Static Analysis of Programs using Semirings

Louise North

Bachelor of Science in Computer Science with Honours
The University of Bath
May 2017

Signed:

Static Analysis of Programs using Semirings

Submitted by: Louise North

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

Abstract

This project aims to make use of a mathematical structure, a semiring, in order to provide a system that performs multiple data-flow analyses using the same core theory. Firstly, there is an investigation into the current work in the domain. Current literature and existing software tools are evaluated, with possible opportunities for development identified; including the potential to analyse energy costs in code. Following the literature review, the requirements, design and implementation of an adaptable, command-line tool that solves Reachability, Live Variables and Reaching Definitions Analysis are presented.

The experimental stage of the project looks at possible analyses that can be created in order to statically analyse energy costs within programs. The beginning of a theoretical approach to analysing energy costs, using the adaptable framework provided by the core implementation of the system, is proposed. The solutions to all analyses are described and the project is finally concluded with a reflection on the work produced as well as identification of potential future work within the domain.

Contents

List of Figures

List of Tables

Acknowledgements

I would first like to thank my dissertation supervisor Prof. Guy McCusker for all of the guidance, expertise and confidence he provided throughout the project. It was a genuine pleasure to be able to complete my final year project under the supervision of someone with such competence and enthusiasm for work within the subject area. Prof. McCusker allowed the project to develop based on my own interests, but was always there to steer me in the right direction when I needed it. Thank you for all of your time and help; my final year project has been far more enjoyable than I could have anticipated.

I would also like to thank my friends and family who have provided unconditional support throughout the project. My course mates, in particular, were always there experiencing the highs and lows of the rollercoaster ride with me. I'd especially like to mention my housemate and dear friend Lucy who has provided encouragement and helped maintain sanity throughout the year in the form of Kinder Bueno chocolate bars, coffee and hysterical laughing fits.

Finally, I'd like to thank the entire Computer Science department at the University of Bath for such an enjoyable and academically rewarding few years. Without the knowledge and experience I'd been given in the earlier years of my time at university, this project would not have developed into what it is.

Chapter 1

Introduction

Static program analysis has been around for many years, with a mature theoretical history as well as implementations in systems such as compilers and integrated development environments (IDEs). Static program analysis means that the analysis is performed without execution of the code and is generally used to improve and validate the code quality by focusing on code structure. When we statically analyse programs there are two properties with which we are often concerned:

- **Correctness:** ensuring that the program does what it is supposed to do
- **Optimisation:** focusing on program efficiency and performance

This project focuses on the latter, looking at optimisation of programs, in particular by focusing on data-flow within programs. So why do we care about optimising code?

By optimising code there are many enhancements that we can benefit from, including quicker execution time and improved energy efficiency. For example, we can use an analysis called Live Variables Analysis, an analysis we shall look at as part of this project, to identify, at each line within some code, which variables could be read before their next write. This sort of information is used by compilers to provide optimised performance and is not a trivial or quick analysis to perform.

At the foundation of program analysis, there is a large amount of mathematics and theory to understand. This project focuses on a formal method of analysis, data-flow analysis, through the use of a mathematical structure called a semiring. In the 1970s, Lehmann (1977) studied how semirings are useful within this domain, by holding certain properties that are extremely beneficial as we shall explore throughout this project. The main point is that semirings can be used to solve the data-flow equations that are used to represent the optimisation problems we consider.

By automating these fairly complex existing analyses, we provide a tool that could be integrated into a larger system for use on a larger scale. Furthermore, by using the same core theory as the foundation to each of the analyses implemented, we provide a system that is

flexible enough to adapt to different problems as well as extend to new ones.

The project begins as a relatively normal software development project, aside from the fact that it is not focused on usability but more so the theoretical background. At this stage, we focus on existing work within the domain in order to build an adaptable, automated tool to perform several mathematical analyses by following the traditional software lifecycle. The outcome of this is then used as a framework for some experimental research.

In the experimental section of the project we progress from the purely structural view of control-flow in programs and intertwine ideas from the real world, looking at how we can use the already implemented adaptable analysis framework to provide new information. We are now interested in more than just theoretical analyses, but consider how real-life issues, including energy costs and cached vs. non-cached variables, can be analysed within code to provide, at least the beginning of, a new theoretical approach to analysing and quantifying real-world costs of program execution.

When looking at energy costs we are interested in the amount of physical energy that is required to execute code. This is especially interesting nowadays since hardware is reaching its peak and is becoming much cheaper to purchase, meaning that the focus is shifting toward optimisation of software to improve efficiency and reduce cost. Some attempts have been made to analyse energy within code, as is examined in the literature, however this is all relatively new and fairly unsuccessful. By adapting the existing analyses, we look at how it may be possible to consider some new analysis examining these real-life concerns.

As part of this experimental analysis, we take the novelty a step further by considering cached versus non-cached variables. By introducing the concept of cached variables we can imagine different associated energy costs leading to a more useful analysis. This idea becomes more prominent as the development of technologies leads to smaller devices reliant on cloud computing services. A smaller amount of onboard memory means that data often needs to be accessed from remote destinations, such as the cloud. We can now consider onboard memory relatively free in comparison to the cost of having to retrieve data remotely.

Overall, there are several significant contributions of this project. Firstly we build a flexible control-flow analysis framework based on semirings. We then use this core framework to implement several previously investigated analyses including Reachability, Live Variables and Reaching Definitions Analysis, all of which are explained in detail in the literature review. Finally, we propose a novel analysis which provides new information, via a theoretical approach, about energy costs within programs.

Throughout the core of the project, and the experimental section, the main aim of the project remains consistent: to use semirings as a foundation to implement analyses for optimisations within program code.

Chapter 2

Literature Survey

2.1 Introduction

Program analysis has been an area of research and evolution for many years, with a great deal of development in the 1970s onwards. There is a mature history within this domain, with a large number of academic papers Landi (1992), Tarjan (1981*b*), Allen & Cocke (1976), Kildall (1973), Bergeretti & Carré (1985), Kam & Ullman (1976), Farrow et al. (1976), Fosdick & Osterweil (1976), Tarjan (1981*a*), Dolan (2013), Lehmann (1977), Abdali & Saunders (1985), Mohri (2002), and comprehensive text books Nielson et al. (1999), Golan (2013), outlining the fundamental topics that will be necessary to understand for the progression of this project. Much of the research that has been carried out focuses on different analysis methodologies with the supporting mathematics that underlies these techniques. More recently, many static analysis tools have been implemented and released for common use in compilers; identifying bugs in code and to aid with optimisation of code.

We shall identify and discuss the elements that are fundamental within this project, to show understanding of the existing work within the domain and so not to reinvent the wheel. In order to provide clarity and a strong foundation of understanding, we shall thoroughly evaluate literature covering a range of relevant topics including:

- Defining programming languages and program structure
- Graphs and how they can be used to represent a program
- Data-flow analysis as a specific methodology
- Existing algorithms for evaluating programs
- Semirings and their applications
- Existing software tools

Furthermore, by attempting to add something novel within a domain with such a mature history, we shall look at assigning real-life costs within the analysis. By 'real-life' costs we mean

costs that are external to the code itself but that are dependent on the code and that must be considered outside of the syntactical elements of the program. For example, we shall consider energy usage when running a program. This is not directly part of the code but we can attempt to associate energy consumption to blocks of code in order to evaluate total energy usage when running certain programs. This has limited literature, with more recent investigation as seen in Liqat et al. (2013) and Schubert et al. (2012). We will identify existing approaches and consider the reasons why real-life costs have not been researched and implemented overly successfully so far.

2.2 Parsing Languages

Before analysis can be performed on any program, the code needs to be parsed. In order to parse a language we must understand how it is defined; we must understand the clear set of rules that are used to generate the given language. In order to keep things simple when explaining the fundamentals, we often define a simple imperative language, and associate data flow information with individual statements.

A program is made up of different elements including variables and expressions. These may include empty statements, assignment statements, sequences of statements, conditional statements and repetitive statements. It is necessary to understand these different types of statements in order to be able to evaluate a program, its behaviour and the effect of the statements.

We shall look at an example of a definition for an abstract syntax describing a simple imperative 'while' language, as provided in Nielson et al. (1999). It is common to consider a defined set of statements based on the syntactic categories:

$$\begin{array}{rcll} a & \in & \textbf{AExp} & \text{arithmetic expressions} \\ b & \in & \textbf{BExp} & \text{boolean expressions} \\ S & \in & \textbf{Stmt} & \text{statements} \end{array}$$

As well as a set of values based on their categories:

$$\begin{array}{rcll} x, y & \in & \textbf{Var} & \text{variables} \\ n & \in & \textbf{Num} & \text{numerals} \\ l & \in & \textbf{Lab} & \text{labels} \end{array}$$

And a set of operators based on similar categories:

$$\begin{array}{rcll} op_a & \in & Op_a & \text{arithmetic operators} \\ op_b & \in & Op_b & \text{boolean operators} \\ op_r & \in & Op_r & \text{relational operators} \end{array}$$

Once we have categories of the sets of expressions, values and operators, it is possible to define the syntax of a language. For example, in this case, and as provided in Nielson et al. (1999):

$$
\begin{array}{lcl}
a & ::= & x \mid n \mid a_1\ op_a\ a_2 \\
b & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{not}\ b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2 \\
S & ::= & [x\ :=\ a]^l \mid [\texttt{skip}]^l \mid S_1; S_2 \mid \texttt{if}\ [b]^l\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ [b]^l\ \texttt{do}\ S
\end{array}
$$

Using these defined categories with defined behaviours, it is possible to parse a language. These rules can be applied recursively allowing for an infinite number of possible statements within the language. To further define this language, we would need to formally define what each item is, for example we would need to formally define the set of arithmetic operators.

A syntax such as the one described above may be considered as a parse tree for the language, as seen in Nielson et al. (1999). A parse tree is simply a graphical representation of a derivation of a sequence, such as a statement in a language. We can now understand the behaviour of a program, using the syntax to decompose the code into atomic elements that interoperate based on these syntactical rules.

It is necessary to notice that there may be different orders in which a statement may be parsed. In some cases this is important and in others it is not. Consider the expression `2+3*4` as an example. If we are applying standard rules of mathematics then the `3*4` should be calculated and then the result added to the `2`. In the case of `2+2+2` it does not make a difference, however, there would be two different parse trees representing it. For these reasons, we need to ensure grammars are unambiguous resulting in unique parse trees for every statement.

Provided is an example of how an expression may be represented as a parse tree, using the standard rules of mathematics. The expression we shall consider is: `z = x + (2 * y)` and the corresponding parse tree is show in Figure 2.1.

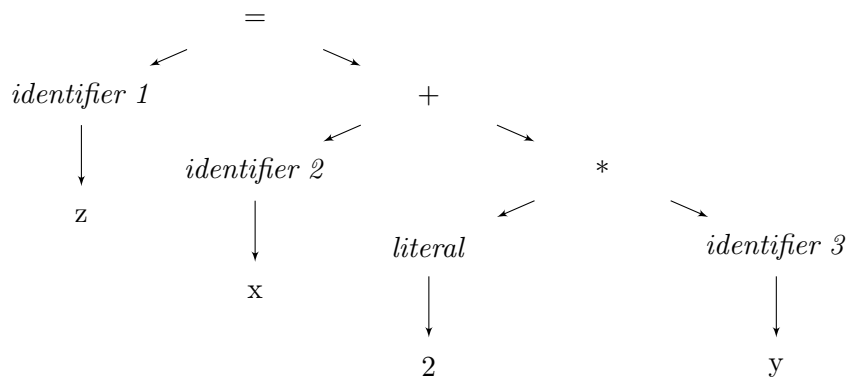

Figure 2.1: An example of a parse tree

Parsing can be carried out using a top-down or a bottom-up approach. Top-down parsing is carried out by starting at the root node and expanding and can be described as finding the leftmost derivation of an input string. Bottom-up parsing considers the leaf nodes first, and

'reduces' a sequence to the start symbol of the grammar. The approach to parsing that is most appropriate, top-down or bottom-up, often depends on the type of grammar that is being parsed.

2.2.1 Parsing Tools

There are many existing parsing tools that have been implemented in order to allow programmers to easily parse code, often using a top-down approach. Many programming languages have associated parsing libraries that allow programmers to generate parsers easily. Some common examples include YACC for C, JavaCC for Java and Parsec for Haskell. These tools use some chosen parsing algorithm, require a formal definition of an input language and produce an output, the parser, within the specified language.

2.3 Graphs

2.3.1 Graph Definitions

Since graph theory has some non-standard terminology, we shall detail some of the common definitions. The following definitions have been composed from the work in Fosdick & Osterweil (1976), Farrow et al. (1976) and Allen & Cocke (1976).

A graph is a finite set of nodes joined by edges and a directed graph has the extra condition that the edges between nodes are directed. Formally, we say a graph G is represented by $G(N, E)$ where N is the set of nodes in the graph and E is the set of pairs of nodes that are joined by the edges in the graph.

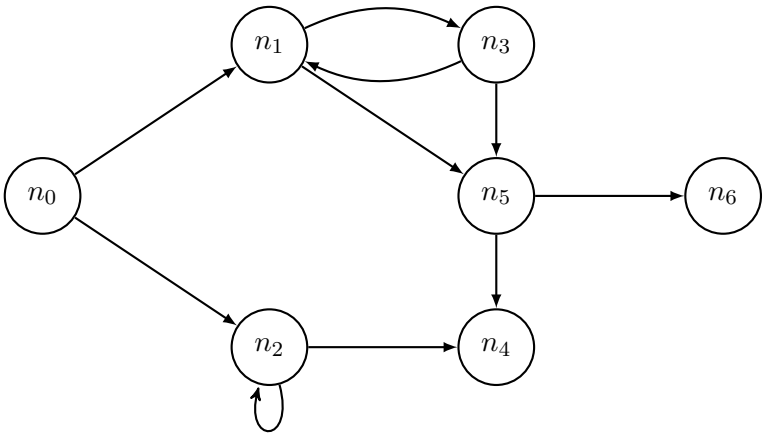

Figure 2.2: An example of a directed graph

Using the definitions as reviewed from the papers Fosdick & Osterweil (1976) and Farrow et al. (1976), we can define the graph displayed in Figure 2.2.

- The set N, of nodes, is $\{n_0, n_1, n_2, n_3, n_4, n_5, n_6\}$
- The set E, of edges, is $\{(n_0, n_1), (n_0, n_2), (n_1, n_3), (n_1, n_5), (n_2, n_2), (n_2, n_4), (n_3, n_1), (n_3, n_5), (n_5, n_4), (n_5, n_6)\}$

A node has an in-degree, the number of predecessors of a node, and an out-degree, the number of successors of a node. For an edge (n_0, n_1): n_0 is a predecessor of n_1 and n_1 is a successor of n_0. A node may be a predecessor and successor of itself, if an edge loops. An entry node to a graph is a node with no predecessors and an exit node of a graph is a node with no successors. In Figure 2.2, the entry node is n_0 and the exit nodes are n_4 and n_6.

A path is a sequence of nodes with the condition that every edge is a pair in the set of nodes and each pair is adjacent. An example of a path in the directed graph in Figure 2.2 could be $P = ((n_0, n_1), (n_1, n_3), (n_3, n_1), (n_1, n_5), (n_5, n_6))$. We define the length of a path as the number of edges in the sequence, otherwise defined as the number of nodes visited in the path minus 1. For the given example of the path P in Figure 2.2, the length of the path is 5. A cycle is when there are a set of edges in a graph that return to a node already in the path. In Figure 2.2 there are two cycles: the cycle between n_1 and n_3 and the cycle where n_2 loops back to itself.

A graph may also be weighted which means that there is some value associated with each edge. This value could represent a range of things. In the context of this project, a weight may be used to assign a real-life cost of a transition between nodes.

A tree is defined $T(N, E)$, like the general definition of a graph. A tree is a specific type of graph where every node has an in-degree of one (a unique predecessor), except for the entry node which has an in-degree of 0. The entry node of a tree is defined as the root, and any node in a tree with no successors is called a leaf. In a tree, there is only ever one path to each node in the tree from the root (entry node) and because of this we can discuss the idea of ancestors and descendants. An ancestor of a node, n, is any node that appears in a path before the node in question, while a descendant is a node that appears after.

Trees become particularly useful when it comes to parsing a language. A syntax can be represented as a tree as shown in the previous section. Using the parse tree in Figure 2.1 as an example, we can define the '=' as the root node and the z, x, 2 and y nodes as leaves of the tree. The leaves are the most atomic elements within the tree structure.

2.3.2 Representing a Program as a Graph

For program analysis, it is useful to represent a program as a graph; allowing us to visualise code as a flow of control. Work in this domain originates as far back as 1947, when work was carried out by Goldstine and Von Neumann as identified in Fosdick & Osterweil (1976), investigating the idea of flowcharts representing the flow of control in a program. Following this, in the 1960s, the idea of applying graph theory to programming arose and this work was carried out by Karl; as identified in Fosdick & Osterweil (1976). More recently, the concept of using graphs to represent programs has been developed further - with many people using this as the basis of research and investigation into program analysis. The following discussion is based on

the work in Fosdick & Osterweil (1976) and Farrow et al. (1976), describing and illustrating how we can represent a program as a graph.

If we consider a program to have only one entry point, which may be true or an abstraction of the truth, a flow graph must have only one entry node. Of course, since there are often many paths that a program can take, giving us a range of results, there may be more than one exit node. Due to the nature of programs, there must always be at least one path from the entry node to every exit node in the graph, since it is impossible to reach an exit node without having begun at the input of the program. Using the previous definition of a graph, we can define a control flow graph as a triple $F(N, E, n_0)$, where the extra element n_0 is the unique entry node which must present be in the set N of all nodes in the graph and which is the successor of all other nodes in the graph.

There are different ways in which a program can be represented as a graph, however, the most common way is to use nodes to represent statements, or a basic block, within a program, and edges to represent the transitions from one block to another. In Tarjan (1981*b*), a basic block of a program is defined as a block of 'consecutive statements' having a single entry and a single exit.

To provide an example, we shall consider the factorial program example as provided in 'Principles of Program Analysis' Nielson et al. (1999), with the pseudocode as shown in Figure 2.3.

$y \leftarrow x$
$z \leftarrow 1$
while $y > 1$ **do**
    $z \leftarrow z * y$
    $y \leftarrow y - 1$
end while
$y \leftarrow 0$

Figure 2.3: Pseudocode for a factorial program

This program has the equivalent control-flow graph as displayed in Figure 2.4.

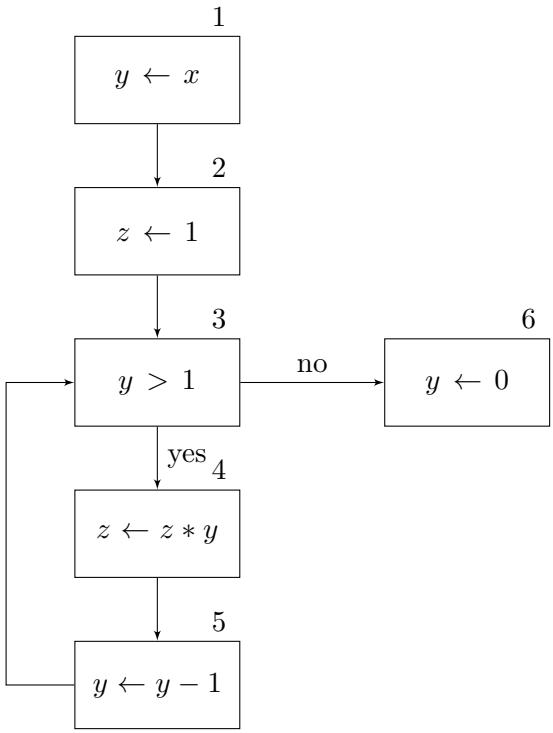

Figure 2.4: Control-flow graph for a factorial program

Using the control-flow graph of a program, it is easy to visualise properties of the program. For example, we can see the start node and any end nodes, as well as cycles and conditionals. Representing a program as a graph provides a useful and standard structure for performing analysis over.

2.4 Data-Flow Analysis: A Specific Methodology

There are many different approaches to static program analysis and data-flow analysis is one of the most traditional ones which has a mature history and features heavily within the domain of compilers Nielson et al. (1999). Data-flow analysis is based on the previously discussed idea of representing a program as a graph, namely a control flow graph, where the nodes are the elementary blocks and the edges represent the flow of control between the blocks Nielson et al. (1999). Nielson's book Nielson et al. (1999) gives an overview of a variety of methods, dedicating an entire chapter to data-flow analysis. There are a range of papers, dating from the 1970s onwards, that provide approaches to data flow analysis too. Some of the papers discuss the general approach to data-flow analysis, while some extend the idea and explain how semirings can be used to perform data-flow analysis. The latter shall be discussed in the following section, while a general idea of data-flow analysis as a technique will be outlined here.

2.4.1 Limitations

There are certain limitations with control-flow graphs for the analysis of programs and we will identify and discuss these issues before explaining approaches, as they must be taken into consideration as part of these approaches. Kildall explains in Kildall (1973) that loops cause issues when using a constant propagation method for data-flow analysis. Constant propagation is concerned with evaluating the values of each variable in a program at each node in the graph, by looking at each path to a certain node. It is clear that this can become infinitely long when loops are involved and therefore Kildall (1973) defines a global analysis algorithm that can be used to compute analysis in a finite number of steps by using approximations.

In Landi (1992), the difficulties with carrying out program analysis are explained, identifying that the analysis framework used is not the problem but rather the structure of programs themselves. Conditionals mean that even in a simple procedural program, there are several possible paths but not all of which correspond to an execution. Since static analysis is not recursive, we must make safe, but not always valid, assumptions such as assuming that all paths through a program are executable. This simplifies the analysis and allows for general application providing useful but not always precise information.

With even more popular complex languages, with more complicated structures such as recursion, static analysis becomes even more difficult. Landi (1992) identifies a common issue called alias, which occurs when multiple variables exist in the execution of a program pointing to the same location in memory. Two types of aliasing are defined in Landi (1992):

- *may alias:* find the aliases that occur during some execution of the program
- *must alias:* find the aliases that occur on all executions of the program

Finding all of the aliases could result in a set of infinite size. In order to get around this issue, an analysis has to make assumptions such as assuming that pointers to values are distinct when, in fact, they are actually equal.

Perhaps the most fundamental point to acknowledge is that most analyses that we interested in are, themselves, computationally undecidable. In order to provide useful information about properties in which we are interested, it is necessary to accept approximations, so not to prevent ourselves from being able to carry out the analysis at all. In Ramalingam (1994), it is established that 'alias analysis is a prerequisite for performing most of the common analyses' and that, in itself, is enough to suggest that even the most common analyses will require some assumptions to be made.

2.4.2 Approaches and Algorithms

The literature that has been selected focuses on data-flow analysis for optimisation of programs. Different algorithms can be used to determine different results, based on what property we may be interested in. Most resources, in particular Fosdick & Osterweil (1976), focus on the idea of viewing a program as a sequence of events, which can be represented as a graph, and evaluating

the effect of these events, how they affect values throughout the program and how values affect the flow of the program. Tarjan (1981*b*) states that we are interested in determining for each basic block (represented as a node in the control flow graph) facts which must be true on entry and exit of the block, regardless of the actual path of execution.

