Josh Treon

Optimising Reduce in Java

Computer Science Tripos, Part II

Trinity College, 2015

Proforma

Name:	**Josh Treon**
College:	**Trinity College**
Project Title:	**Optimising Reduce in Java**
Examination:	**Computer Science Tripos, Part II, 2015**
Word Count:	**10,387**
Project Originator:	Dr Arthur Norman
Supervisor:	Dr Arthur Norman

Original Aims

Improve the performance of Reduce when running in JLisp, a Java-coded Lisp environment. This is to be done by translating 'hotspot' Lisp-coded Reduce functions into Java, a strategy which produced positive results for CSL, a C-coded Lisp environment. A translator must be written in order to accomplish this, and the generated Java must be successfully interfaced with JLisp.

Work Completed

All of the original project goals have been met. Over 416,000 lines of working Java have been generated, which corresponds to more than 4,100 translated Reduce functions. This has resulted in a speedup of JLisp by a very significant factor.

Special Difficulties

None.

Declaration of Originality

I, Josh Treon of Trinity College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

1 Introduction **1**
- 1.1 Context and Motivation . . . 1
- 1.2 Challenges . . . 2
- 1.3 Outline of Tasks Required . . . 3
  - 1.3.1 Writing a Basic Translator . . . 3
  - 1.3.2 Interfacing the Generated Java with JLisp . . . 3
  - 1.3.3 Evaluating Correctness and Performance Change . . . 3
  - 1.3.4 Extension: Expand the Translator . . . 4

2 Preparation **5**
- 2.1 Background on Lisp . . . 5
  - 2.1.1 Lists . . . 6
  - 2.1.2 Special Forms . . . 7
  - 2.1.3 RLisp . . . 7
- 2.2 Background on JLisp . . . 8
  - 2.2.1 Representative Datatypes . . . 8
  - 2.2.2 Built-in Functions . . . 9
  - 2.2.3 JLisp Runtime Environment . . . 9
- 2.3 Resources . . . 9
  - 2.3.1 Reduce Test Scripts and Reference Logs . . . 9
  - 2.3.2 Lisp-to-C Translator . . . 10
  - 2.3.3 Profiling Information . . . 10
  - 2.3.4 Reduce User's Manual . . . 10
- 2.4 Programming Languages . . . 10
- 2.5 Development Environment . . . 11
  - 2.5.1 Machine Specifications . . . 11
  - 2.5.2 Version Control . . . 11
  - 2.5.3 Programming Environment . . . 11
- 2.6 Software Development Approach . . . 12
- 2.7 Initial Experience . . . 12

2.8 Summary 12

3 Implementation **13**
3.1 Structure of Java Output 13
3.2 Translator Design 15
3.3 Atomic Expressions 16
3.3.1 Literal and Variable Tables 16
3.4 Function Calls 17
3.4.1 The `initialiseMe` Method 17
3.4.2 Inlining Functions 19
3.5 Special Forms 20
3.5.1 Conditional Expressions 20
3.5.2 Boolean Predicates 20
3.5.3 Quoted Expressions 22
3.5.4 Sequential Execution 22
3.5.5 Fluid and Global Variables 24
3.5.6 Goto Expressions 25
3.5.7 Return 26
3.5.8 Unreachable Statements 28
3.5.9 Lambda Expressions 29
3.6 Interfacing the Generated Java with JLisp 29
3.6.1 Modifying the Startup Process 29
3.6.2 Identifying Translation Errors 30
3.7 Automating the Translation Process 31
3.7.1 Omitted Functions 31
3.7.2 Irretrievable Function Definitions 32
3.8 Improving the Readablility of the Generated Java 33
3.8.1 Renaming Variables and Literals 33
3.8.2 Name Sanitisation 33
3.9 Summary 34

4 Evaluation **35**
4.1 Translation Statistics 35
4.2 Correctness of Translated Code 36
4.2.1 Reduce Test Scripts 37
4.3 Performance Analysis 38
4.3.1 The Java JIT Compiler 38
4.3.2 Measurements Taken 38
4.3.3 Discussion of Results 39

5 Conclusions **45**

5.1 Further Work 45
5.2 Lessons Learned 46

A Example Translation: `reval1` 47
A.1 RLisp 47
A.2 Lisp 48
A.3 Java 48

B Project Proposal 59

Acknowledgements

I owe a huge portion of this project's success to Dr Arthur Norman. Without his endless capacity to provide expert help and advice at every stage, this dissertation would almost certainly still be an item on my todo list. Thank you!

Chapter 1

Introduction

This dissertation discusses the implementation and evaluation of optimising the performance of Reduce – a computer algebra system – by translating from Lisp to Java a selection of the Reduce source code. The project has been a huge success, meeting (and exceeding) the criteria specified in the project proposal. In total, over 4,100 Reduce functions have been translated and incorporated into the Reduce build, corresponding to more than 416,000 lines of generated Java. The optimised version of Reduce offers a very significant increase in performance over the original.

1.1 Context and Motivation

Reduce is a computer algebra system capable of advanced algebraic operations. Amongst many other capabilities, it can expand and factor polynomials, perform integration and differentiation, and is equipped to carry out many specific calculations for high energy physics. Its development began in the 1960s and in 2008 was made open source.

The source code for Reduce, a collection of functions comprising around 400,000 lines of code, is written in a Reduce-specific dialect: RLisp. This is essentially a high level syntax which is mapped onto Lisp, the list-based programming language.

Building a working version of Reduce involves compiling the Lisp function definitions into Lisp-specific bytecodes. At runtime, an interpreter decodes and executes these bytecodes in order to perform a specified operation. Two Lisp environments which implement the bytecode interpreter currently exist:

CSL (Codemist Standard Lisp) – written in C, and JLisp – written in Java. The production version of Reduce is run using CSL.

JLisp began as an experiment several decades after work first started on Reduce. It was apparent that a Java-based environment offered advantages over CSL, which have become even clearer in recent years:

1. Java is generally much simpler to interface with other systems than C; running Reduce on an Android device is a good such example.
2. The existing framework for Java offers better support for building possible extensions to Reduce. These could include development of a GUI, experimenting with concurrent use and implementing features for networking.
3. A Java version of Reduce can be converted to Javascript and used in a web browser.

CSL has been extensively optimised and refined since its first implementation. The main source of improvement has been through translation of function definitions from Lisp directly into C – such functions are executed from their C-coded definitions instead of being interpreted via their representative bytecodes. This has resulted in a performance increase by a factor of 6-7. Unfortunately, JLisp currently runs very slowly in comparison with CSL – so slowly, in fact, that some initial benchmarks suggest that the unoptimised and optimised versions of CSL are respectively around 8 and 60 times faster than JLisp!

This project explores the hypothesis that an attempt at optimising JLisp in the same way that was done for CSL will also produce an increase in performance. The expected level of this increase is uncertain, due both to the differences between C and Java as languages and to the results achievable in the time available.

1.2 Challenges

Although this project has the potential to deliver rewarding results, its undertaking will not be easy. I will have to become proficient enough in Lisp and RLisp to translate the former into Java using the latter. This is from a point of no experience of either.

I will also be required to gain an understanding of JLisp, which involves navigating a significant amount of code and identifying how its components work together. In order to produce Java that successfully interfaces with JLisp, it will also be necessary to take into account the fundamental differences between Lisp and Java as programming languages.

It would be unrealistic to attempt to learn everything about these systems before starting this project's implementation. As a result, I expect that difficulties may emerge which could not be foreseen during the planning stage. However, this is part of what makes the project interesting and does not deter my enthusiasm!

1.3 Outline of Tasks Required

Here I will provide a brief overview of this project's components. These have been refined since writing the project proposal, and correspond to the goals of the project.

1.3.1 Writing a Basic Translator

Produce a translator that accepts a Reduce function (i.e. a program written in Lisp) as input and outputs behaviourally equivalent Java. It must be capable of translating a small selection of basic Reduce functions.

1.3.2 Interfacing the Generated Java with JLisp

The Java output must be in a form compatible with the existing framework for JLisp. The end result should be that any Java-coded function definitions are executed in preference to interpreting the corresponding Lisp bytecodes.

1.3.3 Evaluating Correctness and Performance Change

It is crucial that the modified version of JLisp exactly replicates the behaviour of the original version. This requires a comprehensive testing process, which will ensure that any generated Java included in the modified version of JLisp

is completely safe. Sufficient timing data must also be collected so that performance differences between the original and modified versions of JLisp can be confidently attributed to the changes I make.

1.3.4 Extension: Expand the Translator

Once the main project components have been implemented, the next step will be to incrementally build more functionality into the translator, which will allow it to accept a greater range of input programs. This will also result in far more potential interactions between different parts of the translator, and almost certainly propel the original translation task to a new level of complexity. For this reason, I view expansion of the translator as an excellent candidate for a project extension.

This is a particularly open-ended goal, since its ideal completion state is achieved only by a 100% rate of successful translation of any valid input program. The scale of this task ensures that I will not run out of things to do, even if progress is rapidly made in the three essential goals!

Chapter 2

Preparation

Before starting work on the project, it was necessary to prepare in a number of ways. I needed to familiarise myself with the important aspects of JLisp and any other existing areas of the Reduce and CSL source code that were relevant. It was also crucial that I felt comfortable with the new languages I would be working with: RLisp and its underlying form, Lisp. Essential parts of any substantial software project were also considered – this includes revision control, testing strategy and development approach.

In the context of the aforementioned preparatory tasks, this chapter will demonstrate how proper steps were taken in order to ensure that the project was carried out in a professional manner. I will begin by providing an overview of the background knowledge that I had to establish in order to proceed with the project's implementation.

2.1 Background on Lisp

Although Lisp encompasses a family of programming languages, the variation used in this project is based on Standard Lisp. Thus unless otherwise specified, any discussion of Lisp in this dissertation will be with reference to Standard Lisp.

Lisp is a high-level programming language that originated in 1958. It employs a list-based notation whereby all list elements are either lists themselves, or atomic values such as integers or variable symbols.

```
(defun add (x y) (+ x y))
```

Note from the example provided – a Lisp function which adds together two input values – that Lisp employs Polish prefix notation, the distinguishing feature of which is the placement of all function symbols and operators to the left of their operands.

Lisp is expression-oriented; unlike many other languages – including Java – there is no distinction between expressions and statements. All code and data is in the form of expressions, and *any* valid expression in Lisp can be *evaluated* to produce a value. This marks a fundamental difference between Lisp and Java, and one which I will have to take account of when translating between the two.

2.1.1 Lists

Lisp is designed for symbolic computing. This is supported by the fundamental Lisp data structure: the list. Lists are recursively defined, created by evaluating the following constructors:

1. `nil`, which denotes the empty list.
2. `(cons x L)`, which creates a list headed by object `x` and followed by the members of list `L`.

For example, to construct a list containing 2 followed by 3, we would evaluate

```
(cons 2 (cons 3 nil))
```

Two key built-in functions are `car` and `cdr`:

`car (cons x L)` evaluates to `x`
`cdr (cons x L)` evaluates to `L`

We can use the `quote` special form to avoid the inconvenience of expressing a list as a series of nested `cons` cells:

```
(quote (2 3))
```

Using an equivalent shorthand syntax we can write

```
'(2 3)
```

Without quoting the expression `(2 3)`, it would be evaluated as a function call – the default Lisp form – with function symbol `2` and argument `3`; `quote` prevents evaluation of the list, and its elements are instead treated as literals,

known as Lisp symbols. A Lisp symbol is an object with a string representation – its name. Symbols are interned – any two symbols with the same name are the exact same object – and can be compared for equality with the built-in function `eq`. Note that symbols and strings are separate datatypes.

In addition to a name, symbols have a value, a function and a property list. A property list is a list of paired elements, each of which associates a property name with a property value. In conjunction with the functions `put` and `get`, properties can be associated with and retrieved from a Lisp symbol, making the property list a useful data structure. For example, the symbol `elephant` might have the following property list:

```
(name "ellie" colour "grey")
```

2.1.2 Special Forms

Special forms are pre-defined operators which provide the control structure for Lisp. The `quote` special form has already been introduced. Another example would be the `cond` special form, used for conditional evaluation in Lisp. All special forms relevant to this project will be suitably explained in the next chapter.

2.1.3 RLisp

RLisp is a high-level Lisp syntax written specifically for Reduce. Although this project was largely carried out using RLisp, it is not particularly relevant for the purposes of reading this dissertation; much more important is an understanding of the underlying Lisp.

