there is no increase in code size and there may be even a small decrease. This guarantee is possible because of the transformation algorithm used. Any suitable 'intermediate' class found is inlined within its enclosing class. The declaration of the intermediate class is then removed from the program. An intermediate class is *not* inlined within multiple enclosing classes. Figure 6.4 illustrates the percentage decrease in code size as a result of declassification. It is obvious that there is no increase in code size in any of the programs tested. The raytrace program which successfully inlined two classes shows a small reduction of 1%. The declass program shows a significant reduction of 5% in code size.

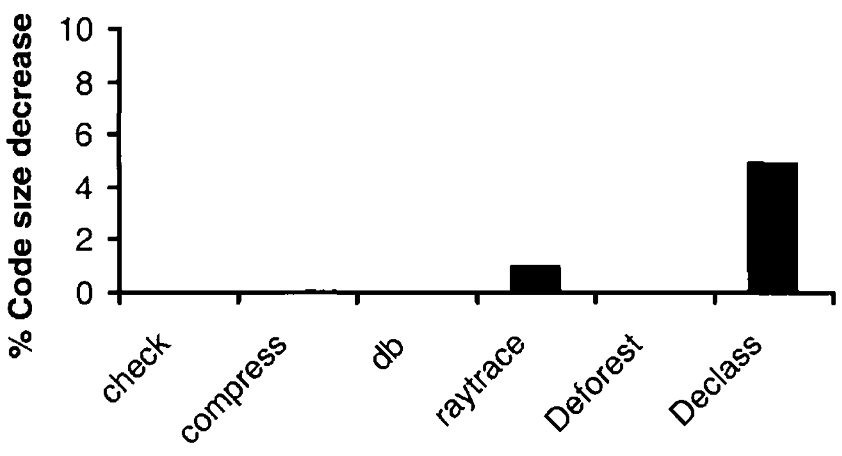


**Figure 6.4 Percentage decrease in program code size**

## 6.4 Further extensions

A number of extensions could be made to the declassification technique which should improve its effectiveness and these are listed in the following subsections The declassification technique concentrates on top-level classes which are declared and instantiated as a field of an enclosing class The analysis algorithm identified other top-level classes that have a usage count of one The declassification technique could be extended to inline these top-level classes, it could also be extended to inline single usage inner classes

### 6.4.1 Local Objects

The empirical study found that there are a considerable number of top-level classes that are instantiated as local variables in methods A local object cannot be inlined successfully if a reference to the local object is returned from the method or passed as a parameter to another class instance Figure 6 5 illustrates only the local objects which are used in this way 11% of the db program classes and 8% of the compress program classes are being used as local objects Each of these classes could be inlined within the method which created them by inlining its fields and expanding each of its method calls

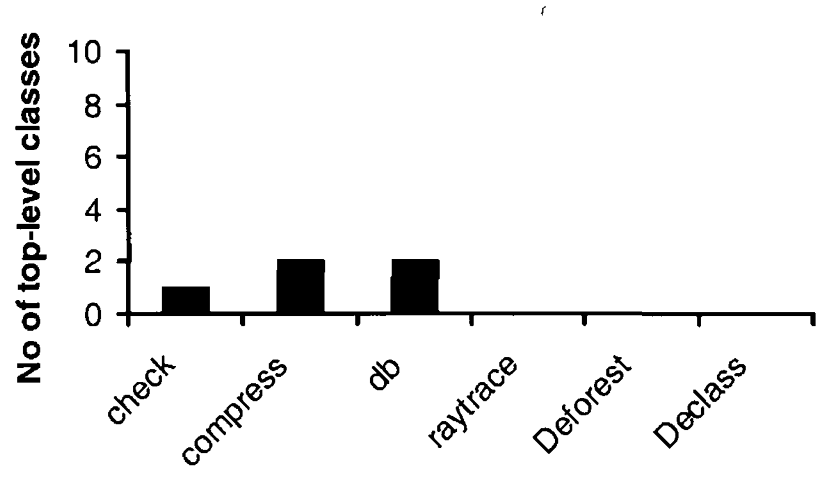


**Figure 6.5 Number of top-level classes created in a method**

Figure 6 6 illustrates how a local object could be inlined Figure 6 6 part (a) illustrates how a class *B* is instantiated in a method *instantiateLocal( )* This class could be inlined within the local method, in a similar way to how inlinable classes are inlined within their enclosing classes In Figure 6 6 part (b) the necessity of declaring and instantiating the class *B* has been removed

| (a) Program before declassification | (b) Program after declassification |
|---|---|
| <pre>class A<br>{<br><br>   public void instantiateLocal( )<br>   {<br>      B myB = new B( ),<br>      int x = myB count,<br>   }<br><br>}<br><br>class B<br>{<br>      int count = 1,<br><br>}</pre> | <pre>class A<br>{<br><br>   public void instantiateLocal( )<br>   {<br>      Object myB = new Object( ),<br>      int count = 1,<br>      int x = count,<br><br>    }<br><br>}</pre> |

