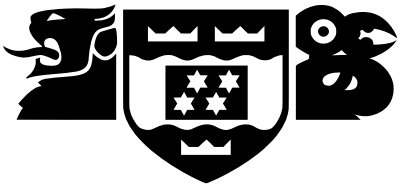
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Mathematics, Statistics and Computer Science
Te Kura Tatau

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

Optimising Java Programs with Pure Functions

Andrew Walbran

Supervisors:
Associate Professor Lindsay Groves
Doctor David Pearce

October 12, 2008

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

Abstract

Java programs often contain expressions involving method calls that are executed many times in a loop but will always return the same value, such as checking the length of a collection through which the loop is iterating. This can help to make the code readable, but makes it less efficient to execute than it could be. Current optimising compilers are generally unable to perform loop invariant code motion on such method calls and object field accesses due to a lack of knowledge of functional purity, aliasing and other information about the program structure. We propose that the problems of finding this information and of using it for optimisation be separated by the use of a number of annotations which we introduce. We describe under what conditions such optimisations can be made, and implement our scheme in JKit.

Acknowledgements

Thank you to my supervisor, Associate Professor Lindsay Groves, for agreeing to supervise this project and for support, advice and constructive criticism. Thanks to Dr. David Pearce for the project idea, for providing supervision even though not officially, for his support, advice and encouragement despite having much else to do, and (together with all the others who have contributed) for creating JKit. Thanks also to the members of ELVIS for the suggestions and references they have provided, not to mention the free pizza at their first and last meetings. Thanks to my office-mates in Vegas for their support these last few weeks, and to the various postgrads who have proofread this report for me. Finally, thanks are due to all who have encouraged and commiserated with me along the way this year.

Contents

Chapter 1

Introduction

1.1 Motivation

Many advances in compiler technology have been made over the past years, and these have improved the performance of Java programs. However, Java and other similar object-oriented languages continue to grow in popularity for a wide range of applications from high-end servers serving large numbers of clients to mobile phones with very limited resources.

Many of the programmers writing these applications do not know enough about the underlying hardware and software to write efficient code, and sometimes the language design makes it difficult for them to know which approach is more efficient. Arguably they should not have to worry about issues at this level anyway, but should be free to concentrate on higher-level problems while the compiler takes care of the low-level problems of optimisation. Furthermore, having programmers doing their own manual optimisation at this level tends to result in less readable code, which is detrimental to maintainability. There is therefore an increasing need for better compilers that can produce more efficient code while requiring less low-level work of programmers.

In this project, we look at one aspect of this. We consider how we can move certain method calls out of loops so that they are executed fewer times, and so make programs run faster. Existing well-known compiler optimisation techniques are able to move simple expressions out of loops, but are generally unable to move method calls because of the unknown side-effects and dependencies that methods may have.

Figure 1.1 shows a simple example of how a programmer might write a loop to iterate through a collection of some sort. Note that `collection.size()` is called on each iteration of the loop, but will always return the same value. We will discuss this more later, but this is due (among other things) to the fact that `TestCollection.size()` is a *pure* method.

```
TestCollection collection = ...
for (int i = 0; i < collection.size(); ++i) {
    System.out.println(collection.get(i));
}
```

Figure 1.1: A simple loop through a collection, written naïvely without any optimisation

If a method call in a loop always returns the same result, such as the call to `collection.size()` in our example, then we can make the program faster by moving the call out of the loop. This is shown in Figure 1.2. To be able to do this safely, we have to know that the call will not have any side-effects other than returning a value, and that nothing will change between

iterations of the loop that might cause the value of `collection.size()` to change.

```
TestCollection collection = ...
int size = collection.size();
for (int i = 0; i < size; ++i) {
    System.out.println(collection.get(i));
}
```

Figure 1.2: The same loop through a collection, optimised to call `collection.size()` only once

While the programmer could make this transformation manually, the original form is easier to read and write. We would like the compiler to make this sort of transformation (and many others like it) automatically.

1.2 Contributions

The main contribution from this project is a method of optimising pure method calls by moving expressions involving them out of loops where possible. This was achieved by the use of Java annotations to break the problem down into parts. We move some of the work of analysing the program (such as detecting aliases) to the programmer or to inference systems which can be developed separately in future projects.

We implemented our optimisation approach in a Java compiler called JKit [1], which is developed here at VUW as a research compiler. We tested our optimisations with a number of small test programs and with two larger programs.

There are also a number of minor contributions adding various small features to JKit. We will list all our contributions in the conclusion of this report.

1.3 Outline

We start in **chapter 2** by giving some background information about functional purity, Java annotations and the JKit compiler in which the optimisations in this project were implemented. We also review some of the existing literature on compiler optimisations and pure functions. We then discuss in **chapter 3** the optimisations which we developed, the annotations which we introduce, and the problems with aliasing which make our task difficult. **Chapter 4** discusses the details of how we implemented these optimisations in JKit, including how we added support to JKit for annotations and for preserving information about loops. It also discusses the flow graph visualisation utility which was developed to aid debugging of these optimisations. In **chapter 5** we discuss the results of evaluating our optimisations on a number of simple benchmark programs and on two real programs. **Chapter 6** then concludes by summarising our contributions and discussing future work suggested by our experiences and results in this project.

Chapter 2

Background

In this chapter we will look at existing compiler optimisations, including loop invariant code movement in particular. We will explain what it means for a method to be *pure*, and look at some of the existing literature relating to pure functions and other related concepts in various languages. We will then briefly introduce Java's *annotation* mechanism (which we will later use), and the JKit compiler in which we implemented our optimisations in this project.

2.1 Compiler optimisation techniques

We reviewed the relevant parts of a number of compiler textbooks to get an overview of currently well-known optimisation techniques. As well as optimisation techniques we mention data flow analysis here. We do not actually use data flow analysis in this project, but it is relevant because inferring the annotations which we will discuss later would require data flow analysis. This possibility is discussed a bit in section 6.2.2 as possible further work. Here we list the relevant sections to which interested readers may wish to refer for more background on this area:

- Chapter 9 of [2] (on data-flow analysis) includes a little about interprocedural analysis. There is also a little about code motion etc., though it is not directly useful to this project.
- Chapter 18 of [3] on loop optimisations may be useful. The same also has chapter 17 on data-flow analysis.
- [4] has some useful information in chapter 10 about code motion, induction variable elimination and strength reduction, along with lots of data-flow analysis.
- 16.3 of [5] has some information about loop optimisations (factoring, strength reduction), and a fair bit of data flow analysis.

The optimisation we implement in this project is a form of loop invariant code motion, also known as loop factoring or 'hoisting'. The idea of this optimisation is to find expressions that always have the same value across all iterations of the loop, and factor them out so that they are evaluated only once rather than each time through the loop. Such expressions are said to be *loop invariant*; *code motion* refers to the process of moving them. Loop invariant code motion is traditionally limited to simple arithmetic or logical expressions using built-in operators. For example, Figure 2.1 shows a simple loop before and after the expression `x * y` is factored out. What we add in this project is the ability to move method calls, as well as field references and array indexing expressions.

```
for (int i = 0; i < 10; ++i) {
    foo(x * y + i);
}
```

(a) Before loop invariant code motion

```
int xy = x * y;
for (int i = 0; i < 10; ++i) {
    foo(xy + i);
}
```

(b) After loop invariant code motion

Figure 2.1: An example of traditional loop invariant code motion. The expression `x * y` is factored out of the loop so that it is only evaluated once, with the result stored in a new variable `xy` for when it is needed.

In [6], Hammond and Lacey looked at applying traditional compiler optimisations (in particular loop unrolling, loop unfolding and loop invariant code motion) to Java bytecode. They avoided dealing with method calls, essentially because they did not have a way of knowing whether the methods were functionally pure. Unlike the authors of this paper we are able to move expressions containing method calls, at least in some cases. The authors found that the optimisations they performed gave an average 4–5% performance increase across a number of benchmarks, which suggests that such optimisation techniques are worthwhile.

2.2 Functional purity

Java compilers are limited in how they can move method calls to optimise Java programs by the fact that methods may in general have *side effects*. For example a method that returns a value may also change the value of one of the object's fields, or even some other state in an apparently unrelated part of the program. The behaviour of methods may also (in some cases) depend on the state of parts of the program to which they have no obvious relationship. For example, a method's return value might depend on the value of a global variable (in Java, this means a static field in some class).

In practice, however, many methods are *functionally pure*: they do not alter any state (so have no side effects), and they return a value depending only on the arguments passed to them (including the implicit `this` parameter). This second condition is sometimes referred to as *functional determinism* [7]. In this report we will sometimes abbreviate 'functionally pure' simply to *pure*.

A range of definitions are possible of what it means for a method to be side-effect free. For example, Finifter et al. [7] require that side-effect free methods only modify objects which are created as part of the execution of the method, as well as not causing any side-effects outside of the language environment except for resource consumption. This definition is sufficient for our idea of purity. Specifically, we require that pure methods do not have any side effects that would make it noticeable for the method to be executed a different number of times, or that might affect the execution of other pure methods.

2.2.1 Example

In the example class shown in Figure 2.2, the `mult` method is pure, but the `set` method is not pure as it alters the state of the object (a side effect). The `multk` method is not pure by this definition either, as it reads a static field which could change, and so may give a different result even if called with the same arguments and without the object having changed. This breaks functional determinism.

```
public class Example {
    private int x;
    public static int k;

    public int mult(int y) {
        return x * y;
    }

    public set(int a) {
        x = a;
    }

    public int multk(int y) {
        return x * y * k;
    }
}
```

Figure 2.2: A class with several methods; only the `mult` method is pure by our definition.

If programmers were to annotate these pure methods as such (or if they could be automatically detected by program analysis), then the compiler would be able to make optimisations which would not otherwise be possible. Such annotations would also be helpful in making the programmer's intent and assumptions clear, resulting in more readable code. We will discuss in section 3.2 the annotations which we introduced for functionally pure methods, and also for methods meeting a weaker restriction that they not modify the state of their target.

2.2.2 Object equivalence

To precisely define what a pure function is (in particular for the definition of functional determinism), it is necessary to define exactly what it means for two return values or arguments to be equivalent. [7] discusses some of the issues here. One question is whether, to be considered equal, two sets of object references must (a) simply refer to equal objects, (b) refer to equal objects with the same aliasing relationships, or (c) have the exact same addresses. These are progressively stronger requirements.