2.2 Background on JLisp

JLisp is a Java-based Lisp system, comprising around 30,000 lines of Java code. Within this environment, Lisp is converted into Lisp-specific bytecodes which are then input as a stream to a bytecode interpreter. The components of JLisp can be classified into three main categories:

1. Representative datatypes
2. Built-in functions
3. JLisp runtime environment

2.2.1 Representative Datatypes

JLisp contains a collection of classes that are used to represent the various datatypes that exist in Standard Lisp. Here I will include brief descriptions of the most important classes.

`LispObject`

`LispObject` is an abstract class from which all other representative datatypes in JLisp are derived. Several fields and methods are included for general functionality – such as a `print` method – as well as empty implementations of methods used in specific subclasses.

`Symbol`

The JLisp `Symbol` object corresponds to a Lisp symbol. A hashmap of symbol names is maintained to ensure that JLisp `Symbol`s retain the singleton property of their Lisp counterparts: to retrieve the JLisp `Symbol` corresponding to a particular Lisp symbol, the `intern` method is called and the symbol's name provided as an argument. The hashmap is then queried and the corresponding `Symbol` object returned. If the `Symbol` object did not exist prior to the call to `intern`, a new one is instantiated.

A JLisp `Symbol`'s value is stored in the `car` field and its property list in the `cdr` field. These fields are so named due to the fact that they are common to all `LispObject` and made JLisp's implementation slightly more flexible.

`LispFunction`

A JLisp `Symbol`'s function is stored in the `fn` field, an instance of `LispFunction`. Contained in this class are method declarations `op0`, `op1`, `op2` and `opn`. These will henceforth be referred to as `op` methods. `0`, `1`, and `2` indicate the number of `LispObject` arguments accepted. `opn` accepts as its argument an array of three or more `LispObject`s. Calling the appropriate `op` method corresponds to a Lisp function call. Thus, we call a function in the following way:

```
Symbol.intern("demo_function").fn.op0();
```

2.2.2 Built-in Functions

Every built-in Lisp function is implemented in the JLisp sources using an instance of `LispFunction` and the appropriate `op` method. Amongst many others, this includes the functions `car` and `cdr` and basic numerical functions such as `plus2` and `times2`.

2.2.3 JLisp Runtime Environment

The JLisp runtime environment supports a command-line interface for Reduce, from which functions in the Reduce sources can be called. On starting JLisp, a core set of functions, symbols and other objects is loaded. This involves creating JLisp objects for all built-in functions and other members of the Lisp environment such as symbols `nil` and `t`, as well as for all Lisp objects which are part of the set of core Reduce packages.

2.3 Resources

2.3.1 Reduce Test Scripts and Reference Logs

Of the approximately 150 Reduce packages, 100 are accompanied by a comprehensive test script designed not only to showcase the package's capabilities, but more importantly to ensure that the underlying system running Reduce works correctly. Alongside every test script is a reference log of the test output which can be used as a comparison to verify the correct behaviour

of Reduce. These tests and logs were used to determine the correctness of my modified version of JLisp.

2.3.2 Lisp-to-C Translator

The Lisp-to-C translation code, written in RLisp, provided a basis from which to start work on the Lisp-to-Java translator. It was important to have a firm understanding of the code before starting the implementation – this would allow me to make an informed decision about which parts to retain and which to discard. However, the high complexity and large size (around 4,000 lines of RLisp) of the Lisp-to-C translator were not at all easily digestible, and eventually led to the decision to write most of the Lisp-to-Java translator from scratch.

2.3.3 Profiling Information

Profiling information for the set of Reduce test scripts was already available in the CSL sources, in a file named `profile.dat`. For each such test, the number of calls made to the bytecode interpreter for each Reduce function used is provided; in essence, this directly represents the function's frequency of use in every test. This was ideal, since I had an effective means of choosing the hotspot functions to prioritise for translation.

2.3.4 Reduce User's Manual

The Reduce User's Manual is available online and provided most of the documentation I needed in order to become familiar with RLisp.

2.4 Programming Languages

Unsurprisingly, the nature of this project dictated the programming languages used in its implementation; functions were translated into Java because JLisp is written in Java.

Although theoretically possible, writing the translator in any language other than RLisp would have been exceptionally inefficient; the Reduce source functions, being coded in RLisp (and therefore Lisp), can be directly input

as data to RLisp. This follows from the property that in Lisp, programs and data share exactly the same list-based structure. In addition, the Lisp-to-C translator is written in RLisp, and my intention to recycle some of its code for use in my own translator meant that I would have to work in the same language.

It is also worth mentioning the use of Bash in this project; many scripts were written and tools such as `grep` and `find` were needed for effective navigation of the Reduce sources.

2.5 Development Environment

2.5.1 Machine Specifications

This project was carried out using my desktop computer, which runs using a 3.3GHz dual-core Intel Core i3 processor under Windows 7. However, I find Linux to be a much more productive programming environment, and elected to use a virtual machine running the Debian distribution for all coding.

2.5.2 Version Control

Subversion was my chosen version control tool for this project. An account with Assembla, a cloud-based code storage host, was created and used as a reliable means for backing up the project repository. Important files were also backed up using Google Drive.

2.5.3 Programming Environment

The project involved a considerable amount of Linux shell navigation and usage of tools such as `grep`. As a result I chose to do all programming using `Vim`, the command-line text editor, in conjunction with `tmux`, an excellent terminal multiplexer. This proved to be a very productive environment; the absence of features usually offered only by an IDE was rarely noticed, and more than made up for by the zero frequency with which I experienced crashes and freezes usually associated with such software!

2.6 Software Development Approach

The methodology of incremental development was the natural choice for this project. Due to the enormous range of functions in the Reduce sources, small expansions in the translator's functionality would almost always result in an increase in translatable functions, and more importantly a new set of edge cases to consider. Since there was not a clearly defined stopping point, it made sense to incrementally build the translator, testing each set of newly eligible functions as new features were added.

2.7 Initial Experience

Thanks to extensive exposure in parts IA and IB of the Computer Science Tripos, I already had a solid grounding in Java before starting this project. I was also fairly comfortable using the Linux shell. In stark contrast was my non-existent prior experience with RLisp and Lisp. Working with software systems as substantial as Reduce and JLisp was something I had not done before.

2.8 Summary

In this chapter, I have discussed the steps taken to ensure that I was well-equipped to begin the implementation of the project. The required background knowledge on Lisp and JLisp was presented, followed by a description of the resources, languages and development environment that would be used. I then provided some context, with respect to my previous experience, to the position from which I would be working on this project.

Chapter 3

Implementation

The primary focus of this chapter will be the process of building the Lisp-to-Java translator, which after finishing the project is capable of translating the vast majority of potential input functions. Also covered will be how the task of interfacing the generated Java with JLisp was successfully carried out, and how the translation process was automated as the need became apparent. A high-level style of description will be adopted; minor details will largely be omitted.

A full example of translator output can be found in Appendix A.

3.1 Structure of Java Output

Before starting work on the translator itself, it was necessary to establish a format for the generated Java that would conform to the existing framework for JLisp.

Each translated function appears in its own class. The body of the function is contained in the appropriately named `op` method, in accordance with the way that JLisp makes calls to functions. In figure 3.1, several 'function classes' are contained in class `U01`; more generally, these function classes are contained within one of many possible `UXX` classes. A mapping is maintained in the `builtins` table for each function between its Lisp name and an instance of the class containing its Java translation. This topic is more extensively discussed in section 3.6.

```
package uk.co.codemist.jlisp.core;
import java.io.*;
import java.util.*;
import java.text.*;

class U01
{

  class J_example extends BuiltinFunction
  {

    public LispObject op1(LispObject arg1) throws Exception
    {
      // function definition here
    }

  }


  .
  .   // more function translations
  .

  Object [][] builtins =
  {
    {"example", new J_example()},
    .
    .
    .
  }

}
```

Figure 3.1: Structure of Java output file.

3.2 Translator Design

As mentioned in the previous chapter, I made the decision to implement the translator from scratch instead of attempting to modify the Lisp-to-C translator. However, it still made sense to retain the code used to prepare the translation environment and perform cleanup at the end of translation. This 'fringe' code required only a few modifications before I was able to get a Java output file resembling the structure shown in figure 3.1.

To avoid confusion between functions written as part of the translator and the Reduce functions being translated, I will refer to the former as 'procedures'. The top-level procedure accepts as input a Lisp function name, a sanitised[1] function name to be used for the corresponding Java class, and the function definition in its Lisp form (a Lisp list). Translation then proceeds as follows:

1. Process function arguments
2. Evaluate and generate Java for function body
3. Generate the `initialiseMe` method

Due to the recursive nature of the way in which expressions in Lisp are evaluated, the translator was designed to traverse the function body – simply a Lisp expression – in a similar fashion. Like the Lisp-to-C translator, my translator generates Java in a single pass through the use of print statements. This sequential approach prevented the use of any form of backtracking – a constraint which forced the anticipation of all potential problems deeper into the recursion at the top level of an expression.

With the recursive approach in mind, it was decided to write a single procedure that would accept any valid Lisp expression and 'evaluate'[2] it, thus allowing for recursion on subexpressions (those which occur within an expression). The set of potential input expressions was separated into three categories:

1. Atomic expressions
2. Function calls
3. Special forms

[1] See section 3.8.2.

[2] In the context of this project's implementation, the term 'evaluate' will also be used (when appropriate) to implicitly refer to the process of parsing and generating Java for a given input Lisp expression.

Thus the following structure for the evaluation procedure, `loadexp`, was used:

```
symbolic procedure loadexp(e, sym);
  begin;
    if atom e then
      .
      . % code for handling atomic expressions
      .
    else if special_form e then
      .
      . % code for handling special forms
      .
    else
      .
      . % code for handling function calls
      .
  end;
```

The argument `sym` stores the name of the Java variable that is used to store the result of the evaluated expression.

3.3 Atomic Expressions

3.3.1 Literal and Variable Tables

Literals (numbers, strings, symbols, booleans) and variable symbols constitute Lisp atomic expressions, or 'atoms'. Atoms are essentially non-list objects. A mapping is maintained between Lisp and Java representations of atoms through the use of two tables: a literal table and a variable table. They both consist of a list of tuple entries of the following format:

```
(Java variable name, Lisp atom)
```

Upon encountering a particular atom while translating a function, both tables are searched for an entry corresponding to the atom. If none exists in either table, an entry is added to the literal table[3]. Local variables can only be introduced as function arguments or as part of the `prog` special form; if

[3]The Java variable name is generated using the Lisp procedure `gensym`, which conveniently ensures that duplicate names are not produced.

no entry exists in either table, the atom must be a literal (fluid and global variables are exceptions to this statement and are discussed further in section 3.5.5).

3.4 Function Calls

Function calls can be thought of as the default type of expression in Lisp, with the following form:

```
(function_name e1 e2 ...)
```

To begin with, the arguments to the function must each be evaluated (these themselves could be function calls!). For example, imagine that the translator is confronted with the following function call:

```
(plus2 a b)
```

We first apply our evaluation procedure to `a` and `b`, storing the results in Java variables `g0000` and `g0001` respectively. The function symbol `plus` is a literal, so its Java variable name can be retrieved from the literal table (or generated if a table entry does not yet exist). This Java variable – `g0002` – points to the JLisp `Symbol` object representing the function symbol, and by accessing its `fn` field, the function can be called:

```
LispObject g0003 = g0002.fn.op2(g0000, g0001);
```

In this case, `g0003` is the variable being instantiated to store the value of the entire expression (ie. the function call). This crucial aspect of the translation process applies more generally: for *every* expression in an input Lisp program, a Java variable must be instantiated to store its value.

3.4.1 The `initialiseMe` Method

As the astute reader may have observed, I have so far neglected to mention how any of the literals found in a Lisp function definition are instantiated in its Java version. Now that I have introduced function calls, this is a good place to address the issue.

The initial design for a function's output Java class specified only the `op` method containing the Java translation of the function body. The translator

did not originally make use of a literal table; instead, evaluating a literal involved immediately declaring and initialising a Java variable to the literal's JLisp representation. The expression `(plus2 1 1)` would translate to:

```
LispObject g0004 = LispInteger.valueOf(1);
LispObject g0005 = LispInteger.valueOf(1);
LispObject g0006 = (Symbol.intern("plus2")).fn.op2(g0004, g0005);
```

Aside from the obvious redundancy, the inefficiency of this approach is greatly exacerbated by the fact that the translated function's `op` method could be called thousands of times in a typical Reduce session. In each of those calls, `intern` is called on the `Symbol` object, and new copies of a `LispObject` are created for each literal.