**Figure 6 6 A local object is inlined within a method**

### 6.4.2 Anonymous objects

An anonymous object which is instantiated in a method could be inlined within the method in a similar way to how you inline a local object A reference to the anonymous object cannot be returned from the method or passed as a parameter to another class instance if inlining is to be successful Anonymous objects that have a usage count of one have a potential of being inlined Figure 6 7 illustrates the number of anonymous objects that are used in this way within the test programs Only one of the six programs has an anonymous object which could be inlined using a technique similar to the declassification technique

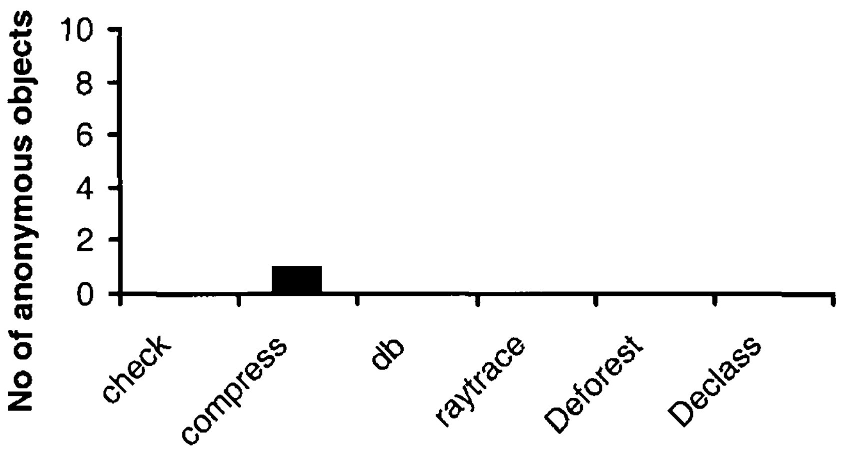


**Figure 6.7 Number of top-level classes created as anonymous objects**

### 6.4.3 Superclasses

The declassification technique could be extended to inline superclasses that have a usage count of one The fields and methods of the superclass could be inlined within its subclass This eliminates the necessity of creating an instance of this superclass within the program The declaration of this class could then be removed from the program The empirical study established that there were no top-level classes whose only use is as a superclass of another class

### 6.4.4 Inner classes

The declassification technique has the ability to identify and inline suitable top-level classes The information gathered by the declassification analysis and transformation algorithms could be extended to facilitate the inlining of inner class In Java each inner class is created as a normal top-level class by the JVM This results in each inner class requiring space and time to be created on the heap Eliminating inner classes should result in reduced memory consumption by the program and increased run-time performance These benefits should occur because of the reasons outlined in section 1 7 and 7 1 The effectiveness will greatly depend on the number of suitable classes found for inlining

#### 6.4.4.1 Member classes

The member class is a suitable class for declassification if it is established by the technique that there is exactly one use of it in the program Figure 6 8 part (a) illustrates how a class *A* has a member class *B* This member class is suitable for inlining if the analysis algorithm establishes that class *A* is the only class to use it Figure 6 8 part (b) displays the transformed program

```
class A
{
        B myB = new B( ),

        private class B extends C
        {
                int x,
        }
}
```

(a) Program before declassification

```
class A
{
        C myB = new C( )
        int x,

}
```

(b) Program after declassification

**Figure 6 8 A member class is inlined within an enclosing class**

### 6 4.4.2 Local Classes

A local class is suitable for inlining if it is used exactly once in the program A reference to the local class cannot be returned from the method or passed as a parameter to another class instance Figure 6 9 part (a) illustrates how a local class could be inlined Class *A* creates and instantiates a local class *B* in its method *createLocal( )* This local class could be inlined within the method *createLocal( )* as illustrated in Figure 6 9 part (b)

```
class A
{
    private void createLocal( )
    {
        B myB = new B( ),
        myB incX( ),


        class B extends C
        {
              private int x,
              void incX( )
              {
                        x++
              },
        }

    }
}
```

(a) Program before declassification

```
class A
{
    private void createLocal( )
    {
        C myB = new C( ),
        private int x,
        x++,

    }
}
```

(b) Program after declassification

**Figure 6 9 A local class is inlined within a method**

### 6 4 4 3 Anonymous Classes

The anonymous class is an ideal candidate for declassification as only a single instance of this class is created each time the containing block is executed Inlining should not be carried out if an anonymous class which is created within a method is returned from the method or passed as a parameter to another class instance Figure 6 10 part (a) illustrates how an anonymous class which extends the class *C* is instantiated and stored in a field *myC* Figure 6 10 part (b) shows how the fields and methods of the anonymous class are inlined within the method *createAnonymous( )*

```
class A
{

   public void createAnonymous( )
   {
      C myC = new C {
            private int x = 1;
            void incX( )
            {
                  x++;
            }
      };

      myC.incX( );
      :
   }
}

class C
{
   . .
}
```

```
class A
{

   public void createAnonymous( )
   {
      C myC = new C( );
      private int x = 1;
      x++;
      :
   }
}

class C
{
      . .
}
```

(a) Program before declassification (b) Program after declassification

**Figure 6.10 An anonymous class is created within a method**

The results of this analysis show that there are a considerable number of local objects which could be inlined within the test programs. The declassification technique could also be extended to encompass inner classes as well as top-level classes. These extensions have the potential of greatly improving its effectiveness.