For our part, we leave this decision up to the programmer: as long as the programmer chooses a consistent definition, our optimisations will be valid. That is, the programmer must use the same definition of equivalence for arguments as for return values when declaring a function pure, and also must only rely on equivalence as they define it when using the results returned by pure methods. We suggest that objects with the same values but different addresses not be considered equivalent in general, as they will behave differently when compared with Java's == operator. There may however be some cases where the programmer can be sure that this will not be a problem, and so can use a looser definition of equivalence.

Being able to offer this flexibility is an advantage of our approach (which we discuss in section 3.2 and later) of having the programmer annotate methods etc. rather than having these properties automatically inferred or checked.

2.3 Use of pure functions

A moderate amount of work has been done on the use of certain functions marked as pure in Java and other imperative programming languages. However, this has mostly been for use in specifications rather than for optimisation, at least in the case of Java.

2.3.1 Pure functions in specifications

In Java, there has been some work done on the use of pure functions for specifications, such as in Java Modelling Language (JML) [8, 9]. Similar work has been done with Spec#[10] for C#. These studies have considered various different levels and definitions of purity. Here we discuss some of these.

Darvas and Müller [11] describe a notion of *weak purity* that allows methods to construct new objects and even return them as long as they do not modify pre-existing objects. This means that a method could return references to different objects when called with the same arguments. As we discussed earlier, this can result in different behaviour later on in the program even though both objects will contain the same values, because Java's `==` operator will give different results. The default implementation of `Object.hashCode()` method in Java also depends on the memory address of the object.

Naumann [12] gives a definition of *observational purity* which allows limited modification of state while still maintaining the effect of purity from the viewpoint of other classes. He gives a formal definition of this observational purity in terms of equivalence of class implementations. This may well be a useful approach to take if a formally-verifiable definition of purity is needed, but in the scope of this project we only need an informal definition.

Leino and Müller [13] have done some work on what they call 'equivalent-results methods' that query a data structure in a consistent and repeatable way. They were able to build a formal model to verify such methods, and dealt with such issues as methods that allocate new objects on the heap. The application for which their work was designed was again program specification, and they mentioned that they planned to implement it in the Spec# verifier Boogie [14].

Barnett et al. [15] again discuss observational purity for use in specification languages. They give a weaker notion of purity that allows specifications to do more while still remaining safe. They compare their approach to the restrictions made on specifications in ESC/Java, JML and Eiffel.

2.3.2 Finding pure methods automatically

Finifter et al. [7] have described a way to automatically find functionally pure methods in programs written in a subset of Java which they call *Joe-E*. This analysis is based on immutable class types declared by the programmer and object-capabilities enforced by the language design. The applications they describe are for security and verifiability of software, but this purity and immutability information would also be useful for applying optimisation techniques such as those developed in this project. Their definition of purity seems to be compatible with the flexible definition made for this project; in fact we based part of our explanation on this paper.

Zhao et al. [16] discuss a simple analysis of Java bytecode at runtime in a JIT compiler to determine whether methods are pure, and using this information to make optimisations such as constant folding and omission of synchronisation operations. They also use information provided by the programmer. They achieved by this an average 1.29% speed improvement across a range of benchmarks.

Both of these approaches seem to be quite limited in the sorts of pure functions they can find, and both restrict when they can work: the first approach works only on Joe-E programs, and the second only at runtime, when information about the whole program is available to the interprocedural analysis.

2.3.3 Other restrictions on functions, and other programming languages

The optimisations in this project require not just information about which functions are pure, but about how the side-effects of other functions may alter the program state. One useful class of methods is those that are not necessarily pure in terms of their return value or other side-effects, but are still guaranteed not to modify the object on which they are called. We will call these *const* methods (after C++), although the terminology is not entirely standard. There is some use both of pure functions and of guarantees like this in existing programming languages.

- C++ allows member functions to be marked as `const` if they will not modify the object they are called on, and so can be called on a constant instance of their class [17]. This is very much like the `@Const` annotation that we will introduce in section 3.2.

- GCC allows C functions to be marked `__attribute__ ((pure))` if they have no side-effects and depend only on their parameters and global variables, or `__attribute__ ((const))` if they have no side-effects and depend only on their parameters [18, 19]. The latter attribute is the closest to our idea of a pure method. It uses these attributes to allow common subexpression elimination and loop optimisation, similar to what we do in this project. Note however that doing such optimisations in C is significantly easier, as functions are called statically (except for the special case of function pointers) in contrast to Java's ubiquitous use of dynamic dispatch. It is also more common in C to pass arguments by value rather than by a pointer like with objects in Java, which means that aliasing is less often a problem.

- Fortran 95 (from the earlier High Performance Fortran) allows pure functions to be marked with the `PURE` keyword, to allow optimisation of `FORALL` loops on parallel systems [20].

2.4 Annotations in Java

This project defines and uses a number of annotations to mark parts of programs.

The annotations mechanism in Java was introduced in JSR-175 [21], and was made part of the core language specification in Java 1.5. It allows Java developers to define arbitrary 'annotations' with which they can then mark various parts of Java programs, such as classes, methods, variable definitions and so on. Annotations can have arbitrary numbers of parameters, but the annotations used in this project are all of the simplest variety known as 'marker' annotations, which have no parameters [22].

As an example, Figure 2.3 shows a simple class which demonstrates some annotations of the sorts used in this project: the class `Foo`, method `bar` and local variable `j` are annotated `@Encapsulated`, `@Pure` and `@Unique` respectively. The meanings of these annotations will be

explained in chapter 3; for now it is enough to know that Java provides the facility to define such annotations. Annotations always start with the `@` character.

```
@Encapsulated class Foo {
   public @Pure int bar(int i) {
      @Unique Integer j = new Integer(i);
      return j.intValue();
   }
}
```

Figure 2.3: A class with some annotations

2.5 JKit

JKit [1] is an experimental Java compiler developed at Victoria University of Wellington, primarily by David Pearce. It is designed to be extensible for research such as this project. The optimisations described later in this report were implemented as a *stage* for JKit.

JKit is designed to compile programs in a number of stages. The Java source code to be compiled is first parsed into an AST, which is then immediately transformed to JKit's flow graph intermediate representation (called JKIL, for JKit Intermediate Language) by `JavaFileReader`. This flow graph form of the program is then manipulated by a number of stages to do things such as type checking, expressing complex constructs (foreach loops, for example) in terms of simpler ones, and making optimisations. The optimisations described in this report were implemented as a new optimisation stage, called `LoopInvariantMovement`. Finally, JKit writes the flow graph out to a Java class file with its `ClassFileWriter`. It is possible to use other writers instead for debugging purposes.

A (control-)flow graph is a directed graph of how control may flow through a program when it is executed. Each vertex will generally correspond to a statement or instruction in the program. Conditional statements (such as `if`) result in vertices with out-degree greater than one as the graph branches (later to rejoin), while loops result in cycles in the graph. The edges corresponding to such conditional and looping constructs are labelled with the condition under which that edge is taken, while unlabelled edges represent the normal, unconditional flow of control.

As an example of the flow graphs that are constructed as the first part of the process outlined above, Figure 2.5 shows the flow graph that JKit generates from the Java source code in Figure 2.4. There are a few things worth noting here about how JKit works:

- The numbers like ':12:7' at the end of each vertex label indicate the corresponding line and character in the Java source file.
- To more easily represent the program structure, some vertices do not contain a statement but exist only for structure. These are used for loops and conditionals.
- JKit prefixes each variable name with a number to indicate the scope of the variable. This allows it to distinguish between two different variables with the same name.

```
private static int foo(int x) {
    if (x > 16) {
        System.out.println(x + " is too big");
        return -1;
    }
    else {
        int factorial = 1;
        for (int i = 1; i <= x; ++i) {
            factorial *= i;
        }
        return factorial;
    }
}
```

Figure 2.4: A method for which we will shown a flow graph

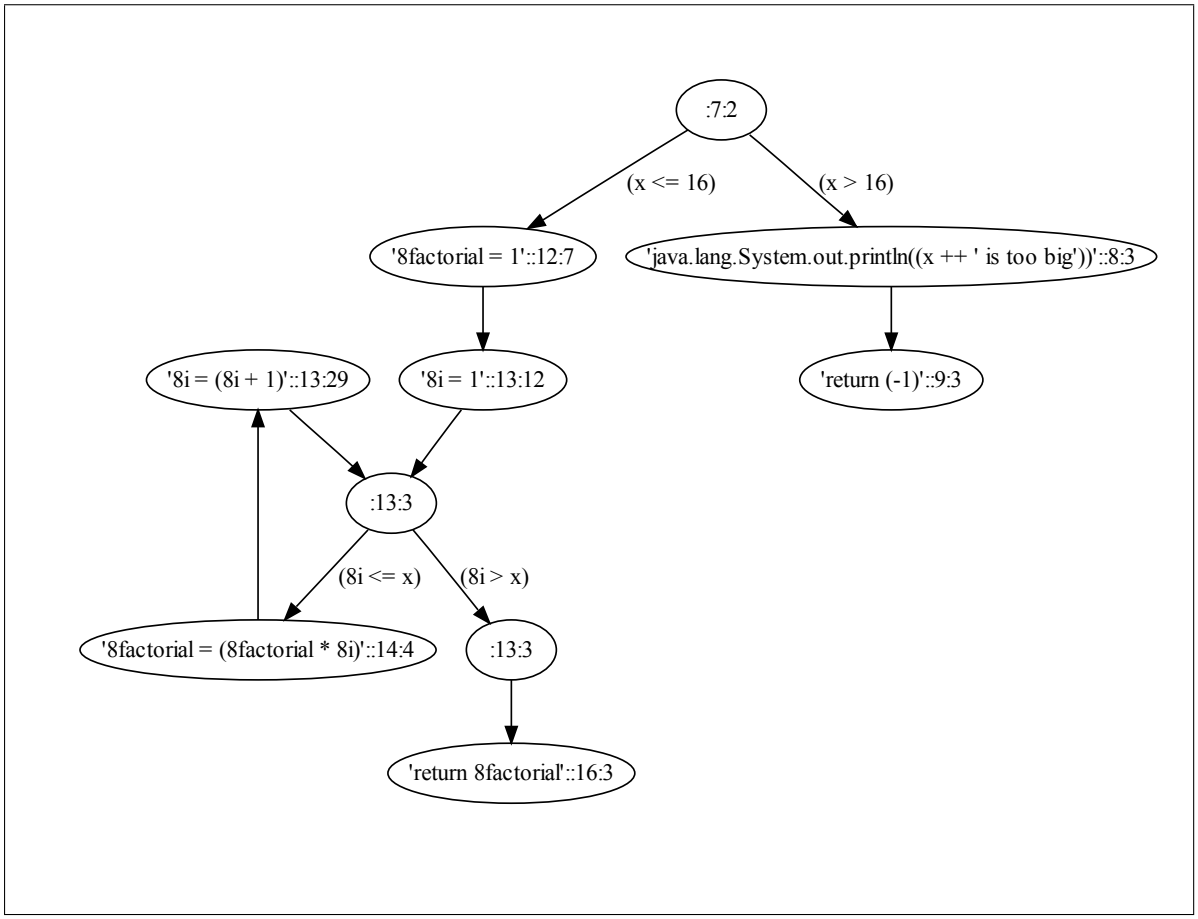


Figure 2.5: The flow graph for the example method show in Figure 2.4

Chapter 3

Optimisations: loop invariant code motion for Java

We will now discuss in detail the approach that we take to determining which method calls and other expressions can be moved out of loops without changing the semantics of the program we are trying to optimise. That is to say, we want the program to run in the same way as far as it can be observed, apart from being faster and perhaps using more or less memory.

