Assembly Representation in a Functional Programming Language

Assembly Language Representation and Graph Generation in a Pure Functional Programming Language

By
Kevin Everets, B.A.Sc.

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 SCIENCE(2004)
(Computer)

McMaster University
Hamilton, Ontario

TITLE: Assembly Language Representation and Graph Generation in a Pure Functional Programming Language

AUTHOR: Kevin Everets, B.A.Sc.(Queen's University)

SUPERVISOR: Dr. Wolfram Kahl

NUMBER OF PAGES: viii, 99

Abstract

In industry many legacy systems exist which run mission or safety critical code which do not have adequate requirements documentation. This thesis demonstrates how the use of a functional programming language eases a flexible and modular approach to the construction of libraries and tool suites that allow the manipulation of assembly language programs. The tools and libraries created with this method are used in a larger project of reverse engineering requirements from legacy assembly programs.

The modules presented operate from the assembled ".lst" format, which is the result of assembling the source files, and includes the calculated address in memory and the binary version of the given instructions. Our libraries provide representations of assembly programs in an abstract data type and as internal graph representations, and conversions to a GXL graph format and to other special-purpose XML representations.

The use of Haskell as an implementation language is explored in the context of a software engineering project, and some of the benefits and disadvantages of such a choice are discussed.

This work was funded by Ontario Power Generation and CITO (Communications and Information Technology Ontario).

Contents

List of Figures

Chapter 1

Background

First, some background on the work is provided. This includes information on the reverse engineering project which this work was created for, and the context in which the project is operating. The changing role of legacy systems in industry is described, as well as the particular instance of legacy systems at Ontario Power Generation, and their needs for the work provided herein.

1.1 Legacy Systems

Several industries have computer systems which were conceived and implemented many years ago. These systems are often running safety and/or mission critical software which was written in a variety of assembly languages. This software was written not with clarity, but with efficiency as the main goal, as computer time was vastly more expensive than programmer time. As such, these programs were written to take full advantage of the architecture of the legacy hardware systems they were written for, and often had to work around the constraints of such architectures. This makes for some convoluted code.

Over time documentation procedures and formats are changed and documentation systems are replaced. These changes result in the documentation for legacy systems and software becoming misplaced or destroyed. When the documentation is for software which is not safety critical, less care is exhibited in both the creation and the maintenance of the documentation as time goes on. The people who designed and wrote the software may be unavailable (working elsewhere, retired, or deceased). Ultimately, companies can and do find themselves running legacy software for which they no longer have (or occasionally never had) the original requirements or design documents, and no personnel who could recreate them. As the hardware ages, it becomes impossible or infeasible to replace failed components and so alternatives are sought, such as emulating the hardware with newer more maintainable machines, or possibly reimplementing the systems on current hardware.

Without proper requirements and design documentation, the temporary fix of emulating the older hardware is frequently the first attempted. It is often easier to validate the emulators than to determine the requirements which the software itself must meet. This fix is rarely sufficient in the long run, however, as occasional maintenance is still required on the software to conform to changes in its environment. Any changes to the software become expensive and hazardous.

These systems have worked well for many years, and might continue to work well for some time into the future, but it would be a very precarious position to depend upon them in their current state. Industries, as a result, find themselves in a difficult situation of having very old, difficult to maintain, poorly documented systems which they are very reliant upon, and which they would like to re-engineer to create more modern, maintainable systems.

1.2 Ontario Power Generation

One such industry is the nuclear industry. Ontario Power Generation (OPG) has software which was originally written 30 years ago which is still in use today, 20 years after the original developers have left. Some of this software was not safety critical, and so it was either not rigorously documented during its design and implementation, or its documentation has been difficult to locate and collate.

Additionally, patches were necessary to the code during its lifetime to deal with changes in plant structure or the surrounding system. Some of these patches had to work within extremely confined resources, and as such the source has become a tangle of layers of patches which must be unwoven to gain full comprehension of the implications of any modification. This leaves a system where making changes is a dangerous and potentially very expensive process. Though it is good that extreme care is taken when updating the running system, it is unfortunate that such care is critical and fear of failure is so high.

OPG is trying to rectify this situation by rigorously examining each piece of software to determine the original requirements, with the view towards a reimplementation where the best current Software Engineering principles are applied to generate functionally correct, robust, maintainable software which will continue to operate without fail for another 30 years. To this end, they are funding (along with CITO – Communications and Information Technology Ontario) a project of reverse-engineering of software requirements from assembly code at McMaster University. The work presented in this thesis is part of that project.

Currently, there are two main machine types used which the Re-Engineering project is examining the source code for: the IBM 1800 and the Varian V75. We (the Reverse Engineering project group described in Section 1.4) were initially given the Boiler Pressure Control code, which is one module of a larger piece of software that runs on the IBM 1800. Later we were given what is thought to be all of the assembly source listings for the complete piece of software, though there remain issues in accurately determining which source files are used to generate a binary image as there are two machines (DCC 1 and DCC 2) with four subunits (UNITS 1-4) and each of these subunits runs a slightly different image than the rest. An attempt has been made to generate a complete view of the code for one unit (DCC1 Unit 1), and more information relating to requirements could potentially be gleaned by comparing the slight differences between the different images for the different machines and subunits.

OPG is no longer using actual IBM 1800 machines and is instead using emulators as a stop-gap measure as mentioned earlier. This has allowed them to avoid issues with maintaining very old computer hardware, at the cost of validating the emulator as an accurate representation of the behaviour of an IBM 1800 machine. They still have all of the software maintenance issues which make changing the code in any way a costly procedure. Therefore it is hoped that the Re-Engineering efforts will allow them to easily recover the requirements of their systems and thus be able to re-implement them on modern systems and move away from the constraints of the IBM 1800 and Varian V75 systems.

1.3 IBM 1800

The IBM 1800 Data Acquisition and Control System was introduced in 1965 [AK03] and is a variant of the IBM 1130 Computing System utilized for process control [AK03]. The IBM 1800 extended the IBM 1130 with additional instructions and extra I/O capabilities. The benefit of using the 1800 is that it was designed for real-time process input/output (both analog and digital). It was also relatively small compared to other computers at the time (being about the size of a large wardrobe).

The instruction set for the IBM 1800 is what we would consider RISC, having 31 separate opcodes. These operate using 8 registers, mainly manipulating only the program counter or Instruction Register (I), the Accumulator (A) and Accumulator Extension (Q) registers. The I, A, and Q registers are all 16-bits in width. Additionally, there are three Index Registers which can be used to store pointers or other information which will be combined with addresses given in the instructions to generate an "Effective Address".

The instructions themselves are one of two lengths, either a 16-bit short instruction, or a 32-bit long instruction (specified by the Format bit at bit position 6 in both formats). Due to the small register space, most operations operate on the memory space directly (where both instructions and data exist simultaneously – there is no separation between them which means that instructions can be, and are, manipulated as though they were data).

Assembly programs are written to adhere to a strict format with well-defined column number ranges for each field, as initial programs were written on punched cards or forms. There are optional labels (up to 5 characters wide, starting with an alphabetical character), opcodes are written using a 1-4 character mnemonic, and the format of the instruction (short, long, indirect, or other) is specified in a single-character field. The tag field is used to indicate whether the address given in the operand is to be added to, stored in, or loaded from the value of one of the three index registers. There's also an "operand & comment" field (which allows operands to be expressed up to the first white space, potentially including commas for delimiting different parameters, and the rest is considered an inline comment). Block comments are also allowed, denoted by a "*" character in the first position of the label field.

In addition to the standard opcodes, the assembler includes some directives such as

ABS to specify absolute addressing, DC for entering raw data into a memory location, EQU for defining mnemonics for particular values to be used in the assembly, ORG to denote where in memory the following instructions should be located, and even a small macro language (which was not used in the OPG code), among others.

The assembly program is assembled into either a binary image which would be loaded directly into the machine, or an assembly listing (LST) file. The LST file is very similar to the original assembly file, but it includes the address which the assembler has determined will hold the given instruction, as well as the object (hexadecimal) representation of the instruction or data (given as either four or eight nibbles to represent the 16 or 32 bit instructions, respectively). The LST also includes the line number (ST.NO.) in the original assembly program where the instruction or data was entered, and an indication (REL) of which 16-bit words use relative or absolute addressing when being loaded into memory. It is this LST file which is manipulated by the tools to generate control flow graphs and provide alternate representations of the assembly programs. Initially the LST files were generated using an open source assembler for the IBM1130 (with the additional instructions for the IBM1800 added). Later OPG provided the LST files which they use, which was in a similar format but generated by their own assembler.

1.4 Re-Engineering Project

The tools and libraries presented in this paper were required for a project of Reverse Engineering at McMaster. The following is a brief overview of the projects goals, and the Tool Suite which is being created to satisfy those goals.

1.4.1 Goals

The goal of the CITO project at McMaster, *Reverse Engineering High-Level Requirements from Assembly Code*, "is to to create methods and tools to assist a developer in reverse engineering a legacy assembly language 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+04a] The idea is to take the existing assembly language programs from industry (source with comments, when available), and while using as minimal human interaction as possible, generate a Requirements Document which can then be used to Re-Engineer the software for modern systems while meeting the same requirements as the original.

There have been previous efforts to deal with this situation by taking the legacy code and translating it into a higher level language, such as C or languages designed specifically for this task. These efforts tend to produce code which, though more portable than the original, is not much more maintainable than the original. Determining which requirements are part of the problem domain and which are due to the chosen architecture is also not often considered with this method.

The project at McMaster is instead generating tools in a comprehensive tool suite to aid in obtaining the higher level requirements.

1.4.2 Tool Suite Architecture

The Reverse Engineering project is generating both a Tool Suite and a Procedure which will be used in its efforts to generate Requirements Documentation from the (possibly annotated) assembly source code. Figure 1.1 (adapted from [CKK+04b]) gives an overview of the interaction between the tools in the tool suite, where arrows are "used by" relations.

The work presented in this thesis fits in the highlighted boxes at the bottom and bottom-right portions of the graph, that is the "Assembly Representation Library & Emulators" and "Graph Generation Tools & Library". This work is thus used by the other main tools in order to generate Semantic Analysis, Graph Analysis, Functionality Analysis & Design Recovery, Timing Analysis, and ultimately the Requirements Validation & Verification (V&V) Tools.

The Assembly Representation Library allows tools to be created which can read in one of several different representations of the assembly language programs. Emulators have already been created using this library.

The Graph Generation Tools & Library creates the Control Flow Graph representation of the assembly language program. It is used for visualization of the operation of the code, as well as manipulation of the resulting graphs in order to perform extended data mining and information retrieval from the patterns therein.

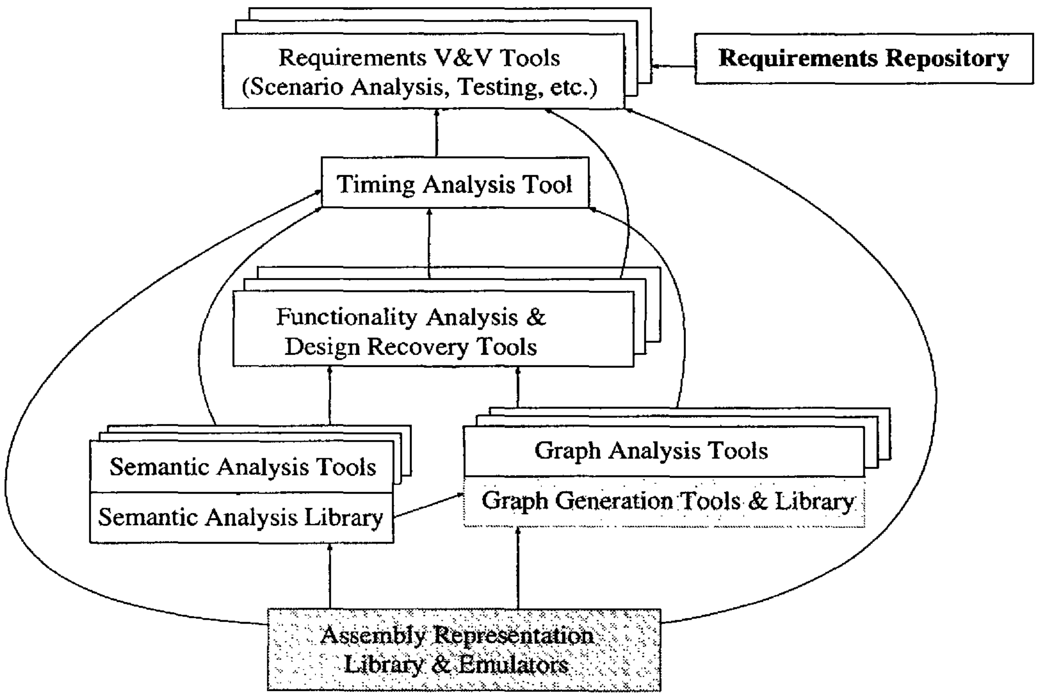

Figure 1.1: Overview of Reverse Engineering Project

Chapter 2

Introduction

Contained in this chapter is an overview of the thesis, as well as some of the main goals of the tools and libraries presented (and how well those goals were achieved). Haskell was used extensively to meet these goals, and the reason for such is described in Section 3.2.1.

2.1 Overview

First, the goals which guided the generation of the tools and libraries used in the Reverse Engineering project are presented. The success of these goals is discussed within each section. Following that, the details of the assembly language representation are shown from the given requirements, through some design and the implementation (which is shown in its entirety as a literate program).

Once the assembly language representation is established, its use in the generation of control flow graphs is explained. The tools which are used to generate and visualize these graphs is also given as literate programs. An example is shown of the results of the code on some assembly code.

Next, some of the auxiliary programs manipulating and interpreting the graph which were created to aid in the reverse-engineering project are described. Additionally, other contributions which were made during the creation of these tools and libraries are described.

2.2 Representing Different Architectures

When generating the initial modules of the Assembly Representation Libraries, a main goal was to allow different architectures to be represented so that with minimal changes the tools could be altered to perform their function on the assembly code for those architectures. The architectures would have differing instruction sets, numbers of registers, sizes and types of memory, and timing mechanisms.

Allowing for the representation of different architectures would thus allow the reuse of the same code for Re-Engineering of IBM 1800 assembly, Varian V75 Assembly in the near term (as those are the two architectures that OPG is most concerned with), as well as MIPS and other architectures that are in more general use.

The use of a pure, lazy functional programming language is a novel means of attaining this goal. The facilities available in such a language were explored and utilized in an effective manner to modularize and abstract the code base in such a way that changes to the internal structures were isolated and localized. The functions operating on these structures and their associated access functions continue to work without modification even though there were substantial changes required within a few modules to support new architectures. The result of this work was that the code was able to be ported to the Varian V75 architecture in relatively little time.

In retrospect, the goal of architecture independence was not fully attained as the changes necessary to support the V75 were more invasive than originally thought, and the source code that needed to be altered was spread across several modules. In the future it would be advantageous to more fully isolate the platform specific code wherever possible. As time did not permit the re-factoring of the code, this work was not performed as part of this project.

2.3 Tool Interaction

Another main goal was to create tools which would work well together and with outside tools. As there are several pre-existing Re-engineering tools, it was decided to provide intermediate representations which would allow those tools to be leveraged where possible. Many of these tools use a graph format called GXL [HWS00] (an acronym for Graph eXchange Language), which is discussed more thoroughly in Section 3.2.2. GXL is an XML sub-language, and it was hoped that the GXL would be able to be used directly by the XML database work of Mark Pavlidis. Unfortunately, the database chosen made working with GXL directly costly in terms of loading time, so a new XML format was designed specifically for that purpose. A conversion program was then created to convert from the GXL representation to the new XML format. This was eased by the conceptual clarity of the previous GXL format.

