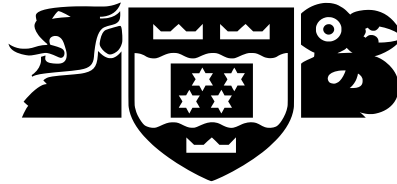
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



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: Milestone 2 report

Andrew Walbran

Supervisor: Associate Professor Lindsay Groves

July 20, 2008

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

Abstract

Programs in Java (and other programming languages) often contain expressions such as method calls that are called many times in a loop but will always have 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. An optimisation is presented to remove this inefficency by a form of loop invariant code movement. This document reviews progress on the second milestone of an Honours project dealing with implementing this optimisation in a Java compiler (namely JKit), based on the annotation of pure methods.

Acknowledgments

Thank you to my supervisor, Associate Professor Lindsay Groves, for agreeing to supervise this project and for support and advice. Thanks to Dr. David Pearce for the project idea, for his support and advice 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 meeting.

Contents

1 Introduction **1**
1.1 Pure functions . . . 1
1.2 Loop optimisation . . . 1
1.3 Implementation . . . 2

2 Achievements and problems **3**
2.1 Recap of first milestone . . . 3
2.2 Implementation of optimisation . . . 3
2.2.1 Achievements . . . 3
2.2.2 Problems . . . 4
2.3 Case studies for evaluation . . . 8
2.3.1 Rationale and requirements . . . 8
2.3.2 SimpleLisp . . . 9
2.3.3 Possible approaches to testing and evaluation . . . 9

3 Review of plans and future goals **11**
3.1 Milestone 2 . . . 11
3.2 Remainder of project . . . 11
3.3 Outline of final report . . . 11
3.4 Conclusion . . . 12

Figures

1.1 A class with several methods; only the mult method is pure by our definition. 2
1.2 Original and optimised code to loop through a collection 2

2.1 SimpleLoop.java, a simple case that can be optimised 5
2.2 Flow graph of SimpleLoop before optimisation 6
2.3 Flow graph of SimpleLoop after optimisation (changes are highlighted) . . . 7

Chapter 1

Introduction

1.1 Pure functions

Java compilers are limited in how they can optimise the use of methods in Java programs by the fact that methods can have side effects. For example a method that returns a value may also change the value of one of the object's attributes, or even some other state in an apparently unrelated part of the program. The behaviour of methods may also (in general) 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 some static field in an unrelated class.

In practice, however, many methods are *pure* functions which return a value depending only on the arguments passed to them (including the implicit `this` parameter), and do not alter any state. If programmers were to annotate these 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.

In the example class shown in Figure 1.1, the `mult` method is pure but the `set` method is not pure as it alters the state of the object. The `multk` method would not be considered 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. 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`.

1.2 Loop optimisation

Given information about which methods are pure, and a number of other annotations to do with aliasing, we can move certain invariant expressions out of loops. This factoring is possible when a pure method is called within a loop with arguments that can be guaranteed to remain the same across all iterations of the loop. This situation is fairly common, such as when checking the size of a collection in a loop condition.

For example, consider the code snippet in Figure 1.2a. If we assume that `MyCollection.size()` is a pure method by the definition above, that `MyCollection.get(int)` does not modify the collection, and that there are no aliases by which the collection may somehow be modified, then this code can safely be transformed into something like that in Figure 1.2b where `MyCollection.size()` is only called once rather than each time through the loop. As method calls are significantly more expensive than access to local variables, this version will be faster. We do need to make some careful checks for aliasing, which are explained in detail

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

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

(a) Calling the size method each time.

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

(b) Only calling the size method once.

Figure 1.2: Original and optimised code to loop through a collection

in section 2.2.2.

1.3 Implementation

This project implements the optimisations described in this report as stages for JKit[1], an experimental Java compiler developed at MCS and designed to be extensible for experiments such as this.

Chapter 2

Achievements and problems

2.1 Recap of first milestone

I completed the following items for the first milestone:

1. Reviewing the literature on pure functions (as well as other restrictions on functions) and compiler optimisation techniques, especially as relating to Java. Producing a bibliography.
2. Implementing a simple system for rendering flow graphs using Graphviz[2], for understanding and debugging the code movement performed by my optimisations.
3. As a test, a very limited proof-of-concept constant folding optimisation stage.
4. Implementing limited but sufficient support for method annotations in JKit.
5. Implementing the basic mechanism for moving method calls out of loops, and moving methods marked as `@moveable` as a test.
6. Moving dereferences of `Array.length` when the array reference is invariant.