This requires that we find expressions which we can guarantee to be *loop invariant*: expressions which will always have the same value across all iterations of the loop. Such expressions can be moved, or *factored*, out of the loop. They need only be evaluated once at the start of the loop, and this value remembered and used whenever the original expressions would have been used.

We will start by discussing a simple case with array lengths, then talk about how we annotate pure methods. We will then look at aliasing and how it makes our problem more difficult, then combine this all and consider under exactly what conditions we can move calls to pure methods.

3.1 Array lengths

Java arrays have fixed length: once an array has been constructed, values in the array may be modified but the length of the array always remains constant. Thus `Array.length` is a common special case of a field that is invariant as long as the array reference is invariant. This means that it is a fairly easy candidate for factoring out of loops; more easy than even pure method calls, which can depend on the internal state of objects. This was the first useful optimisation developed in this project.

The exact condition is this: an expression `foo.length` (where `foo` is an array of some type) is invariant with respect to a loop (and so can be factored out of the loop) if there are no assignments to `foo` within the loop body. That is, there are no statements of the form `foo = ...;` Assignments to elements of `foo` (like `foo[i] = ...;`) are not a problem as they do not affect the length of the array.

For example, Figure 3.1 shows a loop where the expression `foo.length` is invariant and so can be factored out.

```
int[] foo = new int[10];
for (int i = 0; i < foo.length; ++i) {
    foo[i] = i;
}
```

Figure 3.1: A small loop, where the expression `foo.length` is invariant.

3.2 Annotating pure methods

We annotate functionally pure methods with the annotation `@Pure`. We also introduce the `@Const` annotation for methods which are not pure but nevertheless do not change the state of their target object. `@Const` is based on the `const` attribute of member functions in C++. Note that all `@Pure` methods also meet the condition for `@Const`: `@Const` is a weaker restriction.

Our idea of an object's state here is everything about it that a `@Pure` method might depend on. That is, a method annotated `@Const` or `@Pure` must not change anything that might cause another `@Pure` method to return a different value. The simplest way to ensure this is for the method not to change anything at all to do with the object or other objects used in its internal representation, but we can be a little more flexible and allow such methods to change certain things (such as cached results of computations, for example) as long as such changes will not affect the behaviour of `@Pure` methods called on the object or with the object as an argument.

As an example, we revisit the example class first given in Figure 2.2. In Figure 3.2 we annotate this class with our new annotations. We noted in section 2.2 that only `mult` is functionally pure. We can however say of the `multk` method that it does not change the state of the object, and so it can be annotated as `@Const`. This comes in handy later on, when we want to know whether expressions in a loop might modify a particular object. We have also annotated the class as `@Encapsulated`; this annotation relates to aliasing and is explained later on, in section 3.3.2.

```
public @Encapsulated class Example {
    private int x;
    public static int k;

    public @Pure int mult(int y) {
        return x * y;
    }

    public set(int a) {
        x = a;
    }

    public @Const int multk(int y) {
        return x * y * k;
    }
}
```

Figure 3.2: A class with several methods, annotated appropriately.

All 5 annotations which we introduce, including `@Pure` and `@Const`, are summarised in section 3.4.

3.2.1 Discussion

By this approach of annotating pure methods, we divide the problem up into parts that can be solved separately. One part is to make some useful optimisations using this information about functional purity, the other is to derive the information in the first place. In this project we tackle the first part with our compiler stage while leaving the second to the programmer, but future projects could look at ways to derive this information automatically. Once such information is known and the methods are annotated, this knowledge can also be used for other purposes. For example, we mentioned in section 2.3.1 that pure methods may be used in specifications; our annotations could be re-used in checking that such specifications are legal.

3.3 Aliasing

`Array.length` is a special case because it does not depend on any state of the array that can change once it is constructed. In general, however, this is not the case: method calls and field values depend on the state of the objects in question, and this state can be modified by other methods in the program. A difficulty arises when this modification is made indirectly, and the possibility of this is called *aliasing*.

```
public class TestCollection {
    private int size = 0;

    public @Pure int size() {
        return size;
    }

    public @Const Object get() {
        return new Object();
    }

    public void add() {
        ++size;
    }

    public @Const void print() {
        System.out.println("Collection size = " + size);
    }
}
```

Figure 3.3: `TestCollection` class, to be used in the following examples

We will give a number of examples as we explain aliasing in this section; all these examples will use the test collection class `TestCollection` defined in Figure 3.3. To start with, let us consider the easy case, where there is no aliasing. Figure 3.4 shows a simple case where `list.size()` can safely be moved out of the loop. Note that this and the examples that will follow are assumed to be within a method, so `list` is a local variable. Clearly the `list` object will not change within the loop, as there are no methods at all called within the loop, except for `list.size()` which we know to be pure. The list has been freshly created and its reference kept in a local variable, so there is no way that there can be aliases to it anywhere else.

```
TestCollection list = new TestCollection();

for (int i = 0; i < list.size(); ++i) {
}
```

Figure 3.4: Simple case that should be optimised

3.3.1 Two problems with aliasing

Aliasing is the situation where there are several different references (perhaps in local variables, or fields of some object) to the same object. This means that the object can potentially be modified via any of these references, which makes it harder to tell exactly when a given object might be modified.

There are two kinds of aliasing that can cause problems for us: aliases to objects which are directly used in a pure function call, and aliases to objects internal to such objects.

The problem goes like this: suppose we have a local variable `foo` of type `Bar`. This local variable points to some particular instance of the `Bar` class. We need to work out whether this object might change within the course of a loop being executed, so that we can know whether we can safely move expressions which depend on it out of the loop.

At first glance, this might seem fairly straightforward: we simply need to check that `foo` will continue to point to the same object, and that no methods are called on `foo` which will change the object in any way. The former can be checked by looking for assignments to `foo` within the loop body, while the latter can be checked by ensuring that only methods annotated as `@Pure` or `@Const` are called on `foo`. Figure 3.5, for example, shows a simple case where `list.size()` should *not* be moved. In this case the collection is modified within the loop by calling `list.add()`. There is no aliasing, so this is straightforward to detect.

```
TestCollection list = new TestCollection();

for (int i = 0; i < list.size(); ++i) {
    if (i % 2 == 0) {
        list.add();
    }
}
```

Figure 3.5: Simple case that should not be optimised, as the object is modified

However, this is not enough. There are still two ways by which the object in question could be changed.

The first way is if there is some other reference (an *alias*) to the `Bar` object, somewhere other than `foo`. Such a reference could be in another local variable (such as the example in Figure 3.6, where there is another variable called `alias` pointing to the same list as `list`), a field in the class being compiled, or some other class somewhere else in the program (like the example in Figure 3.7, where the `Changer` object has a reference to the list). The first case (in another local variable) could probably be detected with some dataflow analysis, but is still more difficult than in the absence of aliasing. The last case (in another class) is perhaps the worst, because it means that any method call at all could potentially modify the object in question.

The second way is similar but a little more subtle: there could be an alias somewhere to one of the objects which is used *inside the object in question* to represent its internal structure. That is to say, class `Bar` may have a field of type `Baz`, and there may be another reference

```
TestCollection list = new TestCollection();
TestCollection alias = list;

for (int i = 0; i < list.size(); ++i) {
    if (i % 2 == 0) {
        alias.add();
    }
}
```

Figure 3.6: Case that should not be optimised, as the object is modified via a local alias

```
TestCollection list = new TestCollection();
Changer changer = new Changer(list);

for (int i = 0; i < list.size(); ++i) {
    if (i % 2 == 0) {
        changer.change();
    }
}
```

(a) The example itself

```
class Changer {
    TestCollection list;

    public Changer(TestCollection list) {
        this.list = list;
    }

    public void change() {
        list.add();
    }
}
```

(b) A class used in the above example

Figure 3.7: More complicated case that should not be optimised, as the object is modified via an alias inside another class

to this `Baz` object somewhere else, through which it may be modified. Figure 3.8 gives an example of this type of aliasing situation. This is in a sense the reverse of the previous case, as the collection is changed directly but the pure method is called indirectly — looking at it a different way, the object on which the pure method is called (the `CollectionProxy` in our example) is not itself changed, but an object inside it (the `TestCollection`) is changed, which changes its behaviour. Again, this could potentially happen within any method call in the loop.

In both cases modifications could also be made in another thread, but this does not really make the situation any worse as the real problem is that whatever method is running in the other thread has a reference to the object we are worried about. If we can find some way to prevent aliases, then this will include aliases in other threads.

```
TestCollection list = new TestCollection();
CollectionProxy proxy = new CollectionProxy(list);

for (int i = 0; i < proxy.size(); ++i) {
    if (i % 2 == 0) {
        list.add();
    }
}
```

(a) The example itself

```
class CollectionProxy {
    private final TestCollection target;

    public CollectionProxy(TestCollection target) {
        this.target = target;
    }

    public @Pure int size() {
        return target.size();
    }

    public @Const Object get() {
        return target.get();
    }

    public void add() {
        target.add();
    }

    public @Const void print() {
        target.print();
    }
}
```

(b) A class used in the above example

Figure 3.8: Another case that should not be optimised, as the object inside the proxy is modified

3.3.2 Solution: More annotations

Detecting this sort of aliasing automatically is not possible with the usual Java compilation process, as the compiler only compiles one class at a time. Even if it were possible to analyse the entire program at once (being guaranteed that no more classes would be loaded at runtime), it would be very time-consuming and use a lot of memory. Aliasing is a hard problem. In fact, finding possible aliases in a programming language with conditionals, loops and dynamically allocated recursive data structures is undecidable [23, 24]. This means that we cannot in general detect aliasing automatically, although it may be possible in some special cases.