Recognising that literals values only needed to be instantiated once led to an improved design. After translating the function body, the translator iterates over the necessarily fully populated literal table and outputs the `initialiseMe` method and class variables. Due to their prevalence, the Lisp boolean symbols `nil` and `t` are always included.

```
class J_add extends BuiltinFunction
{
.
.   // op method
.
  private void initialiseMe()
  {
    g0005 = Symbol.intern("plus2");
    g0004 = LispInteger.valueOf(1);
    jnil = Jlisp.nil;
    jtrue = Jlisp.lispTrue;
    inited = true;
  }

  private LispObject jnil = null;
  private LispObject jtrue = null;
  private LispObject g0004 = null;
  private LispObject g0005 = null;
}
```

Why not just initialise the class variables to their intended values on decla-

ration? This is in fact what I originally tried. Due to the fact that the JLisp environment is dynamically created during startup, and that the order in which translated function classes are included in that creation could not be reliably determined, I found that the generated Java was referencing objects that did not yet exist! Thus all class variables are initialised to `null`.

By using the boolean flag `inited`, `initialiseMe` need only be called once per function per Reduce session. Hence the problem of creating and making potentially thousands of redundant objects and calls to `intern` respectively is solved. The expression from earlier, `(plus2 1 1)`, would now translate to:

```
LispObject g0006 = ((Symbol)g0005).fn.op2(g0004, g0004);
```

3.4.2 Inlining Functions

It was previously mentioned that the set of functions built into Lisp is fully implemented in JLisp. Most of the very commonly used built-in functions such as `car`, `cdr`, `atom` and `cons` have such simple implementations that the overhead of calling them in the way illustrated so far is much more significant than that of longer, less frequently used functions. I was able to inline a selection of these functions by mirroring their JLisp implementations in my translation code. A list of the these functions is maintained, and allows my translator to conditionally bypass the default handling mechanism for function calls in favour of one of two procedures for inlining.

The translator has an inlining procedure for functions which accept a fixed number of arguments and a separate procedure for those with an unbounded number of arguments. Here is an example of the inlining performed for `plus2`:

```
// Before inlining
LispObject g0006 = ((Symbol)g0005).fn.op2(g0004, g0004);

// After inlining
LispObject g0006 = g0004.add(g0004);
```

I had also noticed that many functions in the Reduce sources made recursive calls. These recursive calls were 'inlined' in the sense that accessing the function's `op` method via the corresponding JLisp `Symbol` object was bypassed.

```
// Before inlining
LispObject g0009 = ((Symbol)g0007).fn.op1(g0008);

// After inlining
LispObject g0009 = this.op1(g0008);
```

3.5 Special Forms

Special forms provide the control structure for Lisp. Each must be explicitly and uniquely handled during translation; treating a special form as a function call simply does not work. Thus separate procedures had to be written for translating each special form that I wanted to support. This constituted by far the most work involved in the translation process.

3.5.1 Conditional Expressions

In Lisp, a conditional expression consists of a list of condition-action pairs. The first expression in the pair is the condition, which is evaluated in the typical manner. If the condition does not evaluate to `nil`, its partner – the second expression – is evaluated. This second expression's value is considered the value of the entire conditional expression.

```
(cond (a1 a2) (b1 b2) ...  (t z2))
```

The first pair in the list for which the condition holds is evaluated. If all conditions have evaluated to `nil`, then the final pair, whose condition is `t` (true) is unconditionally evaluated. However, a final pair of this form is not compulsory; it is analogous to an `else` statement. An example Java translation for the `cond` special form is given in figure 3.2.

Because the number of condition pairs in a Lisp conditional expression is unbounded, I opted for a recursive method of translation.

3.5.2 Boolean Predicates

The boolean predicates `and` and `or` appear in Lisp as special forms. They accept an unbounded number of expressions as arguments and evaluate each

```
(cond ((greaterp x 2) x) ((greaterp y 3) (and y z)) (t z))

LispObject g0041 = jnil;     // store value of cond expression
LispObject g0044 = ((Symbol)g0043).fn.op2(x_arg, g0042_int);
if (g0044 != jnil)
{
  g0041 = x_arg;
}
else
{
  LispObject g0046 = ((Symbol)g0043).fn.op2(y_arg, g0045_int);
  if (g0046 != jnil)
  {
    LispObject g0047 = jnil; // store value of and expression
    if (y_arg != jnil)
    {
      if (z_arg != jnil)
      {
        g0047 = z_arg;
      }
    }
    g0041 = g0047;
  }
  else
  {
    g0041 = z_arg;
  }
}
```

Figure 3.2: Java translation for a `cond` expression. Also includes `and`.

such expression in turn.

`(and e1 e2 e3 ...)`

`(or e1 e2 e3 ...)`

If an argument to an `and` expression evaluates to `nil`, the remaining arguments are ignored and the `and` expression's value is `nil`. Else, the `and` expression will be assigned the value of the final argument. Analogously, an `or` expression is assigned the value of the first argument which does not evaluate to `nil`. If all arguments evaluate to `nil`, the entire `or` expression has the value `nil`.

These properties can be expressed in Java simply by correct use of Java conditional statements. A series of nested `if`s is produced for `and` expressions, and a non-nested set of `if...else if...` statements is produced for `or` expressions. Like the procedure for translating Lisp conditional expressions, the procedures for translating `and` and `or` operate recursively.

3.5.3 Quoted Expressions

It has already been explained that the `quote` special form indicates that an expression should *not* be evaluated, but instead be treated as a literal. This is trivial if the quoted expression is atomic, but catering for quoted lists was less straightforward; I shall remind the reader that list elements can themselves be lists!

The list's Java construction occurs as it is traversed: the creation of new `Cons` objects is inlined[4], and entries in the literal table are either retrieved or generated for every atom in the list. Dotted pairs, a type of `cons` cell which does not require its `cdr` to be a list, can also appear in a quoted list; this only introduced further unwelcome complexity to the process!

3.5.4 Sequential Execution

There are two special forms for sequential execution in Lisp; `progn` and `prog`.

A `progn` block simply consists of a list of expressions, each of which is evaluated in turn. The block evaluates to the value of the final expression in the

[4]This occurs independently of the inlining of functions mentioned in section 3.4.2.

```
(quote (a (b . c) (x y d)))      % (b . c) is a dotted pair

LispObject g0071 =
  new Cons(g0072,
      new Cons(
        new Cons(g0073, g0074),  // dotted pair
        new Cons(
          new Cons(g0075,
            new Cons(g0076,
              new Cons(g0077, jnil))), jnil)));
```

Figure 3.3: Example quoted list and corresponding Java translation.

list. As expected, supporting this type of special form was fairly straightforward.

On the other hand, handling `prog` blocks was one of the most challenging aspects of this project...

```
(prog (varA varB ...)  e1 e2 ...)
```

A `prog` block allows the binding of local variables; these are accessible only within the local context of the block, and have a default value of `nil`. In the example above, `varA` and `varB` are bound as local variables. For each such variable, a Java variable is declared in the translated code and an entry is added to the variable table. In contrast to the treatment of literals, these entries must be removed from the variable table after translating a `prog` block; this ensures that a variable cannot appear outside its scope in the generated Java.

Assigning a New Value to a Variable

With the introduction of local variables to my growing list of supported features came the requirement to accommodate another type of Lisp special form:

```
(setq x e)
```

It can be read as 'assign variable `x` the value of expression `e`'. Translation trivially involves evaluating `e` and setting the Java variable for `x` (retrieved from the variable table) to the result. The `setq` special form does a good job of illustrating the dissonance between Lisp and Java which had to be worked

around. For example, assignment to a conditional expression is perfectly valid in Lisp, whereas in Java a conditional statement (or any other type of statement) cannot be treated in this manner; hopefully it is becoming more clear that part of the challenge of this project was in finding ways to bridge the differences between the two languages in the Java output.

3.5.5 Fluid and Global Variables

For the sake of conciseness, I will refer to fluid variables as 'fluids' and global variables as 'globals'. As would seem likely, global variables are globally accessible once introduced into the Lisp environment. Fluids operate under a form of dynamic scoping unique to RLisp, which will shortly be explained. Globals and fluids have to be declared before use:

```
global '(global_var);
fluid  '(fluid_var);
```

Both fluids and globals are added to the literal table when first encountered during translation. Like function symbols, they are represented in JLisp by a `Symbol` object. This ensures that they are preserved as singletons; different functions can modify and reference the same fluid or global, which cannot be modelled using only local variables in Java. Thus fluids and globals must be globally accessible in JLisp. The value of a fluid or global is stored in the `car` field of the corresponding `Symbol`. Hence, an entry is added to the variable table, mapping the name of the fluid or global to the `car` field of its representative `Symbol`. This means that the fluid or global can be modified and referenced in the generated Java in the same way as local variables.

```
% Literal table
((global_var, g0001), (fluid_var, g0002) ...)

% Variable table
((global_var, g0001.car), (fluid_var, g0002.car) ...)
```

Binding a Fluid Variable

Unlike globals, fluids can be bound as a local variable in a `prog` block, or as an argument in a function definition. If bound as an argument, a fluid will

```
LispObject g0008save = dmodebh_fluid.car;
try
{
  dmodebh_fluid.car = jnil;
  LispObject g0010 = ((Symbol)g0009).fn.op1(u_arg);
  g0006 = g0010;
}
finally
{
  dmodebh_fluid.car = g0008save;
}
```

Figure 3.4: Treatment of a locally bound fluid.

take on the value provided in the function call. If bound as part of a `prog` block, its value will be set to `nil`. These values are only retained within the scope in which the fluid is bound, and are also accessible to any nested functions called from within the binding context. Upon termination of the binding function or `prog` block, the fluid resumes its old value.

Before including support for fluids, the list of function arguments in a function declaration or local variables at the start of a `prog` block would simply have been iterated over, each of its elements being stored in the variable table. A conditional test was added to the iteration so that it can be determined whether the variable is a fluid. If yes, the appropriate entries in both the literal table and the variable table are added.

A new data structure – the 'save' table – was introduced to map each bound fluid to a Java variable used to store the fluid's original value. After declaring the Java 'save' variables, a Java `try` block surrounds the body of the binding context. A `finally` block ensures that even in the event of an exception, fluids are always reassigned to their saved values.

3.5.6 Goto Expressions

A significant difference between Lisp and Java is the provision for goto expressions, which are supported in the former but not the latter. The problem of simulating goto expressions in Java was initially daunting, but eventually I arrived at a working solution.

```
(prog (varA ...)  ...  go_label ...  (go go_label) ...)
```

In Lisp, `go` labels (and thus goto expressions) can only appear within a `prog` block. For this reason, a significant overhaul of the translation unit for sequential execution was required. Translation of a `prog` block begins with a scan for `go` labels. For each label, an entry in a 'goto' table is generated, mapping a label to both a program counter variable and a specific value of the variable. The body of the block is then surrounded by a `for-switch` construction, as illustrated in figure 3.5.

The `switch` statement accepts the program counter variable as its argument; the body of the first `case` statement (`case 0`) corresponds to the sequence of expressions preceding the first goto label, and subsequent `case` statements correspond to successive goto labels. A final `case` statement is used to exit the `for-switch` construction.

Handling a goto expression within the `for-switch` construction is fairly straightforward. The program counter value associated with the label is first retrieved from the goto table. The program counter variable is then set to this value, and the `switch` statement is re-entered through the use of `continue`.

3.5.7 Return

A return expression in Lisp, like goto expressions, can only appear within a `prog` block.

```
(return e)
```

Expression `e` is evaluated, and the enclosing `prog` block is exited; its value is the value of expression `e`. Thus, the containing function is only returned from when returning from the outermost `prog` block. This behaviour is not analogous with the way that the Java return statement works, but I initially chose not to model it in my translator – instead, Lisp return expressions were directly translated into Java return statements. It was only possible to get away with this because the functions I was attempting to translate at that point were fairly basic and did not contain nested `prog` blocks.

The `for-switch` construction enabled me to correctly model Lisp return expressions; the variable representing the value of the `prog` block is set to the value of the return expression and the program counter is set so that execution flows to the final `case` statement, which terminates the `for` loop.

```
int goto_g0003 = 0;     // program counter variable
for(;;)
{
  switch(goto_g0003)
  {
    case 0:
    .
    .                   // code before first goto label
    .
    case 1:             // first goto label

      LispObject g0007 = l_arg == jnil ? jtrue : jnil;
      if (g0007 != jnil)
      {
        g0002 = g0004_loc;
        goto_g0003 = 2; // corresponds to a return expression
        continue;
      }
      LispObject g0008 = l_arg.cdr;
      l_arg = g0008;
      goto_g0003 = 1;   // corresponds to a goto expression
      continue;

    case 2:
      break;            // exit switch block
  }
  break;                // exit for loop
}
```

Figure 3.5: Simple example of the `for-switch` construction.