The Equational Approach for Reaching Definitions

Reaching Definitions Analysis is concerned with determining 'for each program point, which assignments may have been made and not overwritten, when program execution reaches this point along some path' Nielson et al. (1999). In Nielson et al. (1999) an overview of the equational approach to data-flow analysis is provided. This is achieved by 'extracting a number of equations from a program'. Nielson explains two classes of equations:

- equations relating the exit information of a node to entry information for the same node
- equations relating the entry information of a node to exit information of nodes from which there is an edge to a node of interest

Using the factorial program given in Figure 2.3, and the example given in Nielson's book Nielson et al. (1999), we can compose the set of equations used for evaluating reaching definitions. We get the following equations relating to exit information:

$$\begin{aligned}
RD_{exit}(1) &= (RD_{entry}(1) \setminus \{y, l | l \in \mathbf{Lab}\}) \cup \{(y, 1)\} \\
RD_{exit}(2) &= (RD_{entry}(2) \setminus \{z, l | l \in \mathbf{Lab}\}) \cup \{(z, 2)\} \\
RD_{exit}(3) &= RD_{entry}(3) \\
RD_{exit}(4) &= (RD_{entry}(4) \setminus \{z, l | l \in \mathbf{Lab}\}) \cup \{(z, 4)\} \\
RD_{exit}(5) &= (RD_{entry}(5) \setminus \{y, l | l \in \mathbf{Lab}\}) \cup \{(y, 5)\} \\
RD_{exit}(6) &= (RD_{entry}(6) \setminus \{y, l | l \in \mathbf{Lab}\}) \cup \{(y, 6)\}
\end{aligned}$$

This set of exit equations say that on exit from a basic block in the control flow graph, the reaching definitions are all of those that were reaching definitions on entry to the basic block minus any that have been re-assigned plus any newly assigned values. The following equations related to entry information:

$$\begin{aligned}
RD_{entry}(1) &= \{(x, ?), (y, ?), (z, ?)\} \\
RD_{entry}(2) &= RD_{exit}(1) \\
RD_{entry}(3) &= RD_{exit}(2) \cup RD_{exit}(5) \\
RD_{entry}(4) &= RD_{exit}(3) \\
RD_{entry}(5) &= RD_{exit}(4) \\
RD_{entry}(6) &= RD_{exit}(3)
\end{aligned}$$

This set of entry equations says that on entry to a basic block, the reaching definitions are those that were reaching definitions on exit from the successor node. This provides us with a system of 12 equations; the set of equations that define some sets in terms of each other, which can be considered as a function.

The Constraint Based Approach for Reaching Definitions

Nielson et al. (1999) also outlines the constraint based approach as an alternative to the above. This approach extracts a number of inclusions (constraints) out of the program. Nielson states that naturally, and in a similar manner to the equational approach, the constraints are divided into two classes: one set of constraints for expressing the effect of elementary blocks and the other for expressing how control may flow through the program. Using the factorial program given in Figure 2.3, and the example given in Nielson's book Nielson et al. (1999), we get the following constraints related to exit information:

$$\begin{aligned}
RD_{exit}(1) &\supseteq RD_{entry}(1) \setminus \{(y,l)|l \in \mathbf{Lab}\} \\
RD_{exit}(1) &\supseteq \{(y,1)\} \\
RD_{exit}(2) &\supseteq RD_{entry}(2) \setminus \{(z,l)|l \in \mathbf{Lab}\} \\
RD_{exit}(2) &\supseteq \{(z,2)\} \\
RD_{exit}(3) &\supseteq RD_{entry}(3) \\
RD_{exit}(4) &\supseteq RD_{entry}(4) \setminus \{(z,l)|l \in \mathbf{Lab}\} \\
RD_{exit}(4) &\supseteq \{(z,4)\} \\
RD_{exit}(5) &\supseteq RD_{entry}(5) \setminus \{(y,l)|l \in \mathbf{Lab}\} \\
RD_{exit}(5) &\supseteq \{(y,5)\} \\
RD_{exit}(6) &\supseteq RD_{entry}(6) \setminus \{(y,l)|l \in \mathbf{Lab}\} \\
RD_{exit}(6) &\supseteq \{(y,6)\}
\end{aligned}$$

This set of exit constraints says that for an assignment in a basic block there is one constraint that excludes the variable from the entry of the block from reaching the exit and one constraint that includes the new assignment for the entry to following blocks. The following constraints related to entry information:

$$\begin{aligned}
RD_{entry}(1) &\supseteq \{(x,?),(y,?),(z,?)\} \\
RD_{entry}(2) &\supseteq RD_{exit}(1) \\
RD_{entry}(3) &\supseteq RD_{exit}(2) \\
RD_{entry}(3) &\supseteq RD_{exit}(5) \\
RD_{entry}(5) &\supseteq RD_{exit}(4) \\
RD_{entry}(6) &\supseteq RD_{exit}(3)
\end{aligned}$$

This set of entry constraints says that we have a constraint of the form $RD_{entry}(l) \supseteq RD_{exit}(l')$ if it is possible for control to pass from *l'* to *l*. These equations can be rearranged to group all exit constraints for one basic block into one equation. For example:

$$\begin{aligned}
RD_{exit}(1) &\supseteq RD_{entry}(1) \setminus \{(y,l)|l \in \mathbf{Lab}\} \\
RD_{exit}(1) &\supseteq \{(y,1)\}
\end{aligned}$$

can be replaced with:

$$RD_{exit}(1) \supseteq RD_{entry}(1) \setminus \{(y,l)|l \in \mathbf{Lab}\} \cup \{(y,1)\}$$

which is the same as the exit equations in the equations approach, except equality has been replaced with inclusion. Nielson et al. (1999) identifies that this means that any solution to the equational system is also a solution to the constraint system, but the opposite is not necessarily true. One of the issues with the constraint-based approach is that it can require a significant amount of work to collect all of the constraints since constraints with the same left hand side can be generated in many places throughout a program.

Reaching Definitions Analysis

Using the equations described above, we can create general data-flow equations for Reaching Definitions Analysis. The data-flow equations for a basic block S in a control-flow diagram are:

$$\begin{aligned} RD_{entry}(S) &= \bigcup_{p \in pred(S)} RD_{exit}(p) \\ RD_{exit}(S) &= RD_{gen}(S) \cup (RD_{entry}(S) - RD_{kill}(S)) \end{aligned}$$

This says that the set of reaching definitions going in to basic block S are all of the reaching definitions from the predecessors of S. It also says that the reaching definitions coming out of S are all reaching definitions of its predecessors minus those reaching definitions whose variable is killed by S, plus any new definitions generated within S. RD_{gen} and RD_{kill} are both sets of variables. RD_{gen} describes the set of variables that are generated, for example RD_{gen}(S) describes the set of variables that are generated by the basic block S. RD_{kill} describes the set of variables that are re-assigned.

These general equations can be applied to a program and rearranged to form a system of linear equations. Once we have a system of linear equations then they can be solved in order to determine the solution to the analysis. This will be described in detail in the next section.

Live Variables Analysis

Live Variables (or Liveness) Analysis calculates for each program point the variables that could potentially be read before their next write. That is, those variables that are live at exit from the basic block. In a similar manner to Reaching Definitions Analysis, we are able to create general data-flow equations for a given basic block S as follows:

$$\begin{aligned} L_{entry}(S) &= L_{gen}(S) \cup (L_{exit}(S) - L_{kill}(S)) \\ L_{exit}(S) &= \bigcup_{s' \in succ(S)} L_{entry}(s') \end{aligned}$$

where L_{gen}(S) is the set of variables that are used in S before any assignment and L_{kill}(S) are the set of variables that are assigned a value in S. If we also consider a program to terminate with no live variables, we also have the equation:

$$L_{exit}(final) = \emptyset$$

It is possible to notice some symmetry between Reaching Definitions Analysis and Live Variables Analysis in that the entry and exit equations are just swapped.

Other Analyses

Other analyses, that are outlined or mentioned as part of the literature, include:

- **Reachability Analysis** which is different to, and simpler than, Reaching Definitions Analysis. It determines whether a node can be reached from some other node in the graph. That is, that there is a sequence of adjacent edges from one node to the other.
- **Shortest-Path Analysis** which calculates the length of the shortest path between two nodes in a graph. This can be extended to determine the actual route of the shortest path between two nodes.
- **Available Expression Analysis** which calculates, for each program point, which expressions that must have already been computed, not later modified, and therefore that need not be recomputed, on all paths to that program point.
- **Definite Assignment Analysis** which is used to ensure that a variable has been assigned before it is used.
- **Constant Propagation** which is also known as constant folding, recognises constant expressions which can be computed once and used later.
- **Faint Variables Analysis** which takes Live Variables Analysis further by determining whether assignments are useless even if they are considered live.

For each of these analyses it is possible, like Reaching Definitions and Live Variables Analysis, to create a set of equations. The set of equations can be generalised and then rearranged to form the general data-flow equations for solving the analyses.

2.5 Semirings: A Useful Mathematical Structure

Semirings are a useful mathematical structure that can be used to evaluate programs. A helpful paper that brings together a variety of ideas is Dolan (2013). This paper provides an overview of essential mathematical definitions along with examples of semiring applications. There also exist some older papers which provide a more detailed view into the foundations of semirings, including a paper written by Lehmann, Lehmann (1977). This paper is one of the most foundational papers within this area as Lehmann studies how closed semirings and the closure of matrices provide a useful approach for solving analysis problems. This paper is referenced by many of the others due to its importance.

2.5.1 Mathematical Background of Semirings

Using the formal definition from Dolan (2013), a semiring is a mathematical structure consisting of a set R and two binary operations called:

- addition $+$

- multiplication $\cdot$

with the additional requirements that $(R,+)$ is commutative and that $\cdot$ distributes over $+$. Using the equations defined in Lehmann (1977) and Dolan (2013), we say that an algebra is a semiring if, and only if, the following conditions hold:

$$\begin{aligned}
a + (b + c) &= (a + b) + c && \text{(addition is associative)} \\
a + b &= b + a && \text{(addition is commutative)} \\
a + 0 &= a && \text{(is a unit for addition)} \\
a \cdot (b \cdot c) &= (a \cdot b) \cdot c && \text{(multiplication is associative)} \\
a \cdot 1 &= 1 \cdot a && \text{(is a unit for multiplication)} \\
a \cdot (b + c) &= a \cdot b + a \cdot c && \\
(b + c) \cdot a &= b \cdot a + c \cdot a && \text{(multiplication distributes over addition)}
\end{aligned}$$

Within the domain of program analysis, we are often concerned with *closed semirings*. For this reason, we must introduce an additional unary operation called closure which is represented by *. Lehmann (1977) states that an algebra is a closed semiring if, and only if, the conditions for a semiring hold, plus the following condition:

$$\begin{aligned}
a^* &= 1 + a \cdot a^* \\
&= 1 + a^* \cdot a
\end{aligned}$$

We shall now look at a few examples of specific semirings. Perhaps the simplest example is the Boolean semiring which is a commutative semiring, meaning that its multiplication is commutative. It consists of a set of two elements $\{0, 1\}$ and is defined as follows:

$$\begin{aligned}
0 + 0 &= 0 \\
0 + 1 = 1 + 1 &= 1 + 0 = 1 \\
1 \cdot 1 &= 1 \\
1 \cdot 0 = 0 \cdot 0 &= 0 \cdot 1 = 1
\end{aligned}$$

Another useful semiring, which is used within the program analysis domain, is the tropical semiring, which is also well-known as the min-plus semiring. It consists of a set of the non-negative integers with an extra element, identity ∞. The binary operations $+$ and $\cdot$ are defined as min and addition respectively. One more example of a closed semiring is the regular languages. This satisfies all the necessary conditions when $\cdot$ is concatenation, $+$ is union, and $*$ is the Kleene star, as identified in Dolan (2013).

2.5.2 Closure over Matrices

Lehmann (1977) carried out investigation into how it is possible to define closure on matrices over a closed semiring. This work links further and becomes more useful in data-flow analysis. We will look at the application in the next section, but first will look at defining closure over matrices.

We need to show than an $n \times n$ matrix forms a closed semiring, so that we can use the closure operation to calculate useful information about a graph such as reachability Dolan (2013). We shall define a closed semiring for matrices as follows:

Addition and multiplication of an $n \times n$ matrices are defined in the usual way Dolan (2013):

$$\begin{aligned}(A+B)_{ij} &= A_{ij} + B_{ij} \\ (A \cdot B)_{ij} &= \sum_{k=1}^{n} A_{ik} \cdot B_{kj}\end{aligned}$$

This means that we have the binary operations + and $\cdot$ necessary for the definition of a semiring. To define a closed semiring we need to take the further step of defining closure for matrices. Lehmann (1977) defines the closure operation inductively based on the size of the matrix, by splitting the matrix into four sub-matrices. This idea is identified and explained in other literature such as Dolan (2013) and Abdali & Saunders (1985), following the work produced by Lehmann who was at the foundation of this discovery. The definition of closure of an $n \times n$ matrix is defined as follows:

If $n = 1$ then $[a]^* = [a^*]$.

If $n > 1$ and $M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ then Lehmann defines, as identified in Dolan (2013), that its closure will satisfy:

$M^* = \begin{bmatrix} A^* + B'\Delta^* C' & B'\Delta^* \\ \Delta^* C' & \Delta^* \end{bmatrix}$ where $B' = A^*B$, $C' = CA^*$ and $\Delta = D + CA^*B$

It does not matter about the size of the matrix, this rule will apply recursively when splitting $n \times n$ matrices. Perhaps the most useful result, as Dolan (2013) notices, is how 'for any closed semiring R, the $n \times n$ matrices of elements of R form a closed semiring'. This is an extremely useful consequence since, using closure and a suitable choice of the semiring, we are able to use the same approach for many different analysis problems.

2.5.3 Closure of a Matrix for Program Analysis

Representing a Graph as a Matrix

In Dolan (2013) it is recognised that a directed graph can be represented as a matrix. If the graph has n nodes then its adjacency matrix M is constructed as an $n \times n$ matrix where M_{ij} is true if there is an edge from i to j and false otherwise. This can be further extended to represent weighted graphs, by using numerical values representing the costs of the transition between each node rather than Boolean values. For each position M_{ij}, the value in that position represents the length of the edge between i and j. An issue to consider here is what value to use if there is no edge between nodes; Dolan (2013) suggests using a value representing 'unreachable', perhaps by using ∞ as seen in the tropical semiring.

Reachability

Reachability is formulated by composing a Boolean matrix that represents the control flow graph of a program. This will result in an $n \times n$ matrix of Booleans, where 1 represents an edge between nodes and 0 represents the absence of an edge between nodes. This forms a closed semiring and by forming a closed semiring, we are able to use closure in order to calculate reachability. Lehmann (1977) states that 'the closure of a Boolean matrix is its transitive and reflexive closure.' He outlines how closure can be obtained by induction, or by using two different methods: using the Warshall-Floyd-Kleene algorithm (the Gaus-Jordan method) or by using Gaussian elimination (the Gauss method). Dolan (2013) recognises that computing closure by induction has the same complexity as calculating transitive closure using the Floyd-Warshall algorithm.

Path Problems

Like the Reachability Analysis formulated as the closure of a matrix, in order to analyse paths it is as 'simple' as defining the correct semiring and applying it in the same way. For example if, instead of the Boolean semiring, we use the tropical semiring then we can analyse the shortest path in a graph. That is, if we consider a path ab within a graph, then we can use the tropical semiring to identify the shortest path that goes through a and then b. This is similar to Reachability Analysis in that a matrix of 0 and 1 values are used to represent the graph, however, we now introduce a new element ∞ to represent 'unreachable'. By calculating closure, in the same way as for Reachability Analysis, we can determine this length.

Further to the discovery of the length of the shortest path, we may be interested in identifying the specific route that is the shortest. Dolan explains how we can define another semiring that can be used to keep track of information, in order to preserve the data as a list of edges representing the paths. We must remember that there could be multiple paths of the same length within a graph, resulting in the potential for more than one shortest path. The decision must then be made whether to return multiple results or to come up with some solution as to which single path to return. Dolan suggests an approach assuming that nodes are ordered, therefore allowing for a single choice when presented with multiple 'shortest' paths.

Furthermore, in Dolan (2013) the idea of identifying the longest path in a graph is approached. This introduces one of the limitations described previously, that graphs with cycles may have infinitely or arbitrarily long paths. Now some new values have to be introduced into the matrix in order to allow for the consideration of no paths between nodes as well as infinitely long paths due to cycles. Dolan outlines an example of the resulting semiring in Haskell, proving another useful application of semirings within the domain.

2.5.4 Using Data-Flow Equations for Program Analysis

Using matrices to solve linear equations is one of the most fundamental mathematical applications and having the flexibility to define a semiring means we are provided with the powerful tool of being able to define 'linear' Dolan (2013). Expanding on this, it is useful to realise that

linear equations, finite state machines and regular languages can be considered equivalent; when we acknowledge this new idea of what linear is. Dolan notices how, for every transition in a finite state machine, we have a grammar production and that we can group these productions to give a system of equations. We can then produce a matrix representing the transitions and perform closure to solve said equations, resulting in the language accepted by a given state machine.

We can re-arrange data-flow equations for an analysis in order to provide linear equations in the correct format. By using matrices to represent these equations we get a set of linear equations in the format

$$L = M \cdot L + A$$

Dolan (2013) claims that the above equation has a solution given by

$$L = M^* \cdot A$$

This is true because if

$$M^* = 1 + M \cdot M^*$$

then

$$\begin{aligned} L &= (1 + M \cdot M^*)A \\ &= A + MM^*A \\ &= MM^*A + A \\ &= ML + A \end{aligned}$$

which means that it is possible to calculate the solution to the linear equations using the previously described closure of a matrix. This is similar to the previous examples whereby programs are analysed by solving the closure of a matrix, however, in this case there is an extra application in that the solution is found by $M^* \cdot A$.

Analysis of Live Variables

Dolan (2013) gives an example of how a semiring can be applied to Live Variables Analysis. When we consider live variables we are computing 'which assignments in an imperative program assign values which will never be read and which ones may be used again'. This can be used to identify redundant code. To perform the Live Variables Analysis we consider the program's control flow graph, as described in the previous section. Dolan sets out a set of expressions, which are those that are described in the data-flow analysis section, if $IN_s = L_{entry}(s)$, $OUT_s = L_{exit}(s)$, $GEN_s = L_{gen}(s)$ and $KILL_s = L_{kill}(s)$:

$$\begin{aligned} IN_s &= GEN_s \cup (OUT_s - KILL_s) \\ OUT_s &= \bigcup_{s' \in succ[S]} IN_{s'} \end{aligned}$$

which can also be written as:

$$\begin{aligned} IN_s &= (OUT_s \cap \overline{KILL_s}) \cup GEN_s \\ OUT_s &= \bigcup_{s' \in succ[S]} IN_{s'} \end{aligned}$$

Dolan illustrates how a semiring can be defined consisting of a set of variables in the program, where 0 is the empty set and 1 is the set of all variables, $x^* = 1$ for all sets x, $+$ is union and $\cdot$ is intersection. The system of equations can be rearranged and represented as follows:

$$\begin{aligned} OUT_s &= \sum_{s' \in succ[S]} IN_{s'} \\ &= \sum_{s' \in succ[S]} \overline{KILL_{s'}} \cdot OUT_{s'} + GEN_{s'} \\ &= (\sum_{s' \in succ[S]} \overline{KILL_{s'}} \cdot OUT_{s'}) + (\sum_{s' \in succ[S]} GEN_{s'}) \end{aligned}$$

This is a system of affine equations of the form $L = M \cdot L + A$ where A is a vector containing the constant terms and matrix M contains the coefficients. This can be solved using closure where $L = M^* \cdot A$, as described previously.

Analysis of Reaching Definitions

Similarly for Reaching Definitions Analysis the data-flow equations as outlined previously, if $IN_s = RD_{entry}(s)$, $OUT_s = RD_{exit}(s)$, $GEN_s = RD_{gen}(s)$ and $KILL_s = RD_{kill}(s)$:

$$\begin{aligned} IN_s &= \bigcup_{s' \in pred[S]} OUT_{s'} \\ OUT_s &= GEN_s \cup (IN_s - KILL_s) \end{aligned}$$

can also be written as:

$$\begin{aligned} IN_s &= \bigcup_{s' \in pred[S]} OUT_{s'} \\ OUT_s &= (IN_s \cap \overline{KILL_s}) \cup GEN_s \end{aligned}$$

Similar to liveness analysis, a semiring can be defined, represented by:

$$\begin{aligned} IN_s &= \sum_{s' \in pred[S]} OUT_{s'} \\ &= \sum_{s' \in pred[S]} \overline{KILL_{s'}} \cdot IN_{s'} + GEN_{s'} \\ &= (\sum_{s' \in pred[S]} \overline{KILL_{s'}} \cdot IN_{s'}) + (\sum_{s' \in pred[S]} GEN_{s'}) \end{aligned}$$

which, again, can be solved as a system using $L = M^* \cdot A$.

Other Analyses

We have discussed how matrices can be used for solving systems of linear equations and that by defining our semiring we have the flexibility to decide on what 'linear' means for our application

Dolan (2013). Since many problems can be represented as a system of linear equations we are provided with a powerful tool for performing many different analyses using semirings over matrices. By applying the same approach as explained already, for example Live Variables Analysis, we can solve the linear equations for any of the analyses described in the data-flow analysis section. The main idea to take away is that by using the appropriate semiring we can analyse different properties and, as long the equations are of the correct form, it is as simple as just swapping the semiring that is used.

2.6 Assigning Real-Life Costs

2.6.1 Background to Existing Work

It is clear that static analysis techniques have been around for a long time but we must also consider the efforts of cost analysis as part of this. In Liqat et al. (2013) the authors identify existing work within the domain. It is identified that early attempts at cost analysis date back to 1975 when Wegbreit (1975) uses a metric to analyse Lisp programs to produce data such as execution time based on input. Since then it is identified that cost analysis has been developed further, with the first approach applied to energy consumption being in 2008 Navas et al. (2008) in which 'upper-bounds on the energy consumption of Java programs are statically inferred as functions of input data sizes'. The approach more recently taken in Liqat et al. (2013), outlined in Section 6.2.1, claims to analyse at a lower level.

2.6.2 Energy Usage

The research into calculating energy usage from static code is minimal. Some attempts have been made, such as Liqat et al. (2013). It appears that limited effort has been made due to the difficulty of the problem, however, it is interesting to look at the approaches taken in these attempts and to evaluate how successful their attempts have been.

The reason why people are concerned with energy consumption that there is an environmental and financial cost of computing Liqat et al. (2013). We are reaching limits now where hardware is not developing as fast as it used to resulting in a slowdown in advancements of power-efficient hardware. People are concerned with keeping costs down and making the most of the battery lives of portable machines Schubert et al. (2012). A way in which we could focus our efforts is now within software, by considering how software can efficiently make use of such hardware to save energy. To do this, we need to evaluate optimisation techniques that can be used to determine software energy consumption and consider optimal ways of executing them Liqat et al. (2013). This may also help software engineers understand program design better, leading to better design decisions and therefore more energy efficient code.

Static Analysis of Energy Consumption within Embedded Programs

In Liqat et al. (2013) the authors make an attempt at analysing energy consumption within embedded programs, using static analysis and low level energy modelling. The authors develop

a tool which makes use of existing tools and that is able to estimate the energy consumption of an embedded program considering the hardware platform on which it is executed. This is an important factor to take account of when thinking about statically analysing energy - it is dependent on the hardware on which the code is run. In Liqat et al. (2013), the analysis relies on this information, giving the result of executing software on the specified hardware, limiting its uses to those provided. This system is also limited by a specific programming language. They use XC which is a C-based imperative language designed for real-time embedded parallel architectures. The system developed in Liqat et al. (2013) is outlined in the following way:

- The source code is input (an XC program)
- An XC compiler tool (XCC) produces the corresponding ISA (Instruction Set Architecture) program
- A translator generates the associated Horn clauses
- Resource analysis is performed using ISA-level energy models
- Energy consumption is output

The energy models are provided during the static analysis phase by making assertions in order to 'infer information for higher-level entities such as functions'. Creating these energy models at a low-level allows for more precise information corresponding to the effect of the execution of the program on the hardware Liqat et al. (2013). The authors claim to have 'bridged the gap between researchers closer to the hardware area, needed to produce the low level energy models, and others from software, with expertise in static analysis techniques and tools'.

These steps are explained in more depth within the paper and the results of an experimental assessment of the approach are given. The results show a 'reasonably accurate' estimation by the system, however this approach is limited to the XS1 architecture.

Dynamic Energy Profiling for Code Optimisation

In Schubert et al. (2012), the aim is to identify 'energy-hungry' sections of code providing programmers with useful information when attempting to optimise their code and make energy-aware software development decisions. This is not a static analysis tool, however, it includes relevant considerations when approaching energy cost analysis of programs. The authors develop a profiler that links energy consumption and code location taking into consideration:

- **synchronous energy**: consumed in the CPU
- **asynchronous energy**: consumed by peripheral devices

It is identified that 'few tools and methods exist' which is the motivation behind the production of this tool which they name *'eprof'*. This tool allows a user to analyse how different approaches to some solution may affect energy consumption based on how the hardware will be used by given code. The authors explain how classic performance profiling takes a CPU-centric

approach but their approach aims to take into consideration peripheral devices too, stating that these are 'significant energy consumers'.

One factor to consider is *idle* energy. This is energy that is used by the machine being on; independent of any other processes running. On top of this is *dynamic* energy which is the power necessary for running activities on the system - the type of energy that can have a great effect on overall software energy. The authors of Schubert et al. (2012) observe that energy profiling requires two types of information:

- the amount of energy spent
- the code location causing the energy consumption

which results in their energy profiler, *eprof*, implementing two components: 'the observation of energy-relevant activity and the estimation of the amount of energy consumed by this activity'. The tool runs, observing activity, recording stack traces to capture code location and estimating energy consumed.

