SYMBOLIC INTERPRETATION OF LEGACY ASSEMBLY LANGUAGE

Symbolic Interpretation of Legacy Assembly Language

By
Pulak Kumar Chowdhury, BSc. Engg.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements for the Degree of

Master of Applied Science
Department of Computing and Software
McMaster University

MASTER OF APPLIED SCIENCE(2005) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE: Symbolic Interpretation of Legacy Assembly Language

AUTHOR: Pulak Kumar Chowdhury, BSc. Engg.(BUET)

SUPERVISOR: Dr. Jacques Carette

NUMBER OF PAGES: xi, 228

Abstract

Many industries have legacy software systems which are definitely important to them but are however, difficult to maintain due to a lack of understanding of those systems. This occurs as a result of inadequate or inconsistent documentation. Although the costs of redesigning the system may be large, some organizations still plan to reverse engineer the software specification documents from the code to alleviate a large burden from such endeavour. This thesis provides an incremental and modular approach to create a process and tools to extract the semantics of legacy assembly code.

Our techniques consist of static analysis and symbolic interpretation in order to reverse engineer the semantics of legacy software. We examine the case of IBM-1800 programs in detail. From the abstract model of the operational semantics of IBM-1800, we simultaneously obtain an emulator and a symbolic analysis process. Augmented with control flow information, we can use the symbolic analysis to provide complete semantics for the code sequences of interest. We can also generate Data Flow Graphs to depict the flow of data in those code segments. The whole process of extracting semantic information from the assembler codes is fully automated with only a little human intervention at the initial step.

We use Haskell as our implementation language and its important features help us to create modular and well structured software. The literate programming documentation style in this thesis increases the readability and consistency of the implementation's documentation.

The process and the associated tools created in this thesis are used in a large reverse engineering project, which has a goal to extract requirements specification from legacy assembly code. This project is funded jointly by Ontario Power Generation (OPG) and CITO (Communications and Information Technology Ontario).

Acknowledgements

This thesis would not have been possible without the support of many people. Many thanks to my supervisor, Jacques Carette, who guided me through the whole research and read my numerous revisions to correct them. Also thanks to my committee members, Wolfram Kahl and Alan Wassyng who always offered guidance and support. Thanks to my group members whose valuable suggestions helped me a lot. A special thank you goes to my fellow student Olivier Dragon for proof reading and correcting important parts of my thesis. And finally, thanks to my parents and numerous friends who endured this long process with me, always offering support and love.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

Business organizations spend a large part of their efforts and budget maintaining existing software, enhancing with new features and adapting it to newer environments. Studies show that the maintenance of existing software can cost often more than 60 percent of all the development efforts. Maintenance in the life cycle of software is inevitable for reasons like removal of errors, new requirements for the software or introduction of new platforms etc. Maintenance can be defined as the set of activities that occur after the software has been deployed [CG03]. Development of new software from scratch when new requirements arise is grossly impractical as companies make large investments in developing existing software, creating infrastructure and organizational practices around the software, and in training users. Thus, existing software applications are assets to these organizations and as such are needed to be well maintained before being abandoned.

Nevertheless, since these systems were developed decades ago, they are usually written in older languages and use older software engineering methodologies. Legacy software is, henceforth difficult to modify and maintain. Still, the need for change is obvious as these legacy systems are consuming too much maintenance budget and efforts. Moreover these systems are becoming less efficient compared to the systems developed on more sophisticated technology as available today. Most of the software engineering approaches focus mainly on *forward engineering*– that is, on the software

development process where we move from initial requirements to logical design and design to physical implementation of the system. In this thesis, we instead try to take the legacy software perspective, by developing some tools to aid re-engineering the legacy software.

Re-engineering is a process through which "an existing system undergoes an alteration, to be reconstituted in a new form." [CG03]. Generally, the process is comprised of two distinct phases. In the first phase, the software personnel moves backwards, from the existing system to the requirements specification. This helps him to understand the structure of the system and discover ways to modify it. This phase is often called *Reverse Engineering*. During the next phase, the software engineer proceeds forward and actually designs and implements suitable changes. The main task of reverse engineering is *program comprehension*, where the software engineer tries to understand the program structure, working algorithms, data structures. It is to figure out the main components of the software that are needed to be reimplemented. As such, the software engineer needs to identify the main system components, their relationship and an abstract representation of the system to properly realize how the system functions. Complete documentation of the software which is consistent with the implementation very helpful to complete those tasks. Unfortunately, in most cases, complete documentation is not available. In fact, the software engineer will often need to proceed through the tiresome process of recovering the design from the code and rebuild the requirements specification from low level code implementation.

Lack of documentation and poor software engineering techniques during the forward engineering process of developing software are the main factors affecting the cost and effort of reverse engineering [CG03]. Most organizations that have legacy software and who intend to re-engineer their software, suffer this problem of lack of documentation. These software systems were developed at a time when software engineering technology was still in infancy. To make matters worse, in some cases documents were not properly updated during maintenance of the software; leaving them in an inconsistent state with the implementation.

Ontario Power Generation (OPG) is such an organization which has legacy software systems written in platform specific legacy assembly and is having difficulty maintaining those systems. The systems lack proper documentation or have inconsistent documentation, making the maintenance more precarious. These days, OPG

intends to re-engineer their systems into newer ones which will be running on modern platforms. In this respect, they have started a re-engineering project and in the first phase, are trying to extract the original requirements specification of their systems. This reverse engineering process is being hindered by poor and inadequate documentation. During the maintenance, they have tried to improve the consistency between the code and the documentation. However unforeseen causes (one being introduction of new features) have made the documentation rather convoluted. The reverse engineering project at McMaster is intended to help OPG with developing a new tool suite architecture and creating a complete process to reverse engineer the requirements specification of their systems.

The project, named *Reverse Engineering of High-level Requirements from Assembly Code*, aims to create a complete process with necessary methods and tools to help software engineers in reverse engineering legacy software system to high level abstract description of the system with as minimal human interaction as possible [CKK+04]. Finding abstract specifications of a system from low level assembler code includes a set of different activities. Existing tool support is enough for some cases while for the others new methods and tools have to be developed. That is why in the reverse engineering project, a tool hierarchy to extract the requirements description of the legacy software is being developed. We, as a part of the project, intend to create an automated process and associated tools to find the semantical description of the assembler code. Our main goal is to understand the meaning of the low level codes by translating them into mathematical equations. We use different techniques of symbolic analysis coupled with various compiler technology to extract the symbolic meaning of the code. The work presented in this thesis depicts a process to find the semantics of the legacy assembler codes by symbolic analysis.

Symbolic analysis of a program is a static analysis technique [FS03] that executes the instructions of a program with some of the values (of registers, input channels or memory location) as unknown symbols. This generates an ordered sequence of data flow equations which, if solved, gives a precise mathematical representation of the computations done in that program. Existing symbolic analysis tools do not always give accurate representations of the meaning of the program because they tend to introduce approximations to the semantics very early in the processing. By working with systems of symbolic, conditional data flow equations instead of sets of solutions,

we can be more accurate.

While various static analyses, including abstract and symbolic interpretation, have been successfully used for high-level programming languages, the problem becomes considerably more difficult for assembly language programs, and even more so for legacy software. In particular, while the control structures in most modern high-level languages (sequencing, `if-then-else`, `while`, etc) have very well understood semantics and effect the control flow in a predictable fashion, assembly programs liberally use gotos. Branching code is written in such a way that it can not be easily translated to a sequence of high-level structures, at least not without code duplication. Other complications (to be detailed later) include lack of data/code separation, frequent computed gotos, and even some (relatively mild) instances of self-modifying code.

What we are attempting to achieve here is, via symbolic interpretation, control flow analysis, and condition propagation, to represent a program's semantics by a system of conditional symbolic data flow equations. If this system of equations can then be solved in a space of semantically meaningful expressions, we can obtain an understandable representation of the underlying semantics. To a certain extent, we are free to choose our solution space; this allows us to choose spaces with very rich semantics. In particular, instead of choosing a high-level programming language (which would only "move" our understanding problem up some levels instead of resolving it), we choose a variety of specification languages and mathematical languages. Explicitly, we are looking at producing output that can be read natively by both PVS [OSRSC01] and Maple [MGH+01].

1.2 Thesis Organization

In the beginning we present the overview of the problem dealt with in the thesis. The backgrounds of the reverse engineering project and the legacy systems are also included. We then show different techniques which proved useful in the semantic analysis process; followed by discussion on various implementation tools.

Succeeding this, we describe the whole symbolic interpretation process in brief. We also describe different tools and their interaction inside the process. Later, we include various software engineering principles which are followed during the process

development.

The following chapter contains the operational semantics of the IBM-1800 assembler with brief description of IBM-1800 Data Acquisition and Control System. Examples from the abstract model of the operational semantics of assembler instructions are shown. The rest of the model is included in Appendix A.

We then include the implementation techniques of the emulator which are developed on the abstract model of the assembler instructions with sample outputs from the emulator. We also present one-step symbolic emulation tool, which is the first step toward symbolic interpretation process, accompanied with its design and implementation strategies. The presentation style is in literate programming [Knu84]. In this step, we follow the same model of abstract semantics from Appendix A.

Once the one-step symbolic representation is shown, we explain its use in generation of marked control flow graph with a detail documentation of the internal data structure of the graph. We furthermore discuss the tool which is used to generate the Data Flow Equations by multi step symbolic interpretation. The representation of these tools are also given as literate programs.

After the generation of Data Flow Equations (DFE) of the assembler codes, we focus on generating Data Flow Graphs (DFG) from the DFEs. We discuss the detailed process of producing DFG from the DFEs and the tool associated with it (in literate Haskell programs). Some DFG diagrams are also included as examples.

To conclude, we describe the tool to solve the DFEs in their closed forms. We also add several examples of the solved DFEs for different code patterns. Additionally, we present all the Haskell modules which were not discussed in the chapters as appendices.

Chapter 2

Problem Definition

In this chapter, an overview of the problem that we deal with in the thesis is given. First, we provide the background of the problem with the context of legacy systems and a specific instance of legacy systems in Ontario Power Generation (OPG). Next, we include a brief introduction of the reverse engineering project (of which this thesis is a part) and the hierarchical structure of the project components. Later, we present a brief description of the subject matter of this thesis.

2.1 Background

For the last 20 years, computer technologies are booming like never before. New technologies are being introduced very frequently. Often, software system developed in one technology may find itself inefficient within a short span of time due to introduction of newer efficient technologies. Constant technological advance often weakens the business value of the systems which have been developed over the years through huge investments. Another important thing to note is that advancement in hardware technologies is much more faster than that of software. For this reason, many software systems can not take the benefits of newer hardware as they are implemented to take full advantage of the hardware architecture they are written for. Although more cost-effective technologies are available, it is estimated that most of the IT systems are running on legacy platforms. Maintaining and upgrading those systems are some of the most difficult challenges today. It is worthy to change those systems into newer

technologies for gaining most efficient performance while keeping their functionalities intact. But study has indicated that most of these transformation projects took lots of investments and hard work. Still the need of change is obvious as the operation and maintenance budget for those systems range around 85-90% of their total life cost. We can define the systems which are running in older platforms as legacy systems.

2.1.1 Legacy Systems

The Free On-Line Dictionary Of Computing (FOLDOC) defines legacy system as, "A computer system or application program which continues to be used because of the prohibitive cost of replacing or redesigning it and despite its poor competitiveness and compatibility with modern equivalents. The implication is that the system is large, monolithic and difficult to modify." [How05]. Bennett also gives some more detailed characteristics [K.H95] of legacy system:

- it may be written in assembly or an early version of a third generation language.
- probably developed using state-of-the-art software engineering (programming pre 1968) techniques.
- many perform crucial work for the organization.
- generally large.
- generally hard to understand hence hard to maintain.

Many information technology related companies have this kind of systems which were developed around 30 years ago. As said in the definition of legacy system, those software are developed in a time when sophisticated software engineering techniques were not available and people who implemented those systems were not aware of modern design or coding style. In many of the cases, these systems might be running safety critical systems and were written in a variety of assembly languages. Moreover, these software were developed taking efficiency as the main goal and lack the clarity needed for large software systems. During that time, the computation power of the computer was expensive and thus they were implemented to take all the advantages

of the legacy hardware architecture. As such, they are not easily portable into newer systems and ended up in some convoluted code.

Over the years, the coding style and the documentation procedures have changed in the area of software engineering. The styles that were followed in legacy systems became obsolete and convey almost no meaning now-a-days. In the lifetime of the legacy software, as the software was augmented with new features, changed or improved for better performance, the documentations were not adjusted properly in many of the cases. To make the situation worse, for non safety critical part of the software, the later adjustment documents of the software might be missing or are changed in a way which does not make any sense in relevance of the whole system. People who developed and implemented the software may be unavailable (shifted somewhere else, retired) to get help for proper understanding of the code. Consequently, these companies find themselves in a situation where they are depending on legacy software; for which they no longer have original requirements and proper documentation to change or adjust the codes in near future.

A temporary solution of those kind of software might be emulating the legacy hardware underneath and then validating the emulator instead of determining the requirements of the software. This fix is temporarily sufficient, however, the lifetime maintenance and changes are still required which can be expensive and hazardous for the lack of documentation.

Legacy systems have worked fine for many years without proper documentation – they may do well for some future years. But as hardware technologies are now getting better than those of legacy software, so the companies depending on them might be planning to transfer the systems into newer hardware to get better performance. Also they might want to develop the systems using modern software engineering technologies so that they will be reliable and serve them for another 30 years. This may be very expensive but still the organizations find themselves in a difficult position depending on the legacy systems which are hard to maintain and change for poor documentation. So, they would plan to re-engineer their legacy systems to develop modern systems and first step toward it will be extracting the requirements of the legacy software system in a reverse engineering process.

2.1.2 Ontario Power Generation

As indicated in Chapter 1, Ontario Power Generation (OPG) is such an industry which has legacy software systems developed 30 years ago. They are still using those software while the developers of those systems left the industry long ago. Many of its original developers were not software related people and also proper software development strategies were not available during those days. As a big organization, OPG must have tried to use state-of-the-art software engineering methodologies to create those software. Still the systems developed by them lack modern documentation style needed for future maintenance. Moreover, the non safety critical part of the code is either not rigorously documented during implementation or its documentation is difficult to locate. Again, as the developers of these software are from different engineering background, their main goal was to develop an efficient software considering the constraints of the legacy hardware architecture and thus they produced codes which are not easily portable into different platforms.

During the lifetime of those software, different patches of codes were implemented to augment different features and to deal with the change of the power plant structure. Some of those patches have to deal with extreme resource constraints and thus were developed in a convoluted manner creating a layered code. All those patches have to be properly comprehended before gaining the understanding of the changes in the software. Although extreme care is taken during the previous adaptive maintenance of the system, in the long run, maintenance of the software became dangerous and more expensive.

That is why, OPG has launched a four year long reverse engineering project to cope up with the situation and is trying to determine the original requirements of the software by rigorously examining every module of those software. They are aiming to re-implement the software based on the extracted requirements with the modern software engineering methodologies applied to produce correct, robust and maintainable software which may serve them without failure for another 30 years. In the course of the project, they are providing funds (jointly with CITO- Communications and Information Technology Ontario) in a project of reverse engineering of requirements from legacy assembly code at McMaster University. The work in this thesis is a part of that reverse engineering project.

Next, we give a brief snapshot of the hardware and software technologies used in OPG and which are related with our reverse engineering project. Currently, two main hardware machine types in OPG are examined in this project: the IBM-1800 and the Varian V75. The reverse engineering group at McMaster initially got the Boiler Pressure Control (BPC) code, which is a module of the larger piece of software based on IBM-1800 hardware architecture. Most of the code segments presented as examples in this thesis are parts of this BPC code. Later, we are provided with the assembly listings which are supposed to be the whole piece of software for two machines (DCC 1 and DCC 2) with four subunits (UNITS 1-4) and each of the subunits runs a slightly different image than the rest. But still there remain issues in determining which source files are used to generate which image. An attempt was made to generate the complete image of the source code for one machine (DCC 1) and more information were needed to complete that task. The image can be extracted by comparing the slight differences among different images of the machines and subunits.

OPG no longer uses IBM-1800 machines and replaced them with the emulators for those machines. This temporary measure gave them chances to avoid issues related with maintaining legacy computer hardwares while giving away the cost of validating the emulator to represent the true machine behaviour of IBM-1800. Still they have to deal with all the software maintenance issues and hopefully this re-engineering effort will provide them with recovered original requirements. In that way, they can proceed to re-implement their systems in modern hardware platforms and getting rid of the constraints of IBM-1800 and Varian V75 systems.

2.2 Reverse Engineering Project

This section gives a brief overview of the project goal and hierarchical structure of the tool suite developed in the project to satisfy the goal. The work presented in this thesis is a part of the tool suite architecture of the reverse engineering project. A brief overview of our work is also provided in the next section.

2.2.1 Overview

As we stated earlier, industries in Ontario also have safety or mission critical systems that are dependent upon legacy software and hardware systems. Like most of the legacy systems, the documentations of these systems are often convoluted, not updated properly or in some cases missing entirely. Many of these systems are also developed in very old assembly languages. As the demand for migrating those legacy software in newer platforms is increasing in the industries, in order to be able to transfer those assembly language applications to a new, more robust platform, or for easy maintenance and update, companies need to properly extract and document the software's original requirements [CKK+04].

Most of the previous efforts in this area of reverse engineering mainly aim to transfer these "low level" languages into some high level languages like C which are supposed to be equivalent to the "low level" programs. In this project, we are specially concerned with the legacy assembly languages which do not have modern programming style and also the systems implemented in those languages feature arbitrary design decisions resulting from the legacy platform constraints. So transforming those codes into high level ones may produce codes which are convoluted and difficult to read. By this kind of transformation, its not easy to determine which requirements are parts of the problem domain and which ones are due to legacy architecture [CKK+04]. In the reverse engineering project, we instead generate a tool hierarchy to aid obtaining high level requirements of those assembly language applications.

The goal of the CITO project at McMaster named *Reverse Engineering of High-level Requirements from Assembly Code* is "to create methods and tools to assist a developer in reverse engineering a legacy assembly program to a high level requirements specification that is independent of arbitrary design decisions (but still captures the rationales of those decisions in terms of non-functional requirements)." [CKK+04] So the main theme of the project is to take existing legacy assembly language software from the industry and using as minimal human interaction as possible, generate a set of requirements documents for the software which can later be used to re-implement the software in modern platforms satisfying the original requirements (extracted in the reverse engineering process).

2.2.2 Tool Hierarchy

Extracting high-level requirements from assembler code comprises of different activities. We can use existing tool support for some of those while for others we have to create new tools. Thus our reverse engineering project will be generating a tool suite to support different activities by different tools and also a procedure which will be followed to generate the requirements documents. Figure 2.1 (taken from [CKK+04]) gives a presentation of the tool hierarchy and interaction among the tools where arrows denote "used by" relations.

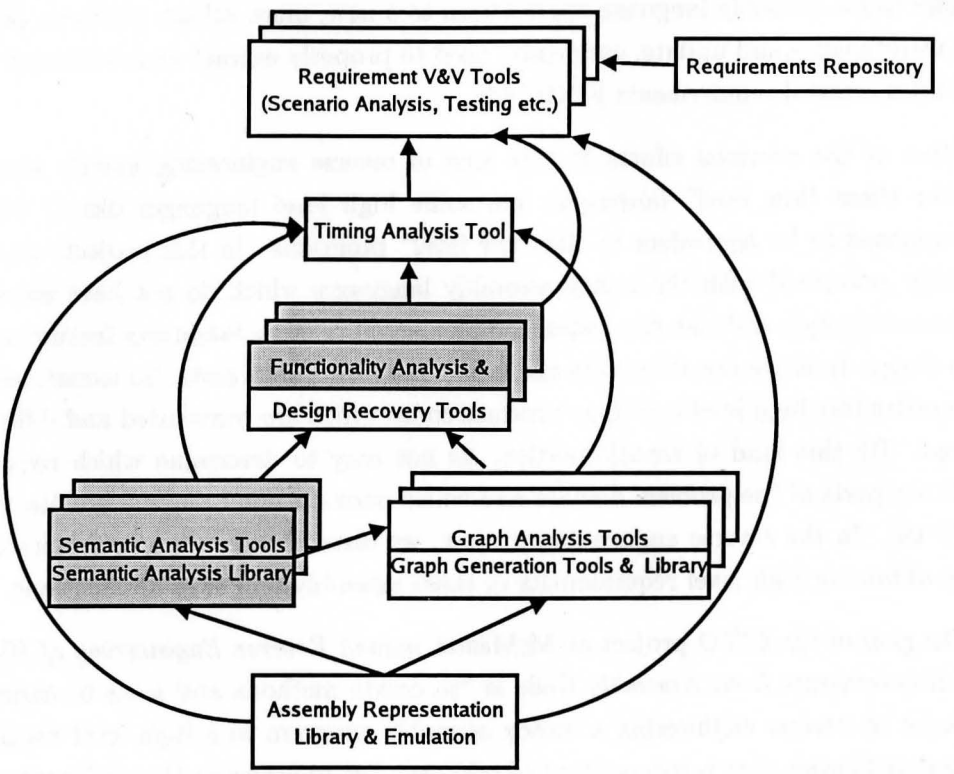


Figure 2.1: Tool Suite Architecture of the Reverse Engineering Project

The work represented in this thesis is the description of the tools that can be fit in the highlighted boxes at the bottom left corner of Figure 2.1 which are "Semantic Analysis Tools" , "Semantic Analysis Library" and "Functionality Analysis Tool".

The output of those tools will be used by Design Recovery Tools and Timing Analysis Tools towards generating the requirements document. In the following section, we give a brief overview of the Semantic Analysis process which is presented in the thesis.

2.3 Semantic Analysis

The semantic analysis tools capture the semantics of the assembly code as an aid to understanding the system's requirements. By combining flow graphs, semantic information and abstract definition, we try to find semantic interpretation of different control structures (for and while loops, if statements etc) in the assembler code. We expect that control structures have to be used along with its semantic model to properly capture the exact semantics of the underlying code although we restrict ourselves to classical structures only.

In the semantic analysis process, our main concern can be described by the following line: "we try to understand what the assembler program does". Given an IBM -1800 assembler program, we represent the meaning of the program by some mathematical equations. Those mathematical equations can be solved to find the computation done by the program or can be pictorially presented to understand the program in a better way. As the assembler codes (we are dealing with) are mainly used for computation in a control circuitry, their meanings can be better represented by a set of mathematical equations. We will be using symbolic interpretation, control and data flow analysis (these terminologies are defined in Chapter 3) techniques to present the meaning of the IBM-1800 assembler programs.

For this work, we shall take for granted that a formal specification of the operational semantics of the assembler is a definite step forward in "understanding" what a program does. This specification should be written in a specification language with consistent semantics. In later steps, the specification will be used to find the meaning of each instructions. Ultimately, for the functions in the assembler codes, we are aiming to (automatically) extract nice closed-form formulas and their graphical representations that express the actual (or idealized) semantics of those functions and thus of the assembler program. In the following subsection, we give a brief overview of the IBM-1800 assembler programs and their pros and cons related to semantic analysis process.

2.3.1 IBM-1800 Assembly Language

The codes written in IBM-1800 legacy assembly language have complex control flow, no data/code separation etc. We have an extremely complete and detailed description of the operational semantics of the machine language [IBM70] which is partly described in Chapter 5. Assembler programs can be assembled into either as a binary image (which can be directly loaded into memory for execution) or as an assembly listing (`.lst`) file. The `.lst` file is almost similar to the original source code, with some extra information like: the relative address of the instruction, the opcode (hexadecimal representation either 16 or 32 bit object), line number in the source code and also a symbol (REL) to indicate which 16-bit words use relative or absolute address during memory loading. To make the discussion more precise, here we present a small code segment of IBM-1800 assembler code (adapted from the `.lst` files provided by OPG):

```
OADDR REL OBJ. S.NO. LABEL OPCD FT OPRNDS
 35B6 0  C129  0677  TRBFB  LD   1  41
 35B7 0  A12A  0678         M    1  42
 35B8 0  1082  0679         SLT      2
 35B9 0  912B  0680         S    1  43
 35BA 0  A12C  0681         M    1  44
 35BB 0  108F  0682         SLT      15
 35BC 0  A92D  0683         D    1  45
 35BD 0  D12E  0684         STO  1  46
```

The `.lst` source code of the assembler program contains the following information:

- op codes and corresponding data (symbolic or immediate, as appropriate),
- relative addresses of the instructions,
- names for code blocks,
- names for "data" memory locations (in comments).

A human reading of those programs and operational specifications finds that

- there is no separation between code and data;
- there are many indirect (computed) jumps;

- there is no "subroutine" concept used, although IBM-1800 supports subroutines within a program;
- there is self modifying code, which however only modifies the content of addresses or registers, in other words operands of the instructions;
- although IBM-1800 assembly language has condition checking for exceptions (carry, overflow etc.), there is no exception handling in the code segments of OPG. At least we could not find any in the code examined;
- there is no stack, only memory;
- there is fixed, known data size (16 bits, 32 bits).

The first 4 items are definite complications for program understanding. The last 3 items are certainly a definite impediment to writing programs, but turn out to be quite useful in program understanding! They provide hard, definite constraints that must hold true for the program to be meaningful. For example, as there is no carry or overflow check in the code examined, then it must be the case that all arithmetic operations must not cause either carries or overflows; this implies that some side predicates must always be true for the program to be meaningful.

One important aspect of those source programs is that they are heavily commented; this is extremely helpful for the larger reverse engineering effort. Unfortunately, little automated use can be made of these comments, as:

- when the programs were maintained, the corresponding comments were not always updated,
- block comments are not always in meaningful locations, so that they cannot be used to identify meaningful blocks of code (i.e. functions),
- line comments do not always correspond to the corresponding instruction.

Needless to say, with the notable exception of "data" memory locations, the comments do not exhibit enough structure to be reliably used in an automated process.

Chapter 3

Tools and Techniques

There has been increasing interest in the application of sophisticated program analysis techniques to software development and maintenance tools. Such tools include those which are used for program understanding, verification, testing, debugging, reverse engineering etc. In this chapter, we present and describe some analysis tools and techniques which are relevant to our symbolic interpretation process.

3.1 Graphs

Graphs are appropriate models for many problems that arise in computer science and its applications. Specially in software engineering, Control Flow Graph (CFG), Data Flow Graph (DFG), Component Graph (CG) etc. give a better analytical approach to understand and characterize software architecture, static and dynamic structure and meaning of the programs [CG03]. From a pictorial sketch (by graphs) of internal structure of the code, it is always more comfortable to recognize the issues related to software engineering analysis. That is why, graphs are always preferred by the software engineers and researchers to understand, re-engineer and analyze codes.

A variety of graph analysis techniques are available for software engineering applications. Control Flow Analysis, Data Flow Analysis, Analysis using Component Graph are some of them. In Control Flow Analysis, Control Flow Graph (CFG) is used to Analyse and understand how the control of the program is transferred from one program point to another. Similarly, Data Flow Analysis uses Data Flow

Graph (DFG) to show and Analyse the data dependencies among the instructions of the program. Component Graph identifies the components of a program ; shows the use relations among those components and is very useful in software architecture identification and recovery.

In our symbolic analysis process, we use Control Flow Graph and Data Flow Graph for assembler program comprehension. In the following subsections, we discuss those two graph structures in detail.

3.1.1 Control Flow Graph

A control flow graph (CFG) [ASU86] of a program is defined by a directed flow graph $G = (N, E, x, y)$ with a set of nodes N and a set of edges $E \subseteq N \times N$. A node $u \in N$ represents a program instruction (statement), an edge $u \rightarrow v \in E$ indicates transfer of control between instructions $u, v \in N$. Node $x \in N$ and node $y \in N$ are the unique start and end node of G, respectively. Consider the control flow graph in Figure 3.1. We have a set of nodes $N = \{x, 1, ...7, y\}$ and a set of edges. The start node is denoted by x and the end node by y.

A CFG may not be connected, that is, some nodes may not be reachable from the start node x. Therefore, whenever we refer to a CFG we mean the subgraph of a CFG such that all nodes in the subgraph are reachable from the start node x and all the nodes can reach the end node y, i.e. every node is assumed to reside on a path from x to y.

3.1.2 Data Flow Graph

A data flow graph (DFG) of a program can be defined the same way as the control flow graph. It is a directed flow graph $G = (N, E)$ with a set of nodes N, a set of edges E (ordered set of node pairs) such that $E \subseteq N \times N$ and a distinct end node $y \in N$. The start node may be one or two depending on the starting instruction of the program.

The nodes and edges are divided into two groups which are different from CFG. Nodes can be of type: Operator Node and Operand Node depending on the content of the node. A node $u \in N$ is an Operator Node if it represents an operation in the program and similarly a node $v \in N$ is an Operand Node if it is an operand of the

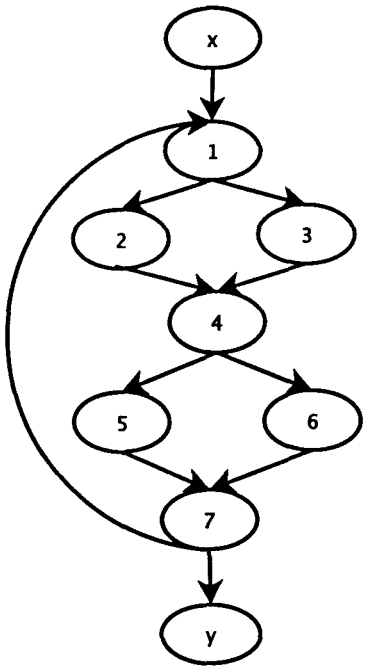


Figure 3.1: Control Flow Graph

program. Shapes of the nodes distinguish among the nodes; box shaped nodes are Operator Nodes and elliptical shape nodes are Operand Nodes. Edges can also be of two types: In Edge and Out Edge. An edge $u \rightarrow v \in E$ indicates an In Edge if $u, v \in N$, u is an operand node and v is an operator node. Again, an edge $v \rightarrow u \in E$ is an Out Edge if $v, u \in N$, v is an operator node and u is an operand node. Let us consider the data flow graph in Figure 3.2. We have a set of nodes $N = \{1, ...6, x, y, z\}$ and a set of edges with the end node 6. Nodes $\{1, ...6\} \in N$ are Operand Nodes and nodes $\{x, y, z\} \in N$ are Operator Nodes. Edges like $4 \rightarrow z$ are called In Edge and edges like $z \rightarrow 6$ are called Out Edge. A DFG may not be connected as there may not be data dependency among all the instructions of a program.

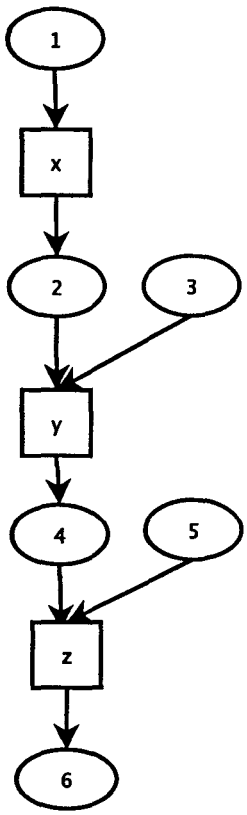


Figure 3.2: Data Flow Graph

3.2 Semantic Analysis

There are two main aspects of a computer language - its syntax and its semantics [NN99]. The syntax defines the correct form for legal programs and the semantics determine what they compute. The syntax is concerned with the grammatical structure of the program while the semantics give the meaning of grammatically correct programs. While the syntax of a language is always formally specified, the more important part of defining its semantics is mostly left to natural language, which is ambiguous and leaves many questions open. Hence methods are developed to describe the semantics of computer languages. Here we shall consider only three approaches. Very roughly, the ideas are as follows [NN99]:

Operational Semantics: The meaning of a construct is specified by the computation it induces when it is executed on a machine. In particular, it is of interest *how* the effect of a computation is produced.

Denotational Semantics: Meanings are modeled by mathematical objects that represent the effect of executing the constructs. Thus only the effect is of interest, not how it is obtained.

Axiomatic Semantics: Specific properties of the effect of executing the constructs are expressed as *assertions*. Thus there may be aspects of the execution that are ignored.

In our symbolic interpretation steps, we are mainly concerned about the operational and denotational semantics. An operational explanation of the meaning of a construct tells *how* to execute it. From the IBM 1800 manual, we use the operational description of all the instructions to develop a semantic model of them. Later this model is used to interpret the symbolic meaning of the assembler code.

Denotational semantics [NN99] is a methodology to define the precise meaning of a computer language. In denotational semantics, a computer language is given by a valuation function that maps programs into mathematical objects considered as their denotation, i.e. meaning. Thus the valuation function of a computer program reveals the meaning of computer programs . At the end of our symbolic interpretation process, we will define the meaning of the assembler code by some mathematical equations i.e. our symbolic interpretation process gives the denotational meaning of the assembler codes without considering the syntactical premises of the program.

3.3 Program Transformation

Program transformation techniques are helpful in the areas of software engineering like program synthesis, reverse engineering, documentation generation etc [Pro05]. Lots of theories, tools and applications on program transformation have been developed for these areas.

What program transformation does is to change one program into another. The language in which the program is written and the resulting program after the transformation are called the source and target languages, respectively. In a program translation scheme, a program is transformed from a source language into a program

in a different target language [Pro05]. Although transformations aim at preserving the exact semantics of a program, it is usually not possible to retain all information across a translation.

In the area of Reverse Engineering, the purpose of program transformation is to extract from a low-level program a high-level program or specification, or at least some higher-level aspects. Reverse engineering raises the level of abstraction and is the dual of program synthesis [Pro05]. Examples of reverse engineering are decompilation in which an object program is translated into a high-level program, architecture extraction in which the design of a program is derived, documentation generation, and software visualization in which some aspect of a program is depicted in an abstract way.

3.4 Implementation Tools

The tool suite architecture of the reverse engineering project is hierarchical and will become more complex as it grows. That is why, it is more important to structure it well. As we know, well-structured software is easy to write and debug. Moreover, it provides a organized collection of modules that can be re-used in course of time to reduce future programming costs. Hughes [Hug90] argued that modularity is the key to efficient and successful programming. Efficient programming languages must support modular programming as well. But modularity means more than modules- the success of decomposing a problem into parts depends directly on the ability to glue solutions together. To assist modular programming, a language must be featured with good glue for modules. In this regard, we look for a programming language genre which provides us with efficient modularity features to generate well organized software.

3.4.1 HASKELL

Hughes [Hug90] showed that conventional languages are more constrained with modularization while functional languages push those limits back. Functional programming is a genre of programming which mainly depends on the evaluation of expressions, rather than execution of commands. Expressions of these languages are formed by

combining functions. In our implementation, we will be using a functional programming language called Haskell to extract and manipulate information from IBM-1800 assembler codes.

Haskell is a pure functional programming language with open source compilers for almost all modern computer and operating systems. In this project, we use Glasgow Haskell Compiler (GHC) [Has] which either generates C code as an intermediate step or on some platforms generates native code. In the following paragraphs, we discuss some important features of Haskell which made useful Haskell as our implementation language.

One of the most important and powerful feature of Haskell (as it is a pure functional language) is functional composition[Has]. Haskell programs can be written as the composition of functions. Operations on a set of data can be represented as functions and functions can be glued together to create complex functions which can describe complex operations on data. Functions can also be passed as parameters to other functions as objects and thus they allow us to create generic functions. These techniques lead to natural modularization of the program. In Haskell, each function is created as a composition of other functions, and thus a hierarchy of functions always exists. Instead of creating large monolithic functions, we can create several small functions and glue them together to create the larger functions. In this way, it presents the software developers a sophisticated technique to create layered and well structured software. Verification of the software for correctness is also more easier as verifying the hierarchical functions for totality can easily assert the correctness of the program.

Haskell offers new ways to encapsulate abstractions [Has]. An abstraction allows us to define an object with the internal logic implementation hidden from outside. Abstraction plays a key role in building modular and maintainable programs. One important abstraction mechanism available in Haskell is the higher-order function. As we mentioned earlier, in Haskell, functions can freely be passed to or returned from other functions, stored as objects in data structures and so on. This can substantially improve the structure and modularity of many programs.

Functional languages like Haskell use lazy evaluation: they only evaluate as much of the program as required to get the answer. This allows us to use infinite types in the programs. For example, functions using varying length lists can share an infinite

list and each function will only evaluate the list necessary for its own execution. This demand-driven evaluation provides powerful "glue" to compose existing programs together. Thus it is possible to re-use programs, or pieces of programs, more frequently than can be done in an imperative style of programming (like C); allowing us to write modular programs easily.

Pattern matching is another important technique for function definition in Haskell. Proper use of pattern matching can produce clear representation of different possible inputs of the function. Although, over use of pattern matching can lead to verbose and convoluted codes.

Like some imperative languages, Haskell also has strong typing system [Has]. This provides facilities to detect and solve typing related errors before compilation and thus reduces the chance of errors like type mismatching or null pointer assignment etc. during runtime. However, in some cases, Haskell's type system is much less restrictive than imperative languages as it provides polymorphism. Polymorphism enhances re-usability of codes as generic functions can be defined to solve similar kinds of problems for different types.

Literate Programming (introduced by Knuth [Knu84]) is a programming methodology where the code and the documentation of the code can be interspersed together in a single file. In that way, a document can describe a program as well as containing it. Haskell provides support for literate programming by combining Haskell programs with LATEX. Thus a single document in Haskell can be compiled into an executable program or typeset directly into a format for publication. This helps the programmer to keep the code documentation up-to-date and in conformance with the implementation.

Haskell relieves the program developer of the storage management [Has]. Storage allocation, initialization and garbage collection are done implicitly. But problems like stack overflow may occur while manipulating large amount of data in a Haskell program. Efficient implementation of the Haskell functions may help the programmer to overcome this disadvantage.

3.4.2 GXL

GXL [Win01] stands for Graph eXchange Language. It is designed to be a standard exchange format for graphs. GXL is an XML sublanguage and the syntax is given by a XML DTD (Document Type Definition). This exchange format offers an adaptable and flexible mean to support interoperability between graph-based tools.

In particular, GXL was developed to enable interoperability between software re-engineering tools and components, such as code extractors (parsers), analyzers and visualizers. In our reverse engineering project, we choose GXL as a graph exchange format for various reasons. Among variety of available graph exchange formats, we require a format which would allow us to represent the semantics of the graph in a formal way and also we can verify the transformation of graphs in a rigorous manner. Ms. Wu and Dr. Kahl [Wu04] have already done some work on the formalization of GXL. In addition, GXL is represented in human readable XML format which might be very advantageous later on.

The most important benefit of GXL for this project is that there exists some re-engineering tools which utilize GXL as exchange formats. At the beginning of the project, it is predicted that some of these tools might be proved useful in the project. Consequently, GXL produced by the tool suite in the project may be verified and tested with the variety of other GXL based tools. Also, the output of these tools can be used by the standardized tools to produce different aspects of extracted information. GXL is also very much flexible to generate various types of graph representation. Wu [Wu04] showed that it is possible to represent Control Flow Graph and Data Flow Graph using different views of the graphs in GXL. In this thesis, We use GXL to output Control and Data Flow Graphs for exchange purposes.

Chapter 4

Process Overview

In this chapter, we present a brief overview of the whole symbolic interpretation process. We also discuss all the tools and their interactions in short. Later, we discuss different software engineering principles and their application throughout our software development process.

4.1 Major Steps in Our Process

As we discussed in Section 2.3, the whole symbolic interpretation process is automatic without any human intervention. In this section, we decribe briefly major steps that are being followed during the symbolic interpretation of IBM-1800 assembly language codes. For each consecutive step, output of one step will be the input of the next step.

Figure 4.1 gives the steps of our overall symbolic interpretation process. More specifically, these are:

- Use the complete operational semantics of IBM-1800 to derive (human assisted)
  - an emulator, as an explicit state transformer. This emulator will take a `.lst` code file as input, a starting state, and finds the final state of the machine after the execution of that code file.
  - a one-step symbolic emulator. This finds the complete symbolic interpretation of any instructions, given as the state transformer induced by the

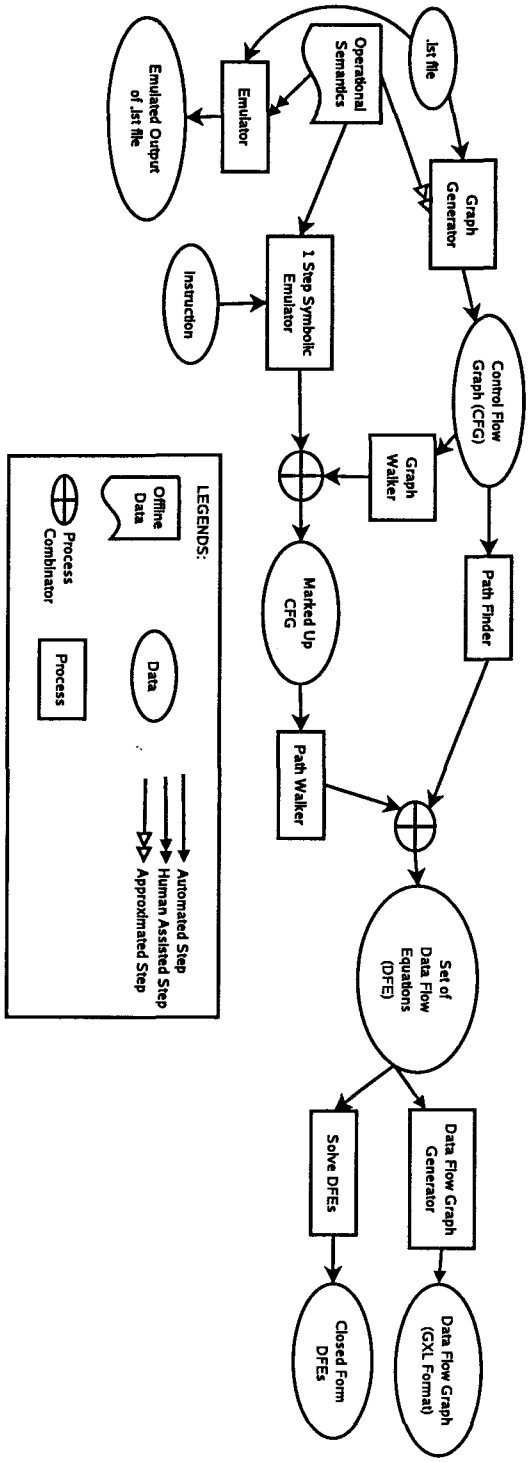


Figure 4.1: The Steps of Symbolic Interpretation Process

operational semantics. The derivation of the one-step symbolic emulator is human assisted in our process. However, It can be automated by creating proper formal representation of the operational semantics. For this reason we show the step as automated in Figure 4.1.