For internal representations of the assembly, a model of the machine and a data type representation of the code and the control flow was generated in Haskell (the discussion of the selection of Haskell is discussed in Section 3.2.1). This allowed other in-house generated tools such as the emulators and GXL validation tools to share internal representations and speed development.

The speed and accuracy with which additional tools were created is a testament to the ability of a pure functional programming language to aid Software Engineering. Using functions as a primary object (instead of the more traditional behaviour of treating functions as ancillary to a data object) and following the small number of guidelines created to develop tools allowed for new tools to be quickly developed and interact well with each other.

2.4 Reusability

Reusability of representations and code was a primary concern in the creation of the Graph Generation Tools and Assembly Representation Library. The use of Haskell as a language for generating the representations and tools has allowed for other tools to be quickly generated reusing the same functions in many cases, or overriding those that need to be modified for a particular use. As an example, the code to read the LST file and create initial control flow is used both by the emulators and the control flow graph visualization programs, and the code to read and write GXL representations is used by many of the tools to interact with each other. The reuse of existing modules to provide quick and accurate development of succinct programs is demonstrated through the creation of several such tools.

Generating interfaces such that these libraries could be used with other programming languages, though possible, has not yet been explored. Doing so would increase the utility of the libraries and extend their reach.

2.5 Information Flow

When representing the code and generating graphs from the code, a goal was to retain as much information as possible (such that, for example, the entire LST file can be reconstructed from the XML representations), and to provide as much additional information as possible at the early stages with the view that unnecessary information would later be filtered.

An example of this strategy in action is that the control flow generation results in more paths than will actually be possible with the code due to the semantics of the interaction of the instructions. Rather than attempt to detect and remove the infeasible paths during the generation of the control flow, the extra paths are left in place and it is then up to the operator (potentially assisted with additional tools) to cull the extraneous flow. Tools have been created to aid in this process using the web interface to the XML database to select and store human-decided infeasible paths, and a tool to filter those edges from the given GXL representation before it is passed on to other tools (such as the Symbolic Emulator).

Chapter 3

Assembly Representation

For the CITO project, there were the following utilities produced: a library of data structures and functions to operate on assembly LST files and perform some initial analysis on them, a library of data structures and functions to operate on GXL files, command line tools to perform those operations and conversions between formats in an automated way, and infrastructure to easily combine those tools into a coherent package and allow for visualization of the results.

The following sections will discuss the requirements, design, and current users of the libraries and tools.

3.1 Requirements

From the Requirements for Reverse Engineering Tools (Rev 0) [CKK+04a], the Assembly Representation Library and Tools needed to provide:

- representations of assembly languages and machine languages,
- representations of assembly and machine programs,
- representations of machine models.

The representations were to be derived from the available documentation, and all information had to be represented in a form that allowed easy inspection and validation.

Emulators were to be used to evaluate code segments for the purpose of testing and answering specific questions about the system's behaviour.

3.2 Design

It was decided to follow in the Unix tradition of creating many small tools that each perform one function well and that can interact through pipes between programs while using shared formats. This allows a person to tie different tools together in order to meet their needs. In this way, tools were created which would read in a given representation of an assembly language program and convert it into one of a few common formats for manipulation. GXL was chosen as the primary common format, for reasons outlined in Section 3.2.2.

Once in the GXL format, the graphs could be manipulated (by having subgraphs taken from them, or portions highlighted or removed, or transformations performed to aggregate pieces into a more abstract view of the graph, all performed by individual programs on the GXL file). Once any desired manipulations had been performed, the GXL files could then be converted to formats more suitable for visualization. A tool for GXL to DOT conversion was thus created which would allow for such visualization (as described in Section 4.3).

It was found that a unique XML schema would assist the development of the XML database as a repository for the assembly code (thus allowing the operator to manipulate the code and its structure using a web browser as the main interface). As a result, an additional tool was created which reused the parsing and graph manipulation code of the original Lst2Gxl program combined with a module describing the new datatype, to form a new Lst2Xml program.

Each of these tools needed to be able to read the Lst file, or read or write the GXL. It was thus desirable to create common libraries for interactions with those representation. Now when any tool is required to work with the representations it can be created quickly using these libraries.

3.2.1 Haskell

Haskell was chosen as the implementation language for the representation and manipulation of the IBM 1800 assembly code. Haskell is a pure functional programming language that is suitable for general purpose use. There are freely available compilers for almost any modern computer and operating system (the compiler used in this project, the Glasgow Haskell Compiler, either generates C code as an intermediate step, or on some architectures can generate native code directly).

One of the best features of Haskell is its great support for the creation and manipulation of abstract data types. Initially the Lst2Gxl program was created without using very much abstraction of the Gxl format. Upon consultation with Dr. Kahl and Dr. Carrette, a new abstraction layer was placed on top of the Gxl representations to generate a new abstract data type combined with generation, manipulation, and access functions. Using this new abstract data type and associated functions resulted in the Lst2Gxl program shrinking significantly in size while increasing the clarity and improving execution times.

Pattern matching is a useful tool for function creation in Haskell. When used judiciously, it allows for succinct representation of the alternate possible inputs (and the functions which must be performed upon them). Over use of pattern matching can, however, lead to very verbose and difficult to comprehend code compared to a smaller, well formulated function.

With the strong typing of Haskell, typing problems are resolved at compile time. This allows any type related problems to be caught and resolved quickly, without the mess of stepping through code to determine what has gone wrong. Once the types of the data and functions are determined, the construction of those functions becomes much more straightforward and there is less chance for error. Though the programs which are compiled thus tend to be error free, when an error does occur it can be much more difficult to solve as it is generally the result of a misunderstanding in the programmer about the nature of the problem.

As Haskell is a pure functional programming language, programs written in it are described primarily by function composition. This turns out to be a very powerful and expressive technique. Each operation on a set of data can be described by a succinct function, and these functions can then be combined to create larger functions that can thus perform quite complex operations on the data. Functions are considered as "first class citizens" and thus can be (and often are) passed as parameters to other functions. Doing so allows one to create, for example, the generic "filter" function that takes as its arguments a function which (given an element of a list) returns a Boolean value, and a list of such elements. It will then apply whatever function is given to each element of the list and return a new list which contain only those elements for which the given function returned the value True. This allows one to create arbitrary filters by creating specific functions to be applied. The result of this method of software creation is that it lends itself to the natural modularization of programs. Each function is itself a composition of other functions, and so there exists a hierarchy whereby each piece can be examined in isolation as well as in the context of the pieces which use it. Creating large monolithic functions is more difficult in Haskell than creating several small functions and tying them together. It thus rewards the software developer who prefers structured development to quick hacking.

Lazy evaluation is provided by Haskell. Performing evaluations in a lazy fashion allows the generation and use of infinite types. For example, an infinite list of a particular value is generated by creating function which returns a list concatenating together the value and the result of evaluating the function. Thus, the infinite list of the value "1" is represented by "ones = 1:ones". Other functions can take this list as a parameter, and operate upon it consuming only as many ones as necessary (as the function itself will operate in a lazy fashion, it only needs to evaluate the ones function enough to satisfy its own needs).

Literate Programming, as introduced by Knuth [Knu84], is a methodology whereby the documentation of the program and the program itself are combined into a single document. Haskell has native support for the creation of Literate Programs by combining Haskell programs with LaTeX. The result is a single document that describes the program as well as contains it, and can thus be compiled into either a executable program, or interpreted as a Haskell program by an appropriate interpreter, or typeset directly into a format for publication or distribution. This encourages the programmer to document their code clearly and keep the documentation in conformance with the implementation.

Difficulties did occur while using Haskell to manipulate large quantities of data. Given all of the data available to graph, the stack size could quickly run out if the functions manipulating them were not carefully written. The garbage collection facilities are often good, but in the case where a couple of operations were being performed on each piece in a large set of data, it was sometimes difficult to keep the memory usage reasonable. As a result, good performance (in terms of both memory utilization as well as execution speed) can be difficult to achieve if one writes naive function implementations.

3.2.2 GXL

GXL [HWS00] is an acronym for Graph eXchange Language. It is a language which was created to be a standard exchange format for graphs (based on GraX [EKW00], TA, and the graph portions of the PROGRES graph rewriting systems, with ideas taken from RSF, RPA and others [Win01]). GXL is an XML sub-language, and is thus easy to manipulate and parse using standard text-based and XML specific tools.

The choice to use GXL was due to many factors. Though there are a plethora of choices for formats of graph representation, one was required that would allow us to express as formally as possible the semantics of the graphs which were being represented, and to be able to verify that transformations upon those graphs had been performed in a safe, coherent manner. The work of formalization of GXL had already begun by Dr. Kahl and Ms. Wu [Wu04]. Additionally, a format which would be human-readable was thought to have been very beneficial.

Among the benefits of GXL, one of the most pertinent is that there currently exists re-engineering tools which utilize GXL for graph exchange. It is hoped that some of those tools will prove useful in the Reverse Engineering project. As such, the GXL generated by these tools must be valid and tested with a variety of other GXL based tools.

GXL also allows flexibility in terms of the kinds of graphs it can represent. It is possible to represent Control Flow Graphs and Data Flow Graphs using different views of the graphs (as described in [Wu04]).

3.3 Implementation

Given in this section is the core of the libraries used for the representation of the Assembly Language Programs and the methods by which the control flow graphs are generated. First, the representation of the LST file format and how it is parsed is presented. Following that, the representation of the Instruction Architecture of the IBM 1800 computer is given, with associated functions for accessing, translating, and manipulating those instructions. Finally, the NextIA module is given where the control flow is determined from the instructions, both in a purely static method (whereby each instruction will be associated with the possible target destinations on an instruction-by-instruction basis), as well as some dynamic interpretation (using the properties of some of the instructions to determine their effect on other instructions, given the control flow graph as a whole).

3.3.1 Lst

Lst is the module which deals with interacting with a LST file directly. It relies on the *Instruction* module (described in Section 3.3.2) to represent and manipulate *Instructions*, the *MyPrelude* module (described in Section E) for text manipulation functions that are useful but not included in the standard Haskell Prelude, and the *Numeric* module (provided by ghc) for functions to **read** and **show** hexadecimal numbers.

```
module Lst where

import Instruction (Instruction, Address, Object, IndAdd,
                    binaryToInstruction)
import MyPrelude (padString, substr, showsH, dropSuffixes,
                 dropPrefix, rights, readDec', readHex')
import Numeric (readHex, showHex)
```

A LST file contains three different types of lines: headers (usually page headers that are repeated at regular intervals), block comments (used to describe different sections within the assembly source), and *Instruction* lines. *Instruction* lines are the most interesting, so we create a datatype to describe that *LstLine*.

It will have an *Address*, a *Rel* field which describes whether the instruction address and displacement are relative or absolute, a binary version of the instruction, an *StNo* (instruction/data line number in the lst file), the interpreted version of the *Instruction* (which we create), a `label` (possibly empty), an *OpCode*, an instruction format (short or long), a *Tag* (used for index register references), an operand, and an in-line *Comment*.

A *Lst* internally is a list of these *LstLines*, and is thus defined as such. An additional data type, *LstFile*, is created which has a name (that being the name of the source file), as well as a list of entries which are either of type *String* (in the case of a header or block comment), or a *LstLine* (in the case of a valid instruction line).

```
data LstLine = LstLine
    { add :: Address
    , lRel :: String
    , bin :: Object
    , lStno :: Int
    , instr :: Instruction
    , lLabel :: String
    , lOpCode :: String
    , lFormat :: Char
    , lTag :: Char
    , lOperands :: String
    , lComment :: String
    }
type Lst = [LstLine]
data LstFile = LstFile
    { lstname  :: String
    , lstlines :: [Either String LstLine]
    }
```

Conditional operators were created to give the edges of the Gxl graphs an attribute conditions which will be used by other programs to easily determine on what condition the edges will be taken.

```
data CondOp = Eq | Lt | Gr | Tr | Fl deriving (Eq)
```

```
instance Show CondOp where
    show Eq = "=="
    show Lt = "<"
    show Gr = ">"
    show Tr = "True"
    show Fl = "False"
```

With the basic datatypes set, we now create a function to parse an instruction line from the list. Each instruction line contains the address of memory in which the instruction will be placed (as the first hex number readable on the line, after the first superfluous character), then two characters which describe the relative addressing of the instruction, then the actual Instruction (or data) in hex form. This instruction will be no longer than 8 characters, has spaces which need removing, and is in hex. So we use the `readHex` function (which returns a list containing a tuple of the value of the hex digit read and a string representing the rest of the given string after the hex digit). After grabbing these two pieces of information, we convert the binary form of the *Instruction* to an internal representation of the *Instruction*, for later processing. Similar processing is done for reading in the rest of the line.

We also set up a nice way to display a *LstLine* which mimics the way the line was read in, by creating an instance of the *Show* class for *LstLine*.

```
parseLstLine :: String → LstLine
parseLstLine s
    = LstLine a r bi stno (binaryToInstruction bi) lbl op f t oprs c
    where a    = (readHex' · drop 1) s
          r    = (take 2 · drop 6) s
          bi   = (readHex' · filter (≠ ' ') · take 8 · drop 9) s
          stno = readDec' $ substr s 20 4
          lbl  = takeWhile (≠ ' ') $ substr s 28 5
          op   = takeWhile (¬ · flip elem " \r") $ substr s 34 4
          floc = substr s 39 1
          tloc = substr s 40 1
          f    = if (length floc > 0) then head floc else ' '
          t    = if (length tloc > 0) then head tloc else ' '
          oprs = takeWhile (≠ ' ') $ substr s 42 11
          c    = substr s 53 40

instance Show LstLine where
    showsPrec n = showsLstLine

showsLstLine l
    = (flip (++) (show i)
        · padString 20 ' '
        · showsH a
        · (' ':) · showHex b)
    where a = add l
          b = bin l
          i = instr l
```

Next, a function is created to determine whether a particular memory address was intended to be used as an instruction or as data (though in reality it could be used as either or both). It is necessary to do this determination at the *Lst* level, instead of at the Instruction level, as it is only in the *Lst* that we can determine what the author intended that address to be used for. All data lines are read in as *Lst* lines.

```
isData :: LstLine → Bool
isData l = lOpCode l `elem` ["DC", "BSS", "DEC", "DECS"]
```

Now the functions are created which actually read in the *Lst* file. To do so, we need to be able to identify lines that are comments, and handle them separately. Comments are lines that don't start with a valid hex number in both the address position (starting at byte 1, continuing for 5 bytes) and the object position (starting at byte 9, and continuing for 8 bytes). The `isCommentLine` function identifies such lines.

```
isCommentLine :: String → Bool
isCommentLine s = length (readHex addrString) ≠ 1
```

```
                 ∨ length (readHex objString) ≠ 1
    where addrString = substr s 1 5
          objString = substr s 9 8
```

So, we create a function which takes a *String* (the contents of a file) and returns the list of *Lst* lines (represented internally as a variable of type *Lst*). To do that, we turn the *String* read in from the file into a list of lines, remove the comments using our `isCommentLine` function, then parse those lines into *Lst* lines.

```
parseLst :: String → Lst
parseLst = map parseLstLine · filter (¬ · isCommentLine) · lines
```

For those times when the comments are required as well, we can generate a *LstFile* which contains the module name and the list of comments and instructions intermingled (a list of *Either* the string or the instruction representing the line). We can convert this *LstFile* data type to a *Lst* datatype by taking just the *LstLine*s (not the `lstname`), and then returning all of the right field (where the comments would be on the left of the *Either* and the instructions themselves are on the right of the *Either*).

```
parseLstWithComments :: String → String → LstFile
parseLstWithComments modulename lst
    = LstFile n (map (λx → if (isCommentLine x)
                           then Left x
                           else Right (parseLstLine x)) (lines lst))
      where n = dropSuffixes $ dropPrefix $ modulename

lstFileToLst :: LstFile → Lst
lstFileToLst = rights · lstlines
```

3.3.2 Instruction

Now we describe the Instruction set of the IBM1800, and how it is interpreted.

```
module Instruction
where
```

To describe the instruction set, we need to make use of the operations and types found in *Bits* (for bitwise operations for signed and unsigned ints), and the generic *Int* functions and types.

```
import Bits (shiftR, shiftL, (.&.), testBit)
import Char (toUpper)
import Data.Word (Word8, Word16, Word32)
import Int (Int8)
import Numeric (showHex)
import MyPrelude (padString)
```

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).