3.5.8 Unreachable Statements

An aspect of the Java compiler which complicated issues is its refusal to compile code containing 'unreachable' statements. The existence of such statements is made possible by the use of `continue` when handling return and goto expressions.

```
if (g0007 != jnil)
{
  g0002 = g0004_loc;
  goto_g0003 = 2;
  continue;
  g0006 = g0002;                // unreachable statement
}
```

Although inclusion of such code could be regarded as bad practice, if syntactically correct its presence should not affect the behaviour of a program. For mechanically generated Java not intended for human eyes, not having to worry about unreachable code would have saved a lot of time and effort at little cost! As a result, I was required to equip my translator with dead code elimination capabilities.

By considering the various scenarios in which the unreachable statements could occur, I arrived at a recursively defined definition of the set of Lisp expressions that would need to be identified, denoted `U` in the following pseudocode:

```
U := (return _)
   | (go _)
   | (cond (_ U) (_ U) ... (t U))
   | (progn ... U ...)
```

After writing a procedure to detect such expressions within a Lisp program, all that remained to do was to conditionally suppress the output of translated code. This task was less simple than it may seem – the binary variable used to indicate whether or not to print output needed to be set and reset in exactly the right places within each of the procedures used to handle the four affected special forms. Needless to say, the working solution to this problem was the product of multiple failed iterations!

3.5.9 Lambda Expressions

```
(lambda (x y) (plus2 x y)) ((times2 2 3) 4)
```

Lambda expressions are treated in my translator as a variation on the `prog` special form. The lambda variables are first added to the variable table. Then the arguments to the expression – in this case, `(times 2 3)` and `4` – are evaluated, and their results assigned to the corresponding lambda variables' representative Java variables. The body of the lambda expression – in this case, `(plus2 x y)`, can then be evaluated. The `for-switch` construction is not required for lambda expressions.

Lambda expressions are fairly common in the Reduce sources. This is because they are employed (in conjunction with goto expressions) in the Lisp mappings of the loop-based RLisp constructs `for each` and `while`. Although implementing support for the `lambda` special form did not appear as complex a task as that for some of the other special forms, I expected there to be a very large number of unpleasant edge cases and only tackled lambda expressions after a heavy round of testing the rest of the translator. My prediction turned out to be correct; it took many frustrating hours of debugging and unholy utterances to tease out the troublesome scenarios.

3.6 Interfacing the Generated Java with JLisp

3.6.1 Modifying the Startup Process

When building Reduce, every function in the Reduce sources is compiled into Lisp-specific bytecodes. When the function is called while running Reduce, the bytecodes are input as a stream to the JLisp bytecode interpreter. A function symbol's `LOSE` property can be set to `t` in order to remove its definition – stored in the `*savedef` property – thus preventing its compilation into bytecodes during the build. I modified the JLisp sources so that on startup, this would be done for every function for which a Java translation exists. So that the translated function can be called, the instance of the Java class mapped to the function's Lisp name in the `builtins` table is stored in the `fn` field of the corresponding JLisp `Symbol`.

3.6.2 Identifying Translation Errors

Errors Detected by the Java Compiler

Unsurprisingly, bugs in my translator were frequently exposed by errors raised by the Java compiler; it would have been highly unrealistic to suppose that I had covered all possible edge cases using just the unit tests I had written, which mainly tested each component of the translator in isolation. As the capabilities of my translator increased, the potential set of inputs grew far too quickly for unit testing to be considered a suitable strategy for debugging. In fact, the vast majority of problems with translation were revealed at the compilation stage.

Tracking Down Build-Breakers

Successful compilation of the generated Java was by no means a guarantee that translation had been performed correctly; the next potential point of failure is the process of buildling a Reduce image. Many checks are performed during this stage and functions that behave incorrectly can cause the build to fail. If not identifiable from inspection of the build logs[5], such functions were identified using a 'binary chop' approach, whereby half of the translated functions would be excluded from the build. The outcome of the build could then be used to determine which half contained the problem function, and the process repeated as necessary. The problem function could then be added to a list of functions to omit, which is properly explained in section 3.7.1. Using this method, the number of build attemps required to track down a problem function grows logarithmically with the total number of translated functions. The same approach was applied to builds which completed but did not produce test output equivalent to that of vanilla JLisp.

Given that a successful Reduce build took around 3 minutes to complete on my machine, this aspect of debugging was *extremely* time-consuming. It was possible to mitigate some of the inconvenience by building a streamlined version of Reduce – Minireduce – which brought the build time down to around 90 seconds. It is safe to say that over the course of this project's implementation, Minireduce saved me a great deal of time!

[5]Throughout this project, stack traces proved to be a very unreliable debugging tool.

3.7 Automating the Translation Process

As the project progressed, it became clear that it was going to be possible to translate a much larger selection of functions than I had initially anticipated. Although the high rate of progress largely welcome, it presented a new set of issues which had to be dealt with.

Using the available profiling information to select candidates for translation, I had been locating function definitions one at a time and checking that they were supported by my translator, before copying and pasting them from the Reduce sources into a file. The functions would then be translated into Java, which would be output to one of the `UXX` files. However, this approach was clearly not going to be scalable when attempting to translate a large number of functions at a time. My attention was drawn to a program, `make-c-code`, which had been written to aid in automating the Lisp-to-C translation process. I will briefly outline its operation:

1. Prepare a list of names of functions to be translated. The list is ordered and built using the information in `profile.dat`. Alternatively, the option to translate everything in the Reduce sources can be selected, in which case there is no need to refer to profiling information.

2. For each function whose name appears in the list of candidates, retrieve its function definition. This is stored in the `*savedef` property of the function symbol.

3. Iterate over the list of functions, supplying the definition of each to the translator, which outputs the translated code to a specified set of files.

My plan was to modify the program so that it was compatible with JLisp and the translator I had written (I suitably named it `make-java-code`). Since it was written using several RLisp features I was not yet familiar with, it took some time before I had produced a version that worked for JLisp.

3.7.1 Omitted Functions

Because I was no longer cherry-picking functions to translate, it was necessary to account for the fact that that my translator would encounter special forms that I had not included support for; the generated Java for any functions containing an unsupported special form would then be incorrect. For example, at one point my translator was not yet able to translate lambda expressions. In order to remedy this situation I produced a list of 'bad' words

corresponding to keywords associated with the unsupported special forms, and arranged that functions containing one or more such words would not be translated. Instead, a commented message identifying the function would appear at the bottom of the Java output file.

Some functions implemented in separate Reduce packages share the same name; this could be for several reasons, but again highlights the fact that large software with multiple contributors is prone to certain issues. In a purely interpreted environment, loading a package which contained one of these functions would simply cause an existing definition to be overwritten, and would not usually be expected to cause problems. However, this mode of operation is not compatible with the interface used in this project between JLisp and the generated Java – there is no support for mapping multiple definitions to the same function name. Hence all functions with name clashes were identified and added to a list of functions to omit from the translation process.

3.7.2 Irretrievable Function Definitions

While getting `make-java-code` to work, an issue began to surface: the definitions for a significant proportion of functions were not being retrieved. In fact, a mere 1,000 of 4,500[6] Reduce function definitions could be found! After establishing that a solution to the problem would not be trivial, I decided to focus on building a working version of JLisp that incorporated as much of the available 1,000 functions in their translated forms as possible. After all, this would still be well ahead of what I had originally expected to accomplish.

When this was eventually achieved I set out to solve the problem that I had placed on hold. I found that the majority of definitions were not being retrieved because Reduce only loads a core selection of packages by default; only functions defined in these core packages will have a populated `*savedef` property. The first idea I had was to iterate over the entire list of Reduce packages and load each one. However, several packages contain fluids and globals with the same name, or introduce fluids and globals which have the same symbol as local variables in another package. The result is incorrect translation and thus this approach could not be used. The solution I eventually came up with was to iterate over all packages, loading each one on its own and running `make-java-code` for just that package. Since packages frequently make use of functions found in other packages, this would have

[6] Approximate figures.

resulted in many duplicate translations. In order to prevent translation of the same function twice, a list to keep track of which functions have so far been translated is maintained.

3.8 Improving the Readablility of the Generated Java

Although originally intended as a possible extension to this project, the case for improving code readability strengthened as I made progress. I was translating increasingly involved functions, which produced longer and more complicated Java output. Tracking the precise behaviour of a translated function had become very difficult, since every Java variable was named under the same format (`gXXXX`). This in turn made debugging close to infeasible in certain cases.

3.8.1 Renaming Variables and Literals

For many situations, only a simple modification to my translator was required: when adding a mapping to the variable or literal tables, I arranged that the Java name for the mapping would simply be the name of the variable or literal, with a suffix attached to specify the type of variable or literal. For example, a local variable `x` in the Lisp program would now appear as `x_loc`. This proved to be invaluable during large bug-busting sessions. However, pathological cases emerged – such as the use of the same variable name in two separate loops – and I resorted back to a naming scheme of `gXXXX_loc` in order to prevent this problem.

3.8.2 Name Sanitisation

Due to a Reduce naming convention, most fluids and globals contain asterisks and other characters illegal in Java in their names. A procedure was written to sanitise any input name so that it would be appropriate for use in Java. Since Lisp uses exclamation marks as an escape character, a procedure also had to be written to sanitise and preserve strings between Lisp and Java.

3.9 Summary

This chapter has discussed the design and implementation of the Lisp-to-Java translator from a high-level perspective. It is capable of correctly translating the majority of the Reduce sources; this will be demonstrated in the following chapter. The task of interfacing the generated Java with JLisp was also described, as well as automation of the translation process. The steps involved in improving the readability of the generated Java were briefly touched upon. A full example of the translator's output is provided in Appendix A.

Chapter 4

Evaluation

All of this project's goals were accomplished. The Lisp-to-Java translator is capable of translating far more than a selection of basic Reduce functions – the original aim. The generated Java is fully compatible with JLisp, and offers a substantial increase in performance while preserving existing behaviour.

This chapter will begin by divulging the collection of translation statistics, which will provide some perspective on the state of the Lisp-to-Java translator after completing this project. Two more distinct stages of this project's evaluation will then be discussed:

1. Establishing the correctness of the translated code.
2. Measuring the change in performance of Reduce resulting from my modifications.

4.1 Translation Statistics

The Reduce test script profiling information names 4,578[1] unique functions. In total, 4,154 were translated and successfully incorporated into the final Reduce build. 185 functions were omitted from translation due to the name clash issue discussed in section 3.7.1. 38 functions required exclusion in order for the full set of Reduce test cases to pass, and 45 contained unsupported special forms. I even managed to find 7 functions that were erroneously

[1]Note that this number is not representative of the entirety of Reduce.

written! The remainder belonged to packages excluded from this evaluation for reasons explained in section 4.2.1.

From this information, the following can be stated:

- 90.7% of functions from the entire set of 4,578 were successfully translated
- 98.0% of functions *eligible* for translation were successfully translated
- 1.08% of functions were not translated due to unsupported special forms
- 0.91% of functions prevented one or more tests from producing ideal results

The successfully translated functions amount to just under 416,709 lines of generated Java (excluding blank lines). This figure was calculated using the following Bash script:

```
grep -v '^\s*$' core/UT*.java core/U0*.java core/U10.java \
| sed '/^\s*\/\//d' | wc -l
```

Clearly, the aim of successfully translating a selection of basic Reduce functions has been met. In fact, the translator is very close to the ideal state of universal operation! It is important to stress that the overall purpose of this project was to produce a faster version of JLisp; translation of Lisp into Java was purely the means by which this was achieved. The degradation in performance due to the few translations omitted through fault of the translator is almost certainly negligible.

4.2 Correctness of Translated Code

It was not considered appropriate to construct a formal proof of the correctness of my translator; the complexity of such a task is well beyond the scope of this project. Instead, correctness is considered in the context of one of this project's goals, which is to replicate the behaviour of vanilla JLisp – the control for this evaluation.

The nature of this project dictated a filter-based approach to detecting translation errors. Given the extremely large input space to my translator – indeed, almost any function written in RLisp – it was clear that individual unit

tests for the components of the translator would not be sufficient in capturing all possible cases.

A function's journey towards inclusion in a Reduce build involves passing through a perilous series of filters unscathed. Failed such journeys expose errors in the translator. I will provide a brief explanation of each filter:

1. Translation into Java: Syntax errors in the translator are instantly exposed by premature termination and an error message. Incorrect use of `car`, `cdr` and their derivatives is also punished at this stage.
2. Compiling the Generated Java: The Java compiler ruthlessly picks out syntax errors, and its complaints about undeclared variables were instrumental in identifying problems with usage of the various mapping tables used during translation.
3. Building Reduce: During the build process, many checks and a limited set of tests are carried out. These will catch some (but not all) erroneous translations.