- Use the `.lst` code file to derive an approximated Control Flow Graph (CFG).
- Combine the CFG and one-step symbolic emulator to derive a marked-up CFG. In this derived graph, each edge of the CFG will contain the complete one-step symbolic interpretation of the instruction contained in the source node of the corresponding edge.
- Find execution paths in the CFG.
- Combine the marked-up CFG and the execution paths to find the dataflow equations (DFE) for the assembler program. In this combination process, we find all the splits and joins in the paths to find the high level control structure of the code.
- Solve those simplified DFEs to find the closed form representations.
- Generating Data Flow Graphs from the DFEs.

A more detailed description and specifications of the inputs and outputs for the important steps are given in the following subsections:

4.1.1 Control Flow Graph Generator

The first phase of this step is being done by Kevin Everets [Eve04] and in this step the approximate control flow graph of the assembly language code is being found. In Kevin's tool, the output is being represented in GXL (Graph Exchange Language) format for easy and standard graph interchange between the tools. It takes a `.lst` code file of the legacy assembly code as input and produces the GXL format CFG (Control Flow Graph) of the assembler code.

As GXL format is an exchange format and is not easy to handle, in the second phase we generate an internal data structure of the control flow graph that contains

only the necessary information for symbolic interpretation of the corresponding program.

Input: Control flow graph in GXL format.

Output: A graph data structure that contains two finite maps: one is between the nodes and their corresponding instruction opcodes and the other is between nodes and next possible edges from the corresponding node.

Type Signature: The implementation is described in Section 8.2.1 with the following type signature:

```
gxlToMyGraph :: Gxl.Gxl  -> Int -> MyGraph
```

4.1.2 Emulator

By using a complete translation of the operational semantics of the IBM-1800, we can create a complete emulator. As mentioned earlier, this emulator can be seen as a state transformer which finds the final machine state (that is different machine components with their final values) after execution of a set of instructions. We "load up" a complete state via reading in a `.lst` file of the code given and also create a representation of the state with all the state components having some initial values. The whole memory is given an initial value by loading the `.lst` file in an array with 2^{16} entries. Then the emulator emulates the execution of the set of instructions given on this initial machine state to find the final state after the execution. Here we assume that the set of instructions (i.e. assembler code) given is always terminating otherwise the emulator will produce some aberrant output without indication.

This step is mainly used to find the correctness of the model of the assembler semantics that we used to develop the next symbolic analysis steps. We have compared the output of this step with standard independently written emulator of IBM-1800 and obtained the same results.

Input: Source code of an IBM-1800 assembler program with initial machine state.

Output: Final machine state after the execution of the program.

Type Signature: The implementation is included in Appendix C with the following type signature:

```
emulate :: Int -> State -> State
```

4.1.3 One Step Symbolic Emulator

The one-step symbolic emulator produces the symbolic representation of the state transformer for an instruction. For each instruction given as input, it produces a symbolic interpretation of the state after execution of that instruction. Instead of concrete values, this representation contains symbolic expressions for the values of the state components that are being changed by the execution of that instruction, and also a symbolic path condition that reflects possible condition induced by the instruction.

As we see in Figure 4.1, One Step Symbolic Emulator is used to interpret symbolically the instruction opcode of nodes in the control flow graph and to annotate the following edges from that node by the corresponding symbolic interpretation. So the combination of the CFG walker function and One Step Symbolic Emulator produce the following input and output.

Input: Control Flow Graph of the assembler code.

Output: A marked-up CFG with the edges labelled by the symbolic interpretation of the instruction associated with their source nodes.

Type Signature: We describe the implementation of One Step Symbolic Emulator in Appendix D with the following type signature:

```
sSemantics_ :: Op -> Instruction -> [(CondFunc,[Func])]
```

The function to create marked-up CFG is discussed in Section 8.2.1 with type signature:

```
doAnnotation :: MyGraph -> MyGraph
```

4.1.4 MultiStep Symbolic Emulator

Basically, this step is a functional combination of the Path Finder and Path Walker functions on the marked-up CFG. This step finds the symbolic interpretation of an assembler code (a set of instructions). Our symbolic interpretation will not find any high level equivalent of the assembler code, instead it finds a set of Data Flow Equations (DFE) which defines the computation done by that code. Definition of DFEs is given in the corresponding chapter.

As the control flow in the assembly language is arbitrary, a major challenge in symbolic analysis of assembly language programs is to model the control flow. We model this arbitrary control flow by a set of execution paths in the program. Every program has a starting point and we can define a program path as a sequence of instructions that can possibly be executed during some run of the program. All program paths begin from the starting point of the program. We try to find some predefined structures (like branching structures, sequential codes, loops) in the program paths to find the symbolic constructs of the program.

Path Finder functions find the program paths in the Control Flow Graph. Using those paths, Path Walker functions find the control structures in the code and gather all the annotations (interpretation of the instructions) of the edges in the paths to find the semantic context of the code.

Input: Marked-up control flow graph of the assembler code.

Output: Data Flow Equations (DFE) which show the high level representation of the code.

Type Signature: This implementation includes two modules: Path Finder and Path Walker. Path Finder module is described in Section 8.5.1 with type signature:

```
nodesFromStart :: MyGraph -> MyNode -> [FinalPath]
```

The function for finding DFEs is discussed in Section 8.5.2 with type signature:

```
findAnntOfGraph :: MyGraph -> MyNode -> [[([ConditionStmt],[Stmt])]]
```

4.1.5 Generating Data Flow Graphs

DFEs show the flow of the data in symbols but we can get better pictorial presentation of the data flow in the Data Flow Graphs (DFG). From the DFEs generated in Multi Step Symbolic Emulator, we produce Data Flow Graph (DFG) which gives better understanding of the data flow in the given chunk of assembler code. In generating the DFG, we first produce a DFG which contains redundant entries. When we create the nodes corresponding to one instruction, we don't know which part of the output value will be used later. So we create some redundant entries (whenever possible) for the output values of the instructions. Some of them may be used in the next

instructions of the code. Consequently, we remove the unused entries (in the garbage collection phase) to give the final representation of the DFG.

Input: Data Flow Equations (DFE) generated by Multi Step Symbolic Emulator.

Output: Data Flow Graph (DFG) of the given code.

Type Signature: We include the implementation in Section 9.4 with the following type signature:

```
dfdGraphToGxlGraph :: GxlGraph -> String ->
                      [[([ConditionStmt], [Stmt])]] -> GxlGraph
```

4.1.6 Solving Data Flow Equations

By solving we mean to find a symbolic expression for each variable in the right hand side of DFE which can be calculated (symbolically) from the previous DFEs. Our Data Flow Equations (DFE) give a sequential set of statements which represents the computation done in the code. We follow two steps to find the solved flow equations for each DFE. First, we evaluate the variables on the right hand side of the statement i.e. we find a symbolic expression for each variable. Then, we substitute the variables with the expression while keeping the operators in place.

At the end, after solving each of the DFEs, we find the inputs and outputs of the code and also the system of equations which defines the relationship of the inputs and outputs. Input means the values which are being read in by the code and the outputs are the final values which are being written to.

Input: Data Flow Equations (DFE) of the assembler code.

Output: Inputs, outputs and system of equatios representing the computation of the code.

Type Signature: We discuss the implementation in Section 10.2 with the type signature:

```
solveAnntOfGraph :: [[([ConditionStmt], [Stmt])]]
                    -> (ConditionStmt,[Recur_Stmt],EvalHistory)
```

4.2 Software Engineering Principles

In this section, we discuss some important software engineering principles which are central to successful software development [CG03] and their role and impact in the development of our reverse engineering process. Although these principles appear to be strongly related, we prefer to describe them separately and in general terms.

4.2.1 Rigor and Formality

Rigor [CG03] stands for precision and exactness- which is an intuitive quality and can't be defined in a rigorous way in software development. Various degrees of rigor can be achieved; the highest among them is called formality where the whole software development process is driven and evaluated by the mathematical laws. We don't have to be always formal during the design phase of the software but we must be able to identify the level of rigor and formality that should be achieved.

In our process where we try to analyze mathematically the IBM-1800 assembler codes, the instructions of IBM-1800 should be modeled as formal mathematical equations. For this reason, we model the natural language description of the IBM-1800 instructions (See Appendix A) in a formal way as a combination of logical and mathematical formulas during the design process. Each instruction is modeled as a state transformer equation with some operations on the state components. During programming (a traditional formal approach in the software development process), we directly translate the formal model of instructions into the programming objects which are automatically checked and verified for correctness by the compilers.

Rigor and formality also apply to whole software process. Rigorous documentation helps the programmers to reuse the codes. Using literate programming style in Haskell, we try to be as formal as possible during the development process which might later be useful in code reuse.

4.2.2 Separation of Concerns

Separation of concerns helps us to deal with different aspects of the problem while concentrating on each independently at a time. Different types of separation of concerns are in practice in software process [CG03]. Most of them are dealt with the

higher level design of the reverse engineering tool suite architecture; where our symbolic interpretation process is a part. One important type of separation of concerns is to work with different parts of the problem separately. Using modular development strategy, we have divided the whole software in the symbolic analysis process into several steps. At each step, we are not concerned with the next steps and necessary adjustment in both the design and the previous steps are made depending on the current step of development. In this way, we can easily concentrate on the current step. While working on generating Data Flow Equations of the assembler codes, we are least concerned with generating Data Flow Graphs; thus cutting the problem into smaller managable subproblems.

4.2.3 Modularity

A system that is composed of modules is called modular. Modularity is essential to build a well structured, layered and maintainable software. As we saw in Figure 4.1, we divide our whole process into modules where each module is taking care of a different part of the process; thus implying the principle of separation of concerns. First, the whole process is decomposed into modules with an initial model of the instructions of IBM-1800. Then, we concentrate on individual module design to manipulate the model; following a top down design process.

As all other functional programming languages, Haskell provides us with important features to modularize the programs (See Section 3.4.1). Each major module in our process uses the output of only a few previous modules; thus inducing low coupling. However, each internal function inside the modules are related strongly to give high cohesion. We keep all the common codes of different modules in separate modules to reduce coupling and also to eliminate repetition of codes from the program.

Interaction of Modules

The success of modular software also depends on proper and faster interaction of modules. We develop explicit import and export list of the functions inside the modules to make the interaction of the modules faster. A pure functional programming language like Haskell helps to create interfaces among the modules in a better way to

aid software engineering.

As there are several re-engineering tools available, we provide all the output of graphs (either control flow or data flow) in GXL (Graph eXchange Language) format (described in section 3.4.2). In the tool suite architecture of the reverse engineering process, all the graph information exchange are in GXL. GXL is an exchange format and is not easy to handle. So we create our own internal data structure of the input graphs to share information needed and to decrease interaction time among the internal modules.

4.2.4 Abstraction

Abstraction is a basic technique for understanding and analyzing complex problems [CG03]. By abstraction, we can ignore the complex details of an object and concentrate on the facts that we think relevant. We use Haskell as our implementation language which has a better abstraction mechanism. Using abstract data types in Haskell, we place an abstract layer on each of the module details. This provides us with better program understading and easily maintainable software.

4.2.5 Anticipation of Change

Software may undergo changes contantly. These changes may be due to elimination of errors or future adaptation in different platforms. Basically, incorporating anticipation of change in the design strategy means to isolate the likely changes in specific portions of the software so that future changes will be restricted to those portions only [CG03].

We have translated the model of the instructions for the IBM-1800 in separate Haskell modules in both the emulator and symbolic emulator. We can just replace those modules by different models of different assemblers to use the process or the software in different platforms.

Reusability

Reusability is a software quality which is strongly effected by the anticipation of change. Reusability of Data Flow Equation (DFE) and Data Flow Graph (DFG)

generation tools in different assemblers was a primary concern of the reverse engineering process. As we said earlier, we can just replace the model of the assembler by a different one to create different DFE or DFG generation tools for that assembler.

The use of Haskell as the implementation language in generating the representation of IBM-1800 assembler facilitates our tools to use the code from the assembler representation and Control Flow Graph generation tools by Kevin Everets [Eve04]. As examples, the code to read the `.lst` file and to create the control flow graph is used by the emulator and symbolic emulator, and the code to read and write GXL representations [Eve04] is used by our tools to produce GXL presentations.

4.2.6 Generality

The principle of generality may be stated as follows: "Every time you are asked to solve a problem, try to focus on the discovery of a more general problem that may be hidden behind the problem at hand" [CG03]. While generating the semantic analysis tools, a prime concern was to develop the tools in a way so that different architectures can be represented and with minimal changes in the tools, we can perform semantic analysis of those architectures. The architecture might have different instruction sets, registers, memory size and timing mechanisms. This increases the portability of the tools on different architectures.

Portability on Different Architectures

We can reuse the same code for the semantic analysis tools of IBM-1800 in different architectures like Varian V75, MIPS and other most commonly used architectures. Haskell provides us with abstraction and modularization mechanism to attain this goal. Using abstraction, we isolate and localize the anticipated change in the internal data structures. Functions operating on those structures and their helping functions will continue to work without much change although the modules cotaining the model of IBM-1800 have to change significantly to incorporate different architectures.

We hope that this independence of architecture will work to support differnt assemblers. In case of changes have to be made across more modules to incorporate different architecture, a provision is left to re-factor the code to fully isolate the platform specific codes.

4.2.7 Incrementality

Incrementality applies to a process that proceeds in a incremental way [CG03]. We can add differnt features of a process in increments. A good software design must incorporte provision to add new features easily. In our semantic analysis process, our first goal was to find the Data Flow Equations of the assembler codes. Our model is designed in a way that later we added different features like generating Data Flow Graphs, solving Data Flow Equations with slight adjustment in the same design. In future, it is possible to add new features like generating pre and post conditons for the assembler codes using this design.

Chapter 5

Operational Semantics of Assembler

This chapter contains the operational semantics of the IBM-1800 Assembly Language and also a brief overview of IBM-1800 Data Acquisition and Control System. This part is mostly taken from IBM-1800 Operating Manual [IBM70]. Later we include an abstract model of the operational semantics which will be used for the symbolic interpretation process.

5.1 IBM-1800 System

The IBM-1800 Data Acquisition and Control System [IBM70] is developed to handle a wide variety of real time applications such as process control and high speed data acquisition. It has the following main physical units-

- 1801 or 1802 Processor-Controller
- 1803 Core Storage
- 1826 Data Adapter Unit
- 1828 Enclosure for Rack Mounting of Analog Input/Output
- 1810 Disk Storage

- Customer Signal Cable for screwing down terminals at the rear of the unit.
- DP I/O equipment

The 1801 and 1802 Processor-Controllers [IBM70] named as stored program computers, consists of a central processing unit (CPU),core storage and I/O channel control circuits. Standard features of the processor-controller include:

- Three index registers
- Twelve levels of interrupt
- Three data channels
- Three interval timers
- An operations monitor
- A programmer's console which may be used to input program manually using console switches.

5.1.1 Stored Program Concept

1801 and 1802 processor controllers are called stored program computers for their following characteristics [IBM70]-

- The stored program contains all the words addressed by the instruction register from the core storage.
- Instructions are normally stored and executed sequentially, beginning with address 0000.
- Sequential execution of a program can be altered by changing the contents of the instruction register.
- Program instructions can be modified by conditions set forth in the program.
- Program is loaded initially from a designated card or paper-tape input unit, or manually from console switches.

- Additional instructions can be entered into core storage during the course of a program.
- There can be any number of degree of subroutines within a program.

The registers of 1801 processor controller that are used to execute instructions are as follows[IBM70]:

- **Accumulator (A)**: It stores one factor of an arithmetic operation; the D register contains the other factor. It contains the result of an arithmetic operation and can be shifted right or left.
- **Accumulator Extension Register (Q)**: An extension of the low order end of the accumulator; 16 bits. It stores the 16 least significant bits of a multiplication operation and the remainder of a division operation.
- **Instruction Address Register (I)**: It is a 16 bit register, connected as a counter to maintain the address of the next instruction.
- **Index Registers (XR)**: Three Index Registers (XR1, XR2, XR3) are mainly used for address modification.
- The other registers are Arithmetic Factor Register (D), Storage Buffer Register (B), Storage Address Register (M), Temporary Accumulator Register (U) and Shift Counter (SC).

The magnetic core storage (1803) works as memory unit for the 1801 processor controller and is self contained on a single SLT board. Core storage arrays are available in two sizes, 4096 (4K) words and 8192 (8K) words. Each word consists of 18 bits [IBM70]. In the core storage (1803), bit 16 and 17 (last two bits of a word) are used for hardware operation. The part of a word where data can be stored is 16 bit in size. That is why, the logical word size in IBM-1800 is considered as 16 bit.

5.1.2 Machine Language

The IBM-1800 machine language has the following important features:

- Data and instructions are handled in binary form in 16 bit words.

- Hexadecimal notation is used to represent the machine language.
- Two word format allows data and instruction words of 32 bits.
- Negative numbers are handled and stored in two's complement form.

5.1.3 Data Format

In the 1800 system, [IBM70] the standard, or single-precision data word is 16 bits in length. Bit positions 0 through 15 represent decimal values of 2^{15} through 2^0 respectively.

Positive numbers are represented in true binary form, whereas negative numbers are in two's complement form. The sign bit (position 0) is always 0 for positive numbers and 1 for negative numbers.

The largest single-precision positive number that can be represented is $2^{15} - 1$ or 32,767 and the largest negative number is -2^{15} or -32768. A double precision number of 32 bits can be used to give a number range from +2,147,483,647 to -2,147,483,648 ($2^{31} - 1$ to -2^{31}).

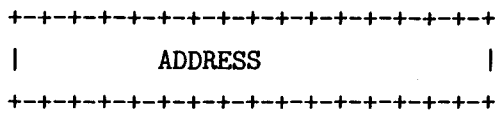
5.1.4 Instruction Format

The instruction defines the basic operation to be performed and contains the factors necessary for developing a core storage address. This core storage address is called the effective address (EA).

Two basic instruction formats are used: a single-word instruction and a two-word instruction [IBM70].

```
Short:
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   OP  |D|F| T |   DISP        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Long:
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   OP  |D|F| T |I|B|   COND    |
```




```
OP = OpCode of the instruction
D = 5th Bit
F = Format (0 = One-Word, 1 = Two-Word)
T = Tag Value
I = Indirect Addressing (0 = Direct, 1 = Indirect)
B = Branch Out (0 = BSC, 1 = BOSC)
COND = Condition flags interrogated on a BSC or
       BSI instruction
DISP = 8 bit displacement address
ADDRESS = Address of the core storage location
```

The two-word instruction contains the full core storage address in the 16 bits of the low order word. The single-word instruction is used when its not necessary to furnish the full core storage address, but only to modify(displace) a base address already existing in a designated 16-bit register. The displacement bits, 8 through 15, can be used to address a range of core storage locations from 127 addresses above the base address to 128 below the base address.

The address portion of a two-word instruction can also be modified by adding to the contents of a designated 16-bit index register.

The bits within the instructions are used in the following manner[IBM70]:

- **Op Code**: The operation to be performed by the instruction in defined by these five bits. There are 26 valid op codes.
- **Format(F)**: This bit selects the instruction format. A "0" indicates a single word instruction and a "1" indicates a two-word instruction.
- **Tag(T)**: These are index tag bits used to select a register for address modification.
- **Displacement**: These eight bits define the displacement value and added to the register specified by the tag bits to develop the effective address(EA). Dis-

placement may be in either positive and negative direction as determined by the sign of the displacement value. A negative displacement value will be in two's complement form with a bit in position 8.

- **Indirect Address(IA)**: This is the indirect address bit in the two-word instruction format except in the modify-index-and-skip instruction with a tag 00 specified. If "0", addressing is direct. If "1", addressing is indirect.

- **Branch Out(BO)**: This bit is used to specify that the branch-or-skip-on-condition instruction is to be interpreted as "branch-out-of-interrupt routine".

- **Conditions**: These six bits specify the indicators to be tested on a branch-or-skip-on-condition instruction.

- **Address**: These 16 bits usually specify a core storage address in a two word instruction. The address can be modified by the contents of an index register or used as an indirect address if the IA bit is on.

5.2 Semantics from Manual

In this section, we include all the instructions of IBM-1800 assembler with their mnemonics. It also contains the hexadecimal representation of the instructions and the meaning of different bits in the instruction code. Later the operational semantics of one instruction is represented with an example.

5.2.1 Instruction Set

The IBM-1800 instruction set is shown in the Figure 5.2.1. An invalid code (0000) enables the programmer to detect an inadvertent branch to a blank area of core storage. Each instruction falls into one of five classes [IBM70]. Note that the instructions which may be used with indirect addressing are indicated in the Indirect Addressing column.

Some instructions perform multiple uses, as specified by their control bits. A more complete breakdown of instructions, including hexadecimal representations, is found

Class	Instruction	Indirect Addressing	Mnemonic
Load and Store	Load Accumulator	Yes	LD
	Double Load	Yes	LDD
	Store Accumulator	Yes	STO
	Double Store	Yes	STD
	Load Index	**	LDX
	Store Index	Yes	STX
	Load Status	No	LDS
	Store Status	Yes	STS
Arithmetic	Add	Yes	A
	Double Add	Yes	AD
	Subtract	Yes	S
	Double Subtract	Yes	SD
	Multiply	Yes	M
	Divide	Yes	D
	And	Yes	AND
	Or	Yes	OR
	Exclusive Or	Yes	EOR
Shift	Shift Left Instructions		
	Shift Left Logical (A)*	NO	SLA
	Shift Left Logical (AQ)*	NO	SLT
	Shift Left and Count (AQ)*	NO	SLC
	Shift Left and Count (A)*	NO	SLCA
	Shift Right Instructions		
	Shift Right Logical (A)*	NO	SRA
	Shift Right Arithmetically (AQ)*	NO	SRT
	Rotate Right (AQ)*	NO	RTE
Branch	Branch and Store I	Yes	BSI
	Branch or Skip on Condition	Yes	BSC(BOSC)
	Modify Index and Skip	**	MDX
	Wait	NO	WAIT
	Compare	Yes	CMP
	Double Compare	Yes	DCM
I/O	Execute I/O	Yes	XIO

* Letters in parentheses indicate registers involved in shift operations.
** refer to the [IBM70] for the individual instruction (MDX and LDX).

Table 5.1: Instruction Set

in the description of each instruction at [IBM70]. In the following subsection, we will show only one instruction with its complete bit representation as example.

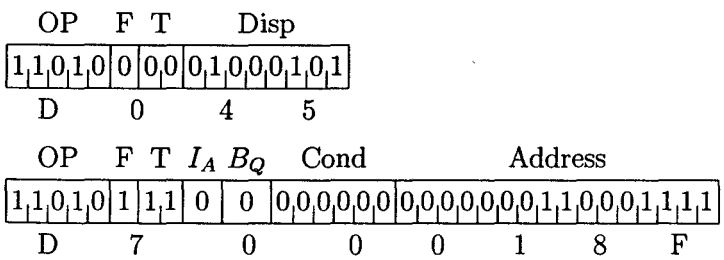
Instruction Format Symbology

Symbols are used to describe the instruction format and objectives. The symbols and their meanings are:

Symbol	Meaning
A	Accumulator
Q	Accumulator Extension Register
Address or Addr	Contents of the address portion of a two word instruction
C(XX)	Contents of core storage at the location specified by XX.
DISP	Contents of the Displacement portion of a one word instruction.
EA	Effective Address
$EA+1$	Next higher address from the Effective Address.
I	Contents of the Instruction Register.
XR1	Contents of Index Register 1.
XR2	Contents of Index Register 2.
XR3	Contents of Index Register 3.
X	Hexadecimal value can be 0-F.

Hexadecimal Representation

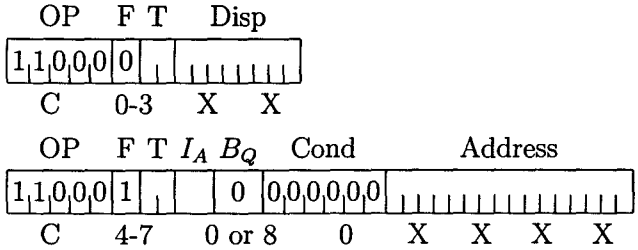
The hexadecimal number is derived by dividing each word into groups of four bits each and assigning a hexadecimal value corresponding to the decimal (BCD) value of each group. The following illustration shows a hexadecimal value for each group of four binary bits.



5.2.2 Instruction Example

LOAD ACCUMULATOR (LD):

Load operations normally transfers data from core storage to the machine register specified in the instruction.



- Transfers the contents of the core storage location specified by the effective address(EA) into the accumulator.
- The contents of the core storage location are unchanged.
- modifier bit 9 = 1 selects auxiliary storage of addressed core storage details.

One Word Instruction:

- Load C(EA) into A.

Hexadecimal Representation	Effective Address
C0XX	I + Disp
C1XX	XR1 + Disp
C2XX	XR2 + Disp
C3XX	XR3 + Disp

Two-Word Instruction, Direct Address:

- Load C(EA) into A.

Hexadecimal Representation	Effective Address
C400XXXX	Addr
C500XXXX	Addr + XR1
C600XXXX	Addr + XR2
C700XXXX	Addr + XR3

Two-Word Instruction, Indirect Address:

- Load C(EA) into A.

Hexadecimal Representation	Effective Address
C480XXXX	C(Addr)
C580XXXX	C(Addr + XR1)
C680XXXX	C(Addr + XR2)
C780XXXX	C(Addr + XR3)

5.3 Model of Operational Semantics

It is clear that every single instruction (for this and most other processors) has a complete operational description. By complete, we mean that every instruction has a premise-free description. Furthermore this operational description straightforwardly induces a denotational semantics, as a pure state transformer, where our state includes the whole memory as well as all registers. Lastly both of these semantics are (by definition) compositional.

More precisely, we want to model the effect of executing an instruction as a total function on states $[\![\ ]\!]$ to be a total function on states:

$$[\![\texttt{Instruction}]\!] : (\texttt{State} \rightarrow \texttt{State})$$

Here, we will be presenting model of only two instructions as example. The full abstract model of the operational semantics of the IBM-1800 assembler instructions is included in Appendix A.

The followings are only a part of the notations used to describe the operational semantics of the instructions of IBM-1800 assembly language [IBM70] at the abstract model in Appendix A. These notations may be used to understand the model of the two instructions presented as example.

`Inst(I)`	Contents of core storage at the location specified by I (Instruction Register). Later we use `i` as its short notation.
`DB`	D (5th) bit of the instruction opcode.
`FB`	Format bit of the instruction opcode.
`displ`	Displacement associated to the instruction.
`addr`	Address defined in the instruction.
O_{6-8}(`i, s`)	Checks bits 6–8 of the opcode, then according to bits 7&8, returns the contents of `I, XR1, XR2, XR3` if bit 6 is 0, otherwise returns value of `0,XR1,XR2,XR3`.
X	O_{6-8}(`i, s`)
`loc`(X)	If `i` is indirect then ^(X + `addr`) else X + `addr`.
$locBS$(`i`)	If `i` is indirect then ^`addr` else `addr`.
`cmdx`(mn, mp)	Compares two values of one state component (specially index registers) before (mp) and after (mn) modification and returns 1 if the modified word changes sign or reaches zero while being modified and 0 otherwise. Used mainly in MDX instructions.

where DB, FB, and displ are implicitly functions of i and `0` denotes an abstract location with constant value 0. All of these notations have state `s` and `Inst(I)` or `i` as implicit arguements unless explicitly defined.

^y	Contents of state component y.
$\delta_y(f)(x)$	Short for $y \leftarrow f(^y, x)$.
$S(x, y)$	Short for $x \leftarrow y$.

where f ranges over a few built-in operations (arithmetic and logical) and y can be any of the components of the domain of `State`.

5.3.1 LOAD/DOUBLE LOAD(LD/LDD)

- Load operations normally transfer data from core storage to the machine register specified in the instruction.

- LOAD transfers the contents of the core storage location specified by the effective address (EA) into the accumulator (A) whereas DOUBLE LOAD loads the contents of core storage specified by the EA and the content of next higher core storage location into the accumulator and its extension (Q) respectively.

- The operational semantics of LOAD and DOUBLE LOAD described in IBM-1800 manual [IBM70] can be modeled as:

If $\texttt{OpCode(i)} \in \{\texttt{LD}, \texttt{LDD}\}$,

$$\begin{aligned}[\![\texttt{i}]\!]s = &\delta_I(+)(1 + \texttt{FB}) \odot \\ &S(A, (\texttt{FB} = 0 \ ? \ \hat{}(X + \texttt{displ}) \ : \ \hat{}\texttt{loc}(X)) \\ &\odot (\texttt{DB} = 1 \ ? \ S(Q, (\texttt{FB} = 0 \ ? \\ &\hat{}(X + \texttt{displ} + 1) \ : \ \hat{}(\texttt{loc}(X) + 1))) \ : \ \mathbb{I})\end{aligned}$$

5.3.2 MODIFY INDEX AND SKIP (MDX)

- Modifies the Instruction Register (I), a specified Index Register (XR), or a core storage word.
- The modifying factor can be Displacement, the Address word, or a specified core storage word.
- It can be modeled as:

If $\texttt{OpCode(i)} \in \{\texttt{MDX}\}$,

$$\begin{aligned}[\![\texttt{i}]\!]s = &(\texttt{FB} = 1 \ ? \\ &(\texttt{tag} = 00 \ ? \ (\delta_I(+)(2 + \texttt{cmdx}((\hat{}\texttt{addr} + \texttt{displ}), \\ &\hat{}\texttt{addr})), S(\hat{}\texttt{addr}, (\hat{}\texttt{addr} + \texttt{displ}))) \\ &: \ (\delta_I(+)(2 + \texttt{cmdx}((X + \texttt{locBS(i)}), X)), S(X, (X + \texttt{locBS(i)})))) \\ &: \ (\texttt{tag} = 00 \ ? \ (\delta_I(+)(\texttt{displ})) \\ &: \ (\delta_I(+)(1 + \texttt{cmdx}((X + \texttt{displ}), X)), S(X, (X + \texttt{displ}))))\end{aligned}$$

Chapter 6

Emulator

This Chapter contains an overview of the IBM-1800 emulator components and the implementation techniques of them. Later we include an output example of the emulator in section 6.5.

6.1 Introduction

Generally, an emulator duplicates/emulates the functions or behaviour of a system with a different system in a way that the second system appears to act like the first system [Wic05]. Unlike in a simulator, it does not try to exactly reproduce the state of the machine being emulated; instead it attempts to generate the exact behaviour.

Theoretically, the Church-Turing thesis [Tur37] concludes that any computing environment can be emulated by another. Although practically, it can be quite difficult, particularly when the exact behaviour of the system is not documented well or missing and has to be extracted from the system to be emulated through reverse engineering. Timing constraints can be another issue in emulating hardware. If the emulator does not compute as fast as the original one, the software to be emulated can perform much worse than it would have on practical hardware.

Most common form of emulators is used to emulate hardware architecture. Hardware emulator is a piece of computer software that emulates the desired behaviour of hardware. Computer programs which are supposed to use that hardware architecture as underlying machine can then run on the corresponding emulator software as if they

were running on original machine. In this way, hardware emulator gives the abstraction of machine computation to the upper layered software. It does so by "emulating" or reproducing the behaviour of the corresponding hardware by accepting the same data, executing the same algorithm in computation and achieving the same results [Wic05].

Ontario Power Generation (OPG) is also using emulators for their legacy IBM-1800 hardware as those hardware are difficult to maintain and it is almost impossible to replace their failed components. They are using those emulators as a stop-gap measure for the time being before transporting their whole system into newer hardware.

As mentioned earlier, the work presented in this thesis is a part of the reverse engineering project, particularly dealing with the IBM-1800 assembler codes of OPG. The first step in our symbolic interpretation process of IBM-1800 assembler programs is to develop a model of the operational semantics (section 5.3) of almost all IBM-1800 assembler instructions. To validate the correctness of the model, we create an emulator of IBM-1800 hardware using that model as an explicit state transformer. As validation of the model is the main issue, we are not deeply concerned with the timing constraints and I/O behaviour of the hardware.

6.2 IBM-1800 Emulator

Typically, an emulator may be divided into modules that correspond roughly to the emulated computer's subsystems . Most often, an emulator contains the following modules [Wic05]:

- a CPU emulator
- a memory subsystem
- various I/O devices emulators

Emulation of I/O devices is often treated as a special case and we are more concerned with the validation of the model of the IBM-1800 operational semantics. That is why, our IBM-1800 emulator is mainly composed of a CPU emulator and a

memory subsystem with the inputs given as initial state values and produces output as final machine state.

6.2.1 Model

The model developed in Section 5.3 shows the effect of execution of an instruction on the machine components (mainly CPU registers and memory). Precisely, the model works as a state transformer in which execution of each instruction produces next state of the machine. A state is composed of the full Memory, the Instruction Register (`I`), Accumulator (`A`), Accumulator Extension Register (`Q`), all Index Registers (`XR1, XR2,XR3`) and the Overflow and Carry bits. During the emulation of IBM-1800 assembler program, we start from an initial machine state and after executing all the instructions in the assembler program, the final state is given as output of the emulator.

In the following sections, we discuss how the memory and the CPU emulator modules are implemented in a functional programming language (HASKELL). The whole code of the emulator is given in appendix B.

6.3 Memory

For the memory subsystem emulation, it is possible to implement the memory simply as an array of elements each sized like an emulated word. However, this model breaks soon as any location in the computer's logical memory does not match physical memory.

Cleary, this happens whenever the emulated hardware allows for advanced memory management (MMU). Even if the emulated computer does not feature an MMU, there are usually other factors that break the equivalence between logical and physical memory (one such feature may be memory mapped I/O). Discussing all those factors is beyond the scope of the thesis. One important advantage in implementing IBM-1800 emulator is that its memory does not have any advance memory management and features like memory mapped I/O. It is also smaller in size (4K or 8K) making it easier to handle with an array.

Our emulator implements memory as a simple array of 2^{16} entries with two simple

functions for writing to and reading from logical memory. Each entry is 16 bit long which makes the total memory array sized equal to 8K.

This module implements memory as an simple array.

```
module Mem
 (Mem, initMem,
  getMem, writeMem)
where

import Data.Word
import Data.Array

type Mem = Array Word16 Word16


initMem :: Mem
initMem = listArray (0,2^16-1) []
```

These functions are used to read and write memory contents at the effective address.

```
getMem :: Mem → Word16 → Word16
getMem m l = m!l

writeMem :: Mem → Word16 → Word16 → Mem
writeMem m l c =  m// [(l, c)]
```

At the starting point of emulation, the memory is initialized with the source assembler code.

This module initializes memory.

```
module MemInit
(  fillMem
 , updateMem
) where
```

```
import Instruction
import Lst
import Mem
import Data.Array
import Bits
```

Initially, the memory is loaded with the assembler program via reading in a `.lst` file of the source assembler program. `fillMem` fills all of memory (an array of 2^{16} entries) with the initial values.

```
fillMem :: Mem → Lst → Mem
fillMem = foldl updateMem

updateMem :: Mem → LstLine → Mem
updateMem m l = if (¬·isLong · instr) l then
                   m // [(add l, fromIntegral (bin l))]
                else
                   m // [(1+add l, fromIntegral (bin l)),
                         (add l, upper (bin l))]
                where upper = fromIntegral · (flip shiftR 16)
```

6.4 CPU Emulator

Emulation of CPU is often the most complicated part of an emulator. The simplest form of a CPU emulator is an interpreter, which follows the execution flow of the emulated program code. It interprets every machine code instruction encountered in the program control flow and executes operations of that instruction in the emulated software processor that are semantically equivalent to the original instruction's. This can be done by assigning a variable for each register, flag and memory entries of the CPU to be emulated. The logic of the CPU can then almost be directly translated into software algorithms, creating a software re-implementation that basically mirrors the original hardware logic.

In our IBM-1800 emulator, the very direct translation of the model of its operational semantics gives us the interpreter type implementation of the CPU. Each

instruction is modeled by its abstract semantics and the execution of instruction is modeled by state transformer. This "State" contains variables for each register, flag and memory in IBM-1800 machine. Each time an instruction is encountered in the program flow, the values of the variables in the state are changed by the operations (implemented in the software) induced by that instruction; resulting in a next state. After execution of all the instructions, we reach the final machine state with all the variables (symbols for machine components) assigned some final values.

In the following subsections, we show the implementation of the model of the instructions and datatypes used to represent the instructions and state.

6.4.1 Instruction

This subsection is fully taken from the Master's Thesis of Kevin Everets [Eve04]. Here we give a brief representation of the instruction architecture of the IBM-1800 assembler to make the later discussions more clear. For better understanding of parsing of the source `.lst` file, translating and manipulating of those instructions, please refer to [Eve04].

An instruction for the IBM1800 can have one of two formats: *Short* (a 16-bit instruction that contains the *Operation*, *Tag*, and *Displacement*), and *Long* (a 32-bit instruction that additionally contains the ability to do indirect addressing, conditions, and information about branching out during an interrupt).

As the instruction has two main formats (a short or a long instruction), a new data structure is made called *Instruction* which can be either a *Short* or a *Long*, with record fields to contain the different information available in each type of instruction.

For the *Long* format of the instruction, the field `disp` is redundant, since it is just a different representation of the last 8 bits of the first instruction word, i.e., the `indAdd`, `brOut` and `cond` fields. This `disp` field of the *Long* alternative is used only as displacement in the case of *MDX* instructions.

```
data Instruction = Short { op   :: Op
                         , dbit :: Bit
                         , tag  :: Tag
                         , disp :: Disp
                         }
```

```
    | Long  { op      :: Op
          , dbit    :: Bit
          , tag     :: Tag
          , indAdd  :: IndAdd
          , brOut   :: BrOut
          , cond    :: Cond
          , disp    :: Disp
          , address :: Address
          }
```

Now, we break down each piece of the *Instruction*, and give its type and meaning. First is the *Op* code, which tells us what type of instruction it is. The *Op* code is normally five bits but the fifth bit (`dbit`) is often used to select between two very similar operations (e.g., a single load vs a double load, or a "branch and skip" vs a "branch and store instruction"). Because of this, we can combine these operations into categories.

```
data Op = LD      -- Ld = Load Accum, Ldd = Double Load
        | ST      -- STO = Store Accumulator, Std = Double Store
        | LSX     -- Ldx = Load Index, Stx = Store Index
        | SLS     -- Sts = Store Status, LSs = Load Status
        | ADD     -- A = Add, Ad = Double Add
        | SUB     -- S = Subtract, Sd = Double Subtract
        | MD      -- M = Multiply, D = Divide
        | AR      -- And = Logical And, Or = Logical Or
        | EOR     -- Logical Exclusive Or
        | SFT     -- Sla = Shift Left Logical A,
                  -- Slt = Shift Left Logical A and Q,
                  -- Slca = Shift Left and Count A,
                  -- Slc = Shift Left and Count A and Q,
                  -- Sra = Shift Right Logical A,
                  -- Srt = Shift Right Logical A and Q
                  -- Rte = Rotate Right A and Qsearch bar google
        | BRANCH  -- Bsc = Branch or Skip on Condition,
```

```
                -- Bosc = Branch out of Interrupts (similar to Bsc)
                -- Bsi = Branch and Store Instruction Register
        | MDX   -- Modify Index and Skip
        | WAIT  -- Wait
        | CMP   -- Cmp = Compare, Dcm = Double Compare
        | XIO   -- Execute I/O
        | BAD   -- An invalid Op code
    deriving (Show, Eq, Ord)
```

Next is the *Tag*, for which we create a new data type to specify which of the four possible index registers (1,2,3 or none) are used in the instruction.

```
data Tag =  I | XR0 | XR1 | XR2 | XR3
            deriving (Show,Ord,Eq)
```

Here we define the *Bit* type that is used to define some of the bit fields of the instruction which is used instead of *Boolean* values as it is sometimes inconvenient to think of *Bit*s in terms of *Boolean*s.

```
data Bit = Zero | One deriving Eq
```

There are a couple of different flags used. In the instruction itself, there is one for indirect addressing (`indAdd` and one for interpreting a *BSC* instruction as a "branch out" (*BOSC*) while in an interrupt routine. All of these are interpreted as *True* if they have a bit value of `1` and *False* if they have a bit value of `0`.

```
type IndAdd = Bit
type BrOut = Bit
```

The *Disp*lacement is an 8 bit signed 2's-complement integer. It usually only exists in the short instruction (though it can also be used by the long version of the *MDX* instruction), and is most often added to the current program counter (I) to determine branch vectors or loading offsets.

```
type Disp = Int8
```

The condition bits are present in the *Long* instruction, and are most often used to modify branches. They, along with the *IndAdd* and *BrOut* flags, are also sometimes used by the *MDX* instruction as an additional *Disp* field. This would be added to the *Address* also present in the long instruction. The *Address* is a 16 bit word. The object representing the full instruction is a 32 bit word.

```
type Cond = Word8
type Address = Word16
type Object = Word32
```

A mapping is now created from the upper four bits of the opcode to the instructions. The *Op* code values are taken from the "IBM 1800 Functional Characteristics" manual. Using the upper four bits allowed for easier grouping of the function of the *Op* codes.

```
opCodeInstruction :: [(Word16, Op)]
opCodeInstruction = [(0xC000, LD)
                    ,(0xD000, ST)
                    ,(0x6000, LSX)
                    ,(0x2000, SLS)
                    ,(0x8000, ADD)
                    ,(0x9000, SUB)
                    ,(0xA000, MD)
                    ,(0xE000, AR)
                    ,(0xF000, EOR)
                    ,(0x1000, SFT)
                    ,(0x4000, BRANCH)
                    ,(0x7000, MDX)
                    ,(0x3000, WAIT)
                    ,(0xB000, CMP)
                    ,(0x0000, XIO)
                    ,(0x5000, BAD)
                    ]
```

6.4.2 State

The IBM 1800 current state includes the state of the memory (mem), the Instruction Register (ir), the Accumulator Register (acc), the Accumulator Extension Register (q), the Index Registers (xr1-3), and the Overflow and Carry Flags (overflow and carry).

```
type State = GenState Mem Word16 Word16 Bit

data GenState mem addr val bit = State
  { mem      :: mem
  , ir       :: addr
  , acc      :: val
  , q        :: val
  , xr1      :: addr
  , xr2      :: addr
  , xr3      :: addr
  , overflow :: bit
  , carry    :: bit
  }  deriving Show
```

6.4.3 Emuating Instruction Execution

Our emulator works as an interpreter and executes one instruction in each step. At the beginning, it takes a number of steps (no. of instructions in the program to be executed), a initial State (***State***) as input and returns a final State after execution of all the instructions in the assembler program.