## 6.5 Summary

The empirical study has shown that the declassification technique as it is currently defined, is not a successful optimization technique because the benefits in terms of increases in run-time performance and decreases in memory consumption are poor. Only a small number of classes were identified for inlining in the test programs. The results from this study showed that when suitable classes are found for inlining it did not have a positive impact on the run-time performance of these programs. The optimization of the declass program resulted in three classes being inlined. This only increased the run-time performance of the program by an average of 1%.

The majority of the test programs show a negligible decrease in memory consumption after optimization Two of the test programs, compress and declass however, have a considerable reduction in memory consumption as a result of declassification An important benefit of the declassification technique is the fact that it does not increase the size of the program code after optimization In all cases it has shown that there is no increase or a small decrease in code size after optimization A number of further extensions that have the potential of improving the performance of the declassification technique, were presented in section 6 4 The effectiveness of the declassification technique is discussed in the next chapter

# Chapter 7 Conclusions

Object-oriented programming is becoming increasing popular because it enables software development to be carried out in a uniform abstract way. It also enables software to be written which is independent of the implementation technique that will be used by the compiler. The object-oriented approach to software development has been praised for its ability to create systems that are flexible, maintainable and capable of evolving to meet changing needs. However, it has been adversely criticized for the amount of memory it requires during execution and the low speed performance of object-oriented languages in comparison to imperative languages. Consequently, object-oriented programs require more aggressive optimization techniques. However object-oriented software is more difficult to reason about and optimize, because of the use of inheritance and dynamically bound method calls. These features were highlighted in chapter 1.

A wide range of optimization techniques have been researched and developed for object-oriented languages. Some of these are presented and discussed in chapters 2 and 3. Empirical studies that have been carried out to assess the benefits of these techniques, have shown that they have varying degrees of success.

## 7.1 Evaluation of the Declassification Technique

The object inlining technique in [Dolby, 1997] has had considerable success in optimizing an object-oriented language. The empirical study made in [Dolby & Chien, 1998] estimates that a program will run up to three times as fast following object inlining. It is estimated that these gains are as a result of

- Eliminating the need for subroutine calls to access the object methods as they are now inlined,
- Enabling direct access to the fields of the inlined object because they are now local to the calling procedure,

- Providing better opportunities for other optimization techniques to be carried out on the inlined object

The aim of the declassification technique is to automatically inline suitable classes within their enclosing class A suitable class is identified by establishing that it is used exactly once in the program This class is then inlined within the class that uses it, which is referred to as its enclosing class The fields and methods of the inlined class become local to its enclosing class The declaration of the inlined class can then be removed from the program source code This transformation not only eliminates the necessity of creating this one instance of the inlinable class but the number of instantiations removed depends on the use of the enclosing class within the program Each time an instance of the enclosing class is created, the necessity of creating an instance of its inlinable class has been eliminated The success of the declassification technique is therefore not only dependent on the number of classes suitable for inlining but is intrinsically dependent on the number of instantiations of their enclosing classes and the use of these enclosing classes within the program

It was hoped that the declassification technique would result in a reduction in the heap space required by the program to run and an improvement in the run-time performance of the program The empirical study shows clearly that this is not the case One of the reasons the results of the evaluation yielded such a small number of suitable inlinable classes could be the fact that the SPEC98 benchmark suite is not particularly object-oriented Some of the programs in this suite are direct translations of Fortran The results of the empirical study are discussed in the following subsections

### 7.1.1 Number of inlinable classes

The empirical study shows that there are very few suitable classes for declassification Most of the test programs have one inlinable class, check, compress, and db The deforest program has no inlinable class The raytrace program has two inlinable classes out of forty one, the percentage of inlinable classes is, therefore, 5% The declass program was the most suitable for optimization as it has three inlinable classes out of twenty one, the percentage of inlinable classes is 14%

The 'one use' constraint of the analysis criteria is too restrictive as very few classes were found which adhered to it It could be argued that this constraint should be relaxed in the following manner to enable more classes to be inlined A class that is used frequently in one particular enclosing class and independently used in another part of the program should be inlined This would, however, require a substantial change to the analysis algorithm to facilitate its ability to guarantee that parts of the program are used independently of other parts of the program It would also break one of the fundamental constraints of the declassification technique, the fact that there is no increase in program code size after optimization

Extensions to the declassification technique as discussed in section 6 4 would enlarge the number of suitable classes This could be done without substantially changing the analysis algorithm