These were described in more detail in my first milestone report[3].

2.2 Implementation of optimisation

In my proposal, I planned for the second milestone to have mostly completed implementing the optimisations in JKit, and to be ready to start testing them with some 'real' programs.

I have been able to do this, though not quite as nicely as I had originally hoped. In particular, the programmer is required to make quite a few annotations of methods, classes and local variables for the optimisation stage to be able to safely determine which expressions to factor out of loops. This requires quite a bit of work before any optimisation can occur. I had planned to make the process more automatic, but it proved more difficult than I had realised, due primarily to the aliasing problem.

2.2.1 Achievements

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

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
   (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 (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 to change objects within the object as the class is annotated as `@Encapsulated`.

For example, in the program given in Figure 2.1, the calls to `list.size()` in `SimpleLoop.main` can safely be factored out of the loop, as `TestCollection.size()` is a pure method, the `list` variable is not assigned within the loop, `list` is `@Unique`, `TestCollection` is marked as `@Encapsulated`, and only `@Pure` or `@Const` methods (`print()`, `size()` and `get()`) are called on `list` within the loop.

Figure 2.2 shows the flow graph of the main method of this program before optimisation. Figure 2.3 shows the result after the optimisation is run, with the calls to `list.size()` correctly being moved out of the loop.

2.2.2 Problems

Aliasing

The problems with aliasing were discussed in my first milestone report.

There are two kinds of aliasing that can cause problems: 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`.

However, this is not enough. There are still two ways by which the object in question could be changed. The first 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, a field in the class being compiled, or some other class somewhere else in the program. The last case 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 to this `Baz` object somewhere else, through which it may be modified. Again, this could potentially happen within any method call in the loop. In both cases modifications could also be made in another thread.