4.2.1 Reduce Test Scripts

Of the 153 packages comprising Reduce, 100 are accompanied by a test script. Each of these packages also includes an 'ideal' test log, which was treated as the reference point for determining the correctness of translated functions used in that package's test script. Virtually all tests make use of functions from other packages, which contributes further rigour to this process.

I began by running the 100 Reduce tests using vanilla JLisp. I then manually compared each of the test logs with the corresponding reference log. This was done using vimdiff, a tool without which I would have painfully struggled to reliably detect discrepancies! Of the 100 Reduce tests, vanilla JLisp produced ideal output for 90. The remaining 10 were thus excluded from this project's evaluation. A further test – 'assert' – also had to be excluded, since it examines the structure of installed (pre-compiled) objects. The modified JLisp necessarily fails since there is no pre-compiled definition for any of the translated functions.

I am very pleased to report that after removing just the few problematic translations mentioned in section 4.1 from the build, the modified JLisp produced ideal output for all 89 tests. Therefore, this project's goal for correctness has been met.

4.3 Performance Analysis

4.3.1 The Java JIT Compiler

An interesting feature of the Java Virtual Machine is the just-in-time (JIT) compiler. It essentially performs a task in close harmony with the nature of this project: hotspot methods are identified during execution and the corresponding Java bytecodes are translated into directly executable machine code. This yields a significant increase in performance, but one which cannot be immediately realised. It will be shown in this project's performance analysis how the effects of the Java JIT compiler were taken into account.

4.3.2 Measurements Taken

For each Reduce test used in this analysis, the time taken to complete the test was measured for the four different implementations of Reduce:

1. Vanilla CSL
2. Optimised CSL
3. Vanilla JLisp
4. Modified JLisp

The test logs produced by running the Reduce test scripts conveniently record the time taken for completion of the test (in milliseconds). I wrote a script to extract the timings from a full set of test logs and output them, along with the test name, to a CSV file. This required spending some time learning to use `sed`, the Unix stream editor, but was well worth the avoidance of the far more time-consuming, error-prone alternative of manually reading each log file. A full set of tests was performed three times for each of the four implementations of Reduce, and the mean time for each test was computed. A fresh instance of Reduce was used for each individual test run.

An additional set of measurements was taken for both versions of JLisp. I was interested in observing the impact on timings caused by repeated runs of the same test in a single Reduce session; this is because it is impossible to ascertain from just one test run whether the Java JIT compiler has had a chance to profile and optimise the target program. Unfortunately, a large number of tests alter the initial Reduce environment on which they rely in order to complete without error. Consequently, I was only able to obtain this

sort of timing for a small subset of the full collection of test cases. These qualifying tests were repeatedly run until their times had converged – presumably the point at which the JIT compiler had no remaining optimisations to make.

To guarantee a fair test environment, I increased the priority of the VirtualBox process (I ran the tests on a Debian virtual machine) to 'High', closed all non-essential programs, and refrained from using my computer for the duration of the tests.

4.3.3 Discussion of Results

In this discussion, speedup is defined as the ratio between the average test time for vanilla JLisp and that for the modified JLisp. The two versions of CSL will be considered analogously.

For all sets of timings, the modified JLisp exhibits a significant improvement in performance compared with its vanilla counterpart. This is reflected in the chart on the following page.

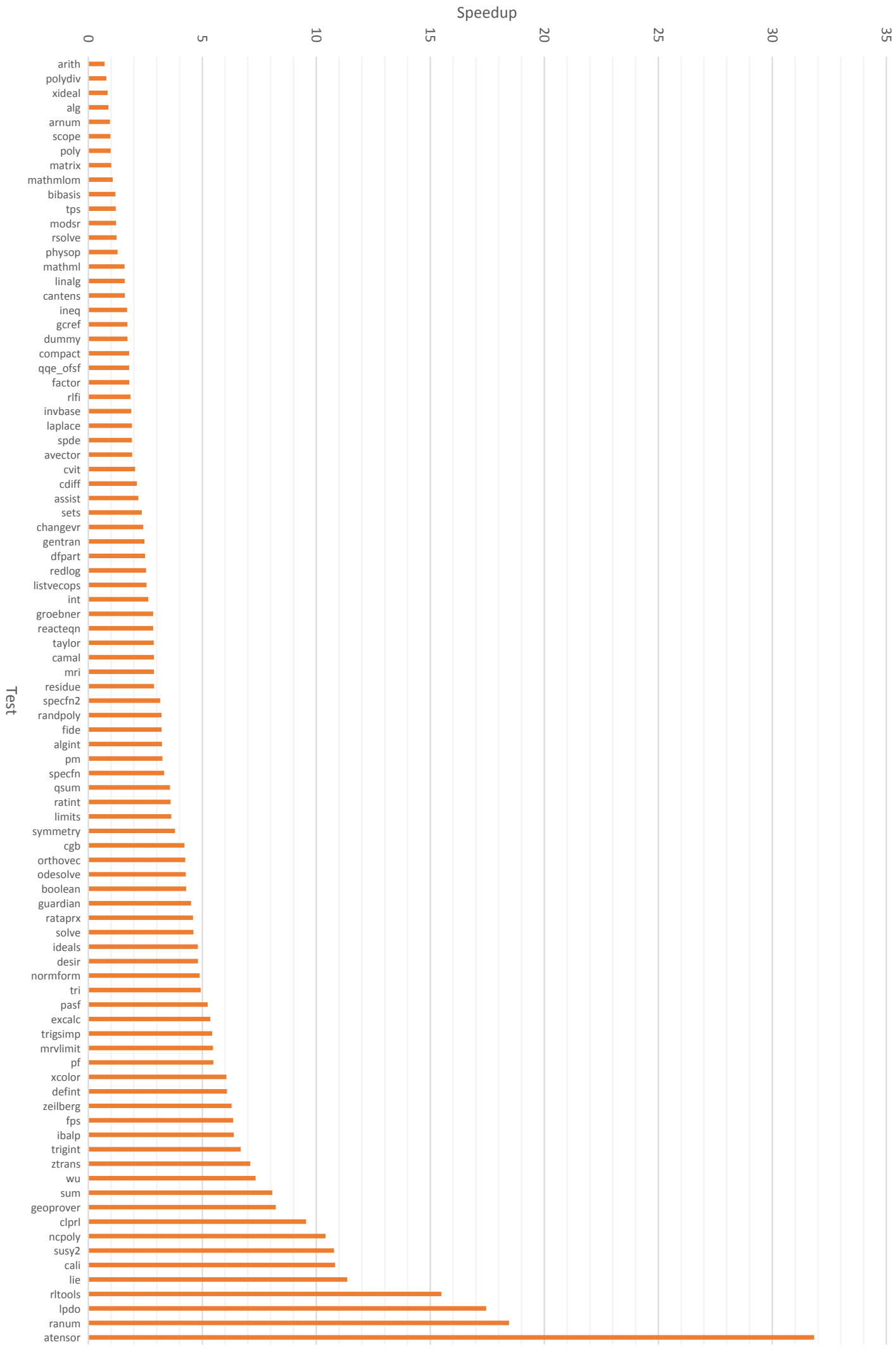
Two different metrics for 'overall' speedup in both JLisp and CSL were calculated:

1. Taking the arithmetic mean of the speedup of each individual test, over all tests
2. Summing the test times for both versions of Reduce and taking the ratio between these totals

The results are as follows:

	CSL	JLisp	JLisp/CSL
Metric 1	6.6	4.4	0.67
Metric 2	7.2	5.9	0.82
% difference	9	29	

Table 4.1: Comparison between speedup of CSL and JLisp using the two different metrics.


Speedup
0
5
10
15
20
25
30
35
Test
arith
polydiv
xideal
alg
arnum
scope
poly
matrix
mathmlom
bibasis
tps
modsr
rsolve
physop
mathml
linalg
cantens
ineq
gcref
dummy
compact
qqe_ofsf
factor
rlfi
invbase
laplace
spde
avector
cvit
cdiff
assist
sets
changevr
gentran
dfpart
redlog
listvecops
int
groebner
reacteqn
taylor
camal
mri
residue
specfn2
randpoly
fide
algint
pm
specfn
qsum
ratint
limits
symmetry
cgb
orthovec
odesolve
boolean
guardian
rataprx
solve
ideals
desir
normform
tri
pasf
excalc
trigsimp
mrvlimit
pf
xcolor
defint
zeilberg
fps
ibalp
trigint
ztrans
wu
sum
geoprover
clprl
ncpoly
susy2
cali
lie
rltools
lpdo
ranum
atensor

Several observations can be made from table 4.1. For both CSL and JLisp, a more favourable speedup was produced using metric 2, under which the weight of each test's individual speedup is proportional to the time taken to complete the test[2]. However, the difference between the two metrics is far more significant for JLisp, at 29%, than for CSL; This is almost certainly due to the fact that the Java JIT compiler has much more opportunity to perform optimisations during the longer tests.

From a user's perspective, it makes sense to assign more importance to the speedup of longer operations. For example, if test A and test B both report a speedup of 2, and the tests originally took 1s and 100s respectively to complete, then the 50s shaved off the time taken to complete test B has far more of an impact than the 500ms improvement in test A. Thus I view metric 2 as a more reliable indicator than metric 1 of the real-world impact that will be made by my modifications to JLisp. If Reduce usage statistics were available, it would have been possible to assign importance to the various packages based on how often they are used. This would have led to an even better metric for determining speedup.

I find the fraction of CSL's speedup achieved by that of JLisp very encouraging; ignoring the 'fringe' code largely shared between the Lisp-to-C and Lisp-to-Java translators, the latter is less than half the size of the former, and considerably less refined. However, direct comparisons between JLisp and CSL should not be taken too seriously because of the vastly different modes of execution between Java and C.

The two data series plotted in figure 4.1 have a correlation coefficient[3] of 0.39. While only weakly positive, it demonstrates a stronger relationship between speedup and original test time than the coefficient of 0.19 for CSL.

[2]Note that due to the variation in values of individual speedup, the distribution of this weight will not be equal between any of the four implementations of Reduce being tested.

[3]Pearson product-moment correlation coefficient.