Instead, we introduce more annotations to allow the programmer to indicate when aliasing cannot occur and so it is safe for the compiler to make optimisations. These annotations could also be added by some future tool to detect special cases where the aliasing problem is easier.

To prevent the first aliasing problem described, we require the programmer to annotate local variables as `@Unique` to say that they are a unique reference to whatever object they point to. This means that there *cannot* be any aliases to it to cause problems. It also means that the method cannot pass references to the `@Unique` object out to non-pure methods, because doing so would allow the reference to escape and lose its uniqueness. Note that because `@Unique` references are local variables they are thread-local, and so safe from other modification even in the presence of concurrency.

For the second problem, we require the programmer to annotate classes as `@Encapsulated` to say that they are well-encapsulated: no other classes will have references to the objects which are referenced in the class's fields, and so those objects will only be changed by the object in question. This is similar to saying that all the class's fields are `@Unique`, except that we only apply `@Unique` to local variables, and in this case it is only aliases *outside* the class that are a problem: an `@Encapsulated` class may have multiple references to its contained objects, as long as they are all within the main object.

Again, this use of annotations breaks the problem up, so that the problem of finding potential aliases can be solved separately to our task of actually making our optimisations. They also offer some flexibility in the definitions. The programmer can choose to take a looser observational interpretation of these definitions: if there are other references to the objects but they will not actually be changed via these references, then the `@Unique` and `@Encapsulated` annotations can still be applied even though the conditions defined above are not strictly met, as it is still safe for the purposes of the optimisation being done.

Discussion

It should be noted at this point that these new annotations (`@Unique` and `@Encapsulated`) provide something a bit like an ownership type-system [25]. A full and verified ownership system (and immutability [26, 27], for that matter) might well be a better solution. [28] discusses different ideas of encapsulation and the application of ownership and other schemes to these. However, ownership and immutability are still very much current research topics and not mature enough for us to easily use in this project. In particular, JKit does not yet have support for any ownership system and implementing it would be beyond the scope of this project.

Uniqueness types are used in functional languages as a way to model side-effects by having a variable, which can only be referred to in one place at any given time, to represent the changing global state [29]. This is related to the idea of linear types. AliasJava's type system [30] includes a *unique* annotation for objects that are not shared, among other type annotations. It combines this idea of uniqueness with ownership for encapsulation.

3.4 Summary of annotations

Table 3.1 lists the 5 annotations which we have introduced in this project to allow the optimisations which we are about to describe to be made. Some of these (`@Pure`, `@Const`, `@Immutable`) are similar to or the same as other definitions used elsewhere and discussed in chapter 2. The remaining two annotations (`@Unique` and `@Encapsulated`) are somewhat more original, though still closely related to other ideas of ownership and encapsulation which we did not have time to fully investigate.

Annotation	Type	Semantics
@Pure	method	The method is side-effect free (does not alter any state), and functionally deterministic (its return value depends only on its parameters, including the receiver).
@Const	method	The method does not alter the state of its receiver object.
@Immutable	class	Objects of this class never change once they are constructed.
@Encapsulated	class	Objects of the class are well-encapsulated, in that their state cannot change except by calling their methods (not by external references to internal objects, for example).
@Unique	variable	There will be no other references to the object pointed to by the local variable.

Table 3.1: Annotations introduced

If these annotations are added to the program where appropriate then our optimisations may be able to apply; if not then our optimisations will not help much or at all but the program will still be compiled validly.

3.5 Moving pure method calls

We now come to the optimisation of pure method calls, which was the main goal of this project. In section 2.2 we introduced the idea of pure methods, and in section 3.2 we discussed our annotations of such methods. We also explained the problem of aliasing in section 3.3, and our solution to it by adding more annotations. We will now combine all that to consider under what circumstances a call to a pure method can safely be factored out of a loop.

3.5.1 Loop invariance

To be able to move a method call out of a loop, we must make sure that it will have no side-effects and that it will always return the same value. If the method is pure then this will be the case if we can ensure that it is always called with the same arguments. [1] When we say the same arguments, we mean (in the case of object types) that the arguments to the pure method must refer to the same objects each time, and that these objects must not be changed between calls. If this is the case, then functional determinism assures us that the method will always return the same value, and so will be loop invariant. Furthermore side-effect-freeness assures us that nothing else will be affected by moving the pure method call as it does not change any state.

[1] If the method is not pure but is side-effect-free, then we can move it if we can somehow check not just that its arguments do not change but that no other state that it may access changes within the loop either. This is much more difficult, especially in the presence of concurrency. We will not consider this.

Arguments here include both the usual arguments to the method, and the target object on which it is invoked. For example, in the method call `a.get(i)`, both `a` and `i` are considered arguments for our purposes. The exception to this is static methods, which are not called on an object and so have no `this` pointer. Note that pure static methods are no more allowed to read or modify non-final static fields (on their own class or otherwise) than any other pure methods. This is because static fields are essentially global variables.

If an argument is of a primitive type, then making sure that it does not change is a relatively simple task: we just check that the variables involved are not assigned to within the body of the loop. Compound expressions are handled by recursively checking all parts involved: for example `a * b + c` will be loop invariant if `a`, `b` and `c` are all invariant.

If, on the other hand, an argument is of an object type, then we must check not just that it has the same value in the sense that it is pointing to the same object, but also that the state of this object does not change. The simplest case for this is if the object is of an immutable class.

Immutable classes

An immutable class is one whose instances are guaranteed never to change once they are constructed. Such classes are often used for holding a few values together, such as a vector or matrix. In this special case it is easy to check that an object will not change within the body of a loop, as it will never change at all.

We ask programmers to annotate immutable classes with the annotation `@Immutable`. Figure 3.9 gives an example of such a class annotated appropriately.

Note that it is not necessary for all the objects within an immutable object to themselves be of immutable classes; it is enough that the immutable class does not allow them to be changed. In fact we can see that `@Immutable` implies `@Encapsulated`.

```
public @Immutable class Point {
    private final double x, y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public @Pure double getX() {
        return x;
    }

    public @Pure double getY() {
        return y;
    }
}
```

Figure 3.9: An example of an immutable class

A more general case

Although some classes may be immutable, there are generally many more on which this restriction cannot be placed. We still want to optimise pure method calls involving such

classes, so we need a more fine-grained way of detecting when objects can be guaranteed not to change.

As discussed in section 3.3, aliasing makes this difficult. In section 3.3.2 we introduced the `@Unique` and `@Encapsulated` annotations to allow us to avoid the problems of aliasing. If we know that an object will not be changed via aliases, then it can only be changed directly by calling methods on it, or assigning to its fields. For simplicity we will not talk about the possibility of assigning directly to fields rather than calling setter methods here; nevertheless both are considered in our implementation. They are equivalent in effect, so assigning to a public field on an object can be considered as a special sort of method call on the object.

We have two annotations for methods, introduced in section 3.2: `@Pure` and `@Const`. Both of these imply that the method in question will not modify the state of the object on which it is invoked. Thus, in the absence of aliases, we can be sure that an object will not change within a loop as long as only `@Pure` and `@Const` methods are called on it.

3.5.2 Summary of method call movement

Method calls (whether in statements or in branch conditions) will be moved out of loops when the following conditions are met, which are sufficient to guarantee that such movement can safely be made with changing the semantics of the program:

1. The method being called is pure, and
2. For each argument to the method (including the target of the method call, unless the method is static):
   (a) The *reference* or primitive value (e.g. variable) is invariant within the loop, and
   (b) Either:
      i. It is a primitive type, or
      ii. The class pointed to is immutable (annotated as `@Immutable`), or
      iii. The reference is unique (i.e. it is a local variable annotated as `@Unique`), there are no statements in the loop which change the object (that is, only `@Pure` and `@Const` methods are called on the object), and there can be nothing else to change objects within the object as the class is annotated as `@Encapsulated`.

Figure 3.10 gives an example of some method calls that can be moved out of a loop and some that cannot:

1. `foo(n)` can be moved out of the loop, as `ManyMovements.foo` is `@Pure`, while `n` is invariant in the loop and is a primitive type (`int`).
2. `bar.baz()` can be moved out of the loop, as `Bar.baz` is `@Pure`, `bar` is invariant and `@Unique`, its class (`Bar`) is `@Encapsulated`, and there are no calls to other methods on `bar` within the loop.
3. `foo(i)` *cannot* be moved out of the loop, because `i` is assigned within the loop (by the loop increment expression `++i`), and so is not invariant.
4. `fresh.baz()` *cannot* be moved out, because `fresh` is assigned within the loop.
5. `baa.legs()` can be moved out of the loop, because `Sheep.legs()` is `@Pure`, `baa` is invariant and `Sheep` is `@Immutable`.

```
public class ManyMovements {
    public static void main(String[] args) {
        int n = 42;
        @Unique Bar bar = new Bar();
        Sheep baa = new Sheep();

        for (int i = 0; i < 10; ++i) {
            int a = foo(n);
            int b = bar.baz();

            int c = foo(i);
            Bar fresh = new Bar();
            int d = fresh.baz();

            int e = baa.legs();
        }
    }

    private static @Pure int foo(int x) {
        return x * 2;
    }
}

@Encapsulated class Bar {
    public @Pure int baz() {
        return 1337;
    }
}

@Immutable class Sheep {
    public @Pure int legs() {
        return 4;
    }
}
```

Figure 3.10: An example of some method calls that can and cannot be moved: `foo(n)`, `bar.baz()` and `sheep.legs()` can be moved out of the loop, while `foo(i)` and `fresh.baz()` cannot be.

3.6 Chapter summary

In this chapter we started by looking at a simple case of array lengths and how we can move references to the `length` field of arrays out of loops under certain conditions. We then went on to describe our annotations for pure and const methods, with the aim in mind of being able to move these pure methods out of loops similarly. We then explained the problem of aliasing, and how it makes it difficult for us to know when objects may change within a loop. To allow us to determine that certain objects would not change within a given loop we introduced a number of annotations to describe particular guarantees that can be made about certain classes and local variables. With this information in hand we were able to find situations where aliasing would not be a problem, and so we could focus on the problem of when exactly certain objects can be known not to change within the context of a loop. Combining this with the information about method purity provided by our first annotations we were able to specify conditions under which method calls can safely be factored out of loops.