To prevent the first aliasing problem, we require the programmer to annotate local variables as `@Unique` to say that they are a unique reference to whatever object they point to. For the second problem, we require the programmer to annotate classes as `@Encapsulated`

```java
public class SimpleLoop {
    public static void main(String[] argv) {
        @Unique TestCollection list = new TestCollection();

        list.add();
        list.add();
        list.add();

        list.print();

        for (int i = 0; i < list.size(); ++i) {
            list.print();
            System.out.println(list.size());
            System.out.println(list.get());
        }

        list.print();
    }
}

/**
 * 'Collection' used to test loop optimisations
 */
@Encapsulated class TestCollection {
    private int size = 0;

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

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

    public void add() {
        ++size;
    }

    public int addSize() {
        return ++size;
    }

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

Figure 2.1: SimpleLoop.java, a simple case that can be optimised

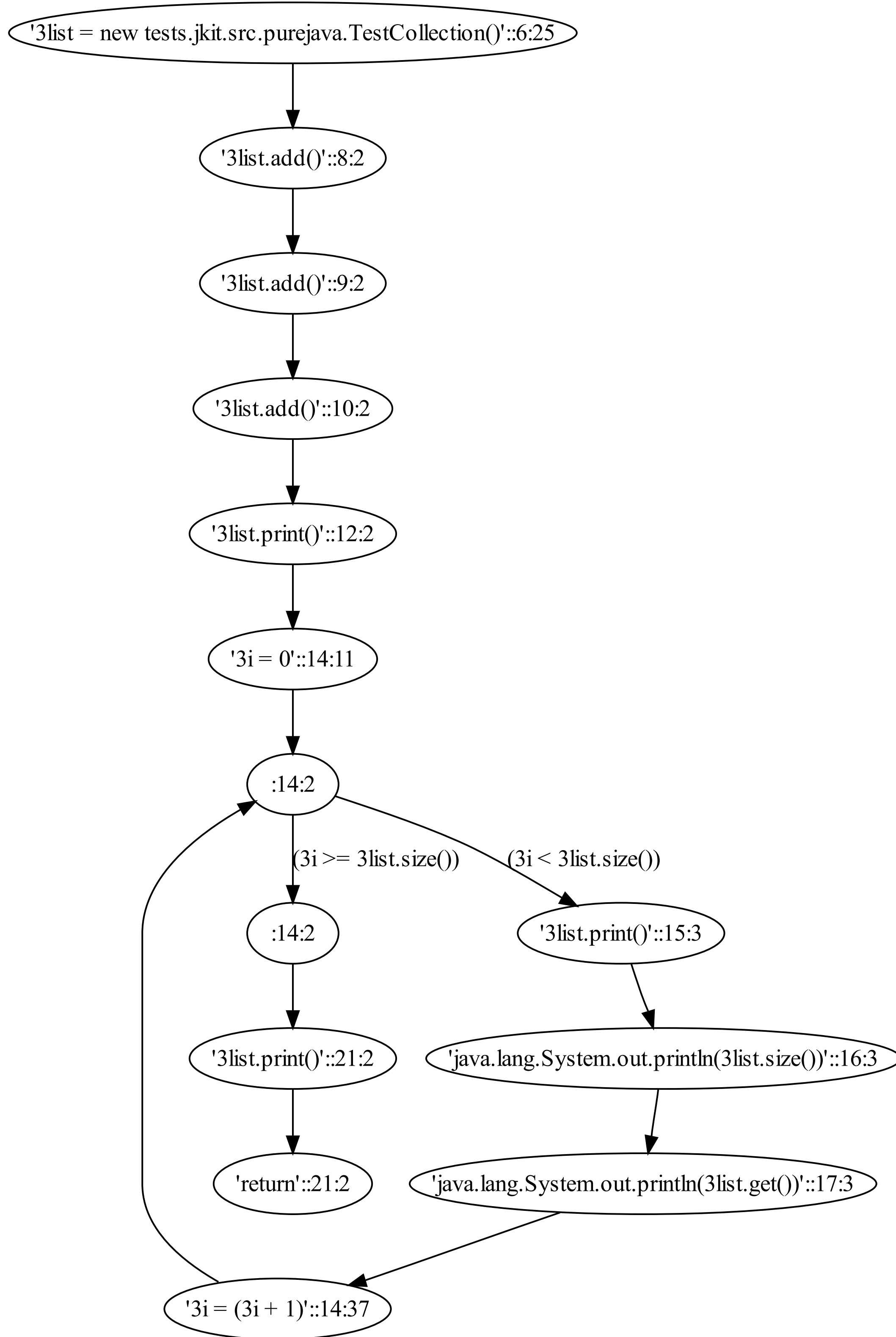


Figure 2.2: Flow graph of SimpleLoop before optimisation

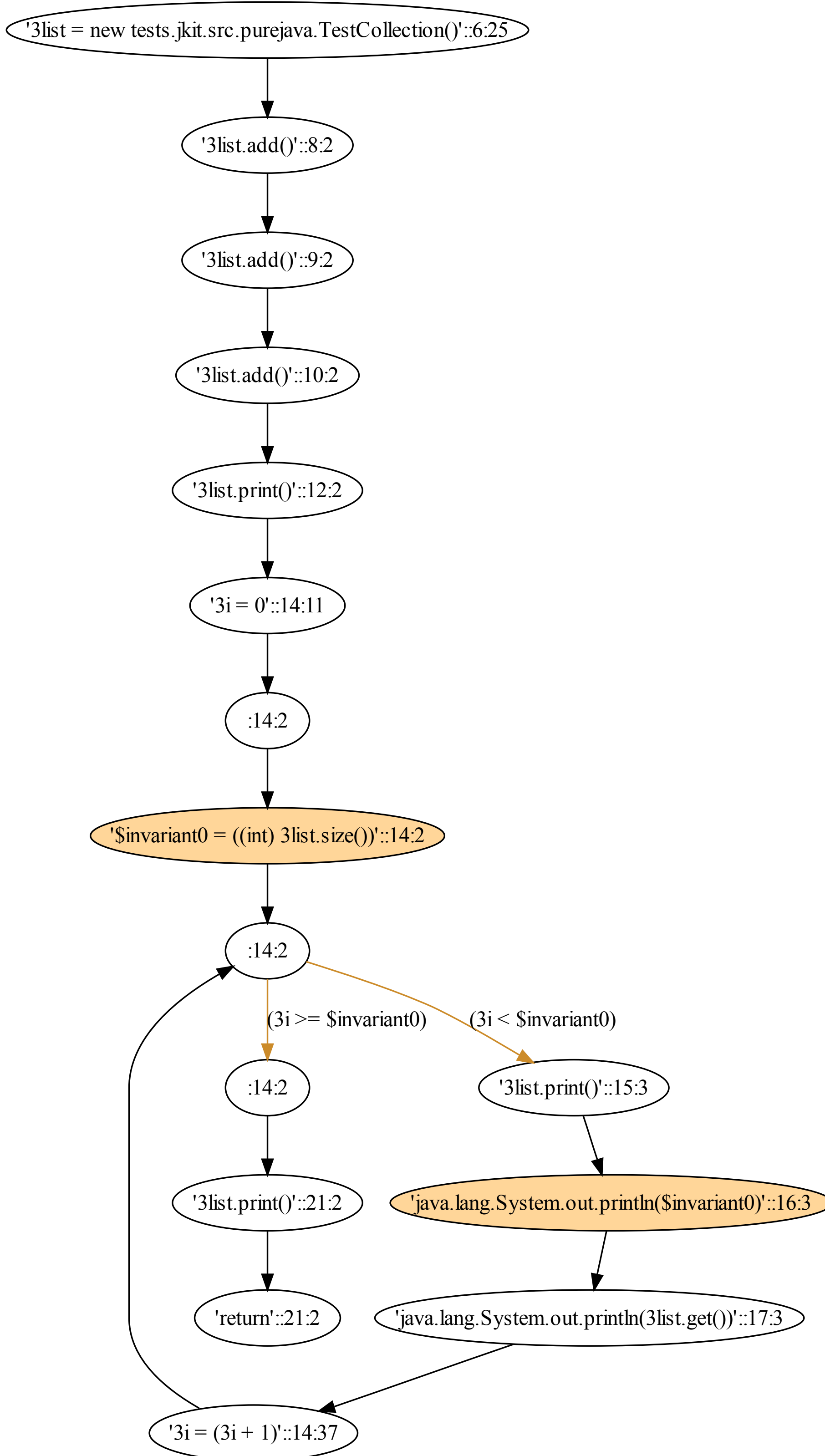


Figure 2.3: Flow graph of SimpleLoop after optimisation (changes are highlighted)

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 the objects will only be changed by the object in question.

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.

It should be noted at this point that these new annotations provide something rather like ownership. A full implementation of ownership would most likely be a better solution, but was judged impractical at this time as JKit does not yet have support for ownership and implementing it would be beyond the scope of this project.

Statements and edge expressions in JKit

JKit separates expressions into two different types: normal statements (such as assignment statements, method calls etc. on their own), and the conditional expressions used in constructs such as `if`, `switch` and loops. The former are represented in the flow graph as statements in points (vertices), whereas the latter are stored on the edges between points.

At first (and as at milestone 1), I was only considering expressions inside points in the flowgraph. This fails to catch many useful cases, such as loop conditions. Loop conditions are potentially a common case: for example, the loop in Figure 2.1 calls `list.size()` within its loop conditions, and this call should be factored out of the loop. Furthermore, it is in fact necessary to consider expressions on such conditional edges for correctness, as there could be a call to a non-pure non-const method within the expression that will modify one of the objects about which we are concerned.

Note that there are two purposes for which statements and expressions within the loop must be checked: firstly when looking for expressions (such as pure function calls) which might be able to be factored out of the loop, and secondly when checking for expressions that might change an object. The former is needed if we are to find all such expressions and so do as much optimisation as we can, while the latter is required for correctness: if we miss such expressions then we may perform unsafe optimisations which (incorrectly) change the behaviour of the program.

I now check conditional edge expressions as well as statements on points. Doing so required filtering the edges in the flow graph to select those which which have their source within the set of points which make up the loop body region.