`emulate` is the main recursive function in the emulator which emulates the steps (instructions) of the assembler program. In each iteration of `emulate`, `step` takes the current ***State*** and after executing the current instruction returns the next ***State***. It interprets every assembler instruction encountered by the semantic definition of that instruction.

`step` fetches the instruction from the memory indicated by current value of the instruction register and determines the valid opcode of that instruction. With the

opcode (*Op*) and current state (*State*), `semantics` interprets the current instruction by its semantic definition (defined by `semantics_`) and execute operations of that instruction in the emulator i.e. assigns values to various variables/components of the state to generate the next state (*State*).

```
emulate :: Int → State → State
emulate 0 s = s
emulate n s = emulate (n-1) (step s)

step :: State → State
step s = semantics inst s
    where inst = getOp (getMem (mem s) (ir s))

semantics :: Op → State → State
semantics o s = semantics_ o inst s
  where
    inst = wordsToInstruction (getMem (mem s) (ir s))
                              (getMem (mem s) ((ir s)+1))
```

`semantics_` defines the semantic interpretation of each instruction in the IBM-1800 assembler. This is a very direct translation of the abstract model of the operational semantics of IBM-1800 instructions defined at Appendix A. Here we only show two instruction semantics translated in HASKELL. The rest of the semantic definition of the instructions are given in Appendix C. Of the two instructions cited, LOAD/ DOUBLE LOAD (*LD*) is a little bit simpler while MODIFY INDEX AND SKIP (*MDX*) is more complex to interpret.

```
semantics_ :: Op → Instruction → State → State
semantics_ LD inst s =
 dIR (1+fb) $ dA (getContentOfMemRefA inst s) s
            $ if dbit inst ≡ Zero
                    then s
                    else dQ (getContentOfMemRefQ inst s) s
    where fb = fBit $ isLong inst
```

```
semantics_ MDX inst s =
 if isLong inst
    then if tag inst ≡ XRO
            then dIR (2+conAdd) $ s {mem = (writeMem (mem s)
                                        (address inst) cMemNew)}
            else dIR (2+condAdd) $ dXR (tag inst) cLocNew s
    else if tag inst ≡ I
            then s {ir = (fromIntegral $ (ir s) +
                         (fromIntegral $ displ::Word16) + 1)}
            else dIR (1+conDisp) $ dXR (tag inst) cDispNew s
 where displ = fromIntegral $ disp inst::Int16
       locL = fromIntegral $ (locBS inst s)::Int16
       cMemOld = (fromIntegral $ getMem (mem s) $ address inst::Int16)
       cMemNew = fromIntegral $ cMemOld + displ::Word16
       conAdd = retDispAdd (cMemOld+displ) cMemOld
                -- Specifically for F = 1 Tag = 00 IA = X
       cXROld = fromIntegral $ (reg1 inst s):: Int16
       cDispNew = fromIntegral $ cXROld + displ::Word16
       cLocNew = fromIntegral $ cXROld +locL::Word16
       conDisp = retDispAdd (cXROld+displ) cXROld -- For F = 0 Tag /= 00
       condAdd = retDispAdd (cXROld+locL)  cXROld -- For F = 1 Tag /= 00
```

6.5 Output Example

For the code segment (taken as an example from the OPG code) cited at Section 2.3.1, the output of the emulator (the initial state of the emulator is specified at Appendix C) will be:

```
IR = 0x35be
A = 0x0
Q = 0x0
XR1 = 0x3808
```

```
XR2 = 0x3808
XR3 = 0x0
Memory Content @IR = 0xd
Memory Content @(Address 0x3815) = 0x3815
Memory Content @(Address 0x3816) = 0x3835
```

The whole memory is implemented as an array of 2^{16} entries. In the output, we only show three entries (manually selected) to make it more precise.

Chapter 7

One Step Symbolic Interpretation

As the control flow of assembler programs is arbitrary, our first step to symbolic analysis is to find the symbolic interpretation of each instruction in a program. This chapter contains the abstract definition of the data structure used to represent each instruction in symbolic form along with the methods created for interpreting the instructions.

7.1 From Operational Semantics

Symbolic analysis [FS03] is a static and global program analysis that examines each expression of the input program only once and try to derive a precise and complete mathematical characterization of the computations.

In this case, we are dealing with the symbolic interpretation of IBM-1800 assembly language instructions. Unlike high level languages, assembly languages do not have predefined control structures in the program syntax. For this reason, we must find different control structures from the sequential instructions and then find some conditional expressions to represent different program variables in the given input code.

So we start the symbolic analysis of IBM-1800 assembly language programs by designing a one-step symbolic emulator. For all statements of the program, our one-step symbolic analysis uses exactly the same description of the semantics as in the Emulator with the components of the state being symbolic. The abstract model of the

semantics of IBM-1800 assembler instructions in Appendix A is used to implement the one step symbolic emulator. Emulator was the testing phase of this model and we finish this successfully as the emulator is working fine. Later, in Section 7.3 we present a segment of the implementation of one step symbolic emulator which shows the implementation as a direct translation of the model.

This one step symbolic emulator interprets each instruction in a program execution path and finds the symbolic representation of each instruction. This means, instead of returning a value, this step produces a representation (as an abstract data-structure) of the state-transformer which corresponds to the current instruction. In effect, each instruction will change some of the state component values (symbolic) to generate the next state. After executing each instruction, the one step symbolic emulator outputs a state transformer representation that contains expressions of the variables that are being changed by the execution of this instruction, and a symbolic path condition representation that reflects possible branching behaviour of the instruction.

The results of this analysis is a "program context" which includes the program semantics for an arbitrary program point. For one step symbolic emulation, the program point is only the current instruction. The program context [FS03] is a symbolic representation of variable values or behaviours arising at run time. Therefore, symbolic analysis can be seen as a compiler that translates a program into a different language. As the target language we employ symbolic expressions and symbolic recurrences.

A program context is defined by (p, s) that includes a path condition p and a state s.

- Path Condition p : The path condition p describes the condition under which control flow reaches a given program statement. For sequential instructions a path condition is always TRUE and for branch instructions it is specified by a logical formula that comprises the conditional expressions of branches taken to reach the program statement.

- State s : The state s is described by a set of variable/value pairs $v_1 = e_1, ..., v_k = e_k$ where v_i is a program variable and e_i a symbolic expression describing the value for $1 \leq i \leq k$. For each program variable v_i there exists exactly one pair $v_i = e_i$ in s.

For all statements of the program our symbolic analysis finds the program contexts that describes the variable values (they may be conditional or unconditional) and the conditions under which the program point is reached. In order to find the symbolic representation of each of the instructions we have implemented a local abstract interpretation that gives us a data structure containing every piece of information of the instruction executed. In the following section, we give the representation of the symbolic interpretation (i.e. how it is implemented).

7.2 Symbolic Interpretation

This module defines the data types for symbolic analysis.

```
module Symbolic
( StateComp(..), Operator(..), TypeCast(..), CondOpSm(..)
, MemRef(..), Val(..), Func(..), CondFunc(..), retCondOpSm
)
where

import OpCode
import IBM1800
import Data.Word (Word8,Word16)
import Data.Int (Int8,Int16)
import Data.Bits
```

We start with different datatype declaration needed for one-step symbolic analysis.

- **State Components:** The following is the datatype definition for the State Components though the name may be misleading. Although a state contains other values (like index registers, instruction register, memory etc.), we use the *Tag* and *MemRef* data types to avoid multiple declarations. For 32 bit operations, both Accumulator (Acc) and Accumulator Extension Register (Q) are considered as a 32-bit value. So we declare AccQ as a State Component that symbolizes the Accumulator and Q as a 32-bit value.

```
data StateComp = Acc | Q | AccQ
  deriving (Eq,Ord,Show)
```

- **Operators:** Operators symbolize the operation done by different instructions in IBM-1800 assembler. We use two different types of operators for the purpose of instructions- (1) Binary Operator and (2) Unary Operator. *Operator* datatype declares the binary operators needed for update operations in symbolic analysis whereas *TypeCast* declares the unary operators. In *TypeCast*, *Upper16* and *Lower16* are used to get the upper and lower 16 bits of a 32-bit values. *Id* means no operation and *Sign* gives the sign bit of any value. The name *TypeCast* was chosen first as the *Upper16* and *Lower16* unary operations typecasts a 32-bit value to 16-bit values.

```
data Operator = Add | Sub | Mul | Div | Mod | And | Or | Xor
              | Shl | Shr  deriving (Eq,Ord)

instance Show Operator where
    show Add = "+"
    show Sub = "-"
    show Mul = "*"
    show Mod = "%"
    show Div = "/"
    show And = "&"
    show Or  = "|"
    show Xor = "~|"
    show Shl = "<<"
    show Shr = ">>"

data TypeCast = Upper16 | Lower16 | Id | Sign
              deriving (Eq, Ord, Show)
```

- **Conditional Operator:** Conditional operators are used to interpret conditions in the branch instructions. Those branching condition can be found in IBM 1800 assembly language manual [IBM70] (See BSC/BSI).

```
data CondOpSm = EqO | LtO | GrO | LEO | GEO | NEO |
                Oe | Om | Op | En | Em | Ep | Phntl
                | Phnte deriving (Eq)

instance Show CondOpSm where
    show EqO = " == 0"
    show LtO = " < 0"
    show GrO = " > 0"
    show LEO = " <= 0"
    show GEO = " >= 0"
    show NEO = " /= 0"
    show Oe = " (Odd)"
    show Om = " (Odd and Minus)"
    show Op = " (Odd and Plus)"
    show En = " (Even)"
    show Em = " (Even and Minus)"
    show Ep = " (Even and Plus)"
    show Phntl = " "
    show Phnte = " "
```

- **Memory Components:** *MemRef* defines another important part of the state component which is the Memory Reference. Memory references can be different depending on the type of memory access in the instructions. *Const* is used to define the value in the displacement part of the instruction for direct displacement or address assignment for LDX/STX whereas *CConst* defines the content of that address for other instructions. *BrConst* is specifically used for branch instructions and defines the address composed with index registers and address content of the instruction. *Direct* defines the address that is the content of the address composed with the index registers and address content of instruction. *BrDirect* is a special memory reference only used in conjunction with BSI and defines the memory reference composed by the value of the content of index registers plus address content of instruction and then added with the offset in the BSI instruction. *Dispmnt* is also a special memory reference in

STX where memory is referenced by the content of instruction register plus the displacement of the instruction (as integer value). *Indirect* is used for indirect memory references. We could reduce the number of memory reference types by unifying them in similar types. On the other hand we want to store more information at this stage for future uses.

```
data MemRef =
    Const    {valC  :: Int16} -- for direct displacement
                              -- /address assignment(for LDX/STX).
  | CConst   {valCC :: Word16} -- for content of displacement
                               -- /address assignment.
  | BrConst  {reg   :: Tag     -- This is specially for the
             ,addrBr:: Word16  -- Branch instructions.
             }
  | BrDirect {reg   :: Tag     -- For BSI instruction, this one
             ,addr  :: Word16  -- is specially used for Indirect
             ,offBD :: Word16  -- addressing.
             }
  | Dispmnt  {reg   :: Tag     -- for displacement
             ,addrC :: Int8
             }
  | Direct   {reg   :: Tag     -- for direct address. In Branch
             ,addr  :: Word16  -- instructions, this one is specially
             }                 -- used as Direct address.
  | Indirect {reg   :: Tag     -- for indirect address.
             ,addr  :: Word16
             } deriving (Eq,Ord)
```

For *Memref* we define an instance to show it more clearly.

```
instance Show MemRef where
    show (Const a) = if (testBit a 15)
                       then "(" ++ show a ++ ")" else show a
    show (CConst a) = "C("++ show a ++ ")"
```

```
show (BrConst r a) = show r ++ "+"++show a
show (BrDirect r a o)  = "C("++ show r ++ " + "
                           ++ show a ++ ")" ++ "+"++show o
show (Dispmnt r a) = "C("++ show r ++ " + " ++ show a ++ ")"
show (Direct r a)  = "C("++ show r ++ " + " ++ show a ++ ")"
show (Indirect r a) = "C(C("++ show r ++ " + "
                                    ++ show a ++ "))"
```

- **Value:** *Val* defines the symbolic value of the expression which is composed of *StateComp* and *MemRef* which can be assigned to any state components. For 16 and 32 bit operations, two different types of *Val* are used.

```
data Val =  Val16 {val161 :: StateComp
                  ,val162 :: MemRef
                  }
         | Val32 {val321 :: StateComp
                  ,val322 :: MemRef
                  }
  deriving (Eq,Show)
```

7.2.1 Datatype for Instructions

The IBM-1800 assembly language instructions can be divided into two major class of operations:

- Assignment Instructions e.g. LOAD, DOUBLE LOAD, STORE, DOUBLE STORE etc.
- Update Instructions e.g. ADD, SUB, MUL, DIV, MDX, BSC, BSI etc. Branch instructions like MDX, BSC, BSI etc. are update instructions as they modify instruction register and in some cases one of the index registers as well.

From those two broad classes we then define the grammar that can describe all the IBM-1800 instructions. Assignment and update instructions respectively assign or update values to either state components or memory components. First we define

some auxiliary data structures that are used to define data types for assignment and update instructions. We declare the datatype for the assignments and updates of 16 and 32-bit operations of different state components (Acc, Q or memory components etc). The name of the operation defines the operation to be done. For example *AssignSC16* assigns some value (symbolic) to the 16 bit state components like Acc or Q. The following table gives a brief description of all the data structures:

Name of Data Structure	Description
AssignSC16	Assigns a 16 bit value to a 16 bit State Component.
UpdateSC16	Updates a 16 bit State Component.
AssignMem16	Assigns a 16 bit value to a memory address.
UpdateSC32	Updates 32 bit State Component(AccQ).
AssignX	Assigns a 16 bit value to one of the Index Registers.
AssignMemX	Assigns Index/Instruction Register values to a memory address.
UpdateX	Updates one of the Index Registers.
UpdateAS	Updates Accumulator during Shift operation.
UpdateAQS	Updates Accumulator and Q during Shift operation.
CondDisp	Used mainly in MDX instructions to update memory components.

The following data structures are not used for the reasons specified.

Name of Data Structure	Reason
UpdateMem16	No instructions updates memory directly.
AssignMem32	Assign 32 bit word to memory is not allowed. For 32 bit assignments its actually assigning two different 16-bit blocks at two consecutive locations in memory; we can handle it with AssignMem16.
UpdateMem32	No instructions updates memory directly.
AssignSC32	No instructions assigns 32-bit values to AccQ.
UpdateIR	We will be using control flow graphs to find the next instruction to be executed and so we did not use any data structures to update Instruction Register (IR).

```
data Func = AssignSC16 {scA16 :: StateComp
```

```
                ,valAS16 :: MemRef
                }
| UpdateSC16    {scU16   :: StateComp
                ,op16    :: Operator
                ,valUS16 :: Val
                }
| AssignMem16   {valA16  :: StateComp
                ,locA16  :: MemRef
                }
| AssignMemX    { valAX  :: Tag
                ,locAX   :: MemRef
                }

| UpdateSC32    {scU32   :: StateComp
                ,op32    :: Operator
                ,valUS32:: Val
                }
| AssignX       {conX    :: Tag
                ,valX    :: MemRef
                }
| UpdateX       {conUX   :: Tag
                ,valUX   :: MemRef
                }

| UpdateAS      {opS16   :: Operator
                ,valS16  :: Word8
                }
| UpdateAQS     {opS32   :: Operator
                ,valS32  :: Word8
                }
| CondDisp      {valD1   :: MemRef
                ,valD2   :: MemRef
                } deriving (Eq)
```

Instead of deriving Show, we create an instance, to make printing more elegant.

```
instance Show Func where
    show (AssignSC16 s v) = show s ++ " := " ++ show v
    show (UpdateSC16 s o v) = show s ++ " := " ++ show (val161 v)
                                    ++ show o ++show (val162 v)
    show (AssignMem16 s c) = show c ++ " := " ++ show s
    show (AssignMemX t c) = show c ++ " := " ++ show t
    show (AssignX t v) = show t ++ " := " ++ show v
    show (UpdateX t v) = show t ++ " += " ++ show v
    show (UpdateSC32 s o v) =  show s ++ "  = " ++ show (val321 v)
                                    ++ show o ++show (val322 v)
    show (UpdateAS o v) = "Acc " ++ show o ++ "= "++ show v
    show (UpdateAQS o v) = "AccQ " ++ show o ++ "= "++ show v
    show (CondDisp c v) = show c ++ "+=" ++ show v
```

7.2.2 Datatype for Conditions

CondFunc declares the datatype for the conditional expressions in the branching instructions. In a conditional expression, the left hand side is compared with the right hand side by a conditional operator. Here we give a brief description of the data structures that are used to define different conditions used in the instructions:

Condition defines the conditional expression for BSC/BSI instructions. For MDX, *CondDispAddT* defines the expression for index registers whereas *CondDispAddM* declares the expression for memory references. *UpdateComp* describes the conditions of Compare instructions.*Tru* is for no condition in case of sequential instructions.

```
data CondFunc = Condition     {scC    :: StateComp
                              ,opC    :: CondOpSm
                              ,stat   :: Bool
                              }
              | CondDispAddT  {scCT   :: Tag
                              ,valCT  :: MemRef
```

```
                        ,sgT    :: Bool
                        }
          | CondDispAddM {locCM   :: MemRef
                        ,valCM  :: MemRef
                        ,sgM    :: Bool
                        }
          | UpdateComp  {scCm   :: StateComp
                        ,opCm   :: CondOpSm
                        ,valCm  :: MemRef
                        }
          | Tru deriving (Eq)

instance Show CondFunc where
    show (Condition s o b) = show s ++"("++show o ++")"++"=="++ show b
    show (CondDispAddT s v o) = show s ++ "+"++ show v
                                ++ "(RZCS==" ++ show o ++ ")"
    show (CondDispAddM c v o) = show c ++ "+"++ show v
                                ++ "(RZCS==" ++ show o ++ ")"
    show (UpdateComp s o v) = show s ++ show o ++ show v
    show Tru = "True"

retCondOpSm :: BrTag → CondOpSm
retCondOpSm  bt = case bt of
                        Al  → Phntl
                        Pl  → GrO
                        Npl → LEO
                        Mn  → LtO
                        Nmn → GEO
                        Zr  → EqO
                        Nzr → NEO
                        Od  → Oe
                        Odm → Om
                        Odp → Op
                        Ev  → En
```

```
Evp → Ep
Evm → Em
Ne  → Phnte
```

7.3 Code Example

Here is a slice of code that implements the LOAD and DOUBLE LOAD instructions to perform symbolic interpretation of those instructions. The whole one step symbolic emulation code is given in Appendix D.

```
sSemantics_ :: Op → Instruction → [(CondFunc,[Func])]
sSemantics_ LD inst =
        [(Tru , (dAssignSC16 Acc memRefA) :
                 if (dbit inst ≡ Zero)
                   then []                              -- LOAD
                   else [dAssignSC16 Q memRefQ])] -- DOUBLE LOAD
 where memRefA = dMemRefA inst
       memRefQ = dMemRefQ inst
```

It is worthwhile to note that this is a very direct translation of the semantic model of the LOAD and DOUBLE LOAD instructions in Appendix A.

Next, we present implementation of another complex instruction called MDX.

```
sSemantics_ MDX inst =
 if isLong inst                                         -- MDX
      then if tag inst ≡ XRO
                then ((dCondDispAddM (CConst {valCC = address inst})
                     (Const{valC = fromIntegral $ disp inst}) True)
                     ,[dCondDisp (CConst {valCC = address inst})
                     (Const{valC = fromIntegral $ disp inst})])
                     :[((dCondDispAddM (CConst {valCC = address inst})
                     (Const{valC = fromIntegral $ disp inst}) False)
                     ,[dCondDisp (CConst {valCC = address inst })
                     (Const{valC = fromIntegral $ disp inst})])]
```

```
            else ((dCondDispAddT (tag inst) dXMem True)
                 ,[dUpdateX (tag inst) dXMem])
                 :[((dCondDispAddT (tag inst) dXMem False)
                 ,[dUpdateX (tag inst) dXMem])]
     else if tag inst ≡ I
            then [(Tru,[])]
            else ((dCondDispAddT (tag inst) dXMem True)
                 ,[dUpdateX (tag inst)
                  (Const{valC = fromIntegral $ disp inst::Int16})])
                 :[((dCondDispAddT (tag inst) dXMem False)
                 ,[dUpdateX (tag inst)
                 (Const{valC = fromIntegral $ disp inst::Int16})])]
   where dXMem = mdxMemRef inst
```

7.4 Output Example

As an example, take the instruction LD 1 41 (Opcode : 0xC129) , the output of the one step symbolic interpreter will be:

```
(True, [A := C(XR1+41)])
```

As well the MDX instructions (Opcode : 0x7500 000D) will produce the following output:

```
(XR1+13(RZCS) == True, [IR += 3, XR1 += 13])
(XR1+13(RZCS) == False, [IR += 2, XR1 += 13])

Here RZCS stands for Reaches Zero or Changes Sign.
```

Chapter 8

Multi Step Symbolic Interpretation

This chapter details the method used for generating Data Flow Equations (DFE) for an assembler program. The functions for interpreting the assembler codes into symbolic DFEs are discussed. Later, examples of DFEs for different types of program segments are provided.

8.1 Introduction

An assembler program is written as a sequence of instructions. The ordering of the instructions as written defines a default sequence of execution [WF03]. Branch instructions may however interrupt the flow and cause control to be transferred elsewhere. This differs from high level languages where all non-linear control flow is encapsulated in the semantics of compound instructions like for or while loops. In assembler programs control flow can be arbitrary.

Consequently, a major challenge to symbolic analysis of assembly language programs is how to handle control flow. Fortunately for us the control flow graph for the list of assembly language instruction to be analyzed is already provided. This control flow graph gives us some perception about how to find the control structures in the list of assembly instructions. Some tools created by a fellow student [Eve04] produces the control flow graph of the codes given and we use that control flow graph to determine the control flow of the instructions.

The approach taken here to model our control flow is to model a program by its set

of execution paths. A program has a distinguished starting point and its execution path is a sequence of instructions that can possibly be executed during some run of the program. Execution always begins at the starting point. Paths are always maximal that is a path is only completed when the program terminates.

In the following sections, we describe the structure of the control flow graph and also how we used that control flow graph for symbolic interpretation of a chunk of assembly language instructions.

8.2 Control Flow Graph

The tools to generate approximated control flow graph (CFG) produce a CFG for a list of assembly language instructions. This program proceeds by only looking at updates to `Instruction Register` and selected memory locations (as used by branching instructions) to approximate the control flow. In general, this approximation is very good, but in a few cases where the code is self-modifying, however, this automated step fails. This is not necessarily a problem since we can also provide hand-written or hand-corrected CFGs as input to the next step.

For each instruction to be executed it creates a node in the graph. Then by traversing the list of instructions, it finds all the possible next addresses from each instruction and creates edges to those nodes from the node of the current instruction. A node contains various information such as its address, the stored opcode, any available textual labels, and the path-condition (to be defined later) of the corresponding instruction. See [Eve04] for more details. Output of the resulting control flow graph is done via GXL (Graph eXchange Language) [Win01], so that standard tools may be re-used to manipulate and display these graphs.

8.2.1 Internal Data Structure of CFG

The GXL is an exchange format and contains more information than we need. Sometimes handling all those information is more difficult than it should be. In order to remedy this problem, we create our own internal graph data structure for the control flow graph from the GXL format that contains only the necessary information for symbolic interpretation of the code.

This module defines the internal data structure for the CFG and also implements important functions to convert GXL represented CFGs into the internal data structure.

```
module MyGraph
( NER, MyNode, AnnotationChoice(..), MyEdge(..), MyGraph
, gxlToMyGraph, doAnnotation, gxlAttrToString
, gxlAttrToBool, isNode, isEdge
)
where

import Instruction
import Symbolic ( CondFunc(Tru), Func())
import OneStep (sSemantics_)
import MyPrelude (readHex')
import qualified Gxl
import Text.XML.HaXml.OneOfN
import Data.List (nub)
import Data.FiniteMap (FiniteMap, emptyFM, fmToList, addListToFM,
                       mapFM, lookupFM)
import Data.Maybe (fromJust)
```

NER below is a given type name for standard GXL Node, Edge, Relation type.

```
type NER = OneOf3 Gxl.Node Gxl.Edge Gxl.Rel
```

As the name suggests the internal data structure of the control flow graph will contain the nodes and the edges. Below are the data type declarations of the nodes and edges of the graph.

A node in *MyGraph* is given by a string: the node name. Each instruction in the code will have a node and the branch that can be created by the opcode of that instruction will be represented by an edge from that node. An Edge is a structure which is comprised of the starting node, ending node of that edge and some other pre-defined attributes. We describe the pre-defined attributes below.

If the address of the starting node is greater than that of the ending node for an edge then the edge is considered as back edge and marked using the `backEdge` attribute of the edge structure. This attribute will be useful later for loop structure recognition.

As we mentioned earlier, each edge from one node will be generated depending on some conditions like True, False etc. *AnnotationChoice* defines that condition on which the edge is generated.

The opcode of the instruction in each node is symbolically interpreted using the one step symbolic emulator. This one step symbolic emulator gives us the meaning of what the instruction does in terms of some symbols. We then combine the interpretations of all the instructions in a code segment to understand what the code does. To understand the meaning of the code, the use of Control Flow Graph is required. So we annotate the edges of the CFG with the symbolic interpretation of instructions in each node. `annotation` field in each edge structure contains the interpretation of the instruction in the node from where the edge is generated. `annotChoice` helps us to annotate the edges with proper interpretation.

```
type MyNode = String

data AnnotationChoice = ATrue | AFalse | ALess | AEqual | AGreater
                      | AVoid deriving (Eq)

instance Show AnnotationChoice where
     show ATrue = "True"
     show AFalse = "False"
     show ALess = "Less"
     show AEqual = "Eq"
     show AGreater = "Gt"
     show AVoid = "Vd"


data MyEdge = MyEdge { edgeFrom :: MyNode
                     ,annotation:: [(CondFunc,[Func])]
                     ,annotChoice:: AnnotationChoice
```

```
                ,backEdge:: Bool
                ,edgeTo :: MyNode
              } deriving (Eq)
```

The show instances defined for the edge structure, *MyGraph* etc. are only for pretty printing. The purpose is to be able to to view the data structure in a nicer fashion. It is not intended to generate XML or some other data format. The output of this pretty printing is not used in other tools.

```
instance Show MyEdge where
        show (MyEdge ef a ac be et )
               = "<EdgeFrom =" ++ show ef ++",Annotation:"
                 ++show a++",AnnotChoice:"++ show ac++",backedge:"
                 ++ show be++",EdgeTo = " ++ show et ++">\n"
```

Node and Edge types for the internal Data Structure of the CFG are defined. We can now create the data structure of the graph. *MyGraph* is a data type that contains two finitemaps. One maps each node with the opcode (either 16 or 32-bit) of the corresponding instruction and the other is a mapping from a node to list of possible edges generated from that node.

We use two finitemaps as we want to distinguish between two mappings without combining *Object* and [*MyEdge*] in a single unmeaningful structure . We could possibly add a pair (*Object*, [*MyEdge*]) and could use a mapping from *MyNode* to (*Object*, [*MyEdge*]) but the Haskell declaration of finitemap does not allow us to do so.

```
data MyGraph = MyGraph (FiniteMap MyNode Object)
                       (FiniteMap MyNode [MyEdge])

instance Show MyGraph where
    show (MyGraph fmo fme) = concat $ foo fmo $ fmToList fme
      where
        foo fmo l = map (λ (key,edg)
                        → "\n<nodeID ="
```

```
                    ++ show key
                    ++ ",OpCode = "
                    ++ show (fromJust $ lookupFM fmo key)
                    ++">\n"
                     ++ (concat $ map show edg)) l
```

Generating Internal Data Structure of the CFG:

These are the main functions where we take a GXL data structure and convert it into *MyGraph*. From the GXL input we get the nodes and the possible edges in the CFG and add them to the *MyGraph* data structure. `nodesToMyNodes` and `edgeListFromNodes` are two important functions used in `gxlGraphToMyGraph` to get the nodes and edges in the CFG.

One interesting point to note is that we use a *Int* argument named `sv` in these functions. In the GXL input of the graph, the node names are structured like `filename ++ "-"++address`. Here `filename` is the name of the input .lst file of the code from which the GXL graph is generated and `address` is the relative address of the instruction in the .lst file. So for a code file name `test.lst`, one instruction with relative address `35b6` will have a node named `test-35b6` in the GXL file. In our data structure we want the nodes named only by the addresses. So we get rid of the `filename` part using this `sv` field which is calculated from the input filename earlier.

```
gxlToMyGraph :: Gxl.Gxl  → Int → MyGraph
gxlToMyGraph (Gxl.Gxl _ (g:gs)) sv = gxlGraphToMyGraph g sv

gxlGraphToMyGraph :: Gxl.Graph → Int → MyGraph
gxlGraphToMyGraph (Gxl.Graph _ _ _ ners) sv
            = MyGraph nodesFM edgesFM
  where myNodes = nodesToMyNodes (nersToNodes ners) sv
        nodesFM = addListToFM emptyFM myNodes
        myEdges = filter (λx → snd x ≠ [])
                    $ edgeListFromNodes ners sv $ map fst myNodes
        edgesFM = addListToFM emptyFM myEdges
```

To get information about the nodes, we need to to find all of them in the GXL graph. **nersToNodes** is an iterative function which finds all the GXL nodes using the Node,Edge,Relation data from the GXL graph.

nodesTOMyNodes makes a list of pairs of *MyNode* and *Object* (opcode of the corresponding instruction in *MyNode*) from a list of GXL nodes. It uses the **nodeToMyNode** function to extract all information needed from one GXL node. **nodeToMyNode** gets node name (minus the **filename** part of the GXL node name) of a node and the opcode of the instruction of that node.

```
nersToNodes :: [NER] → [Gxl.Node]
nersToNodes [] = []
nersToNodes  ((OneOf3 n):ners) = n:nersToNodes ners
nersToNodes ((TwoOf3 _):ners) = nersToNodes ners
nersToNodes ((ThreeOf3 _):ners) = nersToNodes ners

nodeToMyNode :: Int → Gxl.Node → (MyNode,Object)
nodeToMyNode sv n@(Gxl.Node nas _ ndAtt _) = (nID, opc)
        where nID = drop sv (Gxl.nodeId nas)
              opc = fromIntegral $
                    readHex'(gxlAttrToString "binary" ndAtt)::Object

nodesToMyNodes :: [Gxl.Node] → Int → [(MyNode,Object)]
nodesToMyNodes nodes sv = map (nodeToMyNode sv) nodes
```

In this part, we convert the Gxl edges into *MyEdges*. The strategy is to take a list of nodes in the graph and then find all the possible edges from those nodes. All the edge attributes (except the annotation) are then added to the possible edges.

edgeListFromNodes finds a list of pairs of *MyNode* and possible edge list from that node for the list of nodes in the graph using *NER* of the GXL graph. This list of pairs can be added directly in the node to edge list finitemap of *MyGraph*. **edgesFromNode** is an helping function of **edgeListFromNodes** to find all the edges from one node. **edgesFromNode** uses **gxlEdgeToMyGraphEdge** to add all the attributes (except the annotation) of those edges. **gxlEdgeToMyGraphEdge** finds all the needed attribute of

MyEdge using the [*Gxl.Attr*] of the GXL edge. The annotChoice attribute added by this function will be used later to do the annotation of the edges.

```
gxlEdgeToMyGraphEdge :: [Gxl.Attr]→ MyNode → MyNode → MyEdge
gxlEdgeToMyGraphEdge attr ef et = MyEdge {edgeFrom = ef,
                                          annotation = [],
                                          annotChoice = annotate,
                                          backEdge = be,
                                          edgeTo = et}
    where annotate = findAnnotate $ gxlAttrToString "condition" attr
          be = gxlAttrToBool "backedge" attr

edgesFromNode :: [NER] → Int → MyNode → [MyEdge]
edgesFromNode ners sv start = nub [edgeToMyEdge e | e ∈ ners,
                                   isEdge e, edgeFromN start e]
    where edgeFromN  start (TwoOf3 (Gxl.Edge eas _ _ _))
              = ((drop sv $ Gxl.edgeFrom eas) ≡ start)
          edgeToMyEdge (TwoOf3 (Gxl.Edge eas _ att _))
              = gxlEdgeToMyGraphEdge att start
                                     (drop sv $ Gxl.edgeTo eas)

edgeListFromNodes :: [NER] → Int → [MyNode] → [(MyNode,[MyEdge])]
edgeListFromNodes ners sv = map (λmn → (mn,edgesFromNode ners sv mn))
```

8.3 Marked-up Control Flow Graph

Given a control flow graph we want to use the results of the one-step symbolic interpreter to mark-up the edges of the graph with the symbolic representation of the state-transformer corresponding to that edge. The main difficulty is that while the complete interpretation of an instruction can be done symbolically, this cannot be done for even medium sized programs because the resulting output would be so large as to be deprived of use.

Another aspect to consider is that we are only really interested in the semantics

of larger chunks of programs which hopefully correspond to natural functions. These larger chunks invariably contain conditionals and thus it makes no sense to interpret the meaning of one branch of the conditional in a context which does not include the reason why this particular branch was chosen. That is the truth-value of the boolean condition that caused the program to choose that particular branch.

These two aspects have a common remedy. Inspired by [FS03] where similar techniques are used for (very) high-level programs, we define a program context (Section 7.1) to be $[p, s]$ where p is a path condition and s is a state.

The path condition p describes the condition under which control flow reaches a given program statement from a given starting point. Every instruction induces a condition under which each outgoing edge in the control flow graph is followed.

For sequential instructions, this condition is just TRUE, and for branch instructions this condition is a logical formula that encodes the condition expressed by the operational semantics. The path condition at a particular node is the disjunction of all the path conditions of the ingoing edges of that node. The path condition along an outgoing edge is the conjunction of the path condition at the source node and the path condition given by the one-step symbolic emulator.

We use a combinator which weaves a Graph Walker (in our case a depth-first graph traversal) with the one-step symbolic emulator to produce a new function which, given a CFG, will return a marked-up CFG with the edges labeled by a program context.

`doAnnotation` is a Graph Walker function which visits all the edges of the graph and annotates each edge with the proper symbolic interpretation of the instruction related with its starting node. It uses `appAnnotation` to find the appropriate symbolic meaning of the instruction for that edge. In `appAnnotation`, we use the function from One Step Symbolic Emulator (`sSemantics_`) to find the symbolic interpretation of the instruction opcode and then `findPropAnnotate` determines the proper annotation choice.

```
doAnnotation :: MyGraph → MyGraph
doAnnotation (MyGraph fmo fme) = MyGraph fmo fmee
  where  fmee = mapFM (findAnt) fme
         findAnt mn = map (appAnnotation (fromJust $ lookupFM fmo mn))

appAnnotation :: Object → MyEdge → MyEdge
```

```
appAnnotation obj me = me {annotation = ant}
  where  inst = binaryToInstruction obj
         opc = op inst
         ant = findPropAnnotate (sSemantics_ opc inst)
                                (annotChoice me)
```

As mentioned earlier each edge will be annotated depending on the condition of the corresponding edge. This function determines which symbolic interpretation is to be added as the annotation of the edge depending on the opcode and the condition, that is **annotChoice** of the edge. The one-step symbolic emulator produces the list of pairs of conditions and symbolic interpretation of the instruction in a defined manner. So we can easily use the **take** and **drop** Haskell functions to find the proper annotation for each possible condition. Looking up the corresponding **condFunc** is not needed for the predefined ordering of the list.

```
findPropAnnotate :: [(CondFunc,[Func])] → AnnotationChoice
                                → [(CondFunc,[Func])]
findPropAnnotate funcs ATrue  = take 1 funcs
findPropAnnotate funcs AFalse = drop 1 funcs
findPropAnnotate funcs AEqual = take 1 funcs
findPropAnnotate funcs ALess  = take 1 (drop 1 funcs)
findPropAnnotate funcs AGreater = drop 2 funcs
findPropAnnotate funcs AVoid = [(Tru,[])]
```

Below are functions which prove useful in the conversion.

We have to use some helper function to interpret the GXL format data in Haskell format (String, Bool etc.) **gxlAttrToString** and **gxlAttrToBool** are two such kind of functions which find the value of an attribute and convert it into Haskell format.

```
gxlAttrToString :: String → [Gxl.Attr] → String
gxlAttrToString at [] = ""
gxlAttrToString at ((Gxl.Attr attrattr _ _ value):attrs)
                          = if (Gxl.attrName attrattr ≡ at)
                               then (toString value)
```

```
                                    else gxlAttrToString at attrs
                          where toString (FiveOf10 (Gxl.GxlString s)) = s
                                toString _ = ""

gxlAttrToBool :: String → [Gxl.Attr] → Bool
gxlAttrToBool at [] = False
gxlAttrToBool at ((Gxl.Attr attrattr _ _ value):attrs)
                          = if (Gxl.attrName attrattr ≡ at)
                                 then (toBool value)
                                 else gxlAttrToBool at attrs
                          where toBool (TwoOf10 (Gxl.Bool s)) =
                                       if (s ≡ "True") then True
                                                       else False
                                toBool _ = False
```

`findAnnotate` is a small helper function that is used to determine the condition of the edge. In other words, it finds `annotChoice` depending on the `condition` attribute of the GXL edges. This will later be needed to annotate the edges using the one step symbolic emulator.

```
findAnnotate :: String → AnnotationChoice
findAnnotate "True" = ATrue
findAnnotate "False" = AFalse
findAnnotate "==" = AEqual
findAnnotate "&lt;" = ALess
findAnnotate "&gt;" = AGreater
findAnnotate _ = AVoid
```

Functions to determine the identity of GXL nodes or edges.

```
isNode, isEdge :: NER → Bool
isNode (OneOf3 _) = True
isNode _ = False
```

```
isEdge (TwoOf3 _) = True
isEdge _ = False
```

8.4 Data Flow Equations

In a modern high level language non-sequential control flow is encapsulated in a small number of statements that implement variations on the control flow patterns of iteration and alternation. In an assembler program there are no restrictions on the control flow patterns that may be used by the programmer. We therefore need a more general mechanism to describe control flow. As was shown in the earlier section we use set of execution paths to model the control flow of the program.

Representation of Data Flow Equations (DFE) does not need so much of information that is contained in the data structures from one-step symbolic emulator. So we unify all the instruction data structures from the one-step symbolic emulator in some simplified data structures. Here we define the datatype that describes all the instructions in a unified format.

8.4.1 Datatype Definition

This module defines all the important data structures to find the expression for an instruction in IBM-1800 assembler and to evaluate the expressions in a statement. A statement is used to represent a function of an instruction in the assembler code.

```
module Exp
( StateRef(..), BasicExp(..), Expr, Expression, bToE
, eToE, Stmt, Recur_Stmt, Cond(..), BrType(..)
, ConditionExp(..), ConditionStmt
)
where

import Symbolic (StateComp, MemRef, TypeCast(..), Operator,
                 CondOpSm)
import OpCode (Tag)
import Data.Word (Word8)
```

All the instructions in IBM-1800 assembly language perform a similar action. They assign some values to some variables. we can therefore consider all the statements in this assembly language as assignment operations. That is why we define a statement as a pair of State Reference and Expression. A statement represents one operation of an instruction.

