



6CCS3PRJ Final Year RIMPiler

Final Project Report

Author: Aidan Dakhama

Supervisor: Professor Maribel Fernandez

Student ID: 21034945

March 18, 2024

Abstract

The abstract is a very brief summary of the report's contents. It should be about half-a-page long. Somebody unfamiliar with your project should have a good idea of what your work is about by reading the abstract alone.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary.
I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Aidan Dakhama

March 18, 2024

Acknowledgements

It is usual to thank those individuals who have provided particularly useful assistance, technical or otherwise, during your project. Your supervisor will obviously be pleased to be acknowledged as he or she will have invested quite a lot of time overseeing your progress.

Contents

1	Introduction	3
2	Context	4
2.1	Reversible Computing	4
2.2	Programming Languages	4
2.3	Compilers, Interpreters, & Abstract Machines	5
2.4	Reversible Languages	7
3	Requirements	8
3.1	Functional Requirements	8
3.2	Non-Functional Requirements	9
4	Specification	10
4.1	RIMP's Concrete Syntax	10
4.2	RIMP's Semantics	10
4.3	RIMP's Type System	10
4.4	Semantic Transformations for Reversibility	10
4.5	Reverse Function	10
4.6	Abstract Machine	10
5	Design & Implementation	11
5.1	Architecture	11
5.2	Lexing Algorithm	11
5.3	Parsing Algorithm	11
5.4	Semantic Transformations for Reversibility	11
5.5	Reverse Function	12
5.6	Interpreter	13
5.7	Abstract Machine	13
5.8	Compiler	13
6	Legal, Social, Ethical & Professional Issues	14
7	Evaluation	15
7.1	Testing	15
7.2	Requirements	15
7.3	Limitations	15

8 Conclusion and Future Work	16
9 Bibliography	17
A Extra Information	18
B User Guide	19
C Source Code	20

Chapter 1

Introduction

Chapter 2

Context

2.1 Reversible Computing

Reversible computing is a paradigm of computing where computation is both forward deterministic, and backward deterministic [4]. This allows a reversible program to be run forwards, stopped at any point, and then run backwards, completely reconstructing the state in which it started. Unlike traditional computing, which is not backward deterministic, which inevitably leads to information loss, which is dissipated as heat energy [3]. Reversible computing allows the conservation of this information, and therefore energy, throughout computation [1].

Research in this area is diverse, but can be split by their level of abstraction. While there is no widespread use of reversible hardware, there has still be significant research into programming abstractions which may run on top of them.

2.2 Programming Languages

Programming languages serve as abstractions over the underlying computational model, providing a structure easier for humans to reason and communicate instructions with each other and the computer.

The structure of a programming language is largely dictated by the underlying model. Imperative languages abstract over the Turing Machine [5], while Functional languages abstract over Lambda Calculus [2]. Additionally, programming languages are also impacted by the level of abstraction they wish to provide, being split into high-level and low-level languages. A low-level language is one which directly follows the underlying model, without hiding much detail

from the user, while high-level languages aim to hide much of this detail.

The area of programming languages is an open and active area of research. This includes type systems, and memory management.

Give details here, reference some survey papers maybe

2.3 Compilers, Interpreters, & Abstract Machines

Programming languages often cannot be run directly on hardware, and need some form of translation to occur, this can happen through many methods, such as compilers, interpreters, and abstract machines. While they differ in how the program is executed, they share many components.

Additionally, each provide benefits and drawbacks, but work well to complement each other throughout the development process.

2.3.1 Lexing

Lexing is typically performed first in the context of compilation and interpreting. Lexing is the process where a lexer takes some text input, and outputs a list of tokens. Each token corresponds to a part of the original text which makes sense to group together. For example, we may group letters comprising keywords into a keyword token.

This is frequently a crucial step in the process of working with programming languages, as it permits the later stages of the process to disregard irrelevant input elements, such as whitespace.