These can be seen as:

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

D = 5th Bit
F = Format (0 = One-Word, 1 = Two-Word)
T = Tag (00: EA = I + Disp
         01: EA = XR1 + Disp
         02: EA = XR2 + Disp
         03: EA = XR3 + Disp)

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

I = Indirect Addressing (0 = Direct, 1 = Indirect)
B = Branch Out (0 = BSC, 1 = BOSC)
COND = Condition flags interrogated on a BSC or BSI instruction
F = 1, IA = 0:
T = 00: EA = Address
T = 01: EA = Address + XR1
T = 02: EA = Address + XR2
T = 03: EA = Address + XR3

F = 1, IA = 1:
T = 00: EA = Contents (Address)
T = 01: EA = Contents (Address + XR1)
T = 02: EA = Contents (Address + XR2)
T = 03: EA = Contents (Address + XR3)
```

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 a redundant field, `dspl` was added which includes the last 8 bits of the first word of the instruction. It is thus composed of the `indAdd`, `brOut` and `cond` fields. For *MDX* instruction, it is used as the displacement for the *Long* format 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
                    , dspl    :: 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 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)
```

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  -- Index Tag specifying base register
                                       -- (XR0 is used for a base
                                       -- register whose value is
                                       -- always zero)
            deriving (Show,Eq)
```

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
```

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 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
```

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 *Bits* in terms of *Booleans*.

```
data Bit = Zero | One deriving Eq
instance Show Bit where
    show Zero = "0"
    show One  = "1"

boolToBit :: Bool → Bit
boolToBit True = One
boolToBit False = Zero
```

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 = [(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)
                    ]
```

Now, a function is created which will take a 16-bit word (a *Short* instruction, or the upper 16-bits of a *Long* instruction), and return the operation which that instruction represents. It does this by creating a list of values from the opCodeInstruction list where the upper four bits of the word match. If there are more than one, the first is returned. If there are no matches, a *BAD* (invalid) *Op* code is returned.

```
getOp :: Word16 → Op
getOp bs | length opList > 0 = head opList
         | otherwise         = BAD
    where opList = [y | (x,y) ∈ opCodeInstruction, (≡ o) x]
          o = bs .&. 0xF000
```

To display the Instructions in a convenient manner, an instance of the *Show* class is defined for the datatype Instruction. This allows us to say simply "show instruction", and a *String* representation of the instruction will be created.

```
instance Show Instruction where
    show (Short op db t d)
        = (padString 5 ' ' (map toUpper (show op)) ++) ·
          (show db ++) ·
          (show t ++) ·
          (" " ++ ) $
          show d
    show (Long op db t i bo c dp add)
        = (padString 5 ' ' (map toUpper (show op)) ++) ·
          (show db ++) ·
          ((if i ≡ Zero then "L" else "I") ++) ·
          (show t ++) ·
          (" " ++) ·
          (show bo ++) · (" " ++) ·
          (showHex add "" ++) $
          show c
```

To get the 5th bit (getD) and the *Format* bit (getF), we just test the the bit position of the appropriate word (upper or only) in the instruction.

```
getF :: Word16 → Bool
getF bs = testBit bs (15-5)

getD :: Word16 → Bool
getD bs = testBit bs (15-4)
```

To get the *Tag*, we take the instruction, '*AND*' off the relavent two bits, shift the result into place, and convert it to a *Tag* datatype. If a value other than 0, 1, 2, or 3 is found, an exception is thrown as this should never happen. Here for the value 0, we return either a register *XR0* whose value is always zero or *I* depending on the *Format* bit. If it is a short instruction we have to add the instruction address (*I*) with the displacement (*Disp*). If instead it is a long instruction, (that is, the format bit is *One*) then we have to add nothing (*XR0*) to the address given in the instruction.

```
getTag :: Word16 → Tag
getTag bs = case (fromIntegral $ shiftR (bs .&. 0x0300) 8) of
            0x0 → if getF bs then XR0 else I
            0x1 → XR1
            0x2 → XR2
            0x3 → XR3
            _→ error "getTag"
```

We perform a similar operation for obtaining the displacement and the conditions, except that we need not shift for the displacement as the *Disp* is the lower 8 bits.

```
getDisp :: Word16 → Disp
getDisp bs = fromIntegral $ bs .&. 0x00FF

getCond :: Word16 → Cond
getCond bs = fromIntegral $ shiftR (bs .&. 0x003C) 2
```

These functions are used to determine whether a given instruction is a long or a short format instruction, as well as to obtain the indirect bit and branch out bit.

```
isLong :: Instruction → Bool
isLong (Short _ _ _ _) = False
isLong (Long _ _ _ _ _ _ _ _) = True

isInd :: Word16 → IndAdd
isInd bs = if testBit bs (15-8) then One else Zero

isBO :: Word16 → BrOut
isBO bs = if testBit bs (15-9) then One else Zero
```

We need to be able to take either a 16-bit value (in the case of a short instruction) or a 32-bit value (in the case of a long instruction) and convert it into a valid *Instruction* (if such a conversion exists). To do so, we define two functions, one of which generates an *Instruction* from a 16-bit word for the *Short* format, and the other from two 16-bit words for the *Long* format.

```
shortBinaryToInstruction :: Word16 → Instruction
shortBinaryToInstruction bs
    = Short (getOp bs) (boolToBit (getD bs)) (getTag bs) (getDisp bs)

longBinaryToInstruction :: Word16 → Word16 → Instruction
longBinaryToInstruction u l
    = Long (getOp u) (boolToBit (getD u)) (getTag u)
           (isInd u) (isBO u) (getCond u) (getDisp u) l
```

When reading from a LST file a 32-bit word is is read in. If the instruction represented by the 32-bit value is a short instruction, the upper 16 bits will be zero. In that case, we can generate an Instruction by calling `shortBinaryToInstruction` with just the lower bits to generate a short *Instruction*. Otherwise, we call `longBinaryToInstruction` with both the upper and lower bits to generate a long *Instruction*.

```
binaryToInstruction :: Object → Instruction
binaryToInstruction i
```

```
    | i ≤ 0xFFFF = shortBinaryToInstruction (lower i)
    | otherwise = longBinaryToInstruction (upper i) (lower i)
    where upper = fromIntegral · (flip shiftR 16)
          lower = fromIntegral
```

If we are given two 16 bit words (instead of a single 32 bit object), and need to interpret them as an instruction, we can look at the *F* bit of the first word to determine if it is a short or a long instruction. If the *F* bit is one, then we can create a long instruction from both words. Otherwise we create a short instruction from only the first word.

```
wordsToInstruction :: Word16 → Word16 → Instruction
wordsToInstruction w1 w2 | getF w1    = longBinaryToInstruction w1 w2
                         | otherwise = shortBinaryToInstruction w1
```

Now, the reverse operation may need to be performed (given an instruction, generate an object to represent it).

```
instructionToBinary :: Instruction → Object
instructionToBinary i
    | isLong i
        = (shiftL (fromIntegral (opBits
                                  .&. dBit
                                  .&. tagBits
                                  .&. iBit
                                  .&. bBit
                                  .&. cBits))
            16)
          .&. fromIntegral (address i)
    | otherwise = fromIntegral (opBits
                                 .&. dBit
                                 .&. tagBits
                                 .&. fromIntegral (disp i))
   where opBits = head [ x | (x,y) ∈ opCodeInstruction
```

```
                        , (≡ (op i)) y]
     dBit = if (dbit i ≡ One) then 0x0800 else 0
     tagBits = case tag i of
                 XR1 → 0x0100
                 XR2 → 0x0200
                 XR3 → 0x0300
                 _   → 0
     iBit = if (indAdd i ≡ One) then 0x0080 else 0
     bBit = if (brOut  i ≡ One) then 0x0040 else 0
     cBits = fromIntegral (cond i)
```

3.4 Current Users

The library and tools described in this paper are currently being used by both Ontario Power Generation as well as by other members of the CITO project at McMaster.

3.4.1 Ontario Power Generation

After the presentation of the state of the CITO project at McMaster on April 29, 2004, Ontario Power Generation requested that preliminary binaries be given for the Lst2Gxl and gxl2dot programs. They are currently using these programs to provide visualizations of the code to be reviewed in their own efforts to fully understand and re-engineer the IBM1800 and Varian code.

3.4.2 CITO Project Members

There are currently four main users of the library within the CITO project. First, Jun Wu took the initial GXL representation which was a modified version of that generated by HaXml and the initial versions of gxl2dot to create her validation, manipulation, and visualization tools.

Second, Dai Le took the Lst2Gxl code and was successful in porting it from IBM1800 to the Varian architecture. The results of this porting effort are also currently in use at Ontario Power Generation.

Pulak Chowdhury is using the Haskell LST representation code to generate different Emulators of the IBM1800 (both a regular execution as well as a symbolic emulator).

Mark Pavlidis is using the Lst2Xml translator to generate an XML database of the code. This database is then having queries executed against it for further information retrieval and storage (such as which portions of the code are functions, and comments about those portions).

Chapter 4

Control Flow Aspects

In this chapter, the method of generating control flow graphs for the assembly code is provided. The functions created for interpreting the assembly instructions and determining the list of possible next instructions are discussed, along with the graph analysis and visualization tools. A small example of using those tools is then given.

4.1 Control Flow Graph Generation Tools

Control Flow Graph Generation is performed by the Lst2Gxl program, which reads in an assembly LST file, and converts that to an internal representation of a control flow graph, then into a GXL representation of that graph. This is performed by evaluating each instruction from the LST file, and determining the possible next instructions that could be executed based on the instruction type and the operands of that instruction. Currently this is done in a mostly-static fashion.

By mostly-static, what is meant is during the first pass, each instruction is examined as given and for non-branching instructions flow is created to the next instruction (the current instruction's address plus 1 in the case of a short instruction, or plus 2 in the case of a long instruction). For branching instructions, there are additional flows created based on the possible offsets due to the type of instruction (a normal branch will either go to the next instruction, or to some offset, and a compare instruction will go to one of the next three instructions). Indirect branches are marked as such at this point (as there is insufficient information during the first pass to determine potential next instruction addresses). The first pass gives a view of the control flow from the perspective of each instruction in memory remaining as it was first listed, and in isolation from the others. Later passes attempt to fill in more information (probable control flow of indirect branches as a result of subroutine calls or indirect subroutine calls) by examining each instruction in the context of other instructions in the previously generated control flow. The limits to this method are that self-modifying code is not accurately represented (as instructions will be changing as they are executed), there is too much flow generated due to the way that many instructions will interact (some branches will never happen due to the code that precedes and/or follows it), and so there will often be more flow in some cases and less flow in others. In order to represent these changes, a dynamic interpretation of the state of the code must be taken to determine potential new control flow.

When completing the first pass interpretation of the assembly code, the list of possible next instructions was generated by using the IBM 1130/1800 Assembler Manual [IBM65a], the IBM 1800 Functional Characteristics manual [IBM65b], and the IBM 1130 emulator provided by ibm1130.org. This list of next instructions was then combined into an XML format that mimicked as closely as possible the result of using Ontario Power Generations tools to find next instruction candidates. Those were then compared with the generated output, and it was found that they matched very closely, with the exception of three instructions.

In OPG's analysis code, the NOP instruction was not considered, so control flow would break at that point, when in reality control flow would continue on and state would otherwise not be altered by the instruction. The indirect version of the BSI instruction would generate control flow to the address containing the data which was to be used as a destination address, when in fact it should have created flow to the address stored in that data address. Finally, the indirect version of the BSC instruction would give a distance to between the current address and address 12 (which was the result of perl's interpretation of the "c" which was used to indicate that it was not a final destination, but the contents of that field were to be used).

Other than these three differences, the output matched OPG's XML perfectly, indicating that both their interpretation and our interpretations of the provided documentation (over the provided code) matched.

After ensuring that our initial interpretation of the static code was correct, it was found to be feasible to perform some further analysis to attempt to find a more complete interpretation of the possible code paths. This was performed by evaluating the BSI (Branch and Store Instruction Register) instruction. When evaluating each line individually, one could previously only say that the next instruction executed after an indirect branch would be based on the content of some memory location. However, by understanding how the BSI instruction worked, and having access to previous control flow, it was now possible to complete some of these branches.

The BSI instruction is used as a "CALL/RETURN" mechanism. It stands for "Branch and Store Instruction". It works by storing the current value of the Instruction register in the memory address that is referenced as the Effective Address of the BSI instruction (the resulting destination). The instruction that immediately follows that memory address is then executed.

For example, in Figure 4.1, after starting at STRT (Address 0100), a couple of addresses to tables are loaded into the index registers 1 and 2, then a function call is made to FUN2 (from address 0102 to address 0200). At 0200 there is a value of 0 entered at load time, which is then overwritten when the BSI instruction is executed with the value 0103 (the next address after the BSI call in 0102). The first instruction that is executed in the called function FUN2 is the Load in address 0201, which adds an offset (12) to the address stored in Index Register 1 (the address of Table 1) and loads the value in that address. The result is then multiplied by 5, and then a return is made by doing a normal indirect branch (BSC I) to the address stored in FUN2 (Address 0200, containing the value 0103). The next instruction executed is a subtraction of 3 from the value in the accumulator, and the program continues from there. In that way, any time a multiple of five with the value from the 12th offset is desired from the table indexed by Index Register 1, a simple "BSI FUN2" will do so and return to continue execution.

In cases where more data must be passed to the function, the data for the parameters will normally be entered into the memory addresses immediately following the BSI call, as in Figure 4.2. In this case, function FUN3 is called with two parameters, with values of 3 and 1. The address stored in memory location 0400 is thus going to be the address of the first parameter. This is loaded, then the return address is incremented by 1 using the MDX instruction, so that it contains the address of the second parameter. The second parameter is then added to the first, and the return

```
ADDR  LABEL OPCD FT OPERANDS COMMENTS
0100  STRT  LDX  L1 TBL1     LOAD TABLE 1 ADDR INTO INDEX 1
0101        LDX  L2 TBL2     LOAD TABLE 2 ADDR INTO INDEX 2
0102        BSI     FUN2     CALL FUNCTION FUN2
0103        S       3        SUBTRACT 3
...
0200  FUN2  DC      0        PLACEHOLDER FOR RETURN ADDRESS
0201        LD    1 12       LOAD VALUE 12 FROM TABLE 1
0202        M       5        MULTIPLY BY 5
0203        BSC  I  FUN2     RETURN TO CALLING FUNCTION
```

Figure 4.1: An example of source code which uses the BSI instruction

address is incremented once again so it now points to the instruction immediately following the parameters, which is the multiply instruction that will be executed upon the return from the function call.

```
ADDR  LABEL OPCD FT OPERANDS COMMENTS
0300        BSI     FUN3     CALL FUNCTION FUN3
0301        DC      3        FIRST PARAMETER VALUE 3
0302        DC      1        SECOND PARAMETER VALUE 1
0304        M       7        ON RETURN MULTIPLY BY 7
...
0400  FUN3  DC      0        TO STORE RETURN ADDRESS
0401        LD   I  FUN3     LOAD PARAMETER 1
0402        MDX  L  FUN3,1   INCREASE RETURN ADDRESS BY 1
0403        A    I  FUN3     ADD SECOND PARAMETER
0404        MDX  L  FUN3,1   INCREASE RETURN ADDRESS BY 1
0405        BSC  I  FUN3     RETURN TO JUST AFTER DATA
```

Figure 4.2: Use of BSI instruction with two parameters

When creating a control flow graph, previously there would be only the the flow into the function down to the return using the "BSC I" instruction. This is because a pure static analysis means that we do not have the information necessary to determine what is in that memory address. However, once we have generated the initial control flow graph we do have enough information to go through the graph and generate a list of potential return edges (barring any modification of the return address, which we can also check for).