- **State Reference:** A state reference is a variable to which we can assign values as expressions. For this assembly language state reference may be either State Components (Accumulator, Q register, Index registers) or Memory Components. Here we define a datatype for State References in the instructions. *StateComp* can be of type *Acc*, *Q* and *AccQ*. *Tag* is used to represent Index Registers and *MemRef* is for memory references which are defined earlier.

```
data StateRef = SC StateComp | SCX Tag | Mem MemRef
                                        deriving (Eq, Ord)

instance Show StateRef where
    show (SC sc) = show sc
    show (SCX s) = show s
    show (Mem mr) = show mr
```

By using *SC StateComp* in the declaration, we raise the following questions: How do you handle the ambiguity between (*SC Acc*, *SC Q*) and *SC AccQ*? Why is *SC AccQ* used at all?

In some cases, we need distinct values of *Acc* or *Q* after they are modified by an operation on whole *AccQ*. Each time *AccQ* is changed by one instruction, we create two more redundant entries of *Acc* and *Q* on purpose for future instruction references. Those redundant entries will be removed by some kind of garbage collection operation.

We could have reconstructed *AccQ* from *Acc* and *Q* values. However due to the semantics of IBM-1800 assembler language, the operations on the *AccQ* consider it as a 32 bit number. To stay closer to the semantics of those instructions we have created a state component named *SC AccQ*.

- **Expression:** An expression is what can be assigned to a State Reference.

  The datatype for Expression (*Exp*) is defined in the following way:

```
data Exp =  Constant Word8
          | MemoryConstant MemRef
          | Variable StateComp
          | VariableX Tag
          | UnaryOperation (TypeCast,Exp)
          | BinaryOperation (Exp,Operator,Exp)
          | ConditionalValue ((CondFunc,Exp),(CondFunc,Exp))
           deriving (Eq)
```

  But as new tools were developed for solving Data Flow Equation (DFE)s (See Chapter 10) and generating Data Flow Graph (DFG)s (See Chapter 9) and trying to unify all the data types to share the same base *Exp* datatype, we quickly realized this would not be suitable. Then, we change the definition of *Exp* in the following way: the Expression datatype which has both recursive and non-recursive version and also has a base type called *BasicExp* which can be used for basic operation symbols. The non-recursive datatype of Expression is used in findind DFEs. *BasicExp* is used in generating DFGs and the recursive datatype for Expression is used in solving DFEs.

  Below is shown the datatypes for the Expression which are recursive and non-recursive.

  The *BasicExp* data structure defines all the unit expressions in the instructions of IBM-1800 assembly language. The *Constant* alternative is used to represent the constant values that are always defined as part of an instruction, also known as displacement. The maximal size of a constant is 256 (8- bit). *MemoryConstant* defines a memory reference that is to be read or written by an instruction. The name symbolizes that a value is always to be read or written in memory. *Variable* is used to define the expressions which contain state components and *VariableX* is used for instruction and index registers. *SignBit* is a special Constructor in *BasicExp* only used for MDX instructions to get the sign bit of index registers.

```
data BasicExp = Constant Word8
              | MemoryConstant MemRef
              | Variable StateComp
              | VariableX Tag
              | SignBit Tag deriving (Eq, Ord)

instance Show BasicExp where
    show (Constant a) = show a
    show (MemoryConstant a) = show a
    show (Variable a) = show a
    show (VariableX a) = show a
    show (SignBit a) = "Sign"
```

The *UnaryOperation* and *BinaryOperation* data structure define all the unary and binary operations that are being executed by the instructions. The *ConditionalExp* defines the data structure for the conditional values. For conditional values we assume that there can be only two types of expressions (one for *True* and the other for *False*). The *ConditionStmt*s in *ConditionalExp* have an invariant (condition). Later this code can be refactored as *ConditionalExp* (*ConditionStmt*, a, b) where a is the expression for *True* and b for *False* condition.

```
data Exp2 a b   =  UnaryOperation (TypeCast,b)
                | BinaryOperation (b,Operator,b)
                | ConditionalExp ((ConditionStmt,b),
                                  (ConditionStmt,b))
                | Atomic a
                deriving (Eq)
```

Special show instances of the non recursive expression data structure is declared for better printing purposes.

```
type Show' a = a → String
```

```
showExp2 :: Show' a → Show' b → Show' (Exp2 b a)
showExp2 showA showB (BinaryOperation (ex1,op,ex2)) =
        "("++showA ex1 ++")"++ show op ++ showA ex2
showExp2 showA showB (UnaryOperation (op,ex1)) =
        if op ≡ Id then showA ex1
        else show op ++ "("++ showA ex1 ++ ")"
showExp2 showA showB (ConditionalExp ((cf1,ex1),(cf2,ex2))) =
        showA ex1 ++ "(" ++ show cf1 ++ "):"
        ++ showA ex2 ++ "(" ++ show cf2 ++ ")"
showExp2 showA showB (Atomic ex1) = showB ex1


instance (Show b, Show a) => Show (Exp2 b a) where
  show = showExp2 show show
```

This one is a non-recursive version of expression data structure. We use it to generate the expression part of the statements (in other words DFEs) of the IBM-1800 assembler code.

```
type Expr = Exp2 BasicExp BasicExp
```

There is no recursion here. Its only a wrapper around an *existing* type.

The following recursive version of the expression data structure will be used to solve the expressions in statements.

```
data Expression = Expression (Exp2 BasicExp Expression)
                    deriving (Eq)

showExpression (Expression e) = showExp2 showExpression show e

instance Show Expression where
    show = showExpression
```

Special *Functor* instances to convert *Expr* into *Expression*.

```
instance Functor (Exp2 a) where
 fmap f (UnaryOperation (t,b)) =  UnaryOperation (t, (f b))
 fmap f (BinaryOperation (b1,op,b2))
                        = BinaryOperation ((f b1),op,(f b2))
 fmap f (ConditionalExp ((cd1,b1),(cd2,b2)))
             = ConditionalExp ((cd1, (f b1)),(cd2, (f b2)))
 fmap f (Atomic b) = Atomic b

bToE :: BasicExp → Expression
bToE = Expression· Atomic

eToE :: Expr → Expression
eToE = Expression· fmap bToE
```

The *Stmt* datatype is used for each instruction of the assembler program. Left hand side of the *Stmt* is a State Reference *StateRef* and the right hand side is an Expression (*Expr*).

```
data Stmt
  =  Assign (StateRef,Expr) deriving (Eq)

instance Show Stmt where
   show (Assign (s,ex)) = show s ++ " = " ++ show ex
```

This one is a recursive version of statement which will be used in solving the data flow equations.

```
data Recur_Stmt
     = Assignt (StateRef, Expression) deriving (Eq)

instance Show Recur_Stmt where
   show (Assignt (s, exr)) = show s ++ " = " ++ show exr
```

We also unify the conditions in the same data structure using *BasicExp* and *Expr*. Since there are no conditions in the sequential instructions, the condition for them is represented as having no condition and the condition for the branches is represented as a branch condition.

ConditionStmt declares the conditional statement associated with the instructions. It contains two parts - *ConditionExp* and *Cond*. *ConditionExp* defines the condition to be checked. It is an expression which represents the condition of different state components. *NoCondition* in *ConditionExp* is used for sequential instructions whereas *BrCondition* is used for branching instructions. In *BrCondition*, *Expr* determines left-hand side of a conditional expression, *BasicExp* defines the right-hand side and *CondOpSm* is used to represent the operator used to compare those two expressions. *BrType* distinguishes among different types of branches in the assembler language.

Cond defines different values of a conditional expression (defined by *ConditionExp*). As it happens in conditions of an instruction, in *ConditionStmt*, we check symbolically the *ConditionExp* with its value i.e. *Cond*.

```
data Cond = Eql | Ltn | Gtr | Tr | Fl | NC deriving (Eq)

instance Show Cond where
  show Eql = "=="
  show Ltn = "<"
  show Gtr = ">"
  show Tr = "True"
  show Fl = "False"
  show NC = ""

data BrType = BR | MDX | CMP deriving (Eq)

data ConditionExp =  BrCondition Expr CondOpSm BasicExp BrType
                   | NoCondition deriving (Eq)

instance Show ConditionExp where
  show (BrCondition e op be bt) =
```

```
                case bt of
                      BR → show e ++ show op
                      MDX → "Sign1 <> Sign2 || "++ show be
                                                ++ show op
                      CMP → show e ++" CMP " ++ show be
  show NoCondition = ""

data ConditionStmt = Check (ConditionExp,Cond) deriving (Eq)

instance Show ConditionStmt where
    show (Check (cde@(BrCondition e op be bt), cd)) =
                    " ("++show cde++") " ++"=="++ show cd
    show (Check (cde@(NoCondition), cd)) = "True"
```

8.5 Modeling Control Flow

Our strategy for the semantic modeling of assembler program is as follows. From the internal data structure of the control flow graph we find the set of possible paths from the starting point in that program. For each path we get all the instructions and convert them into statement data structure as defined in the previous section. Then we evaluate sequentially all the expressions from the starting point using the stack and find the symbolic inputs, outputs and system of equations (relation between inputs and outputs) on that path. The semantics of a program is then the disjunction of semantics of all possible paths through the program.

If the given program is sequential, then only one path of execution can be found and we just compose all execution of instructions to find the program semantics. The modeling of non-sequential control flow is more complex. We therefore divide the modeling of IBM-1800 assembly language programs into two parts:

- Finding paths in the control flow graph.
- Finding data flow equations for those paths.

8.5.1 Finding Paths in the Control Flow Graph

In this Module we write some functions to find all the possible paths of a graph. We can consider the graph to be cyclic.

```
module FindPath
( PathType(..), Path, FinalPath, nodesFromStart, edgesOfGraph
, findEdgeAnnt, annotationOfPaths, findLoop, nodesFromHere
, nodesFromEdges, makePair
)
where

import MyGraph (MyGraph(..), MyNode, MyEdge, edgeTo, edgeFrom
                , annotation)
import Symbolic (Func, CondFunc)
import Data.List (partition)
import Data.FiniteMap ( eltsFM, lookupFM )

import Observe
```

After finding all the paths the ***PathType*** field associated with the ***FinalPath*** gives us the type of the path. A path can be terminating or cyclic. ***PathType*** defines paths as either terminating (*Term*) or looping (*Loop*).

A path is a list of nodes where two consecutive nodes have an edge in the path. Instead of list of edges, we use list of nodes to represent a path. This is helpful in finding execution paths of a control flow graph using general searching techinques like BFS (Breadth First Search). Also from the list of nodes, we can easily form the list of edges (if needed) in the path.

```
data PathType = Term | Loop deriving (Eq, Show)

type Path = [MyNode]

type FinalPath = (PathType,Path)
```

The strategy to find all the possible paths in the control flow graph is to start from a node (that is designated as a start node), and expand the paths forward from that node in the directed graph. In `nodesFromStart` we take a node as a start node and find all the possible paths from that node. It is like a breadth first expanding of the graph from the start node.

```
nodesFromStart :: MyGraph → MyNode → [FinalPath]
nodesFromStart g n = fst $ extendPath g n
```

`extendPath` is the main function to find all the possible paths in the graph. It takes a (complete paths, paths to continue) pair and extends the paths to continue by one node, whenever possible. By complete paths we mean those paths that either have a dead end (from the last node of those paths there are no outgoing edges) or for which the last node in the path is the repetition of one of its previous nodes; this happens in looping paths. In each iteration of `extendPath1`, we increase all the paths to continue by one node if possible. If one path hits a dead end then it is transferred to the complete paths list with the attribute *Term* signaling it as a terminating path. It is otherwise added to the paths-to-continue list for the next iteration of `extendPath1`. After this addition to the same iteration, we again check the paths to continue to find whether the last node in a path is the same as one of its previous node. If any path of that kind is found, we transfer it to the complete path list with the attribute *Loop* denoting it as a looping path. When the paths-to-continue list is empty this iterative path-finding function ends.

```
extendPath :: MyGraph → MyNode → ([FinalPath], [Path] )
extendPath g n = extendPath1 g ([], [[n]])

extendPath1 :: MyGraph → ([FinalPath], [Path]) → ([FinalPath], [Path])
extendPath1 _ (a,[]) = (a,[])
extendPath1 g (done, todo) = extendPath1 g (newdonelist ++ done,
                                            extended)
    where edgesOut = map (λln → (nodesFromHere g $ head ln, ln)) todo
          (almostdone, tocomplete) = partition (λn → fst n ≡ Nothing)
                                               edgesOut
```

```
        ndlist = map reverse $ map snd almostdone
        ndlist1 = map zipT ndlist
        expand (Just edgl, nodl) = map (λn → n:nodl) edgl
        expand (Nothing, nodl) = error "should not happen, ever!"
        extd1 = concat $ map expand tocomplete
        extd2 = partition (λx → fst x ≡ Loop) (map findLoop extd1)
        extended = map snd (snd extd2)
        loops = map zipL $ map reverse $ map snd $ fst extd2
        newdonelist = ndlist1 ++ loops

zipT nds = (Term, nds)
zipL nds = (Loop, nds)
```

To find the interpretation of the instructions in a path we obtain all the annotations of the edges in that path. Then we evaluate all the annotations from the starting node sequentially to determine the meaning of the instructions of the code in that path.

`annotationOfPaths` finds a list of all the edge annotations from the starting node in a path. It uses `findEdgeAnnt` to find the annotation of one edge and `edgesOfGraph` to find all the edges in a graph. All the nodes of a path are organized in a list of (starting node, ending node) pairs of the edges.

```
edgesOfGraph :: MyGraph → [MyEdge]
edgesOfGraph (MyGraph mgo mgs) = concat $ eltsFM mgs

findEdgeAnnt :: (MyNode, MyNode) → [MyEdge] → [(CondFunc,[Func])]
findEdgeAnnt (x,y) mes = concat [ annotation me |me ∈ mes,
                                                 isEdgeFrom me x ,
                                                 isEdgeTo me y]
        where isEdgeFrom e st = (edgeFrom e ≡ st)
              isEdgeTo   e en = (edgeTo e ≡ en)

annotationOfPaths::MyGraph → [(MyNode,MyNode)] → [(CondFunc,[Func])]
annotationOfPaths mg els = concat $ map findAnntOfEdge els
```

```
        where edgs = edgesOfGraph mg
              findAnntOfEdge el = findEdgeAnnt el edgs
```

This function is used to find the path attribute whether it is looping or terminating.

```
findLoop:: Path → (PathType,Path)
findLoop mnds = if (head mnds) `elem` (tail mnds) then (Loop, mnds)
                                                  else (Term, mnds)
```

As we saw in **extendPath1**, we have to find all the next possible nodes from a node in *MyGraph*. By next possible nodes we mean all the end nodes of the edges generated from the current node. In **nodesFromHere**, a simple lookup in the finitemap from the node to edgelist of the *MyGraph* data structure will work to find the all possible next nodes from the current node.

```
nodesFromHere :: MyGraph → MyNode → Maybe Path
nodesFromHere (MyGraph no ne) = nodesFromEdges · lookupFM ne

nodesFromEdges :: Maybe [MyEdge] → Maybe Path
nodesFromEdges = fmap (map edgeTo)
```

Finding all the paths in the CFG is done in the previous functions. Now we introduce some other functions to find the symbolic interpretation of the instructions in the paths of the CFG.

This function can be used to organize a node list into pairs so that the first node is the starting node and the last one is the ending node of one edge.

```
makePair:: Path → [(MyNode,MyNode)]
makePair iList = zip iList (tail iList)
```

8.5.2 Finding Data Flow Equations

There can be many different types of control flow in assembler code. Given a (connected) subgraph S of a complete CFG C, we say that

- S is *single-entry* if all edges from $C \setminus S$ to S go to a single node of S; this node is called the *entry point* of S. It is also required that all nodes of S be reachable from the entry point.
- S is *single-exit* if all edges from S to $C \setminus S$ go from a single node of S, and this node is called the *exit point* of S. We also require that the dual of a single-exit graph be a single-entry graph.
- E is an *execution path* of S if E is a single-entry, single-exit connected subgraph of S where all nodes besides the entry point have in-degree 1 and all nodes besides the exit point have out-degree 1.
- a *loop* L is a single-entry, single-exit connected subgraph of C where all nodes have in-degree 1 and out-degree 1 except for one node which has out-degree 2. Note that we include the "exit point" in the loop.

As was described in the earlier section, we use here the set of execution paths to model the control flow of a program. For the purposes of this thesis, we only treat single-entry single-exit subgraphs. Given this restriction, we divide control flow graphs into three broad categories (see Figure 8.1 for a pictorial representation):

- **Straight-Line Code (SC):** In other words, an execution path.
- **Generalized Straight-Line Code (GSC):** May contain a branch or jump, but that branch or jump will have all the outgoing edges to different nodes inside that code segment. In other words, globally this code is single-entry single-exit, but may contain multiple execution paths.
- **Looping code (LC):** An execution path which ends in a loop; the exit point of the loop may be extended by a (possibly empty) execution path.

Control flow patterns of assembler programs can exhibit many different structures. For simplicity, we choose these three structures as they represent the most common control flow structures in programming. Other control flow patterns can also be explored in future using the same techniques implemented here.

For each type of control flow graph noted above, we use a different strategy to find the corresponding data flow equations. The following sections will show the strategies to find the DFEs for each type of CFG cited before.

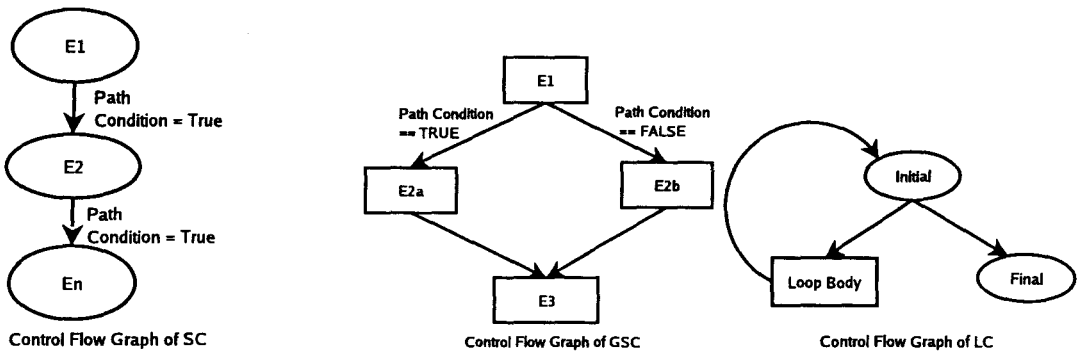


Figure 8.1: Pictorial Representation of Code Categories

In this Module, from a list of symbolic interpretations of some instructions, we find statements equivalent to those symbolic functions. A statement contains a *StateRef* (left-hand side of statement) and an *Expr* (value to be assigned in the *StateRef*) in the symbolic form.

```
module FindExpr
( transformToStmts, transformToConds, findAnntOfGraph, dividePathAnt
, mergePathCond, listCond, loopPathAnt, mergeLoopCond , convToStmt
)
where

import Symbolic (Func(..), CondFunc(..), TypeCast(..), Val(..),
                 StateComp(..), Operator(..), CondOpSm(..))
import MyGraph (MyGraph, MyNode, doAnnotation)
import Exp
import FindPath (PathType(..), annotationOfPaths, nodesFromStart,
                 makePair)
import FindJoin (findSplitJoin, getCommonDiv, getLoopParts)
import Data.Maybe (fromJust)
```

As was stated earlier, each instruction in IBM-1800 assembly language assigns some values to either (parts of) the State Components and/or the Memory.

`transformToStmts` makes this effective by translating every low-level assignment to a higher-level representation (in terms of assignment statements), where the left-hand-side is a *StateRef* and the right-hand-side is an arbitrary expression *Exp*. There are many different types of *Func* for various types of low level assignments in the instructions of the IBM-1800 assembler, so it is non-trivial to use *Func* to determine the values of the state components in the instructions. *Stmt* has a more unified and simple representative data structure for each of those *Func*s and that is why it can be used more efficiently to find the expressions in the statements.

```
transformToStmt:: Func → Stmt
transformToStmt st = case st of
    AssignSC16 s v   → Assign (SC s, UnaryOperation
                                    (Id, (MemoryConstant v)))
    UpdateSC16 s o v → Assign (SC s, BinaryOperation
                                    ((Variable (val161 v)),
                                    o, (MemoryConstant (val162 v))))
    AssignMem16 v l  → Assign (Mem l, UnaryOperation
                                    (Id, (Variable v)))
    AssignMemX v l   → Assign (Mem l, UnaryOperation
                                    (Id, (VariableX v)))
    UpdateSC32 s o v →  Assign (SC s,BinaryOperation
                                    ((Variable (val321 v)),
                                    o, (MemoryConstant (val322 v))))
    AssignX s v      → Assign (SCX s, UnaryOperation
                                    (Id, (MemoryConstant v)))
    UpdateX s v      → Assign (SCX s, BinaryOperation
                                    ((VariableX s),
                                    Add, (MemoryConstant v)))
    UpdateAS o v     → Assign (SC Acc, BinaryOperation
                                    ((Variable Acc),
                                    o, Constant v))
    UpdateAQS o v    → Assign (SC AccQ, BinaryOperation
                                    ((Variable AccQ),
                                    o, Constant v ))
```

```
    CondDisp c v      → Assign (Mem c, BinaryOperation
                                    ((MemoryConstant c),
                                    Add, (MemoryConstant v)))

transformToStmts:: [Func] → [Stmt]
transformToStmts = map transformToStmt
```

We also need to convert all different conditions in the symbolic interpretation in a unified data structure called *ConditionStmt*.

```
transformToCond:: CondFunc → ConditionStmt
transformToCond cf =
  case cf of
      Condition s o b → Check ((BrCondition (Atomic (Variable s))
                                o (Constant 0) BR), trbl b)
      CondDispAddT s v b → Check ((BrCondition (BinaryOperation
                                ((VariableX s), Add,
                                 MemoryConstant v)) EqO
                                 (VariableX s) MDX), trbl b)
      CondDispAddM l v b → Check ((BrCondition (BinaryOperation
                                ((MemoryConstant l), Add,
                                 (MemoryConstant v))) EqO
                                 (MemoryConstant l) MDX), trbl b)
      UpdateComp  s o v → Check ((BrCondition (Atomic (Variable s))
                                 o (MemoryConstant v) CMP),
                                 trcm o)
      otherwise → Check (NoCondition, NC)
  where trbl b = if b ≡ True then Tr
                            else Fl
        trcm cm = case cm of
                    EqO → Eql
                    LtO → Ltn
                    GrO → Gtr
```

```
transformToConds:: [CondFunc] → [ConditionStmt]
transformToConds = map transformToCond
```

We can now find all the annotation of paths in a Graph. We assume that the graph given is a single entry and single exit graph.

Given a start node and a graph, `findAnntOfGraph` finds path conditions and symbolic interpretations of all the instructions in all paths of the graph. First it annotates all the edges in the Graph using the `doAnnotation` function of the *MyGraph* module. Then it finds all the paths in the graph. If only one path is found then it is a Straight-line Code (SC). If two or more paths are found then it is a Generalized Straight-line Code (GSC) structure, and obviously, if there is any looping path then it is a Looping code (LC) structure.

```
findAnntOfGraph :: MyGraph → MyNode → [[([ConditionStmt],[Stmt])]]
findAnntOfGraph mg start
      = if length pl ≡ 1 then csList
                         else if any (λx → fst x ≡ Loop) pl
                                    then cLList
                                    else cfList
  where mg1 = doAnnotation mg
        pl = nodesFromStart mg1 start
        csList = [[([Check (NoCondition, NC)],
                    ((transformToStmts·snd·listCond)
                   $ annotationOfPaths mg1
                     (makePair $ concat $ map snd pl)))]]
        cfList = mergePathCond $ dividePathAnt mg1 start
        cLList = mergeLoopCond $ loopPathAnt mg1 start
```

Straight-Line Code (SC): We gather all the annotations of the edges of the single execution path. We use sequential composition $E1; E2$ to represent this.

Modeling the control flow and finding the semantic context of straight-line code is simple. In particular, there are no new path conditions that are imposed. We just need to find the state transformer corresponding to each statement and using a stack to keep track of the current environment, we sequentially compose all the expressions

representing these state transformers. Since each state transformer obtained from the previous stage is always of the form $V' = f(S)$ where V' is a single state component, and S is a finite set of state components, we obtain a set of the simplest kind of data flow equations.

For SC, `findAnntOfGraph` gets the edge annotation list using `annotationOfPaths` and transforms them to statements by `transformToStmts`.

Generalized Straight-Line Code (GSC): For GSC (refer to middle graph in Figure 8.1), we proceed as follows:

- find the nodes in the code that correspond to a split (a branch instruction) and a join (the meeting point of two different paths which started at a split). This divides the CFG as several SCs, which we label E_1, E_{2a} , E_{2b} , E_3.
- for each of the SCs, we generate a system of data flow equations. We use E_1 to also denote the resulting system (confusion is easily cleared from context).
- Write the system of data flow equations for the whole code as $E_1; (g_t \rightarrow E_{2a} | \neg g_t \rightarrow E_{2b}); E_3$ where g_t denotes the *guard* which corresponds to the choice for branch t and | denotes parallel composition.

The node where the paths in the CFG split is called the "split" node and the node where paths join again is called the "join" node. Since the graph is considered as single-entry and single-exit, so we must have at least one "split" and a "join" node if there are more than one paths in the CFG.

GSC (or branching codes) can be considered as one if-then-else structure. We divide the paths in the GSC into different segments using the "split" and "join" nodes and find symbolic interpretation of all the segments differently.

For an ideal GSC structure, in two of its paths, nodes before the split node and nodes after the join node are similar. In between split and join, the path segment for each path is different from one another. So we can divide the two paths of the GSC into four different path segments. They may be named as First Common (fc), Different Path 1 (pd1), Different Path 2 (pd2) and Second Common (sc) [See Figure 8.2].

`dividePathAnt` finds a list of (path condition, list of interpreted instructions) pair for all the segments of the paths. It first finds "split" and "join" nodes (if any)

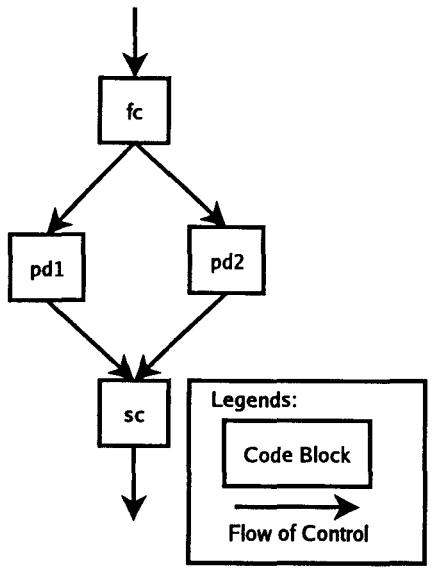


Figure 8.2: Shape of GSC

between the paths and divide the paths into segments using those nodes. Then it finds the paths conditions and function list of the instructions for those segments.

```
dividePathAnt :: MyGraph → MyNode → [[(CondFunc,[Func])]]
dividePathAnt mg st = map (annotationOfPaths mg) $ map makePair lst
     where paths = nodesFromStart mg st
           (splt, jnt) = findSplitJoin paths
           (fc, pd1, pd2, sc) = if (splt ≡ Nothing) ∨ (jnt ≡ Nothing)
                                then error "no intersection"
                                else getCommonDiv
                                     ((snd·head) paths)
                                     ((snd·last) paths)
                                     ((fromJust splt),
                                     (fromJust jnt))
           lst = if length fc ≡ 1
                 then [pd1, pd2, sc]
                 else if length sc ≡ 1
```

```
                    then [fc, pd1, pd2]
                    else [fc, pd1, pd2, sc]
```

`mergePathCond` transforms the functions into statements and combines all the conditions of the instructions into one path condition for each segment of the paths. It then transforms these conditions into *ConditionStmt*. For fc and sc, the path conditions are "True" as they are Straight-line Code (SC). for pd1 and pd2, the path condition depends on the condition of the instruction in the "split" node.

`listCond` combines all the conditions of instructions in one list from a list of (condition, function list) pairs for one segment of path.

After dividing the paths into segments, we must reorganize the statement list for each segment. As mentioned earlier, the symbolic interpretation of the instruction in a node appears as the annotation of its outgoing edges. In the branching structure code, after the "split" node each first edge of two different paths contains the same statement as annotation as they are generated from the same node although their conditions are different. So we will be removing this common statement in the common segment (`fc`) before the split for more precise interpretation.

```
trSnd = transformToStmts · snd
trFst = transformToCond· head·fst
trFsts = transformToConds·fst
nc = Check (NoCondition, NC)

mergePathCond :: [[(CondFunc, [Func])]] →
                    [[([ConditionStmt], [Stmt])]]
mergePathCond cfss =
  if length cfss ≡ 3
      then if (head $ fst f1) ≠ Tru
               then if head inst1 ≡ head inst2
                        then [[([nc],[head inst1]),
                             ([trFst f1], tail inst1),
                             ([trFst f2], tail inst2),
                             ([nc], trSnd f3)]]
                        else [[([trFst f1], inst1),
```

```
                                ([trFst f2], inst2),
                                ([nc], trSnd f3)]]
                    else if head inst2 ≡ head inst3
                                then [[([nc],
                                     (trSnd f1)++[(head inst2)]),
                                     ([trFst f2], tail inst2),
                                     ([trFst f3], tail inst3)]]
                                else [[([nc], trSnd f1),
                                     ([trFst f2], inst2),
                                     ([trFst f3], inst3)]]
          else if head inst2 ≡ head inst3
                           then [[([nc],
                                (trSnd f1)++[(head inst2)]),
                                ([trFst f2], tail inst2),
                                ([trFst f3], tail inst3),
                                ([nc], trSnd f4)]]
                           else [[([nc], trSnd f1),
                                ([trFst f2], inst2),
                                ([trFst f3], inst3),
                                ([nc], trSnd f4)]]
  where  condfn = map listCond cfss
         f1 = head condfn
         inst1 = trSnd f1
         f2 = head (drop 1 condfn)
         inst2 = trSnd f2
         f3 = head (drop 2 condfn)
         inst3 = trSnd f3
         f4 = last condfn

listCond :: [(CondFunc, [Func])] → ([CondFunc], [Func])
listCond cfs = ((map fst cfs), (concat $ map snd cfs))
```

Looping Code (LC):

One significant challenge in modeling any program, symbolically or otherwise, is to correctly model loops. For loops, we will use (symbolic) recurrence equations as a model [FS03]. If we are lucky, these recurrences will be solvable in closed form. Nevertheless we can continue with this implicit representation even if they are not. Frequently, properties of the solution of recurrence equations can be derived from the recurrence itself without needing the closed-form solution.

To make the discussion more concrete, we will use the following sample code as example:

```
OADDR REL OBJ. ST.N.  LABEL  OPCD FT OPRNDS
 35CE  0  1001  0708         SLA      1
 35CF  0  72FF  0709         MDX  2  -1
 35D0  0  70FD  0710         MDX     *-3
```

In this simple loop, the accumulator A value is shifted left by one and XR2 is decreased by one at each loop execution. We can express this change in terms of recurrences: $A_{n+1} = 2 * A_n$ and $\mathtt{XR2}_{n+1} = \mathtt{XR2}_n - 1$, which expresses that the value of the accumulator and XR2 at time $n+1$ are a function of their values at time n, where $n \geq 0$. Since upon loop entry both A and XR2 have a value, we know the necessary initial conditions for this first-order recurrence. We use $\overline{A}$ and $\overline{\mathtt{XR2}}$ to denote these initial values. We can thus represent the symbolic meaning of the loop using these recurrence equations and the initial conditions.

To determine the value of A after the loop terminates, we need to know if and when the loop will stop. We define a stopping criterion ϕ : State $\rightarrow \mathbb{B}$ which will symbolically determine the number of iterations for the loop. This stopping criterion naturally corresponds with the loop condition – which for our simple loop is $\phi = \mathtt{XR2} > 0$. The recurrence equation, initial condition and stopping criterion are sufficient to completely describe all the loop information symbolically.

For each component of the state v, which is modified in a loop, we represent the corresponding information as a function $\mu(v, s, c)$, from the variable, state and a program context c (See Section 7.1). The stopping criteria is given by the path condition of the program context c, and the initial condition is determined from s. The result of μ is a representation of the recurrence equation for that state component.

As in GSC, the starting nodes for LC before the "split" node in the paths and ending nodes after the "join" node are common. In between them there are different path segments for each path which are shown in the Figure 8.3.

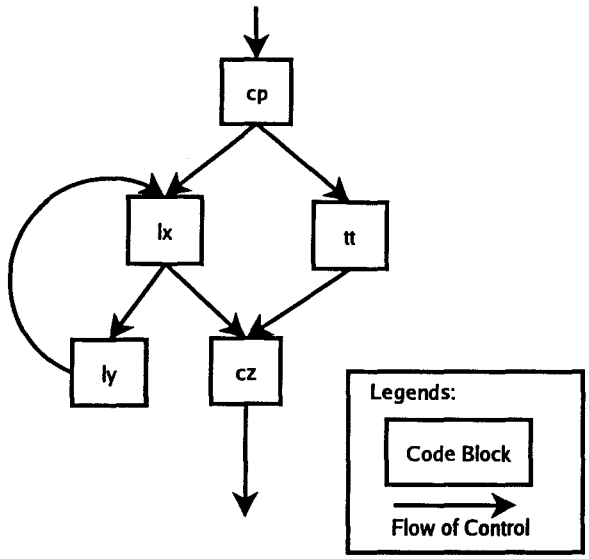


Figure 8.3: Shape of Looping Codes

We can divide two paths of the LC in five different path segments. They may be named as Common P (cp), Terminating T (tt), Looping X (lx), Looping Y (ly) and Common Z (cz). cp, tt, ly and cz must be straight line codes (SC) however lx can be either SC or GSC depending on the number of paths in lx.

LoopPathAnt finds a list of (path condition, list of interpreted instructions) pairs for all the segments of the paths in a LC. As in `dividePathAnt`, it first finds "split" and "join" nodes among the paths and divides the paths into segments using those nodes. Then it determines the path conditions and function list of the instructions for those segments.

```
loopPathAnt :: MyGraph → MyNode → [[[(CondFunc,[Func])]]]
loopPathAnt mg st = [[acp, att, aly, acz], alx]
        where paths = nodesFromStart mg st
              (sp, jn) = findSplitJoin paths
              (cp,tt,lx,ly,cz) = if (sp ≡ Nothing) ∨ (jn ≡ Nothing)
                                 then error "no intersection"
                                 else getLoopParts paths
```

```
                                        ((fromJust sp), (fromJust jn))
        [acp, att, aly, acz] = map (annotationOfPaths mg)
                                        $ map makePair [cp,tt,ly,cz]
        alx = map (annotationOfPaths mg) $ map makePair lx
```

mergeLoopCond works as mergePathCond, that is it transforms the functions into statements and combines all the condition of the instructions into one path condition for each segment of the paths, and then transforms this condition into *ConditionStmt* for LC.

```
mergeLoopCond :: [[[(CondFunc,[Func])]]] →
                    [[([ConditionStmt],[Stmt])]]
mergeLoopCond cfst = [[([nc],trSnd cacp)],
                      [cTT], cLX, [cLY],
                      [([nc],trSnd cacz)]]
    where [cacp, catt, caly, cacz] = map listCond (head cfst)
          cLY = convToStmt caly
          cTT = convToStmt catt
          loopc = map listCond (last cfst)
          cLX = map convToStmt loopc


convToStmt :: ([CondFunc], [Func]) → ([ConditionStmt], [Stmt])
convToStmt cfs = (cds, (trSnd cfs))
       where cds = filter (λx → x ≠ nc) (trFsts cfs)
```

8.6 Examples

The code segments used to generate the following examples are taken from the BPC (Boiler Pressure Control) code of OPG.

8.6.1 SC Example

We present the same code segment in Section 2.3.1:

```
OADDR REL OBJ. S.NO. LABEL OPCD FT OPRNDS
 35B6 0  C129  0677  TRBFB  LD   1  41
 35B7 0  A12A  0678         M    1  42
 35B8 0  1082  0679         SLT     2
 35B9 0  912B  0680         S    1  43
 35BA 0  A12C  0681         M    1  44
 35BB 0  108F  0682         SLT     15
 35BC 0  A92D  0683         D    1  45
 35BD 0  D12E  0684         STO  1  46
```

The Data Flow Equations (DFE) for this code segment are:

```
PathCondition:  True

Instruction Execution:

A := C(XR1 + 41)
AQ := A * C(XR1 + 42)
AQ <<= 2
A -= C(XR1 + 43)
AQ := A * C(XR1 + 44)
AQ <<= 15
A := AQ / C(XR1 + 45)
Q := AQ % C(XR1 + 45)
C(XR1 + 46) := A
```

8.6.2 GSC Example

This chunk of code is a also a part of the the BPC (Boiler Pressure Control) code of OPG.

```
 35C4 0 73FF  0695       MDX   3 -1
 35C5 0 700F  0696       MDX     TRBFE
              0697
```

```
35C6 0 1010  0698 TRBFD SLA     16
35C7 0 D12F  0699       STO   1 47   0
35C8 0 7012  0700       MDX     TROUT
             0701
             0702
35C9 0 0000  0703 DI2F3 DC      0
35CA 0 6203  0704       LDX   2 3
35CB 0 6300  0705       LDX   3 0
35CC 0 4810  0706       BSC     -
35CD 0 7301  0707       MDX   3 1
35CE 0 1001  0708       SLA     1
35CF 0 72FF  0709       MDX   2 -1
35D0 0 70FB  0710       MDX     *-5
35D1 00 66002099 0711   LDX  L2 BPCD
35D3 00 4C8035C9 0712   BSC  I  DI2F3
             0713
             0714
             0715
35D5 0 C209  0716 TRBFE LD    2 9
35D6 0 911B  0717       S     1 27
35D7 0 A130  0718       M     1 48
35D8 0 1005  0719       SLA     5
35D9 0 D12F  0720       STO   1 47
35DA 0 7000  0721       MDX     TROUT
             0722
             0723
             0724
             0725
35DB 0 C12E  0726 TROUT LD    1 46
35DC 0 812F  0727       A     1 47
35DD 0 A132  0728       M     1 50
35DE 0 1089  0729       SLT     9
35DF 0 D123  0730       STO   1 35
```

To understand the execution sequence of this GSC segment, Here we give a graphical control flow pattern (Figure 8.4) of the segment.

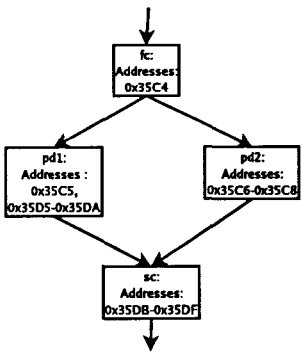


Figure 8.4: Control Flow Graph of the Segment 0x35C4-0x35DF

The following are the DFE presentation for different segments of the GSC segment.

```
Segment: fc

PathCondition:  True

Instruction Execution:
XR3 += (-1)
Sign1 := Sign(XR3)
Sign2 := Sign(XR3+(-1))

Segment: pd1

PathCondition:

(Sign1 <> Sign2 || XR3 == 0) == False

Instruction Execution:

A := C(XR2 + 9)
```

```
A -= (XR1 + 27)
AQ = A * C(XR1 + 48)
A <<= 5
C(XR1 + 47) := A

Segment: pd2

PathCondition:

(Sign1 <> Sign2 || XR3 == 0) == False

Instruction Execution:

A <<= 16
C(XR1 + 47) := A

Segment: sc

PathCondition:  True

Instruction Execution:

A := C(XR1 + 46)
A += C(XR1 + 47)
AQ := A * C(XR1 + 50)
AQ <<= 9
C(XR1 + 35) := A
```

8.6.3 LC Example

Another BPC code segment to present the LC structure is adapted here.

```
35C9 0  0000      0703 DI2F3 DC      0
35CA 0  6203      0704       LDX   2 3
```

```
35CB 0  6300       0705      LDX   3 0
35CC 0  4810       0706      BSC     -
35CD 0  7301       0707      MDX   3 1
35CE 0  1001       0708      SLA     1
35CF 0  72FF       0709      MDX   2 -1
35D0 0  70FB       0710      MDX     *-5
35D1 00 66002099   0711      LDX   L2 BPCD
35D3 00 4C8035C9   0712      BSC   I  DI2F3
```

To understand the execution sequence of this LC segment, Here we give a graphical control flow pattern (Figure 8.5) of the segment.