### 7.1.2 Memory use

It was hoped that the declassification technique would reduce the number of objects created by the program because of its ability to inline classes This, however, has only transpired in a small number of classes The empirical study shows that there are small differences in memory usage between the optimized and unoptimized programs The check and db programs show a negligible decrease in memory use after optimization No declassification has taken place in the deforest program and a decrease in memory use was not expected The raytrace program had some inlining opportunities but only a small reduction in memory use is found between the optimized and unoptimized raytrace programs The declass program shows an important and significant 8% reduction in memory use following the inlining of three classes The compress program shows a 9% decrease in memory use following the inlining of only one class It is possible to deduce that if more classes were inlined, the memory usage of the programs would reduce further

### 7.1.3 Run-time performance

The empirical study shows that there is very little or no difference in the run-time performance of check, compress, db, raytrace and deforest programs It was hoped that the performance of the declass and raytrace programs would be increased as a result of the declassification technique Some classes were inlined and it was hoped that it would reduce the number of memory dereferences as the fields of the inlinable class become local to the enclosing class It should also reduce the number of dynamic dispatches necessary to execute the program The empirical study showed no increase in the run-time performance of the optimized raytrace program and only a 1% increase occurred as a result of optimizing the declass program It could be deduced from these results that the fields and methods of the inlinable classes within the raytrace and declass programs are not highly referenced by their enclosing classes It could also be deduced that the overall poor impact on the run-time performance is predominately due to the small number of classes found suitable for declassification

### 7.1.4 Code size

An important feature which can be attributed to the declassification technique is that it guarantees that there will be no increase in code size This feature cannot be attributed to many other optimization techniques such as object inlining [Dolby, 1997] This is supported by the empirical study, which shows that there is no increase in code size after optimization Some programs that had inlining capabilities show a small but important decrease in code size The code size of the declass program reduced by 5% after declassification

In conclusion, the empirical study has shown that very few classes were found suitable for inlining within the test programs and the declassification technique was not successful in their optimization

## 7.2 Extensions to the declassification technique

A number of possible extensions to the declassification technique were outlined in section 6 4 The empirical study established that there are a considerable number of local objects which could be inlined The compress and db programs have two potential local objects and the check has one, this is an average of 7% of their overall classes The declass and deforest programs had no suitable local objects which could be inlined The empirical study showed that there are very few anonymous objects suitable for declassification It also showed that there are no suitable superclasses for declassification Consequently, pursuing these two extensions to the declassification technique are not worthwhile The inlining of local objects, however, could substantially increase the effectiveness of the declassification technique

The declassification technique concentrated on the inlining of top-level classes There are three types of inner classes, member, local and anonymous classes which could be inlined if it was established that their usage counts are exactly one Java's JVM treats each inner class as a normal top-level class Eliminating these inner classes should therefore provide similar benefits to the ones discussed for top-level classes

## 7.3 Summary

Optimizing object-oriented languages like Java is a challenging task The ultimate goal is to improve the efficiency of program execution while incurring the least number of negative side effects, such as the enlargement of code size

The declassification technique had the potential benefits associated with object inlining which where outlined in section 7 1, but unfortunately very few classes were found suitable for declassification One of the reasons such a small number of inlinable classes were found could be the fact that four of the test programs where taken from the SPEC98 benchmark suite This benchmark suite is not particularly object-oriented as some of the programs are direct translations from the Fortran language It would be interesting to see how the results of the declassification technique would alter if a more extensive empirical study was conducted that had a set of test programs which are intrinsically object-oriented The empirical study showed that when suitable classes are found for inlining it can have a positive effect on the memory consumption of the program Two of the programs in the empirical study showed a significant reduction of between 8 and 9% in memory consumption, as a result of declassification The effect of declassification on the run-time performance of the test programs is negligible It could be argued, however, that this is largely due to the small number of classes found suitable for declassification

The benefits of reduced memory consumption and increased run-time performance are small Taking all the results of this particular empirical study into consideration, it shows a poor optimization result Although this result is poor there are a number of significant features associated with this technique One of these is the fact that there is no run-time cost associated with the technique There is also no increase in the code size of a transformed program, if anything it will shrink in size The analysis and transformation algorithms are less complicated than many other optimization techniques Another feature is that it automatically inlines suitable classes and deletes the original class declaration, without any programmer intervention The programmer does not have to explicitly declare that certain classes should be inlined

Although the declassification technique was not successful in optimizing the test programs, further extensions to this technique could greatly improve its success Figure 6 5 shows that there are a number of local objects suitable for inlining within half of the test programs and section 6 4 4 discusses further possibilities for extending the technique by inlining inner classes These extensions combined with an intrinsically object-oriented set of test programs could greatly improve the effectiveness of the declassification technique

# References

[Agesen *et al* , 1998] Agesen, O , Detlefs, D , Eliot, J , Moss, B Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines In *Proceedings of the ACM SIGPLAN'98 Conference on Programming* Language *Design and Implementation (PLDI)*, pages 269-279, Montreal, Canada, 17-19, (1998)