2.6.3 Evaluation of the Two Approaches

Approach 1, the static analysis of energy consumption within embedded programs, looks at low-level static cost analysis based on a fixed architecture whereas approach 2, dynamic energy profiling for code optimisation, runs code to identify code locations of high-energy activities. Approach 1 shows us what has been achieved so far in the static analysis domain and highlights the main limitations. So far, it seems, static analysis is limited largely by specifying the hardware on which code is run. While approach 2 is not a static analysis, it provides us with important considerations and illustrates how dynamic analysis may contribute to more useful information when examining code for optimisation, such as identifying the 'energy hungry' code locations.

By using these two methods, we can consider ideas for new approaches that could be more useful. Energy analysis could be extended beyond these existing methods by looking at being able to statically assign energy costs to certain code locations, resulting in the ability to identify 'energy hungry' code locations without having to use a dynamic tool. Alternatively, since static analysis is limited by the specification of a fixed hardware, we could imagine a tool which can provide information about energy usage independent of the hardware on which a program is run.

2.7 Existing Software Tools

There are many existing static program analysis tools. Most compilers will contain some sort of analysis, certainly for code correctness, and generally for code optimisation too. The main aim is to minimise the amount of time a program takes to execute but the compiler may also look at optimising the use of resources, in particular shared ones. In addition, IDEs (Integrated Development Environments) and IDE PlugIns may contain some sort of static program analysis.

Some examples of functionality that IDE tools may provide include highlighting unreachable code as well as highlighting logically incorrect code.

Outside of the use in IDEs and compilers, there are a large selection of tools entirely dedicated to the static analysis of programs. Some of these are designed for a specific programming language and some are more general. Since there are so many, we shall look at just one as an example called 'Maplas' *Atkins Maplas* (n.d.) which claims to be 'one of the world's most rigorous and advanced software analysis and verification toolsets'.

The Maplas software works by taking a user program, translating the source code, analysing it using a MAPLAS Analyser and then providing the user with a report of the analysis. Source code is translated into an intermediate language which programs written in any sequential language can be translated into, meaning that the software is not limited to a specific language. The analysers then only need to be concerned with the intermediate language, providing a nice abstraction from the actual source code, but allowing for results specific to the source code. The user is provided with the choice of different types of analysis including: control flow analysis, data use analysis, information flow analysis, semantic analysis and compliance analysis. Using the control flow analysis, which is most relevant to this project, a user can identify concerns such as unreachable code, infinite loops and multiple entry and exit points.

As sophisticated and comprehensive as current existing tools are, they still lack the addition of real-life costs. They tend to either be general tools which means the amount of optimisation they can provide is limited, or tools, such as compilers, that are effective because of how specific they are. In particular, existing tools lack accurate and useful information and optimisations regarding real-life issues such as energy usage.

2.8 Summary and Conclusions

With a broad and mature selection of literature relevant to this project, there is a large amount of background to understand and investigate. We have been able to identify how languages are parsed and have mentioned some of the currently available tools that can aid with parser generation, which will be relevant to the understanding and implementation of this project. Foundational definitions of graphs and graph theory have been introduced, providing a sound understanding of the underlying structure for analyses to be carried out. Furthermore, the important idea of representing a program as a graph, namely a control-flow graph, has been described. This allows us to perform the necessary transition from parsed program code to an appropriate structure for analysis.

Primarily through the work provided in Nielson's book Nielson et al. (1999) we have been able to identify and illustrate the main concepts with relation to data-flow analysis. In particular, we have understood data-flow equations and how they are formed. Through the foundational work of Lehmann (1977), and the more recent work of Dolan (2013) bringing several ideas together, we have been able to define semirings, closure and apply them specifically to data-flow analysis. We have seen how data-flow equations can be used to form a closed semiring for solutions to data analysis problems. These concepts are consolidated with a large selection of

papers investigating and approaching these ideas with slightly different angles and focuses.

Studying some more recent literature, we have been able to investigate existing approaches to performing cost analysis, in particular with an application for energy consumption. These approaches Liqat et al. (2013), Schubert et al. (2012) give us an idea of considerations when tackling this task as well as current successes and limitations of the approaches.

Overall, the literature review provides a solid understanding and identification of ideas that are fundamental to this project. This project will be able to build on the existing knowledge and mathematics behind data-flow analysis using semirings and will aim to implement a system which can assign and analyse real-life energy costs as part of this, by taking into consideration the existing work within the domain.

Chapter 3

Requirements

This chapter addresses the requirements analysis and then provides a formal requirement specification describing the proposed system, an automated static analysis tool to be used for multiple program analyses. The requirement specification covers both functional and non-functional requirements and provides enough information to progress to system design and development. Being thorough, the requirements specification also provides a strong framework for testing the functionality of the developed system later in the software lifecycle.

The aim now is to analyse the needs of the project itself, to scope what the system should and should not do and to produce a comprehensive and measurable set of requirements for the intended system. After development, we shall refer back to this specification in order to assess the success of the project.

3.1 Requirement Sources

Since the nature of this project is not heavily focused on end-users, but primarily mathematical concepts and analysis, the requirements gathering stage is unconventional and somewhat different to traditional methods. Common approaches to requirement elicitation involve discussions and interviews with stakeholders and perhaps even initial user studies. This project is not focused on usability but aims to provide an adaptable tool able to perform complex mathematical analyses and provide a correct output to the user. For this reason, the main concern is providing a system that is mathematically sophisticated and correct, and therefore we rely on slightly different sources for the requirement elicitation stage.

3.1.1 Identifying Stakeholders

Traditionally, a requirements specification is used as part of the agreement between the developers and the stakeholders in order to decide upon and scope what the system is expected to do. These are any individuals or groups who will be directly or indirectly concerned with or affected by the system. We begin by identifying stakeholders for this project.

We can consider stakeholders anyone who is concerned with or could benefit from performing analyses on program code, particularly focusing on optimisation. This could include a wide range of users, for example writers of compilers or individuals concerned with optimising their code, and does not focus on one specific group. If we consider the idea of assigning real-life costs within the analysis, for example quantifying energy costs within code, then this may expand the group even further. Hypothetically, there are a wide range of end-users for this system, but we focus mainly on the technical ability and correctness of the program.

Other stakeholders may include mathematicians and computer scientists contributing to and investigating similar work within the static analysis domain. Since there is a mature theoretical and mathematical background within the domain as well as progress and new approaches being taken, current researchers may be interested in used methodologies, especially when we get to investigating and experimenting with new ideas around analysing real-life energy costs within code.

3.1.2 Introspection

Due to the fact that there is not a specific group of end users and along with the fact that this project is largely experimental, we can consider introspection as an appropriate requirements elicitation technique. In Zowghi & Coulin (2005), it is described that introspection involves the requirements analyst, the author in this case, to elicit requirements based on what he or she believes stakeholders want and need from the system. This technique is often used as a starting point for requirement elicitation and is often consolidated with other traditional techniques.

For this project, the author considers functionality that the system is anticipated to provide within the domain, in order to be adaptable and flexible as well as mathematically correct. The introspection is aided by and backed up by research within the domain.

3.1.3 Domain Expertise

The main source for requirements elicitation is the background within the domain, by examining existing literature and systems. We must heavily rely on the theoretical foundation and existing knowledge so to produce a system that works correctly, relying on existing developments. This was done in the literature survey, which is an extremely valuable source during the requirements elicitation stage of this project. It is necessary to extract and understand the underlying mathematics to allow for correct proofs using the methodologies intended and to be able to consider how this can be implemented as an automated tool.

Within the literature review we highlighted the mathematical requirements for the proposed analyses to be carried out. This includes the data-flow equations that are necessary to be solved as well as the background of semirings which can be used to solve them. Since this is theoretical, the system will rely on this to perform correctly. We must also consider the idea of creating a system that effectively implements this theory so that it is suitably adaptable and automated. These types of analyses are often carried out by compilers, so we could consider the system to be integrated for use within parts of other systems.

3.2 Requirements Analysis

Before formulating the final set of requirements, it is important to analyse the requirements. It is not sensible to have a list of requirements for the system with no classification or prioritisation of them so here we consider how to organise and appropriately define the requirements in a structured and prioritised manner.

3.2.1 Classification

Requirements may be classified into a range of types. For this project we consider two large classes of requirements: functional and non-functional. The functional requirements describe what the system should accomplish technically while non-functional requirements describe how the system should perform. We shall define high-level requirements first by considering the order of flow within the system from input to output, using a similar structure to the literature review, and then provide more detailed requirements and sub-requirements of the higher level ones.

3.2.2 Prioritisation

Requirements should be prioritised in order to determine what is most important to the success of the project and what is less important. There are certain requirements that must be met for the system to operate as well as requirements that are less crucial. By prioritising requirements we can also reduce risk during development stages, by focusing first on the fundamental requirements of the system and then by considering the nice-to-have requirements. We shall first define the different priority groups, using the MoScoW Technique *MoSCoW : Requirements Prioritization Technique* (n.d.):

- Must : mandatory
- Should : of high priority
- Could : preferred but not necessary
- Would : can be postponed and suggested for future execution

Using these priorities, we then consider what priority each requirement should be assigned. The approach used to do this was by considering certain criteria such as stakeholder value, risk, implementation cost and effort, likelihood of success and the relationship with other requirements. By weighing up these different factors, it is possible to determine the pros and cons of fulfilling it and therefore allowing for reasonable analysis for prioritisation.

3.3 Requirements Specification

3.3.1 Functional Requirements

1. Must provide a way to input source code to the program.

(a) Must accept source code for the imperative 'while' language (included nested structures) described by the abstract syntax:

Expressions:

$$\begin{array}{rcll} a & \in & \textbf{AExp} & \text{arithmetic expressions} \\ b & \in & \textbf{BExp} & \text{boolean expressions} \\ S & \in & \textbf{Stmt} & \text{statements} \end{array}$$

Values:

$$\begin{array}{rcll} x, y & \in & \textbf{Var} & \text{variables} \\ n & \in & \textbf{Num} & \text{numerals} \end{array}$$

Operators:

$$\begin{array}{rcll} op_a & \in & Op_a & \{*, /, +, -\} \\ op_b & \in & Op_b & \{\text{and, or}\} \\ op_r & \in & Op_r & \{<, >, \geq, \leq, ==\} \end{array}$$

Syntax:

$$\begin{array}{rcl} a & ::= & x \mid n \mid a_1\ op_a\ a_2 \\ b & ::= & \texttt{true} \mid \texttt{false} \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2 \\ S & ::= & x\ :=\ a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S\ \texttt{end} \end{array}$$

(b) Should accept source code via a .txt file rather than command line input.

2. Must provide a way for the user to specify the analysis to be performed.

3. Must be able to produce correct analysis solutions for a valid input program.

   (a) Must be able to compute Reachability Analysis, by computing closure of an adjacency matrix.

   (b) Should be able to compute Live Variables Analysis, by generating and solving the following equations:

$$\begin{aligned} OUT_s &= \sum_{s' \in succ[S]} IN_{s'} \\ &= \sum_{s' \in succ[S]} \overline{KILL_{s'}} \cdot OUT_{s'} + GEN_{s'} \\ &= \left(\sum_{s' \in succ[S]} \overline{KILL_{s'}} \cdot OUT_{s'}\right) + \left(\sum_{s' \in succ[S]} GEN_{s'}\right) \end{aligned}$$

(c) Should be able to compute Reaching Definitions Analysis, by generating and solving the following equations:

$$\begin{aligned} IN_s &= \sum_{s' \in pred[S]} OUT_{s'} \\ &= \sum_{s' \in pred[S]} \overline{KILL_{s'}} \cdot IN_{s'} + GEN_{s'} \\ &= (\sum_{s' \in pred[S]} \overline{KILL_{s'}} \cdot IN_{s'}) + (\sum_{s' \in pred[S]} GEN_{s'}) \end{aligned}$$

(d) Could be able to compute some analysis considering real-life costs.

4. Should be adaptable for different analyses to be performed.
   (a) Each analysis should use the same core theory in order to execute.
5. Must output the result in the form of a matrix clearly representing the analysis solution.
   (a) Must output the result of the analysis to the command line.
   (b) Could output the result of the analysis to a .txt file

3.3.2 Non-Functional Requirements

1. Must run on a standard Windows machine.
2. Must run via command line.
3. Must output correct analyses (this means the same results that would be computed if the analysis was solved manually).
4. Must handle small programs, that is programs with up to 20 nodes.
5. Should be able to handle programs of reasonable size, that is programs with up to 100 nodes.
6. Should compute the solution to small programs within a reasonable amount of time. By reasonable, we consider within 1 second.
7. Should be intuitive to use - a new user with no experience of the system but with a provided user guide should be able to perform analyses.
8. Should be implemented in a programming language suitable for the project described.
9. Should be adaptable for different analyses.
10. Could be adaptable for possible integrations with other systems.

Chapter 4

Design

Following the requirements of the system, we can consider how the actual system can be designed so to fulfil these requirements as effectively as possible. We will consider a high-level overview of the system design and then detail some of the design decisions within each of the different modules of the system. We often consider separate modules for a system so to separate sections which could in theory be replaced by a new implementation of this module and which can function independently of the other functionality in the system.

4.1 Programming Language Choice

The choice of programming language should reflect the problem at hand. While there is a large selection of programming languages that could be chosen to implement a solution, it is important to consider certain features of programming languages that are suitable for a specific problem.

The programming language that has been chosen for this project is Haskell *Haskell* (n.d.). While the author does not have much experience in functional programming, in particular Haskell, it provides beneficial features that suit the style of this project. By choosing a programming style that is relatively new to the author, it provides the opportunity to learn and understand more than simply just a solution to the problem. It also provides a new way of thinking about programming as well as experience with the functional style.

A large benefit of choosing Haskell when considering the parser module of the system is Parsec *Parsec* (n.d.). Parsec is a parser combinator library which can be used to parse a defined grammar and produce a representation of the parse tree for the provided code. This makes generating the parser a lot simpler which is especially useful since parsing the language is not the main focus of this project, just an important stage within the implementation of it.

Haskell is an appropriate choice for the system for several other reasons. An important reason for choosing Haskell is that it is a high-level language and provides good support for manipulation of nested data structures. When we consider the nature of the problem at hand, and the data

structures necessary to represent program data, this is a valuable feature. Furthermore, it does not take a lot of code to implement solutions that might take a lot more code in other languages.

For many reasons, including the main reasons outlined above, Haskell seems like an appropriate choice for this project. The most difficult consideration is the learning curve necessary for the author to benefit from the main features of the functional style; however, the suitability of the language outweighs the possible overhead, at least during the design stage of the project.

4.2 High-Level Overview

By using the requirements of the system it is possible to imagine a flow through the system and begin to identify the different modules that are necessary within the system. The basic flow through the program is as follows:

- Input source code and analysis choice into the program
- Parse the source code
- Organise the parsed code into the appropriate representation
- Perform analysis on the organised representation
- Output the result of the analysis

The flow through the program is very linear, with one module passing its computed output as input to the next until the output is displayed to the user. The only interaction that a user has with the system is inputting the program code and choice of analysis and then viewing the displayed result.

4.3 System Architecture

We shall now consider a more detailed description of the system as a whole, considering the different analyses that will be computed. Figure 4.1 illustrates an overview of the system, including: input, system components, program data and output, and the flow of control and data between them.

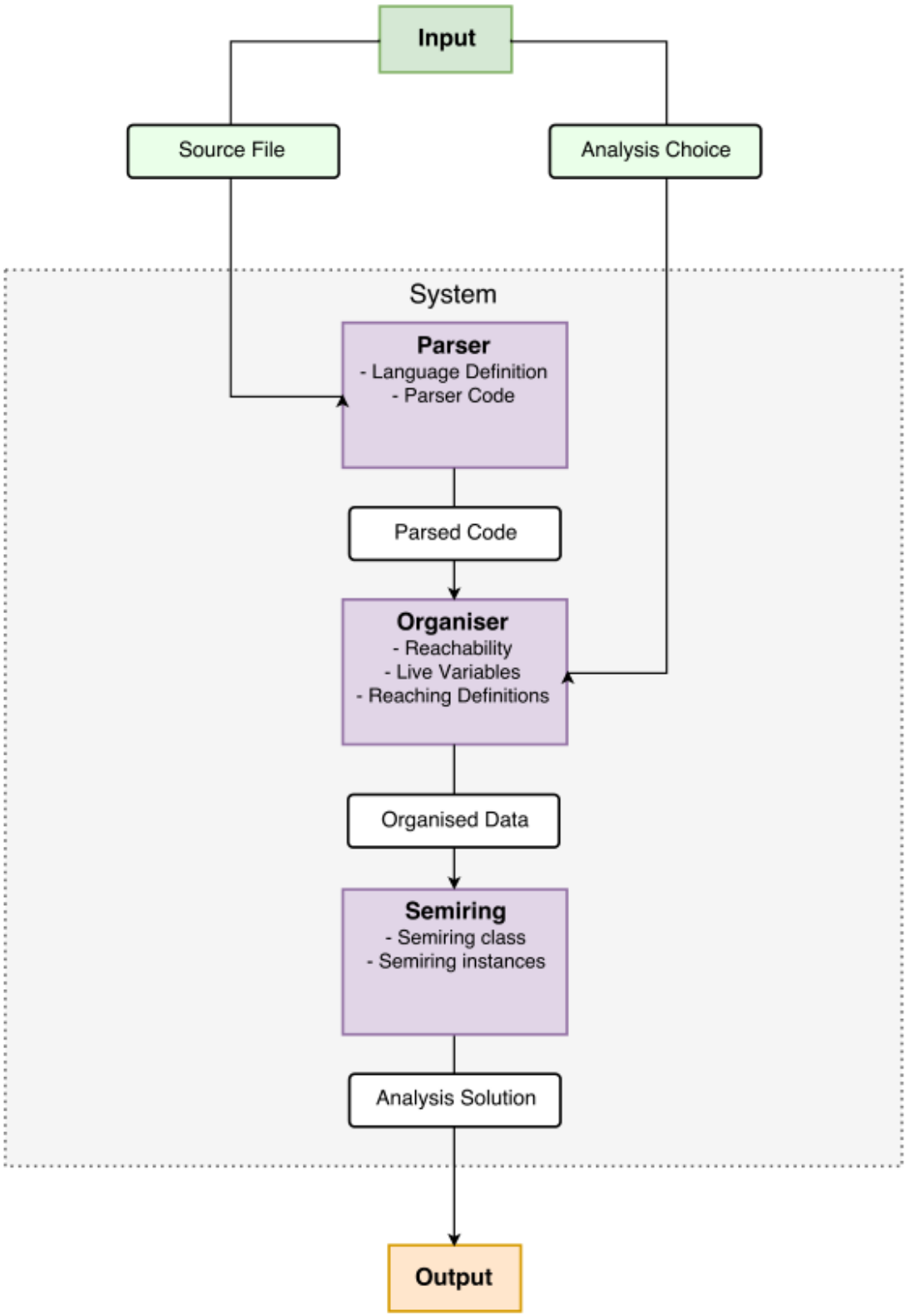

Figure 4.1: Overview of System Architecture

4.3.1 Input

There is only one input interaction that the user needs to provide the system. This is one simple command, via keyboard input, containing two pieces of information:

- the analysis to be performed
- the file, containing source code, that the analysis should be performed upon

The information will then be used appropriately with the name of the file for analysis being passed to the parser and the analysis to be performed being used to apply the appropriate organisation of the data.

4.3.2 System Components

There are three main system components within the system. These are the main modules of the system that perform a high-level task as identified in the high-level overview above. We shall now describe them in more detail.

Throughout this section, explicit examples will be provided so to clarify understanding of the design. For each example, we shall consider a factorial program with the code and control flow diagram as shown in Figure 4.2 and Figure 4.3 respectively.

$y \leftarrow x$
$z \leftarrow 1$
while $y > 1$ **do**
    $z \leftarrow z * y$
    $y \leftarrow y - 1$
end while
$y \leftarrow 0$

Figure 4.2: Pseudocode for a factorial program

Parser

The parser needs to be designed so to correctly parse the defined imperative 'while' language. It should accept a .txt file containing the source code to be analysed, and provide some appropriate representation of the parse tree for the code provided, such that the rest of the system can understand and appropriately organise the input. The parser must define the syntax of the language as defined in Requirement 1.

As the chosen language for the project is Haskell, the parser will be generated by using the Parsec parser combinator library *The Parsec Package* (n.d.). Parsec can be used to parse expressions and statements in a given language such as the simple imperative 'while' language defined in Requirement 1. This is done by defining data types, each type of expression and

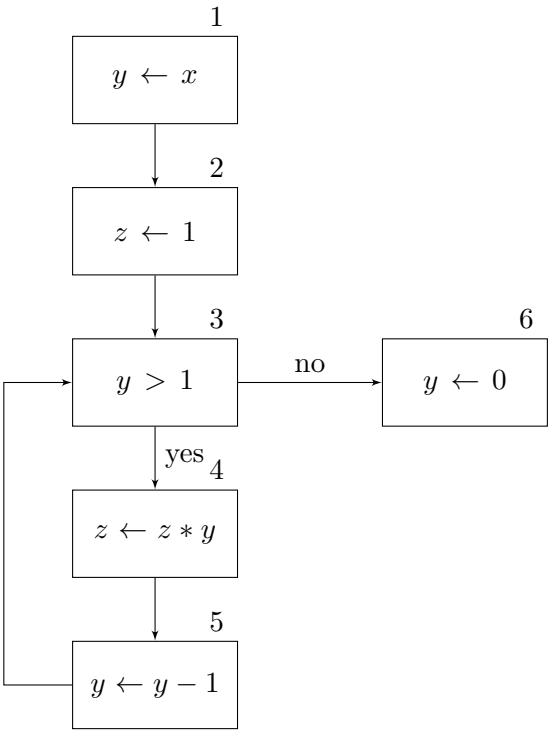

Figure 4.3: Control-flow graph for a factorial program

operators. We then create the language definition including reserved names and reserved operators that are part of the grammar and this is used to create the lexer which does the lexing of the code. We then write the code to actually parse sequences of statements, the program. The Haskell Wiki provides an explained example of creating a parser using the Parsec library *Parsing a simple imperative language* (n.d.). By understanding how the Parsec library works and by defining the grammar of the language being parsed, it is simple to create the necessary parser.

Using our factorial program, we expect the parser to produce some internal representation of the parse tree illustrated in Figure 4.4. The description of how this will be internally represented is described in the section 4.3.3 'Program Data'.

Organiser

The organiser is, perhaps, the most complex module to consider. Since there are several analyses that the system will be able to perform, it is important to consider how the program information should be organised. There are three main analyses to consider: Reachability, Live Variables and Reaching Definitions Analysis, with the aim to extend a new analysis considering real-life costs. For this reason, it is important to design a system that takes advantage of similarities between the analyses. The main thing that each of the analyses relies on is the control flow diagram representing the program and the adjacency matrix derived from this. We shall now consider each analysis, the information that needs to be organised for it and an outline of an algorithm for doing so.

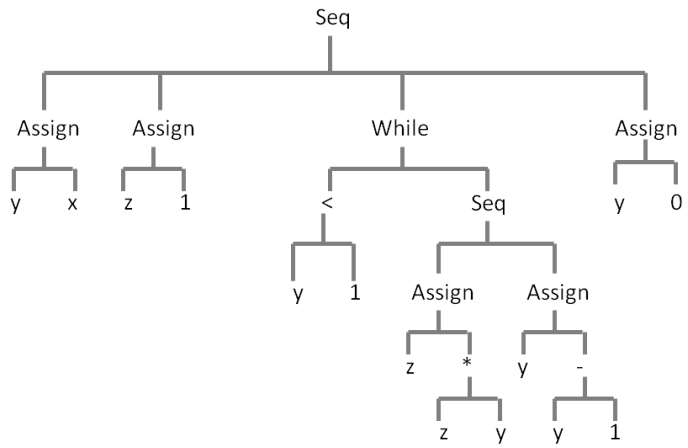

Figure 4.4: Parse Tree for the Factorial Program

The Adjacency Matrix
All analyses rely on the control-flow graph as they are all based on data-flow in the program. The adjacency matrix is another way of representing the flow of control in the program and for this reason all analyses rely on the adjacency matrix. The program should be able to take the parsed representation of the code and create the adjacency matrix. It is necessary to consider the number of nodes in the control flow graph as this will be the dimension of the $n \times n$ matrix, as well as identifying any successors of each node.