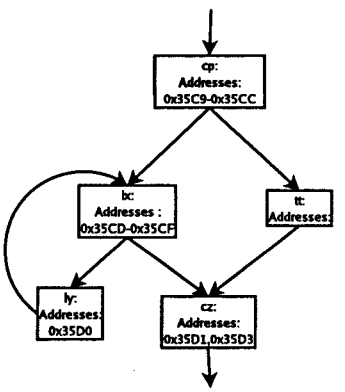


Figure 8.5: Control Flow Graph of the Segment 0x35C9-0x35D3

```
Segment: cp

PathCondition:  True

Instruction Execution:

XR2 := 3
XR3 := 0
```

```
Segment: tt

PathCondition:

Instruction Execution:

Segment: lx

PathCondition:

(A < 0) ==False, (Sign1 <> Sign2 || XR3 == 0) ==False

Instruction Execution:

XR3 += 1
Sign1 := Sign(XR3)
Sign2 := Sign(XR3+1)
A <<= 1

PathCondition:

(A < 0) ==True

Instruction Execution:

A <<= 1

PathCondition:

(A < 0) ==False, (Sign1 <> Sign2 || XR3 == 0) ==True

Instruction Execution:
```

```
XR3 += 1

Segment: ly

PathCondition:

(Sign1 <> Sign2 || XR2 == 0) ==False

Instruction Execution:

XR2 += (-1)
Sign1 := Sign(XR2)
Sign2 := Sign(XR2+(-1))

Segment: cz

PathCondition:  True

Instruction Execution:

XR2 := 8345
```

The data flow equations (DFE) in all the three previous examples give us a high level representation of the computation done in the assembly code. Still more work can be done on these DFEs to understand the meaning of the code better. The following chapters show us different ways to interpret these DFEs.

Chapter 9

Generating Data Flow Graphs

Contained in this chapter is the tool used to generate Data Flow Graph (DFG) from the assembler code. In different subsections, the internal data structure of the Data Flow Graph, functions to create the DFG from the Data Flow Equations (DFE), and the garbage collection step of the DFG are discussed. Later, we also provide some examples DFGs which are generated using this tool.

9.1 Data Flow Graph

A data-flow graph (DFG) is a graph which represents data dependencies among a number of operations in a program. Definition and structure of DFG are given earlier in Section 3.1.2. DFGs are very important in data flow analysis at runtime.

9.2 DFG Generation Process

In this section we give a brief overview of the data flow graph generation process. Figure 9.1 shows different internal steps of DFG generation. We start the DFG generation process by taking the Data Flow Equations of the corresponding code segment as input. We then produce a Data Flow Graph with redundant entries. These redundant entries can be added in the DFG for various reasons. For example, when any value is assigned to *AccQ* (Combined Accumulator and its extension register) in the next instructions, we may need the value of either *Acc* (Accumulator) or its extension

register (*Q*). So we add two more entries of *Acc* and *Q* in the DFG. The entries that are not used by the later instructions are removed in the Garbage Collection phase. Not all of the unused entries are removed. The rules for removing unused nodes in the DFG are discussed in detail in Section 9.5.

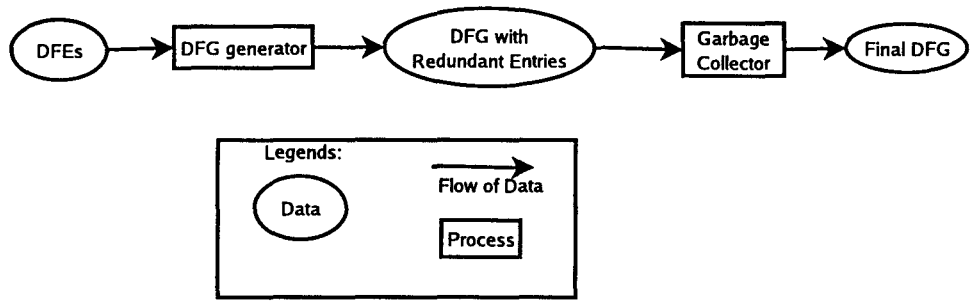


Figure 9.1: DFG Generation Process

In the following sections, we provide three different modules in DFG generation process. Section 9.3 shows the internal data structure implementation of the DFG. Following that, we include the module for generating a DFG. The DFG produced in that module contains redundant nodes. To remedy that the Garbage Collection is given in Section 9.5.

9.3 Internal Data Structure of DFG

In this module, we define all the data types needed by the Data Flow Graph generation tool.

```
module Dfg
( OperatorNode(..), OperandNode, OEdge(..), InEdges
, OutEdges, DfdGraph, UseType, UsedNode, NodeMap
)
where

import Exp (BasicExp, ConditionExp, Cond)
import Stack
```

```
import Observe
```

Like other graph structures, a Data Flow Graph (DFG) contains nodes and edges. Edges in a DFG represents the flow of data from one operation to the other. Nodes in a DFG are of two types: Operand or Operator. Operand nodes represent the data and operator nodes the operation carried on the data in the instructions. Thus in our DFG data structure the nodes are divided into operand and operator nodes.

The *OperandNode* data type declares the node for the operands (data) of the operations. It contains *BasicExp* expression type which defines the basic expression for the data. *OperatorNode* defines the type for different types of operations. In IBM-1800, operations inside instructions can be Unary, Binary or Conditional. We declare a special *OperatorNode* called *Join* to define the join of same data value from different branches.

We use one important integer fields (`ci`) to declare *OperatorNode* and *OperandNode*s. `ci` distinguishes between different nodes with the same label coming from different instructions. In most cases, this integer field gives the no. of the instruction from which the node is generated.

Instead of deriving *Ord*, we create our own ordering for the *OperatorNode*s. The derived *Ord* was not working as it was ordering depending only on the operator in the node. But we may have different *OperatorNode*s with same label (operator). As such, we were getting aberrant edges among the nodes while using derived *Ord*. To solve the problem, we clearly distinguish among the *OperatorNode*s by a predefined sequence.

In our DFG data structure, we assume that there will only be edges from the operator node to operand node and vice versa. There must not be any edges between two nodes of the same kind. *InEdges* defines the edges from the *OperandNode* to *OperatorNode* whereas *OutEdges* defines the edges from the *OperatorNode* to *OperandNode*. In the definition of *OutEdges*, we use another data type: *OEdge*. From an *OperatorNode*, depending on the type of that node, we can have edges to different *OperandNode*s where each edge may contain conditions like True, False, Equal etc. *OEdge* defines different types of edges with the conditions associated with them. For *OneEdge* in *OEdge*, there is no condition associated with the edge. This

is used for unconditional data flow. From the semantical understanding of the assembler we can determine that there can be at most three outgoing edges from one *OperatorNode*. That is why we created a data type from the outgoing edges for one *OperatorNode* which contains three options: *OneEdge* (for unconditional operators), *TwoEdge* (for branching operators) and *ThreeEdge* (for *CMP* operators). This gives us the strong typographical setting for the data type and can be used to avoid errors during runtime. For *InEdges* we use a simple list of *OperatorNode* as the successor of the *OperandNode* since we are not sure how many times a *OperandNode* will be referenced later in other instructions.

So our DFG structure *DfdGraph* contains two finitemaps: One for the *InEdges* and the other for *OutEdges*; one finitemap from the *OperandNode* to *OperatorNode* and one from the *OperatorNode* to *OEdge*.

```
data OperatorNode =
     Unary TypeCast Int
   | Binary Operator Int
   | ConditionVal ConditionExp Int
   | Join Int Int deriving (Eq)

instance Show OperatorNode where
   show (Unary tc ci) =  show tc
   show (Binary op ci) = show op
   show (ConditionVal cf ci) = show cf
   show (Join ci1 ci2) = "Join"

instance Ord OperatorNode where
    compare (Unary t n) (Unary s m)
         | t ≡ s    = compare n m
         | t ≤ s    = LT
         | otherwise = GT
    compare (Unary t n) (Binary s m) = LT
    compare (Binary t n) (Unary s m) = GT
    compare (ConditionVal t n) (Unary s m) = LT
    compare (Unary t n) (ConditionVal s m) = GT
```

```
    compare (ConditionVal t n) (Binary s m) = LT
    compare (Binary t n) (ConditionVal s m) = GT
    compare (Binary t n) (Binary s m)
         | t ≡ s     = compare n m
         | t ≤ s     = LT
         | otherwise = GT
    compare (ConditionVal t n) (ConditionVal s m)
                     = compare n m
    compare (Unary n m) (Join t s) = LT
    compare (Join t s) (Unary n m) = GT
    compare (ConditionVal n m) (Join t s) = LT
    compare (Join t s) (ConditionVal n m) = GT
    compare (Join t s) (Binary n m) = GT
    compare (Binary n m) (Join t s) = LT
    compare (Join t s) (Join n m)
         | t ≡ n     = compare s m
         | t ≤ n     = LT
         | otherwise = GT


data OperandNode = ExpressionNode BasicExp Int deriving (Eq)

instance Show OperandNode where
    show (ExpressionNode e ci) = show e

instance Ord OperandNode where
    compare (ExpressionNode e1 ci1) (ExpressionNode e2 ci2)
         | e1 ≡ e2     = compare ci1 ci2
         | e1 ≤ e2     = LT
         | otherwise   = GT

data OEdge = OneEdge OperandNode
           | TwoEdge (Cond,OperandNode) (Cond,OperandNode)
```

```
        | ThreeEdge (Cond,OperandNode) (Cond,OperandNode)
                    (Cond,OperandNode) deriving (Eq)

instance Show OEdge where
   show (OneEdge on) = show on
   show (TwoEdge (cn1,on1) (cn2,on2)) = show on1
                                        ++ ","++ show on2
   show (ThreeEdge (cn1,on1) (cn2,on2) (cn3,on3)) = show on1
                                        ++ "," ++ show on2
                                        ++ ","++show on3

type InEdges = (OperandNode, [OperatorNode])
type OutEdges = (OperatorNode, OEdge)

data DfdGraph = DfdGraph (FiniteMap OperandNode [OperatorNode])
                         (FiniteMap OperatorNode OEdge)
```

This data type is used for garbage collection and indicates whether a node is used by the next nodes.

UsedNode contains two finitemaps: one for *OperandNode* and the other for *OperatorNode*. In the finitemaps, each node (operand or operator) is mapped to *UseType* that can be either *Used* or *Unused*. Depending on the *UseType* of the nodes, we will get rid of the unused nodes during garbage collection phase of DFG generation.

```
data UseType = Used | Unused deriving (Eq)
instance Show UseType where
       show Used = "used"
       show Unused = "unused"

data UsedNode = UsedNode (FiniteMap OperandNode UseType)
                         (FiniteMap OperatorNode UseType)
```

Below is the same environment that is declared in Stack.lhs. A type class is defined there for these type of environments. This instance associates *BasicExp* with

OperandNode and is used for same node lookup.

```
type NodeMap = BasicExp → OperandNode

instance Stack BasicExp OperandNode where
    createStack e = ExpressionNode e 0
    addEntry s (k,v) = λd → if k≡d then v else s d
    addEntries = foldl addEntry
    lookupEntry s k = s k
```

9.4 DFG Generation

In this module we convert the Data Flow Equations (DFE) generated from a code segment into a Data Flow Graph (DFG). The DFEs of the code segment show the data flow from one instruction to the next by representing the data as symbolic values and the instructions represented as statement using those data symbols. However, DFG gives us the pictorial presentation of the data flow from one operation to the next. DFG gives a clear understanding of data dependency among the operations in the code.

```
module Dfe2Dfg (dfdGraphToGxlGraph)
where

import MyPrelude (fst3, thrd3)
import GxlGraph (GxlGraph, addEdges, addNodes,)
import Symbolic (StateComp(..), TypeCast(..))
import Stack
import Exp
import Dfg
import Dfe2DfgCommon
import GarbageCollect (markNodes, garbageCollectOfDfdGraph)
import Data.FiniteMap (emptyFM, fmToList, keysFM, addListToFM,
                       addListToFM_C, addToFM,
                       addToFM_C, lookupFM)
```

```
import Data.Maybe (fromJust)
import Data.List (find, delete, deleteBy)

import Observe
```

dfdGraphToGxlGraph is the main function used to generate the Data Flow Graph (DFG)s from the Data Flow Equation (DFE)s. As seen in the Figure 9.1 that shows the steps to produce DFGs, we first generate the internal data structure of the DFG, *DfdGraph*, that contains the redundant entries of data nodes which are unused. dfeNodesEdgesToGraph and dfeNodesEdgesToGraphBr are used to generate *DfdGraph* for SC and GSC respectively. After generating the DFG, we garbage collect the *DfdGraph* by marking the nodes with their *UseType* and removing the unused nodes from the graph using the garbageCollectOfDfdGraph function. Finally, for exchange and display purposes, we convert the DFG into GXL format, that is we create a new DFG in GXL by converting the *InEdge*, *OutEdge* into GXL edges and *OperandNode* and *OperatorNode* into GXL nodes.

```
dfdGraphToGxlGraph :: GxlGraph → String →
                      [[([ConditionStmt], [Stmt])]] → GxlGraph
dfdGraphToGxlGraph ggraphs name cstmts = addEdges (inEdges++outEdges)
                              $ addNodes (opndNodes++optrNodes) ggraphs
     where (i, dfdGraphBGC, nMap)
                          = if (length $ concat cstmts) ≡ 1
                               then dfeNodesEdgesToGraph
                                    (DfdGraph emptyFM emptyFM) 1
                                     createStack (concat $ map snd
                                     $ concat cstmts)
                               else dfeNodesEdgesToGraphBr
                                    (DfdGraph emptyFM emptyFM) 1
                                     createStack (concat cstmts)
           usedNode1 = markNodes dfdGraphBGC (i-1)
                                 (UsedNode emptyFM emptyFM) nMap
           (DfdGraph fmopn fmope) = garbageCollectOfDfdGraph
                                                dfdGraphBGC usedNode1
```

```
inEdges =  pairsToGxlEdges $ inEdgeToIds name
                                    (fmToList fmopn)
outEdges = reverse $ outEdgeToGxlEdges name
                                    (fmToList fmope) []
opndNodes = map idToGxlNode $ map (opnodeToId name)
                                    (keysFM fmopn)
optrNodes = map idToGxlNode $ map (optnodeToId name)
                                    (keysFM fmope)
```

We now discuss how to create the internal data structure (*DfdGraph*) of the DFG from the DFEs.

`dfeNodesEdgesToGraph` is used to create the DFG for the SC type codes. This function adds all the nodes and edges in the graph iteratively. In one iteration, it finds all the nodes that have to be added in the DFG for one instruction. For the input nodes, it decides which nodes have links to the data nodes from the previous instructions. It separates those *OpearndNode*s from the other input nodes and does not add them in the DFG as they can be replaced by the previous occurrence of them (`findAppNodes`). The rest of the input nodes are added to the DFG directly. Using this information, it finds all the feasible edges from the edge list of that instruction and adds them in the DFG (`addAppEdges`). As the output data nodes are yet to be referenced by the next instructions, we add those nodes in the DFG with their successor list as empty (`makeBlankPairs`).

`findAppNodes` uses *NodeMap* which maps the data symbol (*BasicExp*) to the *NodeId* to check whether the previous data nodes have been referenced by the new input nodes. *NodeMap* contains all the data symbols and their corresponding *NodeId*s in the *DfdGraph*. In each iteration, all the new input (feasible) and output nodes are added to the *NodeMap* for future reference. *Int* is used to give all nodes a unique ID in each iteration.

```
dfeNodesEdgesToGraph :: DfdGraph → Int → NodeMap
                     → [Stmt] → (Int,DfdGraph,NodeMap)
dfeNodesEdgesToGraph dgrphs i nMap [] = (i,dgrphs,nMap)
dfeNodesEdgesToGraph dgrphs@(DfdGraph fmopn fmope) i nMap (fn:fns)
    = dfeNodesEdgesToGraph  dfg1 (i+1) newNMap fns
```

```
where nodels = dfeToNodes fn i
      inodes = findAppNodes nMap (fst3 nodels) ([],[])
      newNMap = addToNodeMap nMap ((fst inodes)
                                   ++(thrd3 nodels))
      edgePairs = dfeToEdgePairs fn i
      newFmope =  addListToFM fmope (snd edgePairs)
      dfg0 = makeBlankPairs (thrd3 nodels)
                            (DfdGraph fmopn newFmope)
      dfg1 = addAppEdges (fst edgePairs)
                         dfg0 (snd inodes)
```

`dfeNodesEdgesToGraphBr` generates the DFG for branching structure codes. As we saw in Figure 8.2, for the GSC, there can be four different segments of code: First Common (`fc`), Different Path 1 (`pd1`), Different Path 2 (`pd2`) and Second Common (`sc`). All of these four segments can be considered as `sc` type. So we can use `dfeNodesEdgesToGraph` to create a DFG for these segments. In finding the DFEs, depending on the structure, we can find that in a single entry single exit subgraph, there are always some instructions in the `fc` segment but there may not be any `sc` segment. The first implementation of `dfeNodesEdgesToGraphBr` is for that type of structure of GSC where there are no `sc` whereas the second one is for an ideal GSC with four segments of code.

Both of the implementations have everyting in common except for creating the DFG of the last segment. We start with generating the DFG for `fc` and then we add the condition node in the DFG that produces two different segments. Condition node is a special type of *OperatorNode*. `conditionToiNodeEdges` creates the condition node and adds the input edges to that node from the `fc` segment nodes. After that, we generate the nodes and edges in the DFG for both `pd1` and `pd2` segment respectively. `conditionTooNodeEdges` adds the output edges from the condition node to two different segment nodes. After generating the nodes and edges for pd1 and pd2, we find the leaf nodes (with no outgoing edges) for both of those segments which may be used as data in the next `sc` segment. For each pair of leaf nodes which are common in the two segments, we create a join node (another special type of *OperatorNode*) using `zippOperandNodes` and `addJoinNodes`. This join node acts as an *Or* operation for both of those common nodes. The first implementation ends here as we do not

have an sc segment. In the second implementation, we add the nodes and edges for the sc segment and thus finish generating the DFG.

```
dfeNodesEdgesToGraphBr :: DfdGraph → Int → NodeMap
              → [([ConditionStmt],[Stmt])] → (Int,DfdGraph, NodeMap)
dfeNodesEdgesToGraphBr dgrphs i nMap [cst1,cst2, cst3]
                              = (i4,dfdg4,nm4)
        where (i1,dfdg1,nm1) = dfeNodesEdgesToGraph dgrphs i
                                        nMap (snd cst1)
              stmt1 = (last·snd) cst1
              dfd2 = conditionToiNodeEdges dfdg1 (i1+1)
                                (head $ fst cst2) stmt1
              (i2,dfdg2,nm2) = dfeNodesEdgesToGraph dfd2
                                (i1+2) nm1 (snd cst2)
              (i3,dfdg3,nm3) = dfeNodesEdgesToGraph dfdg2
                                (i2+1) nm1 (snd cst3)
              stmt2 = (head·snd) cst2
              stmt3 = (head·snd) cst3
              dfd3 = conditionTooNodeEdges dfdg3 (i1+2) (i2+1)
                               (head $ fst cst2) stmt2 stmt3
              (DfdGraph fmo2 fme2) = dfdg2
              (DfdGraph fmo3 fme3) = dfdg3
              leafs2 = findLeafs dfdg2 (i1+2) i2
                                (keysFM fmo2) []
              leafs3 = findLeafs dfdg3 (i2+1) i3
                                (keysFM fmo3) []
              zippedNodes = zipOperandNodes leafs2
                                 leafs3 nm1 []
              (i4,dfdg4, nm4) = addJoinNodes dfd3 (i3+1)
                                 createStack zippedNodes
dfeNodesEdgesToGraphBr dgrphs i nMap [cst1, cst2, cst3, cst4]
                              = (i5,dfdg5,nm5)
        where (i1,dfdg1,nm1) = dfeNodesEdgesToGraph dgrphs i
                                        nMap (snd cst1)
```

```
stmt1 = (last·snd) cst1
dfd2 = conditionToiNodeEdges dfdg1 (i1+1)
                    (head $ fst cst2) stmt1
stmt2 = (head·snd) cst2
(i2,dfdg2,nm2) = dfeNodesEdgesToGraph dfd2
                    (i1+2) nm1 (snd cst2)
(i3,dfdg3,nm3) = dfeNodesEdgesToGraph dfdg2
                    (i2+1) nm1 (snd cst3)
stmt3 = (head·snd) cst3
dfd3 = conditionTooNodeEdges dfdg3 (i1+2) (i2+1)
                  (head $ fst cst2) stmt2 stmt3
(DfdGraph fmo2 fme2) = dfdg2
(DfdGraph fmo3 fme3) = dfdg3
leafs2 = findLeafs dfdg2 (i1+2) i2
                   (keysFM fmo2) []
leafs3 = findLeafs dfdg3 (i2+1) i3
                   (keysFM fmo3) []
zippedNodes = zipOperandNodes leafs2
                     leafs3 nm1 []
(i4,dfdg4, nm4) = addJoinNodes dfd3 (i3+1)
                    createStack zippedNodes
(i5,dfdg5, nm5) = dfeNodesEdgesToGraph dfdg4 i4
                                  nm4 (snd cst4)
```

As we saw in **dfeNodesEdgesToGraph**, the probable list of nodes for one instruction is generated in one iteration. **dfeToNodes** converts all the operands and operators of an instruction to nodes either as *OperandNode*s or *OperatorNode*s. It divides the nodes as input, operator and output nodes. The input nodes are divided so that they can be checked for the edges that may come from the previous nodes. For output nodes, no such verification is necessary and they can be added in the DFG directly. If the output data is *AccQ*, we create two more output nodes of *Acc* and *Q* as they may be referenced in the future instructions.

The input and operator nodes created in one iteration will have the same *Int* and the output nodes will have *Int*+1 as as a part of their IDs. Output nodes get *Int*+1

as they may be the input nodes of the next instruction.

```
dfeToNodes:: Stmt → Int → ([OperandNode],
                           [OperatorNode], [OperandNode])
dfeToNodes (Assign (sr, UnaryOperation (tc,ex))) i =
    ([(ExpressionNode ex i)],[(Unary tc i)],[srnd])
   where srnd = conv2Expr sr (i+1)
dfeToNodes (Assign (sr, BinaryOperation (ex1, op, ex2))) i =
  if (sr ≡ SC AccQ) then
      ([(ExpressionNode ex1 i), (ExpressionNode ex2 i)],
       [(Binary op i), (Unary Upper16 i), (Unary Lower16 i)],
       [srnd , (ExpressionNode (Variable Acc) (i+1)),
       (ExpressionNode (Variable Q) (i+1))])
  else
      ([(ExpressionNode ex1 i), (ExpressionNode ex2 i)],
       [(Binary op i)], [srnd])
  where srnd = conv2Expr sr (i+1)
```

Like the probable nodes for one instruction, we also create the probable edge list for that instruction. `dfeToEdgePairs` creates all the probable edge pairs of an instruction in the DFD. Later we will remove the redundant edges from these lists by checking the previous references of the input nodes.

```
dfeToEdgePairs:: Stmt → Int → ([InEdges], [OutEdges])
dfeToEdgePairs (Assign (sr, UnaryOperation (tc,ex))) i=
    ([(ExpressionNode ex i, [(Unary tc i)])],
     [(Unary tc i, OneEdge srnd)])
   where srnd = conv2Expr sr (i+1)
dfeToEdgePairs (Assign (sr, BinaryOperation (ex1, op, ex2))) i =
    if (sr ≡ SC AccQ) then
        ([(ExpressionNode ex1 i, [(Binary op i)]),
         (ExpressionNode ex2 i, [(Binary op i)]),
         (srnd, [(Unary Upper16 i), (Unary Lower16 i)])],
         [(Binary op i, OneEdge srnd), (Unary Upper16 i,
```

```
          OneEdge (ExpressionNode (Variable Acc) (i+1))),
          (Unary Lower16 i, OneEdge (ExpressionNode
          (Variable Q) (i+1)))])
    else ([(ExpressionNode ex1 i, [(Binary op i)]),
          (ExpressionNode ex2 i, [(Binary op i)])],
          [(Binary op i, OneEdge srnd)])
  where srnd = conv2Expr sr (i+1)
```

Some input nodes from one instruction can't be added as they are the reference of the previous nodes. From a list of input nodes, **findAppNodes** divides the input nodes into two groups: one contains nodes which do not have previous occurrences and the other contains nodes paired with their previous node occurrence.

```
findAppNodes:: NodeMap → [OperandNode] →
    ([OperandNode],[(OperandNode, OperandNode)]) →
    ([OperandNode],[(OperandNode, OperandNode)])
findAppNodes nMap [] (opndss, ndopnds) = (opndss, ndopnds)
findAppNodes nMap (opnd@(ExpressionNode e ci):opnds)
        (opndss, ndopnds) = findAppNodes nMap opnds rest
      where rest = if prevNode ≠ testNode then
                       (opndss, [(prevNode, opnd)]++ndopnds)
                   else  (opndss++[opnd], ndopnds)
            prevNode = lookupEntry nMap e
            testNode = ExpressionNode e 0
```

Here we find the appropriate *InEdge*s that have to be added in the DFG. For each instruction, there may be some of the input nodes which are references of the previous nodes (we have found them in **findAppNodes**). So we have to remove some of the probable input edges to the *OperatorNode* and add new input edges to the *OperatorNode* fromthe previous references of those *OperandNode*s.

```
addAppEdges:: [InEdges] → DfdGraph →
                [(OperandNode, OperandNode)] → DfdGraph
addAppEdges ines (DfdGraph fmo fme) [] = (DfdGraph nfmo fme)
```

```
        where nfmo = addListToFM fmo ines
addAppEdges iess (DfdGraph fmo fme) ((nd, opnd):ndopnds) =
    addAppEdges newndLst (DfdGraph nfmo fme) ndopnds
        where ndst = find (λx → fst x ≡ opnd) iess
              newndLst = if ndst ≠ Nothing
                            then deleteBy (λx y→ y ≡ x)
                                 (fromJust ndst) iess
                            else iess
              nfmo = addToFM_C (λx y → x++y) fmo
                          nd (snd $ fromJust ndst)
```

conditionToiNodeEdges creates the "Condition" node in the branching structure code and adds the incoming edges to that node. The incoming edges to the "Condition" node comes from the output nodes of the last instruction of a First Common (fc) segment. It gets the last instruction of the fc segment and adds edges to the "Condition" node from the output nodes of that instruction. In case of the MDX instruction (one special branching instruction), it also adds the "Sign" nodes which are used as input nodes to the "Condition" node (addSignNodes).

ciNodeToDfdGraph acts as a helping function of conditionToiNodeEdges and adds all the edges to the "Condition" node in the DFG.

```
conditionToiNodeEdges :: DfdGraph → Int →
                         ConditionStmt → Stmt → DfdGraph
conditionToiNodeEdges dfg ci cf stm = dfg1
          where cnd = condToNodes cf ci
                nodes = dfeToNodes stm (ci-2)
                outnodes = thrd3 nodes
                outnode@(ExpressionNode be1 oi) = head outnodes
                innode = fromJust $ find (λx@(ExpressionNode be i)
                                    → be ≡ be1) (fst3 nodes)
                dfg0 = addSignNodes dfg cnd innode outnode
                dfg1 = ciNodeToDfdGraph dfg0 cnd outnodes

ciNodeToDfdGraph :: DfdGraph → OperatorNode →
```

```
                    [OperandNode] → DfdGraph
ciNodeToDfdGraph dfg cnd [] = dfg
ciNodeToDfdGraph dfg@(DfdGraph fmo fme) optr (opnd:opnds) =
            ciNodeToDfdGraph (DfdGraph nfmo fme) optr opnds
      where nfmo = addToFM_C (λx y → x++y) fmo opnd [optr]
```

addSignNodes creates two special "Sign" nodes for the MDX branching instruction which contains the sign of two values (before and after MDX instruction) of the specified index register and also the *OperatorNode*s for those "Sign" nodes. It also adds those nodes and edges from them to the "Condition" Node in the DFG. In case of other branching instructions, it does nothing.

```
addSignNodes:: DfdGraph → OperatorNode → OperandNode
                        → OperandNode → DfdGraph
addSignNodes dfg@(DfdGraph fmo fme) cnd@(ConditionVal
                     ce@(BrCondition e cop be bt) i)
       opnd1@(ExpressionNode be1@(VariableX t1) i1)
       opnd2@(ExpressionNode be2@(VariableX t2) i2) =
              if bt ≡ MDX then (DfdGraph nfmo nfme)
                          else dfg
         where signN1 = Unary Sign (i1+2)
               signN2 = Unary Sign (i2+2)
               opnd11 = ExpressionNode (SignBit t1) 1
               opnd12 = ExpressionNode (SignBit t2) 2
               nfmo = addListToFM_C (λx y → x++y) fmo
                           [(opnd1,[signN1]),(opnd2,[signN2]),
                            (opnd11,[cnd]),(opnd12,[cnd])]
               nfme = addListToFM fme [(signN1, OneEdge opnd11),
                                       (signN2, OneEdge opnd12)]
```

conditionTooNodeEdges adds all the edges from the "Condition" node to the branches in the DFG. It also adds different edge labels depending on the branch condition of that branch. coNodeToDfdGraph is used to add those edges.

```
conditionTooNodeEdges :: DfdGraph → Int → Int →
```

```
                        ConditionStmt → Stmt → Stmt → DfdGraph
conditionTooNodeEdges dfg@(DfdGraph fmo fme) di1 di2
                cf@(Check (cde,cd)) stm1 stm2 = dfg2
          where cnd = condToNodes cf (di1-1)
                innode1 = head·fst3 $ dfeToNodes stm1 di1
                innode2 = head·fst3 $ dfeToNodes stm2 di2
                oed = TwoEdge (cd,innode1) ((if cd ≡ Tr
                            then Fl else Tr),innode2)
                dfg2 = coNodeToDfdGraph dfg cnd oed

coNodeToDfdGraph :: DfdGraph → OperatorNode
                              → OEdge → DfdGraph
coNodeToDfdGraph dfg1@(DfdGraph fmo fme) optr ies
                 = DfdGraph fmo (addToFM fme optr ies)
```

`findLeafs` finds all the leaf nodes in one segment of DFG. By leaf nodes, we mean those nodes which do not have any outgoing edges.

`findLeafs` uses `isLeaf` to find whether a node is a leaf or not. `isLeaf` checks the *Int* field of the *OperandNode* with the segment *Int* boundaries to find whether it is a node of that segment. The nodes of the final instruction of the segment labeled as *StateComp* (i.e. *Acc*, *Q*, *AccQ*) which do not have any outgoing edges will be leaf nodes. All other nodes of that segment (except the *StateComp* labeled nodes) with no outgoing edges will also be leaf nodes.

```
findLeafs :: DfdGraph → Int → Int → [OperandNode]
                      → [OperandNode] → [OperandNode]
findLeafs dfg1 si fi [] lopnds = lopnds
findLeafs dfg1 si fi (opnd:opnds) lopnds
          = findLeafs dfg1 si fi opnds
                  $ if (isLeaf opnd dfg1 si fi)
                          then (lopnds++[opnd])
                          else lopnds

isLeaf :: OperandNode → DfdGraph → Int → Int → Bool
```

```
isLeaf opnd@(ExpressionNode e ci) (DfdGraph fmo fme) si fi =
    if (ci < si) ∨ (ci > fi)
       then False
       else case e of
           (Variable _) → (ci ≡ fi) ∧
                          ((fromJust $ lookupFM fmo opnd) ≡ [])
           otherwise → ((fromJust $ lookupFM fmo opnd) ≡ [])
```

After finding the leaf nodes for each code segments, `zipOperandNodes` makes a list of pairs among those nodes where a pair contains similar nodes from two different segments. By similar *OperandNodes*, we mean those nodes which have a similar data label associated with them. For one *OperandNode*, if it can not find any similar node in the other branch then it looks up the *NodeMap* to find another node entry of that label. That node will be from the fc (First Common) segment as this *NodeMap* will only contain the *NodeIds* from the fc segment.

```
zipOperandNodes :: [OperandNode] → [OperandNode]
                 → NodeMap → [(OperandNode, OperandNode)]
                 → [(OperandNode, OperandNode)]
zipOperandNodes [] [] nMap opndpairs = opndpairs
zipOperandNodes (opnd1@(ExpressionNode e1 ci1):opnds1)
          opnds2 nMap opndpairs
           = zipOperandNodes opnds1 newopnds2
              nMap $ if opnd2 ≠ Nothing
                       then opndpairs ++ [(opnd1, fromJust $ opnd2)]
                       else opndpairs ++ [(opnd1, elseopnd2)]
              where opnd2 = find (λ(ExpressionNode e2 ci2)
                                  → e1 ≡ e2) opnds2
                    newopnds2 = if opnd2 ≠ Nothing
                                  then delete (fromJust $ opnd2)
                                       opnds2
                                  else opnds2
                    elseopnd2 = lookupEntry nMap e1
zipOperandNodes [] (opnd2@(ExpressionNode e2 ci2):opnds2)
```

```
                nMap opndpairs =
                  zipOperandNodes [] opnds2 nMap
                    (opndpairs ++ [(newopnd1, opnd2)])
            where newopnd1 = lookupEntry nMap e2
```

addJoinNodes is a special function to add "Join" nodes between the similar *OperandNodes* in the two branches. For each pair of *OperandNodes*, it creates a "Join" node that includes the two *Int* field in the *OperandNodes*. In this way, we can distinguish among different "Join" nodes. It also creates edges from the *OperandNode* pair to the "Join" node and a new data node which has an incoming edge from the "Join" node.

```
addJoinNodes :: DfdGraph → Int → NodeMap → [(OperandNode,OperandNode)]
                                    → (Int,DfdGraph, NodeMap)
addJoinNodes dfg1 i nMapsc [] = (i, dfg1, nMapsc)
addJoinNodes dfg1@(DfdGraph fmo fme) i nMapsc
             ((opnd1@(ExpressionNode e1 ci1),
             opnd2@(ExpressionNode e2 ci2)):opndpairs)
              = addJoinNodes (DfdGraph nfmo nfme) (i+1)
                             newnMapsc opndpairs
          where newJoin = Join ci1 ci2
                nfmo = addListToFM fmo [(opnd1, [newJoin]),
                                        (opnd2, [newJoin])]
                newopnd1 = ExpressionNode e1 i
                nfme = addToFM fme newJoin (OneEdge newopnd1)
                newnMapsc = addEntry nMapsc (e1, newopnd1)
```

9.5 Garbage Collection

Garbage means unwanted or useless material. By garbage collection we mean to remove those unused materials from the final output. When generating Data Flow Graph (DFG), for many instructions we have to generate some data and operator nodes that might be used by other nodes of the following instructions. However after the DFG is generated some of them might not be used at all. We may hence have

some nodes that are not used in the code by other operations. Not all of the unused data nodes are garbage. The output data nodes of the last instruction are not used in the current code segment but they may still be useful in the next one. That is why we can declare a node Garbage as the node which is not the latest of its kind (i.e. after this node there is at least one occurrence of this data as output of other instructions).

In this module we garbage collect all the nodes from a DFG which might have redundant (garbage) data nodes.

```
module GarbageCollect
(markNodes, garbageCollectOfDfdGraph)
where

import Dfg
import Exp (BasicExp (..))
import Stack
import Data.FiniteMap (keysFM, addListToFM, addToFM, delFromFM,
                       lookupFM)
import Data.Maybe (fromJust)
```

The strategy to clean the DFG is as follows: We use ***UsedNode*** data structure with two finitemaps (one for operand nodes and the other for operator nodes) to determine whether a node is used or not. First we mark all the operand nodes as used or unused by taking a look at its successor list. We then use the finitemap for the operand nodes in the ***UsedNode*** to mark the unused operators. After marking all the nodes, we just remove them from the DFG data structure if they are unused and also not the latest of their type.

`markNodes` marks all the nodes in the DFG with their ***UseType***. As we mentioned earlier, it first marks the ***OperandNode***s using the `markOpnds` function and then the ***OperatorNode***s with the `markOptrs` function.

When marking an ***OperandNode*** as ***Unused***, we have to check three conditions: (1) Whether its successor list is empty (2) It is not a part of the last instruction and (3) It is not the latest value of its kind. `retUseType` verifies those conditions and returns the ***UseType*** for each ***OperandNode***. By looking at the integer value in the

OperandNode and its successor list from the finitemap of *DfdGraph*, we can determine the first two conditions. *NodeMap* in the `retUseType` is a mapping of a data value to its most recent *NodeId*. By searching the *NodeMap*, we can determine the last condition and then mark the *OperandNode* with its *UseType*.

```
markNodes:: DfdGraph → Int → UsedNode → NodeMap → UsedNode
markNodes dfg1@(DfdGraph fmon fmoe) i (UsedNode fmoun fmoue) nMap =
          (UsedNode fmNun fmNue)
        where opnds = keysFM fmon
              optrs = keysFM fmoe
              fmNun = addListToFM fmoun $ markOpnds
                        dfg1 i nMap opnds []
              fmNue = addListToFM fmoue $ markOptrs
                        dfg1 (UsedNode fmNun fmoue) optrs []


markOpnds :: DfdGraph → Int → NodeMap → [OperandNode]
     → [(OperandNode, UseType)] → [(OperandNode, UseType)]
markOpnds dfg1 i nMap [] opus = opus
markOpnds dfg1 i nMap (opnd:opnds) opus =
               markOpnds dfg1 i nMap opnds
                    (opus++[(opnd,uType)])
     where uType = retUseType opnd i nMap dfg1

retUseType :: OperandNode → Int → NodeMap
                    → DfdGraph → UseType
retUseType opnd@(ExpressionNode e ci) i nMap
               (DfdGraph fmon fmoe)
       = case e of
            (Variable _) → if (ci ≠ i) ∧
                              ((fromJust $ lookupFM fmon opnd) ≡ [])
                                 then if boolUse then Used
                                                 else Unused
                                 else Used
```

```
            otherwise → Used
   where boolUse = (lookupEntry nMap e ≡ opnd)
```

After marking the *OperandNode*s, we start marking the *OperatorNode* with `markOptrs`. Without loss of generality, we can safely assume that all the *OperatorNode*s (except the *Unary* node) are used in the DFG. So we only check the *Unary OperatorNode*s to determine their *UseType*. One advantage of the *Unary* nodes are that the *OEdge* from them will always be *OneEdge* (i.e. only one edge comes out from those nodes). So we check the *UseType* of the successor *OperandNode*. If its used then the *UseType* of the *OperatorNode* is *Used* otherwise opposite.

```
markOptrs :: DfdGraph → UsedNode → [OperatorNode]
             → [(OperatorNode, UseType)] → [(OperatorNode, UseType)]
markOptrs dfg1 used1 [] optus = optus
markOptrs dfg1@(DfdGraph fmon fmoe)
          used1@(UsedNode fmoun fmoue)
          (optr@(Unary tc i):optrs) optus =
            markOptrs dfg1 used1 optrs (optus++[(optr,uType)])
  where opnd = if (lookupFM fmoe optr ≡ Nothing)
                     then error "should not happen"
                     else fromJust $ lookupFM fmoe optr
        on = case opnd of
               (OneEdge on1) → on1
               _ → ExpressionNode (Constant 0) 0
        uType = if on ≠ (ExpressionNode (Constant 0) 0)
                  ∧ ((fromJust $ lookupFM fmoun on) ≡ Unused)
                            then Unused
                            else Used
markOptrs dfg1 used1 (optr:optrs) optus =
            markOptrs dfg1 used1 optrs (optus++[(optr,Used)])
```

After marking all the nodes with their *UseType* in *UsedNode*, we garbage collect the nodes in the *DfdGraph*.

`garbageCollectOfDfdGraph` is the main function to remove the garbage nodes from the DFG. First it removes the garbage *OperandNode*s with

garbageCollectOfOpnds and then removes the garbage *OperatorNode*s with garbageCollectOfOptrs to give the final garbage collected Data Flow Graph.

```
garbageCollectOfDfdGraph :: DfdGraph → UsedNode → DfdGraph
garbageCollectOfDfdGraph dfd1@(DfdGraph fmo fme)
                         used1@(UsedNode fmuo fmue)
        = (DfdGraph fmNn fmNe)
      where opnds = keysFM fmuo
            optrs = keysFM fmue
            (DfdGraph fmNn fme) = garbageCollectOfOpnds dfd1
                                                  used1 opnds
            (DfdGraph fmo fmNe) = garbageCollectOfOptrs dfd1
                                                  used1 optrs
```

In garbageCollectOfOpnds, it first finds the *UseType* of each *OperandNode*. If it is *Unused* then it removes the *OperandNode* from the finitemap otherwise it updates the successor *OperatorNode* list by garbageCollectOfLst.

```
garbageCollectOfOpnds :: DfdGraph → UsedNode → [OperandNode] → DfdGraph
garbageCollectOfOpnds (DfdGraph fmo fme) used1 [] = (DfdGraph fmo fme)
garbageCollectOfOpnds (DfdGraph fmo fme)
                      used1@(UsedNode fmuo fmue) (opnd:opnds)
          = garbageCollectOfOpnds (DfdGraph fmNo2 fme)
                                  (UsedNode fmuo fmue) opnds
        where optrLst = fromJust $ lookupFM fmo opnd
              updatedLst = garbageCollectOfLst optrLst used1 []
              fmNo0 = delFromFM fmo opnd
              fmNo1 = addToFM fmNo0 opnd updatedLst
              fmNo2 = if ((fromJust $ lookupFM fmuo opnd) ≡ Unused)
                                    then delFromFM fmo opnd
                                    else fmNo1


garbageCollectOfLst:: [OperatorNode] → UsedNode
```

```
                    → [OperatorNode] → [OperatorNode]
garbageCollectOfLst [] used1 optrss = optrss
garbageCollectOfLst (optr:optrs) used1@(UsedNode fmuo fmue) optrss =
        garbageCollectOfLst optrs used1
              $ if (fromJust $ lookupFM fmue optr) ≡ Unused
                    then optrss
                    else optrss++[optr]
```

In garbageCollectOfOptrs, it just removes the *OperatorNode*s with *Unused UseType*.

```
garbageCollectOfOptrs :: DfdGraph → UsedNode → [OperatorNode]
                                                → DfdGraph
garbageCollectOfOptrs dfd1 uNode [] = dfd1
garbageCollectOfOptrs (DfdGraph fmo fme) (UsedNode fmoe fmue)
                                        (optr:optrs)
                      = garbageCollectOfOptrs (DfdGraph fmo fmNe)
                                              (UsedNode fmoe fmue)
                                               optrs
       where fmNe = if ((fromJust $ lookupFM fmue optr) ≡ Unused)
                                       then delFromFM fme optr
                                       else fme
```

9.6 DFG Examples

The examples cited here are for the segments taken from the Boiler Pressure Control (BPC) code of OPG.