Chapter 4

Implementation in JKit

The optimisations developed for this project and described in chapter 3 were implemented as a stage for JKit. JKit was introduced in section 2.5.

4.1 Architecture

A JKit stage called `LoopInvariantMovement` was written to implement these optimisations. JKit runs each method that it compiles through this stage for it to make any desired transformations to the flow graph.

For each loop in the method, it looks for expressions which can be factored out of the loop. There are two places in which it must do this: in statements on vertices of the flow graph (which JKit calls points), and on edges of the flow graph, which represent conditions for looping and conditional constructs. Each suitable expression is then replaced with a new local variable, and new points are added to the flow graph just before the start of the loop to initialise these new variables with their respective expressions.

The new variables are named like `$invariant`x`_`y, where x is the number of the loop in the method and y is the number of the expression factored. The $ sign ensures that the new variable does not have the same name as any existing variable, as Java syntax does not allow variable names to start with $.

Finding expressions to factor is done recursively, such that the largest possible subexpression will be factored. If an expression is not suitable to be factored, then its subexpressions are considered in the same way. For example, the expression `a.size() + b.size()` may not be invariant as a whole, but the subexpression `b.size()` might be and so it can be factored out of the loop.

To be factored out of the loop, an expression must both be invariant with respect to the loop, and *worth factoring*. The second condition is necessary so that trivial expressions (such as literals, for example) are not factored out of the loop: such expressions may well be invariant, but moving them out of the loop would make performance worse rather than better due to the overhead of the extra local variable.

An expression is considered worth factoring if it includes a method invocation, field dereference or array indexing operation. This condition might be worth revisiting to consider including more expressions which are expensive enough that moving them out of loops would improve performance.

An expression is *invariant* with respect to a loop if it is one of:

1. A literal (like `42` or `"xyzzy"`).

2. A local variable which is not assigned to within the loop.

3. An application of one of the built-in operators (unary, binary or ternary operators, cast, `instanceof`) with invariant operands.

4. A dereference of the `length` field on an invariant array.

5. A dereference of any field on an object reference which is not only an invariant reference, but points to an object which does not change during the loop (explained in the next paragraph).

6. A pure method call, where all the arguments (including the target object) are invariant references to objects which do not change during the loop (or are primitive types).

7. An array indexing expression where the array is an invariant reference to an unchanging array, and the index is invariant.

These conditions are checked by the `LoopInvariantMovement.isInvariantReference` method.

Several of these conditions require checking that a particular object, to which a reference refers, does not change within the scope of the loop. This is the situation discussed in section 3.5.1. Firstly, the reference must be invariant as defined above. Then it must also be either:

1. A primitive type (not actually a reference, just a value).

2. A reference to an object of a class which is annotated `@Immutable`.

3. A unique reference (i.e. a local variable annotated as `@Unique`) to an object, where there are no statements in the loop which change the object (that is, only `@Pure` and `@Const` methods are called on the object), and there can be nothing to change objects within the object as the class is annotated `@Encapsulated`.

4. A unique reference to an array, where there are no statements in the loop which change the array (that is, there are no assignments to elements of the array).

This is checked by the `LoopInvariantMovement.isInvariantReferenceToUnchangingObject` method.

If two or more loops are nested, then we consider them in order from the outside in, with the innermost loop being considered last. This allows us to factor expressions out of as many levels of loops as possible. From the point of view of the optimiser a loop is just a set of statements (and edge expressions) with a distinguished head node. All statements within the inner loop are also within the outer loop, so no special cases are needed to deal with nested loops. For example, in Figure 4.1 we have two nested loops. The expression `test.length` is invariant with respect to not just the inner loop but also the outer loop, so it can be factored out of both and evaluated once outside the outer loop.

4.2 Adding support for annotations

When this project was begun, JKit did not have any support for Java annotations (described in section 2.4) — it parsed them, but then ignored them completely. It was therefore necessary to add support for annotations to JKit. This required extending the flow graph classes to allow annotations to be attached to methods, classes and local variables, and then extending the `JavaFileReader` to take annotations from the AST and store them in the flow graph. The optimisation stages such as ours can then read the annotations from the flow graph as they need.

```
public class NestedFactoring {
   public static void main(String[] args) {
      int[] test = new int[10];

      for (int i = 0; i < 10; ++i) {
         for (int j = 0; j < 10; ++j) {
            System.out.println(test.length);
         }
      }
   }
}
```

Figure 4.1: `NestedFactoring` example program: The expression `test.length` can be factored out of both loops.

I have added only basic support for annotations on methods, classes and local variables. The way I have done this is not very general, it does not support annotations with parameters, and it does not check that the annotations used have been declared. Nor does it support reading and writing annotations from and to class files, so it is necessary to compile all annotated source files at the same time for their annotations to be able to be used for our optimisations. It will need to be extended or rewritten at some point, but it is perfectly sufficient for what I needed to do for this project.

4.3 Marking loop bodies

Optimisation stages in JKit work on the flow graph form which JKit uses as its intermediate representation. While this in many ways makes it easier to analyse the structure of programs, it does make it a lot harder to reconstruct high-level information about control structures such as loops. To solve this, the part of JKit that generates the flow graph from the AST was modified to save information about each loop (which would otherwise be thrown away) in a *region*. Regions are a concept that already existed in JKit, but were previously only used for storing information about exception handling. I created a new type of region to be used to store this information about loops.

Each region stores the set of all points that are part of the loop, and the distinguished point which is the entry point of the loop. This information is needed to be able to scan through loops to find expressions to factor, statements that might modify certain objects, and so on.

Figure 4.2 illustrates the two loop regions that would be generated for the small program with nested loops shown in Figure 4.1. Note that the region for the inner loop is a subset of the region for the outer loop.

Due to time constraints we only implemented the creation of loop regions for `for` and `while` loops, but it would be straightforward to add support for `do-while` and `foreach` loops too.

4.4 Flow graph visualisation

As we mentioned in section 2.5, JKit uses flow graphs for its intermediate representation of each method, and so this is what the optimisation stage for this project manipulates and transforms. For testing and debugging it is useful to be able to see exactly what these flow

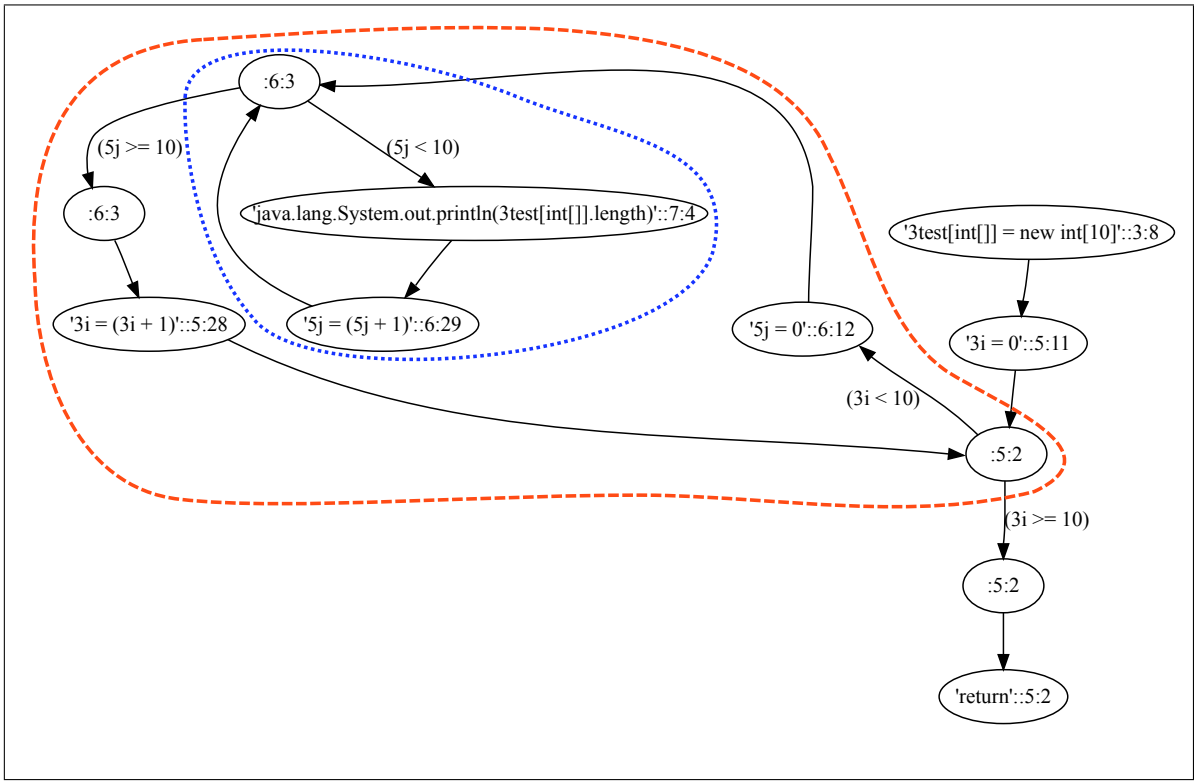


Figure 4.2: Flowgraph for the `NestedFactoring` example. The loop regions corresponding to each of the two nested `for` loops are circled with dotted lines. One of these regions is a subset of the other, corresponding to the nesting of the loops.

graphs are and how the optimisation stage changes them. Therefore, I wrote a small amount of code to output the flow graph from JKit to the `.dot` file format used by Graphviz[31]. I then rendered them with `dot` (from the Graphviz package), to produce diagrams such as Figures 2.5 and 4.2.

This is a fairly straightforward procedure; JKit's representation of the flow graph is simply dumped to the standard output stream in the appropriate format before and after running our optimisation stage on each method. Each point in the flow graph is written out as a vertex with its hashcode as id and the string representation of its contents as label. Edges are edges, with their condition (if any) as label. We did create a small (one line) shell script to strip out some of the unnecessary verbosity of JKit's string representation of expressions, but this was very much ad-hoc. Nonetheless, we found this visual representation of JKIL quite helpful in understanding how our optimisations were or were not working while we were developing them.