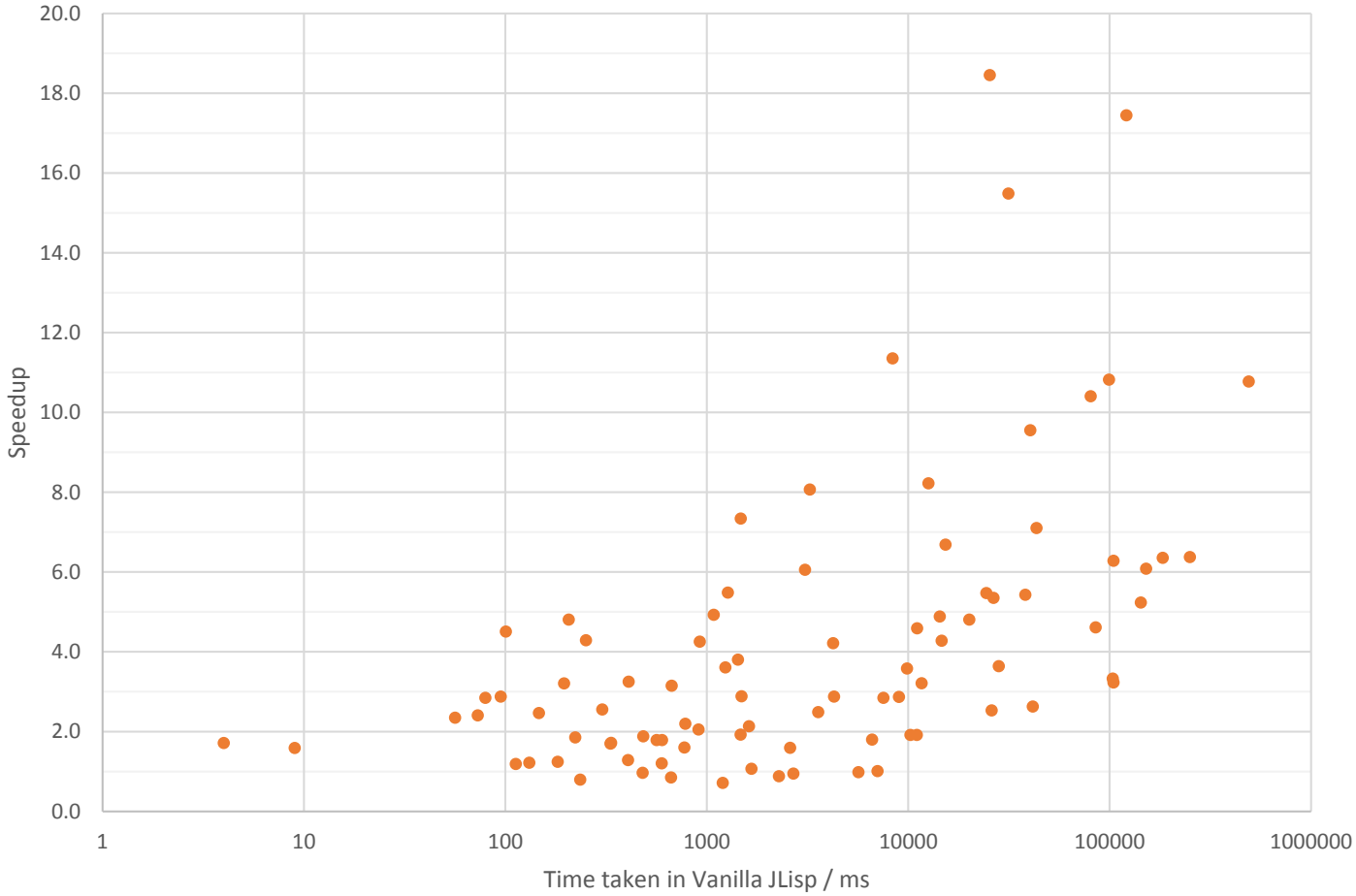
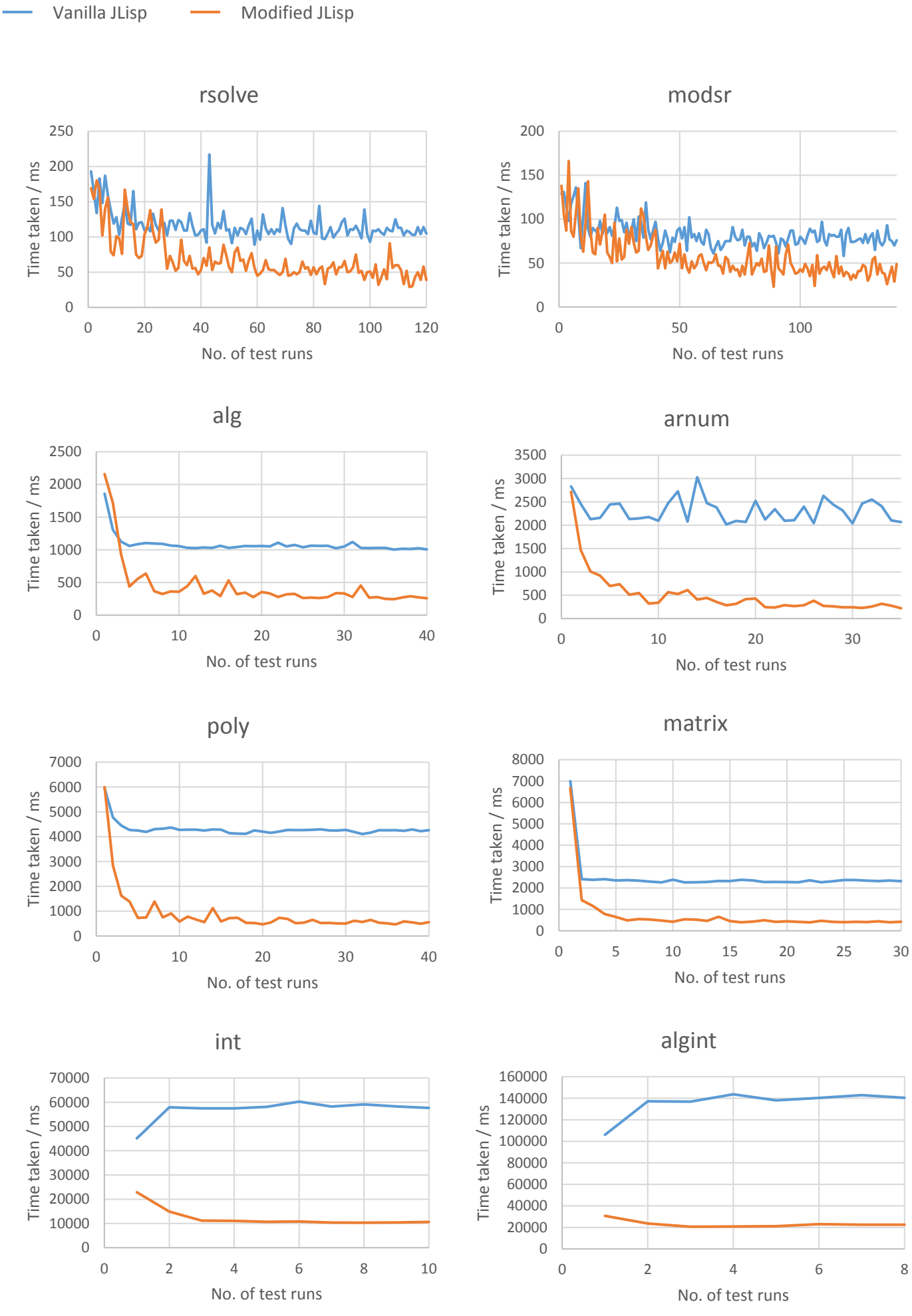


Figure 4.1: Plot of test speedup against the time taken for vanilla JLisp to complete the test.

Significant evidence of optimisation by the Java JIT compiler emerges from repeated runs of a selection of tests. These were repeated until the completion time had converged; this should represent the point at which no further optimisations are made by the JIT compiler. Even at this point of convergence, the time taken for individual test runs fluctuates. Hence I could only *estimate* the time taken for a test at its convergence point, and did so by averaging over multiple measurements taken after reaching this point. The set of graphs on the next page gives an indication of the convergence characteristic for each featured package test.


Vanilla JLisp
Modified JLisp
rsolve
Time taken / ms
No. of test runs
modsr
Time taken / ms
No. of test runs
alg
Time taken / ms
No. of test runs
arnum
Time taken / ms
No. of test runs
poly
Time taken / ms
No. of test runs
matrix
Time taken / ms
No. of test runs
int
Time taken / ms
No. of test runs
algint
Time taken / ms
No. of test runs

Test	Speedup		Speedup Ratio
	JLisp(1)	JLisp(i)	
rsolve	1.1	2.5	2.2
modsr	0.9	1.9	2.0
alg	0.9	3.6	4.2
arnum	1.0	8.5	8.2
poly	1.0	7.7	7.7
matrix	1.0	5.6	5.5
int	2.0	5.6	2.8
algint	3.4	6.2	1.8

Table 4.2: JLisp speedup after one run versus the value converged upon after many runs. Speedup under JLisp(1) is calculated using times taken for the first run of a test. Speedup under JLisp(i) is calculated using times taken at the point of convergence. Values are to 2 significant figures.

For all tests, the time taken at the point of convergence is visibly lower for the modified JLisp than for vanilla JLisp. The fluctuations in time taken are much greater in relative terms for the shorter tests, which suggests either a deviation characteristic at the level of the Java Virtual Machine[4] or merely an error-prone timer. Interestingly, the performance of vanilla JLisp for the longest two tests, int and algint, actually *decreased* after the first run. As things currently stand, it is not entirely clear why this is the case.

Not only did the modified JLisp perform better than vanilla JLisp at the point of convergence, but it was also more extensively optimised by the JIT compiler. This is reflected in the fact that the speedup for JLisp(i) was significantly higher than that of JLisp(1). Initially this was puzzling, but I soon discovered that the Java JIT compiler only performs method inlining – its main tool for optimisation – for methods less than a certain size. I also found out that this had not been considered when the JLisp bytecode interpreter had been written; its `interpret` method is well over 1,000 lines long. Conversely, the majority of methods generated by my translator are quite a lot smaller than this!

Thus it would appear that credit for a significant proportion of the improvement in JLisp's performance is due to a previously unconsidered side effect of executing functions from their Java translations!

[4]Although an interesting issue, it has little impact on the vast majority of Reduce test cases so I did not further investigate it.

Chapter 5

Conclusions

In this project I set out to establish whether translating Lisp into Java would affect the performance of JLisp, a Java-coded Lisp environment designed for use with the Reduce computer algebra system. This was to be carried out in the context of a similar approach that was applied to CSL, a C-coded Lisp environment, and was to involve writing a translator to mechanically produce Java given a Lisp function as input.

The project has been very successful – the original aims have all been met and where possible, quite far exceeded. The translator I have written has led to a Reduce build which incorporates 4,154 translated functions – more than 416,000 lines of generated Java. This has been extensively tested and is several times faster than the version available at the start of this project.

5.1 Further Work

There is still plenty of work to be done. The Lisp-to-Java translator does not currently accept all inputs programs and some of the translations it produces do not operate correctly, meaning that there are still some cases to account for!

I also expect that there is scope for optimising the generated Java during translation. For example, separating very large methods into several smaller methods is likely to result in more optimisation from the Java JIT compiler. There is also a lot of potential for improving JLisp.

The improved performance yielded by this project means that running Reduce in Java is much more viable than it previously was. This opens the possibility of writing a version of Reduce for Android and implementing a graphical interface.

Finally, I would like to be able to translate the translator using itself! I see no reason why this cannot be done, and it would have been attempted if there was more time.

5.2 Lessons Learned

Working with large software systems was a novelty and even though it had been considered, I had underestimated the increased potential for things to go wrong; problems were often not confined to the contents of a single file or directory and thus difficult to isolate. As a result, the lost time which had been scheduled for translation work had to be made up elsewhere.

Working with Lisp introduced me to a very refreshing programming paradigm, and having to translate it into Java (a rather different language!) has brought to my attention many programming 'delicacies' which I otherwise would never have thought about. I am now convinced that the best way to learn a new programming language is to find yourself racing against the clock to do something useful with it!

The project has been thoroughly enjoyable, and many valuable skills and lessons have been learned through its realisation.

Appendix A

Example Translation: `reval1`

This section is here to give the reader an idea of the scale of translation that was eventually achieved.

A.1 RLisp

```
symbolic procedure reval1(u,v);
   if null !*revalp then u
   else
   (begin scalar x,y;
      if null u then return nil
       else if stringp u then return u
       else if fixp u
        then return if flagp(dmode!*,'convert) then reval2(u,v) else u
       else if atom u
        then if null subfg!* then return u
              else if idp u and (x := get(u,'avalue))
               then if u memq varstack!* then recursiveerror u
                     else <<varstack!* := u . varstack!*;
                            return if y := get(car x,'evfn)
                                     then apply2(y,u,v)
                                    else reval1(cadr x,v)>>
              else nil
       else if not idp car u % or car u eq '!*comma!*
        then if (x := get(caar u,'structfn)) then return apply(x,{u})
              else errpri2(u,t)
       else if car u eq '!*sq
        then return if caddr u and null !*resimp
                     then if null v then u else prepsqxx cadr u
                    else reval2(u,v)
       else if flagp(car u,'remember) then return rmmbreval(u,v)
       else if flagp(car u,'opfn) then return reval1(opfneval u,v)
       else if x := get(car u,'psopfn)
         then <<u := apply1(x,cdr u);
                if x := get(x,'cleanupfn) then u := apply2(x,u,v);
                return u>>
```

```
      else if arrayp car u then return reval1(getelv u,v);
      return if x := getrtype u then
                 if y := get(x,'evfn) then apply2(y,u,v)
                   else rerror(alg,101,
                               list("Missing evaluation for type",x))
             else if not atom u
                   and not atom cdr u
                   and (y := getrtype cadr u)
                   and null(y eq 'list and cddr u)
                   and (x := get(y,'aggregatefn))
                   and (not(x eq 'matrixmap) or flagp(car u,'matmapfn))
                   and not flagp(car u,'boolean)
                   and not !*listargs and not flagp(car u,'listargp)
              then apply2(x,u,v)
             else reval2(u,v)
  end) where varstack!* := varstack!*;
```