[Baker-Finch, 1992] Baker-Finch, C A Relevance and Contraction A Logical Basis for Strictness and Sharing Analysis Submitted to the journal of Functional Programming, (1992)

[Black *et al* , 1986] Black, A , Hutchinson, N , Jul, E , Levy, H Object structure in the emerald system In proceedings of OOPSLA '86 pages 78-86 ACM September 1986

[Bodik *et al* , 1998] Bodik, R , Gupta, R , Soffa Mary Complete Removal of Redundant Expressions Department of Computer Science, June 1998

[Borning & Ingalls, 1982] Borning, A , Ingalls, D A type declaration and inference system for Smalltalk In Ninth Symposium on Principles of Programming Languages, pages 133-141, (1982)

[Briggs & Copper, 1994] Briggs, P , Cooper, K Effective partial redundancy elimination In Proceedings of the Conference on Programming Language Design and Implementation pages 159-170, June 1994

[Briggs *et al* , 1996] Briggs, P , Cooper, K , Simpson, L Value Numbering Software Practice and Experience, (1996)

[Bruce *et al* , 1993] Bruce K , Crabtree, J , Murtagh, T , van Gent, R Safe and decidable type checking in an object-oriented language In Proceedings of OOPSLA'93 pages 29-46, (1993)

[Budimlic & Kennedy 1997] Budimlic, Z , Kennnedy, K Optimising Java Theory and Practive Software Practice and Experience 9, 6 pages 445-463, June 1997

[Budimlic & Kennedy, 1998] Budimlic, Z , Kennnedy, K Static Interproedural Optimizations in Java Center for Research on Parallel Computation, Rice University, Technical Report CRPC-TR98746

| | |
|---|---|
| [Burstall & Darlington, 1977] | Burstall, R , Darlington, J A Transformation System for Developing Recursive Programs, Journal of the ACM 24(1) pages 44-67, (1977) |
| [Calder & Grunwald & Zorn 1994] | Calder, B , Grunwald, D , Zorn, B Quantifying differences between C and C++ programs Technical Report CU-CS-698-94, University of Colorado, Boulder, January 1994 |
| [Cardelli, 1984] | Cardelli, L A semantics of multiple inheritance In Gilles Kahn, David MacQueen and Gordon Plotkin, editors, *Semantics of Data Types* pages 51-68 Springer-Verlag (LNCS 173), (1984) |
| [Chambers & Dean & Grove, 1996] | [Chambers, C , Dean, J , Grove, D Whole program optimization of object-oriented languages University of Washington Seattle, Technical Report 96-06-02(1996) |
| [Chambers & Ungar, 1989] | Chambers, C , Ungar, D Customization Optimizing Compiler Technology for Self, A Dynamically-Typed Object-Oriented Programming Language SIGPLAN Notices, 24(7) 146-160 In Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, July 1989 |
| [Chambers & Ungar, 1990] | Chambers, C , Ungar, D Iterative Type Analysis and Extended Message Splitting Optimising Dynamically-Typed Object-Oriented Programs SIGPLAN Notices, 25(6) 150-164 In Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, July 1990 |
| [Chambers *et al* , 1989] | Chambers C , Ungar, D , Lee, E An Efficient Implementation of Self, A Dynamically-Typed Object-Oriented Programming Language Based on Prototypes In OOPSLA '89 Conference Proceedings, pages 49-70, New Orleans, LA 1989 Published as SIGNPLAN Notices 24(10) October 1989 |
| [Chambers *et al* , 1995] | Chambers, C , Dean, J , Grove, D A Framework for Selective Recompilation in the Presence of Complex Intermodule Dependencies In 17$^{th}$ International Conference on Software Engineering, Seattle, WA, April 1995 |
| [Chien et al , 1997] | Chien, A , Dolby, J , Ganguly, B , Karamcheti, V , Zhang, X Supporting high level programming with high performance The illinois concert system In proceedings of the Second International Workshop on |

High-level Parallel Programming Models and Supportive Environments, April 1997

[Chien, 1993] Chien, A Concurrent Aggregates Supporting Modularity in Massively-Parallel Programs MIT Press, Cambridge, MA, (1993)

[Chilimbi & Hill & Larus, 1999A] Chilimbi, T , Hill, M , Larus, J Cache-Conscious Structure Layout In Proceedings of ACM SIGPLAN'99 Conference on Programming Language Design and Implementation, (1999)

[Chilimbi & Hill & Larus, 1999B] Chilimbi T , Hill, M , Larus, J Cache-Conscious Structure Definition In Proceedings of ACM SIGPLAN'99 Conference on Programming Language Design and Implementation, (1999)

[Chin, 1990] Chin W Automatic Methods for Program Transformation Ph D thesis, Imperial College, University of London, July 1990