2.3.2 Parsing

Parsing is the process of transforming some source, typically a token stream, into a hierarchical structure such as an abstract syntax tree (AST). This is done following some specification for the grammar of the language.

The grammar defines what is a valid set of symbols that can appear, and the order in which they are allowed to appear.

I have references somewhere for both of these

2.3.3 Compilers

A compiler is a system which translates some source language to a target language, allowing the program to be directly executed on some system. This is done through some kind of

transformation from a source code language to a machine code language, which can then be directly executed on hardware.

During the process of compilation, there are typically other processes taking place. Optimisation is the process of producing some new program which is semantically identical to the original, however, minimises some property of it, such as run time, or power consumption.

Typically, compilers are placed at the end of the development cycle. Before a program is passed to its users, it will be compiled. This is done for many reasons, such as:

1. Ease of use: The compiled target can often be run directly.
2. Efficiency and speed: The compiler typically optimises many factors of the program.
3. Protecting IP: The original human-readable source is mutated, typically to an unrecognisable degree.

Add citations for each point

2.3.4 Interpreters

Interpreters are programs which immediately evaluate a program they are given. Unlike compilers, they do not produce a new output which can be run after. Instead, interpreters usually produce an abstract syntax tree and evaluate directly from that.

Interpreters are, for this reason, faster to start, and so can be a useful tool for fast iteration while developing.

However, as interpreters are expected to immediately execute the program they are given, they don't have time to optimise like compilers do, and so are typically far slower to run the same program through when compared to compilers.

2.3.5 Abstract Machines

Like interpreters, abstract machines evaluate program representations, however, they diverge in their operational methodologies, and the level of abstraction. Abstract machines function at a lower level of abstraction compared to interpreters. This difference makes abstract machines particularly valuable in debugging, as they allow the developer to observe small changes in program state after each step is evaluated.

2.4 Reversible Languages

Reversible languages are languages which in some way enforce the constraints of reversible computing. That is, a program in a reversible language can be stopped at some point, and returned to the exact same initial state.

2.4.1 Methods of Reversibility

There are several methods languages can employ to achieve reversibility. One such method is state capture, this is where all relevant state is recorded at each execution step, so when reversing the computation, you simply go back to the last command, and refresh the state to the previous point.

Another way to achieve reversibility is to prevent any executions which are not inherently backwards deterministic. For example, in many languages it is common to be able to overwrite a variable with a new value, such as $x = 5$. However, using this method, we would prohibit use of this kind of assignment as it would lose any state associated with x . Instead we would prefer statements of the form $x += 5$.

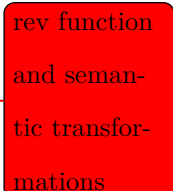
Another consideration of reversible languages is what we refer to as transparency, this can be thought of in a way as how high-level they are. Languages can be classed as transparent if they reflect the underlying reversible model being used, and opaque if they hide this. Essentially, if the developer has to be aware that they are targeting a reversible model of computation, then the language is transparent (or low-level), however, if not it is opaque (or high-level).

2.4.2 Janus

What is it? Why benefits does it have? What has been researched so far? What are examples of reversible languages? What gaps remain?

2.4.3 Hermes

2.4.4 RIMP



rev function
and seman-
tic transfor-
mations

Chapter 3

Requirements

The following lists the metrics and functionalities which this project aims to fulfil, and by which this project will be evaluated. Within the specification, RIMPiler will be used to refer to the software produced in this project.

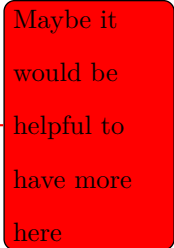
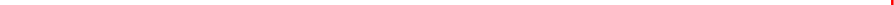
3.1 Functional Requirements

1. RIMPiler must produce a token stream from a valid RIMP source file.
2. RIMPiler must produce a valid abstract syntax tree from a token stream produced by RIMPiler.
3. RIMPiler must be able to perform transformations on the abstract syntax tree to ensure the reversibility of the language.
4. RIMPiler must be able to produce an abstract syntax tree corresponding to the reverse of another abstract syntax tree.
5. RIMPiler must be able to evaluate an abstract syntax tree, providing information about the value of variables until the reversal point.
6. RIMPiler must be able to be run through an abstract machine.
7. The RIMPiler abstract machine must allow to step forwards and backwards.
8. RIMPiler must be able to compile to the JVM providing information about the value of variables until the reversal point.