9.6.1 SC Example (Before Garbage Collection)

This SC segment is adapted from Section 2.3.1:

```
OADDR REL OBJ. S.NO. LABEL OPCD FT OPRNDS
 35B6 0  C129  0677  TRBFB  LD   1  41
```

```
35B7 0  A12A  0678          M    1  42
35B8 0  1082  0679          SLT     2
35B9 0  912B  0680          S    1  43
35BA 0  A12C  0681          M    1  44
35BB 0  108F  0682          SLT     15
35BC 0  A92D  0683          D    1  45
35BD 0  D12E  0684          STO  1  46
```

The Data Flow Graph (DFG) before garbage collection is in Figure 9.2:

9.6.2 SC Example (After Garbage Collection)

During garbage collection, we remove all the redundant entries from the DFG. In Figure 9.3, we present the same DFG in the Subsection 9.6.1 after garbage collection. The DFG gives us a clear indication of data dependency inside the function which is not easily visible in the DFEs.

9.6.3 GSC Example

This following is the GSC segment from Section 8.6.2:

```
35C4 0 73FF  0695         MDX   3 -1
35C5 0 700F  0696         MDX     TRBFE
             0697
35C6 0 1010  0698 TRBFD   SLA     16
35C7 0 D12F  0699         STO   1 47   0
35C8 0 7012  0700         MDX     TROUT
             0701
             0702
35C9 0 0000  0703 DI2F3   DC      0
35CA 0 6203  0704         LDX   2 3
35CB 0 6300  0705         LDX   3 0
35CC 0 4810  0706         BSC     -
35CD 0 7301  0707         MDX   3 1
```

```
35CE 0 1001  0708        SLA     1
35CF 0 72FF  0709        MDX   2 -1
35D0 0 70FB  0710        MDX     *-5
35D1 00 66002099 0711    LDX   L2 BPCD
35D3 00 4C8035C9 0712    BSC   I  DI2F3
             0713
             0714
             0715
35D5 0 C209  0716 TRBFE  LD    2 9
35D6 0 911B  0717        S     1 27
35D7 0 A130  0718        M     1 48
35D8 0 1005  0719        SLA     5
35D9 0 D12F  0720        STO   1 47
35DA 0 7000  0721        MDX     TROUT
             0722
             0723
             0724
             0725
35DB 0 C12E  0726 TROUT  LD    1 46
35DC 0 812F  0727        A     1 47
35DD 0 A132  0728        M     1 50
35DE 0 1089  0729        SLT     9
35DF 0 D123  0730        STO   1 35
```

Figure 9.4 shows the DFG for this GSC segment. Data dependencies among the segments in the GSC can be identified from this DFG.

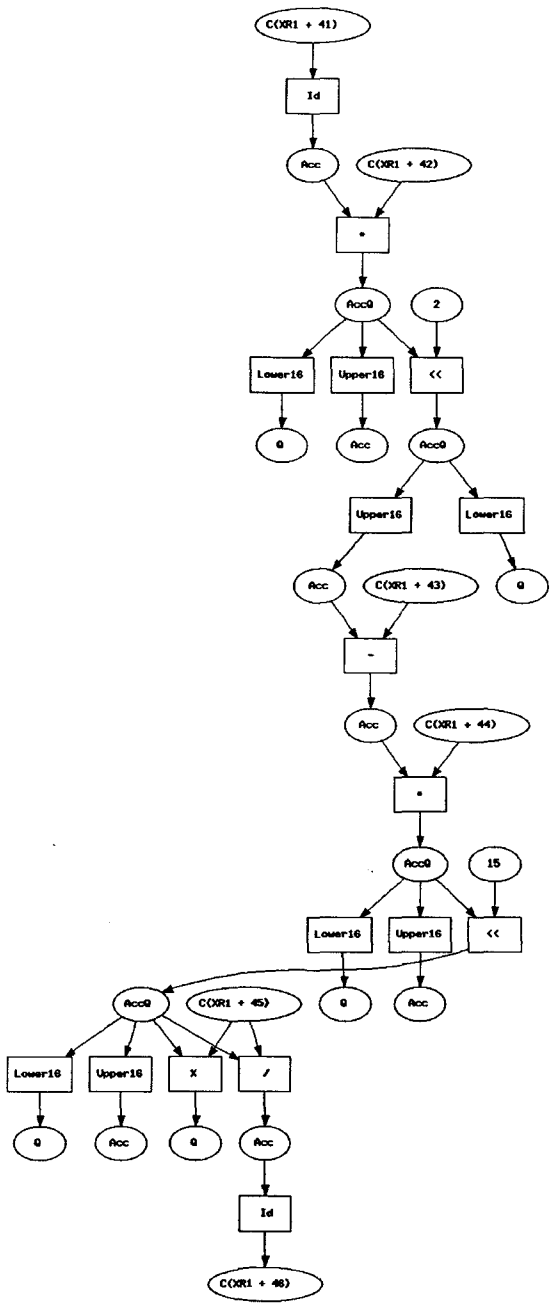


Figure 9.2: Data Flow Graph of the Segment 0x35B6-0x35BD (Before Garbage Collection)

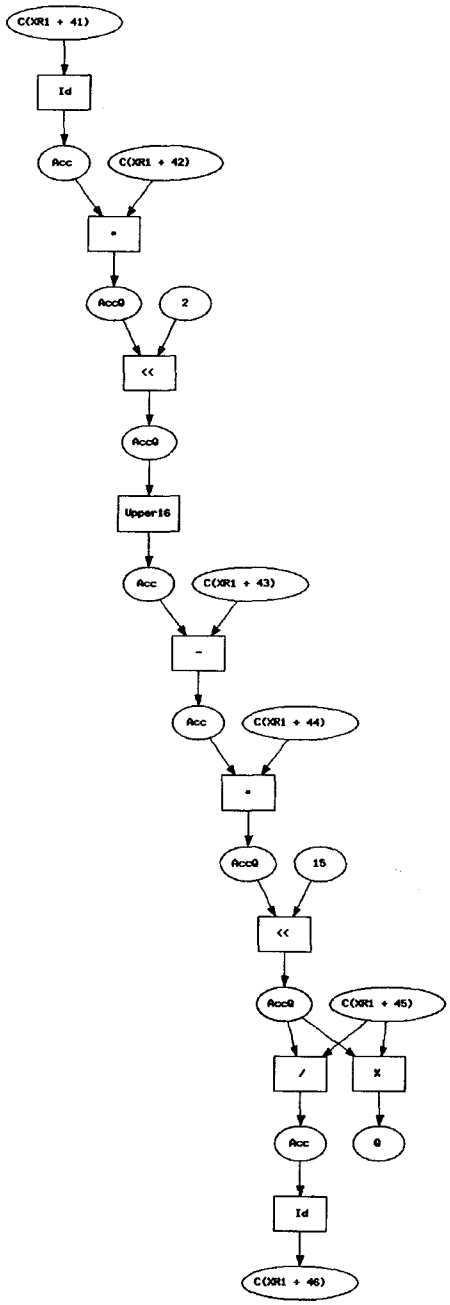


Figure 9.3: Data Flow Graph of the Segment 0x35B6-0x35BD (After Garbage Collection)

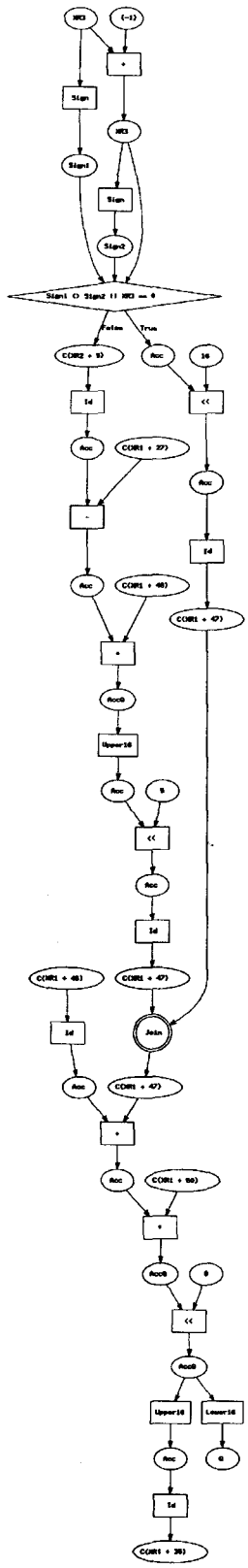


Figure 9.4: Data Flow Graph of the Segment 0x35C4-0x35DF

Chapter 10

Solving Data Flow Equations

In this chapter, we present the method for solving the Data Flow Equations (DFE) which are generated by the symbolic interpretation of assembler programs. Later, we include some examples of solved DFEs for different patterns of program segments.

10.1 Introduction

In general, to solve an equation for a given variable, we need to "undo" whatever has been done to the variable [Wic05]. In our Data Flow Equations (DFEs) generated from the assembler code, we get a sequential set of statements which represents the computation done in the given code. As we mentioned earlier, a statement assigns a symbolic value (represented as an expression of symbolic variables) to a variable. Each statement is a DFE which is comprised of some variables and an operator. It shows the flow of data to and from the operator that is which variables are taken as input in that operator and which variable is given as output. In solving DFEs symbolically, we follow two steps to find the solved flow equations for each DFE:

- Evaluate the variables on the right hand side (expression part) of the statement. By evaluation of variables, we mean to find a symbolic value (if any) represented as an expression which has been assigned to that variable by the previous DFEs.
- Substitute that symbolic value in place of the variable with the operator of the statement in place.

We can make this discussion more precise by citing a simple example. Let us consider the ordered equations $A = F(X, Y); B = G(A, X, Z)$. The first equation has the variable on the right hand side which does not have any previous values assigned. So first equation is automatically solved. The second equation has A, X, Z as input, but since A is already assigned a value by the previous equation, it can be substituted. So the second equation becomes $B = G(F(X, Y), X, Z)$. Consequently the solved set of equations will be $A = F(X, Y); B = G(F(X, Y), X, Z)$.

Thus we will find a solved expression for the output variable of a statement with all its input variables evaluated and replaced by the symbolic value.

After solving each of the DFEs, we start to find the system of equations that summarizes the computation done in the code. The system of equations shows the input and output relationship of the assembler code. So at this stage, we determine the inputs and outputs of the code and find those solved equations where the outputs are represented as the function of inputs.

We can again consider the previous example to show how to find the inputs and outputs of a set of equations. Initially, the empty code sequence has neither input nor output. By proceeding inductively, the first equation tells us that X, Y are part of the "input", and A the output. The second equation has A, X, Z as input, but since A is already known to be an output of the system, it can be eliminated. More precisely, the inputs of equation n are the free variables of the right-hand side of equation n, minus the outputs from stage $n - 1$, union the inputs from stage $n - 1$. The output variables at stage n is the output of stage $n - 1$ union the variable on the left-hand side of stage n. Working this through, the above has X, Y, Z as inputs and A, B as outputs.

10.2 Finding System of Equations

10.2.1 Solving Data Flow Equations

In this Module, from a list of symbolic interpretations of some instructions, we find functional expressions of them. By function expression, we mean to represent the outputs of the code as functions of the inputs symbolically. The functions in this module are used to evaluate the expressions, find inputs and outputs in a code segment

and give the input output relationship (Data Flow Equations in functional expression form) of the code segment.

```
module SolveExpr (Input, Output, dividePathCondSym)
where


import MyPrelude (fst3, snd3, thrd3)
import MyGraph (MyGraph, MyNode)
import Symbolic (StateComp (..), Operator (..), TypeCast (..))
import OpCode (Tag(..))
import Stack
import FindExpr (findAnntOfGraph)
import Exp
import Data.List (nub, delete)
```

Next we define some useful types for the functions in this module.

We want to create a way to look up the definition of the current value of a *StateRef*. It depends on the path of evaluation taken from the start of the CFG. We keep the history of that path in a Stack. The result is a pair consisting of a *ConditionStmt* that expresses the path condition and an actual *Exp*.

```
type ConditionalValue = (ConditionStmt, Expression)

instance Stack StateRef ConditionalValue where
    createStack (SC a) = (Check (NoCondition,NC),
                          Expression $ Atomic $ Variable a)
    createStack (SCX a) = (Check (NoCondition,NC),
                          Expression $ Atomic $ VariableX a)
    createStack (Mem a) = (Check (NoCondition,NC),
                          Expression $ Atomic $ MemoryConstant a)
    addEntry s (k,v) = λd → if k≡d then v else s d
    addEntries s l = foldl addEntry s l
    lookupEntry s k = s k
```

CondAndStmts gives a name to the pair of conditions and the list of statements in one path of the control flow graph of an assembler code.

The second type *EvalHistory* is an evaluation environment which contains the evaluation of the statements with the conditions to reach that statement in a path of statements . This evaluation depends on the previous statements in the path of the control flow graph.

```
type CondAndStmts = ([ConditionStmt],[Stmt])
type EvalHistory = StateRef → ConditionalValue
```

The types for inputs and outputs are also declared as a list of state references.

```
type Input = [StateRef]
type Output = [StateRef]
```

Now, we can find all the annotations of the execution paths in a Graph. We use some other functions from modules like `findExpr` and *Stack* to evaluate the annotations of paths and find the symbolic interpretation and condition (on which the path will be executed) of all the paths in the graph to find the functional representation of the code.

Given a start node and a graph, `solveAnntOfGraph` finds path conditions and symbolic interpretations of the instructions of all the paths in the graph. First it annotates all the edges in the Graph using `doAnnotation` function of *MyGraph* module. Then it finds all the paths in the graph. As we said earlier, according to number of paths and their structures, we can divide the codes into Straight-line Code (SC), Generalized Straight-line Code (GSC) and looping code (LC).

Straight line codes mean those codes that have a straight line control flow. Modeling the control flow and finding the semantic context of straight line codes are easier. As there are no branch or jump in these codes, so the path condition is always TRUE.

If we go through the control flow graph then we can find one execution path only. For that path, we take each instruction from the starting node and find the statement data structure for it using the abstract data type defined in Chapter 8. Then we evaluate each statement sequentially using a stack environment and find the appropriate expression for each state reference

For SC, `solveAnntOfGraph` gets the list of edge annotations using `annotationOfPaths` and transforms them to statements by `transformToStmts`. Then it evaluates them by `evalofStmts`.

GSC (or branching codes) can be considered as a split and join structure. The node where the two paths of GSC splits is called the "split" node and the node (if there is any) where two paths again join is called the "join" node. We divide the two paths into different segments using the "split" and "join" nodes and evaluate them separately to find the functional representation of the whole code.

For GSC, the evaluation process is a little bit different. `solveAnntOfGraph` uses other functions like `dividePathAnt`, `mergePathCond` and `evalOfPaths` for the evaluation. We leave the solving of looping code DFEs for future work.

`getStmts` and `getLastStack` are two small helping functions to split the statement list and the final evaluation environment from the output of evaluation process.

```
solveAnntOfGraph :: [[([ConditionStmt], [Stmt])]]
                    → (ConditionStmt,[Recur_Stmt],EvalHistory)
solveAnntOfGraph cstmts
      = if (length $ concat cstmts) ≡ 1
           then (Check(NoCondition,NC),(init·showStmts) csList,
                                              snd $ head csList)
           else (fst cfList, getStmts cfList, getLastStack cfList)
  where csList = (evalOfStmts createStack)
                 ([Check(NoCondition,NC)],(concat $ map snd
                                    $ concat cstmts))
        cfList = evalOfPaths $ concat cstmts

getStmts :: (ConditionStmt,[[(Recur_Stmt,EvalHistory)]])
                              → [Recur_Stmt]
getStmts cstmts = concatMap (init · showStmts) $ snd cstmts

getLastStack :: (ConditionStmt, [[(Recur_Stmt,EvalHistory)]])
                              → EvalHistory
getLastStack cstmts = snd $ head·head $ snd cstmts
```

`evalOfPaths` evaluates all the segments (fc, pd1, pd2, sc) in two paths of the GSC and finds the conditions of the paths with evaluated statements in those paths. It completely depends on the ordering of the segments generated in `dividePathAnt` and evaluates the segments according to the ordering. For example, if there is any Second Common (sc) segment between those two paths, then the evaluation of the statements in sc will depend on the conditional values generated from pd1 and pd2. Similarly the evaluation of statements in pd1 and pd2 depends on the values from fc (First Common) segment.

```
evalOfPaths :: [([ConditionStmt], [Stmt])]
                  → (ConditionStmt,[[(Recur_Stmt,EvalHistory)]])
evalOfPaths [([Check(NoCondition, NC)],st1),cst2,cst3]
              = (head (fst cst3),[sst3,sst2,sst1])
     where sd = snd·head
           sst1 = evalOfStmts createStack
                    ([Check(NoCondition,NC)],st1)
           sst2 = evalOfStmts (sd sst1) cst2
           sst3 = evalOfStmts (sd sst1) cst3
evalOfPaths [cst4,cst5,([Check(NoCondition, NC)],st6)]
              = (Check (NoCondition, NC),[sst6,sst5,sst4])
     where sd = snd·head
           sst4 = evalOfStmts createStack cst4
           sst5 = evalOfStmts createStack cst5
           sst6 = evalOfStmts1 (sd sst4) (sd sst5)
                     ([Check(NoCondition,NC)],st6)
evalOfPaths [cst7,cst8,cst9,cst10]
               = (head (fst cst10),[sst10,sst9,sst8,sst7])
      where sd = snd·head
            sst7 = evalOfStmts createStack cst7
            sst8 = evalOfStmts (sd sst7) cst8
            sst9 = evalOfStmts (sd sst7) cst9
            sst10 = evalOfStmts1 (sd sst8) (sd sst9) cst10
```

After performing the transformation of all the low level functions in the instructions of the code into statements, our main task is to evaluate them to find the functional expression of the outputs that is to represent the outputs as equations of the inputs of that code.

`evalOfStmts` evaluates a list of statements of a Straight-line Code (SC) segment. It takes an environment and *CondAndStmts* and gives us the evaluated statement and the *EvalHistory* after the evaluation of each statement in the list.

`evalOfStmts1` also does the same thing but it works specially for the straight line code after the "join" in the Generalized Straight-line Code (GSC). It also takes two evaluation environments as arguments from the two different paths in the GSC code.

At the starting, *Stmt* is initialized as a fictitious value *XR0* = 0 which works as the bottom of the evaluation environment. This is done intentionally so that we can determine the bottom easily as no other instructions can generate *XR0* = 0.

```
evalOfStmts :: EvalHistory → CondAndStmts
               → [(Recur_Stmt,EvalHistory)]
evalOfStmts stk (cf, stmts)
          = foldl (interpret (head cf)) [((Assignt (SCX XR0,
            eToE $ Atomic $ Constant 0)),stk)] stmts

evalOfStmts1 :: EvalHistory → EvalHistory → CondAndStmts
                            → [(Recur_Stmt,EvalHistory)]
evalOfStmts1 stk1 stk2 (cf,stmts)
          = foldl (interpret1 stk1 stk2
            (head cf)) [((Assignt (SCX XR0, eToE $ Atomic
                        $ Constant 0)), createStack)] stmts
```

This simple helper function separates the evaluated statements for output purpose.

```
showStmts :: [(Recur_Stmt,EvalHistory)] → [Recur_Stmt]
showStmts = map fst
```

The following functions are used to evaluate the expression and interpret the statements in symbolic form. As defined earlier, each statement represents one function of an instruction and right-hand side of a statement is an expression. We use

EvalHistory (a lookup environment which contains all the previously evaluated expressions) to find the proper expression.

eval takes an *Expr* for evaluation and looks up in the evaluation environment to find any previous evaluated entries for that *Expr* and returns that evaluated expression. We use this function to evaluate expressions for SC segments.

```
eval :: Expr → EvalHistory → Expression
eval exp stk =
 case exp of
   Atomic (Variable st)  → snd $ lookupEntry stk (SC st)
   Atomic (VariableX st) → snd $ lookupEntry stk (SCX st)
   Atomic (MemoryConstant st) → snd $ lookupEntry stk (Mem st)
   (UnaryOperation (op,ex1)) →
    let a = eval (Atomic ex1) stk
    in Expression $ UnaryOperation (op,a)
   BinaryOperation (ex1,op,ex2) →
    let a = eval (Atomic ex1) stk
        b = eval (Atomic ex2) stk
    in Expression $ BinaryOperation (a,op,b)
   otherwise → (eToE exp)
```

eval1 also takes an *Expr* and condition as argument and returns the evaluated expression. This is a special function for evaluation and works only for the straight line code segment after the "join" in the GSC. It takes three evaluation environments (two for previous two paths in the GSC and one for current code segment after the join) and returns the evaluated expression using three environments. It also returns the new evaluation environment with the evaluated *Expr* included in it.

```
eval1 :: Expr → ConditionStmt → EvalHistory → EvalHistory
              → EvalHistory → (Expression,EvalHistory)
eval1 exp cf stk1 stk2 stk =
 case exp of
   Atomic (Variable st)  → getPropExpr (Atomic $ Variable st)
                                (SC st) cf stk stk1 stk2
```

```
Atomic (VariableX st) → getPropExpr (Atomic $ VariableX st)
                                    (SCX st) cf stk stk1 stk2
Atomic (MemoryConstant st) → getPropExpr (Atomic
                                     $ MemoryConstant st)
                                    (Mem st) cf stk stk1 stk2
UnaryOperation (op,ex1) →
 let (a,b) = eval1 (Atomic ex1) cf stk1 stk2 stk
 in (Expression (UnaryOperation (op,a)), b)
BinaryOperation (ex1,op,ex2) →
 let ex = eval1 (Atomic ex1) cf stk1 stk2 stk
     a = fst ex
     b = eval1 (Atomic ex2) cf stk1 stk2 (snd ex)
 in (Expression (BinaryOperation (a,op,(fst b))), (snd b))
ConditionalExp ((cf1,ex1),(cf2,ex2)) →
 let ex = eval1 (Atomic ex1) cf stk1 stk2 stk
     a = fst ex
     b = eval1 (Atomic ex2) cf stk1 stk2 (snd ex)
 in (Expression (ConditionalExp ((cf1,a),(cf2,(fst b)))),
                    snd b)
otherwise → (eToE exp, stk)
```

To find the proper expressions for the terms of the statements after the "join" in the GSC code is a little bit complex. Each time we get an expression to be evaluated, we have to look up the current evaluation environment to find an entry for it. If we can't find any, we take a look at two previous environments for the different paths. A conditional expression is formed for the corresponding expression using the expressions found in those two environments and is also added in the current evaluation environment.

`getPropExpr` does this for `eval1` and looks up all the three evaluation environments to return the proper evaluated form (either conditional or unconditional) of the expressions and a new evaluation environment with the new *Expr* in it.

```
getPropExpr :: Expr → StateRef → ConditionStmt → EvalHistory →
               EvalHistory → EvalHistory → (Expression,EvalHistory)
```

```
getPropExpr exp asb cf stk stk1 stk2 =
    if condCheck (snd $ lookupEntry stk asb)
         then (let cft1@(cf1, tf1) = lookupEntry stk1 asb
                   cft2@(cf2, tf2) = lookupEntry stk2 asb
                   ex1 = Expression $ ConditionalExp (cft1,cft2)
               in (if (condCheck tf1) ∧ (condCheck tf2)
                    then (snd $ lookupEntry stk asb, stk)
                    else (ex1, addEntry stk (asb,(cf,ex1)))))
         else (snd $ lookupEntry stk asb, stk)
    where condCheck e =  e ≡ (eToE exp)
```

So far we have introduced functions to find the evaluation of expressions. Now, we define functions to interpret the whole statement which may contain several expressions in it.

`interpret` interprets each statement in the *Stmt* list using the `eval` to evaluate each expression in the statement and returns the new interpreted statement. It also outputs a new evaluation environment which will be used to interpret the next statements in the *Stmt* list. This function is for SC segments.

```
interpret :: ConditionStmt → [(Recur_Stmt,EvalHistory)]
             → Stmt → [(Recur_Stmt,EvalHistory)]
interpret cf prev@((_,stck) : _) (Assign (name,e)) =
      let v = eval e stck
      in addProperEntry name v cf stck prev
```

`interpret1` does the same as `interpret` but it works for the straight line segment after the "join" in the GSC. It takes the evaluation environment for previous two paths and uses `eval1` to evaluate the expressions.

```
interpret1 :: EvalHistory → EvalHistory → ConditionStmt
              → [(Recur_Stmt,EvalHistory)] → Stmt
              → [(Recur_Stmt,EvalHistory)]
interpret1 stk1 stk2 cf prev (Assign(name,e)) =
   let stck = snd $ head prev
```

```
      v = eval1 e cf stk1 stk2 stck
   in addProperEntry name (fst v) cf (snd v) prev
```

addProperEntry is a helping function for interpret and interpret1. It is only used to divide the combined *AccQ* value expression to insert two entries for *Acc* and *Q* in both of the evaluation environment and evaluated *Stmt* list. When the *StateRef* to be assigned in the statement is *AccQ*, we may need distinct *Acc* and *Q* from the combined *AccQ* value in the upcoming statements for evaluation. That is why, we update the evaluation environment and the *Stmt* list by two new statements of *Acc* and *Q* each time we face *AccQ* in the left-hand side of the *Stmt*.

```
addProperEntry:: StateRef → Expression → ConditionStmt
                 → EvalHistory → [(Recur_Stmt,EvalHistory)]
                 → [(Recur_Stmt,EvalHistory)]
addProperEntry name e cf stk prev = case name of
   (SC AccQ) → (Assignt (name,e),(addEntries stk [(name,
               (cf,propExprAQ)), ((SC Acc),(cf,Expression
               $ UnaryOperation (Upper16, propExprAQ))),
               ((SC Q),(cf,Expression $ UnaryOperation
               (Lower16, propExprAQ)))])): prev
   _         → (Assignt (name,e),(addEntry stk (name,
               (cf, e)))): prev
   where propExprAQ = findProperExpr e stk
```

In binary operations where the value is to be stored in *AccQ*, we face two different situations. For Shift operations, we normally shift the whole *AccQ* (first argument) by the amount given in the second argument. But for the operations like multiply etc. the first argument is always *Acc* and we store the value in the *AccQ*. findProperExpr finds the proper expression for the *AccQ* in those cases.

```
findProperExpr :: Expression → EvalHistory → Expression
findProperExpr (Expression (BinaryOperation(ex1,op,ex2))) stck
               = Expression $ BinaryOperation(exx,op,ex2)
 where ex3 = snd $ lookupEntry stck (SC AccQ)
```

```
        ex4 = snd $ lookupEntry stck (SC Acc)
        exx = if op 'elem' [Shl,Shr] then ex3 else ex4
findProperExpr e _ = e
```

10.2.2 Finding Inputs and Outputs

Next, we define some functions to find the set of inputs and outputs of a list of statements. Each basic expression (except the constants) on the right-hand side of interpreted statements can be considered as inputs and all the state references on the left-hand side of statements as outputs (some exceptions remain for Acc and Q), given that all duplications are removed.

One important thing to remember is that the statement list from where we find the inputs and outputs is in the reverse order that is the first statement we face in the list corresponds to the last instruction of the code.

`inputSet` uses a recursive function called `findInput` to find all the inputs from a list of *Stmt* and then to remove the duplicates. In each expression of the right-hand side of the *Stmt*, `findInput` recursively finds all the basic expressions (except the constants) as inputs.

```
inputSet :: [Recur_Stmt] → Input → Input
inputSet [] iset = iset
inputSet ((Assignt (name,exp)):stmts) iset
      = inputSet stmts ((findInput exp) ++ iset)

findInput :: Expression → Input
findInput exp = case exp of
    Expression (Atomic (Constant a)) → []
    Expression (Atomic (MemoryConstant a)) → [Mem a]
    Expression (Atomic (Variable a)) → [SC a]
    Expression (Atomic (VariableX a)) → [SCX a]
    Expression (UnaryOperation (_,ex1)) → findInput ex1
    Expression (BinaryOperation (ex1,_,ex2))
                          → (findInput ex1)
                          ++ (findInput ex2)
```

```
    Expression (ConditionalExp ((_,ex1),(_,ex2)))
                          → (findInput ex1)
                          ++ (findInput ex2)
```

The right-hand side of each *Stmt* will be an output with some exceptions for *Acc*, *Q* and *AccQ*. One exception is like the following: if the next state reference to be included in the output list is *AccQ* and *Acc* is already in the output list then we remove the previous entry of *Acc* from the output list and include *AccQ* and two new entries of *Acc* and *Q* in the list.

appOutPut determines which ones of *Acc*, *Q* and *AccQ* should stay in the output list.

```
outputSet :: [Recur_Stmt] → Output → Output
outputSet [] oSet = oSet
outputSet ((Assignt (name,e)):stmts) oSet = outputSet stmts $
    if (appOutput name oSet) then
        if name ≡ (SC AccQ) ∧ (SC Acc) `elem` oSet then
            (name:(SC Acc):(SC Q):(delete (SC Acc) oSet))
        else (name:oSet)
    else oSet

appOutput :: StateRef → Output → Bool
appOutput name os = case name of
   SC AccQ → SC Acc `notElem` os ∨ SC Q `notElem` os
   SC Acc → SC AccQ `notElem` os
   SC Q   → SC AccQ `notElem` os
   _ → True
```

10.2.3 Finding System of Equations

These functions are used to return the input output relationship from a list of statements and a list of inputs that is they return the Data Flow Equation (DFE)s. By Data Flow Equation, we mean the representation of outputs in terms of input data flow.

For each input that is *StateRef*, `findFstStmt` finds the first evaluated expression of that *StateRef* from the evaluation environment and returns the interpreted *Stmt*.

```
listOfEqns :: [Recur_Stmt] → EvalHistory → Output
                    → [Recur_Stmt] → [Recur_Stmt]
listOfEqns stmts stk [] stmteq = reverse stmteq
listOfEqns stmts stk (os:oList) stmteq
          = listOfEqns stmts stk oList
            ((findFstStmt os stk stmts):stmteq)

findFstStmt :: StateRef → EvalHistory → [Recur_Stmt]
                        → Recur_Stmt
findFstStmt asb stk (st@(Assignt (name,e)):stmts)
   = if (expOfStateRef asb ≡ (snd $ lookupEntry stk asb))
        then if asb ≡ name then st
                           else findFstStmt asb stk stmts
        else Assignt (asb, snd $ lookupEntry stk asb)

expOfStateRef :: StateRef → Expression
expOfStateRef (SC a) = Expression $ Atomic $ Variable a
expOfStateRef (SCX a) = Expression $ Atomic $ VariableX a
expOfStateRef (Mem a) = Expression $ Atomic $ MemoryConstant a
```

This function is mainly used to divide the output of `findAnntofGraph` in the conditions, execution sequence, inputs, outputs and system of equations (input output relationship) of the code segment for the graph.

```
dividePathCondSym :: MyGraph → MyNode
     → (ConditionStmt,[Recur_Stmt],Input,Output,[Recur_Stmt])
dividePathCondSym mg start = ((fst3 cfStmt),scfStmt,input,
                                            output,sysList)
  where cfStmt = solveAnntOfGraph $ findAnntOfGraph mg start
        scfStmt = reverse $ snd3 cfStmt
        stk = thrd3 cfStmt
```

```
output = nub (outputSet scfStmt [])
input = nub (inputSet scfStmt [])
sysList = listOfEqns scfStmt stk output []
```

10.3 Example

10.3.1 Straight-line Code

The following code segment is a straight-line code already cited in section 2.3.1.

```
OADDR REL OBJ. S.NO. LABEL OPCD FT OPRNDS
 35B6 0  C129  0677  TRBFB  LD   1  41
 35B7 0  A12A  0678         M    1  42
 35B8 0  1082  0679         SLT     2
 35B9 0  912B  0680         S    1  43
 35BA 0  A12C  0681         M    1  44
 35BB 0  108F  0682         SLT     15
 35BC 0  A92D  0683         D    1  45
 35BD 0  D12E  0684         STO  1  46
```

The symbolic interpretation for this code segment is given below. As this is a sequential code segment, so the path condition is always TRUE.

```
PathCondition: True

Instruction Execution:

A = C(XR1 + 41),

AQ = C(XR1 + 41)*C(XR1 + 42),
e give an example of a straight line code and
AQ = ((C(XR1 + 41))*C(XR1 + 42))<<2,

A = (Upper16(((C(XR1 + 41))*C(XR1 + 42))<<2))-C(XR1 + 43),
```

```
AQ = ((Upper16(((C(XR1 + 41))*C(XR1 + 42))<<2))-C(XR1 + 43))
                *C(XR1 + 44),

AQ = (((Upper16(((C(XR1 + 41))*C(XR1 + 42))<<2))-C(XR1 + 43))
                *C(XR1 + 44))<<15,

A = ((((Upper16(((C(XR1 + 41))*C(XR1 + 42))<<2))-C(XR1 + 43))
                *C(XR1 + 44))<<15)/C(XR1 + 45),

Q = ((((Upper16(((C(XR1 + 41))*C(XR1 + 42))<<2))-C(XR1 + 43))
                *C(XR1 + 44))<<15)%C(XR1 + 45),

C(XR1 + 46) = ((((Upper16(((C(XR1 + 41))*C(XR1 + 42))<<2))
               -C(XR1 + 43))*C(XR1 + 44))<<15)/C(XR1 + 45)
```

Different inputs, outputs and system of equations (relation among the inputs and outputs) of this chunk of code is shown below:

```
Input: C(XR1 + 41),C(XR1 + 42),C(XR1 + 43),C(XR1 + 44),C(XR1 + 45)

Output: C(XR1 + 46),AQ,A,Q

System of Equations:

C(XR1 + 46) = ((((Upper16(((C(XR1 + 41))*C(XR1 + 42))<<2))
               -C(XR1 + 43))*C(XR1 + 44))<<15)/C(XR1 + 45),

AQ = (((Upper16(((C(XR1 + 41))*C(XR1 + 42))<<2))-C(XR1 + 43))
                   *C(XR1 + 44))<<15,

A = ((((Upper16(((C(XR1 + 41))*C(XR1 + 42))<<2))-C(XR1 + 43))
                   *C(XR1 + 44))<<15)/C(XR1 + 45),
```

```
Q = ((((Upper16(((C(XR1 + 41))*C(XR1 + 42))<<2))-C(XR1 + 43))
                *C(XR1 + 44))<<15)%C(XR1 + 45)
```

If we can give symbolic names of the inputs and outputs then the system of equations will be more clear and easier to understand. Using symbolic names for the input and output variables, the previous system of equations will be:

```
Input:    C(XR1 + 41) : ERR
          C(XR1 + 42) : 1.067
          C(XR1 + 43) : PREV_ERR
          C(XR1 + 44) : K
          C(XR1 + 45) : 5861


Output: C(XR1 + 46) : DELX


System of Equations:

    DELX = (((Upper16((ERR*1.067)<<2)- PREV_ERR)*K)<<15)/5861

    AQ = (((Upper16((ERR*1.067)<<2)- PREV_ERR)*K)<<15)

    Q = (((Upper16((ERR*1.067)<<2)- PREV_ERR)*K)<<15)%5861

    A = (((Upper16((ERR*1.067)<<2)- PREV_ERR)*K)<<15)/5861
```

These equations definitely identifies the computation done by the code. The block comments of the functions inside the assembler program which describe the code show similar equations. These equations can also be used to generate tabular specification of the functions. In this way, we can proceed to find the specification documents of the assembly code.

10.3.2 Generalized Straight-Line Code (GSC):

The modeling of non-sequential branch structure is more complex than sequential codes. It involves finding that kind of structure from the execution paths of a program and then finding data flow equations for that structure. While solving these data flow equations, we have to realize the data dependency in all the segments. This code segment is the same as in Section 8.6.2.

```
35C4 0 73FF  0695        MDX   3 -1
 35C5 0 700F  0696         MDX     TRBFE
              0697
 35C6 0 1010  0698 TRBFD SLA       16
 35C7 0 D12F  0699        STO   1 47    0
 35C8 0 7012  0700        MDX     TROUT
              0701
              0702
 35C9 0 0000  0703 DI2F3 DC        0
 35CA 0 6203  0704        LDX   2 3
 35CB 0 6300  0705        LDX   3 0
 35CC 0 4810  0706        BSC     -
 35CD 0 7301  0707        MDX   3 1
 35CE 0 1001  0708        SLA       1
 35CF 0 72FF  0709        MDX   2 -1
 35D0 0 70FB  0710        MDX     *-5
 35D1 00 66002099 0711    LDX   L2 BPCD
 35D3 00 4C8035C9 0712    BSC   I  DI2F3
              0713
              0714
              0715
 35D5 0 C209  0716 TRBFE LD      2 9
 35D6 0 911B  0717        S      1 27
 35D7 0 A130  0718        M      1 48
 35D8 0 1005  0719        SLA       5
 35D9 0 D12F  0720        STO    1 47
```

```
35DA 0 7000  0721        MDX     TROUT
             0722
             0723
             0724
             0725
35DB 0 C12E  0726 TROUT  LD    1 46
35DC 0 812F  0727        A     1 47
35DD 0 A132  0728        M     1 50
35DE 0 1089  0729        SLT     9
35DF 0 D123  0730        STO   1 35
```

Inputs, Outputs, Path Condition and Solved system of equations of this segment of code are shown below. This system of equations seems a little bit complex since it uses big symbolic names of the variables. If we can use discrete values or small representative symbolic names of the variables, these equations will be more easier to understand. Again, these equations can be used to find tabular specification of the functions which may lead us to find the final specification documents of the assembler program.

```
PathCondition: True

Instruction Execution:

Input: 0, C(XR1 + 46),C(XR2 + 9),C(XR1 + 27),
       C(XR1 + 48),A,C(XR1 + 50),XR3,(-1)

Output: XR3,A,Q,AQ,C(XR1 + 47),C(XR1 + 35)

System of Equations:

XR3 = 0,

A = Upper16((((C(XR1 + 46))+(Upper16(((C(XR2 + 9))-C(XR1 + 27))
  *C(XR1 + 48)))<<5((Sign1 <> Sign2 || XR3 == 0) ==False)
```

```
 :(Acc)<<16((Sign1 <> Sign2 || XR3 == 0) ==True))*C(XR1 + 50))<<9),

Q = Lower16((((C(XR1 + 46))+(Upper16(((C(XR2 + 9))-C(XR1 + 27))
 *C(XR1 + 48)))<<5((Sign1 <> Sign2 || XR3 == 0) ==False)
 :(Acc)<<16((Sign1 <> Sign2 || XR3 == 0) ==True))*C(XR1 + 50))<<9),

AQ = (((C(XR1 + 46))+(Upper16(((C(XR2 + 9))-C(XR1 + 27))*C(XR1 + 48)))<<5
     ((Sign1 <> Sign2 || XR3 == 0) ==False)
     :(Acc)<<16((Sign1 <> Sign2 || XR3 == 0) ==True))
     *C(XR1 + 50))<<9,

C(XR1 + 47) = (Upper16(((C(XR2 + 9))-C(XR1 + 27))*C(XR1 + 48)))<<5
        ((Sign1 <> Sign2 || XR3 == 0) ==False)
        :(Acc)<<16((Sign1 <> Sign2 || XR3 == 0) ==True),

C(XR1 + 35) = Upper16((((C(XR1 + 46))+(Upper16(((C(XR2 + 9))-C(XR1 + 27))
    *C(XR1 + 48)))<<5((Sign1 <> Sign2 || XR3 == 0) ==False)
    :(Acc)<<16( (Sign1 <> Sign2 || XR3 == 0) ==True))
    *C(XR1 + 50))<<9)
```

Chapter 11

Discussion and Future Work

In this chapter, we discuss the contributions we made in the thesis, limitations faced during the research process and future works that may be done depending on the works presented in the thesis.

11.1 Contribution

Reverse engineering can be seen as an opposite process of compilation. In compilation, we create lower level representation of the program from higher level. In reverse engineering, we follow the other way: from the lower level code to high level specifications. That is why, we try to follow the process of reverse compilation by generating flow graphs and incorporating symbolic computation techniques in the process to get different views of the code. We see our main contribution as adapting the tools from symbolic computation, compiler construction (data flow and control flow graphs) and denotational semantics to the situation of understanding and reverse-engineering the semantics of legacy assembler programs. An important contribution of our approach is that we have made only a few simplifying assumptions, and yet managed to derive mathematical descriptions of subprograms. One of the important simplifying assumptions is to ignore carry and overflow conditions during instruction execution. In legacy assemblers, the control flow of the codes is a little bit awkward. Still we can find a way to represent the control flow using mathematical equations; not by high level procedural constructs.