A.2 Lisp

```
(progn (put (quote reval1) (quote number!-of!-args) 2) (de reval1 (u v) (
cond ((null !*revalp) u) (t ((lambda (varstack!*) (declare (special varstack!*))
(prog (x y) (cond ((null u) (return nil)) ((stringp u) (return u)) ((fixp u) (
return (cond ((flagp dmode!* (quote convert)) (reval2 u v)) (t u)))) ((atom u) (
cond ((null subfg!*) (return u)) ((and (idp u) (setq x (get u (quote avalue))))
(cond ((memq u varstack!*) (recursiveerror u)) (t (progn (setq varstack!* (cons
u varstack!*)) (return (cond ((setq y (get (car x) (quote evfn))) (apply2 y u v)
) (t (reval1 (cadr x) v))))))))) (t nil))) ((not (idp (car u))) (cond ((setq x (
get (caar u) (quote structfn))) (return (apply x (list u)))) (t (errpri2 u t))))
((eq (car u) (quote !*sq)) (return (cond ((and (caddr u) (null !*resimp)) (cond
((null v) u) (t (prepsqxx (cadr u))))) (t (reval2 u v))))) ((flagp (car u) (
quote remember)) (return (rmmbreval u v))) ((flagp (car u) (quote opfn)) (return
(reval1 (opfneval u) v))) ((setq x (get (car u) (quote psopfn))) (progn (setq u
(apply1 x (cdr u))) (cond ((setq x (get x (quote cleanupfn))) (setq u (apply2 x
u v)))) (return u))) ((arrayp (car u)) (return (reval1 (getelv u) v)))) (return
(cond ((setq x (getrtype u)) (cond ((setq y (get x (quote evfn))) (apply2 y u v)
) (t (rerror (quote alg) 101 (list "Missing evaluation for type" x))))) ((and (
not (atom u)) (not (atom (cdr u))) (setq y (getrtype (cadr u))) (null (and (eq y
(quote list)) (cddr u))) (setq x (get y (quote aggregatefn))) (or (not (eq x (
quote matrixmap))) (flagp (car u) (quote matmapfn))) (not (flagp (car u) (quote
boolean))) (not !*listargs) (not (flagp (car u) (quote listargp)))) (apply2 x u
v)) (t (reval2 u v)))))) varstack!*)))))
```

A.3 Java

```
class J_reval1 extends BuiltinFunction
{

  public LispObject op2(LispObject u_arg, LispObject v_arg) throws Exception
  {
    if (!inited) initialiseMe();
    LispObject top_return = jnil;
    LispObject g1054 = jnil;
    LispObject g1055 = bhrevalp_fluid.car == jnil ? jtrue : jnil;
    if (g1055 != jnil)
```

```
{
  g1054 = u_arg;
}
else
{
  LispObject g1056 = jnil;
  LispObject g1058save = varstackbh_fluid.car;
  try
  {
    varstackbh_fluid.car = varstackbh_fluid.car;
    LispObject g1059 = jnil;
    LispObject g1061_loc = jnil;
    LispObject g1062_loc = jnil;
    int goto_g1060 = 0;
    for(;;)
    {
      switch(goto_g1060)
      {
        case 0:
          goto_g1060 += 1;
          LispObject g1063 = jnil;
          LispObject g1064 = u_arg == jnil ? jtrue : jnil;
          if (g1064 != jnil)
          {
            g1059 = jnil;
            goto_g1060 = 1;
            continue;
          }
          else
          {
            LispObject g1066 = ((Symbol)g1065).fn.op1(u_arg);
            if (g1066 != jnil)
            {
              g1059 = u_arg;
              goto_g1060 = 1;
              continue;
            }
            else
            {
              LispObject g1068 = ((Symbol)g1067).fn.op1(u_arg);
              if (g1068 != jnil)
              {
                LispObject g1069 = jnil;
                LispObject g1071 = ((Symbol)g1070).fn.op2(dmodebh_fluid.car, g1072);
                if (g1071 != jnil)
                {
                  LispObject g1074 = ((Symbol)g1073).fn.op2(u_arg, v_arg);
                  g1069 = g1074;
                }
                else
                {
                  g1069 = u_arg;
                }
                g1059 = g1069;
                goto_g1060 = 1;
                continue;
              }
              else
              {
                LispObject g1075 = u_arg.atom ? jtrue : jnil;
                if (g1075 != jnil)
                {
```

```
  LispObject g1076 = jnil;
  LispObject g1077 = subfgbh_fluid.car == jnil ? jtrue : jnil;
  if (g1077 != jnil)
  {
    g1059 = u_arg;
    goto_g1060 = 1;
    continue;
  }
  else
  {
    LispObject g1078 = jnil;
    LispObject g1080 = ((Symbol)g1079).fn.op1(u_arg);
    if (g1080 != jnil)
    {
      LispObject g1082 = ((Symbol)g1081).fn.op2(u_arg, g1083);
      g1061_loc = g1082;
      if (g1061_loc != jnil)
      {
        g1078 = g1061_loc;
      }
    }
    if (g1078 != jnil)
    {
      LispObject g1084 = jnil;
      LispObject g1086 = ((Symbol)g1085).fn.op2(u_arg, varstackbh_fluid.car);
      if (g1086 != jnil)
      {
        LispObject g1088 = ((Symbol)g1087).fn.op1(u_arg);
        g1084 = g1088;
      }
      else
      {
        LispObject g1089 = new Cons(u_arg, varstackbh_fluid.car);
        varstackbh_fluid.car = g1089;
        LispObject g1090 = jnil;
        LispObject g1091 = g1061_loc.car;
        LispObject g1092 = ((Symbol)g1081).fn.op2(g1091, g1093);
        g1062_loc = g1092;
        if (g1062_loc != jnil)
        {
          LispObject[] g1095 = {g1062_loc, u_arg, v_arg};
          LispObject g1096 = ((Symbol)g1094).fn.opn(g1095);
          g1090 = g1096;
        }
        else
        {
          LispObject g1097 = g1061_loc.cdr.car;
          LispObject g1098 = this.op2(g1097, v_arg);
          g1090 = g1098;
        }
        g1059 = g1090;
        goto_g1060 = 1;
        continue;
      }
      g1076 = g1084;
    }
    else
    {
    }
  }
  g1063 = g1076;
}
```

```
else
{
  LispObject g1099 = u_arg.car;
  LispObject g1100 = ((Symbol)g1079).fn.op1(g1099);
  LispObject g1101 = g1100 == jnil ? jtrue : jnil;
  if (g1101 != jnil)
  {
    LispObject g1102 = jnil;
    LispObject g1103 = u_arg.car.car;
    LispObject g1104 = ((Symbol)g1081).fn.op2(g1103, g1105);
    g1061_loc = g1104;
    if (g1061_loc != jnil)
    {
      LispObject g1106 =
        new Cons(u_arg, jnil);
      LispObject g1108 = ((Symbol)g1107).fn.op2(g1061_loc, g1106);
      g1059 = g1108;
      goto_g1060 = 1;
      continue;
    }
    else
    {
      LispObject g1110 = ((Symbol)g1109).fn.op2(u_arg, jtrue);
      g1102 = g1110;
    }
    g1063 = g1102;
  }
  else
  {
    LispObject g1111 = u_arg.car;
    LispObject g1113 = ((Symbol)g1112).fn.op2(g1111, g1114);
    if (g1113 != jnil)
    {
      LispObject g1115 = jnil;
      LispObject g1116 = jnil;
      LispObject g1118 = ((Symbol)g1117).fn.op1(u_arg);
      if (g1118 != jnil)
      {
        LispObject g1119 = bhresimp_fluid.car == jnil ? jtrue : jnil;
        if (g1119 != jnil)
        {
          g1116 = g1119;
        }
      }
      if (g1116 != jnil)
      {
        LispObject g1120 = jnil;
        LispObject g1121 = v_arg == jnil ? jtrue : jnil;
        if (g1121 != jnil)
        {
          g1120 = u_arg;
        }
        else
        {
          LispObject g1122 = u_arg.cdr.car;
          LispObject g1124 = ((Symbol)g1123).fn.op1(g1122);
          g1120 = g1124;
        }
        g1115 = g1120;
      }
      else
      {
```

```
    LispObject g1125 = ((Symbol)g1073).fn.op2(u_arg, v_arg);
    g1115 = g1125;
  }
  g1059 = g1115;
  goto_g1060 = 1;
  continue;
}
else
{
  LispObject g1126 = u_arg.car;
  LispObject g1127 = ((Symbol)g1070).fn.op2(g1126, g1128);
  if (g1127 != jnil)
  {
    LispObject g1130 = ((Symbol)g1129).fn.op2(u_arg, v_arg);
    g1059 = g1130;
    goto_g1060 = 1;
    continue;
  }
  else
  {
    LispObject g1131 = u_arg.car;
    LispObject g1132 = ((Symbol)g1070).fn.op2(g1131, g1133);
    if (g1132 != jnil)
    {
      LispObject g1135 = ((Symbol)g1134).fn.op1(u_arg);
      LispObject g1136 = this.op2(g1135, v_arg);
      g1059 = g1136;
      goto_g1060 = 1;
      continue;
    }
    else
    {
      LispObject g1137 = u_arg.car;
      LispObject g1138 = ((Symbol)g1081).fn.op2(g1137, g1139);
      g1061_loc = g1138;
      if (g1061_loc != jnil)
      {
        LispObject g1140 = u_arg.cdr;
        LispObject g1142 = ((Symbol)g1141).fn.op2(g1061_loc, g1140);
        u_arg = g1142;
        LispObject g1143 = jnil;
        LispObject g1144 = ((Symbol)g1081).fn.op2(g1061_loc, g1145);
        g1061_loc = g1144;
        if (g1061_loc != jnil)
        {
          LispObject[] g1146 = {g1061_loc, u_arg, v_arg};
          LispObject g1147 = ((Symbol)g1094).fn.opn(g1146);
          u_arg = g1147;
          g1143 = u_arg;
        }
        g1059 = u_arg;
        goto_g1060 = 1;
        continue;
      }
      else
      {
        LispObject g1148 = u_arg.car;
        LispObject g1150 = ((Symbol)g1149).fn.op1(g1148);
        if (g1150 != jnil)
        {
          LispObject g1152 = ((Symbol)g1151).fn.op1(u_arg);
          LispObject g1153 = this.op2(g1152, v_arg);
```

```
                    g1059 = g1153;
                    goto_g1060 = 1;
                    continue;
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
LispObject g1154 = jnil;
LispObject g1156 = ((Symbol)g1155).fn.op1(u_arg);
g1061_loc = g1156;
if (g1061_loc != jnil)
{
  LispObject g1157 = jnil;
  LispObject g1158 = ((Symbol)g1081).fn.op2(g1061_loc, g1093);
  g1062_loc = g1158;
  if (g1062_loc != jnil)
  {
    LispObject[] g1159 = {g1062_loc, u_arg, v_arg};
    LispObject g1160 = ((Symbol)g1094).fn.opn(g1159);
    g1157 = g1160;
  }
  else
  {
    LispObject g1162 =
      new Cons(g1163,
          new Cons(g1061_loc, jnil));
    LispObject[] g1165 = {g1166, g1161_int, g1162};
    LispObject g1167 = ((Symbol)g1164).fn.opn(g1165);
    g1157 = g1167;
  }
  g1154 = g1157;
}
else
{
  LispObject g1168 = jnil;
  LispObject g1169 = u_arg.atom ? jtrue : jnil;
  LispObject g1170 = g1169 == jnil ? jtrue : jnil;
  if (g1170 != jnil)
  {
    LispObject g1171 = u_arg.cdr;
    LispObject g1172 = g1171.atom ? jtrue : jnil;
    LispObject g1173 = g1172 == jnil ? jtrue : jnil;
    if (g1173 != jnil)
    {
      LispObject g1174 = u_arg.cdr.car;
      LispObject g1175 = ((Symbol)g1155).fn.op1(g1174);
      g1062_loc = g1175;
      if (g1062_loc != jnil)
      {
        LispObject g1176 = jnil;
        LispObject g1177 = ((Symbol)g1112).fn.op2(g1062_loc, g1178);
        if (g1177 != jnil)
        {
          LispObject g1179 = u_arg.cdr.cdr;
          if (g1179 != jnil)
          {
```

```
          g1176 = g1179;
        }
      }
      LispObject g1180 = g1176 == jnil ? jtrue : jnil;
      if (g1180 != jnil)
      {
        LispObject g1181 = ((Symbol)g1081).fn.op2(g1062_loc, g1182);
        g1061_loc = g1181;
        if (g1061_loc != jnil)
        {
          LispObject g1183 = jnil;
          LispObject g1184 = ((Symbol)g1112).fn.op2(g1061_loc, g1185);
          LispObject g1186 = g1184 == jnil ? jtrue : jnil;
          if (g1186 != jnil)
          {
            g1183 = g1186;
          }
          else
          {
            LispObject g1187 = u_arg.car;
            LispObject g1188 = ((Symbol)g1070).fn.op2(g1187, g1189);
            if (g1188 != jnil)
            {
              g1183 = g1188;
            }
          }
          if (g1183 != jnil)
          {
            LispObject g1190 = u_arg.car;
            LispObject g1191 = ((Symbol)g1070).fn.op2(g1190, g1192);
            LispObject g1193 = g1191 == jnil ? jtrue : jnil;
            if (g1193 != jnil)
            {
              LispObject g1194 = bhlistargs_fluid.car == jnil ? jtrue : jnil;
              if (g1194 != jnil)
              {
                LispObject g1195 = u_arg.car;
                LispObject g1196 = ((Symbol)g1070).fn.op2(g1195, g1197);
                LispObject g1198 = g1196 == jnil ? jtrue : jnil;
                if (g1198 != jnil)
                {
                  g1168 = g1198;
                }
              }
            }
          }
        }
      }
    }
  }
}
if (g1168 != jnil)
{
  LispObject[] g1199 = {g1061_loc, u_arg, v_arg};
  LispObject g1200 = ((Symbol)g1094).fn.opn(g1199);
  g1154 = g1200;
}
else
{
  LispObject g1201 = ((Symbol)g1073).fn.op2(u_arg, v_arg);
  g1154 = g1201;
}
```

```
            }
            g1059 = g1154;
            goto_g1060 = 1;
            continue;
          case 1:
            break;
        }
        break;
      }
      g1056 = g1059;
    }
    finally
    {
      varstackbh_fluid.car = g1058save;
    }
    g1054 = g1056;
  }
  top_return = g1054;
  method_count += 1;
  return top_return;
}

private void initialiseMe()
{
  bhrevalp_fluid = Symbol.intern("*revalp");
  varstackbh_fluid = Symbol.intern("varstack*");
  g1065 = Symbol.intern("stringp");
  g1067 = Symbol.intern("fixp");
  g1070 = Symbol.intern("flagp");
  dmodebh_fluid = Symbol.intern("dmode*");
  g1072 = Symbol.intern("convert");
  g1073 = Symbol.intern("reval2");
  subfgbh_fluid = Symbol.intern("subfg*");
  g1079 = Symbol.intern("idp");
  g1081 = Symbol.intern("get");
  g1083 = Symbol.intern("avalue");
  g1085 = Symbol.intern("memq");
  g1087 = Symbol.intern("recursiveerror");
  g1093 = Symbol.intern("evfn");
  g1094 = Symbol.intern("apply2");
  g1202 = Symbol.intern("reval1");
  g1105 = Symbol.intern("structfn");
  g1107 = Symbol.intern("apply");
  g1109 = Symbol.intern("errpri2");
  g1112 = Symbol.intern("eq");
  g1114 = Symbol.intern("*sq");
  g1117 = Symbol.intern("caddr");
  bhresimp_fluid = Symbol.intern("*resimp");
  g1123 = Symbol.intern("prepsqxx");
  g1128 = Symbol.intern("remember");
  g1129 = Symbol.intern("rmmbreval");
  g1133 = Symbol.intern("opfn");
  g1134 = Symbol.intern("opfneval");
  g1139 = Symbol.intern("psopfn");
  g1141 = Symbol.intern("apply1");
  g1145 = Symbol.intern("cleanupfn");
  g1149 = Symbol.intern("arrayp");
  g1151 = Symbol.intern("getelv");
  g1155 = Symbol.intern("getrtype");
  g1164 = Symbol.intern("rerror");
  g1166 = Symbol.intern("alg");
  g1161_int = LispInteger.valueOf(101);
```

```
    g1163 = new LispString("Missing evaluation for type");
    g1178 = Symbol.intern("list");
    g1182 = Symbol.intern("aggregatefn");
    g1185 = Symbol.intern("matrixmap");
    g1189 = Symbol.intern("matmapfn");
    g1192 = Symbol.intern("boolean");
    bhlistargs_fluid = Symbol.intern("*listargs");
    g1197 = Symbol.intern("listargp");
    jnil = Jlisp.nil;
    jtrue = Jlisp.lispTrue;
    method_count = 0;
    inited = true;
  }

  private LispObject jnil = null;
  private LispObject jtrue = null;
  private LispObject bhrevalp_fluid = null;
  private LispObject varstackbh_fluid = null;
  private LispObject g1065 = null;
  private LispObject g1067 = null;
  private LispObject g1070 = null;
  private LispObject dmodebh_fluid = null;
  private LispObject g1072 = null;
  private LispObject g1073 = null;
  private LispObject subfgbh_fluid = null;
  private LispObject g1079 = null;
  private LispObject g1081 = null;
  private LispObject g1083 = null;
  private LispObject g1085 = null;
  private LispObject g1087 = null;
  private LispObject g1093 = null;
  private LispObject g1094 = null;
  private LispObject g1202 = null;
  private LispObject g1105 = null;
  private LispObject g1107 = null;
  private LispObject g1109 = null;
  private LispObject g1112 = null;
  private LispObject g1114 = null;
  private LispObject g1117 = null;
  private LispObject bhresimp_fluid = null;
  private LispObject g1123 = null;
  private LispObject g1128 = null;
  private LispObject g1129 = null;
  private LispObject g1133 = null;
  private LispObject g1134 = null;
  private LispObject g1139 = null;
  private LispObject g1141 = null;
  private LispObject g1145 = null;
  private LispObject g1149 = null;
  private LispObject g1151 = null;
  private LispObject g1155 = null;
  private LispObject g1164 = null;
  private LispObject g1166 = null;
  private LispObject g1161_int = null;
  private LispObject g1163 = null;
  private LispObject g1178 = null;
  private LispObject g1182 = null;
  private LispObject g1185 = null;
  private LispObject g1189 = null;
  private LispObject g1192 = null;
  private LispObject bhlistargs_fluid = null;
  private LispObject g1197 = null;
```

```
}
```