If we consider the factorial program that we are using as an example, the resulting adjacency matrix is as follows:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

In this adjacency matrix a 0 in position M_{ij} represents no flow of control between node i and node j, whereas a 1 in position M_{ij} represents a flow of control between node i and node j. For example, the 1 in row 1 column 2 means that there is flow of control from node 1 to node 2, which describes the arrow from node 1 to node 2 in the control-flow graph.

In order to produce the adjacency matrix, for each node in the control flow graph we need to be able to determine any successor nodes. This gets complicated when considering 'while' and 'if' statements as they involve jumps, rather than a linear flow to the next node. For example, in a 'while' loop, if the Boolean test passes the flow continues to the next node but if it fails it jumps to the first statement following the 'while' loop. Similarly, at the end of a

'while' loop it is necessary to jump back to the Boolean test. In order to do this, the system needs to be able to calculate the 'size' of jumps using a representation of the parsed program - see section Chapter 5 for the actual implementation of this.

Reachability
The solution to Reachability Analysis is found by performing closure on the adjacency matrix of the control flow graph. Once we have computed the adjacency matrix for the program, this is a simple computation of the closure using the appropriate semiring definition - the Boolean semiring.

Live Variables
Live Variables Analysis is more complicated than Reachability Analysis as it involves the solving of the data-flow equations. These equations were studied in the literature review but are repeated here:

$$\begin{aligned} OUT_s &= \sum_{s' \in succ[S]} IN_{s'} \\ &= \sum_{s' \in succ[S]} \overline{KILL_{s'}} \cdot OUT_{s'} + GEN_{s'} \\ &= (\sum_{s' \in succ[S]} \overline{KILL_{s'}} \cdot OUT_{s'}) + (\sum_{s' \in succ[S]} GEN_{s'}) \end{aligned}$$

Since these equations are of the form $L = M \cdot L + A$, we can find the solution to the analysis by $L = M^* \cdot A$ where * represents the closure of the matrix. For this reason, we do not need just the matrix M, but also the vector A.

First we need to consider the $KILL$ and GEN sets for each node in the control flow graph. The $KILL$ set for a node is the set of variables assigned and the GEN set is the set of variables that are used in the node before any assignment. Once these sets have been calculated, we can form the matrix M. The matrix M is formed in a similar manner to the adjacency matrix in Reachability Analysis. However, wherever there is a 0 (i.e. no successor) in the adjacency matrix, this is represented by the empty set (i.e. no variables in the program) and wherever there is a 1 (i.e. a successor) in the adjacency matrix, this is represented by the $\overline{KILL}$ set of the successor node.

Similarly, we create the vector A. Vector A is formed by considering successors to each node and by using the GEN set, or the union of sets if multiple successors, for each successor node.

Considering the factorial program, we result in $L = ML + A$ as follows:

$$\begin{pmatrix} OUT_1 \\ OUT_2 \\ OUT_3 \\ OUT_4 \\ OUT_5 \\ OUT_6 \end{pmatrix} = \begin{pmatrix} \emptyset & \overline{KILL_2} & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \overline{KILL_3} & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \overline{KILL_4} & \emptyset & \overline{KILL_6} \\ \emptyset & \emptyset & \emptyset & \emptyset & \overline{KILL_5} & \emptyset \\ \emptyset & \emptyset & \overline{KILL_3} & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{pmatrix} \cdot \begin{pmatrix} OUT_1 \\ OUT_2 \\ OUT_3 \\ OUT_4 \\ OUT_5 \\ OUT_6 \end{pmatrix}$$

$$+ \begin{pmatrix} GEN_2 \\ GEN_3 \\ GEN_4 \cup GEN_6 \\ GEN_5 \\ GEN_3 \\ \emptyset \end{pmatrix}$$

Which rearranges into the form $L = M^* \cdot A$ as follows:

$$\begin{pmatrix} OUT_1 \\ OUT_2 \\ OUT_3 \\ OUT_4 \\ OUT_5 \\ OUT_6 \end{pmatrix} = \begin{pmatrix} \emptyset & \overline{KILL_2} & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \overline{KILL_3} & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \overline{KILL_4} & \emptyset & \overline{KILL_6} \\ \emptyset & \emptyset & \emptyset & \emptyset & \overline{KILL_5} & \emptyset \\ \emptyset & \emptyset & \overline{KILL_3} & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{pmatrix}^* \cdot \begin{pmatrix} GEN_2 \\ GEN_3 \\ GEN_4 \cup GEN_6 \\ GEN_5 \\ GEN_3 \\ \emptyset \end{pmatrix}$$

which is what is actually computed for the solution.

In order to organise the matrix M and vector A for Live Variables Analysis, we use the adjacency matrix considered in Reachability Analysis, but adapt it as necessary in order to produce the matrix and vector containing the necessary information.

Reaching Definitions
Reaching Definitions Analysis is similar to Live Variables Analysis as it involves the solving of similar data-flow equations:

$$\begin{aligned} IN_s &= \sum_{s' \in pred[S]} OUT_{s'} \\ &= \sum_{s' \in pred[S]} \overline{KILL_{s'}} \cdot IN_{s'} + GEN_{s'} \\ &= (\sum_{s' \in pred[S]} \overline{KILL_{s'}} \cdot IN_{s'}) + (\sum_{s' \in pred[S]} GEN_{s'}) \end{aligned}$$

Again we need to consider the $KILL$ and GEN sets for each node in the control flow graph. This time it is not simply a set of variables, but a set of variable and node number pairs, representing a variable definition at a certain point in the program. The GEN set represents the sets of variables that are generated in the node and the $KILL$ set describes the set of variables re-assigned.

We create matrix M and vector A in a similar manner to Live Variables Analysis, however, we are now considering predecessors rather than successors. This can be achieved simply by considering the transpose of the adjacency matrix.

Considering the factorial program, we result in $L = M^* \cdot A$ as follows:

$$\begin{pmatrix} IN_1 \\ IN_2 \\ IN_3 \\ IN_4 \\ IN_5 \\ IN_6 \end{pmatrix} = \begin{pmatrix} \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \overline{KILL_1} & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \overline{KILL_2} & \emptyset & \emptyset & \overline{KILL_5} & \emptyset \\ \emptyset & \emptyset & \overline{KILL_3} & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \overline{KILL_4} & \emptyset & \emptyset \\ \emptyset & \emptyset & \overline{KILL_3} & \emptyset & \emptyset & \emptyset \end{pmatrix} . \begin{pmatrix} INIT \\ GEN_1 \\ GEN_2 \cup GEN_5 \\ GEN_3 \\ GEN_4 \\ GEN_3 \end{pmatrix}$$

Where $INIT$ is the set of all definitions at the start of the program, i.e. before any variables have been defined.

Organiser Summary

It is possible to see how each of these analyses relies heavily on the adjacency matrix of the control flow graph, and how Live Variables and Reaching Definitions Analysis are symmetric to one another, allowing for the design of each algorithm to be an extension of the previous. This means there is one central part of the design that is central to all analyses; the creation of the adjacency matrix.

Semirings

The semiring module requires complex mathematical understanding and appropriate design of semirings. Once the appropriate semiring is defined, the implementation is trivial. For the different analyses, we need to define different closed semirings.

Firstly, we need to design the semiring class of which each semiring definition will be an instance. To do this, we consider the necessary features of a semiring as well as their types. A closed semiring of type a consists of:

- the zero element of type a
- the one element of type a
- the binary operation plus of type a → a → a
- the binary operation multiplication of type a → a → a
- closure of type a → a

For all analyses, we need to consider the closure of a matrix, along with the matrix operations. For this reason, we must design a semiring instance for an $n \times n$ matrix. This is covered in 'Fun with Semirings' Dolan (2013) so we can re-use this in the design of this system. Matrix closure, for an $n \times n$ matrix, has been evaluated in the literature review and the implemented algorithm should follow this.

Reachability
For Reachability Analysis, the matrix is populated with Boolean values; the value 'true' if there

is a successor to the node and 'false' if there is not. For this reason, we need to consider an instance of the semiring class - the Boolean semiring, which is defined as follows:

- zero: false
- one: true
- plus: disjunction
- multiplication: conjunction
- closure: true

Live Variables
For Live Variables Analysis, the matrix M and vector A are populated with sets (using the Haskell Data.Set package) of variables. For this reason, we need to consider an instance of the semiring class - the sets of variables semiring, which is defined as follows:

- zero: the empty set
- one: set of all variables in the program
- plus: union
- multiplication: intersection
- closure: one

Reaching Definitions
Reaching Definitions Analysis is similar to Live Variables Analysis but, instead of the matrix M and vector A being populated with sets (using the Haskell Data.Set package) of variables, they are populated with sets of pairs containing variables and node labels. For this reason, we need to consider an instance of the semiring class - the sets of pairs semiring, which is defined as follows:

- zero: the empty set
- one: set of all possible pairs of variables and labels in the program
- plus: union
- multiplication: intersection
- closure: one

4.3.3 Program Data

Program data is information that the program itself has generated and which is now internally represented within the system. This is important since the system cannot simply interpret and organise source code without some understanding of it.

Parsed Code

The parsed code is the internal representation of the parse tree for the given input code. Using this information we are able to consider the control flow diagram for the program and therefore interpret the program as necessary for analyses. A program is either a statement or a sequence of statements in the language. Using Parsec, we can easily generate said structure. On the Haskell Wiki there is an example of parsing an imperative 'while' language *Parsing a simple imperative language* (n.d.), very similar to the on described in Requirement 1 (a).

If we use this example as the base for our design and consider the factorial program, we get the internal representation of the parsed code as shown in Figure 4.5.

```
Seq [Assign ‘‘y’’ (Var ‘‘x’’), Assign ‘‘z’’ (Num 1), While (RBinary
Greater (Var ‘‘y’’) (Num 1)) (Seq [Assign ‘‘z’’ (ABinary Multiply (Var
‘‘z’’) (Var ‘‘y’’)), Assign ‘‘y’’ (ABinary Subtract (Var ‘‘y’’) (Num
1))]), Assign ‘‘y’’ (Num 0)]
```

Figure 4.5: Internal representation of parsed factorial program

Organised Program Data

This is the representation of the program necessary for the analysis being performed. For this reason, there are three main representations, one for each analysis being considered, as follows: Reachability, Live Variables and Reaching Definitions Analysis.

Reachability

For Reachability Analysis the organised code will be an $n \times n$ matrix of Boolean values, where n is the number of nodes in the control flow graph of the program. This matrix will be the adjacency matrix where the value is True if there is a successor to the node and False if there is not a successor to the node. The representation of the matrix will be a list of lists, i.e. [[Bool]].

For the factorial program, this will result in the representation of the adjacency matrix as shown in Figure 4.6.

```
[[False, True, False, False, False, False], [False, False, True, False,
False, False], [False, False, False, True, False, True], [False, False,
False, False, True, False], [False, False, True, False, False, False],
[False, False, False, False, False, False]]
```

Figure 4.6: Internal representation of factorial program organised for Reachability Analysis

Live Variables
For Live Variables Analysis the organised code will be the matrix M and the vector A. Matrix M is an $n \times n$ matrix of sets where a value is the empty set if there is no successor to the node and is the complement of the $KILL$ set if there is a successor.

For the factorial program, this will result in the representation of the adjacency matrix populated with the appropriate sets as shown in Figure 4.7.

```
[[[],[‘‘x’’,‘‘y’’],[],[],[],[]], [[],[],[‘‘x’’,‘‘y’’,‘‘z’’],[],[],[]],
[[],[],[],[‘‘x’’,‘‘y’’],[],[‘‘x’’,‘‘z’’]], [[],[],[],[],[‘‘x’’,‘‘z’’],[]],
       [[],[],[‘‘x’’,‘‘y’’,‘‘z’’],[],[],[]], [[],[],[],[],[],[]]]
```

Figure 4.7: Internal representation of factorial program organised for Live Variables Analysis - matrix M

Vector A will be a vector, a $1 \times n$ matrix, of sets where a value is the empty set if there is no successor to the node and is the union of GEN sets of all successors if there is one or more successors to the node.

For the factorial program, this will result in the representation of the vector populated with the appropriate sets as shown in Figure 4.8.

```
[[[]], [[‘‘y’’]], [[‘‘y’’,‘‘z’’]], [[‘‘y’’]], [[‘‘y’’]], [[]]]
```

Figure 4.8: Internal representation of factorial program organised for Live Variables Analysis - vector A

Reaching Definitions
Like Live Variables Analysis, the organised code will be the matrix M and the vector A. Matrix M is an $n \times n$ matrix of sets where a value is the empty set if there is no predecessor to the node and is the complement of the $KILL$ set if there is a predecessor.

For the factorial program, this will result in the representation of the adjacency matrix populated with the appropriate sets as shown in Figure 4.9.

Vector A will be a vector, a $1 \times n$ matrix, of sets where a value is the empty set if there is no predecessor to the node and is the union of GEN sets of all predecessor if there is one or more predecessor to the node.

For the factorial program, this will result in the representation of the vector populated with the appropriate sets as shown in Figure 4.10.

```
[[[],[],[],[],[],[]], [[(‘‘z’’,0),(‘‘z’’,2),(‘‘z’’,4)],[],[],[],[],[]],
          [[],[(‘‘y’’,0),(‘‘y’’,1),(‘‘y’’,5),(‘‘y’’,6)],[],
               [],[(‘‘z’’,0),(‘‘z’’,2),(‘‘z’’,4)],[]],
      [[],[],[(‘‘y’’,0),(‘‘y’’,1),(‘‘y’’,5),(‘‘y’’,6),(‘‘z’’,0),
                  (‘‘z’’,2),(‘‘z’’,4)],[],[],[]],
      [[],[],[],[(‘‘y’’,0),(‘‘y’’,1),(‘‘y’’,5),(‘‘y’’,6)],[],[]],
      [[],[],[(‘‘y’’,0),(‘‘y’’,1),(‘‘y’’,5),(‘‘y’’,6),(‘‘z’’,0),
                  (‘‘z’’,2),(‘‘z’’,4)],[],[],[]]]
```

Figure 4.9: Internal representation of factorial program organised for Reaching Definitions Analysis - matrix M

```
[[[(‘‘y’’,0),(‘‘z’’,0)]], [[(‘‘y’’,1)]], [[(‘‘y’’,5),(‘‘z’’,2)]], [[]],
                         [[(‘‘z’’,4)]], [[]]]
```

Figure 4.10: Internal representation of factorial program organised for Reaching Definitions Analysis - vector A

Analysis Solution
The solutions to the analyses will be matrices with a form dependent on the analysis being performed.

Reachability
For Reachability Analysis, the output will be a $n \times n$ matrix, where n is the number of nodes in the control flow graph, representing the reachability in the program. For the factorial example, the analysis solution will be as shown in Figure 4.11.

```
   [[True,True,True,True,True,True], [False,True,True,True,True,True],
  [False,False,True,True,True,True], [False,False,True,True,True,True],
[False,False,True,True,True,True], [False,False,False,False,False,True]]
```

Figure 4.11: Internal representation of factorial program Reachability Analysis solution

Live Variables
For Live Variables Analysis, the output will be a vector where each value represents the variables that are live on exit from the node in the position. For the factorial example, the analysis solution will be as shown in Figure 4.12.

```
[[[‘‘y’’]], [[‘‘y’’,‘‘z’’]], [[‘‘y’’,‘‘z’’]], [[‘‘y’’,‘‘z’’]],
                    [[‘‘y’’,‘‘z’’]], [[]]]
```

Figure 4.12: Internal representation of factorial program Live Variables Analysis solution

Reaching Definitions

For Reaching Definitions Analysis, the output will be a vector where each value represents the definitions that reach entry to the node in the position. For the factorial example, the analysis solution will be as shown in Figure 4.13.

```
[[[(‘‘y’’,0),(‘‘z’’,0)]], [[(‘‘y’’,1),(‘‘z’’,0)]],
   [[(‘‘y’’,1),(‘‘y’’,5),(‘‘z’’,2),(‘‘z’’,4)]],
   [[(‘‘y’’,1),(‘‘y’’,5),(‘‘z’’,2),(‘‘z’’,4)]],
        [[(‘‘y’’,1),(‘‘y’’,5),(‘‘z’’,4)]],
   [[(‘‘y’’,1),(‘‘y’’,5),(‘‘z’’,2),(‘‘z’’,4)]]]
```

Figure 4.13: Internal representation of factorial program Reaching Definitions Analysis solution

Output

The output of the program is the analysis solution. This will be displayed to the user via the GHCi terminal. This is a raw and possibly inelegant way of displaying the result and is discussed in the final chapter as part of future work.

Chapter 5

Implementation

Now that the design of the system has been considered, we can progress to the implementation stage of the process. This is where the requirements and design are realised and an actual system developed. The entire anticipated system has been implemented as per the design, with some experimental implementation around the idea of real-life cost analysis. Since the implementation followed the design, the structure of this section will similarly reflect that.

5.1 Code Files and Structure

When implementing the system there were several files that have been created reflecting the design. This means that relevant code is sectioned into files making the system more manageable and maintainable. The files created are as described below:

Runner.hs : This is where the input/output of the system takes place. It contains the high-level functions that are input by the user in order to perform analyses and output the result.

ParseLanguage.hs : This contains the definition of the imperative 'while' language and parser code for producing the internal representation of the parse tree for a given program.

GeneralOrganiser.hs : This contains code that organises the parsed code, that is general to all three analyses.

ReachabilityOrganiser.hs : This contains the code that organises the parsed code into the adjacency matrix for Reachability Analysis. This relies on GeneralOrganiser.hs.

LiveVariablesOrganiser.hs : This contains the code that organises the parsed code into the matrix M and vector A for Live Variables Analysis.

ReachingDefsOrganiser.hs : This contains the code that organises the parsed code into the matrix M and vector A for Reaching Definitions Analysis.

Semirings.hs : This contains the code that defines the semirings necessary for the implemented analyses.

5.2 Parser

The core implementation of the parser is taken from *Parsing a simple imperative language* (n.d.) where an almost identical parser to the one necessary for the defined language is provided. This resource presents a parser for the 'while' language that is introduced in Nielson et al. (1999) so links nicely with the literature already examined. Since the parser is not the main focus of this project, but is an extremely important step of the solution, it was decided that this would be the best approach for implementation. By re-using existing code it meant time could be saved on this stage of implementation and used more wisely within other stages. In particular, this time was valuable when experimenting with the idea of analysing real-life costs within a program.

Of course, due to slight subtleties between the languages, and the possible need to extend the language definition, time had to be spent understanding the parser and how the Parsec library works. First we consider the arithmetic and Boolean expressions, defining the data for each, and ensuring that each type of expression is accounted for in the correct format. Some expressions rely on arithmetic, Boolean or relational operators so we ensure these are all present too. Once we have all types of operators and expressions defined, we can implement a new data structure for statements according to the grammar of the language.

With the data structures defined, the actual parser code needed to be implemented. This began with the language definition which included defining reserved names and reserved operation names used in the syntax of the language. These are names that cannot be used by identifiers. Using the definition, Parsec creates the lexer.

The parser then needed to perform the main parsing of the code. Since a program in the language is either a statement or a sequence of statements, we consider a function that parses at least one statement, or a list of statements separated by a semicolon. We define parsers for each different type of statement in the language using parsers created as part of the lexer. Similarly, we define a parser for each type of expression using operator tables in order to define each operator and how it is used (i.e. whether it is an infix or prefix operator).

Using the example on the Haskell website *Parsing a simple imperative language* (n.d.) it was simple to understand each stage and implement it, adapting the slight subtleties where necessary.

5.3 Organiser

The organiser section of the implementation was the most thought provoking part of implementation. It required the most thinking about the best approach to implementation and relied on

no existing examples of code. There are four sections to the implementation of the organiser and we shall look at each in detail in the order of implementation due to dependencies.

5.3.1 General Organiser

The general organiser contains code that is important for all three analyses. It contains a data structure for a 'flattened' nested structure; an intermediate step used to ease the creation of the adjacency matrix from the parsed representation of the code. This is implemented as follows:

```
data NestedList a = Statement a | List [NestedList a] deriving (Show,
    Eq)
```

Calculating the Number of Nodes

```
-- calculate the number of nodes in the control flow graph
calculateNodes :: Stmt -> Int -> Int
calculateNodes (Seq []) n = n + 0
calculateNodes (Seq (x:xs)) n = calculateNodes x n + calculateMore xs n
calculateNodes (Assign _ _) n = n + 1
calculateNodes (While _ a) n = 1 + calculateNodes a n
calculateNodes (If _ a b) n = 1 + calculateNodes a n + calculateNodes
    b n
calculateNodes Skip n = n + 1

calculateMore :: [Stmt] -> Int -> Int
calculateMore xs n = foldr (\ x -> (+) (calculateNodes x n)) n xs
```

The function calculateNodes is used to calculate the number of nodes in the control-flow graph of the program. This is done by taking the parsed representation of the program and iterating over it in such a way that it counts the number of nodes, using pattern matching to identify different types of statements. For a sequence of statements the implementation calls calculateNodes for the first statement in the sequence and then calculateMore on the rest. The calculateMore function takes a list of statements and calls calculateNodes on each item in the list.

For 'assign' and 'skip' statements the value is simply n plus 1, where n is the current count of nodes. For 'while' statements the value is n plus 1 (the Boolean test) plus the number of nodes inside the 'while' block. For this reason, we call calculateNodes on the sequence of statements following the Boolean test. Similarly, for an 'if' statement the value is n plus 1 (the Boolean test) plus the number of nodes inside the true branch of the 'if' block plus the number inside the false branch of the 'if block'.

Creating the Control-Flow Graph

```
calculateCfg :: Stmt -> NestedList String -> NestedList String
calculateCfg (Seq []) _ = List []
calculateCfg (Seq (x:xs)) (List n) = List (calculateCfg x (List n) :
    calculateMoreCfg xs (List n))
calculateCfg (Seq (x:xs)) (Item n) = List (calculateCfg x (Item n) :
    calculateMoreCfg xs (Item n))
calculateCfg (Assign _ _) _ = Item "Assign"
calculateCfg (While _ a) (List n) = List (Item "While" : [calculateCfg
    a (List n)])
calculateCfg (While _ a) (Item n) = List (Item "While" : [calculateCfg
    a (Item n)])
calculateCfg (If _ a b) (List n) = List ([Item "If"] ++ [calculateCfg
    a (List n)] ++ [calculateCfg b (List n)])
calculateCfg (If _ a b) (Item n) = List ([Item "If"] ++ [calculateCfg
    a (Item n)] ++ [calculateCfg b (Item n)])
calculateCfg Skip _ = Item "Skip"

calculateMoreCfg :: [Stmt] -> NestedList String -> [NestedList String]
calculateMoreCfg xs (List n) = foldr (\ x -> (++) [calculateCfg x
    (List n)]) [] xs
calculateMoreCfg xs (Item n) = foldr (\ x -> (++) [calculateCfg x
    (Item n)]) [] xs
```

This is used to aid with the creation of the adjacency matrix by first creating an intermediate, flattened, nested structure representing the parsed code. This structure makes implementation of the adjacency matrix simpler due to the way that the nested structure is composed. The function calculateCfg is used to do this. It takes the parsed representation of the program, like calculateNodes and iterates over it in such a way that results in a NestedList - the nested structure.

Each node in the control-flow graph is represented by a **String**, either: "Assign", "While", "If" or "Skip". Within a while block, after the Boolean test, all statements are within a nested list. Similarly in an **if** block, after the Boolean test, all statements in the true branch are within a nested list and all statements in the false branch are within a second nested list. This results in the nested list structure that makes creating the adjacency matrix much simpler, by just considering lengths of lists and nested lists.

Counting the Number of Nodes in part of a Control-Flow Graph

```
countCfg :: NestedList a -> Int
countCfg (Item _) = 1
countCfg (List xs) = sum $ map countCfg xs
```

The function countCfg is used to calculate the number of nodes in a section of the control-flow graph. It takes a NestedList, which could be the entire control-flow graph or just a part of it, and calculates the number of nodes. The function needs to iterate over the NestedList structure and reach the deepest nested list or lists in order to calculate the total number. This is done by mapping over the structure and adding 1 every time a Statement is identified.