At first I considered only edges having both their source and their destination within the set of points which make up the loop body region. On further consideration this criterion on edges showed not to be quite right, as we also want (for example) edges leading out of the loop from the loop header, as they are also evaluated on each iteration of the loop. Thus the correct criterion is simply to select all edges having their source within the set of points which make up the loop body region.

2.3 Case studies for evaluation

2.3.1 Rationale and requirements

To evaluate how useful the optimisations developed in this project are and how feasible they are for 'real-world' programs, it is necessary to develop a number of case studies of real programs on which to run tests. These programs must be appropriately annotated, both

correctly so that the optimisations are not applied in cases where it would in fact be unsafe to move code out of loops, and as fully as possible so that the optimisations are applied in as many cases as possible so as to achieve the maximum possible performance increase. Unfortunately it is non-trivial to do this, as it requires a reasonably detailed understanding of how the program in question works and how the different parts interact.

It may be possible to develop some tools to assist by automating part of the process, though it is not clear whether this would be feasible within the scope of this project. Such tools must either be given the entire source code to work with at once to do whole-program analysis, or leave the programmer to check whether the guarantees made can in fact be honoured in the presence of subclassing (for example). Either way, such automated tools seem inevitably to require rather stricter definitions of concepts such as encapsulation and uniqueness (for avoiding aliases) than the relatively loose observational view that can be taken by a programmer who understands the system as a whole.

2.3.2 SimpleLisp

SimpleLisp [4] is a Lisp interpreter written by David Pearce. It includes both the interpreter itself and a GUI editor. So far, I have used only the interpreter part as a case study. I have manually added appropriate annotations (`@Pure`, `@Const`, `@Immutable`, `@Encapsulated` and `@Unique`) to the code, and tested compiling it with JKit using the optimisations developed in this project. I found that there were 16 cases where references to `Array.length` could be moved out of loops, and 8 where the main optimisation of moving pure method calls applied.

2.3.3 Possible approaches to testing and evaluation

There are a number of ways in which these optimisations could be evaluated with a set of test programs: the number of times the optimisations can be applied can be measured, the total speed improvement for some typical run of the compiled program can be measured, or we can even try each possible application of the optimisation and measure the speed improvement individually. Most likely I will only collect statistics on the first two.

It might also be worth considering some way to check the correctness of the optimisation: that is to say, verifying that program behaviour is unchanged by the optimisations. How to do this comprehensively is unclear, though some degree of testing could be done by comparing program output on several given inputs, checking that the optimised and unoptimised versions give the same output in all cases.

Chapter 3

Review of plans and future goals

3.1 Milestone 2