Discuss the point of the project here to motivate why these are the requirements set

3.2 Non-Functional Requirements

1. RIMPiler should compile short programs within 10 seconds on a modern desktop computer.
2. RIMPiler should give some level of feedback when things go wrong to help guide the user to the issue.



Maybe it
would be
helpful to
have more
here



quantify

Chapter 4

Specification

4.1 RIMP's Concrete Syntax

What is it? Why is it like this? How it differs from others? How it achieves the requirements?

4.2 RIMP's Semantics

What is it? Why is it like this? How it differs from others? How it achieves the requirements?

4.3 RIMP's Type System

What is it? Why is it like this? How it differs from others? How it achieves the requirements?

4.4 Semantic Transformations for Reversibility

What is it? Why is it like this? How it differs from others? How it achieves the requirements?

4.5 Reverse Function

What is it? Why is it like this? How it differs from others? How it achieves the requirements?

4.6 Abstract Machine

What is it? Why is it like this? How it differs from others? How it achieves the requirements?

Chapter 5

Design & Implementation

5.1 Architecture

What are the components? How are they connected? Why is it done like this? How does it help meet the requirements?

5.2 Lexing Algorithm

What is the theory of the lexing algorithm used? What are alternatives? Why use this method? How was it actually implemented?

5.3 Parsing Algorithm

What is the theory of the parsing algorithm used? What are alternatives? Why not use parser generators? Why use this method? How was it actually implemented?

5.4 Semantic Transformations for Reversibility

What is the theory used? How does it differ from RIMPs paper? What motivates that? What were some of the compromises with it? How was it implemented? Does it meet the needs of the requirements?