Chapter 5

Evaluation

We tested our optimisations firstly on a number of small benchmark programs written specifically for the purpose, and secondly on some larger existing programs to which we had to add annotations. Here we will discuss the details and results of these tests.

5.1 Benchmarks

Here we will discuss the test programs that we used, how we conducted our tests, and then our results.

5.1.1 Benchmark programs

Five simple programs were written to test the performance improvements provided by our optimisations in a number of situations which we expected to be amenable to such optimisation.

SimpleLoop

This program (shown in Figure 5.1) simply loops through a dummy collection, calling the `size()` method each time through the loop as part of the loop condition. It does not modify the collection (or do anything else) within the loop body, so these calls can be factored out of the loop. This leaves a truly trivial loop.

Our first version of this program did not use the `CollectionInterface` interface, but simply had `list` be of type `TestCollection`. In this case, however, the HotSpot VM's JIT compiler was apparently able to do some degree of optimisation which hid the effect of our loop factoring optimisation. We suspect that it was inlining[1] the `size()` method and then perhaps simplifying the loop. For it to be able to do this, however, it had to be able to statically determine which `size()` method was being called, which in turn required it to know at compile time the runtime class of `list`. We made the test program more difficult by creating the `CollectionInterface` interface (of which `TestCollection` is an implementation) and declaring `list` to be of type `CollectionInterface`. This prevents the JIT compiler from statically resolving the dynamic dispatch for the `size()` method call, and so prevents it from inlining the method (or whatever else it was doing).

[1] To *inline* a method call is to replace the call with the code of the method itself.

```
public class SimpleLoop {
    public static void main(String[] argv) {
        System.out.println(test() + " ms");
        System.out.println(test() + " ms");
        System.out.println(test() + " ms");
    }

    private static long test() {
        @Unique CollectionInterface list = new TestCollection(1000000000);

        long startTime = System.currentTimeMillis();
        for (int i = 0; i < list.size(); ++i) {
        }
        return System.currentTimeMillis() - startTime;
    }
}
```

Figure 5.1: SimpleLoop benchmark: Here the expression `list.size()` can be factored out of the loop, as it is a pure method and the list is not modified in the loop.

SimpleLoopArray

This program is shown in Figure 5.2. This is similar to the previous program, but uses an array rather than a collection class. Here the expression to be factored was `Array.length` rather than a method call, and so the speed improvement was less pronounced. Note that we do not need to know whether `array` is a unique reference, as the length of an array cannot change anyway.

```
public class SimpleLoopArray {
    public static void main(String[] argv) {
        System.out.println(test() + " ms");
        System.out.println(test() + " ms");
        System.out.println(test() + " ms");
    }

    private static long test() {
        char[] array = new char[500000000];

        long startTime = System.currentTimeMillis();
        for (int i = 0; i < array.length; ++i) {
        }
        return System.currentTimeMillis() - startTime;
    }
}
```

Figure 5.2: SimpleLoopArray benchmark: The expression `array.length` is invariant in the loop and can be moved out.

NestedProduct

This program (Figure 5.3) takes two `char` arrays of numbers (all 0 as it happens, but this should not matter to the optimisation), and finds the sum of all products of a number from

the first array with a number from the second. It does this with two nested loops, one for each array.

In this case not only can dereferences of `Array.length` (i.e. `a.length`, `b.length`) be factored out of both loops, but also the expression `a[i]` indexing the outer array can be moved out of the inner loop. However, to do this we *do* need to know that `a` is a unique reference, as — unlike the length of the array — elements of the array can be changed.

```
public class NestedProduct {
    public static void main(String[] argv) {
        System.out.println(test() + " ms");
        System.out.println(test() + " ms");
        System.out.println(test() + " ms");
    }

    private static long test() {
        @Unique char[] a = new char[10000];
        @Unique char[] b = new char[20000];

        int sum = 0;

        long startTime = System.currentTimeMillis();
        for (int i = 0; i < a.length; ++i) {
            for (int j = 0; j < b.length; ++j) {
                sum += a[i] * b[j];
            }
        }
        return System.currentTimeMillis() - startTime;
    }
}
```

Figure 5.3: NestedProduct benchmark: `a.length` and `b.length` can be factored out of both loops, while `a[i]` can be factored out of the inner loop.

NestedListProduct

This is the same as the previous program, except using our 'collection' class rather than arrays. See Figure 5.4. Similarly to the previous case, we can move `a.size()` and `b.size()` out of both loops, and `a.get(i)` out of the inner loop.

IntersectSquares

This program (Figure 5.5) constructs a number of rectangles with (pseudo-)random positions and sizes, then counts the number of times they intersect each other. For consistency the pseudo-random-number generator is seeded with a constant at the start of the program, so it always constructs the same rectangles. `Rectangle` is an `@Immutable` class with `@Pure` methods, implemented as the method names suggest.

In this case, rather like in the case of NestedProduct, some of the expressions indexing arrays and then calling a method on the result can be factored out of the inner loop. For example, `rectangles[i].getX()` can be factored out of the inner loop, as can `rectangles[i].getX() + rectangles[i].getWidth()` be. In all there are 16 expressions involving method calls and array indexing operations that can be factored out of one or other of the loops, and 6 involving array lengths.

```
public class NestedListProduct {
    public static void main(String[] argv) {
        System.out.println(test() + " ms");
        System.out.println(test() + " ms");
        System.out.println(test() + " ms");
    }

    private static long test() {
        @Unique TestCollection a = new TestCollection(10000);
        @Unique TestCollection b = new TestCollection(20000);

        int sum = 0;

        long startTime = System.currentTimeMillis();

        for (int i = 0; i < a.size(); ++i) {
            for (int j = 0; j < b.size(); ++j) {
                sum += a.get(i) * b.get(j);
            }
        }

        return System.currentTimeMillis() - startTime;
    }
}
```

Figure 5.4: NestedListProduct benchmark: `a.size()` and `b.size()` can be factored out of both loops, while `a.get(i)` can be factored out of the inner loop.

5.1.2 Methodology

Each test program was compiled with Sun's standard `javac` compiler, and with JKit using our optimisations. The resulting class files were then each run with Sun's HotSpot Java 6 VM [32] and with the Kaffe JVM [33]. In particular, this version of HotSpot:

```
andrew@rise:test cases$ java -version
java version "1.6.0_03-p3"
Java(TM) SE Runtime Environment (build 1.6.0_03-p3-mark_07_feb_2008_10_42-b00)
Java HotSpot(TM) Client VM (build 1.6.0_03-p3-mark_07_feb_2008_10_42-b00, mixed mode)
```

And this version of Kaffe:

```
andrew@rimu:~/COMP489/test cases$ kaffe -version
java full version "kaffe-1.4.2"

kaffe VM "1.1.8"
[...]
Engine: Just-in-time v3   Version: 1.1.8   Java Version: 1.4
Heap defaults: minimum size: 5 MB, maximum size: unlimited
Stack default size: 256 KB
```

Although HotSpot is a widely-used VM on desktop computers, mobile devices such as cellphones must run considerably lighter-weight VMs due to memory restrictions and architecture support. Such VMs will likely not have as much ability to optimise code in their JIT compiler (if they even do JIT). We chose Kaffe as a representative of this situation:

```
public class IntersectSquares {
   public static void main(String[] argv) {
      System.out.println(test() + " ms");
      System.out.println(test() + " ms");
      System.out.println(test() + " ms");
   }

   private static long test() {
      final int NUM_RECTS = 100000;
      final int WIDTH = 5000;
      final int HEIGHT = 30000;
      final int MAX_WIDTH = 10;
      final int MAX_HEIGHT = 10;

      final Random rand = new Random(123);

      long startTime = System.currentTimeMillis();

      //Generate random rectangles
      @Unique Rectangle[] rectangles = new Rectangle[NUM_RECTS];
      for (int i = 0; i < rectangles.length; ++i) {
         rectangles[i] = new Rectangle(rand.nextInt(WIDTH),
          rand.nextInt(HEIGHT), rand.nextInt(MAX_WIDTH),
          rand.nextInt(MAX_HEIGHT));
      }

      //Count intersections
      int intersections = 0;
      for (int i = 0; i < rectangles.length; ++i) {
         for (int j = i + 1; j < rectangles.length; ++j) {
            if (
              ((rectangles[j].getX() >= rectangles[i].getX() &&
                  rectangles[j].getX() <= rectangles[i].getX() + rectangles[i].getWidth())
                || (rectangles[i].getX() >= rectangles[j].getX() &&
                  rectangles[i].getX() <= rectangles[j].getX() + rectangles[j].getWidth()))
              && ((rectangles[j].getY() >= rectangles[i].getY() &&
                  rectangles[j].getY() <= rectangles[i].getY() + rectangles[i].getHeight())
                || (rectangles[i].getY() >= rectangles[j].getY() &&
                  rectangles[i].getY() <= rectangles[j].getY() + rectangles[j].getHeight()))
              ) {
                ++intersections;
            }
         }
      }

      System.out.println(intersections + " intersections");

      return System.currentTimeMillis() - startTime;
   }
}
```

Figure 5.5: IntersectSquares benchmark

although it is not widely used on such devices, it does do less optimisation than HotSpot, and so should benefit more from our compile-time optimisations.

The runs with HotSpot were made on a 2.8 GHz Pentium D with 2 GiB of RAM running NetBSD. Kaffe was not available on this machine, so the tests with Kaffe were made on a 1.6 GHz Celeron M with 2 GiB of RAM running Linux. It is not our intention to compare performance between HotSpot and Kaffe, but rather to look at the relative performance of the optimised and unoptimised versions of the test programs in each of these JVMs.

The test programs were written to run their respective tests 3 times and report times for each run. Each test program was run twice with each combination of compiler and JVM, giving a total of 6 times for each. These 6 measurements were then averaged to get the results shown in the next section.

5.1.3 Results

Figure 5.6 and Figure 5.7 show the running times for each benchmark with and without our optimisation, with each VM. In many cases the improvement is more pronounced when running with Kaffe, as expected due to Kaffe's lesser degree of optimisation in its JIT compiler. The average improvement over the 5 programs was a 26% decrease in running time for HotSpot and 38% for Kaffe.

The improvement is most striking for the `SimpleLoop` test, as the loop here really does nothing but call the `size()` method that is factored. There is still a noticeable improvement in the other cases, except for SimpleLoopArray with Kaffe.