For milestone 2, I planned to have mostly completed the optimisations in JKit, and be ready to start testing them with some sample programs. I have indeed done this: section 2.2 detailed how I have implemented the optimisations originally planned, and section 2.3 discussed how I have started to collect and adapt a number of programs with which to evaluate the optimisations.

3.2 Remainder of project

The main task I still have left to do is to evaluate my optimisations with a range of existing programs. This is discussed in section 2.3.3. It is a fairly large task as it requires manually annotating lots of existing code so that the optimisations can be applied to it, which requires understanding how the code works and how the different classes in the program interact. It also requires some thought as to how to measure the effect of my optimisations on the program.

There are a few minor changes I still wish to make to my implementation, mostly refactoring to make the code cleaner and generalisation to catch some more complicated cases.

As I mentioned in my milestone 1 report, there may still be scope for some simple optimisations other than loop invariant code motion to be made with the use of pure functions. For example, pure method calls where the return value is ignored can be removed completely.

I mentioned in my proposal the possibility of implementing checking of const / pure annotations, or even implementing functionality to derive pure and const annotations for existing code. I have now realised that this is a larger and more difficult task than I first thought it might be, and beyond the scope of this project. This could perhaps make a good Masters project. Having said that, it might be useful to have some limited, semi-automated version of this to use for annotating the programs that I will use to test the optimisations, as mentioned in section 2.3.1.

3.3 Outline of final report

The final report for this project will be strucured something like this:

1. Introduction
    (a) Motivation

(b) Contributions
(c) Outline

2. Background
   (a) Literature review for compiler optimisations and use of pure functions
   (b) Annotations in Java
   (c) Aliasing problems
   (d) JKit

3. Optimisations: loop invariant code movement for Java
   (a) `Array.length`
   (b) Pure method calls
      i. Immutable classes
      ii. A more general case

4. Implementation in JKit
   (a) Adding support for annotations
   (b) Challenges
   (c) Flow graph visualisation

5. Case studies for evaluation
   (a) SimpleLisp interpreter
   (b) JKit parser
   (c) *[another case study]*

6. Conclusions
   (a) Contributions
   (b) Future work

3.4 Conclusion

In summary, I have successfully completed the goals I set out for Milestone 2. In particular, I have implemented the main optimisation I planned, and started to prepare some case studies with which to test it. Unfortunately more work is required of the programmer than originally hoped, as it is necessary to annotate the program with lots of extra information to allow the compiler to determine when optimisations can safely be applied. There may be potential for this to be automated, but probably not within the scope of this project (at least not to a large extent). I still have to evaluate the optimisations further by testing them with several substantial existing programs.

Bibliography

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

[2] "Graphviz." http://www.graphviz.org/.

[3] A. Walbran, "Optimising Java programs with pure functions: Milestone 1 report." http://purejava.files.wordpress.com/2008/05/proj_report_outline.pdf, May 2008.

[4] D. Pearce, "ENGR202 2008T1, software design | MSCS | Victoria University of Wellington." http://www.mcs.vuw.ac.nz/courses/ENGR202/2008T1/assessment/assignment4/index.shtml.