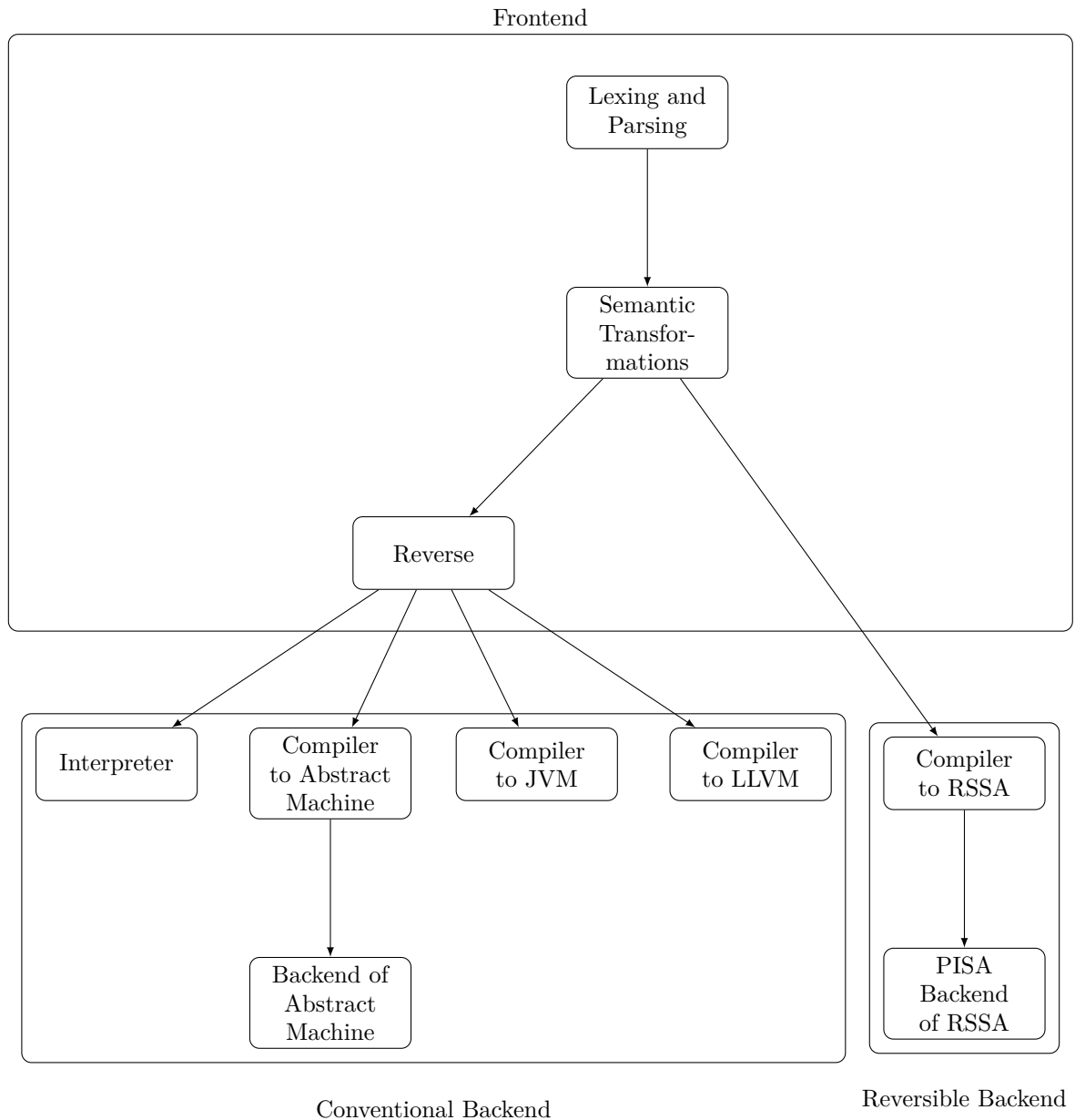


Figure 5.1: High level overview of the system architecture

5.5 Reverse Function

What is the theory used? How does it differ from RIMPs paper? What motivates that? What were some of the compromises with it? How was it implemented? Does it meet the needs of the requirements?

5.6 Interpreter

What are some ways of implementing an interpreter/evaluator? How was it implemented? Why?

5.7 Abstract Machine

What are ways of implementing an abstract machine? Why not use a stack machine implementation? Why not compile to an intermediary? How was it done?

5.8 Compiler

What ways are there to implement a compiler? What targets are there? Why chose this specific compiler method target combination? How was it done?

Chapter 6

Legal, Social, Ethical & Professional Issues

What license is it? How does this help?

How is it written? How does this help?

Low power computing. How does that help low income and the environment?

Chapter 7

Evaluation

7.1 Testing

How was testing approached? How effective was it? Why not use another approach? Diff testing, fuzzing, AI?

7.2 Requirements

Were the requirements met? To what standard? What was not met?

7.3 Limitations

What are some issues with any of the requirements? Are there any requirements that were met, but jeopardised something else? Or any requirements which were fundamentally bad?

Chapter 8

Conclusion and Future Work

What is demonstrated here? How is it unique? What could be done better? What are some avenues for future exploration?

Chapter 9

Bibliography

- [1] Charles H. Bennett and Rolf Landauer. The fundamental physical limits of computation. *Scientific American*, 253(1):48–57, 1985.
- [2] Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, pages 346–366, 1932.
- [3] R. Landauer. Irreversibility and Heat Generation in the Computing Process. *IBM Journal of Research and Development*, 5(3):183–191, July 1961.
- [4] Kalyan S. Perumalla. *Introduction to Reversible Computing*. CRC Press, September 2013.
- [5] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

Appendix A

Extra Information

Appendix B

User Guide

Appendix C

Source Code