[Chin, 1991] Chin, W Generalising deforestation for all first-order functional programs In Journees de Travail sur L'Analyse Statique en Programmation Equationnelle, Fonctionnelle et Logique, pages 173-181, (1991)

[Chin, 1992] Chin W Safe fusion of functional expressions, in Proceeding of the ACM Conference on Lisp and Functional Programming, pages 11-20, (1992)

[Chow et al , 1997] Chow, F , Chan, S , Kennedy, R , Liu, S -M , Lo, R , Tu, P A new algorithm for partial redundancy elimination based on SSA form In Proceedings of the ACM SIGPLAN '97 Conf On Prog Language Design and Impl , pages 273-286, June 1997

[Cierniak & Li, 1997] [Cierniak, M , Li, W Optimizing Java Bytecodes Concurrency Practice and Experience 9 6 pages 427-444, June 1997

[Cytron *et al* , 1991] Cytron, R , Ferrante, J Rosen, B , Wegman, M , Zadeck, F Efficiently Computing Static Single Assignment Form and the Control Dependence Graph ACM Transactions on Programming Languages and Systems (1991)

[Dean & Chambers & Grove, 1995B] Dean J Chambers C Grove D Selective Specialization for Object-Oriented Languages SIGPLAN Notices In Proceeding of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, June 1995

[Dean & Grove & Chambers, 1995] Dean, J , Grove, D , Chambers, C Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis In Proceedings ECOOP '95, Aarhus, Denmark Springer-Verlag, August 1995

[Diwan & McKinley & Moss, 1998] Diwan, A , McKinley, K , Moss, J Type-based alias analysis See PLDI[1998], pages 106-117 (1998)

[Diwan *et al* , 1992] Diwan, A , Eliot, J , Moss, B , Hudson, R Compiler support for garbage collection in a statically typed language In Proceeding of SIGPLAN'92 Conference on Programming Languages Design and Implementation, volume 27 of ACM SIGPLAN Notices, ACM Press pages 273-282, (1992)

[Diwan, 1998] Diwan, A , McKinley, K , Moss, J Type-based alias analysis In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 106-117, Montreal, Canada, June, Notices 33 5 May 1998

[Dolby & Chien, 1998] [Dolby, J , Chien, A An evaluation of automatic object inline allocation techniques In Proceedings of the Thirteenth Annual Conference on Object-Oriented Programming Languages, Systems and Applications (OOPSLA), Vancouver, British Columbia, October 1998 Available at http //www-csag cs uiuc edu/papers/oopsla-98 ps

[Dolby & Chien, 2000] Dolby, J , Chien, A An Automatic Object Inlining Optimization and its Evaluation Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 345 - 357, Vancouver, British Columbia Canada, May 2000

[Dolby, 1997] [Dolby, J Automatic Inline Allocation of Objects In Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation pages 7-17, Las Vegas, Nevada June 1997

[Ellis & Stroustrup, 1990] Ellis M , Stroustrup, B The Annotated C++ Reference Manual Addison-Wesley, (1990)

[Gay & Steensgaard 1998] Gay D , Steensgaard B Stack Allocating Objects in Java Microsoft Research Technical Reports of the Advanced Programming Languages Group, (1998) [Goldberg & Robson, 1983] Goldberg, A , Robson, D Smalltalk-80 The Language and its Implementation Addison-Wesley, (1983)

[Gosling & Joy & Steele, 1996] [Gosling, J , Joy, B , Steele, G The Java Language Specification Addison-Wesley Longman, Inc , (1996)

[Grove *et al* , 1995] Grove, D , Dean, J , Garrett, C , Chambers, C Profile-Guided Receiver Class Prediction In Proceedings of the Tenth Annual Conference on Object-Oriented Programming Languages, Systems and Applications (OOPSLA), Austin, Texas United States, pages 108 - 123, October 1995

[Hamilton & Jones , 1991A] Hamilton, G W , Jones, S Extending deforestation for first-order functional programs, in Proceedings of the 1991 Glasgow WorkShop on Functional Programming, pages 134-145, August 1991

[Hamilton & Jones , 1991B] Hamilton, G W , Jones, S Transforming programs to eliminate intermediate structures, in *Journees de Travail sur L'Analyse Statique en Programmation Equationelle Fonctionnelle et Logique*, pages 182-188 October 1991

[Hamilton & Jones, 1990] Hamilton, G W , Jones, S Compile-time garbage collection by necessity analysis Proceedings of the 1990 Glasgow workshop on Functional Programming, pages 66-70, Springer-Verlag Workshops in Computing (1990)

[Hamilton, 1992] Hamilton, G W Sharing Analysis of Lazy First-order fucntional programs Proceedings of the workshop on Static Analysis, pages 68-78, 1992

[Hamilton, 1995] Hamilton, G W Compile-time garbage collection for lazy functional languages Proceedings of the 1995 International Workshop on Memory Management LNCS 986, (1995)

[Hamilton, 1995B] Hamilton, G Higher-order deforestation Technical Report 95-07, University of Keele (1995)