To do this, we go through the entire graph looking for indirect edges, and for each one we find, we look for all of the BSI instructions that reference the same memory location. New edges are then generated to point to the next instruction immediately following each BSI instruction. In this way we can complete most of the function calls in the control flow graph.

4.1.1 NextIA

Here we define a module which generates the list of next possible instructions to execute when given a list of instructions, thus creating the elements used to make a Control Flow Graph.

```
module NextIA where
import Instruction
import Lst
import Numeric (showHex)
import Data.List (nub)
import MyPrelude (showsH, limit, fst3, thrd3)
```

Now we create a datatype to contain a relation between instructions, that being which instruction(s) may be executed following a given instruction. This is represented by a triple which contains the "From" address, the "To" address, and a set of flags to describe whether that execution path is part of a subroutine call (a *Bsi*), is a long instruction that is being executed, or is the result of an indirect reference.

A list of these tuples is brought together in the ***NextList*** data type.

```
type Next = (Address, Address, Flags)
type NextList = [Next]
data Flags = Flags
    { bsi :: Bool
    , long :: Bool
    , indirect :: Bool
    , condition :: CondOp
    } deriving (Eq,Show)
```

The nextIA function uses an *Address* and an instruction (which is stored at that address), and returns a list of possible addresses that could be executed next (and whether or not those are indirect addresses). If the instruction is not a branch (and not bad), then just return either address + 1 (if it is a short instruction), or address + 2 (if it is a long instruction).

This code does not currently handle branches that are partially indexed by Tag (ie, index registers). For now they are ignored as they are not used in the code from OPG.

```
nextIA :: LstLine → [(Address,Bool,CondOp)]
nextIA l
    | (op i ≡ BAD) ∨ isData l = []
    | (op i ≡ BRANCH ∧ dbit i ≡ One)
        = if long
          then if (cond i ≠ 0)
               then [(address i,ind,Tr),(curr+2,False,Fl)]
               else [(address i,ind,Tr)]
          else [(curr+1,False,Fl),(curr+2,False,Tr)]
    | (op i ≡ BRANCH ∧ dbit i ≡ Zero)
        = if long
          then if (cond i ≠ 0)
               then [(curr+2,False,Fl),(1+address i,ind,Tr)]
               else [((if ind then 0 else 1) + address i, ind,Tr)]
          else [(2 + curr + (fromIntegral · disp) i, False,Tr)]
    | op i ≡ MDX = if long
                   then if tag i ≡ XR0
                        then [(curr+2,False,Fl),(curr+3,False,Tr)]
                        else [(curr+2,False,Fl)]
                   else if tag i ≡ I
                        then [(curr + (fromIntegral · disp) i + 1,
                              False,Tr)]
```

```
                         else [(curr + 1, False,Fl),
                               (curr + 2, False,Tr)]
  | op i ≡ WAIT = []
  | op i ≡ CMP  = [(curr+1, False,Gr)
                  ,(curr+2, False,Lt)
                  ,(curr+3, False,Eq)
                  ]
  | otherwise = if long
                then [(curr+2,False,Tr)]
                else [(curr+1,False,Tr)]
  where i = instr l
        curr = add l
        long = isLong i
        ind = if (long ∧ (indAdd i) ≡ One)
              then True
              else False
```

Next we convert each line from the *Lst* file into a list of possible *Next* addresses. This is done by calling the **nextIA** function given, pulling apart the (*Address*, *IndAdd*, *CondOp*) tuples, and putting them back together but this time with the current address at the beginning of each tuple, and flags on the end. This results in a list of (*Address*, *Address*, *Flags*) tuples, where the first address is where we came from, and the second address is where we're going to.

```
lstLineToNextList :: LstLine → NextList
lstLineToNextList l = zip3 as nexts flags
    where (nexts, inds, conds) = (unzip3 · nextIA) l
          as = (add l):as
          flags = zipWith (Flags bsi long) inds conds
          bsi = (((≡BRANCH) · op · instr) l)
                ∧ (((≡ Zero) · dbit · instr) l)
          long = (isLong · instr) l
```

Now, the stuff that is less certain: how to deal with branches that may (or may not) return, etc.

First, generate the initial *NextList*, then go through that to find additional potential edges.

```
generateNextList :: Lst → NextList
generateNextList = concatMap lstLineToNextList
```

The additional *NextList* is going to be that which gives us the additional assumed edges. We can get these by filling in the return branches of subprocedure calls. The reason we can do this is that the Bsi instruction works in such a way that we can determine with a good deal of certainty all of the possible return branches. So, to get this list we take the initial *NextList*, then find all of the indirect edges in that list and find the *Bsi* instructions that would leave the current Program Counter in the destination of that *Next* tuple.

```
additionalNextList :: NextList → NextList
additionalNextList [] = []
additionalNextList nextList
    = concat $ [findBsi n nextList | n ∈ nextList, isInd n]
      where isInd = indirect · thrd3
```

To actually find the *Bsi* instructions that we're looking for, we plough through the graph of preceding next lists looking for *Bsi* instructions with a target of the location that is the *Next* in our indirect *Next* + 1 (as it will be executing the instruction just after the return location). We collect all of these into a list of *Next* tuples with the source of our original *Next* tuple's destination, and a destination of the *Bsi* instruction +1 (if it was a short instruction) or +2 (if it was a long instruction). This lands just after the *Bsi* instruction in either case, to complete the return.

```
findBsi :: Next → NextList → NextList
findBsi n@(curr,next,_) ns = [(curr
                              , pc+(if long pf then 2 else 1)
                              ,(Flags False False False Fl))
                              | (pc,pn,pf) ∈ ns
                             , bsi pf
                             , (¬ · indirect) pf
```

```
                                          , pn ≡ next + 1
                                          ]
```

To actually do our search, we take a *NextList* (which we'll search through), and the node that we're searching from. We then return the list of all nodes that preceded that node, then the list of all nodes that preceded those ones, and so on, and nub away the duplicates at each step. We know we're done when we try to get more preceding nodes, and we don't find any (we've found the limit where another application of the function returns the value we started with).

```
breadthFirst :: NextList → Next → NextList
breadthFirst ns n = limit step start
    where start = [n]
          step xs = nub (xs ++ concat [precNexts n ns | n ∈ xs])
```

The precNexts are the list of *Next* tuples that have as their destination the source of the given *Next* tuple.

```
precNexts :: Next → NextList → NextList
precNexts (c,_,_) ns = [ n | n@(x,y,_) ∈ ns, y ≡ c]
```

Now we can remove the indirect links for which we have filled in the return.

```
removeIndirect :: NextList → NextList → NextList
removeIndirect orig addit = [o | o ∈ orig
                                , (¬ · isInd) o
                                  ∨ (¬ · inAddit addit) o]
    where isInd = indirect · thrd3
          inAddit as (from,_,_) = from ‘elem‘ map fst3 as
```

Next is the function which can be used to add the necessary links for indirect *Bsi* instructions for which we can make a pretty good guess at its final destination. We take the full *Lst* (with data as well as instructions), and a given *NextList*. We then return that *NextList*, concatenated with a list of filled in edges which are the result of going through the given *NextList* and looking at all of the *Bsi* instructions that are *Indirect*, and for which we have found at least one *Data* statement that is referenced by the *Bsi*. If we have found such a *Data* instruction, then we generate a new *Next* tuple with the source of the *Bsi* instruction, and a destination of that *Data* instruction + 1 (as the return value will be stored in that *Data* instruction and the following instruction executed).

```
fillIndirectBSI :: Lst → NextList → NextList
fillIndirectBSI ls ns
    = ns ++ [(nc,1+head (indData ls nn),(Flags False False False Fl))
              | (nc,nn,nf) ∈ ns, bsi nf, indirect nf, foundData ls nn]
    where indData ls a = [fromIntegral (bin l) | l ∈ ls, add l ≡ a]
          foundData ls a = length (indData ls a) > 0
```

4.2 Graph Analysis Tools

Once a control flow graph has been generated, some analysis can be done upon it to gather useful information. To that end, tools were created which would allow the gathering of statistics on the graphs, the generation of subgraphs (to more easily manipulate smaller subsections of the entire graph), and perform some functions relating to the early identification of candidates for functions.

4.3 Visualization

As we were able to generate control flow graphs in an abstract way, and there was a preexisting tool to take a textual representation of a graph and convert it to a graphical representation (performing the layout of nodes and edges, labelling, and storing in various graphical formats), it seemed natural to tie these pieces together. For that purpose, the gxl2dot program was created which took the GXL output from the graph generation tools and generated a dot file which is used by the program "dot" in the graphviz [GKN02] package. There was an existing gxl2dot program included with graphviz, but it had poor support for GXL (none of the examples given by the GXL authors worked) and would experience a Segmentation Fault when dealing with a graph it did not understand.

To remedy this, our own version of gxl2dot was created in Haskell, which allowed us to reuse the GXL representation previously created, and simply translate from this format to the Dot format (using a library created by Dr. Kahl). This then allowed us to do special markup of the GXL to enhance visualization (adding features such as displaying only pertinent attributes, colourising edges and nodes based on arbitrary functions operating on those elements, and other possibilities not yet fully explored).

A Makefile was then created which would allow a user to simply call, for instance, "make BPC-poster.ps", which would execute the pipeline of calling Lst2Gxl on the BPC.lst and storing the resulting GXL as BPC.gxl, then calling gxl2dot to read in the GXL file and store the result as BPC.dot, then calling dot to generate a postscript file, following that, calling "poster" to scale and split the postscript up into pages suitable for printing and recombining to form a poster. Finally, a graphical postscript viewer would be called to display the poster on the screen (and allow for printing). The time to perform all of these operations was approximately 2 minutes for the Boiler Pressure Control code, with the pipeline running on an Athlon XP 2600+ with 1 GiB of RAM. The bulk of the processing time is spent by dot performing the layout of the graph, with the LST to GXL interpretation taking on the order of 3 seconds.

4.4 Example of Control Flow Graph Generation

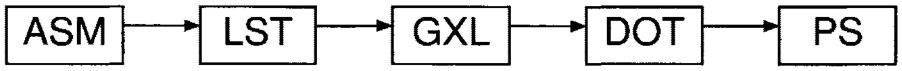

Figure 4.3: Pipeline to generate visualization of Control Flow Graph

In Figure 4.3 the pipeline of formats through which data is transferred to go from the base Assembly (ASM) format to the final PostScript (PS) format.

The sequence of instructions starts as an Assembly file (including comments, as shown in Figure 4.4), and is then transformed into a LST file by the IBM 1800

```
*ASM SMALL REV01-1106 REV01          DATE 04-04-21   SMALL
      ABS
COREA EQU     /3400       START CORE ADDRESS
      ORG     COREA-1
      LDX    3 0          SET COEF TABLE INDEX ZERO
      CMP    1 28         IS U(L) MORE THAN 0.1
      MDX     OPT3        YES
      NOP                 NO
      MDX     LSDV        USE ZERO FOR INDEX
OPT3  CMP    1 30         U(L) MORE THAN 0.8
      MDX     OPT4        YES
      NOP                 NO
      LDX    3 4
      MDX     LSDV
OPT4  CMP    1 32         U(L) MORE THAN 0.9
      MDX    3 4          YES, REQUIRE 12 FOR INDEX
      NOP
      MDX    3 8          NO, USE 8 FOR INDEX
LSDV  LD     1 21
END   EQU     *      THIS DC REQUIRED TO FORCE
*                         OBJECT OUTPUT
      END     END
```

Figure 4.4: A sample of IBM 1800 Assembly

assembler (which takes the source and translates each instruction and data line into its equivalent hexadecimal representation, as well as its ultimate location in memory, as shown in Figure 4.5).

The LST file is then translated by Lst2Gxl to a Graph eXchange Language (GXL) representation which is generated by examining each instruction and determining the possible next instructions which will be executed after the current instruction. This forms a graph of possible control flow paths. The GXL representation if the code sample is given in Figure 4.6.

The GXL graph is then translated into the dot language (as specified by the graphviz dot documentation). Labelling of nodes and edges, and colourization is done during this step, with hints placed in the GXL graph. The result of this can be seen in Figure 4.7.

```
33ff 6300       6 |       LDX   3 0       SET COEF TABLE INDEX ZERO
3400 b11c       7 |       CMP   1 28      IS U(L) MORE THAN 0.1
3401 7002       8 |       MDX     OPT3    YES
3402 1000       9 |       NOP             NO
3403 7009      10 |       MDX     LSDV    USE ZERO FOR INDEX
3404 b11e      11 | OPT3  CMP   1 30      U(L) MORE THAN 0.8
3405 7003      12 |       MDX     OPT4    YES
3406 1000      13 |       NOP             NO
3407 6304      14 |       LDX   3 4
3408 7004      15 |       MDX     LSDV
3409 b120      16 | OPT4  CMP   1 32      U(L) MORE THAN 0.9
340a 7304      17 |       MDX   3 4       YES, REQUIRE 12 FOR INDEX
340b 1000      18 |       NOP
340c 7308      19 |       MDX   3 8       NO, USE 8 FOR INDEX
340d c115      20 | LSDV  LD    1 21
```

Figure 4.5: A sample of IBM 1800 LST file, generated from the assembly

```
<gxl><graph id="CodeSequence" >
  <node id="c33ff"><attr name="opcode"><string>LDX</string></attr></node>
  <edge from="c33ff" to="c3400"/>
  <node id="c3400"><attr name="opcode"><string>CMP</string></attr></node>
  <edge from="c3400" to="c3401"/>
  <edge from="c3400" to="c3402"/>
  <edge from="c3400" to="c3403"/>
  ...
  <node id="c3404"><attr name="label"><string>OPT3</string></attr>
                <attr name="opcode"><string>CMP</string></attr></node>
   ...
</graph></gxl>
```

Figure 4.6: A sample from the GXL resulting from processing the LST file

The DOT file is then read by the dot package to generate a postscript file, which is then either printed or displayed on the screen. This graphical representation can be seen in Figure 4.8.

```
digraph cCodeSequence {
 c33ff [label="33ff\nopcode = LDX"];
 c33ff -> c3400 [label=""];
 c3400 [label="3400\nopcode = CMP"];
 c3400 -> c3401 [label=""];
 c3400 -> c3402 [label=""];
 c3400 -> c3403 [label=""];
 c3401 [label="3401\nopcode = MDX"];
 c3401 -> c3404 [label=""];
 ...
}
```

Figure 4.7: A sample of Dot, generated from the GXL

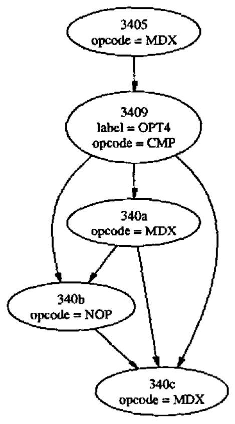

Figure 4.8: A sample of IBM 1800 LST file, generated from the assembly

Chapter 5

Other Tools

While creating the tools for interpreting and visualizing the assembly code, it was found that some additional tools would also be useful. There was a request for some statistics about the graphs, and so a small program was created which allows the user to quickly obtain a summary of information about the graph (numbers of nodes, edges, length of back edges, etc). The desire to be able to obtain a subgraph was also expressed, and so a tool was created for that purpose. Due to having existing modules to adequately describe the datatypes involved in these programs, the generation of the programs themselves was very straightforward and resulted in compact code that expressed the solutions well (the statistics functions being rather self explanatory, and the subgraph functions reading as they were expressed mathematically and thus easing verification).