Morris and Filman [MF96], Feldman and Friedman [FF95], Ward [War00] etc. developed systems to translate assembler codes into a high-level language. In our process, instead of finishing up at high-level programming language (which may only "move" our realization up some levels instead of resolving it), we prefer mathematical languages i.e. producing output suitable for both PVS [OSRSC01] and Maple [MGH+01]. As we said earlier, in our process we have integrated compiler techniques with symbolic analysis; highly influenced by Watson and Fidge [WF03] and Fahringer and Scholz[FS03]. Watson and Fidge [WF03] have described a technique for assembler semantics that is based on advanced compiler theory and technology, as well as programming language semantics. Their work is close to ours, except that while they are describing a theoretical framework, we have a working reverse-engineering tool. As we do, they used execution paths to find the semantics of assembly language programs. Fahringer and Scholz[FS03] developed an approach to find the symbolic interpretation of imperative programs written in a high-level language. We applied some of their constructs (path conditions, recurrence equations) in our approach to interpret assembler programs. As assembler programs do not have any predefined control structures, we used an explicit control flow graph to find execution paths, and used them to guide our symbolic interpretation. In this way, we create a newer blend of compiler technologies with symbolic interpretation techniques in our semantic analysis process.

The whole process shown in Figure 4.1 is fully automated. Interaction among modules are without any human intervention making the process more robust and easily verifiable. Still we can get intermediate representation of outputs at different steps to get the inner look of the process. Control Flow Graph generation from the assembler codes, symbolic analysis with Data Flow Graphs – all of these processes are automated. This may inspire the researchers working in the upper layers of the tool suite architecture in the reverse engineering project to make a fully automated stream from the assembler code to high level specifications.

In this thesis, we only deal with three special control flow patterns of assembler programs. More control flow structures can be incorporated in the process using the same techniques followed. As we use execution paths to model control flow of the program, other control flow patterns can also be detected with less effort. As such, we believe that incorporating more control flow patterns will not effect the efficiency

of the tools.

The tools developed in this thesis are "proof of concept". These tools exhibit a concept of symbolic analysis which may be implemented in a larger scale. They are not designed as software products which need rigorous software engineering methodologies to be practiced. We still try to follow important software engineering principles to make those tools efficient.

Effective use of pure functional programming language like Haskell results in a set of tools and modules which are precise, compact and interact well with each other. Special modularization features (Section 3.4.1) of Haskell help us to create hierarchical software architecture. Haskell provides us with good abstraction mechanism and proper use of these abstraction techniques results in smaller, precise and easy reading codes. The tools also interact well with the outside tools and are flexible enough to incorporate new features in them. The codes included in the thesis are easily comprehensible and we hope that any fellow student will be able to translate them into different architecture within a limited time. The documentation style of literate programming also increases the understandability of the code. In brief, we are convinced that Haskell is a good programming language to produce well structured software abiding by all software engineering principles.

Making some successful contribution in the tool suite architecture of the reverse engineering process was not our only goal. We try to publish the outcome of the research done in this thesis as some scholarly publications. In this regard, one of our paper "Symbolic Interpretation of Legacy Assembly Language" is accepted and will be published in the 12th Working Conference on Reverse Engineering (WCRE), Pittsburgh PA(Carnegie Mellon),USA, November 8 to 11.

11.2 Limitations

When I was first introduced in the project, Dr. Jacques Carette told me to use Haskell as an implementation language. He has described all the important features of Haskell for good programming practice and inspired me to use Haskell. Haskell was the first functional programming language that I have ever used. My programming style was more of imperative genre and it was difficult for me to adapt the style of functional programming. Although I always try to overcome this shortcoming, one

might find some imperative style of coding in some parts of my Haskell codes.

I always try to explain the terse functions where some convoluted and obtuse codes appear in sufficient prose with literate Haskell programming style. Still the clarity of the code might be improved by using the extended abstraction mechanisms of Haskell. More refactoring can be done to make the code more clear and understandable and redundant code can also be eliminated.

Since our goal is an automated process, less amount of work is done on producing better displays for the outputs of intermediate steps. We always try to make the intermediate step outputs more human readable. Still some further works can be done to create better presentation of those outputs.

At the beginning of the project, we aimed to (automatically) identify functions in a program, extract nice closed-form formulas and pre/post-conditions that express the actual (or idealized) semantics of those functions. Automatically identifying functions and finding pre/post-conditions are not covered in this thesis. The time limit of the completion of Masters' degree is an obstacle in this process. Though I hope that some future works will be done on this symbolic interpretation process to attain the ultimate goal.

11.3 Future Works

The solution presented in this thesis is still preliminary. As indicated earlier, only some special kinds of control flow are currently analyzed. Work can be done on analyzing more complex control flow – quite a bit of weird control flow can be recast in our settings by a simple duplication of some of the code. As said in Section 11.2, automatically identifying functions in an assembler program is another area where further effort can be given.

Moreover, we have shown ways to generate the DFEs for all three kinds of control flow graphs. We have produced the Data Flow Graphs (DFG) and solved the DFEs for SC and GSC structures. These works can be given a final touch by implementing the functions for producing DFG and solving DFEs for the looping codes. Duplication of the codes implemented with some adjustment would work for looping codes.

As well, future work can be done on simplifying the output expressions. During simplification, duplicate constructs can be eliminated in the DFEs and also we can

build new constructs for common parts in the DFEs and make reference to those common parts to generate new DFEs which are clear and easily readable. New tools can also be developed to push boolean conditions through systems of (more complex) data flow equations.

Redundant descriptions of the semantics can also be explored; for example a left-shift instruction can be described either as a left-shift or as a multiplication by 2. The "best" description usually depends on how that value is used in other contexts. If it is used in a context of arithmetic operations, then multiplication by 2 is likely more descriptive, whereas if used in a context of bit operations, a shift is likely better. The work of [KAC04] is relevant here.

11.3.1 Graph Transformer

As the legacy assembly language codes were written long time ago, the control flow structures are not in procedural constructs and not easy to interpret. A new step can be included in the semantic analysis process to find the bad control flow structures in the code and replace them by good control flow patterns. We have done some work on this step and tried to find bad control flow patterns to replace them with pre-defined good flow patterns.

Another important aspect is that any time we can find some paths as infeasible, they may be eliminated. This simplifies the analysis tremendously. So our current analysis should be seen as one step in a fixed-point analysis, where each analysis pass provides a better approximation to the complete semantics.

Right now, the control flow graph we are dealing with is an approximation of the precise control flow graph. Some run time emulation of the codes can be done and we can find the actual addresses for the indirect addressing. This can be helpful for modifying the control flow graph by actual edges between nodes to get the accurate control flow graph.

11.3.2 Finding Preconditions

Often we are interested about the partial correctness of programs. A program can be defined as partially correct, with respect to a given precondition and a postcondition, if the initial state satisfies the precondition and if the program terminates, the final

state satisfies the postconditions. Now if we are given postconditions for the program, we can try to use the data flow equations to "push backwards" (as in backwards state transformers) these predicates for obtaining preconditions.

Additionally, as no exceptions (in other words carry, overflow etc.) are handled by the programs we reviewed, we can add these as post-conditions as well, and also propagate them backwards through the data flow equations. Let us consider the control flow graph in Figure 11.1. Suppose that a, b, c are all state variables in this context. At node Q, the initial state is $I = \{a \mapsto a_0, b \mapsto b_0, c \mapsto c_0\}$, and at node P, assume that we have $c = a + b$. As we know that c must be a valid value ($< 2^{16}$ on the IBM-1800), we can conclude that $a + b < 2^{16}$ is true at P. We can push this backwards to (potentially) derive additional necessary conditions at Q, as well as pushing it forward to (potentially) find a more precise description of the state at R, by potentially removing some infeasible paths.

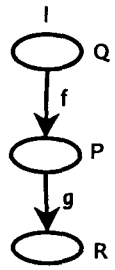


Figure 11.1: Finding Preconditions

Appendix A

Semantic Model of Instructions

This appendix defines the semantic model of the operational semantics of IBM-1800 assembler instructions. We use this abstract model in the whole symbolic interpretation process.

A.1 Semantics of Instructions

We define our model as a state transformer where the effect of executing an instruction is stated as a total function on states i.e. after execution of each instruction, the state changes to one form to another. It can be defined as:

$$[\![\texttt{Instruction}]\!] : (\texttt{State} \rightarrow \texttt{State})$$

A State (`s`) symbolizes a machine state where all the machine components are represented as variable. We define the State to be the (partial) function which contains the full Memory, the Instruction Register (`I`), Accumulator (`A`), Accumulator Extension Register (`Q`), all Index Registers (`XR1, XR2,XR3`) and the Overflow and Carry bits in its domain, and the range is either a 16-bit value (most cases) or a one-bit value (for Overflow and Carry).

We need to use some notations to represent the operational semantics of the instructions of IBM-1800 assembly language:

`Inst(I)`	Contents of core storage at the location specified by I (Instruction Register). Later we use `i` as its short notation.
`DB`	D (5th) bit of the instruction.
`FB`	Format bit of the instruction.
`displ`	Displacement associated to the instruction.
`addr`	Address defined in the instruction.
`tag`	Tag value associated with the instruction.
`cond`	Condition defined in the instruction.
`brtype`	Branching instruction type such as BSC Short (`BSCS`), BSC Long (`BSCL`) or BSI (`BSI`).
$O_{6-8}(\mathtt{i},\mathtt{s})$	Checks bits 6–8 of the opcode, then according to bits 7&8, returns the contents of `I, XR1, XR2, XR3` if bit 6 is 0, otherwise returns value of `0,XR1,XR2,XR3`.
X	$O_{6-8}(\mathtt{i},\mathtt{s})$
$\mathtt{loc}(X)$	If `i` is indirect then $\hat{}(X + \mathtt{addr})$ else $X + \mathtt{addr}$.
$locBS(\mathtt{i})$	If `i` is indirect then `^addr` else `addr`.
O_{8-9}	returns different shift instruction operations according to bits 8-9 of the instruction.
$\mathtt{defCond}(cd, bt)$	With the two arguments, it decides the value by which the instruction register (I) will be incremented in a branching instruction. *cd* defines the types of `cond` in the instruction and *bt* indicates the branching instruction type (`brtype`).
$\mathtt{cmdx}(mn, mp)$	Compares two values of one state component (specially index registers) before (*mp*) and after (*mn*) modification and returns 1 if the modified word changes sign or reaches zero while being modified and 0 otherwise. Used mainly in MDX instructions.

where DB, FB, and displ are implicitly functions of i and `0` denotes an abstract location with constant value 0. All of these notations have state `s` and `Inst(I)` or `i` as implicit arguements unless explicitly defined.

$\hat{}y$	Content of state component y.
$\delta_y(f)(x)$	Short for $y \leftarrow f(\hat{}y, x)$.
$\sigma_z(f)(x,y)$	Short for $z \leftarrow f(x,y)$.
$S(x,y)$	Short for $x \leftarrow y$.
$\texttt{comp}(x,y)$	Compares the contents of x with the contents of y and returns instruction register modification value.
$p:q$	A 32 bit value with p representing higher 16 bits and q as lower 16 bits.

where f ranges over a few built-in operations (arithmetic and logical), y can be any of the components of the domain of `State`.

Almost all the instructions are implemented in the abstract model except LOAD/STORE STATUS (LDS/STS), WAIT, EXECUTE I/O (XIO) as they are not used in the IBM-1800 assembler code of OPG.

A.1.1 LOAD/DOUBLE LOAD(LD/LDD)

If $\texttt{OpCode(i)} \in \{\texttt{LD}, \texttt{LDD}\}$,

$$
\begin{aligned}
[\![Inst]\!]s = &\delta_I(+)(1+\texttt{FB})\odot \\
&(S(A, (\texttt{FB}=0\ ?\ \hat{}(X+\texttt{displ})\ :\ \hat{}\texttt{loc}(X))) \\
&\odot (\texttt{DB}=1\ ?\ (S(Q, (\texttt{FB}=0\ ? \\
&\hat{}(X+\texttt{displ}+1)\ :\ \hat{}(\texttt{loc}(X)+1))))\ :\ \mathbb{I})
\end{aligned}
$$

A.1.2 STORE/DOUBLE STORE(STO/STD)

If $\texttt{OpCode(i)} \in \{\texttt{STO}, \texttt{STD}\}$,

$$
\begin{aligned}
[\![Inst]\!]s = &\delta_I(+)(1+\texttt{FB})\odot \\
&S((\texttt{FB}=0\ ?\ \hat{}(X+\texttt{displ})\ :\ \hat{}\texttt{loc}(X)), A) \\
&\odot (\texttt{DB}=1\ ?\ (S((\texttt{FB}=0\ ? \\
&\hat{}(X+\texttt{displ}+1)\ :\ \hat{}(\texttt{loc}(X)+1)), Q))\ :\ \mathbb{I})
\end{aligned}
$$

A.1.3 LOAD INDEX/STORE INDEX (LDX/STX)

If $\texttt{OpCode(i)} \in \{\texttt{LDX}, \texttt{STX}\}$,

$$\begin{aligned}[\![Inst]\!]s = &\delta_I(+)(1+\texttt{FB})\odot \\ &(\texttt{DB}=1\ ?\ (S((\texttt{FB}=0\ ? \\ &\hat{}(I+\texttt{displ})\ :\ \hat{}(\texttt{locBS(i)})), X)) \\ &:\ (S(X, (\texttt{FB}=0\ ?\ \texttt{displ}\ :\ \texttt{locBS(i)})))))\end{aligned}$$

A.1.4 ADD/DOUBLE ADD(A/AD)

If $\texttt{OpCode(i)} \in \{\texttt{A}, \texttt{AD}\}$,

$$\begin{aligned}[\![Inst]\!]s = &\delta_I(+)(1+\texttt{FB})\odot \\ &(\texttt{DB}=1\ ?\ (\delta_{AQ}(+)(\texttt{FB}=0\ ? \\ &(\hat{}(X+\texttt{displ})\ :\hat{}(X+\texttt{displ}+1)) \\ &:\ (\hat{}\texttt{loc}(X) : \hat{}(\texttt{loc}(X)+1)))) \\ &:\ (\delta_A(+)(\texttt{FB}=0\ ?\ \hat{}(X+\texttt{displ})\ :\ \hat{}\texttt{loc}(X))))\end{aligned}$$

A.1.5 SUBTRACT/DOUBLE SUBTRACT(S/SD)

If $\texttt{OpCode(i)} \in \{\texttt{S}, \texttt{SD}\}$,

$$\begin{aligned}[\![Inst]\!]s = &\delta_I(+)(1+\texttt{FB})\odot \\ &(\texttt{DB}=1\ ?\ (\delta_{AQ}(-)(\texttt{FB}=0\ ? \\ &(\hat{}(X+\texttt{displ})\ :\hat{}(X+\texttt{displ}+1)) \\ &:\ (\hat{}\texttt{loc}(X) : \hat{}(\texttt{loc}(X)+1)))) \\ &:\ (\delta_A(-)(\texttt{FB}=0\ ?\ \hat{}(X+\texttt{displ})\ :\ \hat{}\texttt{loc}(X))))\end{aligned}$$

A.1.6 MULTIPLY/DIVIDE(M/D)

If $\texttt{OpCode(i)} \in \{\texttt{M}, \texttt{D}\}$,

$$
\begin{aligned}
[\![Inst]\!]s = & \delta_I(+)(1 + \mathtt{FB})\odot \\
& (\mathtt{DB} = 0\ ?\ (\sigma_{AQ}(*)(A, (\mathtt{FB} = 0\ ? \\
& (\hat{}(X + \mathtt{displ})\ :\ (\hat{}\mathtt{loc}(X)))))) \\
& :\ (\sigma_A(\mathtt{DIV})(AQ, (\mathtt{FB} = 0\ ? \\
& (\hat{}(X + \mathtt{displ})\ :\ (\hat{}\mathtt{loc}(X))))), \\
& \sigma_Q(\mathtt{MOD})(AQ, (\mathtt{FB} = 0\ ? \\
& (\hat{}(X + \mathtt{displ}) :\ (\hat{}\mathtt{loc}(X)))))))
\end{aligned}
$$

[here DIV stands for quotient operation and MOD stands for remainder operation.]

A.1.7 LOGICAL AND/OR(AND/OR)

If $\mathtt{OpCode(i)} \in \{\mathtt{AND}, \mathtt{OR}\}$,

$$
\begin{aligned}
[\![Inst]\!]s = & \delta_I(+)(1 + \mathtt{FB})\odot \\
& (\mathtt{DB} = 1\ ?\ (\delta_A(\mathtt{OR})(\mathtt{FB} = 0\ ? \\
& \hat{}(X + \mathtt{displ})\ :\ \hat{}\mathtt{loc}(X))) \\
& :\ (\delta_A(\mathtt{AND})(\mathtt{FB} = 0\ ? \\
& \hat{}(X + \mathtt{displ})\ :\ \hat{}\mathtt{loc}(X))))
\end{aligned}
$$

A.1.8 LOGICAL XOR (XOR)

If $\mathtt{OpCode(i)} \in \{\mathtt{XOR}\}$,

$$
\begin{aligned}
[\![Inst]\!]s = & \delta_I(+)(1 + \mathtt{FB})\odot \\
& (\mathtt{DB} = 0\ ?\ (\delta_A(\mathtt{XOR})(\mathtt{FB} = 0\ ? \\
& \hat{}(X + \mathtt{displ})\ :\ \hat{}\mathtt{loc}(X)))\ :\ \mathbb{I})
\end{aligned}
$$

A.1.9 SHIFT (SLA/SLT/SRA/SRT)

Shift instructions can be divided into two major classes: Shift Left and Shift Right. These two major classes are defined by the operation code of the instruction. Each of these major classes are divided into subclasses. We use O_{8-9} function to find the different subclasses of the shift instructions. Although each class can be divided into four classes, some of them are not used in the assembler code we examine. That is why, in this model we only implement those shift operations which are practically used in OPG code. Shift Left Logical A (SLA), Shift Left Logical A & Q (SLT) [from Shift Left operations] , Shift Right Logical A (SRA), Shift Right A & Q (SRT) [from Shift Right Operations] are implemented in the following model. Shift Left and Count A (SLCA), Shift Left and Count A & Q (SLC) and Rotate Right A & Q (RTE) are not implemented. Detection of correct shift instructions type by O_{8-9} is a part of implementation details and will not be discussed here. For simplicity, here we can only assume that O_{8-9} only returns $0, 1$ for A and AQ operations respectively.

If $\texttt{OpCode(i)} \in \{\texttt{SLA}, \texttt{SLT}, \texttt{SRA}, \texttt{SRT}\}$,

$$
\begin{aligned}
[\![Inst]\!]s = & \delta_I(+)(1)\odot \\
& (\texttt{DB} = 0\ ?\ (O_{8-9} = 0\ ?\ (\delta_{AQ}(<<)(\texttt{displ})) \\
& :\ (\delta_A(<<)(\texttt{displ}))) \\
& :\ (O_{8-9} = 0\ ?\ (\delta_{AQ}(>>)(\texttt{displ}))\ :\ (\delta_A(>>)(\texttt{displ}))))
\end{aligned}
$$

A.1.10 BRANCH AND SKIP/ BRANCH AND STORE(BSC/BSI)

If $\texttt{OpCode(i)} \in \{\texttt{BSC}, \texttt{BSI}\}$,

$$
\begin{aligned}
[\![Inst]\!]s = & (\texttt{DB} = 1\ ?\ (\delta_I(+) \\
& (\texttt{defcond}(\texttt{cond}, (\texttt{FB} = 0\ ?\ \texttt{BSCS}\ :\ \texttt{BSCL})))) \\
& :\ (\texttt{FB} = 0\ ?\ (S(\hat{}((X + \texttt{displ}), I)), S(I, (1 + X + \texttt{displ}))) \\
& :\ (\delta_I(+)(\texttt{defcond}(\texttt{cond}, \texttt{BSI})))))
\end{aligned}
$$

A.1.11 MODIFY INDEX AND SKIP (MDX)

If $\texttt{OpCode(i)} \in \{\texttt{MDX}\}$,

$$
\begin{aligned}
[\![Inst]\!]s = &(\texttt{FB} = 1\ ? \\
&(\texttt{tag} = 00\ ?\ (\delta_I(+)(2 + \texttt{cmdx}((\hat{}\texttt{addr} + \texttt{displ}), \\
&\hat{}\texttt{addr})), S(\hat{}\texttt{addr}, (\hat{}\texttt{addr} + \texttt{displ}))) \\
&:\ (\delta_I(+)(2 + \texttt{cmdx}((X + \texttt{locBS(i)}), X)), S(X, (X + \texttt{locBS(i)}))))) \\
&:\ (\texttt{tag} = 00\ ?\ (\delta_I(+)(\texttt{displ})) \\
&:\ (\delta_I(+)(1 + \texttt{cmdx}((X + \texttt{displ}), X)), S(X, (X + \texttt{displ})))))
\end{aligned}
$$

A.1.12 COMPARE (CMP)/ DOUBLE COMPARE (DCM)

If $\texttt{OpCode(i)} \in \{\texttt{CMP}, \texttt{DCM}\}$,

$$
\begin{aligned}
[\![Inst]\!]s = &\delta_I(+)\ (\texttt{DB} = 0\ ? \\
&(\texttt{comp}(A, (\texttt{FB} = 0\ ?\ \hat{}(X + \texttt{displ})\ :\ \hat{}\texttt{loc}(X)))) \\
&:\ (\texttt{comp}(AQ, (\texttt{FB} = 0\ ?\ (\hat{}(X + \texttt{displ})\ :\ \hat{}(X + \texttt{displ} + 1)) \\
&:\ (\hat{}\texttt{loc}(X)\ :\ \hat{}(\texttt{loc}(X) + 1))))))
\end{aligned}
$$

Appendix B

Common Codes

This appendix includes all the common codes needed for different module implementation.

B.1 IBM-1800

Here we define some of the auxiliary functions needed to carry out the semantic operations of the IBM 1800 instructions.

```
module IBM1800
(  BrTag(..), SftTag(..), State, o6to8
 , regST,loadX, dXR, o8to9, sCount, addDisp
 , loc, locBS, getMemRefA, getMemRefQ
 , getContentOfMemRefA, getContentOfMemRefQ
 , dA, dQ, dAQ, dIR, delIR, retDispAdd
 , mergeAQ, returnA, returnQ, bool2Word
 , retCondLong, retCondShort, changeIRInBr
 , compA, compAQ, repeatShift16, repeatShift32
) where

import Mem
import OpCode
import Instruction
```

```
import Data.Word
import Data.Int
import Data.Bits
```

We start by defining some of the Tags required to represent the Branching, Compare and Shift conditions.

```
data BrTag = Al | Ne          -- Always, Never
           | Pl | Npl         -- Plus, Not Plus
           | Mn | Nmn         -- Minus, Not Minus
           | Zr | Nzr         -- Zero, Not Zero
           | Ev | Evp | Evm   -- Even, Even or Plus, Even or Minus
           | Od | Odm | Odp   -- Odd, Odd and Minus, Odd and Plus
             deriving Show

data SftTag = Sa | Saq | Sca | Scaq deriving Show -- Shift Tag
```

IBM-1800 current state includes the state of the memory (mem), the Instruction Register (ir), the Accumulator Register (acc), the Accumulator Extension Register (q), the Index Registers (xr1-3), and the Overflow and Carry Flags (overflow and carry).

```
type State = GenState Mem Word16 Word16 Bit

data GenState mem addr val bit = State
  { mem      :: mem
  , ir       :: addr
  , acc      :: val
  , q        :: val
  , xr1      :: addr
  , xr2      :: addr
  , xr3      :: addr
  , overflow :: bit
  , carry    :: bit
  }
 deriving Show
```

The following function checks bits 6-8 of the instruction and if bit 6 (Format Bit) is 0 then returns the value of ir, xr1, xr2, xr3 or if bit 6 is 1 then returns the value of 0, xr1, xr2, xr3 according to tag bits. Here bit 6 is tested previously and according to that bit tag is set to either *I* or *XR0* (which is always Zero).

```
reg1 :: Instruction → (State → Word16)
reg1 inst = case tag inst of
              I   → ir
              XR0 → const 0x0
              XR1 → xr1
              XR2 → xr2
              XR3 → xr3

o6to8 :: Instruction → State → Word16
o6to8 = reg1
```

A special function for the LOAD and STORE index instruction.

```
regST :: Instruction → (State → Word16)
regST inst = case tag inst of
               XR1 → xr1
               XR2 → xr2
               XR3 → xr3
               _   → ir
```

If necessary, the following should be generalized to accept functions with monadic result.

```
tagRegUpdate :: Tag → (addr → addr)
                  → GenState mem addr val bit
                  → GenState mem addr val bit
tagRegUpdate I   f st = st {ir  = f $ ir  st}
tagRegUpdate XR0 f st = st
tagRegUpdate XR1 f st = st {xr1 = f $ xr1 st}
tagRegUpdate XR2 f st = st {xr2 = f $ xr2 st}
tagRegUpdate XR3 f st = st {xr3 = f $ xr3 st}
```

```
loadX :: Tag → addr → GenState mem addr val bit
                    → GenState mem addr val bit
loadX t cx = tagRegUpdate t (const cx)


dXR :: (Num addr) => Tag → addr
       → GenState mem addr val bit
       → GenState mem addr val bit
dXR t cx = tagRegUpdate t (cx +)
```

o8to9 is used to return the Shift Tags to determine the kind of shift operation to be executed depending on the 8th and 9th bit of the instruction.

```
o8to9 :: Instruction → SftTag
o8to9 inst = case fromIntegral $ shiftR ((fromIntegral
                                 $ disp inst::Word8) .&. 0xC0) 6 of
             0x0 → Sa
             0x1 → Sca
             0x2 → Saq
             0x3 → Scaq
```

We use sCount to return the no. of shift count as in Fig.3-13 of the IBM-1800 manual [IBM70]. The shift count should be the lower order 6 bits of either displacement or XR1/XR2/XR3 depending on the tag bits.

```
sCount :: (Num b) => Instruction → GenState mem Word16 val bit → b
sCount inst s = fromIntegral $ x .&. 0x003F
      where x = case tag inst of
                I   → fromIntegral $ (disp inst)::Word16
                XR1 →  (xr1 s)
                XR2 →  (xr2 s)
                XR3 →  (xr3 s)
```

The following functions are used to calculate the effective addresses of memory.

`addDisp` adds the displacement of the instruction to the defined index register value to return the effective address. `loc` returns the effective address of memory for long instructions. The effective address is calculated either by adding the address with the corresponding index register value or the content of the address added with the index register value of the instruction depending on the indirect bit of the instruction.

`locBS` function returns the effective address for the LOAD and STORE INDEX instructions.

`getMemRef` and `getContentOfMemRef` are used to get the memory reference and the content of the memory reference respectively. Both of them have two versions for the current effective address (`getMemRefA`, `getMemRefQ`, `getContentOfMemRefA`, `getContentOfMemRefQ`).

```
addDisp :: Instruction → State → Word16
addDisp inst s = fromIntegral $ (x +
                 (fromIntegral $ disp inst :: Int16))::Word16
        where x = fromIntegral $ o6to8 inst s::Int16

loc :: Instruction → State → Word16
loc inst s = if indAdd inst ≡ One then getMem (mem s) x' else x'
  where
    x = fromIntegral $ (o6to8 inst s)
    x' = fromIntegral $ x + address inst

locBS :: Instruction → Bool → State → Word16
locBS inst ls s =
     if isLong inst
        then if indAdd inst ≡ One
               then getMem (mem s) (address inst)
               else address inst
        else if ls
               then (ir s) + (fromIntegral $ disp inst::Word16)
               else fromIntegral $ disp inst::Word16

getMemContent :: State → Word16 → Word16
```

```
getMemContent s effadd = getMem (mem s) effadd

getContentOfMemRef:: Int → Instruction → State → Word16
getContentOfMemRef ofs inst s =
    getMemContent s $ if isLong inst
                      then fromIntegral $ (loc inst s)+ofs
                      else fromIntegral $ (addDisp inst s)+ofs

getMemRef:: Int → Instruction → State → Word16
getMemRef ofs inst s = if isLong inst
                       then fromIntegral $ (loc inst s) + ofs
                       else fromIntegral $ (addDisp inst s) + ofs

getMemRefA, getMemRefQ,
getContentOfMemRefA,
getContentOfMemRefQ :: Instruction → State → Word16
getMemRefA = getMemRef 0
getMemRefQ = getMemRef 1
getContentOfMemRefA = getContentOfMemRef 0
getContentOfMemRefQ = getContentOfMemRef 1
```

Now we have to define functions to update the state components: Accumulator, Q Register, Index Registers, Accumulator:Q Registers etc.

```
dA :: Word16 → State → State
dA a s = s {acc = a}

dQ :: Word16 → State → State
dQ q s = s {q = q}

dAQ :: (Word16,Word16) → State → State
dAQ (a,q) s = s {acc = a, q = q}

dIR :: Word16 → State → State
```

```
dIR a s = s {ir = irNew}
   where irNew = (ir s) + a

delIR :: Word16 → State → State
delIR a s = s {ir = a}
```

This function returns True(1) if the modified factor changes sign or reaches zero while being modified and False(0) otherwise.

```
retDispAdd :: Int16 → Int16 → Word16
retDispAdd cN cP = if (cN ≡ 0) ∨ ((cP > 0) ∧ (cN < 0))
                      ∨ ((cP < 0) ∧ (cN > 0)) then 1 else 0
```

These functions are used for 32 bit operations.

```
mergeAQ :: State → Int32 -- merges the A and Q register.
mergeAQ s = fromIntegral $  (shiftL (fromIntegral $ q s :: Int32) 16)
                         .|. (fromIntegral $ acc s ::Int32)

mergeMem :: State → Word16 → Int32 -- merges the contents of the memory
                                   -- addresses.
mergeMem s add = fromIntegral $  (shiftL (fromIntegral $
                 getMem (mem s) (add+1) :: Int32) 16)
                 .|. (fromIntegral $ getMem (mem s) add ::Int32)

returnA :: Int32 → Word16 -- gets the Accumulator register value
                          -- from the AQ register.
returnA aq = fromIntegral $ aq .&. 0x0000FFFF

returnQ :: Int32 → Word16 -- gets the Accumulator Extension Register
                          -- value from the AQ register.
returnQ aq = fromIntegral $ shiftR (aq .&. 0xFFFF0000) 16
```

`bool2Word` converts the boolean value to `word16` value.

```
bool2Word:: bool → Word16
bool2Word True = 1
bool2Word False = 0
```

These functions are used to return the Condition Tags of the instructions.

```
retCondLong :: Instruction → BrTag
retCondLong inst = case cond inst  of
                     0x01 → Od
                     0x02 → Npl
                     0x03 → Odm
                     0x04 → Nmn
                     0x05 → Odp
                     0x06 → Zr
                     0x08 → Nzr
                     0x0A → Mn
                     0x0C → Pl
                     0x0E → Ne
                     otherwise → Al

retCondShort :: Instruction → BrTag
retCondShort inst = case fromIntegral
                        $ shiftR ((disp inst) .&. 0x3C) 2 of
                           0x01 → Ev
                           0x02 → Pl
                           0x03 → Evp
                           0x04 → Mn
                           0x05 → Evm
                           0x06 → Nzr
                           0x08 → Zr
                           0x0A → Nmn
                           0x0C → Npl
                           0x0E → Al
                           otherwise → Ne
```

changeIRInBr changes the instruction register value depending on the branch conditions.

```
changeIRInBr:: BrTag → State → State
changeIRInBr bt s =
      case bt of
          Al  → if dbit inst ≡ Zero
                    then delIR (addL+1)
                         s {mem = (writeMem (mem s) locL $ 2+ir s)}
                    else if isLong inst
                           then delIR addL s
                           else dIR 2 s
          Pl  → deltaIR s inst (> 0)
          Npl → deltaIR s inst (≤ 0)
          Mn  → deltaIR s inst (< 0)
          Nmn → deltaIR s inst (≥ 0)
          Zr  → deltaIR s inst (≡ 0)
          Nzr → deltaIR s inst (≠ 0)
          Ev  → deltaIR1 s inst $ ((acc s) .&. 0x0001) ≡ 0
          Evp → deltaIR1 s inst $ ((acc s) .&. 0x0001) ≡ 0
                       ∨((acc s) .&. 0x8000) ≡ 0
          Evm → deltaIR1 s inst $ ((acc s) .&. 0x0001) ≡ 0
                       ∨((acc s) .&. 0x8000) ≠ 0
          Od  → deltaIR1 s inst $ ((acc s) .&. 0x0001) ≠ 0
          Odm → deltaIR1 s inst $ ((acc s) .&. 0x0001) ≠ 0
                       ∧ ((acc s) .&. 0x8000) ≠ 0
          Odp → deltaIR1 s inst $ ((acc s) .&. 0x0001) ≠ 0
                       ∧ ((acc s) .&. 0x8000) ≡ 0 ∧ (acc s) ≠ 0
          Ne  → if isLong inst
                     then dIR 2 s
                     else dIR 1 s
where addL = getMemRefA inst s

deltaIR s inst cond = if dbit inst ≡ Zero
```

```
                    then deltaIRBsi s cond
                    else if isLong inst
                           then deltaIRLong s cond
                           else deltaIRShort s cond
  where deltaIRShort s cond = if (cond $ acc s)
                                 then dIR 2 s
                                 else dIR 1 s
        deltaIRLong  s cond = if (cond $ acc s)
                                  then delIR locL s
                                  else dIR 2 s
        deltaIRBsi s cond
              = if (cond $ acc s)
                  then delIR (locL+1)
                    s {mem = (writeMem (mem s) locL $ 2+ir s)}
                  else dIR 2 s

deltaIR1 s inst cond = if dbit inst ≡ Zero
                       then deltaIRBsi1 s cond
                       else if isLong inst
                              then deltaIRLong1 s cond
                              else deltaIRShort1 s cond
 where deltaIRShort1 s cond = if cond
                                  then dIR 2 s
                                  else dIR 1 s
       deltaIRLong1  s cond = if cond
                                  then delIR locL s
                                  else dIR 2 s
       deltaIRBsi1 s cond
             = if cond
                 then delIR (locL+1)
                   s {mem = (writeMem (mem s) locL $ 2+ir s)}
                 else dIR 2 s
```

compA and compAQ are used to compare two values in the Compare and Double Compare instructions.

```
compA:: Int16 → Int16 → Word16
compA a b = case compare a b of
                EQ → 3
                LT → 2
                GT → 1

compAQ:: Int32 → Int32 → Word16
compAQ a b = case compare a b of
                EQ → 3
                LT → 2
                GT → 1
```

Here are some auxiliary functions to emulate Shift Instructions.

```
repeatShift16 a x f = repeatShift16' (f a x) (x-1) f

repeatShift32 a x f = repeatShift32' (f a x) (x-1) f

repeatShift16' a x f
       | x = (a,x)
       | ((a .&. 0x8000) ≡ 0) = repeatShift16' (f a 1) (x-1) f
       | otherwise = (a,x)

repeatShift32' a x f
       | x = (a,x)
       | ((a .&. 0x80000000) ≡ 0) = repeatShift32' (f a 1) (x-1) f
       | otherwise = (a,x)
```

B.2 Stack

This module just defines a Stack class.

```
module Stack where
```

The following defines a Stack class, with stack of type `a → b`, and keys of type `a`, values of type `b`.

```
class Stack a b where
    createStack :: a → b
    addEntry :: (a → b) → (a,b) → (a → b)
    addEntries :: (a→b) → [(a,b)] → (a → b)
    lookupEntry :: (a → b) → a → b
```

Appendix C

Emulator

This appendix presents the implementation of the emulator. This is a very direct translation of abstract model of the instructions in Appendix A.

C.1 Lst2String

Tool to convert a `.lst` file to a String – mostly an intermediate program to decouple *Lst2Gxl*.

```
module Main where
import IBM1800
import Instruction
import Lst
import System (getArgs)
import Numeric (showHex)
import Mem
import MemInit
import GHC.Show
import Data.Array
import Emulate
import Bits

main :: IO ()
```

```
main = do [infile] ∈ getArgs
          myLst    ∈ readFile infile
          putStrLn ((showState · emulate 8 · initState) myLst)
          putStr "Done."
```

These are the initialization of the state and the State representation.

```
initState:: String → State
initState l = State {mem = fillMem initMem $ parseLst l,
                     ir  = 0x35B6,
                     acc = 0x07,
                     q   = 0,
                     xr1 = 0x3808,
                     xr2 = 0x3808,
                     xr3 = 0,
                     overflow = Zero,
                     carry = Zero
                     }
showState :: State → String
showState s  =
     "IR = 0x" ++ sh (ir s) ++ "\n"
  ++ "A = 0x" ++ sh (acc s) ++ "\n"
  ++ "Q = 0x" ++ sh (q s) ++ "\n"
  ++ "XR1 = 0x" ++ sh (xr1 s) ++ "\n"
  ++ "XR2 = 0x" ++ sh (xr2 s) ++ "\n"
  ++ "XR3 = 0x" ++ sh (xr3 s) ++ "\n"
  ++ "memory content @IR = 0x" ++ sh ((mem s)!(ir s)) ++ "\n"
  ++ "memory content @(Address 0x3815) = 0x"
                            ++ sh ((mem s)!(0x3815)) ++ "\n"
  ++ "memory content @(Address 0x3816) = 0x"
                            ++ sh ((mem s)!(0x3816))
     where sh x = showHex x ""
```

C.2 Emulate

This is a bare bones emulator.

```
module Emulate (emulate)
where

import OpCode
import Instruction
import IBM1800
import Mem
import Data.Word
import Data.Int
import Data.Bits
```

Our emulator takes a number of steps, a State and returns a State.

```
emulate :: Int → State → State
emulate 0 s = s
emulate n s = emulate (n-1) (step s)

step :: State → State
step s = semantics_ op inst s
    where op = getOp (getMem (mem s) (ir s))
          inst = wordsToInstruction (getMem (mem s)
          (ir s)) (getMem (mem s) ((ir s)+1))
```

The followings are the implementation of the semantic definitions of all the instructions.

```
semantics_ :: Op → Instruction → State → State
semantics_ LD inst s =          -- LOAD/DOUBLE LOAD
 dIR (1+fb) $ dA (getContentOfMemRefA inst s) s
            $ if dbit inst ≡ Zero
                 then s
                 else dQ (getContentOfMemRefQ inst s) s
    where fb = bool2Word $ isLong inst
```

```
semantics_ ST inst s =          -- STORE/DOUBLE STORE
 dIR (1+fb) $ s{ mem = (writeMem (mem s)
                        (getMemRefA inst s) $ acc s)}
            $ if dbit inst ≡ Zero
                then s
                else s {mem = (writeMem (mem s)
                              (getMemRefQ inst s) $ q s)}
    where fb = bool2Word $ isLong inst

semantics_ LSX inst s =
 dIR(1+fb) $ if dbit inst ≡ Zero
               then  loadX (tag inst)
                     (locBS inst False s) s -- LOAD INDEX
               else  s {mem = (writeMem (mem s)
                  (locBS inst True s) cX)} -- STORE INDEX
   where fb = bool2Word $ isLong inst
       cX     =  regST inst s

semantics_ ADD inst s =
 dIR (1+fb) $ if dbit inst ≡ Zero
              then dA (fromIntegral $ addNewL::Word16) s
              else dAQ ((returnA addMemCont),(returnQ addMemCont)) s

 where fb = bool2Word $ isLong inst
       aq = mergeAQ s
       addNewL = (fromIntegral $ acc s::Int16) +
                 (fromIntegral $ getContentOfMemRefA inst s::Int16)
       addMemCont = aq + (mergeMem s $ getMemRefA inst s)

semantics_ SUB inst s =
 dIR (1+fb) $ if dbit inst ≡ Zero
               then dA (fromIntegral $ subNewL::Word16) s
               else dAQ ((returnA subMemCont),(returnQ subMemCont)) s
```

```
 where fb = bool2Word $ isLong inst
       aq = mergeAQ s
       subNewL = (fromIntegral $ acc s::Int16) -
                 (fromIntegral $ getContentOfMemRefA inst s::Int16)
       subMemCont = aq - (mergeMem s $ getMemRefA inst s)

semantics_ MD inst s =
dIR (1+fb) $ if dbit inst ≡ Zero
               then dAQ ((returnA mulNewL),(returnQ mulNewL)) s
                                        -- MULTIPLICATION
               else dAQ (divD,modD) s -- DIVISION
  where fb = bool2Word $ isLong inst
       aq = mergeAQ s
       addL = fromIntegral $ getContentOfMemRefA inst s:: Int32
       mulNewL  =  fromIntegral $ (fromIntegral $ acc s::Int16)
                              * addL::Int32
       divD = fromIntegral $ aq `div` addL::Word16
       modD = fromIntegral $ aq `mod` addL::Word16

semantics_ AR inst s =
dIR (1+fb) $ if dbit inst ≡ Zero
                then dAcc andNewD s -- AND
                else dAcc orNewD s  -- OR
 where fb = bool2Word $ isLong inst
       addAR = getContentOfMemRefA inst s
       andNewD  =  acc s .&. addAR
       orNewD   =  acc s .|. addAR

semantics_ EOR inst s =
dIR (1+fb) $ if dbit inst ≡ Zero
                then dAcc xorNewD s -- XOR
                else s
 where fb = bool2Word $ isLong inst
       addAR = getContentOfMemRefA inst s
       xorNewD  =  acc s `xor` addAR
```

```
semantics_ SFT inst s =
 dIR 1 $ case dbit inst of
       Zero → case o8to9 inst of     -- SHIFT LEFT
                Sa   → s {acc = sltNewA, carry = cfLA}
                Saq  → s {acc = (returnA sltNewAQ),
                          q = (returnQ sltNewAQ),
                          carry = cfLAQ}
                Sca  → s {acc = newA,
                          carry = if newX ≠ 0 then One
                                              else Zero}
                Scaq → s {acc = (returnA newA1),
                          q   = (returnQ newA1),
                          carry = if newX1 ≠ 0 then One
                                               else Zero}
       One  → case o8to9 inst of     -- SHIFT RIGHT
                Sa   → s {acc = srtNewA, carry = cfRA}
                Saq  → s {acc = (returnA srtNewAQ),
                          q = (returnQ srtNewAQ),
                          carry = cfRAQ}
                Sca  → s
                Scaq → s {acc = (returnA newA2),
                          q   = (returnQ newA2),
                          carry = if newX2 ≠ 0 then One
                                               else Zero}
 where x  = sCount inst s
       aq = mergeAQ s
       sltNewA = shiftL (acc s) x
       srtNewA = shiftR (acc s) x
       sltNewAQ  =  shiftL aq x
       srtNewAQ  =  shiftR aq x
       (newA,newX) = repeatShift16 (acc s) x shiftL-- shift left Sca
       (newA1,newX1) = repeatShift32 aq x rotateL-- shift left Scaq
       (newA2,newX2) = repeatShift32 aq x rotateR-- rotate right Scaq
```

```
      cfLA  = if (testBit (acc s) (16-x)) then One else Zero
      cfRA  = if (testBit (acc s) (x-1))  then One else Zero
      cfLAQ = if (testBit aq (16-x)) then One else Zero
      cfRAQ = if (testBit aq (x-1))  then One else Zero

semantics_ BRANCH inst s =
if dbit inst ≡ Zero -- BSI
   then if isLong inst
          then changeIRInBr (retCondLong inst) s
          else delIR (addL+2) $ s {mem = writeMem (mem s)
                                     (displ+1) $ 1+ir s}
   else if isLong inst -- BRANCH/SKIP
          then changeIRInBr (retCondLong inst) s
          else changeIRInBr (retCondShort inst) s
  where addL = getMemRefA inst s

semantics_ MDX inst s =
 if isLong inst  -- MDX
    then if tag inst ≡ XR0
           then dIR (2+conAdd) $  s {mem = (writeMem (mem s)
                                      (address inst) cMemNew)}
           else dIR (2+condAdd) $ dXR (tag inst) cXRNew s
    else if tag inst ≡ I
           then s {ir = (fromIntegral $ (ir s) +
                   (fromIntegral $ addL::Word16) + 1)}
           else dIR (1+condAdd) $ dXR (tag inst) cXRNew s
 where addL = fromIntegral $ locBS inst False s:: Int16
       cMemOld = (fromIntegral $ getMem (mem s)
                                  $ address inst::Int16)
       cMemNew = fromIntegral $ cMemOld + addL::Word16
       conAdd = retDispAdd (cMemOld+addL) cMemOld
              -- This one is specifically for F = 1 Tag = 00 IA = X
       cXROld = fromIntegral $ (reg1 inst s):: Int16
       cXRNew = fromIntegral $ cXROld + addL::Word16
```

```
        condAdd = retDispAdd (cXROld+addL) cXROld
                -- This one is for F = 0/1 Tag /= 00

semantics_ CMP inst s =
 dIR (if dbit inst ≡ Zero
        then compA (fromIntegral $ acc s :: Int16)
                   addCompC -- COMPARE
        else compAQ aq (mergeMem s addComp)) s
                              -- DOUBLE COMPARE
 where addComp = getMemRefA inst s
       addCompC = fromIntegral $ getContentOfMemRefA inst s:: Int16
       aq = mergeAQ s

semantics_ _ _ s = s
```

C.3 Other Modules

The other modules of the emulator, *Mem*.lhs and *MemInit*.lhs (See Section 6.3), are already included in the thesis.