[Hamilton, 1996] Hamilton, G W Higher-order Deforestation, in Proceedings of the Eighth International Symposium on Programming, Logics, Implementation and Programs (PLILP '96), Vol 1140 of Lecture Notes in Computer Science, pages 213-227, (1996)

[Hamilton 1998] Hamilton, G W Usage Counting Analysis for Lazy Functional Languages Information and Computation 146(2) pages 100-137, (1998)

| | |
|---|---|
| [Harrison & Waldron, 1999] | Harrison, O , Waldron, J *Profiling Java memory demographics for Garbage Collection* Purposes Working Paper CA-2299, Dublin City University, (1999) |
| [HOlzle & Ungar, 1994] | HOlzle U , Ungar, D Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback SIGNPLAN Notices, 29(6) pages 326-336 In Proceeding of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, June 1994 |
| [Hosking *et al* , 1999] | Hosking A , Nystrom, N , Whitlock, D , Cutts, Q , Diwan, A Partial Redundancy Elimination for Access Path Expressions In *Proceedings of the International Workshop on Aliasing in Object-Oriented Systems*, Lisbon, Portugal, June 1999 |
| [Hudak, 1987] | Hudak, P A Semantic Model of reference counting and its Abstraction In S Abramsky and c Hankin, editors, Abstract Interpretation of Declarative Languages 45-62 (1987) |
| [Hughes, 1991] | Hughes, S Static analysis of store use in functional programs PhD Thesis, University of London (1991) |
| [Hutchinson, 1987] | Hutchinson, N Emerald An Object-Based Language for Distributed Programming PhD thetis, University of Washington, Department of Computer Science, Seattle, Washington TR-87-01-01, (1987) |
| [JavaCC, 1999] | Java Compiler Compiler version 1 1 (Parser Generator), copyright © 1996-1999 Sun Microsystems Inc http //www metamata com URL last accessed on 04/September/2001 |
| [Johnson, 1992] | Johnson, R Documenting Frameworks Using Patterns In Proceedings OOPSLA 92, pages 63-76 Published as ACM SIGNPLAN Notices, volume 27,number 10, October 1992 |
| [Jones & Le [Metayer 1989] | Jones, S B , Le Metayer, D Compile-Time Garbage Collection by Sharing Analysis Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture 54-74, (1989) |
| [Kaplan & Ullman, 1978] | Kaplan, M , Ullman, J A general scheme for the automatic inference of variable types In Fifth Symposium on Principles of Programming Languages, pages 60-75, (1978) |
| [Knoop & Steffen, 1992A] | Knoop, J Sheffen, B The interprocedural coincidence thoerem In proceedings of the 4th |

Conference on Compıler Constructıon (CC) Lecture Notes ın Computer Scıence, vol 641 Sprınger-Verlag, Berlın, pages 125-140, (1992)

[Knoop & Steffen, 1992B] Knoop, J , Sheffen, B Optımal ınterprocedural partıal redundancy elımınatıon Extended abstract In Addenda to Proceedıngs of the 4th Conference on Compıler Constructıon Lecture Notes ın Computer Scıence, Sprınger-Verlag, Berlın, (1992)

[Knoop *et al* 1994] Knoop, J , Ruthıng, O , Sheffen, B Optımal code motıon Theory and practıce ACM Trans On Programmıng Languages and Systems, 16(4) pages 1117-1155, (1994)

[Laud, 2001] Laud, P Analysıs for Object Inlınıng ın Java JOSES Java Optımızatıon Strategıes for Embedded Systems, Genoa, Italy, Aprıl 1 2001

[Lee & Zorn, 1997] Lee, H B ,Zorn B G BIT A Tool for ınstrumentıng Java Bytecodes In Proceedıngs of the 1997 USENIX Symposıum on Internet Technologıes and Systems (USITS'97), PAGES 73-83, December 1997

[Lım & Stolcke, 1991] Chu-Cheow, L , Stolcke, A Sather Language Desıgn and Performance Evaluatıon Technıcal Report TR 91-034, Internatıonal Computer Scıence Instıtute, May 1991

[Lındholm & Yellın, 1996] [Lındholm, T , Yellın, F The Java Vırtual Machıne Specıfıcatıon Addıson-Wesley Longman, Inc , (1996)

[Lınton *et al* , 1989] Lınton, M , Vlıssıdes, J , Calder, P Composıng User Interfaces wıth InterVıews IEEE Computer, 2(2) 8-22, February 1989

[Marlow & Wadler, 1992] Marlow, S , Wadler, P Deforestatıon for Hıgher-Order Functıons, ın Proceedıngs of the Fıfth Annual Glasgow Workshop on Functıonal Programmıng, 1992 pp 154-165, (1992)

[Marlow, 1996] Marlow, S Deforestatıon for hıgher-order functıonal programs Ph D thesıs, Glasgow Unıversıty, (1996)