Kaffe did not manage to run all the tests all the time. On SimpleLoopArray it would use a lot of memory (significantly more than HotSpot), and consequently swap a lot. It would also segfault in some cases. We suspect this is because its garbage collector is not as good as HotSpot's. SimpleLoopArray does allocate a 500 MB array for each test run, and if this were not garbage collected before the next run through then memory usage would increase a lot. We think that this is responsible for skewing the measurements and making the optimised version of SimpleLoopArray slower to run with Kaffe than the unoptimised version.

It was also necessary to use a simpler version of IntersectSquares with Kaffe, as Kaffe was much slower than HotSpot to run this test (around 15 times slower on the unoptimised version). In particular, it was reduced to 10000 rectangles (rather than 100000) and a height of 5000 was used rather than 30000 so as to have a reasonable number of rectangles still intersect. 3290 rectangles were found to intersect in the original version, and 205 in the version used with Kaffe.

5.2 Case studies

To get a better idea of how much our optimisations would improve the performance of real programs, we tested them with two programs written by other people. Because it is necessary to add annotations to the programs for our optimisations to work, and because adding these annotations requires the programmer to understand how the program works, it was necessary for us to choose fairly small programs. We chose two, which we will describe below. In both cases we added our 5 annotations from section 3.4 where appropriate, and also made some minor refactoring in some cases to make the program more amenable to optimisation.

Unfortunately, the results were not as good as we had hoped.

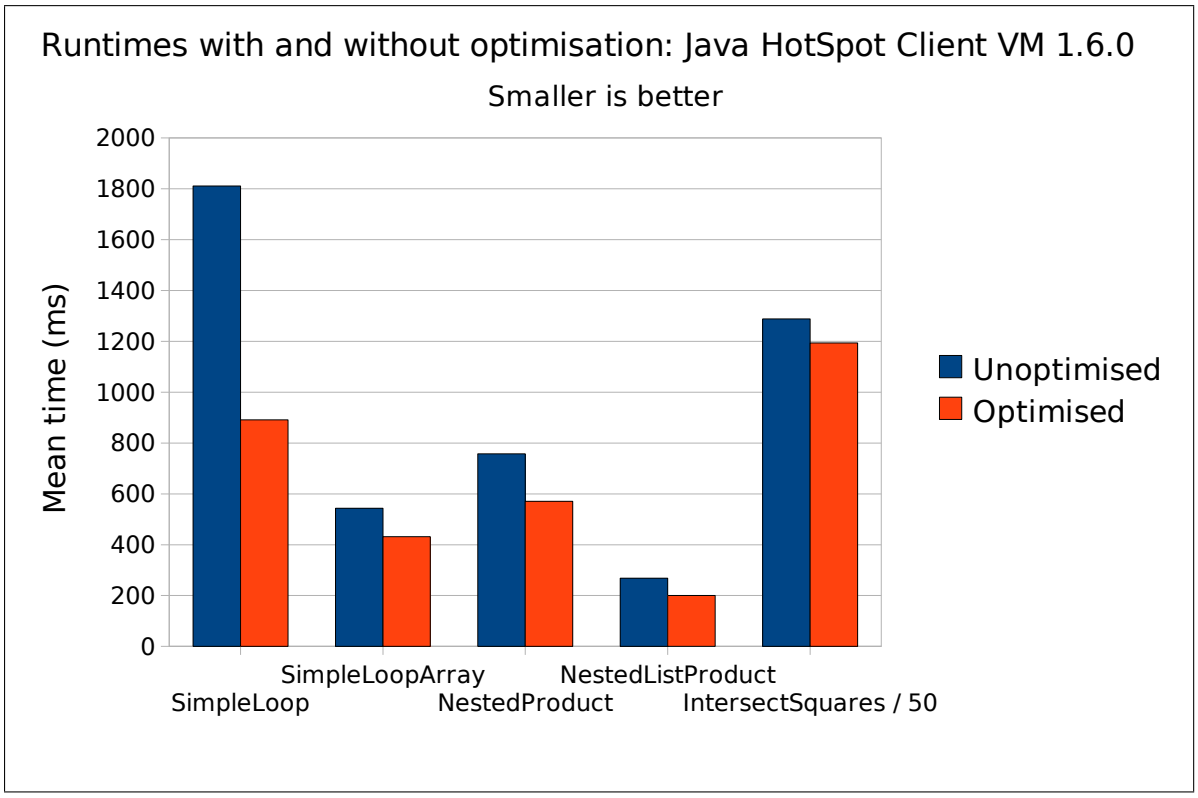


Figure 5.6: Run times with Sun HotSpot VM. The times for IntersectSquares were divided by 50 so as to fit on the graph.

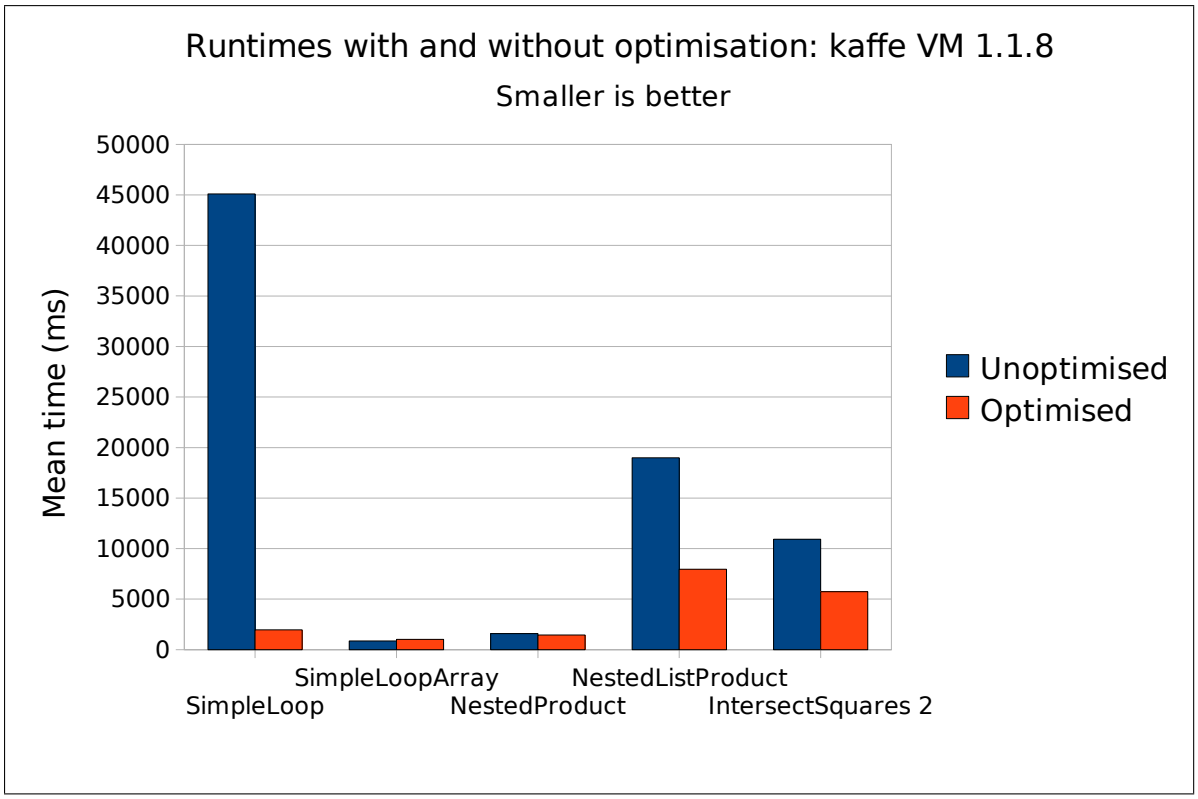


Figure 5.7: Run times with Kaffe VM. A different version of IntersectSquares was used here, see the text.

5.2.1 SimpleLisp interpreter

SimpleLisp [34] is a simple LISP interpreter written by David Pearce. We only considered the interpreter itself (2086 lines of code in 20 source files, with a total of 31 classes) and not the GUI, as it is easier and more accurate to make performance measurements when there is no user interaction to interfere with the measurement process. We only added annotations and did not otherwise refactor the program. We wrote a test program in LISP with which to test the interpreter; this program simply adds a large number of integers. We also tested the interpreter with the Fibonacci number program provided with it, though with the input hardcoded rather than read from the user so as not to interfere with timing.

Once appropriately annotated, there were a reasonable number of places where our optimisations were able to be applied: 30 expressions involving field dereferences (16 of `Array.length`, but also 14 involving other fields) were factored out of loops, and 8 involving method calls. All of these method calls were to `LispList.size()`.

However, despite all these changes made to the program, there was no noticeable speedup when running our test LISP programs. This seems to be because the optimisations were made in parts of the interpreter which are not actually executed very often or at least do not take a very large proportion of execution time. This was true even for the addition test program, which was chosen because the arithmetic functions were one place where some minor optimisations were made. This is a little disappointing, but perhaps to be expected.

5.2.2 GeoffTrace

GeoffTrace is a simple raytracer (487 lines of code in 8 source files, making 8 classes) written in Java by Geoffrey Spurr and provided to us as a potentially suitable program with which to test our optimisations.

Unfortunately, due to a number of bugs in JKit (involving left associativity of arithmetic operators, the use of `continue` statements in foreach loops, and abstract methods) it was necessary to rewrite parts of the program to avoid these operations before the program could even be compiled correctly by JKit. That done, we added our annotations and did a little refactoring of some classes to make them more functional in style and so more likely to be open to loop factoring of pure methods.

After all this, however, no cases were found where our optimisations would apply. This is due partly to what the raytracer needs to do, and partly to the style in which it is written. It was not written in a very functional style, and it was already fairly well optimised manually, so there were no cases where our techniques could make improvements.

While this outcome was disappointing, it did serve to emphasise that our optimisation is most applicable for programs written in a fairly functional style. Such a style is also likely to be easier to read and reason about, so encouraging it is worthwhile both in mind of our optimisation techniques and besides.

Chapter 6

Conclusions

We now conclude by summarising the contributions of this project and considering possible directions for future work.

6.1 Contributions

The major contributions of this project were:

1. Defining when it is possible (while compiling Java programs) to move expressions containing the following out of loops:
   (a) Dereferences of `Array.length`
   (b) Pure method calls
   (c) Field dereferences, and array indexing expressions
2. Implementing these optimisations in JKit.
3. An experimental study of the effects of these optimisations on runtime performance of a number of small test programs with two different JVMs.
4. Study of the application and effects of the optimisations to two real programs.