5.1 Statistics

The GxlStats program (given in Appendix C.1) reads in a GXL file and generates some statistics for that graph. Currently the list of statistics generated are: Number of nodes, number of start nodes (those having no edges leading into them), number of end nodes (those having no edges leaving them), number of backedges (where the address of the current node is greater than the address of the next node), and the average length of backedges (that is, how far on average a jump back in the code is from the current instruction).

An example run of the GxlStats program is given in Figure 5.1.

```
\$ GxlStats bpc.gxl
Reading from bpc.gxl
Computing Statistics
Number of nodes: 1046
Number of start nodes: 60
Number of end nodes: 48
Number of back edges: 114
Average length of back edges: -2055.9736
```

Figure 5.1: An example run of the GxlStats program

5.2 Subgraph

The subgraph functions (Appendix C.2) take a GXL graph, a start node, and an end node. The GXL graph is then converted to a Data.Graph (the format and use of which is described in [KL95]), and in doing so the edges leading into the start node and out of the end node are removed. The vertices of the desired subgraph then become the intersection of the reachable vertices from the start node in the graph, with the reachable vertices from the end node in transpose of the graph (where each edge is in the opposite direction from the original graph).

In this way, a graph such as that in Figure 5.2 can have the subgraph taken from nodes 3405 to 340c to result in the subgraph in Figure 5.3.

5.3 Early Function Identification

Once we have the control flow graphs, it is useful to attempt automatic detection of function blocks. This will aid the user of the Reverse Engineering tools in identifying sections of the code which can be used to generate tables to describe functionality or to generate higher level views of the code with the functions being represented by

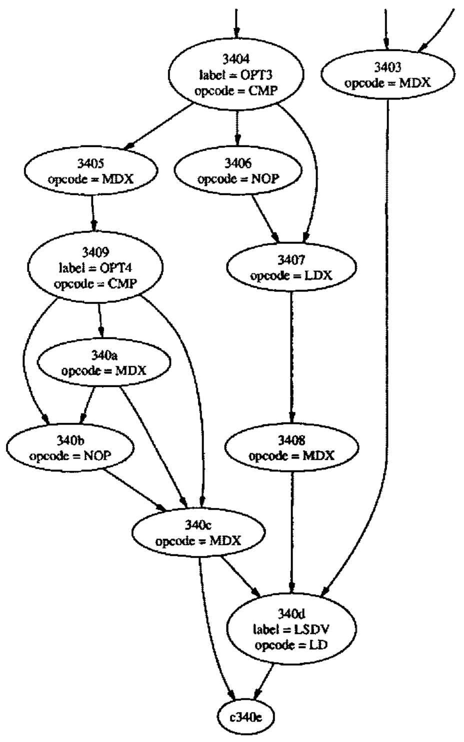

Figure 5.2: A graph from which a subgraph will be taken

single blocks. These representations can be verified to be valid homomorphisms of the code by the tools generated by Jun Wu [Wu04].

An initial attempt at early identification of function blocks used the idea of identifying Single Entry, Single Exit (SESE) regions of the code (based on the work of Johnson et al. [JPP94]). The idea was to take a control flow graph with defined start and end nodes and create from it a grouping of nodes based on each group containing only one edge entering and one edge leaving the group. This idea can be modified to instead consider the SESE regions based on nodes instead of edges, where the roles of nodes and edges are reversed such that each region is denoted by its start node and end node. This view is more natural and likely more useful when using SESE regions in the search for functions as each function call will generate an edge to the first node in the function, which itself might branch to several other nodes within the function. The last node in a function will be the return node, with again many edges leaving

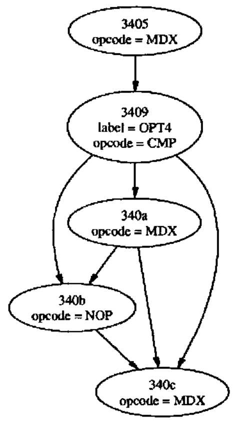

Figure 5.3: A subgraph

the return node and entering the nodes just after those calling the function. A SESE region generated based on edges would not create a useful grouping for this function, whereas an SESE region generated based on nodes would find the single start and exit nodes. For instance, the graph in Figure 5.4 is a grouping of the function from 4000 to 4004, based on a SESE region denoted by the start node 4000 and the end node 4004, being called from 1000, 2000, and 3000 and returning to 1001, 2001, and 3001. Attempting a SESE grouping based on entry and exit edges would only allow a small group around 4001 and 4002.

5.4 Other Contributions

Beyond the assembly representation and graph generation tools, other contributions were made to the project relating to graph visualization and infrastructure, as well as to outside projects such as the IBM1130.org assembler.

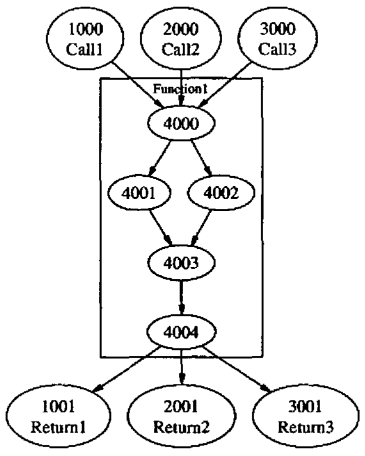

Figure 5.4: Single Entry Single Exit region with a node perspective

5.4.1 IBM1130.org

During the development of the assembly representation libraries and tools, it was quite helpful to be able to use an emulator from a very similar machine, the IBM 1130. There is a group of enthusiasts who maintain this emulator as well as some associated software (Operating Systems, Disk Managers, etc.) for what appears to be nostalgia reasons. The assembler and emulator are available from `http://www.ibm1130.org`. As the IBM 1130 and the IBM 1800 were so similar, it was helpful to generate LST files using the IBM 1130 assembler (before we had complete LST files as generated by Ontario Power Generation), as well as observing how portions of the code would work under their emulator. The 1130 was, however, missing a few of the instructions that were present in the 1800. Namely, the CMP, DCM, and DECS instructions were added. These have now been submitted back to ibm1130.org for inclusion in future releases.

5.4.2 Infrastructure

Additional contributions were made towards the project through source control organization and build infrastructure. Within the source tree, there were a few revisions of placement of source and binary files. These were relatively cheap to make as we were using Subversion as opposed to CVS. As we are using Subversion, one can move directory structures and still maintain history on the contents of the directories automatically. Doing a similar operation in CVS would not have allowed this.

The current source organization for the tools has settled on a structure approximating the Filesystem Hierarchy Standard (FHS) [QRY04] with `bin`, `man`, and `src` directories. Within the src directory, there is a directory for each program, as well as a `common` directory which holds the source modules used by several programs.

The `bin` directory holds compiled binaries of the tools, and the `man` directory holds man pages for each of the tools, written using the roff type-setting system so as to be readable by the common man program found on most UNIX-like operating systems.

Other infrastructure which was contributed was the Makefile used by several components for generating binaries for the Haskell programs.

A program to change passwords was also generated using the Web Authoring System Haskell (WASH) [Thi03]. This allowed the generation of a small Common Gateway Interface (CGI) program which does input verification and notification of bad input. This CGI was placed on the server running apache and protected with the same password file that was being modified (that used initially only for subversion repository access). Doing so allows the program to read the username (which was verified with a password) from the environment variable provided by Apache once it has performed the authentication.

Chapter 6

Conclusion & Outlook

The use of a pure functional programming language has resulted in a suite of tools and modules which are compact, succinct, and interact well with one another. The virtues of Haskell in particular have aided in the development process in a way that other programming paradigms would not have done. The code that has been generated is comprehensible enough that an undergraduate student was able to, in a very limited time, understand and port them to an additional architecture.

The appropriate use of abstraction resulted in smaller, more easily understood code. Functional programming lends itself well to the benefits of abstraction, and the mechanisms for both abstraction and function composition provided by Haskell are powerful and easy to use.

The tools created interact well with each other as well as with tools developed outside this project. They are flexible and as such, can be (and have been) improved, modified, and portions incorporated into other tools quickly and easily.

Though the code written is in general easily understood, the nature of the terse functions can result in convoluted and obtuse code that requires sufficient prose to explain. Haskell's built-in use of Literate Programming gives the opportunity to generate the documentation at the same time and in the same location as the implementation, but it is still the responsibility of the software designer to make use of that facility. This capability was used in the literate code presented in Section 3.3.1, Section 3.3.2, and others. This allowed the code produced to come close to the ideal of "the design is the code", where the design document is itself directly executable.

Though documented explanations of the code have been done, the clarity of the code can also be improved by making more extended use of the abstraction capabilities of the language. Refactoring the code to make use of more of the built-in functions of the Prelude and libraries associated with the compiler would also result in code which is more clear and understandable to the reader.

In summary, this work has explored the use of a pure functional programming language in the context of a real-life Software Engineering project to create useful data representations and tools that act upon those representations in a coherent manner. The benefits and costs of using such a language were explored. As a result, it is my opinion that such a language is an effective tool in good Software Engineering practice.

Appendix A

Lst2Gxl

Lst2Gxl is an application which reads in a LST file which contains the original ASM assembly file, plus the result of assembling (i.e., the calculated address in memory as well as the instruction itself assembled into one or two 16-bit words). Lst2Gxl is composed of five main modules, represented in Figure A.1. The following is a description of each of those modules in detail.

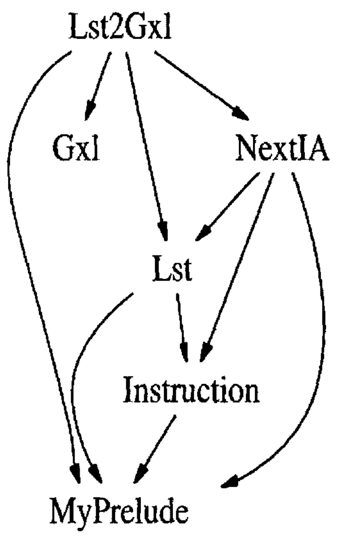

Figure A.1: Overview of Lst2Gxl, and its module dependencies

A.1 GxlGraph

This is the abstraction of GXL Graphs, and access functions for those graphs.

Import & Export List

The following is the import and export list for the *GxlGraph* abstraction layer.

```
module GxlGraph ( GxlGraph,
                  NodeId,
                  GxlNode,
                  GxlEdge,
                  GxlAttr,
                  EdgePath,
                  NodePath,
                  GxlGxl,
                  makeGxl,
                  makeGraph,
                  makeNodeId,
                  makeNode,
                  makeEdge,
                  makeBoolAttr,
                  makeStringAttr,
                  addNode,
                  addEdge,
                  addNodes,
                  addEdges,
                  addNodeAttrib,
                  addEdgeAttrib,
                  edgesFrom,
                  edgeStartNode,
                  edgeEndNode,
                  isFrom,
                  isTo
```

```
                   )
where

import qualified Gxl
import Text.XML.HaXml.OneOfN (OneOf3(..), OneOf10(..))
import Text.XML.HaXml.Xml2Haskell (Defaultable(..))
import Char (isAlpha)
import List (find)
import MyPrelude (unEntity)
```

Exported Type Abstractions

Now, we generate the type abstractions to be exported. In this way, the programs using this module will not know (or care) exactly what a *GxlGraph* or its constituent parts are. They will only be given the functions necessary to generate and modify these graphs.

Some of these abstractions will automatically have their instances generated for the Equality (*Eq*) and Show classes. Others will have one ore both of these instances created manually so as to provide additional control of the output or what is meant by equality.

```
newtype GxlGraph = GxlGraph Gxl.Graph  deriving (Eq,Show)
newtype NodeId   = NodeId   String     deriving (Eq)
newtype GxlNode  = GxlNode  Gxl.Node   deriving (Eq,Show)
newtype GxlEdge  = GxlEdge  Gxl.Edge   deriving (Eq)
newtype GxlAttr  = GxlAttr  Gxl.Attr   deriving (Eq,Show)

type GxlGxl   = Gxl.Gxl
type EdgePath = [GxlEdge]
type NodePath = [GxlNode]

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

Instances of Show

Some better *Show* instances are now generated, so now one can do something like:

```
show $ makeEdge (makeNodeId "Test" "345d") (makeNodeId "Test" "345e")
```

and get back:

```
"Test-345d -> Test-345e"
```

instead of the large mess that would have resulted had the derived *Show* instances been used.

```
instance Show GxlEdge where
    showsPrec _ e = (shows (edgeFrom e))
                    · (" -> " ++)
                    · (shows (edgeTo e))

instance Show NodeId where
    showsPrec _ (NodeId s) = (s ++)
```

Generation Functions

The following functions are used to generate graphs, nodes, edges, and attributes.

First, a function to make a default graph with no nodes, edges, or rels. This function will take the name of the graph (which will have a *ValidIdRef* restriction placed upon it) and generate an otherwise empty graph with some of the default attributes (that edges have ids, that it is not a hypergraph, and that the edges are directed) set.

```
makeGraph :: String → GxlGraph
makeGraph name
    = GxlGraph (Gxl.Graph
                 (Gxl.Graph_Attrs
                  (mkValidIdRef name)
                  Nothing
```

```
                    (Default Gxl.Graph_edgeids_true)
                    (Default Gxl.Graph_hypergraph_false)
                    (Default Gxl.Graph_edgemode_defaultdirected)
                   )
                   Nothing
                   []
                   []
                  )
```

Next are functions to generate a *GxlNode* using a *NodeId* (which also must be generated using the `makeNodeId` function), or a *GxlEdge* (which requires the *NodeId* of the start node and the end node).

```
makeNode :: NodeId → GxlNode
makeNode (NodeId nid) = GxlNode ( Gxl.Node
                                  (Gxl.Node_Attrs nid)
                                  Nothing
                                  []
                                  []
                                )

makeEdge :: NodeId → NodeId → GxlEdge
makeEdge (NodeId from) (NodeId to) = GxlEdge ( Gxl.Edge
                                               ( Gxl.Edge_Attrs
                                                 Nothing
                                                 from
                                                 to
                                                 Nothing
                                                 Nothing
                                                 Nothing
                                               )
                                               Nothing
                                               []
                                               []
```

```
                                                            )
```

The following is a function to take a graph name and a unique ID (we currently don't check uniqueness), and from that generate a *NodeId*. A *NodeId* must be a valid IDREF, so it must start with one of a letter, an '_' or a ':' and then be followed by zero or more letters, digits, combining characters, extenders, or elements of ".-_:" (we currently only ensure that it begins with a letter).

```
makeNodeId :: String → String → NodeId
makeNodeId graphname id = NodeId (prefix ++ "-" ++ id)
    where prefix = mkValidIdRef graphname
```

Make a valid *IdRef* out of a string (make sure it is prefixed by a letter, an underscore or a colon).