5.3.2 Reachability Organiser

Within the Reachability organiser, the adjacency matrix is created. This is a matrix of Boolean values, represented by [[**Bool**]]. The function createMatrix takes the nested list structure of the control-flow graph, created by the function calculateCfg in the general organiser, the number of nodes in the control-flow graph and keeps count of the current node while iterating to create the matrix. The number of the current node is necessary for calculating when to jump, for example at the end of the 'while' block.

The function createMatrix iterates over the nested structure, representing the control-flow graph, identifying different statements. There are several options, the main ones being the different types of statements in the language:

- Assign statement: get this row of the matrix using the number of nodes in the control flow graph and the number of the current node, then call createMatrix on the rest of the structure
- While statement: create a nested list containing the matrix row for the Boolean test node using the number of nodes in the control flow graph, the current node number and the length of the rest of the 'while' block, then call nestedWhile on the rest of the 'while' block and then createMatrix on the rest of the structure
- If statement: create a nested list containing the matrix row for the Boolean test node using the number of nodes in the control flow graph, the current node number and the length of the rest of the 'if' true branch, then call nestedIf on the rest of the 'if' block and createMatrix on the rest of the structure
- Skip statement: get this row of the matrix using the number of nodes in the control flow graph and the number of the current node, then call createMatrix on the rest of the structure

Different functions are needed for 'while' and 'if' statements as they require extra information. The function nestedWhile has two extra arguments to keep track of the node numbers of the Boolean test for the 'while' statement as well as the last node in the 'while' block. This means it is possible to determine when to jump back to the Boolean test when the end of the 'while' block is reached, as well as where to jump to if the Boolean test fails. Similarly for the nestedIf function which has the two extra arguments end and jump. The end is used to determine the end of the true branch and the jump is to keep track of the node number of the last node in the false branch, i.e. where to jump to at the end of the true branch. These two functions iterate like the createMatrix function but include the extra information to handle the different cases appropriately.

There are several different functions to get matrix rows depending on different cases. These functions are described below. We shall use two main example control-flow graphs of programs, see Figure 5.1 and Figure 5.2, to illustrate where each different case is necessary.

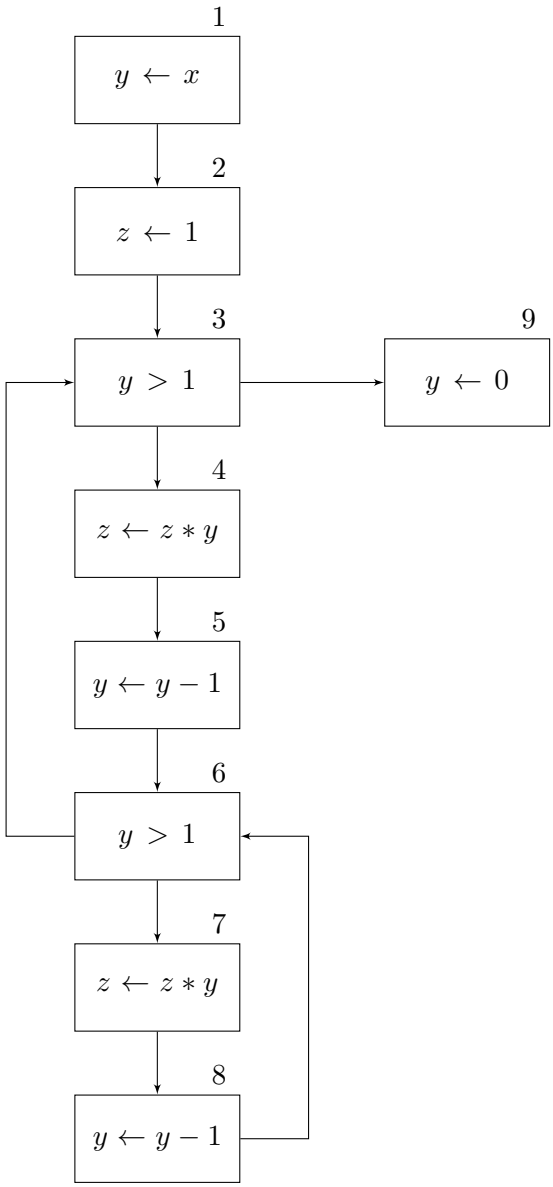

Figure 5.1: Control-flow graph for a Nested 'While' Structure

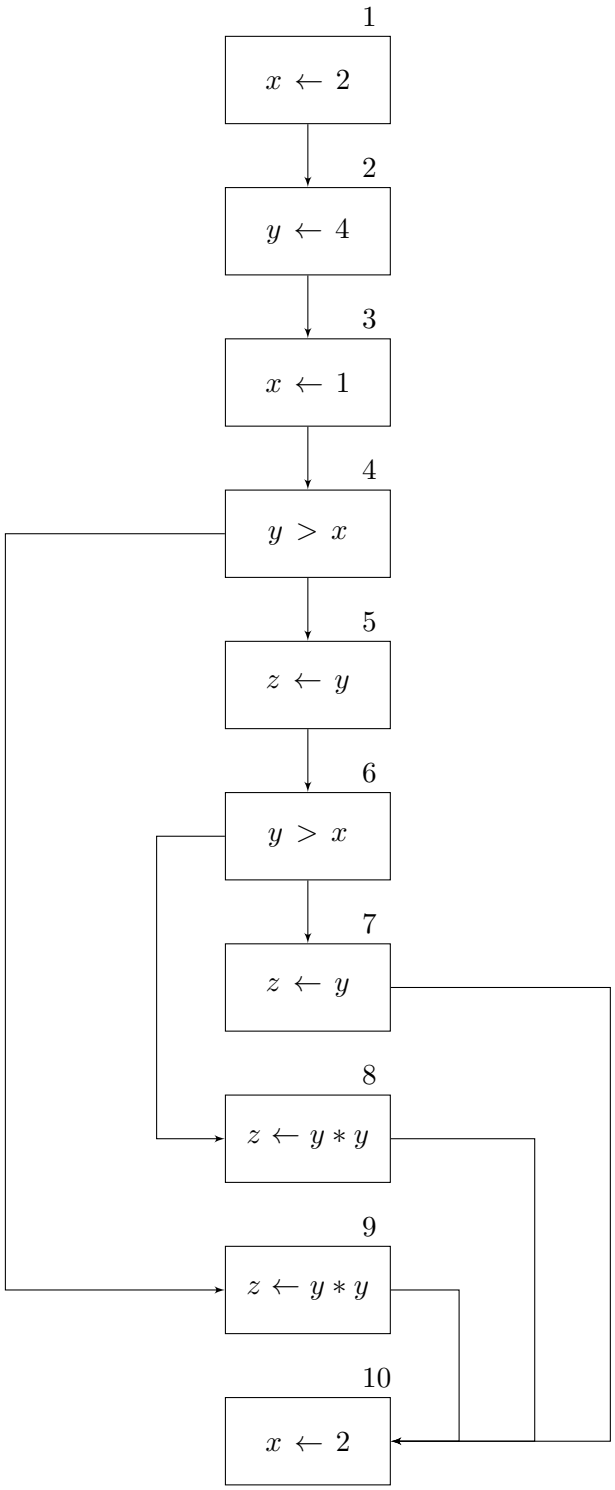

Figure 5.2: Control-flow graph for a Nested 'If' Structure

getMatrixRow
The simplest and most commonly used function is getMatrixRow which takes the number of nodes in the control-flow graph as well as the current node number. If the node number is the final node in the graph then the row is simply a list of **False** values since there is never a successor to the final node. Otherwise, the matrix row is a list of **False** values but with **True** in the position of the current node number + 1, identifying that there is a flow of control between the current node and the next node. For example, in Figure 5.1 this case would be used for nodes 1, 2 and 9.

getWhileMatrixRow
The function getWhileMatrixRow is the function used to get the matrix row for the Boolean test node in an un-nested 'while' statement. The function takes as arguments: the number of nodes in the control flow graph, the current node number and the number of the node following the entire 'while' block, a value to keep count of the position in the row as it is composed and the list representing matrix row so far. The function keeps count of the position in the row as it composes it, entering **False** as the value for all positions other than the number of the current node + 1, identifying a flow of control between the current node and the following node, and the first node after the 'while' block, identifying a flow of control between the current node and that one. For example, in Figure 5.1 this case would be used for node 3.

getIfMatrixRow
The function getIfMatrixRow is the function used to get the matrix row for the Boolean test node in an 'if' statement. This function takes as arguments: the number of nodes in the control flow graph, the current node number, the number of the node at the end of the true branch of the 'if' statement, a value to keep count of the position in the row as it is composed and the list representing matrix row so far. The function keeps count of the position in the row as it composes it, entering **False** as the value for all positions other than the number of the current node + 1, identifying a flow of control between the current node and the first node in the true branch, and the node at the end of the true branch + 1, identifying a flow of control between the current node and the first node in the false branch. For example, in Figure 5.2 this case would be used for nodes 4 and 6.

getNestedMatrixRow
The function getNestedMatrixRow is similar to the getMatrixRow function, except it is used inside the nestedWhile function. This function requires extra arguments so that it can calculate the correct matrix row considering the extra necessity for a jump back to the Boolean test at the end of a 'while' block. The function takes as arguments: the number of nodes in the control flow diagram, the current node number, the node number for the Boolean test of the 'while' statement, the node number for the final node in the 'while' block, the current position for the composition of the matrix row and a list representing the matrix row so far. If the current node is not the final node in the 'while' block, the function calls getMatrixRow and composes the matrix row as it does when it is not a statement within a nested 'while'. Otherwise, the function composes the matrix row by putting **False** in all positions except the position representing flow of control between the current node and the Boolean test for the 'while' statement. For example, in Figure 5.1 this case would be used for nodes 4, 5, 7 and 8.

getIfThenMatrixRow
The function getIfThenMatrixRow is similar to the getMatrixRow and getNestedMatrixRow functions, but works slightly differently due to the structure of an 'if/else' statement. It is used to get the matrix row for assign and 'skip' statements within an 'if/else' statement. It takes as arguments: the number of nodes in the control flow diagram, the current node number, the node number for the node at the end of the true branch of the 'if' statement, the node number for the first node after the 'if' statement, the current position for the composition of the matrix row and a list representing the matrix row so far. If the current node is not last node in the true branch of the 'if' statement then the function calls getMatrixRow and composes the matrix row as it does when it is not a statement within a nested 'if'. Otherwise, the function composes the matrix row by putting **False** in all positions except the position representing flow of control between the final node in the true branch and the first node after the 'if' statement. For example, in Figure 5.2 this case would be used for nodes 5, 7, 8 and 9.

getNestedWhileMatrixRow
The function getNestedWhileMatrixRow is similar to getWhileMatrixRow but is used for nested 'while' statements due to the fact that the flow of control is slightly different. The function takes as arguments: the number of nodes in the control flow diagram, the current node number, the node number for the start of the outer 'while' Boolean test, the node number for the end node of the nested 'while' block, the current position for the composition of the matrix row and a list representing the matrix row so far. It works in a similar way to getWhileMatrixRow, but it must determine whether to put a **True** value representing flow of control between the current node and the first node after the 'while' block, if there are statements following the nested 'while' but still enclosed in the outer 'while' loop, otherwise it puts a **True** value representing flow of control between the current node and the Boolean test of the outer 'while' statement. For example, in Figure 5.1 this case would be used for node 6.

5.3.3 Live Variables Organiser

Live Variables Analysis relies on the adjacency matrix, so the adjacency matrix created for Reachability Analysis is used to aid with implementation of the necessary structures for Live Variables. Within this section of the code, the matrix M and vector A, as described in the equations solving Live Variables Analysis, are created. In order to do this, the $\overline{KILL}$ and GEN sets for each node are needed.

Creating the GEN sets
The GEN sets are calculated by calling the calculateGens function which takes 2 arguments: the parsed representation of the program as produced by the parser and a list of all the GEN sets calculated so far. The result is a list of sets, [Set **String**], where the i^{th} element in the list represents the GEN set for the i^{th} node in the control flow graph. This list is created like the control-flow graph representation, created using the calculateCfg function in the general organiser, and the function iterates over the parsed structure in the same way.

The difference to the function calculateCfg is that, instead of using a **String** to represent each type of statement, a set of variables is entered into that node position. For Live Variables, the

GEN set is the set of variables that are used in the node before any assignment. This set is composed slightly differently for each different type of statement, as follows:

- Assignment statement: We do not consider the variable on the left hand side of the assignment as this is not a variable used before any assignment in the node. We need a set of variables that are used on the right hand side of the assignment which could be either a variable, a number or an arithmetic expression. If the right hand side is simply a variable, the set is a singleton set containing just that variable. If the right hand side is a single number, the set is the empty set since no variable is used. If the right hand side is an arithmetic expression, the function getAGenSet, see Listing 5.1, is called. This takes the arithmetic expression and composes a set of all variables used within it.
- While statement: A 'while' statement is a Boolean test followed by a sequence of statements. For this reason, we compose the set of variables used in the Boolean test and then continue by appending to the list the *GEN* sets for the nodes within the 'while' block, by calling calculateGens on the rest of the 'while' statement. For the Boolean test we are concerned with a Boolean expression, BExpr as defined in the parser, so the function getRGenSet, see Listing 5.2, is called. This takes a Boolean expression and composes a set of all variables used within it.
- If statement: An 'if' statement is a Boolean test followed by a sequence of statements for the true branch and another sequence of statements for the false branch. For this reason, we compose the set of variables used in the Boolean test and then continue by appending to the list the *GEN* sets for the nodes within the true branch and then the *GEN* sets for the nodes within the false branch. The *GEN* set for the Boolean test is composed as for in a 'while' statement.
- Skip statement: A 'skip' statement never uses any variables, so this is always just the empty set.

Listing 5.1: Calculate Variables used in Arithmetic Expression

```
getAGenSet :: AExpr -> Set String
getAGenSet (ABinary _ (Var x) (Var y)) = (Set.union (Set.singleton x)
    (Set.singleton y))
getAGenSet (ABinary _ (Var x) (Num _)) = (Set.singleton x){}
getAGenSet (ABinary _ (Num _) (Var y)) = (Set.singleton y)
getAGenSet (ABinary _ (Var x) y) = (Set.union (Set.singleton x)
    (getAGenSet y))
getAGenSet (ABinary _ x (Var y)) = (Set.union (getAGenSet x)
    (Set.singleton y))
getAGenSet (ABinary _ x y) = (Set.union (getAGenSet x) (getAGenSet y))
```

Listing 5.2: Calculate Variables used in Boolean Expression

```
getRGenSet :: BExpr -> Set String
getRGenSet (RBinary _ (Var x) (Var y)) = (Set.union (Set.singleton x)
    (Set.singleton y))
```

```
getRGenSet (RBinary _ (Var x) (Num _)) = (Set.singleton x)
getRGenSet (RBinary _ (Num _) (Var y)) = (Set.singleton y)
getRGenSet (RBinary _ (Var x) y) = (Set.union (Set.singleton x)
    (getAGenSet y))
getRGenSet (RBinary _ x (Var y)) = (Set.union (getAGenSet x)
    (Set.singleton y))
getRGenSet (RBinary _ x y) = (Set.union (getAGenSet x) (getAGenSet y))
```

Once the entire structure has been iterated over, the result is a list of sets of variables (represented by strings) and the set for a given node i can be accessed by taking the i^{th} element of the list.

Creating the $\overline{KILL}$ sets

The $KILL$ set for a node is the set of variables assigned a value in the node. For Live Variables we need the $\overline{KILL}$ set, the complement of the $KILL$ set, which means we need to know all of the variables in the program. In order to do this, the function getAllVarsInProgram, see Listing 5.3, is called. This takes the parsed representation of the program as well as a set representing the variables 'found' so far. It iterates over the structure in a similar way to calculateGens and composes one set by taking the union (Set.**union**), of all of the variables in the program.

Listing 5.3: Calculate All Variables in the Program

```
getAllVarsInProgram :: Stmt -> Set String -> Set String
getAllVarsInProgram (Seq []) _ = Set.empty
getAllVarsInProgram (Seq (x:xs)) n = Set.union (getAllVarsInProgram x
    n) (getMoreVarsInProgram xs n)
getAllVarsInProgram (Assign x (Var y)) _ = (Set.union (Set.singleton
    x) (Set.singleton y))
getAllVarsInProgram (Assign x (Num _)) _ = Set.singleton x
getAllVarsInProgram (Assign x y) _ = Set.union (Set.singleton x)
    (getAGenSet y)
getAllVarsInProgram (While x a) n = Set.union (getRGenSet x)
    (getAllVarsInProgram a n)
getAllVarsInProgram (If x a b) n = Set.union (Set.union (getRGenSet x)
    (getAllVarsInProgram a n)) (getAllVarsInProgram b n)
getAllVarsInProgram Skip _ = Set.empty

getMoreVarsInProgram :: [Stmt] -> Set String -> Set String
getMoreVarsInProgram xs n = foldr (\ x -> Set.union
    (getAllVarsInProgram x n)) Set.empty xs
```

In order to calculate the $\overline{KILL}$ set, the function calculateKills is called. This is similar to calculateGens except it takes one extra argument, the set of all variables in the program, so that it can create the complement set. Since we are only concerned with the set of variables assigned a value in the node, for Boolean tests and 'skip' statements the set will always be the entire set of variables in the program. For an assign statement, the $\overline{KILL}$ set is the set of all variables in the program minus the variable being assigned in that node. This is calculated by taking the

difference, using the function Set.difference.

Once the entire structure has been iterated over, the result is a list of sets of variables (represented by strings) and the set for a given node i can be accessed by taking the i^{th} element of the list.

Creating matrix M
Since matrix M is the adjacency matrix, but with **False** values replaced with the empty set and **True** values replaced with a $\overline{KILL}$ set, we can use the adjacency matrix already created in the Reachability organiser.

The function createLVMatrixM takes the adjacency matrix, an **Int** to keep count of the column (node) and the list of all $\overline{KILL}$ sets. It iterates over the adjacency matrix, a [[**Bool**]] structure, replacing each value as necessary, using the integer count to access the correct $\overline{KILL}$ set in each position. The final result is a matrix represented as a list of lists, [[Set **String**]].

Listing 5.4: Live Variables Analysis - Creating Matrix M

```
createLVMatrixM :: [[Bool]] -> Int -> [Set String] -> [[Set String]]
createLVMatrixM [] _ _ = []
createLVMatrixM (x:xs) count kills = iterateMatrixRow x count kills ++
    createLVMatrixM xs count kills

iterateMatrixRow :: [Bool] -> Int -> [Set String] -> [[Set String]]
iterateMatrixRow [] _ _ = []
iterateMatrixRow (x:xs) count kills = if x == True then
    [[kills!!count] ++ iterateMatrixRow2 xs (count+1) kills]
     else [[(Set.empty)] ++ iterateMatrixRow2 xs (count+1) kills]

iterateMatrixRow2 :: [Bool] -> Int -> [Set String] -> [Set String]
iterateMatrixRow2 [] _ _ = []
iterateMatrixRow2 (x:xs) count kills = if x == True then
    [kills!!count] ++ iterateMatrixRow2 xs (count+1) kills
     else [Set.empty] ++ iterateMatrixRow2 xs (count+1) kills
```

Creating vector A
Vector A is composed in a similar way to matrix M by again considering the adjacency matrix, however, it is a $1 \times n$ matrix (a vector) rather than an $n \times n$ matrix. It is also composed by considering the GEN sets rather than the $\overline{KILL}$ sets. In order to create the vector, rather than composing lists for each row, a single set is generated by taking the union of the necessary values.

Listing 5.5: Live Variables Analysis - Creating Vector A

```
createLVVector :: [[Bool]] -> Int -> [Set String] -> [[Set String]]
createLVVector [] _ _ = []
createLVVector (x:xs) count gens = iterateVectorRow x count gens ++
    createLVVector xs count gens
```

```
iterateVectorRow :: [Bool] -> Int -> [Set String] -> [[Set String]]
iterateVectorRow [] _ _ = []
iterateVectorRow (x:xs) count gens = if x == True then [[Set.union
    (gens!!count) (iterateVectorRow2 xs (count+1) gens)]]
     else [[Set.union Set.empty (iterateVectorRow2 xs (count+1) gens)]]

iterateVectorRow2 :: [Bool] -> Int -> [Set String] -> Set String
iterateVectorRow2 [] _ _ = Set.empty
iterateVectorRow2 (x:xs) count gens = if x == True then Set.union
    (gens!!count) (iterateVectorRow2 xs (count+1) gens)
     else Set.union Set.empty (iterateVectorRow2 xs (count+1) gens)
```

5.3.4 Reaching Definitions Organiser

Reaching Definitions Analysis is almost identical to Live Variables Analysis in terms of composing the matrix M and vector A, the main difference being that the [[**Bool**]] passed to the functions is the transpose of the adjacency matrix rather than the original adjacency matrix. The only other difference is that instead of considering sets of variables, we are now considering sets of variable and label pairs. This means that the generation of the GEN and $\overline{KILL}$ sets is slightly different.

Calculating a different CFG representation for Reaching Definitions Analysis
For Reaching Definitions Analysis, a different control-flow graph is created as different information is more useful for creating the GEN and $\overline{KILL}$ sets. In this analysis it is necessary to know the variables that are assigned and re-assigned in each node. The function calculateRDCfg is almost identical to calculateCfg in the general organiser and iterates over the parsed representation of the program in the same way. The Reaching Definitions control-flow graph is almost the same, except instead of having the **String** "Assign" for assign statements, we have the variable that is assigned instead.

Creating the GEN sets
To calculate the GEN sets the function calculateRDGens is used. The NestedList structure is iterated over in a similar way to the way a NestedList is iterated over when calculating the general control-flow graph representation. A count of the node number is kept while iterating over the structure so to be able to create pairs of variables and node labels that are necessary for Reaching Definitions Analysis. For the GEN sets it is fairly simple, we are concerned with the variables that are written/assigned by each node and therefore for 'while', 'if' and 'skip' statements this is just the empty set. For assign statements we create a set containing one pair with the variable being assigned, the Statement element in this new representation, and the current node number.

Once the entire structure has been iterated over, and like creating the GEN sets for Live Variables Analysis, the result is a list of sets of pairs and the set for a given node i can be accessed by taking the i^{th} element of the list.

Creating the $\overline{KILL}$ sets
Calculating the $\overline{KILL}$ sets is done by the function calculateRDKills and is very similar to the way the GEN sets is calculated, except this time we are concerned with the complement of all variables that are re-assigned in the node. For this reason, we need to know all of the pairs of variables and nodes assigned in the program. This is calculated by the function getAllDefsInProgram, see Listing 5.6, which iterates over the GEN set and generates all possible pairs of assigns in the program. This includes, for each variable, a pair of the form (var, 0) where the 0 represents an unassigned variable at the beginning of the program.

Listing 5.6: Calculate All Definitions in the Program

```
getAllDefsInProgram :: [Set (String, Int)] -> Set (String, Int)
getAllDefsInProgram (x:[]) = x
getAllDefsInProgram (x:xs) = if (x == Set.empty) then (Set.union
    Set.empty (getAllDefsInProgram xs)) else (Set.union (Set.union x
    (getAllDefsInProgram xs)) (Set.singleton (fst (Set.elemAt 0 x), 0)))
```

In order to generate the $\overline{KILL}$ sets by the calculateRDKills, the NestedList structure is iterated over in the same way as the calculateRDGens function. This time though the function composes the $\overline{KILL}$ sets and uses the GEN sets to do this (to generate all the definitions in the program). We are concerned with variables that are re-assigned by each node and therefore for 'while', 'if' and 'skip' statements the complement is the entire set of variables and label pairs in the program, generated by getAllDefsInProgram. For assign statements it is the set containing all of the definitions in the program other than the assigned variable and label pair for the current node.

Once the entire structure has been iterated over the result is a list of sets of pairs and the set for a given node i can be accessed by taking the i^{th} element of the list, similar to creating the $\overline{KILL}$ sets for Live Variables Analysis.

5.4 Semirings

Following the design of the semirings, the implementation was fairly trivial. First the semiring class, of which the necessary semirings are instances of, was defined as follows:

```
class Semiring a where
    zero    :: a
    one     :: a
    sAdd    :: a -> a -> a
    sMult   :: a -> a -> a
    closure :: a -> a
```

Then, for each semiring necessary, the instance was implemented. For example, for the Boolean semiring the following instance was implemented:

```
instance Semiring Bool where
    zero       = False
    one        = True
    sAdd       = (||)
    sMult      = (&&)
    closure _  = True
```

The slightly more difficult instance to implement was the Matrix Semiring. This implementation was taken from Dolan (2013) where an appropriate matrix data structure had been implemented as well as the necessary matrix operations. Since the work had already been done, and the understanding of it and working of the mathematics examined in the literature survey, the decision was made to re-use this implementation. This meant that time was saved and could be used elsewhere; in areas more interesting to the development and experimental section of this project.