Appendix B

Project Proposal

Computer Science Part II Project Proposal

Optimising REDUCE in Java

Josh Treon, Trinity College

Originator: Dr Arthur Norman

24th October 2014

Project Supervisor: Dr Arthur Norman

Director of Studies: Dr Arthur Norman

Project Overseers: Dr Robert Watson & Professor Peter Robinson

Introduction

"REDUCE is an interactive system for general algebraic computations of interest to mathematicians, scientists and engineers." Some of the things it is capable of are analytic differentiation and integration, expansion and ordering of polynomials and rational functions, and arbitrary precision integer and real arithmetic.

The REDUCE source code is essentially written in Lisp. When running REDUCE, the Lisp code is compiled into Lisp-specific bytecodes and fed through a Lisp interpreter. There currently exist two implementations of such an interpreter - one written in C (CSL) and one written in Java (JLisp).

Unfortunately, JLisp runs extremely slowly. To provide some context to this statement, a benchmark function was written in Lisp and timed running natively as well as in CSL and JLisp. Given the same arguments in each case, the function was computed in approximately the following times: Lisp - 15s, CSL - 2s, JLisp - 120s. The function when written directly in Java computed in less than 1s.

My project will be to rewrite several parts of JLisp and measure and analyse the change in performance resulting from my modifications. It will be useful in a broader sense because a Java implementation of REDUCE offers several advantages over a version written in C:

1. Java is often much easier to use when interfacing with other software; an obvious example here is running REDUCE on Android.
2. A Java version of REDUCE can be converted to Javascript and used in a web browser.
3. The existing framework for Java makes building extensions for REDUCE a much simpler and quicker task than doing so using C. Such examples could include developing a GUI, experimenting with concurrent use and implementing network access.

Work to be Done

To begin, I will need to establish familiarity with my working environment. This will entail installing REDUCE, gaining an understanding of how it works, and most importantly, looking through the JLisp code until I am comfortable with how it operates.

In order to produce a convincing evaluation, it will be necessary to carry out some initial benchmark tests to determine the relative performance of the main components of REDUCE. This will involve native Lisp code, CSL, JLisp and some translations of small functions from REDUCE into C and Java that I will manually write.

Since REDUCE consists of around 400,000 lines of code, and JLisp 30,000, it will not be feasible to translate or optimise every aspect of the software. Instead, I will use profiling tools to identify the most frequently used areas of code and focus on modifying those, either by manually translating them or by mechanical conversion into Java. This will concern approximately the most frequently called 5% of functions. It is possible that in several cases issues with translation will arise. In such situations, my approach will be to omit the problematic function from the set of candidates.

There will also be the task of deciding which parts of REDUCE to hand-translate into Java and which to mechanically convert; it is very likely that mechanical translation will be much easier to perform on purely numeric functions than on more complicated ones. Should this be the case, I will have to investigate the potential trade-off between speed of translation and the associated performance change of manually versus mechanically converted code.

The code for CSL should provide a suitable base from which to start making improvements to JLisp; my supervisor wrote CSL and later optimised it in the manner that I am now proposing for this project. Indeed, without access to such a critically useful resource, the scope of the task would almost certainly have been too great for the amount of time and importance it has been allocated.

I will develop a testing framework as the project progresses so that I can continuously assess the performance change resulting from my modifications. This is an important step because it will let me determine at which point the improvements are sufficiently marginal that I can shift my focus to extension work, and will also enable me to provide a comprehensive presentation of the performance changes realised.

Being a substantial piece of software, it is expected that during the course of the project I will discover bugs in the REDUCE source code. Should this happen, I will try (within reason) to fix them, and at the very least alert my supervisor to their presence.

Much of the work will involve programming in Java and C, or reading existing such code. Fortunately I am familiar enough with both languages that I will

not have to devote a lot of time to related learning. However, although I encountered Lisp in Part IB lectures, my experience with it is very limited and I will need to spend time developing a better understanding of it in order to work on this project.

Resources Required

I will primarily work on the project using my own PC, which runs Windows 7 with Debian Linux in a virtual machine.

I have set up a Subversion repository which will provide cloud backup for my code, as well as offering easy access for my supervisor.

Two copies of REDUCE will be stored—one for my modified version and one for the existing version—so that I can compare them when testing and evaluating. This will require around 50MB storage space, which should not be a problem.

Since my plan is to translate the most frequently called functions in REDUCE, an important requirement is that I identify them through profiling a realistic corpus. REDUCE includes an extremely comprehensive set of scripts that test and demonstrate its capabilities. I am confident, after discussion with my supervisor, that these scripts will be more than sufficient. In addition, they will assist in verifying that any modifications made to JLisp preserve the correct behaviour of the program.

Should my computer suddenly fail, I will be able to continue working using the MCS machines in the Computer Laboratory.

Success Criterion for the Main Result

My project will be successful if I can translate a small selection of basic functions in REDUCE to Java and preserve the correctness of the existing version of the program. This can be determined by running the included test scripts. In addition, I should be able to produce a full evaluation explaining the change in performance arising from any modifications that I make. This should include a comparison of test results from the original and modified versions of REDUCE.

Possible Extensions

After satisfying the main criterion for success, I would seek to implement a larger selection of the most frequently used functions in REDUCE. The next step would be to have converted all of the top 5% most called functions (aside from the problematic ones mentioned earlier), again with correctness preserved. This could also include a comparison of the performance of each of the individual REDUCE tests between the existing and modified versions.

As mentioned in the introduction, a very attractive property of the Java version of REDUCE is that it can be converted to Javascript and run in a browser. Exploring the use of this feature and potentially making improvements to it could prove to be a useful activity.

Timetable

1. **October 27th - November 9th** Install REDUCE and set up a repository with two copies of the source code. Familiarise myself with the code, especially JLisp. Perform free-standing tests to establish the current capabilities of REDUCE.
2. **November 10th - November 23rd** Start profiling REDUCE to identify the most active regions of the program. Look in detail at the CSL code and strategies used to improve performance. Try applying these in manual translation of small examples to Java. Start developing testing framework for the project.
3. **November 24th - December 7th** Start on mechanical conversion to Java of pure numeric code, using existing profiling analysis to select code for conversion. Continue work on test framework.
4. **December 8th - January 11th** Continue core translation work, paying attention to test results to decide between manual and mechanical conversion for each part of the REDUCE code in question.
5. **January 12th - January 25th** Write progress report and ensure that basic parts of the project can be demonstrated to overseers as part of the presentation.
6. **January 26th - February 15th** Translation of more challenging functions in REDUCE. Start writing dissertation.

7. **February 16th - March 8th** Start work on extensions if appropriate. Work for core project should be mostly written up.
8. **March 9th - 19th April** Finish work on extensions. Hand in dissertation draft.
9. **20th April - May 3rd** Perform full set of timing and correctness tests on both the original and modified versions of REDUCE. A full performance evaluation should be produced by the end of this period.
10. **May 4th - May 15th** Finish dissertation. Hand in early to provide a buffer period for any unforseen circumstances and to leave time for crucial exam revision.