```
mkValidIdRef :: String → String
mkValidIdRef n | length n > 0 = if (isAlpha hn ∨ hn ‘elem‘ ":_")
                                 then n
                                 else '_':n
               | otherwise     = "_"
    where hn = head n
```

The following function takes a *GxlGraph* and makes a full *Gxl* out of it (which can then be passed to other functions which understand more about the composition of a *Gxl*). The given graph is the only graph created in the *Gxl*.

```
makeGxl :: GxlGraph → GxlGxl
makeGxl (GxlGraph g) = Gxl.Gxl (Gxl.Gxl_Attrs (Default "")) (g:[])
```

Here are the functions which generate either a *Boolean* attribute or a *String* attribute. These just contain the name and the value (in the case of string attribute, the value has all of the <>& characters translated into their entity reference equivalents).

```
makeBoolAttr :: String → Bool → GxlAttr
makeBoolAttr name value
    = GxlAttr (Gxl.Attr (Gxl.Attr_Attrs Nothing name Nothing)
               Nothing [] (TwoOf10 (Gxl.Bool (show value))))

makeStringAttr :: String → String → GxlAttr
makeStringAttr name value
    = GxlAttr (Gxl.Attr (Gxl.Attr_Attrs Nothing name Nothing)
               Nothing [] ((FiveOf10 · Gxl.GxlString) v))
    where v = unEntity value
```

Modification Functions

The following functions modify *GxlGraphs* by adding nodes and edges to them. They also modify *GxlNodes* and *GxlEdges* by adding attributes to them.

```
addNode :: GxlGraph → GxlNode → GxlGraph
addNode (GxlGraph (Gxl.Graph a b c ners)) (GxlNode n)
    = GxlGraph (Gxl.Graph a b c ((OneOf3 n):ners))

addNodes :: [GxlNode] → GxlGraph → GxlGraph
addNodes nodes graph = foldl addNode graph nodes

addNodeAttrib :: GxlAttr → GxlNode → GxlNode
addNodeAttrib (GxlAttr a) (GxlNode (Gxl.Node nattr typ as graphs))
    = GxlNode (Gxl.Node nattr typ (a:as) graphs)

addEdge :: GxlGraph → GxlEdge → GxlGraph
addEdge (GxlGraph (Gxl.Graph a b c ners)) (GxlEdge e)
    = GxlGraph (Gxl.Graph a b c ((TwoOf3 e):ners))

addEdges :: [GxlEdge] → GxlGraph → GxlGraph
addEdges edges graph = foldl addEdge graph edges

addEdgeAttrib :: GxlAttr → GxlEdge → GxlEdge
addEdgeAttrib (GxlAttr a) (GxlEdge (Gxl.Edge eattr typ as graphs))
    = GxlEdge (Gxl.Edge eattr typ (a:as) graphs)
```

Query Functions

The following functions allow the user to make queries about the *GxlGraphs* that they have. The kinds of queries that can be performed are:

- Given a graph and a node, return all edges that start at that node
- Return all of the nodes or all of the edges in a graph
- Determine if an *Edge* originates, or terminates at a *Node*
- Given a graph and an edge, return the node that is at the start or end of that edge.

```
edgesFrom :: GxlGraph → GxlNode → [GxlEdge]
edgesFrom g n = [e | e ∈ edges g, e ‘isFrom‘ n]

nodes :: GxlGraph → [GxlNode]
nodes (GxlGraph (Gxl.Graph _ _ _ ners)) = nodesFromNERS ners

edges :: GxlGraph → [GxlEdge]
edges (GxlGraph (Gxl.Graph _ _ _ ners)) = edgesFromNERS ners

edgesFromNERS :: [NER] → [GxlEdge]
edgesFromNERS []                 = []
edgesFromNERS ((TwoOf3 e):ners) = (GxlEdge e) : edgesFromNERS ners
edgesFromNERS (_:ners)          =               edgesFromNERS ners

nodesFromNERS :: [NER] → [GxlNode]
nodesFromNERS []                 = []
nodesFromNERS ((OneOf3 n):ners) = (GxlNode n) : nodesFromNERS ners
nodesFromNERS (_:ners)          =               nodesFromNERS ners

isFrom :: GxlEdge → GxlNode → Bool
isFrom e n = nodeId n ≡ edgeFrom e
```

```
isTo :: GxlEdge → GxlNode → Bool
isTo e n = nodeId n ≡ edgeTo e

edgeStartNode :: GxlGraph → GxlEdge → Maybe GxlNode
edgeStartNode g e = find ((≡) (edgeFrom e) · nodeId) (nodes g)

edgeEndNode :: GxlGraph → GxlEdge → Maybe GxlNode
edgeEndNode g e = find ((≡) (edgeTo e) · nodeId) (nodes g)

nodeId   :: GxlNode → NodeId
nodeId   (GxlNode (Gxl.Node att _ _ _)) = NodeId (Gxl.nodeId   att)
edgeFrom :: GxlEdge → NodeId
edgeFrom (GxlEdge (Gxl.Edge att _ _ _)) = NodeId (Gxl.edgeFrom att)
edgeTo   :: GxlEdge → NodeId
edgeTo   (GxlEdge (Gxl.Edge att _ _ _)) = NodeId (Gxl.edgeTo   att)
```

A.2 Lst2Gxl

Tool to convert a *Lst* file to *Gxl*.

```
module Main where
import NextIA (lstLineToNextList, additionalNextList,
               generateNextList, fillIndirectBSI,
               Next(..), NextList(..), Flags(indirect, condition))
import Lst (Lst(..), LstLine(..), LstFile(..), CondOp(..),
            parseLstWithComments, isData, lstFileToLst)
import Instruction (Instruction(..), Op(..), isLong)
import GxlGraph
import Text.XML.HaXml.Xml2Haskell (fWriteXml, Defaultable(..))
import System (getArgs)
import Numeric (showHex)
import Char (isAlpha)
import MyPrelude (showH, replace)
```

```
main :: IO ()
main = do [infile,outfile] ∈ getArgs
          lstFile ∈ readFile infile
          let gxl = lstToGxl $ parseLstWithComments infile lstFile
          fWriteXml outfile gxl
          putStrLn "Done."
```

Now we convert the *Lst* to *Gxl* by converting the *Lst* to a single graph.

```
lstToGxl :: LstFile → GxlGxl
lstToGxl l = makeGxl (lstToGraph l)
```

Converting a lst to a graph file involves making a graph where the defaults are edgeids, hypergraph, and edges are directed. Then, go through the lst lines, converting only those that are instructions (which translates to those that are not data) to *Node*s, *Edge*s, and *Rel*s (*NER*s).

```
lstToGraph :: LstFile → GxlGraph
lstToGraph l = (addEdges (newedges ++ edges)
                 · addNodes nodes
                 · makeGraph
                ) name
    where name = lstname l
          lst = lstFileToLst l
          onlyInstr = filter (¬ · isData) lst
          nodes = map (lstLineToNode name) onlyInstr
          edges = concatMap (lstLineToEdges name lst) onlyInstr
          newedges = (nextListToEdges name
                      · additionalNextList
                      · generateNextList) lst
```

Now we take the *LstLine*, and generate a node that will represent it. To do so, we give the *Node* an id which is the calculated address of that instruction in memory. We also add whatever attributes which can be determined by the list line and which we deem necessary to the node by using the `llToNodeAttrs` function (defined below).

```
lstLineToNode :: String → LstLine → GxlNode
lstLineToNode name ll
    = (llToNodeAttrs ll
        · makeNode
        · makeNodeId name
        · showH
        · add) ll
```

The *Lst* line is converted to the the list of edges associated with that node by first getting the *NextList*, and then converting that *NextList* to a list of *Edges*.

```
lstLineToEdges :: String → Lst → LstLine → [GxlEdge]
lstLineToEdges name l line = ((nextListToEdges name)
                              · fillIndirectBSI l
                              · lstLineToNextList) line
```

To convert a *NextList* to a list of *Edges*, we just map the `nextToEdge` function onto every *Next* tuple in the list.

```
nextListToEdges :: String → NextList → [GxlEdge]
nextListToEdges name = map (nextToEdge name)
```

The `nextToEdge` function takes a *Next* tuple (containing the current address, the potential next address, and a flag to indicate whether this is a direct or an indirect reference), and created an *Edge* from the current address to the next address. If this is an indirect reference, then we create an attribute for that *Edge* with the name of "indirect" and the value of "True". If this is a backedge (the next address is less than the current address), then we add an attribute with the name of "backedge" and the value of "True".

When generating an *Edge*, there's two different kinds of attributes as defined by GXL, the predefined attributes of the edge itself (`edgeId`, `edgeFrom`, `edgeTo`, `edgeFromOrder`, `edgeToOrder`, and `edgeIsDirected`), as well as our generated additional (generic) attributes that can be *Strings*, *Bools*, etc. In this case, we potentially add two of these *Bool* attributes for "indirect" edges or "backedge".

```
nextToEdge :: String → Next → GxlEdge
nextToEdge name (curr,next,flag)
    = (addInd · addBack · addCond) $ makeEdge
      (makeNodeId name (showH curr))
      (makeNodeId name (showH next))
    where addInd = if indirect flag
                   then (addEdgeAttrib
                         (makeBoolAttr "indirect" True))
                   else id
          addBack = if next < curr
                    then (addEdgeAttrib
                          (makeBoolAttr "backedge" True))
                    else id
          addCond = addEdgeAttrib (makeStringAttr
                                    "condition"
                                    (show $ condition flag)
                                   )
```

The current attributes we glean from the *LstLine* are the label, format, tag, operands, and comments (if any of the above) that are associated with this address, as well the opcode. We can colour the *Node* using the `colourNode` function to create a "color" attribute.

```
llToNodeAttrs :: LstLine → GxlNode → GxlNode
llToNodeAttrs ll = nonEmpty "label" (lLabel ll) ·
                   nonEmpty "address" (showH (add ll)) ·
                   addSAttr "opcode" (lOpCode ll) ·
                   addSAttr "binary" (showHex (bin ll) "") ·
                   nonEmpty "rel" (lRel ll) ·
                   nonEmpty "stno" (show (lStno ll)) ·
                   nonEmpty "format" ([lFormat ll]) ·
                   nonEmpty "tag" ([lTag ll]) ·
                   nonEmpty "operands" (lOperands ll) ·
```
-- bsi "operands" (lOperands ll) .

```
                   nonEmpty "comment" (lComment ll) ·
                   colourNode ll
    where nonEmpty name value = (if (value ≡ "" ∨ value ≡ " ")
                                  then id
                                  else (addSAttr name value)
                                 )
          addSAttr n v = addNodeAttrib (makeStringAttr n v)
          bsi name value = (if (lOpCode ll ≡ "BSI")
                             then (nonEmpty "operands" (lOperands ll))
                             else id)
```

To colour a node, we look at properties of that node, and if it matches our criteria, then we assign a colour to it. Otherwise, we return an empty *Attr* list (meaning an unspecified colour).

```
colourNode :: LstLine → GxlNode → GxlNode
colourNode ll | op instruction ≡ MDX ∧ isLong instruction
                   = addNodeAttrib (makeStringAttr "color" "blue")
              | otherwise = id
    where instruction = instr ll
```

Appendix B

Lst2Xml

Once the Lst2Gxl tool was completed, it was relatively straightforward to generate a new tool to deal with the XML format required for importing into the project's XML database. A new module, Xml1800, was created to hold the type definitions of the XML format, and a Lst2Xml program was then written incorporating that module, as well as the required modules shared with Lst2Gxl.

B.1 Xml1800

```
module Xml1800 where

import MyPrelude(unEntity)

data LstXml = LstXml Module
data Module = Module Name [Line]
type Name = String
data Line = Header String
          | BlockComment StNo String
          | AdditionalComment StNo String String User Modified
          | Instruct Addr Rel Object StNo Label OpCd Format
                      Tag Operands Remark [From] [To]
type StNo = String
type User = String
type Modified = String
type Addr = String
type Rel = String
type Object = String
type Label = String
type OpCd = String
type Format = String
type Tag = String
type Operands = String
type Remark = String
type From = String
type To = String

instance Show LstXml where
    showsPrec _ = showsLstXml

showsLstXml (LstXml m) = ("<?xml version=\"1.0\"?>\n" ++) ·
                         ("<lst>" ++) ·
                         showsModule m ·
                         ("</lst>" ++)

showsModule (Module name ls) = ("<module>\n" ++) ·
                               showsElement "name" name · ('\n':) ·
                               showsLines ls ·
                               ("</module>" ++)

showsLine (Header str) = showsElement "header" str
showsLine (BlockComment stno str) = ("<blockComment>" ++) ·
                                    showsElement "stno" stno ·
                                    showsElement "string" str ·
                                    ("</blockComment>" ++)
showsLine (AdditionalComment stno str1 str2 u m) =
    ("<additionalComment>" ++) ·
    showsElement "stno" stno ·
    showsElement "short" str1 ·
    showsElement "long" str2 ·
    showsElement "user" u ·
    showsElement "modified" m ·
    ("</additionalComment>" ++)
showsLine (Instruct a rel o s l oc f t op r fs ts) =
    ("<instruction>" ++) ·
    showsElement "addr" a ·
    showsElement "rel"  rel ·
    showsElement "object" o ·
    showsElement "stno" s ·
    showsElement "label" l ·
    showsElement "opcd" oc ·
    showsElement "format" f ·
    showsElement "tag" t ·
    showsElement "operands" op ·
    showsElement "remark" r ·
    showsElements "from" fs ·
    showsElements "to" ts ·
    ("</instruction>" ++)

showsLines [] = id
showsLines (l:ls) = ("<line>" ++)
                  · showsLine l
                  · ("</line>\n" ++)
                  · showsLines ls

showsElement _ "" = id
showsElement name value = ('<' :) · (name ++) · ('>':) ·
                          (unEntity value ++) ·
                          ("</" ++) · (name ++) · ('>':)
```

```
showsElements _ [] = id
showsElements n (e:es) = showsElement n e · showsElements n es
```

B.2 Lst2Xml

Tool to convert a Lst file to an Xml version of the list

```
module Main where
import NextIA (lstLineToNextList, additionalNextList,
               generateNextList, removeIndirect,
               Next(..), NextList(..), Flags(indirect))
import Lst (Lst(..), LstLine(..), LstFile(..),
            parseLstWithComments, lstFileToLst)
import Xml1800
import System (getArgs)
import Numeric (showHex)
import MyPrelude (showH,substr,readDec')

main :: IO ()
main = do [infile,outfile] ∈ getArgs
          myLstFile ∈ readFile infile
          let myLst = parseLstWithComments infile myLstFile
          writeFile outfile · show · lstToXml $ myLst
          putStrLn "Done."

lstToXml :: LstFile → LstXml
lstToXml l = LstXml $ Module (lstname l) $ lstToLines l

lstToLines :: LstFile → [Line]
lstToLines l = map (lstLineToXmlLine nextList) $ lstlines l
    where nextList = removeIndirect origNext additional ++ additional
          additional = additionalNextList origNext
          origNext = (generateNextList · lstFileToLst) l
```

```
lstLineToXmlLine _ (Left comment)
    | isBC comment = genBlockComment comment
    | otherwise    = Header comment
    where isBC s = substr s 28 1 ≡ "*"
          genBlockComment s
              = BlockComment (show $ readDec' $ substr s 20 4)
                             (substr s 28 100)
lstLineToXmlLine nextList (Right line)
    = Instruct a rel o s l oc f t op r fs ts
    where a = (showH · add) line
          rel = lRel line
          o = (flip showHex "" · bin) line
          s = (show · lStno) line
          l = lLabel line
          oc = lOpCode line
          f = [lFormat line]
          t = [lTag line]
          op = lOperands line
          r = lComment line
          fs = [showH x | (x,y,_) ∈ nextList, y ≡ add line]
          ts = [showH y | (x,y,_) ∈ nextList, x ≡ add line]
```

Appendix C

General Gxl Tools

In this appendix, the literate Haskell source for the auxiliary GXL tools is provided. The GxlStats and GxlSubGraph modules were mentioned in Section 4.2, and presented here in detail.

C.1 GxlStats

Provided here is the statistics program described in Section 5.1. This is some code to generate a few useful statistics over a *Gxl* graph. It does this by first converting a *Gxl* graph into an internal representation (that given by King & Launchbury [KL95], the implementation of which is given in the *Data.Graph* module).