5.5 Runner

The runner file is the main file of the program. This file imports all of the others and applies the functions in the necessary order to perform the different analyses. It is also the only file that deals with IO. Each of the three main functions simply take one argument - the name of the file containing the program to be analysed - and outputs the solution to the analysis.

Reachability

```
reachability :: String -> IO()
reachability file = do
    x <- parseFile file
    let nodes = (calculateNodes x 0)
    let cfg = (calculateCfg x (List []))
    print (closure (Matrix (createMatrix cfg nodes 0)))
```

The reachability function is the one used to perform Reachability Analysis. It takes the name of the program file and parses it. Once it has the parsed representation it calculates the control-flow graph as well as the number of nodes in it. These can then be used to create the adjacency matrix and perform closure on it.

Live Variables

```
liveVariables :: String -> IO()
liveVariables file = do
    x <- parseFile file
    let nodes = (calculateNodes x 0)
    let cfg = (calculateCfg x (List []))
    let gens = (calculateKills x (getAllVarsInProgram x Set.empty)
        [Set.empty])
```

```
    let kills = (calculateKills x (getAllVarsInProgram x Set.empty)
        [Set.empty])
    print (changeLast 0 (nodes-1) (sMult (closure
        (Matrix (createLVMatrixM (createMatrix cfg nodes 0) 0 kills)))
        (Matrix (createLVVector (createMatrix cfg nodes 0) 0  gens))))
```

The liveVariables function is the one used to perform Live Variables Analysis. For Live Variables, more information needs to be generated. The function takes the name of the program file and parses it. Once it has the parsed representation it calculates the control-flow graph as well as the number of nodes in it. This is the same as for Reachability Analysis. It then generates the GEN and $\overline{KILL}$ sets which are necessary for generating the matrix M and vector A. The matrix M is generated using the $\overline{KILL}$ sets and the vector A is generated using the GEN sets. Closure is performed on the matrix M and then the result of this is multiplied by vector A.

Finally, the final row of the matrix must always have in every position the empty set. This is because in Live Variables Analysis it is true that on exit from the final node there are no live variables as it is the end of the program. The function changeLast ensures that the final row of the matrix is all empty sets.

Reaching Definitions

```
reachingDefinitions :: String -> IO()
reachingDefinitions file = do
    x <- parseFile file
    let nodes = (calculateNodes x 0)
    let cfg = (calculateCfg x (List []))
    let rdCfg = (calculateRDCfg x (List []))
    let gens = (calculateRDGens rdCfg 1 [])
    let kills = (calculateRDKills rdCfg 1 [] gens)
    print (sMult (closure
        (Matrix (createRDMatrixM (transpose (createMatrix cfg nodes
            0)) 0 kills)))
        (Matrix (createRDVector (transpose (createMatrix cfg nodes 0))
            0 0 gens)))
```

The reachingDefinitions function is the one used to perform Reaching Definitions Analysis. Reaching Definitions is performed in almost exactly the same way as Live Variables Analysis, except we now consider the transpose of the adjacency matrix when composing matrix M and vector A.

Chapter 6

Testing and Evaluation

Once the system has been implemented, it is important to test it in order to validate and verify what has been produced. In order to do this we reflect back to the requirements set out at the beginning which will determine the success of the project. We also need to test on a lower level for correctness. We will first look at unit level testing followed by whole program level testing. Finally, we will re-examine the requirements and conclude the success of the project based on the testing and results.

Note: a test suite of files (example programs) used for testing accompany this document. When we refer to more tests than the examples provided being carried out then we are referring to this test suite.

6.1 Unit Level Testing

Unit level testing is low-level testing which was carried out during development of each component of the system in order to ensure correctness. In this section we examine the implementation of low-level functions within the code, ensuring that they produce the expected output. It is important to consider not only common cases, but also boundary and unusual cases, so to ensure robustness and reliability of the program.

6.1.1 Parser Testing

In order to test the parser component of the system we run the parseFile function with a range of different programs of different structures. By doing this, we ensure that the parser can handle different program structures correctly, producing the expected output.

Valid Inputs

The simple program used for the majority of testing was the factorial program. We provide the function parseFile with a .txt file containing the factorial program in the given concrete syntax

for the language.

Listing 6.1: Valid Factorial Input

```
y := x;
z := 1;
while (y>1) do
        (z := z*y;
        y := y-1)
end;
y:=0
```

Given the implentation of the parser, the output for this program was expected, as shown in Figure 6.1.

```
Seq [Assign ``y'' (Var ``x''),Assign ``z'' (Num 1),While (RBinary Greater
(Var ``y'') (Num 1)) (Seq [Assign ``z'' (ABinary Multiply (Var ``z'') (Var
``y'')),Assign ``y'' (ABinary Subtract (Var ``y'') (Num 1))]),Assign ``y''
                                (Num 0)]
```

Figure 6.1: Parsed representation of the Factorial Program

Following this simple example with one 'while' statement further valid inputs were tested. This included a range of statements, sequences of statements and nested structures for the program including nested 'while' and 'if' statements and combinations of the two. Similarly, it was important to test all of the different operators (arithmetic, Boolean and relational) in order to ensure the correct implementation of the parser.

Invalid Inputs

As well as valid inputs, invalid inputs were tested. This focused on programs with subtle syntax errors that should not be parsed. As an example, the factorial program with a missing semicolon between a sequence of statements was tested. The program was:

Listing 6.2: Invalid Factorial Input

```
y := x;
z := 1;
while (y>1) do
        (z := z*y
        y := y-1)
end;
y:=0
```

The result was as shown in Figure 6.2.

```
                    (line 5, column 9):
                      unexpected ‘‘y’’
          expecting ‘‘*’’, ‘‘/’’, operator, ‘‘;’’ or ‘‘)’’
              *** Exception:  user error (parse error)
```

Figure 6.2: Invalid program error

This was the expected result, since the program does not follow the rules for the concrete syntax of the 'while' language defined. For similar invalid inputs, the expected error messages were provided as anticipated.

An important thing to comment on is that these errors are basic. They provide useful information about where the syntax issue is and what the problem is, however they are displayed in a very raw way. Since the aim of this project was to implement an experimental and adaptable system for possible integration with other systems, or for a potential graphical user interface to be implemented over the top in the future, this sort of error message is acceptable.

6.1.2 General Organiser Testing

In the general organiser section of the system, we look at testing the correct implementation for calculating the number of nodes in a program and composing the correct nested structure representing the control-flow graph. Again, it is important to test programs with a range of structures in order to ensure robustness. To calculate the expected result, the control-flow graph of a program was created manually allowing for the number of nodes to be counted as well as the representation of the control-flow graph as a nested structure to be calculated.

The function calculateNodes takes a Stmt as an argument so we use the parsed representation of the factorial program as produced by the parser in the first test. The control-flow graph for this program is shown in Figure 4.3 therefore we can see that this function should result in a count of 6 nodes. This was the correct result. In addition to this test, nested structures with different combinations of statements were tested in order to ensure that iteration and counting over the parsed representation is correct for all valid inputs.

The function calculateCfg computes the nested structure representing the control-flow graph. For the factorial program we expect the nested structure to be as shown in Figure 6.3.

```
   List [Statement ‘‘Assign’’,Statement ‘‘Assign’’,List [Statement
‘‘While’’,List [Statement ‘‘Assign’’,Statement ‘‘Assign’’]],Statement
                            ‘‘Assign’’]
```

Figure 6.3: Nested Factorial Program

Since these functions will only ever be used on a Stmt (recall the data type of a parsed program) once a program has been parsed, we can assume that valid Stmts will only ever be used as input. For this reason, we only considered valid inputs but ensured that a wide range of the different structures were considered.

6.1.3 Reachability Organiser Testing

Within the Reachability organiser, we are testing whether the creation of the adjacency matrix is correct by testing the createMatrix function. This takes a NestedList **String**, which is the output of the calculateCfg function in the general organiser, so we can use this output from previous tests. Furthermore, since this function will only ever be used with the output of calculateCfg, we can assume only valid inputs.

As part of this, we are ensuring that the other functions within this file are correct. In order to test *all* of the functions, we need to test all cases making sure that all functions are called. This means we need to consider, again, a wide range of sequences of statements and program structures so to cover all possiblilities. This tests the correct iteration over the NestedList **String** structure as well as the correct composition of the matrix and each matrix row.

For the factorial program, we expect the resulting matrix to be as shown in the Design chapter in Figure 4.6. When tested, the output was the expected result. This simple test only covered a single 'while' statement, so further tests were carried out as necessary to cover all cases.

6.1.4 Live Variables Organiser Testing

Within the Live Variables section of the code there were several things to test:

- the calculation of the GEN sets
- the calculation of the $\overline{KILL}$ sets
- the creation of matrix M
- the creation of vector A

For the calculation of the GEN sets by the function calculateGens we take a Stmt as created by the function parseFile so we are concerned with the output of the parser. Since this function will only ever be used following the parser, we can assume only valid inputs, as the parser will identify any invalid programs. Anything that is validly parsed by the parser can be a valid input for this function.

For the factorial example, we expect the GEN sets to be as shown in Table 6.1. Within the program, this is represented as a [Set **String**] where the i^{th} element of the list is the GEN set for node number i. The result, as expected, of the test is as shown in Figure 6.4 and therefore the test passed.

Table 6.1: *GEN* sets for the factorial program

Node	*GEN*
1	x
2	∅
3	y
4	y, z
5	y
6	∅

```
[fromList [‘‘x’’],fromList [],fromList [‘‘y’’],fromList
     [‘‘y’’,‘‘z’’],fromList [‘‘y’’],fromList []]
```

Figure 6.4: *GEN* sets for factorial program - test result

In order to ensure that all of the different cases are tested, a variety of tests were carried out with all different types of arithmetic and Boolean expressions.

In order to calculate the $\overline{KILL}$ sets a set of all the variables in the program is needed. We test getAllVarsInProgram in order to ensure that this set is correct and complete. The function takes a Stmt as the argument therefore we can assume any valid parsed program is an input. For the factorial program the set of all variables in the program is $\{x, y, z\}$ therefore the result `fromList [‘‘x’’,‘‘y’’,‘‘z’’]` means the test passes. Again, we run more tests on a variety of programs ensuring that the function computes the correct set for a wide range of structures of programs.

The function calculateKills is similar to calculateGens but takes an extra argument; the set of all variables in the program. Therefore we test the function calculateKills in the same way as calculateGens. For the factorial program, we expect the $\overline{KILL}$ sets to be as shown in Table 6.2. Within the program, this is represented as a [Set **String**] where the i^{th} element of the list is the $\overline{KILL}$ set for node number i. The result, as expected, of the test is as shown in Figure 6.5 and therefore the test passed.

Table 6.2: $\overline{KILL}$ sets for the factorial program

Node	$\overline{KILL}$
1	x, z
2	x, y
3	x, y, z
4	x, y
5	x, z
6	x, z

```
[fromList [‘‘x’’,‘‘z’’],fromList [‘‘x’’,‘‘y’’],fromList
[‘‘x’’,‘‘y’’,‘‘z’’],fromList [‘‘x’’,‘‘y’’],fromList [‘‘x’’,‘‘z’’],fromList
[‘‘x’’,‘‘z’’]]
```

Figure 6.5: $\overline{KILL}$ sets for factorial program - test result

Using the generation of the GEN and $\overline{KILL}$ sets we create the matrix M and vector A. For this reason, we must test the functions used to do this, ensuring that the creation is correct. Since the functions rely on the creation of the adjacency matrix and the created GEN and $\overline{KILL}$ sets that have already been tested, we are simply checking that the values in the adjacency matrix are replaced with the correct sets.

The function createLVMatrixM was tested to ensure that the matrix M is populated with the correct values. For the factorial program, we expect the matrix M to be as shown below:

$$\begin{pmatrix} \emptyset & \{x,y\} & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \{x,y,z\} & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \{x,y\} & \emptyset & \{x,z\} \\ \emptyset & \emptyset & \emptyset & \emptyset & \{x,z\} & \emptyset \\ \emptyset & \emptyset & \{x,y,z\} & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{pmatrix}$$

The result of the test was as shown in Figure 6.6. This is the internal representation of the matrix shown above and therefore the test passed.

```
[[fromList [],fromList [‘‘x’’,‘‘y’’],fromList [],fromList [],fromList
[],fromList []], [fromList [],fromList [],fromList
[‘‘x’’,‘‘y’’,‘‘z’’],fromList [],fromList [],fromList []], [fromList
[],fromList [],fromList [],fromList [‘‘x’’,‘‘y’’],fromList [],fromList
[‘‘x’’,‘‘z’’]],[fromList [],fromList [],fromList [],fromList [],fromList
[‘‘x’’,‘‘z’’],fromList []],[fromList [],fromList [],fromList
[‘‘x’’,‘‘y’’,‘‘z’’],fromList [],fromList [],fromList []],[fromList
[],fromList [],fromList [],fromList [],fromList [],fromList []]]
```

Figure 6.6: Matrix M for the factorial program

Similarly, the function createLVVector was tested to ensure that the vector A is populated with the correct values. For the factorial program, we expect the vector A to be shown as below:

$$\begin{pmatrix} \{x,y\} \\ \{x,y,z\} \\ \{x,y,z\} \\ \{x,z\} \\ \{x,y,z\} \\ \emptyset \end{pmatrix}$$

The result of the test was as shown in Figure 6.7. This is the internal representation of the vector and therefore the test passed.

```
[[fromList [``x'',``y'']],[fromList [``x'',``y'',``z'']],[fromList
     [``x'',``y'',``z'']],[fromList [``x'',``z'']],[fromList
          [``x'',``y'',``z'']],[fromList []]]
```

Figure 6.7: Vector A for the factorial program

Of course, like all other functions, it was not just the factorial program that was tested in this way. Many programs were considered during testing to cover a variety of cases covering all different code paths.

6.1.5 Reaching Definitions Organiser Testing

In a similar manner to testing Live Variables Analysis, there were several things to test including:

- the calculation of the GEN sets
- the calculation of the $\overline{KILL}$ sets
- the creation of matrix M
- the creation of vector A

For Reaching Definitions Analysis, we also needed to test the calculation of the slightly different nested structure used. Testing of the function calculateRDCfg was extremely similar to how calculateCfg was tested in the general organiser, since the structure should be the same but with "Assign" statements replaced with the variable being assigned. For the factorial example, we expect the output to be the nested structure as shown in Figure 6.8.

```
List [Statement ``y'',Statement ``z'',List [Statement ``While'',List
        [Statement ``z'',Statement ``y'']],Statement ``y'']
```

Figure 6.8: Vector A for the factorial program

Since this is just a simple example, more complicated and longer programs were tested successfully.

Next, we are concerned with testing the function calculateRDGens which uses the result of calculateRDCfg. Since this function will only ever be used following the function calculateRDCfg,

we can assume only valid inputs, as stages before will identify any invalid programs. For the factorial example, we expect the GEN sets to be as shown in Table 6.3. Within the program, this is represented as a [Set (**String**, **Int**)] where the i^{th} element of the list is the GEN set for node number i. The result, as expected, of the test is as shown in Figure 6.9 and therefore the test passed.

Table 6.3: GEN sets for the factorial program

Node	GEN
1	(y, 1)
2	(z, 2)
3	$\emptyset$
4	(z, 4)
5	(y, 5)
6	(y, 6)

```
[fromList [(‘‘y’’,1)],fromList [(‘‘z’’,2)],fromList [],fromList
    [(‘‘z’’,4)],fromList [(‘‘y’’,5)],fromList [(‘‘y’’,6)]]
```

Figure 6.9: GEN sets for factorial program - test result

In order to ensure that all of the different cases are tested, a variety of tests were carried out with all different types of arithmetic and Boolean expressions as part of them.

In order to calculate the $\overline{KILL}$ sets a set of all the definitions in the program is needed. We test getAllDefsInProgram in order to ensure that this set is correct and complete. The function takes the GEN sets as the argument. For the factorial program the set of all variables in the program is $\{(y,0),(y,1),(y,5)(y,6)(z,0),(z,2),(z,4)\}$ therefore the result `fromList [(‘‘y’’,0),(‘‘y’’,1),(‘‘y’’,5),(‘‘y’’,6),(‘‘z’’,0),(‘‘z’’,2),(‘‘z’’,4)]` means the test passes. Again, we run more tests on a variety of programs ensuring that the function computes the correct set for a wide range of structures of programs.

The function calculateRDKills is similar to calculateRDGens. Therefore we test the function calculateKills in the same way as calculateGens. For the factorial program, we expect the $\overline{KILL}$ sets to be as shown in Table 6.4. Within the program, this is represented as a [Set **String**] where the i^{th} element of the list is the $\overline{KILL}$ set for node number i. The result, as expected, of the test is as shown in Figure 6.10 and therefore the test passed.

Table 6.4: $\overline{KILL}$ sets for the factorial program

Node	$\overline{KILL}$
1	(z, 0), (z, 2), (z, 4)
2	(y, 0), (y, 1), (y, 5), (y, 6)
3	(y, 0), (y, 1), (y, 5), (y, 6), (z, 0), (z, 2), (z, 4)
4	(y, 0), (y, 1), (y, 5), (y, 6)
5	(z, 0), (z, 2), (z, 4)
6	(z, 0), (z, 2), (z, 4)

```
[fromList [(‘‘z’’,0),(‘‘z’’,2),(‘‘z’’,4)], fromList
[(‘‘y’’,0),(‘‘y’’,1),(‘‘y’’,5),(‘‘y’’,6)], fromList
[(‘‘y’’,0),(‘‘y’’,1),(‘‘y’’,5),(‘‘y’’,6),(‘‘z’’,0),(‘‘z’’,2),(‘‘z’’,4)],
fromList [(‘‘y’’,0),(‘‘y’’,1),(‘‘y’’,5),(‘‘y’’,6)], fromList
[(‘‘z’’,0),(‘‘z’’,2),(‘‘z’’,4)], fromList [(‘‘z’’,0),(‘‘z’’,2),(‘‘z’’,4)]]
```

Figure 6.10: $\overline{KILL}$ sets for factorial program - test result

Using the generation of the GEN and $\overline{KILL}$ sets we create the matrix M and vector A. For this reason, we must test the functions used to do this, ensuring that the creation is correct. Since the functions rely on the creation of the adjacency matrix and the created GEN and $\overline{KILL}$ sets that have already been tested, we are simply checking that the values in the transpose adjacency matrix are replaced with the correct sets.

The function createRDMatrixM was tested to ensure that the matrix M is populated with the correct values. For the factorial program, we expect the matrix M to be as shown below:

$$\begin{pmatrix} \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ SetA & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & SetB & \emptyset & \emptyset & SetA & \emptyset \\ \emptyset & \emptyset & SetC & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & SetB & \emptyset & \emptyset \\ \emptyset & \emptyset & SetC & \emptyset & \emptyset & \emptyset \end{pmatrix}$$

where $SetA = \{$(z, 0), (z, 2), (z, 4)$\}$, $SetB = \{$(y, 0), (y, 1), (y, 5), (y, 6)$\}$ and $SetC = \{$(y, 0), (y, 1), (y, 5), (y, 6), (z, 0), (z, 2), (z, 4)$\}$.

The result of the test was as shown in Figure 6.11. This is the internal representation of the expected matrix and therefore the test passed.

```
[[fromList [],fromList [],fromList [],fromList [],fromList [],fromList
  []], [fromList [(‘‘z’’,0),(‘‘z’’,2),(‘‘z’’,4)],fromList [],fromList
    [],fromList [],fromList [],fromList []], [fromList [],fromList
[(‘‘y’’,0),(‘‘y’’,1),(‘‘y’’,5),(‘‘y’’,6)],fromList [],fromList [],fromList
  [(‘‘z’’,0),(‘‘z’’,2),(‘‘z’’,4)],fromList []], [fromList [],fromList
                                  [],fromList
 [(‘‘y’’,0),(‘‘y’’,1),(‘‘y’’,5),(‘‘y’’,6),(‘‘z’’,0),(‘‘z’’,2),(‘‘z’’,4)],
 fromList [],fromList [],fromList []], [fromList [],fromList [],fromList
[],fromList [(‘‘y’’,0),(‘‘y’’,1),(‘‘y’’,5),(‘‘y’’,6)],fromList [],fromList
                  []], [fromList [],fromList [],fromList
 [(‘‘y’’,0),(‘‘y’’,1),(‘‘y’’,5),(‘‘y’’,6),(‘‘z’’,0),(‘‘z’’,2),(‘‘z’’,4)],
                  fromList [],fromList [],fromList []]]
```

Figure 6.11: Matrix M for the factorial program

Similarly, the function createLVVector was tested to ensure that the vector A is populated with the correct values. For the factorial program, we expect the vector A to be shown as below:

$$\begin{pmatrix} \{(y,0),(z,0)\} \\ \{(y,1)\} \\ \{(y,5),(z,2)\} \\ \emptyset \\ \{(z,4)\} \\ \emptyset \end{pmatrix}$$

The result of the test was as shown in Figure 6.12. This is the internal representation of the vector and therefore the test passed.

```
    [[fromList [(‘‘y’’,0),(‘‘z’’,0)]],[fromList [(‘‘y’’,1)]],[fromList
[(‘‘y’’,5),(‘‘z’’,2)]],[fromList []],[fromList [(‘‘z’’,4)]],[fromList []]]
```

Figure 6.12: Vector A for the factorial program

Of course, like all other functions, it was not just the factorial program that was tested in this way. Many programs were considered during testing to cover a variety of cases covering all different code paths.

6.1.6 Semiring Testing

In order to test the semiring code, closure was performed on different types of matrices for each analysis. By doing this, we tested the Semiring (Matrix a) instance as well as the Semiring **Bool** instance for Reachability, the Semiring (Set **String**) instance for Live Variables and the Semiring (Set (**String**, **Int**)) instance for Reaching Definitions.

As a simple example of a test we performed closure on the adjacency matrix of the factorial program; thereby computing the solution to reachability. This test ensures that the Boolean semiring as well as the matrix semiring are implemented correctly and result in expected output. We provide the function closure the adjacency matrix, of type Matrix, as show in Figure 6.13.

```
Matrix [[False,True,False,False,False,False],
[False,False,True,False,False,False], [False,False,False,True,False,True],
[False,False,False,False,True,False],
[False,False,True,False,False,False],
[False,False,False,False,False,False]]
```

Figure 6.13: Adjacency matrix for the factorial program

The result of performing closure is shown in Figure 6.14. It is possible to see that this test passes. It is the matrix we expect as the correct solution as described in the Design chapter.

```
Matrix [[True,True,True,True,True,True], [False,True,True,True,True,True],
  [False,False,True,True,True,True], [False,False,True,True,True,True],
 [False,False,True,True,True,True], [False,False,False,False,False,True]]
```

Figure 6.14: Closure of the adjacency matrix for the factorial program

Further tests of a similar nature were performed in order to test that the other semiring instances were implemented correctly.

6.2 Program Level Testing

At a higher level, we test the whole program. This involves providing the system with a .txt file containing a program and ensuring that the output result is correct. In order to ensure correctness, we can rely on examples of analyses from literature that has been examined as well as carrying out some analyses on paper and comparing the manually calculated solution with the system solution.

Since the unit level testing has ensured that each individual function operates as expected, at the program level we test that composition of these functions results in the correct output. At this level we are testing the three main functions, one for each analysis provided by the system. These are the reachability , liveVariables and reachingDefinitions functions which each take as an argument a .txt file containing the program code and then return the analysis solution.

6.2.1 Reachability

The reachability function composes certain functions so to compute the closure of the adjacency matrix. For the factorial program, the expected output is the matrix:

$$\begin{pmatrix} True & True & True & True & True & True \\ False & True & True & True & True & True \\ False & False & True & True & True & True \\ False & False & True & True & True & True \\ False & False & True & True & True & True \\ False & False & False & False & False & False \end{pmatrix}$$

The result of the test was as shown in Figure 6.15. This is the program representation of the matrix solution and therefore the test passed.

```
Matrix [[True,True,True,True,True,True], [False,True,True,True,True,True],
  [False,False,True,True,True,True], [False,False,True,True,True,True],
 [False,False,True,True,True,True], [False,False,False,False,False,True]]
```

Figure 6.15: Solution to factorial Reachability Analysis - test result

6.2.2 Live Variables

The liveVariables function composes certain functions so to compute the closure of matrix M multiplied by vector A. For the factorial program, the expected output is as shown in Table 6.5.