Appendix D

One Step Symbolic Emulator

Here we include the implementation of one step symbolic emulator. This is a direct translation of the abstract model of instructions in Appendix A.

D.1 OneStep

Here we implement semantic definition of all the instructions. All the implementations of instructions use the same model used in the Emulator part.

```
module OneStep (sSemantics_)
where

import OpCode (Op(..), Tag(..), Address)
import Instruction (dbit, tag, Bit(..),
       isLong, address, indAdd, disp, Instruction)
import IBM1800 (SftTag(..), o8to9, retCondLong, retCondShort)
import Symbolic
import Data.Word (Word8,Word16)
import Data.Int (Int16)
import Data.Bits ((.&.))
```

sSemantics_ takes an opcode and an instruction as input and finds the symbolic interpretation of the functions of that instruction as ***Func*** and the condition associated with that instruction as ***CondFunc***.

The model of the operational semantics of instructions is described in Appendix A.

```
sSemantics_ :: Op → Instruction → [(CondFunc,[Func])]

sSemantics_ LD inst =
 [(Tru, (dAssignSC16 Acc memRefA) :
 if  dbit inst ≡ Zero
      then []                              -- LOAD
      else [dAssignSC16 Q memRefQ])] -- DOUBLE LOAD
 where memRefA = dMemRefA inst
       memRefQ = dMemRefQ inst

sSemantics_ ST inst =
 [(Tru, (dAssignMem16 memRefA Acc) :
        if dbit inst ≡ Zero
            then []                              -- STORE
            else [dAssignMem16 memRefQ Q])]  -- DOUBLE STORE
 where memRefA = dMemRefA inst
       memRefQ = dMemRefQ inst

sSemantics_ LSX inst =
 [(Tru,
 if (dbit inst ≡ Zero)
       then [dAssignX (regX inst) (addX inst)]     -- LOAD INDEX
       else [dAssignMemX (regX inst) (stX inst)])] -- STORE INDEX

sSemantics_ ADD inst =
 [(Tru,
 if (dbit inst ≡ Zero)
      then [dUpdateSC16 Acc  Add
             Val16{val161 = Acc,  val162 = memRefA}]-- ADD
      else [dUpdateSC32 AccQ Add
         Val32{val321 = AccQ, val322 = memRefA}])]-- DOUBLE ADD
 where memRefA = dMemRefA inst
```

```
sSemantics_ SUB inst =
 [(Tru,
 if (dbit inst ≡ Zero)
   then [dUpdateSC16 Acc  Sub
      Val16{val161 = Acc,  val162 = memRefA}] -- SUBTRACT
   else [dUpdateSC32 AccQ Sub
    Val32{val321 = AccQ, val322 = memRefA}])]-- DOUBLE SUBTRACT
 where memRefA = dMemRefA inst
```

In Multiplication, although the operands are word16 but for the update operation, we have to use dUpdateSC32 as we have to update ACCQ. So for consistency, we use Val32 instead of Val16.

Same thing happens for the division operation, although the first operand in the division operation is word32 but the value to be updated is word16. So we used dUpdateSC16 and Val16 is used instead of Val32.

```
sSemantics_ MD inst =
 [(Tru,
 if (dbit inst ≡ Zero)
   then [dUpdateSC32 AccQ  Mul
        Val32{val321 = Acc, val322 = memRefA}] -- MULTIPLICATION
      else [(dUpdateSC16 Acc  Div
              Val16{val161 = AccQ,val162 = memRefA}) -- DIVISION
             ,(dUpdateSC16 Q    Mod Val16{val161 = AccQ,
                                          val162 = memRefA})])]
 where memRefA = dMemRefA inst

sSemantics_ AR inst =
 [(Tru,
 if (dbit inst ≡ Zero)
      then [dUpdateSC16 Acc  And
             Val16{val161 = Acc,val162 = memRefA}] -- AND
      else [dUpdateSC16 Acc  Or
             Val16{val161 = Acc,val162 = memRefA}])]-- OR
 where memRefA = dMemRefA inst
```

```
sSemantics_ EOR inst =
 [(Tru,
 if (dbit inst ≡ Zero)
      then [dUpdateSC16 Acc  Xor
             Val16{val161 = Acc,val162 = memRefA}]-- XOR
        else [])] -- TODO
 where memRefA = dMemRefA inst

sSemantics_ SFT inst =
 [(Tru,
 if (dbit inst ≡ Zero)
       then case o8to9 inst of -- SHIFT LEFT
             Sa   → [dUpdateAS Shl x]
             Saq  → [dUpdateAQS Shl x]
             Sca  → []
             Scaq → []
       else case o8to9 inst of -- SHIFT RIGHT
             Sa   → [dUpdateAS Shr x]
             Saq  → [dUpdateAQS Shr x]
             Sca  → []
             Scaq → [])]
 where x  = fromIntegral $ (fromIntegral
                        $ (disp inst)::Word8) .&. 0x3F

sSemantics_ BRANCH inst =
  if (dbit inst ≡ One)
      then deltaIRS $ retCondOpSm
           $ if isLong inst   -- SKIP/BSC/BOSC
                then  retCondLong inst
                else  retCondShort inst
      else if isLong inst    -- BSI
             then deltaIRBsi $ retCondOpSm $ retCondLong inst
             else [(Tru,(dAssignMemX I memRefA):[])]
 where  memRefA = dMemRefA inst
```

```
        deltaIRS condt =
           if condt `elem` [Phntl, Phnte]
              then [(Tru,[])]
              else ((dUpdateCond Acc condt True),[]):
                  [((dUpdateCond Acc condt False),[])]
        deltaIRBsi condt =
           if condt ≡ Phntl
              then [(Tru,(dAssignMemX I memRefA) :[])]
              else if condt ≡ Phnte
                     then [(Tru,[])]
                     else ((dUpdateCond Acc condt True),
                           (dAssignMemX I memRefA):[]):
                          [((dUpdateCond Acc condt False),[])]

sSemantics_ MDX inst =
 if isLong inst                                        -- MDX
      then if tag inst ≡ XRO
              then ((dCondDispAddM (CConst {valCC = address inst})
                   (Const{valC = fromIntegral $ disp inst}) True)
                   ,[dCondDisp (CConst {valCC = address inst})
                   (Const{valC = fromIntegral $ disp inst})])
                   :[((dCondDispAddM (CConst {valCC = address inst})
                   (Const{valC = fromIntegral $ disp inst}) False)
                   ,[dCondDisp (CConst {valCC = address inst })
                   (Const{valC = fromIntegral $ disp inst})])]
              else ((dCondDispAddT (tag inst) dXMem True)
                   ,[dUpdateX (tag inst) dXMem])
                   :[((dCondDispAddT (tag inst) dXMem False)
                   ,[dUpdateX (tag inst) dXMem])]
      else if tag inst ≡ I
              then [(Tru,[])]
              else ((dCondDispAddT (tag inst) dXMem True)
                   ,[dUpdateX (tag inst)
                    (Const{valC = fromIntegral $ disp inst::Int16})])
```

```
                    :[((dCondDispAddT (tag inst) dXMem False)
                    ,[dUpdateX (tag inst)
                    (Const{valC = fromIntegral $ disp inst::Int16})])]
    where dXMem = mdxMemRef inst

sSemantics_ CMP inst =
  if (dbit inst ≡ Zero)                    -- COMPARE
   then (UpdateComp Acc Eq0 memRefA, [])
      : (UpdateComp Acc Lt0 memRefA, [])
      : [(UpdateComp Acc Gr0 memRefA, [])]
   else (UpdateComp AccQ Eq0 memRefA, [])
      : (UpdateComp AccQ Lt0 memRefA, [])
      : [(UpdateComp AccQ Gr0 memRefA, [])] -- DOUBLE COMPARE
  where memRefA = dMemRefA inst
```

These are different helping functions used to assign values (symbolic) to different data types declared for the functions of the IBM-1800 instructions.

```
dAssignSC16 :: StateComp → MemRef → Func
dAssignSC16 sc v = AssignSC16 {scA16 = sc,
                               valAS16 = v}

dUpdateSC16 :: StateComp → Operator → Val → Func
dUpdateSC16 sc op v = UpdateSC16 {scU16 = sc,
                      op16 = op, valUS16 = v }

dAssignMem16 :: MemRef → StateComp → Func
dAssignMem16 v sc = AssignMem16 {locA16 = v, valA16 = sc}

dUpdateSC32 :: StateComp → Operator → Val → Func
dUpdateSC32 sc op v = UpdateSC32 {scU32 = sc,
                      op32 = op, valUS32 = v }

dAssignX :: Tag → MemRef → Func
```

```
dAssignX t v = AssignX {conX = t, valX = v}

dUpdateX :: Tag → MemRef → Func
dUpdateX t v = UpdateX {conUX = t, valUX = v}

dAssignMemX :: Tag → MemRef → Func
dAssignMemX t v = AssignMemX {valAX = t, locAX = v}

dUpdateAS :: Operator → Word8 → Func
dUpdateAS op t = UpdateAS {opS16 = op, valS16 = t}

dUpdateAQS :: Operator → Word8 → Func
dUpdateAQS op t = UpdateAQS {opS32 = op, valS32 = t}

dUpdateCond :: StateComp → CondOpSm → Bool → CondFunc
dUpdateCond s op st = Condition {scC = s,
                                 opC = op, stat = st}
```

These are used to assign values to different *CondFunc* structures.

```
dCondDispAddT :: Tag → MemRef → Bool → CondFunc
dCondDispAddT t v b = CondDispAddT {scCT = t,
                                    valCT = v, sgT = b}

dCondDispAddM :: MemRef → MemRef → Bool → CondFunc
dCondDispAddM c v b = CondDispAddM {locCM = c,
                                    valCM = v, sgM = b}

dCondDisp :: MemRef → MemRef → Func
dCondDisp c v = CondDisp {valD1 = c, valD2 = v}
```

Small helper functions to assign values to different types of memory references depending on the instruction fields.

```
qa :: Bit → Tag → Instruction → Address → MemRef
```

```
qa One  s i o = Indirect { reg = s, addr = address i + o }
qa Zero s i o = Direct   { reg = s, addr = address i + o }

qbr :: Bit → Tag → Instruction → Word16 → MemRef
qbr One  s i o = BrDirect{reg = s, addr = address i, offBD = o}
qbr Zero s i o = BrConst {reg = s, addrBr = address i +
                                            fromIntegral o}
```

Here we assign values to memory references depending on the instruction.

```
dMemRef :: Int → Instruction → MemRef
dMemRef offset inst = if isLong inst
    then qa (indAdd inst) (tag inst) inst (fromIntegral offset)
    else Dispmnt { reg = tag inst,
                   addrC = disp inst + fromIntegral offset}

brMemRef :: Int → Instruction → MemRef
brMemRef offset inst = if isLong inst
    then qbr (indAdd inst) (tag inst) inst (fromIntegral offset)
    else BrConst{reg = tag inst,
                 addrBr = (fromIntegral $ disp inst::Word16)
                        + fromIntegral offset}

mdxMemRef :: Instruction → MemRef
mdxMemRef inst = if isLong inst
                   then if indAdd inst ≡ One
                      then CConst{valCC = address inst}
                      else Const{valC =
                          fromIntegral $ address inst:: Int16}
                   else Const{valC = fromIntegral
                                   $ disp inst::Int16}

dMemRefA, dMemRefQ, brMemRefA, brMemRefQ :: Instruction → MemRef
dMemRefA  = dMemRef 0
```

```
dMemRefQ  = dMemRef 1
brMemRefA = brMemRef 0
brMemRefQ = brMemRef 1
```

Special functions for the LOAD and STORE index instructions. The purpose of those functions are illustrated in the reference Manual page 2/70 3-10, 3-11 [IBM70].

```
regX :: Instruction → Tag
regX inst = if tag inst  ≡ XR0 then I
                                 else tag inst
```

```
addX :: Instruction → MemRef  -- Reference Manual page 2/70 3-10
addX inst = if isLong inst
               then if indAdd inst ≡ One
                       then CConst {valCC = address inst}
                       else Const {valC = fromIntegral
                                  $ address inst::Int16}
               else Const {valC = (fromIntegral
                                  $ disp inst::Int16)}
```

```
stX :: Instruction → MemRef -- Reference Manual page 2/70 3-11
stX inst = if isLong inst
              then if indAdd inst ≡ One
                      then Indirect {reg = XR0,
                                     addr = address inst}
                      else CConst {valCC = address inst}
              else Dispmnt{reg = I , addrC = disp inst}
```

D.2 Other Modules

The only other module to generate one step symbolic interpretation of the instructions is *Symbolic*.lhs (See Section 7.2) which is already included in the thesis.

Appendix E

Marked-up Control Flow Graph Generator

This appendix includes all the modules to generate the Marked-up Control Flow Graph.

E.1 Gxl2MyGraph

This is the main module to generate the internal data structure of Control Flow Graph with all of its edges annotated.

This module takes a GXL graph and makes an internal data structure representation of that GXL graph. This GXL graph is the control flow graph of an IBM-1800 assembler code segment. In the internal data structure of the graph we maintain only those information needed to generate the symbolic interpretation of the code segment related to the GXL graph.

```
module Main where

import qualified Gxl
import Text.XML.HaXml.Xml2Haskell
import System (getArgs)
import MyGraph
import Data.List
```

```
import Control.Monad.Error

import IO
```

The main function of the program takes (possibly) two arguments: There are one GXL file that is the Control Flow subgraph, and by reading it we make the our internal representation of the subgraph.

```
main = (do
        [infile, outfile] ∈ getArgs
        putStrLn ("Reading from "++infile)
        value ∈ fReadXml infile :: IO Gxl.Gxl
        putStrLn ("Writing to "++outfile)
        let sval = length $ takeWhile (≠ '.') infile
        putStrLn (show sval)
        if (outfile ≡ "-")
           then putStr $ show $  doAnnotation
                       $ gxlToMyGraph value (sval+1)
           else do writeFile outfile $ show
                   $ doAnnotation $ gxlToMyGraph value (sval+1)
                   putStrLn "Done."
      ) `catchError` usage


usage :: IOError → IO ()
usage e = do
          putStrLn "Usage: Gxl2MyGraph [input.gxl] [output]"
```

E.2 Other Modules

The other modules to produce the marked-up CFG, ***MyGraph*.lhs** (See Section 8.2.1), ***OneStep*.lhs** (See Appendix D), are added previously in the thesis.

Appendix F

Data Flow Equations Generator

Here we include the implementation of Data Flow Equations (DFE) generator.

F.1 Graph2Expr

This tool is used to find the symbolic interpretation of a given code segment i.e. for each instruction in the code it produces one or more equivalent symbolic statements which represent the semantics of the program.

```
module Main where

import qualified Gxl
import Text.XML.HaXml.Xml2Haskell
import System (getArgs)
import MyGraph (gxlToMyGraph)
import Exp (Stmt, ConditionStmt)
import FindExpr (findAnntOfGraph)
import Control.Monad.Error
import IO
```

The main function of the program takes three arguments: They are one GXL file that is the Control Flow subgraph (of the code segment), start node, and the output file to be generated. By reading the GXL file, we make the our internal representation

of the subgraph and then find the symbolic interpretation of the code in the output file.

```
main = (do
        [infile, st, outfile] ∈ getArgs
        putStrLn ("Reading from "++infile)
        value ∈ fReadXml infile :: IO Gxl.Gxl
        putStrLn ("Writing to "++outfile)
        let sv = length $ takeWhile (≠ '.') infile
        putStrLn (show sv)
        if (outfile ≡ "-")
           then putStr $ show $ exprTuplePrint
                 $ findAnntOfGraph (gxlToMyGraph value (sv+1)) st
           else do writeFile outfile $ show
                 $ exprTuplePrint $ findAnntOfGraph
                                    (gxlToMyGraph value (sv+1)) st
                   putStrLn "Done."
      ) `catchError` usage

usage :: IOError → IO ()
usage e = do
          putStrLn "Usage: Graph2Expr [input.gxl] start [output]"
```

These functions are used for pretty printing. They just separate different outputs and print them in a nice manner.

```
exprTuplePrint :: [[([ConditionStmt],[Stmt])]] → [String]
exprTuplePrint = map tuplePrint1

tuplePrint1 :: [([ConditionStmt], [Stmt])] → String
tuplePrint1 = (concat·(map tuplePrint0))

tuplePrint0 (pCond,exec) =
      "  PathCondition:  " ++ show pCond ++
      "  Instruction Execution:  " ++ show exec
```

F.2 FindJoin

Although the name suggests only to find "join", this module has some functions to find the "split" and "join" node of the two paths of branching structure. The meaning of "split" and "join" node are defined in ***FindExpr*** module.

```
module FindJoin
( findSplitJoin, getCommonDiv
, findLoopPart
)
where

import MyGraph (MyGraph, MyNode)
import FindPath (PathType(..), FinalPath, Path, nodesFromStart)
import Data.List (partition, ∩, (\\))
import Data.Maybe (fromJust, Maybe)
```

Without loss of generality, we can assume that the nodes of the paths before the "split" node and the nodes after the "join" node (if any) are the same. So the strategy to find the "split" is to compare the nodes of the paths from the starting node of those paths and when different nodes are found in the paths, the node just before the different nodes is the "split" node.

Similarly to find "join" node, we start with the rest of the paths after the "split" node and compare the nodes of the paths to find one common node which will be the "join" of the paths.

`findSplitJoin` finds the "split" and "join" of the two paths. It uses `findSplit` to find the "split" node. Then it uses `findJoin` with the rest of the paths after the "split" (we can get it from `findSplit`). In `findJoin`, it just compares the nodes in one path with the node elements of other paths to find one common node, which will be the "join" node of the paths.

```
findSplitJoin :: [FinalPath] → (Maybe MyNode, Maybe MyNode)
findSplitJoin fps = (splt, join)
   where (ptl1,splt) = findSplit fps Nothing
```

```
        join = findJoin ptl1 (head ptl1)

findSplit :: [FinalPath] → Maybe MyNode → ([FinalPath], Maybe MyNode)
findSplit fps nds = if cond ≡ True then findSplit fps1 nds1
                                   else (fps, nds)
          where fsnd = (head·snd)
                ndc = fsnd $ head fps
                cond = and $ map (λln → fsnd ln ≡ ndc) fps
                fps1 = map (λx → (fst x, (tail·snd) x)) fps
                nds1 = Just ndc

findJoin :: [FinalPath] → FinalPath → Maybe MyNode
findJoin fps (_,[]) = Nothing
findJoin fps fstp = if cond ≡ True
                    then Just ndc
                    else findJoin fps (fst fstp, (tail·snd) fstp)
          where ndc = (head·snd) fstp
                cond = and $ map (λln → ndc `elem` (snd ln)) fps
```

As we mentioned in ***FindExpr*** module, we have to divide the paths in the branching structure into four different segments to find the symbolic expression of the code. `getCommonDiv` is a helping function to divide the paths in segments using "split" and "join" node with simple list functions like `takeWhile` and `dropWhile` and is used to divide the paths in GSC structure.

FindLoopPart also finds different segments of the paths in the Looping Code (LC) structure and uses `getLoopParts` to find those segments.

```
type Split = MyNode
type Join = MyNode
type FirstComm = Path
type SecondComm = Path
type PtDiff = Path

type ComP = Path
```

```
type TermT = Path
type LoopX = [Path]
type LoopY = Path
type ComZ = Path

getCommonDiv :: [MyNode] → [MyNode] → (Split, Join)
              → (FirstComm, PtDiff, PtDiff, SecondComm)
getCommonDiv pt1 pt2 (st, jn) = (fc, pd1, pd2, sc)
 where fc = takeWhile (≠ st) pt1 ++ [st]
       pd1 = dropWhile (≠ st) ((takeWhile (≠ jn) pt1) ++ [jn])
       pd2 = dropWhile (≠ st) ((takeWhile (≠ jn) pt2) ++ [jn])
       sc = dropWhile (≠ jn) pt1


getLoopParts :: [FinalPath] → (Split, Join)
              → (ComP, TermT, LoopX, LoopY, ComZ)
getLoopParts fps (st, jn) = (cp++[st],tt, lx,ly,cz)
 where  cp = takeWhile (≠ st) $ (snd·head) fps
        ppairs = partition (λx → fst x ≡ Term) fps
        findTL = (dropWhile (≠st))·snd
        fpsT = map findTL (fst ppairs)
        fpsL = map findTL (snd ppairs)
        findLastC = (dropWhile (≠jn))·head
        cz =  findLastC fpsT
        findDiv = takeWhile (≠ jn)
        fdT = map (++[jn]) $ map findDiv fpsT
        fdP = map (++[jn]) $ map findDiv fpsL
        lx = ∩ fdT fdP
        tt = concat $ lx \\ fdT
        ly = findLastC fpsL


findLoopPart :: MyGraph → MyNode → (ComP, TermT, LoopX, LoopY, ComZ)
```

```
findLoopPart mg st =
       if (sp ≠ Nothing ∨ jn ≠ Nothing)
          then getLoopParts paths (fromJust sp, fromJust jn)
          else error "no join split"
     where paths = nodesFromStart mg st
           (sp,jn) = findSplitJoin paths
```

F.3 Other Modules

The other modules to generate data flow equations are ***Exp*.lhs** (See Section 8.4.1), ***FindPath*.lhs** (See Section 8.5.1) and ***FindExpr*.lhs** (See Section 8.5.2) which are already included in the thesis.

Appendix G

Data Flow Graph Generator

Here we include the implementation of Data Flow Graph generator.

G.1 DFDGxl

The main module is called DFDGxl.

This tool converts Data Flow Equations for an assembler code to a GXL file of Data Flow Graph (DFG). Before creating the GXL representation of the DFG, we create our own internal DFG which is converted to GXL file for exchanging with next standard tools like `gxl2dot` and `dot`.

```
module Main where
import qualified Gxl
import GxlGraph (makeGxl, makeGraph, addOrdAttr, GxlGxl, GxlGraph)
import MyGraph (gxlToMyGraph)
import FindExpr (findAnntOfGraph)
import Exp (Stmt, ConditionStmt)
import Dfe2Dfg (dfdGraphToGxlGraph)
import Text.XML.HaXml.Xml2Haskell (fReadXml, fWriteXml)
import System (getArgs)
import Observe

main = do [infile, start, outfile] ∈ getArgs
```

```
    value ∈ fReadXml infile :: IO Gxl.Gxl
    let name = takeWhile ('.' ≠) infile
    let gxl = dfeToGxl name $ findAnntOfGraph
              (gxlToMyGraph value ((length name)+1)) start
    fWriteXml outfile gxl
    putStrLn "Done."
```

Here we convert the graph file into the GXL graph.

```
dfeToGxl :: String → [[([ConditionStmt], [Stmt])]] → GxlGxl
dfeToGxl name cstmts = makeGxl $ dfeToGraph name cstmts
```

Converting Data Flow Equations to a graph file involves making a graph which has default **edgeid** and **hypergraph** attributes and edges are directed. **dfdGraphToGxlGraph** generates the GXL file from the DFG internal data structure and then adds a graph attribute (in) to keep the original order of the operand nodes i.e. order of expressions in the instructions.

```
dfeToGraph:: String → [[([ConditionStmt], [Stmt])]] → GxlGraph
dfeToGraph name cstmts = addOrdAttr "in" $ dfdGraphToGxlGraph
                                    (makeGraph name) name cstmts
```

G.2 Dfe2DfgCommon

This module contains some helper functions which are straight forward. It also includes all the functions to convert the DFG into GXL format.

```
module Dfe2DfgCommon
( makeId, opnodeToId, optnodeToId, inEdgeToIds, pairsToGxlEdges
 , makeOpnOptPairs, outEdgeToGxlEdges, idToGxlEdge, idToGxlNode
 , stringToNodeAttrs, condToNodes, makeBlankPairs, addToNodeMap
 , conv2Expr
)
where
```

```
import MyPrelude (snd3)
import GxlGraph (GxlNode, GxlEdge, NodeId, makeNode,
                 makeEdge, makeNodeId,
                 makeStringAttr, addNodeAttrib, addEdgeAttrib)
import Data.FiniteMap (addToFM)
import Stack
import Exp
import Dfg
```

Helper functions for removing redundant codes.

```
labelAndNode name e ci opnd s = (e, makeId (name ++ ci) opnd, s)
makeId name x = makeNodeId name $ show x


optToId name optn = snd3 $ optnodeToId name optn
opdToId name opdn = snd3 $ opnodeToId name opdn
```

opnodeToId creates the label, ***NodeId*** and the shape for each ***OperandNode***.

```
opnodeToId :: String → OperandNode → (String, NodeId, String)
opnodeToId name (opnd@(ExpressionNode e@(SignBit t) ci)) =
    labelAndNode name (show e++show ci) (show ci) opnd "ellipse"
opnodeToId name (opnd@(ExpressionNode e ci)) =
    labelAndNode name (show e) (show ci) opnd "ellipse"
```

optnodeToId creates the label, ***NodeId*** and the shape for each ***OperatorNode***.

```
optnodeToId :: String → OperatorNode → (String, NodeId, String)
optnodeToId name (optr@(Unary tc i))
      = labelAndNode name (show tc) (show i) optr "box"
optnodeToId name (optr@(Binary op i))
      = labelAndNode name (show op) (show i) optr "box"
optnodeToId name (optr@(ConditionVal cf i))
      = labelAndNode name (show cf) (show i) optr "diamond"
```

```
optnodeToId name (optr@(Join ci1 ci2))
       = labelAndNode name "Join" (show ci1 ++ show ci2)
                          optr "doublecircle"
```

inEdgeToIds creates a list of *NodeId* pairs for the *InEdge*s.

```
inEdgeToIds :: String → [InEdges] → [(NodeId, NodeId)]
inEdgeToIds name ines = concat $ map (makeOpnOptPairs name []) ines
```

pairsToGxlEdges creates GXL edges from pairs of *NodeId*s as source and destination node.

```
pairsToGxlEdges :: [(NodeId, NodeId)] → [GxlEdge]
pairsToGxlEdges = map (uncurry makeEdge)
```

makeOpnoptPairs makes pairs of an *OperandNode* and its successor list of *OperatorNode*s.

```
makeOpnOptPairs :: String → [(NodeId, NodeId)]
                → InEdges → [(NodeId, NodeId)]
makeOpnOptPairs name opnds (opnd,[]) = opnds
makeOpnOptPairs name opnds (opnd,(optr:optrs))
 = makeOpnOptPairs name (opnds++[(sn,fn)]) (opnd,optrs)
      where sn = opdToId name opnd
            fn = optToId name optr
```

OutEdgeToGxlEdges creates the *GxlEdge*s for the *OutEdge*s and uses idToGxlEdge to make the *GxlEdge*s.

```
outEdgeToGxlEdges :: String → [OutEdges] → [GxlEdge] → [GxlEdge]
outEdgeToGxlEdges name [] ndnds = ndnds
outEdgeToGxlEdges name ((ns,(OneEdge on)): ines) ndnds =
   outEdgeToGxlEdges name ines (ndnds++[idToGxlEdge (sn, NC, fn)])
      where sn = optToId name ns
            fn = opdToId name on
outEdgeToGxlEdges name ((ns,(TwoEdge (nc1,on1)
```

```
                          (nc2,on2))): ines) ndnds =
  outEdgeToGxlEdges name ines (ndnds++ (map idToGxlEdge
                        [(sn,nc1,fn1),(sn,nc2,fn2)]))
      where sn = optToId name ns
            fn1 = opdToId name on1
            fn2 = opdToId name on2
outEdgeToGxlEdges name ((ns,(ThreeEdge (nc1,on1)
            (nc2,on2) (nc3,on3))): ines) ndnds =
  outEdgeToGxlEdges name ines (ndnds++(map idToGxlEdge
       [(sn, nc1, fn1),(sn, nc2, fn2),(sn, nc3, fn3)]))
      where sn = optToId name ns
            fn1 = opdToId name on1
            fn2 = opdToId name on2
            fn3 = opdToId name on3
```

idToGxlEdge creates GXL edge from triples of source and destination *NodeId*s and *Cond* which are created from *OutEdge*s.

```
idToGxlEdge :: (NodeId, Cond, NodeId) → GxlEdge
idToGxlEdge (nd1, cd, nd2) = addEdgeAttrib eattr (makeEdge nd1 nd2)
       where eattr = makeStringAttr "label" (show cd)
```

This function makes a GXL node and also add its name and shape attributes.

```
idToGxlNode :: (String, NodeId, String) → GxlNode
idToGxlNode (name, nodeName, shape)
        = stringToNodeAttrs "shape" shape node
 where node = stringToNodeAttrs "label" name
                        $ makeNode nodeName
```

This Function adds the name attribute of the node.

```
stringToNodeAttrs :: String → String → GxlNode → GxlNode
stringToNodeAttrs nm v = addNodeAttrib (makeStringAttr nm v)
```

condToNodes is used to create the condition nodes.

```
condToNodes :: ConditionStmt → Int → OperatorNode
condToNodes (Check (brc,cd)) i = ConditionVal brc i
```

makeBlankPairs adds the output *OperandNode*s of one instruction in the *DfdGraph* with their successor list empty.

```
makeBlankPairs :: [OperandNode] → DfdGraph → DfdGraph
makeBlankPairs [] dfg1 = dfg1
makeBlankPairs (opnd:opnds) dfg1@(DfdGraph fmo fme)
        = makeBlankPairs opnds (DfdGraph nfmo fme)
       where nfmo = addToFM fmo opnd []
```

addToNodeMap adds a list of *OperandNode*s in the *NodeMap*.

```
addToNodeMap :: NodeMap → [OperandNode] → NodeMap
addToNodeMap nMap [] = nMap
addToNodeMap nMap (opnd@(ExpressionNode e ci): opnds) =
     addToNodeMap (addEntry nMap (e,opnd)) opnds
```

conv2Expr does an important conversion from *StateRef* to *OperandNode*. When we are adding output entries to the *NodeMap*, all of the output entries in the statements are *StateRef*s. Whereas when we are looking up for entries in the *NodeMap*, *OperandNode* entries are being looked up. This function converts *StateRef* to *OperandNode* which contains *BasicExp* entries so that they can be added in the *NodeMap* for lookup.

```
conv2Expr :: StateRef → Int → OperandNode
conv2Expr (SC sc) i = ExpressionNode (Variable sc) i
conv2Expr (SCX scx) i = ExpressionNode (VariableX scx) i
conv2Expr (Mem mr) i = ExpressionNode (MemoryConstant mr) i
```

G.3 Other Modules

The other modules to generate data flow equations are *Dfg*.lhs (See Section 9.3), *Dfe2Dfg*.lhs (See Section 9.4) and *GarbageCollect*.lhs (See Section 9.5) which are already included in the thesis.

Appendix H

DFE Solver

Here we include the implementation of Data Flow Equation solver.

H.1 FindPathAnt

The main module is called FindPathAnt.

This tool is used to find the symbolic interpretation of a given code segment i.e. the solution of the functional expressions of inputs and outputs. It also finds the execution sequence and system of equations of the given code segment. We take the Control Flow Graph of that code as input.

```
module Main where

import qualified Gxl
import Text.XML.HaXml.Xml2Haskell
import System (getArgs)
import MyGraph
import SolveExpr (Input, Output, dividePathCondSym)
import Exp
import Data.List (intersperse)
import Control.Monad.Error
import IO
```

The main function of the program takes three arguments: They are one GXL file that is the Control Flow subgraph (of the code segment), start node, and the output file to be generated. By reading the GXL file, we make the our internal representation of the subgraph and then find the symbolic interpretation of the code in the output file.

```
main = (do
        [infile, start, outfile] ∈ getArgs
        putStrLn ("Reading from "++infile)
        value ∈ fReadXml infile :: IO Gxl.Gxl
        putStrLn ("Writing to "++outfile)
        let sval = length $ takeWhile (≠ '.') infile
        putStrLn (show sval)
        if (outfile ≡ "-")
           then putStr $ show $ prtyPrint value start sval
           else do writeFile outfile $ show
                                $ prtyPrint value start sval
                   putStrLn "Done."
      ) `catchError` usage

usage :: IOError → IO ()
usage e = do
          putStrLn "Usage: FindPathAnt [input.gxl] start [output]"
```

These functions are used for pretty printing. They just separate different outputs and print them in a nice manner.

```
prtyPrint :: Gxl.Gxl → String → Int → [String]
prtyPrint value st sv = tuplePrint tupleList
 where tupleList = dividePathCondSym (gxlToMyGraph value (sv+1)) st

tuplePrint :: (ConditionStmt,[Recur_Stmt],Input,Output,[Recur_Stmt])
                    → [String]
tuplePrint (pCond,exec,input,output,systEq) =
```

```
intersperse " " (
"PathCondition:" : show pCond :
"Instruction Execution:" : show exec :
"Input:" : show input :
"Output:" : show output :
"System Of Equations:" :show systEq :[])
```

H.2 Other Modules

The only other module to solve data flow equations is *SolveExpr*.lhs (See Section 10.2) which is already included in the thesis.

Bibliography

[ASU86] Alfred V. Aho, Ravi Sheti, and Jeffery D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[CC88] D. L. Clutterbuck and B. A. Carre. The verification of low-level code. *Software Engineering Journal*, 3(3):97–111, May 1988.

[CG03] Dino Mandrioli Carlo Ghezzi, Mehedi Jazayeri. *Fundamentals of Software Engineering.* Prentice Hall, 2nd edition, 2003.

[CKK+04] J. Carette, W. Kahl, R. Khedri, M. Lawford, K. Sartipi, and A. Wassyng. Procedure for reverse engineering of high-level requirements from assembly code. Technical Report Revision-0, Reverse Engineering Project, Dept. of CAS, McMaster University, July 2004.

[DMW05] Ivo Düntsch, Wendy MacCaull, and Michael Winter, editors. *8th International Conference on Relational Methods in Computer Science (RelMiCS 8) and 3rd International Workshop on Applications of Kleene Algebra, St. Catherines, Ontario, Canada, Feb. 22–26 2005*, 2005. (participants' proceedings, to appear).

[Eve04] Kevin Everets. Assembly language representation and graph generation in a pure functional programming language. Master's thesis, Dept. of Computing and Software, McMaster University, December 2004.

[FF95] Y.A. Feldman and D. A. Friedman. Portability by automatic translation; a large scale case study. In *Proc. 10th Knowledge-Based Software Engineering Conference*, 1995.

[FS03] Thomas Fahringer and Bernhard Scholz. *Advanced Symbolic Analysis for Compilers: New Techniques and Algorithms for Symbolic Program Analysis and Optimization*, volume 2628 of *Lecture Notes in Computer Science*. Springer, 2003. DBLP, http://dblp.uni-trier.de.

[Has] *The Haskell Home Page.* Electronically available at http://www.haskell.org/.

[How05] D. Howe, editor. *The Free On-line Dictionary of Computing.* June 2005. Electronically available at http://wombat.doc.ic.ac.uk/.

[Hug90] John Hughes. *Why Functional Programming Matters, In D. Turner, Editor, Research Topics in Functional Programming.* Addison Wesley, 1990.

[IBM70] *IBM Field Engineering Theory of Operation, 1800 Data Acquisition and Control System, Processor-Controller.* IBM Systems Development Division, Product Publications, Department G24, San Jose, California 95114, 1970.

[KAC04] Wolfram Kahl, Christopher Kumar Anand, and Jacques Carette. Choices in data flow for declarative assembly. In Düntsch et al. [DMW05]. (participants' proceedings, to appear).

[K.H95] K.H.Bennett. Legacy systems: Coping with success. In *IEEE Software*, volume 12, No. 1, pages 19–23, January 1995.

[Knu84] Donald E. Knuth. *Literate Programming*, volume 27(2), pages 97–111. The Computer Journal, 1984.

[LB96] Tom Lake and Tim Blanchard. Reverse engineering of assembler programs: A model-based approach and its logical basis. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'96).* IEEE Computer Society, 1996.

[MF96] P. Morris and R. Filman. Mandrake: A tool for reverse-engineering ibm assembly code. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'96)*, pages 58–65, November 1996.

[MGH+01] Michael B. Monagan, Keith O. Geddes, K. Michael Heal, George Labahn, Stefan M. Vorkoetter, James McCarron, and Paul DeMarco. *Maple 7 Programming Guide.* Waterloo Maple Inc., 2001.

[NN99] Hanne Riis Nielson and Flemming Nielson. *Semantics With Applications: A Formal Introduction.* John Wiley and Sons, July 1999.

[OCF+88] I. M. O'Neill, D. L. Clutterbuck, P. F. Farrow, P. G. Summers, and W. C. Dolman. The formal verification of safety-critical assembly code. In W. D. Ehrenberger, editor, *Safety of Computer Control Systems 1988*, pages 115–120. International Federation of Automatic Control, Pergamon Press, November 1988.

[OSRSC01] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS System Guide, Language Reference and Prover Guide.* Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001.

[Pro05] *Program-Transformation.Org.* June 2005. Electronically available at http://www.program-transformation.org.

[RPK96] S. N. Roberts, R. L. Piazza, and D. G. Katz. A portable assembler reverse engineering environment (PARE). In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'96).* IEEE Computer Society, 1996.

[Tur37] A.M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume Series 2, 42, pages 230–265, (1936-37). Electronically available at http://www.abelard.org/turpap2/tp2-ie.asp.

[War00] Martin Ward. Reverse engineering from assembler to formal specifications via program transformations. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, NOV 2000.

[WF03] Geoffrey Watson and Colin Fidge. Modelling assembler programs with an application to compilation. Technical Report 03-GW-1, Software Verification Research Centre, The University of Queensland, July 2003.

[Wic05] *Wikipedia, The Free Encyclopedia.* June 2005. Electronically available at http://en.wikipedia.org.

[Win01] Andreas Winter. Exchanging graphs with GXL. Technical Report 9-2001, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1,Koblenz, 2001. D-56075.

[Wu04] Jun Wu. Formalization of GXL in Z notation. Master's thesis, Dept. of Computing and Software, McMaster University, 2004.