[McDowell, 1998] McDowell, C Reducıng garbage ın Java SIGNPLAN Notıces, 33(9)pages 84-86 (1998)

[Mılner *et al* , 1990] Mılner, R , Tofte, M , Harper R The Defınıtıon of Standard ML The MIT Press (1990)

[Mılner, 1978] Mılner, R A theory of type polymorphısm ın programmıng Journal of Computer and System Scıences, 17 348-375 (1978)

[Mitchell *et al* , 1993] Mitchell, J , Honsell, F , Fisher, K  A lambda calculus of objects and method specialization  In 1993 IEEE Symposium on Logic in Computer Science, pages 26-38, June 1993

[Mohnen, 1995] Mohnen, M  Efficient Compile-Time Garbage Collection for Arbitrary Data Structures  University of Aachen, (1995)

[Morel & Renvoise, 1979] Morel, E , Renvoise, C  Global optimization by suppression of partial redundancies  Communications of the ACM 22 2, pages 96-103, (1979)

[Mowry & Lam & Gupta, 1992] Mowry, T , Lam  S , Gupta, A  Design and evaluation of a compiler algorithm for prefetching  In Proceedings of the Fifth International Conference on Architectural
Support for Programming Languages and Operating Systems, (ASPLOS V), pages 62-73, (1992)

[Mycroft, 1981] Mycroft, A  Abstract Interpretation and Optimising Transformations for Applicative Programs  PhD thesis, University of Edinburgh, (1981)

[Oxhøj *et al* , 1992] Oxhøj, N , Palsberg, J , Schwartzbach, M  Making Type Inference Practical  In *Proceedings of ECOOP'92*, Sixth European Conference on Object-Oriented Programming, pages 329-349  Springer-Verlag (LNCS 615), Utrecht The Netherlands  July 1992

[Palsberg & Schwartzbach, 1991] Palsberg, J , Schwartzbach, M  Object-Oriented Type Inference  In *Proceedings of OOPSLA'91*, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications, pages 146-161, Phoenix, Arizona, October 1991

[Plevyak & Chien  1994] Plevyak, J , Chien, A  Precise concrete type inference of object-oriented programs  In proceedings of OOPSLA 1994, Object-Oriented Programming Systems, Languages and Architectures, pages 324-340, (1994)

[Plevyak & Chien, 1995] Plevyak, J , Chien, A  Type directed cloning for object-oriented programs  In Proceeding of the Workshop for Languages and Compilers for Parallel Computing, pages 566-580, (1995)

[Plevyak  1996] Plevyak, J  Optimization of Object-Oriented and Concurrent Programs  PhD thesis  University of Illinois at Urbana-Champaign, (1996) [Seidl & Sørensen, 1997] Seidl, H , Sørensen, M Constraints to Stop Higher-Order Deforestation, in Proceedings of the Twelfth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pages 400-413, (1997)

[Seidl, 1996] Seidl, H Integer Constraints to Stop Deforestation, in Proceedings of the European Symposium on Programming, pages 326-340, (1996)

[Shivers, 1988] Shivers, O Control flow analysis in scheme In SIGPLAN Conference on Programming Language Design and Implementation, pages 164-74 (ACM 1988)

[Shivers, 1991] Shivers, O Control-Flow Analysis of Higher-Order Languages PhD thesis, Carnegie Mellon University Department of Computer Science, Pittsburgh, PA, May 1991 Also CMU-CS-91-145

[Simpson, 1996] Simpson, L Value Numbering Technical Report, Rice University, Available via ftp, September 1996

[Smetsers *et al* 1993] Smetsers, S , Barendsen, E , van Eekelen, M , Plasmeijer, R Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs Technical Report 93-4, University of Nijmegen, (1993)

[SPEC JVM, 1998] SPEC JVM98 Benchmarks http //www spec org/osg/jvm98 URL last accessed on 08/August/2002

[Steffen, 1996] Steffen, B Property oriented expansion In Proceeding of the Int Static Analysis Symposium (SAS'96), volume 114 LNCS, pages 22-41, Germany 1996

[Suzuki, 1981] Suzuki, N Inferring types in Smalltalk In Eighth Symposium on Principles of Programming Languages, pages 187-199, (1981)

[Wadler, 1984] Wadler, P L Listlessness is better than laziness lazy evaluation and garbage collection at compile-time Proc ACM Symp On Lisp and Functional Programming, pages 45-52, (1984)

[Wadler, 1985] Wadler, P L Listlessness is better than laziness II composing listless functions, in Proc Workshop on Programs as Data Objects, Copenhagen, Lecture Notes in Computer Science 217 282-305, (1985)

[Wadler, 1990] Wadler, P Deforestation Transforming Programs To Eliminate Trees Theoretical Computer Science 73 pages 231-248, (1990)

[Wright & Baker-Finch, 1993] Wright, D A ,Baker-Finch, C C Usage Analysis with Natural Reduction Types Third International Workshop on Static Analysis, (1993)