Table 6.5: Factorial Live Variables Analysis Solution

Node	Live Variables
1	{y}
2	{y, z}
3	{y, z}
4	{y, z}
5	{y, z}
6	$\emptyset$

The result of the test was as shown in Figure 6.16. This is the program representation of the matrix solution and therefore the test passed.

```
Matrix [[fromList [``y'']],[fromList [``y'',``z'']],[fromList
[``y'',``z'']],[fromList [``y'',``z'']],[fromList [``y'',``z'']],[fromList
[]]]
```

Figure 6.16: Solution to factorial Live Variables Analysis - test result

We can also rely on the Live Variables Analysis example provided in the literature. In 'Principles of Programming' Nielson et al. (1999) there is an example given by the program below.

Listing 6.3: Example Live Variables Analysis Code

```
x:=2;
y:=4;
x:=1;
if (y>x) then
        (z:=y)
else
        (z:=y*y);
x:=z
```

For the 'Principles of Programming' Nielson et al. (1999) program, the expected output is as shown in Table 6.6.

Table 6.6: 'Principles of Programming' Live Variables Analysis Solution

Node	Live Variables
1	∅
2	{y}
3	{x, y}
4	{y}
5	{z}
6	{z}
7	∅

The result of the test was as shown in Figure 6.17. This is the program representation of the matrix solution and therefore the test passed.

```
Matrix [[fromList []],[fromList [``y'']],[fromList
[``x'',``y'']],[fromList [``y'']],[fromList [``z'']],[fromList
[``z'']],[fromList []]]
```

Figure 6.17: Solution to 'Principles of Programming' Live Variables Analysis - test result

6.2.3 Reaching Definitions

The reachingDefinitions function composes certain functions so to compute the closure of matrix M multiplied by vector A. For the factorial program, the expected output is as shown in Table 6.7.

Table 6.7: Factorial Reaching Definitions Analysis Solution

Node	Reaching Definitions
1	{(y, 0), (z, 0)}
2	{(y, 1), (z, 0)}
3	{(y,1), (y, 5), (z, 2), (z, 4)}
4	{(y,1), (y, 5), (z, 2), (z, 4)}
5	{(y,1), (y, 5), (z, 4)}
6	{(y,1), (y, 5), (z, 2), (z, 4)}

The result of the test was as shown in Figure 6.18. This is the program representation of the matrix solution and therefore the test passed.

```
Matrix [[fromList [(‘‘y’’,0),(‘‘z’’,0)]],[fromList
          [(‘‘y’’,1),(‘‘z’’,0)]],[fromList
[(‘‘y’’,1),(‘‘y’’,5),(‘‘z’’,2),(‘‘z’’,4)]],[fromList
[(‘‘y’’,1),(‘‘y’’,5),(‘‘z’’,2),(‘‘z’’,4)]],[fromList
     [(‘‘y’’,1),(‘‘y’’,5),(‘‘z’’,4)]],[fromList
    [(‘‘y’’,1),(‘‘y’’,5),(‘‘z’’,2),(‘‘z’’,4)]]]
```

Figure 6.18: Solution to factorial Reaching Definitions Analysis - test result

We can also rely on the Reaching Definitions Analysis example provided in the literature. In 'Principles of Programming' Nielson et al. (1999) there is an example given by the program below.

Listing 6.4: Example Live Variables Analysis Code

```
x:=5;
y:=1;
while (x>1) do
        (y:=x*y;
        x:=x-1)
end
```

For the 'Principles of Programming' Nielson et al. (1999) program, the expected output is as shown in Table 6.8.

Table 6.8: 'Principles of Programming' Reaching Definitions Analysis Solution

Node	Reaching Definitions
1	{(x, 0), (y, 0)}
2	{(x, 1), (y, 0)}
3	{(x, 1), (x, 5), (y, 2), (y, 4)}
4	{(x, 1), (x, 5), (y, 2), (y, 4)}
5	{(x, 1), (x, 5), (y, 4)}

The result of the test was as shown in Figure 6.19. This is the program representation of the matrix solution and therefore the test passed.

```
Matrix [[fromList [(``x'',0),(``y'',0)]],[fromList
          [(``x'',1),(``y'',0)]],[fromList
[(``x'',1),(``x'',5),(``y'',2),(``y'',4)]],[fromList
[(``x'',1),(``x'',5),(``y'',2),(``y'',4)]],[fromList
          [(``x'',1),(``x'',5),(``y'',4)]]]
```

Figure 6.19: Solution to 'Principles of Programming' Reaching Definitions Analysis - test result

Like all other tests, these are just a small selection examples of the tests that were carried out. In addition to these tests we tested with more complicated program structures.

If we provide an invalid program as input, since the first function used is parseFile, we get an error message such as the one displayed when testing the function.

6.3 Testing Challenges

Of course, not all of these tests passed first time round. The idea of the testing process is to identify bugs in the code, to fix the bugs and to re-test resulting in a final set of tests that all pass, as shown above. During this testing process, we encountered cases that we may not have considered initially, logical errors as well as simple mistakes. By testing we identified these issues so that we could ensure they were all removed for the final product.

One of the most complex parts to test was the composition of the adjacency matrix for a program. Since the adjacency matrix is core to all of the analyses it is the most relied on part of the implementation and there are an infinite number of programs that could be input to the program meaning that all sorts of combinations needed to be tested. For this reason, we tested with many combinations of nested structures in order to ensure all cases were accounted for. Many of the adjacency matrices for programs with nested structures were not correct on the first test. This meant the code for each individual case had to be identified and changed. Usually, this was something simple, such as adjusting a numerical value passed as an argument, and

in some cases it meant an extra function, for example a new function representing a case that had not been considered initially. Once the fix for a certain test had been made, all previously passed tests were re-tested in order to ensure that they still passed.

6.3.1 Example Bug Identified

We shall now look at an actual bug that was identified during the testing of the program. In this situation, it was due to a case that had not been considered during the implementation of the system and, since the case had not been accounted for, the implementation did not provide the adjacency matrix as expected. The program in question has the source code below and the corresponding control-flow graph as shown in Figure 6.20.

Listing 6.5: Example Program Causing Bug in Code

```
y:=x;
z:=1;
while (y>1) do
        (z:= z*y;
        y:=y-1;
        while (y>1) do
        (z:= z*y;
        y:=y-1)
end;
z := z*y)
end;
y:=0
```

The structure of this program is an enclosing 'while' loop, containing some assign statements followed by a nested 'while' loop followed by another assign statement. A test similar to this, considering a structure with a 'while' loop containing some assign statements followed by a nested 'while' loop but without extra assign statements after the nested 'while' loop passed. The situation that had not been considered was a nested 'while' loop which was not at the end of the enclosing 'while' loop. The expected result was as shown in Figure 6.22 - but the actual result on the first run of the test was as shown in Figure 6.21. The problem is in row 6 of the matrix, where the fail result says there is flow of control from the nested Boolean test to the enclosing Boolean test. As can be seen in Figure 6.20 this is not where the flow of control is, it should be to the statement following the nested 'while' loop.

Ultimately, the test passed following some fixes in the code. The fix that was required was, when creating the matrix row for a 'while' Boolean test, to determine whether there are any extra statements after the end of the nested 'while' loop. With this consideration in place, the code produced the expected output, as shown in Figure 6.22.

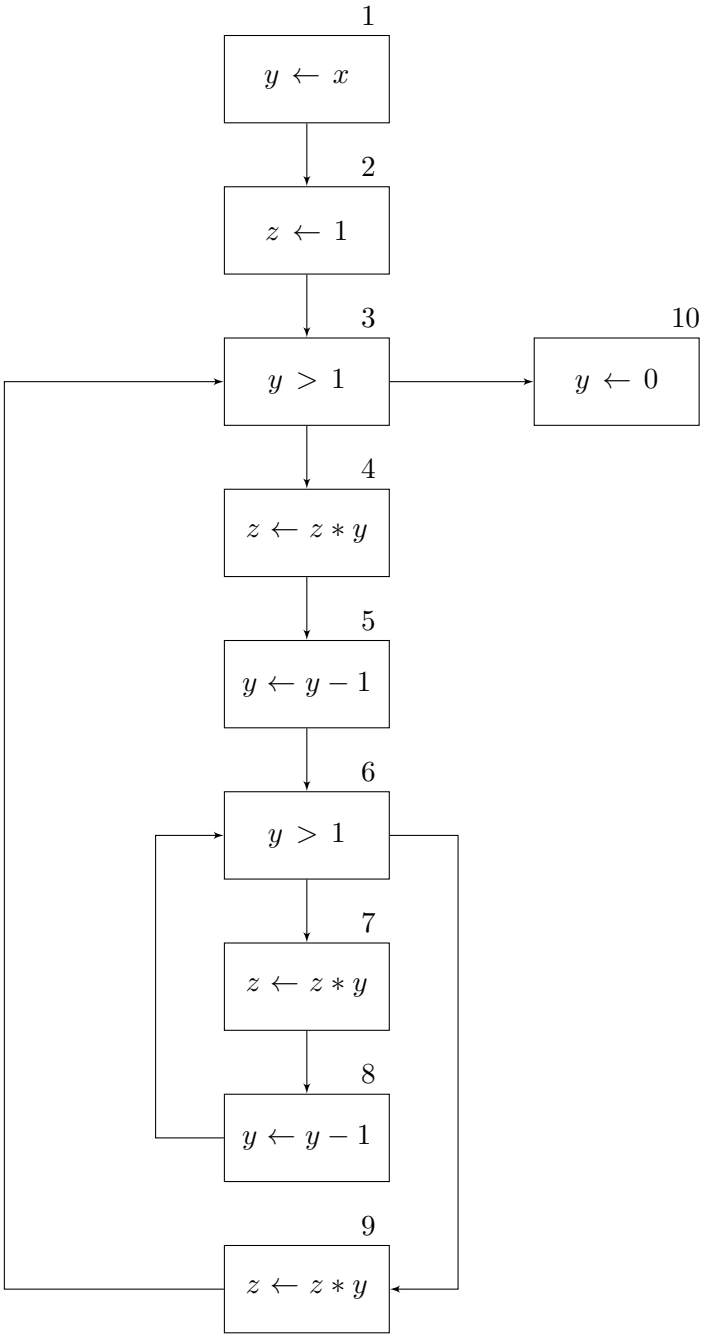

Figure 6.20: Control-flow graph for the Example Bug

```
[[False,True,False,False,False,False,False,False,False,False],
 [False,False,True,False,False,False,False,False,False,False],
  [False,False,False,True,False,False,False,False,False,True],
 [False,False,False,False,True,False,False,False,False,False],
 [False,False,False,False,False,True,False,False,False,False],
  [False,False,True,False,False,False,True,False,False,False],
 [False,False,False,False,False,False,False,True,False,False],
 [False,False,False,False,False,True,False,False,False,False],
  [False,False,True,False,False,False,False,False,False,False],
[False,False,False,False,False,False,False,False,False,False]]
```

Figure 6.21: Example bug - fail result

```
[[False,True,False,False,False,False,False,False,False,False],
 [False,False,True,False,False,False,False,False,False,False],
  [False,False,False,True,False,False,False,False,False,True],
 [False,False,False,False,True,False,False,False,False,False],
 [False,False,False,False,False,True,False,False,False,False],
  [False,False,False,False,False,False,True,False,True,False],
 [False,False,False,False,False,False,False,True,False,False],
 [False,False,False,False,False,True,False,False,False,False],
  [False,False,True,False,False,False,False,False,False,False],
[False,False,False,False,False,False,False,False,False,False]]
```

Figure 6.22: Example bug - pass result

6.4 Requirement Fulfilment

With testing and bug fixing complete, we shall now examine each requirement as set out at the start of the project in order to evaluate success of the system. Using the tests carried out we can verify the system. We mainly focus on program level testing, since requirements specify what a system should do and not how it should be done. At the program level we tested the output of the system given a certain input and this is the high-level that the requirements specify.

6.4.1 Functional Requirements

Requirement (1) is fulfilled as demonstrated by the program level testing. At program level, we provide the system with a .txt file containing source code. This source code is accepted by the program (assuming that it is a valid program in the syntax) and the solution is computed. Part (a) and part (b) of the requirement are fulfilled since the programs accepted are described by the abstract syntax in part (a) as well as the program accepting input from a .txt file rather

than via command-line input.

Requirement (2) is fulfilled as demonstrated by the program level testing. At program level, we use one of three high-level functions. Each of these functions specifies a different analysis, therefore the is a simple way for the user to specify the analysis to be performed.

Requirement (3) is fulfilled as demonstrated by the program level testing. Part (a), (b) and (c) of this requirement are fulfilled since the user can specify one of the three functions reachability, liveVariables and reachingDefinitions and the system produces the correct output as shown in the tests for each of these functions. For part (d) of this requirement, please see the next chapter (Energy Experimentation) where further work has been carried out.

Requirement (4) is fulfilled as it is possible to see that different analyses can be performed with slight adaptations or extensions on the first analysis implemented, Reachability Analysis. Part (a) of the requirement is met because each analysis relies on the adjacency matrix computed which is the same core theory and which could be used for further analyses. Please see the next chapter (Energy Experimentation) where further work has been carried out, extending on this core implementation too.

Requirement (5) is mostly met. As can be seen by the program level testing, the result of the solution is represented in the form of a matrix. Part (a) of the requirement is met since the output is displayed via the command line, however, part (b) has not been met since the output is not sent to a .txt file. This is something that could perhaps be implemented in the future, but was not a high priority requirement and other work was prioritised as more important and/or interesting than this.

6.4.2 Non-Functional Requirements

All testing was performed on a standard Windows machine by running the program via GHCi command line. For this reason, requirements (1) and (2) are successfully met. In theory, the system should run via GHCi command line on any machine.

Requirement (3) is fulfilled because the analysis results for each program level test carried out are the same as those computed manually. For some of the examples provided we relied on given examples as set out in the literature and for other tests the solutions were computed manually using the same methods.

The majority of programs tested were small programs. These include programs with a small number of nodes in the control-flow diagram - for example programs with fewer than 20 nodes. The system easily handled these small programs and therefore requirement (4) was met. Furthermore, each of these tests computed within one second, meaning that requirement (6) also passed.

Requirement (5) specified that “the system should be able to handle programs of reasonable size, that is programs with up to 100 nodes”. A large program was tested as part of the test

suite and the program was able to handle it, it just meant that computation time (especially for Reachability) was noticeably longer. For Reachability, this was on average one minute longer while for the other analyses it was on average 6 seconds longer, most likely due to closure being performed on different data types.

There was no usability testing so assessing requirement (7) is more difficult, however, given the simplicity of user interaction, i.e. only needing to input an analysis and file name, the author concludes that the system is intuitive to use. Given a user guide explaining the different analysis options, and how to input them, it would be simple for a new user to be able to use the system. In addition, since the system was produced mainly for experimental reasons rather than usability, this requirement is less important than others.

The system was implemented using Haskell as the chosen language. The reason for choosing this language was explained in the design chapter of this report. Given the success of the requirements as a whole and the implementation being in Haskell, we can conclude that requirement (8) passes.

The system is adaptable for different analyses since three analyses have been implemented. There is a core implementation used for all three analyses, the implementation of the adjacency matrix. This implementation is used again during experimentation in the next chapter (refer to Energy Experimentation). For this reason, we can determine that requirement (9) is met.

Finally, we evaluate the success of requirement (10). The system is adaptable for possible integrations with other systems since it has been developed at such a raw, low level. Since the system is just a command line tool as this stage, a graphic user interface could be built on top or some scripting could be used to pipeline it into another system such as a compiler.

6.4.3 Summary

All of the high priority requirements ("must" and "should") requirements have been met successfully. Some of the lower priority requirements ("could") have been met but not all. The only requirement that has not been implemented at all is the functional requirement (5) part (b) which says that the output could be provided via a .txt file. Since the idea of experimenting with real-life cost analysis was far more interesting, this requirement was left out. There is not a huge loss because of this as the project is not heavily focused on user experience but is more concerned with the adaptability and experimentation possible with this type of analysis system.

Chapter 7

Energy Experimentation

Following the implementation and testing of the core capabilities of the system, the idea of analysing energy within a program was briefly experimented with. This chapter looks at the adaptation of the existing work in order to provide some groundwork for an idea, work produced as part of this as well as potential developments to take this idea further.

7.1 Basic Worst Case Energy Analysis

The initial step was to create some way of analysing worst case energy within a program. In order to do this, it was necessary to come up with some way of quantifying energy as well as producing suitable equations for solving the analysis.

By using the existing analyses techniques the idea of worst case energy used, on entry to and exit from nodes, in the program was considered. The worst case energy on exit from a node is the worst case energy consumed on entry to the node plus the energy used (or generated) within that node. The semiring necessary for this analysis is almost identical to the one for longest path analysis, examined in Dolan (2013), by using the max-plus semiring. There are three possible types of values: unreachable, non-negative integer costs, and infinity. In the max-plus semiring the zero value is unreachable, the unit element is the energy cost 0 (since 0 is the identity of the semiring multiplication), addition is max and multiplication is addition. The binary operations addition (max) and multiplication (addition) both extended to include the unreachable and infinity elements as follows:

$+$	unreachable	n	infinity
unreachable	unreachable	n	infinity
n'	n'	$max(n, n')$	infinity
infinity	infinity	infinity	infinity

$\cdot$	unreachable	n	infinity
unreachable	unreachable	unreachable	unreachable
n'	unreachable	$n + n'$	infinity
infinity	unreachable	infinity	infinity

Composing the equation was a complex task. In order to rely on the core framework, the equation needed to be in the same format as the others used; we needed an equation of the form $L = ML + A$. Using the max-plus semiring, the equation produced is as follows:

$$OUT_S = \left(\sum_{S' \in pred[S]} OUT_{S'} \cdot COST_S \right) + COST_S$$

Where $COST$ is the quantified energy used in a node and OUT is the worst case amount of energy used on exit from a node. The equation is of the form $L = ML + A$ and therefore, like the other analyses, can be solved by computing $L = M^* \cdot A$. The matrix M is the transpose of the adjacency matrix, as S' is the predecessor, where a 'False' value is replaced with the 'Unreachable' value from the Energy Semiring and a 'True' value in row i is replaced with the energy cost of the node i. The vector A is just the energy cost of each node. We compute closure of matrix M using the max-plus semiring and then multiply the result by vector A. Since addition in the max-plus semiring is max, we always take the worst case value for a node and then, since multiplication is addition, we add the cost of the current node giving us the worst case energy on exit from a node.

In order to quantify energy in the program we must be able to assign energy costs. There are several different operations we can consider including:

- memory reads
- memory writes
- arithmetic operations
- boolean operations

In order to make the analysis as representative as possible, we should assign each operation some suitable cost. In Leite et al. (2014) the cost of different operations is calculated, so we can use this to come up with some relative cost for each operation. Since this is hard coded in the program, if we do not care about some operation or operations then we can simply assign those a 0 cost. Using these relative costs, we are able to compare different implementations to problems and so on.

In order to implement this analysis it was just a case of adapting the previously implemented analyses, again displaying the adaptability and using the core theory of the adjacency matrix. In the general organiser, in addition to the creation of the nested structure representing the control-flow diagram, another nested structure was also implemented containing a pair for each node instead of a simple string representing the type of statement. The pair contains two values, the type of statement as well as the amount of energy used in that node. This is computed by considering the number of memory reads, memory writes, arithmetic operations and boolean operations. When creating the $COST$ vector, for each node we just take the second element in this pair. The adjacency matrix (transposed) and the $COST$ vector are all that is necessary to create the matrix M and vector A for the analysis and this is done in a similar way to the

previous analyses.

The energy value meant that a new data structure and semiring needed to be implemented. For this reason, we define the Energy data structure as:

```
data Energy = Energy Int | Unreachable | Infinite deriving (Show, Eq)
```

where the Energy value represents the cost. The value Unreachable, the zero element, is used in the creation of matrix M when there is no flow of control between nodes and Infinite is used in the Energy semiring instance when there is worst case infinite energy. The Energy semiring instance is defined as follows:

```
instance Semiring Energy where
    zero       = Unreachable
    one        = Energy 0
    sAdd x Unreachable = x
    sAdd Unreachable x = x
    sAdd Infinite _ = Infinite
    sAdd _ Infinite = Infinite
    sAdd (Energy x) (Energy y) = Energy (max x y)
    sMult _ Unreachable = Unreachable
    sMult Unreachable _ = Unreachable
    sMult Infinite _ = Infinite
    sMult _ Infinite = Infinite
    sMult (Energy x) (Energy y) = Energy (x + y)
    closure Unreachable = Energy 0
    closure (Energy 0) = Energy 0
    closure _ = Infinite
```

which is almost identical to the longest path semiring defined in Dolan (2013), since the analysis is similar.

7.1.1 Analysis Results

In order to test the implementation, we provided hard coded test values for memory reads, memory writes, arithmetic operations and boolean operations. In real-life, these should be representative of the relative costs of each operation as can be identified as described in Leite et al. (2014). For testing we used the following values:

```
-- Test values
memWrite = 4;
memRead = 3;
bOp = 1;
aOp = 2;
```

If we consider the factorial example, the values for generating costs per node is as shown in Figure 7.1.

```
[Energy 3,Energy 4,Energy 4,Energy 12,Energy 9,Energy 4]
```

Figure 7.1: Energy cost per node in factorial program

If we run the analysis we get the result as shown in Figure 7.2. This is the expected result, with infinite energy as soon as the 'while' statement is reached, but identifies the main limitation of the analysis.

```
Matrix [[Energy 3],[Energy 7],[Infinite],[Infinite],[Infinite],[Infinite]]
```

Figure 7.2: Worst case energy cost in factorial program

As another example, we consider the following program, which does not contain a 'while' statement:

```
x:=2;
y:=4;
x:=1;
if (y>x) then
        (z:=y)
else
        (z:=y*y);
x:=z
```

If we run the worst case energy analysis on this program we get the result as shown in Figure 7.3. This is the expected result and, as there is no 'while' statement, does not result in infinite energy as a worst case. This example displays when the analysis is more useful.

```
Matrix [[Energy 4],[Energy 8],[Energy 12],[Energy 19],[Energy 26],[Energy
31],[Energy 38]]
```

Figure 7.3: Worst case energy cost in factorial program

7.1.2 Limitation

The one fundamental limitation of this analysis is that as soon as you reach a 'while' loop the worst case energy is always infinity. This is a major limitation since as soon as there is a 'while' loop in the source code the analysis becomes almost useless. For this reason, it was necessary to consider ways in which the analysis could be improved so to provide actual useful information. With the worst case energy implementation as a working foundation we progressed to consider further ideas.

7.2 Cached vs. Non-Cached Variables

The next idea considered was cached vs. non-cached variables. There exists work in this domain, for example Li et al. (1995) and Stappert & Altenbernd (2000), but these resources focus on worst case execution time (WCET), rather than energy costs, and are less concerned with a general theoretical approach based on the framework we have already provided as the core of this project. Furthermore, Stappert & Altenbernd (2000) only considers 'straight-line' programs, that is programs without loops, and therefore does not address the limitation that we are attempting to face here.

If we are able to identify when variables are cached or not, and we consider cached variables as 'free' to read or write since their cost may be significantly smaller than non-cached variables, then some 'while' loops could be considered 'free' (or close to) if all the variables read/written are in the cache. This could be particularly useful when we consider systems that, for example, have to access remote destinations in order to access data. Devices are becoming smaller, such as wearable technology, which results in less onboard memory. This means we rely on remote data access more and more which not only takes more time but also a lot more energy. There is even a difference when we compare the energy necessary for a cached access rather than an access from main memory, but the necessity for remote access makes the need for this sort of analysis even more apparent.

The analysis implemented is similar to Reaching Definitions Analysis and the solution tells us whether, in each node, all the variables that could be used are cached or not. There are actually two functions for this analysis:

- cachedOrNotPerVar is effectively the same as Reaching Definitions Analysis except this time we append extra information to each pair, resulting in a triple of the form (Variable, Node Number Defined, Cached or Not)
- cachedOrNotAll which results in one value per node, summarizing the results of cachedOrNotPerVar, specifying whether all the variables are cached or not

In order to implement this, a new type of assignment statement was added to the language meaning that there were two types: the standard assignment statement as in the original language and a cached assignment statement meaning that the variable being assigned is cached. This meant that everywhere that an "Assign" is identified in the code, we also needed to add cases for the "CachedAssign" too. The concrete syntax for a cached assign is similar to the standard assign, but with the new reserved word "cache" before it. For example, if we are assigning x to y the standard assign statement would be x:=y; but the cached assign statement would be cache x:=y;.