```
module Main where
import System
import qualified Gxl
import Data.Graph
import Data.Array
import Text.XML.HaXml.Xml2Haskell
import Text.XML.HaXml.OneOfN
import MyPrelude
```

The main thrust of the program is to read in a *Gxl* file (name given as an argument on the command line), then say we're going to compute statistics, then compute the statistics and show them.

```
main :: IO ()
main = do
       [infile] ∈ getArgs
       putStrLn ("Reading from "++infile)
       value ∈ fReadXml infile :: IO Gxl.Gxl
       putStrLn ("Computing Statistics")
       putStrLn $ getStats $ value
```

First, we need to convert the *Gxl.Graph* file into a *Data.Graph*. To do this, we first take the *Gxl.Graph* apart enough to get at the *Node*s, *Edge*s, and *Rel*s. We then create a *Data.Graph* from these, by going through and finding each *Node*, then finding each *Edge* that goes from that node to another node. This list of edges is then fed into the **graphFromEdges** graph constructing function from *Data.Graph* to give us a *Graph*, and a conversion function that takes each vertex number, and gives us the *Gxl* node that represents along with the *NodeId*, and the *NodeId*s of all subsequent nodes in the *Gxl.Graph*.

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

gxlGraphToDataGraph :: Gxl.Graph
                    → (Graph, Vertex → (Gxl.Node, String, [String]))
gxlGraphToDataGraph (Gxl.Graph gas _ _ ners)
    = graphFromEdges $ nersToEdges ners ners

nersToEdges :: [NER] → [NER] → [(Gxl.Node, String, [String])]
nersToEdges all [] = []
nersToEdges all ((OneOf3 n@((Gxl.Node nas typ attrs gs))):ners)
    = (n
      , Gxl.nodeId nas
      , nodeIdsFromNode n all):nersToEdges all ners
nersToEdges all ((TwoOf3 _):ners) = nersToEdges all ners
nersToEdges all ((ThreeOf3 _):ners) = nersToEdges all ners

nodeIdsFromNode :: Gxl.Node → [NER] → [String]
nodeIdsFromNode n ners = [edgeTo e
                         | e ∈ ners
                         , isEdge e
                         , edgeFromN n e]
    where edgeFromN (Gxl.Node nas _ _ _)
                    (TwoOf3 (Gxl.Edge eas _ _ _))
                = (Gxl.edgeFrom eas) ≡ (Gxl.nodeId nas)
          edgeTo (TwoOf3 (Gxl.Edge eas _ _ _)) = Gxl.edgeTo eas
```

Now that we can move between *Gxl.Graphs* and *Data.Graphs*, we can now get our set of statistics from the *Gxl* file by going through each graph in the gxl and getting the statistics from that graph, and concat all these statistics together into one string to output. In the normal case, there will be only a single graph in each *Gxl* file.

```
getStats :: Gxl.Gxl → String
getStats (Gxl.Gxl _ gs) = concatMap getGraphStats gs

getGraphStats :: Gxl.Graph → String
getGraphStats g
    = (("Number of nodes: " ++ )
       · ((show · length · vertices) graph ++)
       · ("\nNumber of start nodes: " ++)
       · ((show · startNodes) graph ++)
       · ("\nNumber of end nodes: " ++)
       · ((show · endNodes) graph ++)
       · ("\nNumber of back edges: " ++)
       · ((show · length) be ++)
       · ("\nAverage length of back edges: " ++)
       · ((show · avgLength) be ++)
       · ("\nBack Edges: " ++) · (unlines be ++)) "\n"
    where (graph, f) = gxlGraphToDataGraph g
          be = backEdges g
```

The functions we use to get the stats we want follow (these are currently hackish):

```
startNodes :: Graph → Int
startNodes g = length [x | x ∈ indegrees, x ≡ 0]
    where indegrees = elems $ indegree g

endNodes :: Graph → Int
endNodes g = length [x | x ∈ outdegrees, x ≡ 0]
    where outdegrees = elems $ outdegree g

backEdges :: Gxl.Graph → [String]
backEdges (Gxl.Graph _ _ _ ners)
    = [Gxl.edgeFrom (eas be)
       ++ " -> "
       ++ Gxl.edgeTo (eas be)
       | be ∈ ners, isEdge be
      , Gxl.edgeTo (eas be) < Gxl.edgeFrom (eas be)]
    where eas (TwoOf3 (Gxl.Edge attr _ _ _)) = attr

avgLength :: [String] → Float
avgLength ss = average [fromIntegral (start s - end s) | s ∈ ss]
               where start s = readHex' s
                     end s = readHex' $ drop 8 s
```

Here's some functions which prove useful in the conversion

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

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

Here's some old statistic functions which were used before the internal *Data.Graph* representation was used. They may be easier to comprehend, but are generally much less efficient.

```
getNumberNodes :: [NER] → Int
getNumberNodes ners = length [n | n ∈ ners, isNode n]

numberStartNodes :: [NER] → Int
numberStartNodes ners
    = length [n | n ∈ ners, isNode n, noEdgesTo ners n]

numberEndNodes :: [NER] → Int
numberEndNodes ners
    = length [n | n ∈ ners, isNode n, noEdgesFrom ners n]

noEdgesTo :: [NER] → NER → Bool
noEdgesTo ners (OneOf3 node)
    = length [e | e ∈ ners, isEdge e, isEdgeTo e node] ≡ 0
    where isEdgeTo (TwoOf3 (Gxl.Edge eas _ _ _))
                   (Gxl.Node nas _ _ _)
              = (Gxl.edgeTo eas) ≡ (Gxl.nodeId nas)

noEdgesFrom :: [NER] → NER → Bool
noEdgesFrom ners (OneOf3 node)
    = length [e | e ∈ ners, isEdge e, isEdgeFrom e node] ≡ 0
    where isEdgeFrom (TwoOf3 (Gxl.Edge eas _ _ _))
                     (Gxl.Node nas _ _ _)
              = (Gxl.edgeFrom eas) ≡ (Gxl.nodeId nas)
```

C.2 GxlSubGraph

Here is the program described in Section 5.2. This is a bit of code to try to grab just a portion of a full *Gxl* graph. The user will give it two nodes, and if they are indeed connected in a path then the portion of the graph between the two nodes will be given.

```
module Main where
```

```
import MyPrelude
import System(getArgs)
import Numeric
import Data.Graph
import List
import qualified Gxl
import Text.XML.HaXml.Xml2Haskell
import Text.XML.HaXml.OneOfN
import Control.Monad.Error
```

The main function in this program takes four arguments: the file to read in, the file to write to, and the start and end nodes of the subgraph that is desired. If there is an error (usually due to bad command line arguments), then the usage of the program is displayed. If there is is no error, then the *Gxl* file is read into `value` by the `fReadXml` function of the *HaXml* source. The `start` and `end` values are then interpreted, and the gxl of the subgraph of `value` from `start` to `end` is printed out in the `outfile` file.

```
main :: IO ()
main = (do
       [infile, outfile, start, end] ∈ getArgs
       putStrLn ("Reading from "++infile)
       value ∈ fReadXml infile :: IO Gxl.Gxl
       fWriteXml outfile (gxlSubGraph value start end)
       ) `catchError` usage

usage :: IOError → IO ()
usage e = putStr $ unlines
          [ "GxlSubGraph"
          , "-----------"
          , "Usage: GxlSubGraph [input gxl] [output gxl] start end"
          , ""
          , "For Example:"
          , "    GxlSubGraph bpc.gxl bpc-sub.gxl 353d 38c9"
          , "to get the nodes between 0x353d and 0x38c9"
          ]
```

So the function we want to implement takes a *Gxl* graph, a start vertex, and an end vertex, and returns a *Gxl.Graph* that contains solely the part of the graph (if any exists) between those two nodes. If there are edges that lead off from this subgraph, then they are all pruned.

To do this, we first get the list of nodes that are actually in a path from start to end, then filter out anything that doesn't have anything to do with those nodes.

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

gxlSubGraph gxl@(Gxl.Gxl gxlatt ((Gxl.Graph gas gt ga ners):gs))
            start
            end
    = Gxl.Gxl gxlatt [(Gxl.Graph gas gt ga goodNers)]
    where goodNers = filter (goodNer f goodvs) ners
          goodvs = if goodV dataStart ∧ goodV dataEnd
                   then subGraph datagraph (v dataStart) (v dataEnd)
                   else []
          v (Just x) = x
          goodV (Just _) = True
          goodV Nothing  = False
          dataStart = findVertex (datagraph,f) start
          dataEnd   = findVertex (datagraph,f) end
          (datagraph, f) = gxlToDataGraph start end gxl

goodNer :: (Vertex → (Gxl.Node, String, [String]))
        → [Vertex]
        → NER
        → Bool
goodNer f vs (OneOf3 node) = node `elem` goodNodes
          where goodNodes = map (fst3 · f) vs
goodNer f vs (TwoOf3 (Gxl.Edge eas _ _ _))
```

```
    = (Gxl.edgeFrom eas) ‘elem‘ goodNodeIds
      ∧ (Gxl.edgeTo eas) ‘elem‘ goodNodeIds
          where goodNodeIds = map (snd3 · f) vs
goodNer f vs (ThreeOf3 r) = False
```

First, convert the first graph from a *Gxl.Gxl* to a *Data.Graph*. We pass along the start and end nodes so that edges going to the start node and edges leading from the end node can be severed so as to make generating a subgraph an easy operation of finding the intersection of the vertices reachable from the start with the vertices in the transposed graph which are reachable from the end.

We begin by converting a *Gxl.Gxl* to a *Data.Graph*. That is, we take the first graph in a *Gxl* and return the result of that converted to a *Data.Graph*, along with a function to go from a *Vertex* in the *Data.Graph* back to a triple of the node in the *Gxl*, the node id of the vertex, and the node ids of the next vertices in the graph.

```
gxlToDataGraph :: String
                → String
                → Gxl.Gxl
                → (Graph, Vertex → (Gxl.Node, String, [String]))
gxlToDataGraph s e (Gxl.Gxl _ (g:gs)) = gxlGraphToDataGraph’ s e g
```

The gxlToDataGraph just calls gxlGraphToDataGraph once with the first graph in the *Gxl*. gxlGraphToDataGraph does the actual conversion from the *Gxl.Graph* to the *Data.Graph* by calling the graphFromEdges function from *Data.Graph*, with a list of edges that we derive from the gxl in nersToEdges.

```
gxlGraphToDataGraph’ :: String
                      → String
                      → Gxl.Graph
                      → (Graph, Vertex
                          → (Gxl.Node, String, [String]))
gxlGraphToDataGraph’ s e (Gxl.Graph gas _ _ ners)
    = graphFromEdges $ nersToEdges s e ners
```

nersToEdges is still taking the start and end node ids (represented by *String*s), along with the list of all *Node*s, *Edge*s, and *Rel*s, and returns a list of edges in the format that graphFromEdges expects. That is, a list of triples of the form (*Node*, *NodeId*, [*NodeIds*]) where the *Node* is the actual *Gxl.Node*, the first *NodeId* is the *NodeId* from that *Node*, and the list of *NodeId*s are the next nodes in the graph.

```
nersToEdges :: String → String → [NER]
            → [(Gxl.Node, String, [String])]
nersToEdges s e ners = nersToEdges' s e ners ners
```

nersToEdges actually calls nersToEdges' which does more of the heavy lifting in the form of keeping around the whole list of *NER*s, as well as the ones that haven't been processed yet. This list of unprocessed *NER*s is then pattern matched to find out if it is a *Node*, an *Edge*, or a *Rel*. If it's a node, then the tuple is generated using the nodeIdsFromNode function, and then concatenated to the list of nersToEdges' of the remaining *NER*s. If it's an *Edge* or a *Rel*, then just return the list from converting all of the following *NER*s.

```
nersToEdges' s e all [] = []
nersToEdges' s e all ((OneOf3 n@((Gxl.Node nas typ attrs gs))):ners)
    = (n, Gxl.nodeId nas, nodeIdsFromNode s e n all)
      : nersToEdges' s e all ners
nersToEdges' s e all ((TwoOf3 _):ners) = nersToEdges' s e all ners
nersToEdges' s e all ((ThreeOf3 _):ners) = nersToEdges' s e all ners
```

nodeIdsFromNode takes the start and end nodes, the *Gxl.Node* we're dealing with, and the list of all *NER*s to produce a list of next *NodeId*s. To do this, it returns the edgeTo of an edge e, where e comes from the list of all *NER*s, it is an edge, and the edge is not going to the start node, or from the end node, hence severing the graph so that our subgraph function will be happy. Auxiliary functions to determine whether the edges looked at are from our node n, where the edge is going, and whether it's going to the start, or coming from the end.

```
nodeIdsFromNode :: String → String → Gxl.Node → [NER] → [String]
nodeIdsFromNode start end n ners
    = [edgeTo e | e ∈ ners,
                  isEdge e,
                  edgeFromN n e,
                  (¬ · toStart) e,
                  (¬ · fromEnd) e]
    where edgeFromN (Gxl.Node nas _ _ _)
                    (TwoOf3 (Gxl.Edge eas _ _ _))
                = (Gxl.edgeFrom eas) ≡ (Gxl.nodeId nas)
            edgeTo  e = (Gxl.edgeTo · eAttr) e
            toStart e = Gxl.edgeTo (eAttr e) ≡ start
            fromEnd e = Gxl.edgeFrom (eAttr e) ≡ end
            eAttr (TwoOf3 (Gxl.Edge eas _ _ _)) = eas

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

Idea of the first subgraph function, is to include all of the nodes that exist between the **start** and the **end**, by finding all of the nodes in the graph that are **reachable** from the **start**, and then all of the nodes that are **reachable** from the **end** (if all the edges were flipped), and then take the intersection of those to get all of the nodes that are inbetween. This doesn't work out well on its own, as there are many superfluous nodes included.

So, the second subgraph function gets closer to what we want by doing a topological sort of the graph (an arrangement of the vertices into a linear sequence, $[v_1..v_n]$ such that for all i, j where $i < j$ there are no edges from $v_j \longrightarrow v_i$). We then drop everything up until the **start** node we are interested in, then take everything up until the **end** node we are interested in. This method also includes some superfluous nodes, but not nearly as many as in the first method.

```
subGraph :: Graph → Vertex → Vertex → [Vertex]
subGraph graph start end = (reachable graph start)
                           ‘∩‘
                           (reachable (transposeG graph) end)
```

```
subGraph2 :: Graph → Vertex → Vertex → [Vertex]
subGraph2 graph start end
    = (start:) · (end:)
      $ (takeWhile (≠ end) · dropWhile (≠ start)) vs
    where vs = topSort graph
```

Given a *Graph* and a *NodeId*, find the vertex representing that *Node* in the *Graph*.

```
findVertex :: (Graph, Vertex → (Gxl.Node, String, [String]))
           → String
           → Maybe Vertex
findVertex (g,f) key = if length matches > 0
                       then Just (head matches)
                       else Nothing
    where vs = vertices g
          matches = [v | v ∈ vs, (snd3 · f) v ≡ key]
```

C.3 GxlToDot

A little gxl to dot converter so that a GXL file can be easily visualised.