There were also a number of minor contributions:

1. Implementing a simple system for rendering flow graphs automatically from JKit's internal representation of Java methods, using Graphviz[31], for understanding and debugging the code movement performed by our optimisations.
2. Implementing (limited) support for method, class and variable annotations in JKit.
3. Implementing loop regions in JKit to keep track of loops when moving from the AST to the JKIL flow graph.

6.1.1 Limitations

For simplicity we ignored a few minor issues in our design and implementation. These would need to be done for a full practical implementation of our techniques, but should not be particularly difficult:

- JKit should load and save annotations from and to class files, rather than only reading them from source files.

- Foreach loops and do-while loops should be supported in addition to for loops and while loops.

6.2 Future work

There are a number of ways forward from this project, both in extending our approach to other optimisation techniques than loop invariant code movement, and in looking at other ways to get the information needed to make our optimisations.

6.2.1 Common subexpression elimination

In this project we looked at expressions that are invariant within loops. Another similar optimisation technique is to find common subexpressions within larger expressions or blocks of code, and (assuming they all evaluate to the same value) factor them out. Just like loop invariant code motion, current implementations of common subexpression elimination are generally limited to fairly simple expressions such as built-in arithmetic operators. However, the same approach that we took to loop invariant code motion of method calls should be able to be applied here with only minor changes, building on existing well-known techniques for common subexpression elimination [35]. This would allow pure methods which are common subexpressions to be eliminated under appropriate conditions (very much like the conditions required for our optimisations in this project).

For example, Figure 6.1 shows a few lines of a program where `list.size()` is used several times but will always have the same value. We can optimise this (as shown in Figure 6.2) by only calling `list.size()` once, storing the result in a new local variable, and reusing this value where it is needed.

```
TestCollection list = ...

int a = list.size() * 3;
int b = f(list.size()) - g(list.size() + 2);
```

Figure 6.1: A small code snippet where `list.size()` is a common subexpression.

```
TestCollection list = ...

int listSize = list.size();
int a = listSize * 3;
int b = f(listSize) - g(listSize + 2);
```

Figure 6.2: The same code snippet, with the common subexpression eliminated.

6.2.2 Automatically deriving annotations

We used annotations in this project to separate the various problems of finding which methods are pure, where aliases can and cannot occur and so on from our main task of making loop optimisations using this information. We (as the programmer of the various programs with which we tested our optimisations) added these annotations manually. It would be nice for this process to be automated.

Unfortunately, this is not possible in general: we mentioned in section 3.3.2 that the aliasing problem is undecidable, and even for some of the other annotations which are not undecidable per se we generally do not have enough information at compile time to be able to infer them completely automatically with no input from the programmer. Part of the problem is that Java programs may be compiled in separate parts without information available about all classes that will be used at runtime. Because subclasses may be used in place of their superclasses this means that we cannot necessarily predict the behaviour of an object we are passed just by looking at how its class is written: we could find at runtime that we are given a subclass which we were not aware of when compiling the method that uses the object. The exception to this is final classes, which are guaranteed not to have subclasses.

However, despite all these limitations, it may well be possible to automatically derive some of our annotations in some special cases, and to provide some level of checking that annotations provided by the programmer are correct (or at least possibly correct). Subclassing, including the possibility of subclasses that are not known at compile time, makes this difficult. In the case of final classes, it should be possible and fairly simple to check whether methods can be annotated `@Const` by looking at what they do to fields on their object. Checking whether a method can be annotated `@Pure` would be a little harder, but still feasible: as well as meeting the requirements for `@Const` it would be necessary to make sure that it does not modify or read static fields anywhere nor call any methods which are not themselves `@Pure`. A final class can be annotated `@Immutable` at least in the case that all its fields are final and of primitive or immutable types.

It would also be a good idea to annotate `@Pure` and `@Const` methods in the standard libraries, probably through a combination of automatic inference and manual work.

This automated derivation of annotations would sacrifice the flexibility that is allowed by the programmer making the annotations, though it could be combined with manual annotation to regain some of this flexibility where needed.

6.2.3 Using immutable and ownership types

We noted in section 3.3.2 that ownership and immutable types might provide a better way to avoid aliasing problems. It would be worthwhile to investigate the different type systems available that provide support for ownership and immutability. The guarantees provided by such a type system may well be sufficient to identify situations where objects cannot be changed via aliases and so can be guaranteed not to change within the scope of a certain loop. The type system used in AliasJava [30] would also be worth further consideration.

If ownership and immutability become popular and widely used in real programs then our making use of them would require less extra work of the programmer than our current approach with our `@Unique`, `@Encapsulated` and `@Immutable` annotations, as programmers would already be providing the necessary information.

As yet, however, ownership and immutability are not used very widely, and it remains to be seen whether they are practical for use outside of academia.

Bibliography

[1] D. Pearce, "The Java compiler kit (JKit)." `http://homepages.mcs.vuw.ac.nz/~djp/jkit/`.

[2] K. D. Cooper and L. Torczon, *Engineering a Compiler*. San Francisco, California: Morgan Kaufmann, 2004.

[3] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java*. Cambridge, UK: Cambridge University Press, 2nd ed., 2002.

[4] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques, and Tools*. Reading, Mass: Addison-Wesley Pub. Co, 1986.

[5] C. N. Fischer and R. J. LeBlanc, *Crafting a Compiler*. Menlo Park, California: Benjamin/Cummings, 1988.

[6] S. Hammond and D. Lacey, "Loop transformations in the ahead-of-time optimization of Java bytecode," in *15th International Conference on Compiler Construction* (A. Mycroft and A. Zeller, eds.), Springer Verlag, 2006.

[7] M. Finifter, A. Mettler, N. Sastry, and D. Wagner, "Verifiable functional purity in Java," in *15th ACM Conference on Computer and Communication Security (CCS 2008)*, (Alexandria, Virginia, USA), ACM, Oct. 2008.

[8] G. Leavens, A. Baker, and C. Ruby, "JML: A notation for detailed design," *Kluwer International Series in Engineering and Computer Science*, pp. 175–188, 1999.

[9] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs, "JML: notations and tools supporting detailed design in Java," *OOPSLA 2000 Companion*, pp. 105–106, 2000.

[10] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# programming system: An overview," *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, vol. 3362/2005, pp. 49–69, 2005.

[11] Ádám Darvas and P. Müller, "Reasoning about method calls in interface specifications," *Journal of Object Technology*, vol. 05, pp. 59–85, June 2006.

[12] D. A. Naumann, "Observational purity and encapsulation," *FASE 2005*, 2005.

[13] K. R. M. Leino and P. Müller, "Verification of equivalent-results methods," Lecture Notes in Computer Science, Springer-Verlag, 2008. To appear.

[14] M. Barnett, B.-Y. Chang, R. DeLine, B. Jacobs, and K. Leino, "Boogie: A modular reusable verifier for object-oriented programs," *Formal Methods for Components and Objects*, vol. 4111/2006, pp. 364–387, 2006.

[15] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun, "99.44% pure: Useful abstractions in specifications," *FTfJP'2004*.

[16] J. Zhao, I. Rogers, C. Kirkham, and I. Watson, "Pure method analysis within Jikes RVM," in *Third International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS 2008)*, (Paphos (Cyprus)), July 2008.

[17] M. Cline, "[18] const correctness, C++ FAQ lite." `http://www.parashift.com/c++-faq-lite/const-correctness.html#faq-18.10`.

[18] "Function attributes - using the GNU compiler collection (GCC)." `http://gcc.gnu.org/onlinedocs/gcc-4.0.0/gcc/Function-Attributes.html`.

[19] R. Love, "With a little help from your compiler." `http://blog.rlove.org/2005/10/with-little-help-from-your-compiler.html`, Oct. 2005.

[20] M. Metcalf, "Fortran 95," *SIGPLAN Fortran Forum*, vol. 15, pp. 19–22, 1996.

[21] A. Buckley, D. Coward, and J. Bloch, "JSR 175: A metadata facility for the Java™ programming language." `http://www.jcp.org/en/jsr/detail?id=175`, Nov. 2006.

[22] "Annotations." `http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html`, 2004.

[23] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems*, vol. 1, pp. 323–337, Dec. 1992.

[24] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 1467–1471, Sept. 1994.

[25] D. G. Clarke, J. M. Potter, and J. Noble, "Ownership types for flexible alias protection," *SIGPLAN Notices*, vol. 33, pp. 48–64, Oct. 1998.

[26] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieżun, and M. D. Ernst, "Object and reference immutability using Java generics," in *ESEC/FSE*, (Dubrovnik, Croatia), pp. 75–84, ACM, Sept. 2007.

[27] J. Östlund, D. Clarke, B. Åkerblom, and T. Wrigstad, "Ownership, uniqueness and immutability," in *Aliasing, Confinement and Ownership in Object-oriented Programming*, (Berlin), pp. 19–29, Aug. 2007. In conjunction with ECOOP 2007.

[28] J. Noble, R. Biddle, E. Tempero, A. Potanin, and D. Clarke, "Towards a model of encapsulation," in *International Workshop on Aliasing, Confinement and Ownership (IWACO)*, (Darmstadt, Germany), pp. 73–87, July 2003. Held in conjunction with ECOOP 2003.

[29] E. de Vries, R. Plasmeijer, and D. M. Abrahamson, "Uniqueness typing simplified," in *Implementation and Application of Functional Languages, 19th International Symposium, IFL 2007* (O. Chitil, Z. Horváth, and V. Zsók, eds.), vol. 5083 of *Lecture Notes in Computer Science*, (Freiburg, Germany), pp. 201–218, Springer, 2007.

[30] J. Aldrich, V. Kostadinov, and C. Chambers, "Alias annotations for program understanding," in *ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, (Seattle, Washington, USA), pp. 311–330, ACM, 2002.

[31] "Graphviz." `http://www.graphviz.org/`.

[32] "Java SE HotSpot at a glance." http://java.sun.com/javase/technologies/hotspot/.

[33] "Kaffe.org." http://www.kaffe.org/.

[34] D. Pearce, "ENGR202 2008T1, software design." http://www.mcs.vuw.ac.nz/courses/ENGR202/2008T1/assessment/assignment4/index.shtml.

[35] J. Cocke, "Global common subexpression elimination," in *A symposium on compiler optimization*, (Urbana-Champaign, Illinois), pp. 20–24, July 1970.