With this in place, we could use the solution to Reaching Definitions Analysis as well as the nested structure representing the control-flow graph to perform the analysis. First we get the result of Reaching Definitions and then, for each variable, we use the control-flow graph to identify whether at the label it was defined it was cached or not.

7.2.1 Analysis Results

In order to test the implementation we initially consider the factorial program as it has been throughout the entire testing so far, with no cached assign statements. Of course, for this program we expect the resulting output to be of all variables not cached. The result of the function cachedOrNotPerVar is as shown in Figure 7.4 and the result of function cachedOrNotAll is as shown in Figure 7.5. These tests resulted in the expected results and therefore passed.

```
[[fromList [(‘‘y’’,0,‘‘Undefined’’),(‘‘z’’,0,‘‘Undefined’’)]], [fromList
       [(‘‘y’’,1,‘‘Assign’’),(‘‘z’’,0,‘‘Undefined’’)]], [fromList
              [(‘‘y’’,1,‘‘Assign’’),(‘‘y’’,5,‘‘Assign’’),
         (‘‘z’’,2,‘‘Assign’’),(‘‘z’’,4,‘‘Assign’’)]], [fromList
              [(‘‘y’’,1,‘‘Assign’’),(‘‘y’’,5,‘‘Assign’’),
         (‘‘z’’,2,‘‘Assign’’),(‘‘z’’,4,‘‘Assign’’)]], [fromList
  [(‘‘y’’,1,‘‘Assign’’),(‘‘y’’,5,‘‘Assign’’), (‘‘z’’,4,‘‘Assign’’)]],
        [fromList [(‘‘y’’,1,‘‘Assign’’),(‘‘y’’,5,‘‘Assign’’),
            (‘‘z’’,2,‘‘Assign’’),(‘‘z’’,4,‘‘Assign’’)]]]
```

Figure 7.4: Cached vs. Non-Cached per variable in factorial program

```
       [[‘‘ll Undefined/Cached’’],[‘‘Not All Cached’’],[‘‘Not All
Cached’’],[‘‘Not All Cached’’],[‘‘Not All Cached’’],[‘‘Not All Cached’’]]
```

Figure 7.5: Cached vs. Non-Cached all in factorial program

Following the testing of the factorial program with no cached variables, we could consider the program where every variable is cached. For this reason, we changed all of the assign statements in the program to cached assign statements, as follows:

```
cache y:=x;
cache z:=1;
while (y>1) do
       (cache z:= z*y;
       cache y:=y-1)
end;
cache y:=0
```

In this case, we expect the results output to be for all variables cached. The result of the function cachedOrNotPerVar is as shown in Figure 7.6 and the result of function cachedOrNotAll is as shown in Figure 7.7. These tests resulted in the expected results and therefore passed.

```
[[fromList [(``y'',0,``Undefined''),(``z'',0,``Undefined'')]], [fromList
    [(``y'',1,``CachedAssign''),(``z'',0,``Undefined'')]], [fromList
        [(``y'',1,``CachedAssign''),(``y'',5,``CachedAssign''),
   (``z'',2,``CachedAssign''),(``z'',4,``CachedAssign'')]], [fromList
        [(``y'',1,``CachedAssign''),(``y'',5,``CachedAssign''),
   (``z'',2,``CachedAssign''),(``z'',4,``CachedAssign'')]], [fromList
        [(``y'',1,``CachedAssign''),(``y'',5,``CachedAssign''),
                (``z'',4,``CachedAssign'')]], [fromList
        [(``y'',1,``CachedAssign''),(``y'',5,``CachedAssign''),
       (``z'',2,``CachedAssign''),(``z'',4,``CachedAssign'')]]]
```

Figure 7.6: Cached vs. Non-Cached per variable in cached factorial program

```
[[``All Undefined/Cached''],[``All Undefined/Cached''],[``All
Cached''],[``All Cached''],[``All Cached''],[``All Cached'']]
```

Figure 7.7: Cached vs. Non-Cached all in cached factorial program

Following the testing of the factorial program with all cached variables, we could consider the program where there is a combination of cached and non-cached variables. We test the following program:

```
cache y:=x;
cache z:=1;
while (y>1) do
        (cache z:= z*y;
        y:=y-1)
end;
cache y:=0
```

In this example the y inside the loop is not cached so we expect the results to differ. The result of the function cachedOrNotPerVar is as shown in Figure 7.8 and the result of function cachedOrNotAll is as shown in Figure 7.9. These tests resulted in the expected results and therefore passed.

```
[[fromList [(‘‘y’’,0,‘‘Undefined’’),(‘‘z’’,0,‘‘Undefined’’)]], [fromList
    [(‘‘y’’,1,‘‘CachedAssign’’),(‘‘z’’,0,‘‘Undefined’’)]], [fromList
           [(‘‘y’’,1,‘‘CachedAssign’’),(‘‘y’’,5,‘‘Assign’’),
   (‘‘z’’,2,‘‘CachedAssign’’),(‘‘z’’,4,‘‘CachedAssign’’)]], [fromList
           [(‘‘y’’,1,‘‘CachedAssign’’),(‘‘y’’,5,‘‘Assign’’),
   (‘‘z’’,2,‘‘CachedAssign’’),(‘‘z’’,4,‘‘CachedAssign’’)]], [fromList
           [(‘‘y’’,1,‘‘CachedAssign’’),(‘‘y’’,5,‘‘Assign’’),
                (‘‘z’’,4,‘‘CachedAssign’’)]], [fromList
           [(‘‘y’’,1,‘‘CachedAssign’’),(‘‘y’’,5,‘‘Assign’’),
       (‘‘z’’,2,‘‘CachedAssign’’),(‘‘z’’,4,‘‘CachedAssign’’)]]]
```

Figure 7.8: Cached vs. Non-Cached per variable in combined factorial program

```
[[‘‘All Undefined/Cached’’],[‘‘All Undefined/Cached’’],[‘‘Not All
Cached’’],[‘‘Not All Cached’’],[‘‘Not All Cached’’],[‘‘Not All Cached’’]]
```

Figure 7.9: Cached vs. Non-Cached all in combined factorial program

Following these tests, many more were carried out in order to ensure that the expected output was produced when different combinations of assign and cached assign statements were used within programs.

7.3 Combined Worst Case Energy Cost Analysis with Cached vs. Non-Cached Variables

At the next stage of the experimental analysis we consider combining the two analyses produced so to provide more useful information. This is similar to the basic worst case energy analysis but now we use the result of the ‘cached vs. non-cached variables’ analysis in order to compose the *COST* values for each node in the control-flow graph.

In order to implement this, we need the nested control-flow graph structure to contain extra information. For each node we have a triple of the form (**String**, Set **String**, Set **String**) where the first element represents the type of statement (i.e. “Assign” or “CachedAssign” and so on), the second element represents the variables that are written in the node and the third element represent the variables that are read in the node. We can then iterate over this structure to create the *COST* values by considering the exact variables that are read and written in each node.

When computing the *COST* values we can determine whether writes are cached or not by the type of statement (“Assign” or “CachedAssign”) so it is simple to determine the energy cost of writes, however, it is more complex for the calculation of read costs. In order to compute

the read costs we use the extended Reaching Definitions Analysis for cached vs. non-cached variables and use the node number to determine whether the variables being read must be cached or not. Since we are considering worst case energy consumption, if a variable may or may not be cached we must assume the worst case. That is, if at least one of the reaching definitions of the variable in question is not cached then we assume a non-cached access.

With the *COST* vector computed, we solve the worst case energy cost analysis equation as before.

7.3.1 Analysis Results

In order to test the implementation we consider the factorial program again. Now we are only concerned with memory reads and writes; we consider arithmetic and Boolean operations free. With no cached assign statements, we expect the result to be the same as the basic worst case energy analysis, with 'free' operations other than memory reads and writes. This was the case. With all cached assign statements we expect the result at every node to be Energy 0 since we are considering all cached reads/writes as 'free'. The result is displayed in Figure 7.10 and therefore the test passed.

```
Matrix [[Energy 0],[Energy 0],[Energy 0],[Energy 0],[Energy 0],[Energy 0]]
```

Figure 7.10: Factorial program with all variables cached

A more interesting case is when we have some non-cached assign before the 'while' statement, where the non-cached variable is not used in the loop and therefore the loop is still considered 'free'. For this test we use the following adapted factorial program:

```
a:=x;
cache y:=x;
cache z:=1;
while (y>1) do
        (cache z:= z*y;
        cache y:=y-1)
end;
cache y:=0
```

For the result of this, we expect all of the nodes to have Energy 4 values since this is the cost of a non-cached memory write. Since at this node, the variable x is undefined we consider this the same as a cached variable, i.e. 'free'. The result, as expected, of this test is displayed in Figure 7.11 and therefore the test passed.

```
Matrix [[Energy 4],[Energy 4],[Energy 4],[Energy 4],[Energy 4],[Energy
4],[Energy 4]]
```

Figure 7.11: Factorial program with one variable cached not used in the loop

Again, similar to all other testing throughout the project, many more tests were carried out in order to ensure that the expected output was produced when different combinations of assign and cached assign statements were used within programs. By combining two already tested analyses there was only a small amount to test this time.

Chapter 8

Conclusions

In this chapter we take time to reflect on the entire project and the processes taken as a whole. At the beginning of the project we proposed the idea of a static program analysis tool that could be used to perform optimisation analyses. We evaluated appropriate literature within the domain so to justify such a system as well as identify open areas for development and future work. Following this we composed a set of requirements for the proposed system, followed by design and an appropriate implementation guided by this. With real-life costs identified as an area of potential research, we spent some time experimenting with the idea of analysing energy consumption within code.

With testing complete and a reflection on the success of the system provided based on the requirements set out at the start, it is important to conclude the project in its entirety.

8.1 Project Overview

Before we reflect upon the system, we remind ourselves of the inital aims of the project as well as the system that has been produced.

The inital aims of the project were to be able to understand and implement multiple optimisation analyses as part of a static analysis tool that is adaptable for many analyses. As part of this, it would be necessary to understand some core complex mathematics including, as a foundation to the theory of the system, the use of semirings to solve data-flow equations.

The system produced is a command line static analysis tool that takes, as input, the name of the analysis to be performed as well as a .txt file containing the source code to be analysed. The source code is accepted by the abstract syntax described in functional requirement (1). Upon execution, the system computes the solution to the analysis specified and displays the solution to the user via command line. The only interaction that the user needs with the system is the input at the beginning and the rest of the process is totally automated.

In summary, the system can perform Reachability, Live Variables and Reaching Definitions

Analysis. In addition, the system can perform some basic and experimental energy consumption analysis which could be used as a basis for future developments.

8.2 System Evaluation

We now evaluate the actual system that has been implemented. The testing that was performed verified that the system meets the specification so here we focus on validation which focuses on whether, by following the specification, the system actually serves a purpose. Since there is no direct user of the system, we do not focus on the system meeting user needs. Insead, we look at the contribution the work holds within the domain and how the experimental developments may be useful.

Like any project, nothing is perfect or smooth sailing all the time. Implementation can often take longer than anticipated and time constraints or deadlines can limit the scope of the project. This did occur in this case, so there was some impact on the project outcome, however the issues did not affect the project in a hugely significant way. We must also acknowledge the complexity and non-triviality of this project. Building a system, in a relatively new language to the author, with a huge amount of mathematical understanding to grasp is a huge success in itself. It is important to critique both the successes in the project and reflect on areas for improvement.

8.2.1 Successes

First we focus on the successes of the project; the ways in which the system serves its purpose. In order to do this we acknowledge the main contributions of the project. In summary, there are three main significant contributions:

- an adaptable and flexible control-flow analysis framework based on semirings
- the implementation of several previously developed analyses
- the proposal of a novel analysis of energy consumption in programs

Firstly, the system developed is a simple, adaptable and flexible framework. This means that existing analyses can be implemented using it but it can also be used to extend existing analyses and even develop new ones. Being a low-level standalone component, the system can be used on its own as is but also has the potential to be integrated into a larger system. The tool uses existing knowledge about semirings and their applications and provides a strong core for many analyses we are interested in.

Secondly, the implemented system provides three common optimisation analyses: Reachability, Live Variables and Reaching Definitions Analysis. These are analyses that are extremely useful in important and commonly used systems such as compilers and, although perhaps people don't realise it, these analyses result in optimised performance for many programs. Each of these analyses are easily accessible to any user as all they need is the function name for each

analysis and the name of the .txt file containing the source code to be analysed.

Finally, an adapted version of the tool proposed a new, novel analysis which takes into consideration real-life costs for optimisation. This is something that was difficult to comprehend initially and was a low priority requirement due to the fact that it was difficult to determine whether there would be time to experiment with such ideas in the time frame given. The fact that this has been investigated and implemented, at least as the beginning of an idea for a theoretical way of analysing real-life costs, is a huge success of the project and provides something new within the subject area.

Overall the project has been a success. It fits a purpose within the domain and its usability is perfectly adaquate in terms of its intended use. The functionality of the core analyses is complete and the adaptable framework has proved its success through implementation of a new, experimental analysis.

8.2.2 Areas for Improvement

Although the project overall has been a success, that does not mean that the system is without flaws. For this reason, we must discuss areas that could be improved, and perhaps that would have been given more time.

Hardcoded Sets for Live Variables and Reaching Definitions Semirings

For the Live Variables and Reaching Definitions semirings, the 'one' element, the unit, is the set of all variables and the set of all possible definitions within the program respectively. This means that the analyses are explicitly limited to the sets that are defined in the semirings code. This is not something that was anticipated as to be difficult to implement during the design stage but became apparent during implementation.

Since the set of all variables or definitions is reliant on the program being analysed and it is not simple to implement this in the semiring code, the easy solution taken was to hardcode values. Having excess variables or definitions in the sets does not impact the analysis so we define a set large enough to allow for reasonable sized programs. In an ideal world, this is something that should be improved upon so not to limit the programs that can be analysed by the system.

Infinite Energy Costs due to 'While' Loops

In the experiment analyses considering worst case energy costs, if we encounter a 'while' loop which uses uncached variables the worst case becomes infinite. By implementing the cached variables we are able to provide more useful information by highlighting loops that could be more problematic than others, but providing more useful information does not remove the limitation.

In addition to considering cached vs. non-cached variables, another way to make the worst case

energy cost analysis more useful would be to compute any possible upper bounds for 'while' loops. If an upper bound can be computed then a 'while' loop would not result in infinite energy cost but the cost of one iteration multiplied by the upper bound, making the analysis far more useful. This was not implemented as part of this project because the mathematics behind it would make it substantially more complex and time consuming, but is something that could be considered in order to improve this analysis.

8.2.3 Future Work

Having reached the end of the project, we can consider what future work may be considered. Of course the end of the project does not mean the end of research in this subject area or the development of systems similar to this tool. The areas for improvement that have been identified are obviously things that could be worked on in the future, but are more limitations of the system rather than things that may be considered beyond the scope of this project.

The potential future work is outlined here:

- Extension of the simple imperative 'while' language so to deal with more complex cases and program structures.
- The development of a graphical user interface over the system to improve usability and interactivity of the system if used alone.
- The integration of the system into some larger system than can benefit from these analyses.
- The use of the framework to implement more, similar analyses.
- The use of the experimental energy consumption analysis to develop more or more useful theoretical analyses incorporating real-life costs.

8.2.4 Conclusion

Although there are identifed areas for improvement as well as future work, the system has met most of the requirements and therefore has been a success. The scope of the project was limited at the beginning so that the amount of work that was expected to be achieved was realistic. The experimental focus of the project was prioritised over the usability of the system since what is interesting about this system is the way in which is can be adapted and extended. The fact that we were able to experiment with the idea of analysing real-life costs, on top of a system that had been verified as robust and adaptable, was the major success of this project and therefore the compromise of having small limitations of the system are considered acceptable.

8.3 Personal Evaluation

Having evaluated the system itself, we now turn to how project went and lessons that were learnt along the way.

8.3.1 Program analysis is complex

The most difficult part of the project was the very first part. Once all of the relevant literature had been gathered, taking the time to read, and more importantly understand, all of the concepts and theory was the most challenging task. For an undergraduate in a final year, the complexity of the mathematics was a huge step up. With a system that relied on implementing this theory, it was vital to understand it before implementing. Of course, concepts became more apparent during the actual implementation stages, but there would have been no way of starting without a solid grasp of the subject area initially. The literature review was of significant importance in understanding the topics and consolidating the knowledge aquired from multiple sources.

8.3.2 Reusing existing code is beneficial

The code for the parser was the major part of the code that was reused from an existing source *Parsing a simple imperative language* (n.d.). Time was taken to understand the workings of the code so it could be adapted, especially when adding in the new statement for cached assign statements, but having some readily available and explained code was hugely beneficial to the project. Even being able to use Parsec parser combinator library in itself was a huge help.

Had this existing work not been available, a much more significant amount of time would have had to have been spent implementing the parser section of the code and would have most likely significantly reduced the amount of time being left to the experimental section of the project. Having carried out the experimental section of the project and seeing the results it definitely would have been a shame if this work had not been carried out.

8.3.3 Thinking time is valuable time

Before jumping into implementation it is worth spending time thinking. Spending time thinking is not time wasted on implementation, in fact it often reduces the amount of time spent overall. Particularly with a theoretically complex project, it is important to think about what actually needs to be done. Drawing diagrams, making notes and sketching out ideas may delay the start of the actual coding but makes the job a lot easier. Thinking allows for possible problems to be identified before they arise.

8.3.4 It is difficult to decide when to stop

When carrying out a project the scope is defined at the beginning but, even with the scope defined, it is still difficult to decide when to stop. Is anything ever perfect? How much time should be spent making something as close to perfect as possible? Of course deadlines have a huge influence over when to finally stop working on a project but during each stage it is difficult to decide when to move on. At times it was decided that progression to the next stage was more important than perfection of the previous, but making this decision is difficult. The benefits of an implemented complete system outweigh the benefits of one perfectly implemented

component.

8.4 Final Comments

In order to conclude the project, we revisit the introduction where we set out the project and identified the main aims.

The first statement to consider is:

"*This project focuses on the latter, looking at optimisation of programs, in particular by focusing on data-flow within programs*"

In particular:

"*This project focuses on a formal method of analysis, data-flow analysis, through the use of a mathematical structure called a semiring*"

Thus, it would be fair to conclude that the project has remained focused on the area in which is was meant to.

A futher statement identified in the introduction highlights the idea of an adaptable system, recall:

" *..by using the same core theory as the foundation to each of the analyses implemented, we provide a system that is flexible enough to adapt to different problems as well as extend to new ones.*"

This has been proved through the implementation of three core analyses reliant on the adjacency matrix and semirings, as well as the experimental analyses which are also reliant on the adjacency matrix and semirings. Therefore, we can conclude that this aim has also been successfully met.

Finally, at the end of the introduction, we recall the main aim of the project:

"*to use semirings as a foundation to implement analyses for optimisations within program code*"

The project has been centred around semirings and how, with the correct definition of a semiring with associated data-flow equations, different analyses can be implemented using the same core framework.

We can finally conclude that the project has been an overall success.

Bibliography

Abdali, S. K. & Saunders, B. D. (1985), 'Transitive closure and related semiring properties via eliminants', *Theoretical Computer Science* **40**, 257–274.

Allen, F. E. & Cocke, J. (1976), 'A program data flow analysis procedure', *Commun. ACM* **19**(3), 137–.
URL: *http://doi.acm.org/10.1145/360018.360025*

Atkins Maplas (n.d.), `http://malpas-global.com/`. Accessed: 2016-11-12.

Bergeretti, J.-F. & Carré, B. A. (1985), 'Information-flow and data-flow analysis of while-programs', *ACM Trans. Program. Lang. Syst.* **7**(1), 37–61.
URL: *http://doi.acm.org/10.1145/2363.2366*

Dolan, S. (2013), 'Fun with semirings: A functional pearl on the abuse of linear algebra', *SIGPLAN Not.* **48**(9), 101–110.
URL: *http://doi.acm.org/10.1145/2544174.2500613*

Farrow, R., Kennedy, K. & Zucconi, L. (1976), Graph grammars and global program data flow analysis, *in* 'Foundations of Computer Science, 1976., 17th Annual Symposium on', IEEE, pp. 42–56.

Fosdick, L. D. & Osterweil, L. J. (1976), 'Data flow analysis in software reliability', *ACM Comput. Surv.* **8**(3), 305–330.
URL: *http://doi.acm.org/10.1145/356674.356676*

Golan, J. S. (2013), *Semirings and their Applications*, Springer Science & Business Media.

Haskell (n.d.), `https://www.haskell.org`. Accessed: 2016-11-24.

Kam, J. B. & Ullman, J. D. (1976), 'Global data flow analysis and iterative algorithms', *J. ACM* **23**(1), 158–171.
URL: *http://doi.acm.org/10.1145/321921.321938*

Kildall, G. A. (1973), A unified approach to global program optimization, *in* 'Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages', POPL '73, ACM, New York, NY, USA, pp. 194–206.
URL: *http://doi.acm.org/10.1145/512927.512945*

Landi, W. (1992), 'Undecidability of static analysis', *ACM Lett. Program. Lang. Syst.* **1**(4), 323–337.
URL: *http://doi.acm.org/10.1145/161494.161501*

Lehmann, D. J. (1977), 'Algebraic structures for transitive closure', *Theoretical Computer Science* **4**(1), 59–76.

Leite, A., Tadonki, C., Eisenbeis, C. & De Melo, A. (2014), 'A fine-grained approach for power consumption analysis and prediction', *Procedia Computer Science* **29**, 2260–2271.

Li, Y.-T., Malik, S. & Wolfe, A. (1995), Efficient microarchitecture modeling and path analysis for real-time software, *in* 'Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE', IEEE, pp. 298–307.

Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., Lopez-Garcia, P., Grech, N., Hermenegildo, M. V. & Eder, K. (2013), Energy consumption analysis of programs based on xmos isa-level models, *in* 'International Symposium on Logic-Based Program Synthesis and Transformation', Springer, pp. 72–90.

Mohri, M. (2002), 'Semiring frameworks and algorithms for shortest-distance problems', *Journal of Automata, Languages and Combinatorics* **7**(3), 321–350.

MoSCoW : Requirements Prioritization Technique (n.d.), `https://businessanalystlearnings.com/ba-techniques/2013/3/5/moscow-technique-requirements-prioritization`. Accessed: 2017-03-10.

Navas, J., Mendez-Lojo, M. & Hermenegildo, M. V. (2008), 'Safe upper-bounds inference of energy consumption for java bytecode applications'.

Nielson, F., Nielson, H. R. & Hankin, C. (1999), *Principles of Program Analysis*, Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Parsec (n.d.), `https://wiki.haskell.org/Parsec`. Accessed: 2016-11-24.

Parsing a simple imperative language (n.d.), `https://wiki.haskell.org/Parsing_a_simple_imperative_language`. Accessed: 2016-11-24.

Ramalingam, G. (1994), 'The undecidability of aliasing', *ACM Transactions on Programming Languages and Systems (TOPLAS)* **16**(5), 1467–1471.

Schubert, S., Kostic, D., Zwaenepoel, W. & Shin, K. G. (2012), Profiling software for energy consumption, *in* 'Green Computing and Communications (GreenCom), 2012 IEEE International Conference on', IEEE, pp. 515–522.

Stappert, F. & Altenbernd, P. (2000), 'Complete worst-case execution time analysis of straight-line hard real-time programs', *Journal of Systems Architecture* **46**(4), 339–355.

Tarjan, R. E. (1981*a*), 'Fast algorithms for solving path problems', *J. ACM* **28**(3), 594–614. **URL:** *http://doi.acm.org/10.1145/322261.322273*

Tarjan, R. E. (1981*b*), 'A unified approach to path problems', *J. ACM* **28**(3), 577–593. **URL:** *http://doi.acm.org/10.1145/322261.322272*

The Parsec Package (n.d.), `https://hackage.haskell.org/package/parsec`. Accessed: 2016-11-24.

Wegbreit, B. (1975), 'Mechanical program analysis', *Commun. ACM* **18**(9), 528–539.
URL: *http://doi.acm.org/10.1145/361002.361016*

Zowghi, D. & Coulin, C. (2005), Requirements elicitation: A survey of techniques, approaches, and tools, *in* 'Engineering and managing software requirements', Springer, pp. 19–46.