This converter has been tested with almost all of the examples from the GXL web page. They work quite well, except for the hierarchical graphs. I found out that their visualisation of the "Complex Example" was inconsistent with the GXL. I'm curious what tool they use to create those visualisations.

First, some preliminary stuff: name the module, import some *System* functions as well as the XML specific functions to deal with the importing of the GXL file.

```
module Main where

import System (getArgs)
import IO

import Text.XML.HaXml.Wrappers (fix2Args)
import Text.XML.HaXml.Xml2Haskell
import Gxl
import List
import qualified Dot
import Text.XML.HaXml.OneOfN
import System
import Control.Monad.Error
```

When creating the dot file, we will keep only a few selected attributes to display (to keep the overall graph small as including every attribute can lead to a physically large graph).

```
attributesToKeep = ["label", "address", "opcode", "operands"]
```

The main function of the program is to take (possibly) two arguments: read the first one in as a GXL file, and output the DOT to the second file.

```
main = (do
        args ∈ getArgs
        let infile = args!!0
        let outfile = args!!1
        let hilight = drop 2 args
        putStrLn ("Reading from "++infile)
        value ∈ fReadXml infile :: IO Gxl
        putStrLn ("Writing to "++outfile)
        let dotGraph = gxlToDotGraph value hilight
        if (outfile ≡ "-")
           then putStr $ show $ dotGraph
           else do writeFile outfile $ show $ dotGraph
                   putStrLn "Done."
       ) `catchError` usage

usage :: IOError → IO ()
usage e
```

```
    = do
      putStrLn "Usage: gxltodot [gxl] [dot] [nodelist to hilight]"
```

Now, the meat of the conversion. We take a *Gxl* data structure, and convert it into a *Dot* data structure. To do this, we take only the first graph that we find in a *Gxl* file, and create a new *DotGraph* out of that graph by going through each of the *Node*s, *Edge*s, and *Rel*s in the *Gxl* and converting them to the equivalent *Dot* statements. For *Node*s and *Edge*s, this is normally a one-to-one translation (except in the case of subgraphs), but *Rel*s (being hyperedges) get translated into a single node and an edge for each tentacle. Thus, for each *Node* and *Edge* we return a single *DotStmt*, and for each *Rel* we return a list of *DotStmt*s.

```
gxlToDotGraph :: Gxl → [String] → Dot.DotGraph
gxlToDotGraph (Gxl _ (g:gs)) h = graphToDotGraph g h

graphToDotGraph :: Graph → [String] → Dot.DotGraph
graphToDotGraph (Graph gas _ _ ners) h
    = Dot.DotGraph
      ((dotScrub · graphId) gas)
      (concatMap (nerToStmts h) ners)

nerToStmts :: [String] → OneOf3 Node Edge Rel → [Dot.Stmt]
nerToStmts h (OneOf3 n)   = nodeToDotStmts n h
nerToStmts _ (TwoOf3 e)   = edgeToDotStmts e
nerToStmts _ (ThreeOf3 r) = relToDotStmts r
```

Here's where the conversion of each piece takes place.

```
nodeToDotStmts :: Node → [String] → [Dot.Stmt]
nodeToDotStmts n@(Node nas _ att gs) h
    = (Dot.Node
       (dotScrub nId)
       (("label",(concatMap attrToString att))
        :hilightNode n h)
```

```
      ):map subGraphToDotStmt gs
    where nId = nodeId nas

hilightNode :: Node → [String] → [Dot.Attr]
hilightNode (Node nas _ _ _) h
    | (nodeId nas) ‘elem‘ h = [("style","filled"),("color","red")]
    | otherwise = []

edgeToDotStmts :: Edge → [Dot.Stmt]
edgeToDotStmts e@(Edge eas _ att gs)
    = (Dot.Edge
       ((dotScrub · edgeFrom) eas)
       ((dotScrub · edgeTo) eas)
       (("label", (concatMap attrToString att))
        :(edgeOrdersToDotAttrs eas)++hilightEdge e)
      ):map subGraphToDotStmt gs

hilightEdge :: Edge → [Dot.Attr]
hilightEdge (Edge eas _ _ _)
    | isBackedge eas = [("style", "bold"), ("color","red")]
    | otherwise = []
    where isBackedge eas = edgeFrom eas > edgeTo eas

relToDotStmts :: Rel → [Dot.Stmt]
relToDotStmts (Rel ras _ as gs res)
    = (Dot.Node
       rId
       [("label",(concatMap attrToString as))
       ,("shape","diamond")]
      ):(map (relEndToStmt rId) res)
    where rId = dotScrub $ unMaybe $ relId ras

relEndToStmt :: String → Relend → Dot.Stmt
relEndToStmt relId (Relend (Relend_Attrs trg role dir sOrd eOrd) as)
    = Dot.Edge relId (dotScrub trg) [("dir",dirToStr dir),
                                     ("label",unMaybe role),
                                     ("taillabel", unMaybe sOrd),
                                     ("headlabel", unMaybe eOrd)]
        where dirToStr (Just Relend_direction_in) = "back"
              dirToStr (Just Relend_direction_out) = "forward"
              dirToStr (Just Relend_direction_none) = "none"
              dirToStr _ = "none"
              sOrdToStr (Just s) = s

unMaybe :: Maybe String → String
unMaybe (Just x) = x
unMaybe (Nothing) = ""

edgeOrdersToDotAttrs :: Edge_Attrs → [Dot.Attr]
edgeOrdersToDotAttrs eas = (if isF then (("taillabel",unMaybe f):)
                             else id) ·
                            (if isT then (("headlabel",unMaybe t):)
                             else id) $ []
    where f = edgeFromorder eas
          t = edgeToorder eas
          isF = f ≠ Nothing
          isT = t ≠ Nothing

attrToDotAttr :: Attr → Dot.Attr
attrToDotAttr (Attr (Attr_Attrs _ name _) _ _ v)
    = (name,showsOneOf10 v "")

subGraphToDotStmt :: Graph → Dot.Stmt
subGraphToDotStmt (Graph gas _ _ ners)
    = Dot.Subgraph
      ((dotScrub · graphId) gas)
      (concatMap (nerToStmts []) ners)
    where
```

Node and *Edge* names in dot can't include the following characters: ".-" (and maybe more), so these have to be scrubbed.

```
dotScrub = filter (¬ · (‘elem‘ ".-"))

attrToString (Attr (Attr_Attrs _ name _) _ _ oOf10)
    | name ‘elem‘ attributesToKeep = (name ++) ·
                                     (" = " ++) ·
                                     (showsOneOf10 oOf10) $ "\n"
    | otherwise = ""
```

The following is my horrible hack to display possible attribute values in a nice form. I should probably just change the Gxl.hs file to have more sensible *Show* instances than the crappy derived ones. On the plus side, I got to play with `intersperse`, which I didn't know about before.

```
showsOneOf10 :: OneOf10
                Locator
                Gxl.Bool
                Gxl.Int
                Gxl.Float
                GxlString
                Gxl.Enum
                Seq
                Set
                Bag
                Tup
              → ShowS
showsOneOf10 (TwoOf10 (Gxl.Bool b)) = (b ++)
showsOneOf10 (ThreeOf10 (Gxl.Int i)) = (i ++)
showsOneOf10 (FourOf10 (Gxl.Float f)) = (f ++)
```

```
showsOneOf10 (FiveOf10 gStr) = (gxlStringToStr gStr ++)
showsOneOf10 (SixOf10 (Gxl.Enum e)) = (e ++)
showsOneOf10 (SevenOf10 (Seq ss))
    = ('{':)
      · ((concat · intersperse "," $ map seq_ToStr ss) ++)
      · ('}':)
showsOneOf10 (EightOf10 (Set ss)) = (show ss ++)
showsOneOf10 (NineOf10 (Bag bs)) = (show bs ++)
showsOneOf10 (TenOf10 (Tup ts))
    = ('(':)
      · ((concat · intersperse "," $ map tup_ToStr ts) ++)
      · (')':)

seq_ToStr (Seq_Locator l) = show l
seq_ToStr (Seq_Bool b) = show b
seq_ToStr (Seq_Int i ) = show i
seq_ToStr (Seq_String s) = gxlStringToStr s

tup_ToStr (Tup_Locator l) = show l
tup_ToStr (Tup_Bool b) = show b
tup_ToStr (Tup_Int (Int i)) = i

gxlStringToStr (GxlString s) = s
```

Appendix D

WASH - Password Changing

This is a little CGI program to change passwords in an htpasswd file. It uses WASH (Web Authoring System Haskell) to handle most of the nastiness about writing a CGI.

Compile it (after installing WASH) with:

ghc -package WASH-CGI –make -o pwchange pwchange.lhs

```
module Main where

import Prelude hiding (head, span, div, map)
import HTMLMonad
import CGI
import System
import Data.List (intersperse)
```

First, I setup some constants to be used (where the actual htpasswd file lives, which command is used to manipulate it, and what flags are needed for that command).

```
passfile = "/etc/svn/reveng/svn-auth-file"
htpasswd = "/usr/bin/htpasswd"
htflags = "-mb"
chkhtpass = "true"
```

The main portion of the program just does a "run" of the mainCGI.

```
main = run mainCGI
```

The `mainCGI` will execute a `standardQuery` from WASH which creates a page that will pose the question (titled Password Change), and provide prompts for the username, the old password (to verify), and the new password (typed twice so that they're sure they got it right). All of this is done in a table, so auxiliary functions quenstions and pwprompt were created to generate the rows of the table with columns for the prompt and the `indputField` or `passwordInputField`. The final row has a submit button which passes the values from the four fields to the changepass function.

UPDATE: I removed the "Username" and "Old Password" stuff, as this is just going to be an htaccess protected cgi-bin with "require valid-user" (which will provide the username and have checked the password before they can execute the cgi).

```
mainCGI
    = (standardQuery "Password Change"
        (table
          (do newF ∈ pwprompt "New Password: " (fieldSIZE 20)
              nw2F ∈ pwprompt "New Password (again): " (fieldSIZE 20)
              tr (td (submit (F2 newF nw2F) changepass
                    (fieldVALUE "Change"))))))
```

The changepass functions takes one argument, which is the four fields from the query page combined (which are then converted from a special type which forces them to be `nonEmpty` to a standard string). A "same" value is also generated which is true when the new password was typed the same twice.

```
changepass (F2 n_f n2_f) =
  let newpasswd = unNonEmpty (value n_f)
      newpassd2 = unNonEmpty (value n2_f)
      same = newpasswd ≡ newpassd2
```

First, we ensure that the old password is valid for the user. This is done using an external program specified by `chkhtpass`. That's given the password file, the username, and the old password. If it returns with a success, then `validpw` is *True*, otherwise it is *False*.

```
  in do username ∈ io $ getEnv "REMOTE_USER"
        validpw ∈ if same ∧ username ≠ ""
                    then systemSuccess [chkhtpass,passfile,username]
                    else io (do return False)
```

Then, if the old password is valid, an attempt is made to update the password using the htpasswd program. If the exit status indicates a *Success*, then success will be *True*. If the old password was invalid or the update failed, then success is *False*.

```
        success ∈ if validpw
                    then systemSuccess [htpasswd,
                                         htflags,
                                         passfile,
                                         username,
                                         newpasswd]
                    else io (do return False)
```

Finally, we create a standard page that tells the user whether the result of their attempt at a password change was successful or failed. In the future, this should probably tell them why it failed, but for this quick & dirty hack, they can just ask and administrator if necessary.

```
        htell $ standardPage
                   (if success then "Success" else "Failed!")
                   (text ("Changing password for ")
                    ## text (username) ##
                    if success then (text (" successful! "))
                    else (text (" failed!")))
```

Here are the auxiliary functions for posing the questions and password prompts, as well as the function which performs a system call and returns whether or not the result indicates *Success*.

```
question prompt attrs =
  tr_T (do td_S (text prompt)
           td_S (inputField attrs))

pwprompt prompt attrs =
  tr_T (do td_S (text prompt)
           td_S (passwordInputField attrs))

systemSuccess xs = io (do result ∈ system $ unwords xs
                          return $ result ≡ ExitSuccess)
```

Appendix E

Other Useful Functions (MyPrelude)

Some extras that I find useful

```
module MyPrelude where
import Numeric
import Data.Word
```

padString will pad out a *String*, s, until it is at least l characters long, using padchar to fill it out from the beginning.

```
padString :: Int → Char → String → String
padString l padchar s
    | length s < l = padString l padchar (padchar:s)
    | otherwise    = s
```

A revised readHex to just get the first hex number read (if any), or return 0 if no hex numbers were found.

```
readHex' :: (Num a) => String → a
readHex' s | length r > 0 = fst $ head $ r
           | otherwise    = 0
    where r = readHex s
```

```
readDec' :: String → Int
readDec' s | length r > 0 = fst $ head $ r
           | otherwise    = 0
    where r = reads s
```

An average function, from "A Gentle Introduction to Haskell".

```
average :: (Fractional a) => [a] → a
average xs = sum xs / fromIntegral (length xs)
```

A quick function to check if a string contains a space at the location specified by n (1-based, hence the $n - 1$).

```
isSpace :: String → Int → Bool
isSpace cs n = cs!!(n-1) ≡ ' '
```

Substring acts similarly to perl's `substr` command, where given a list `s`, an `offset`, and a `length`, returns a list of that length starting at after the `offset`. So, for example,

substr "fred" 1 3

will return "red".

```
substr s start length = (take length · drop start) s
```

Some auxiliary functions for cleaning up the hex display of addresses. They are to be displayed in hex (without any prefix denoting such, hence the dropping of the "0x"), and '0'-padded out to 4 nibbles.

```
showsH :: Word16 → String → String
showsH w = (padString 4 '0') · showHex w

showH w = showsH w ""
```

Definition of functions to get the first, second, or third item in a triple.

```
fst3 :: (a,b,c) → a
fst3 (x,_,_) = x

snd3 :: (a,b,c) → b
snd3 (_,y,_) = y

thrd3 :: (a,b,c) → c
thrd3 (_,_,z) = z
```

The following function gets the limit of a function. That is, it repeatedly applies a function until the result is the same as the last application.

```
limit :: Eq a => (a → a) → a → a
limit f x
    | x ≡ next = x
    | otherwise = limit f next
    where
    next = f x
```

Now, a function to replace all occurances of one character in a string with another string. This is useful for doing things such as replacing all instances of the character "<" with the string "<".

```
replace :: Char → String → String → String
replace x ys = concatMap (λc → if (c ≡ x) then ys else [c])
```

Functions to drop the prefix (everything up to the last '/'), the suffix (everything after and including the last '.'), and every suffix (everything after and including the first '.') of a string.

```
dropPrefix = reverse · takeWhile ('/' ≠) · reverse
dropSuffix = reverse · tail · dropWhile ('.' ≠) · reverse
dropSuffixes = takeWhile ('.' ≠)
```

Now a function is defined which, from a list of *Either* a or b, will return a list of all of the bs.

```
rights :: [Either a b] → [b]
rights = foldr (either (flip const) (:)) []
```

Often it is necessary to escape entity references so that, for example, all occurances of the character '¡' are replaced with "&lt;". The `unEntity` function handles this by proceeding through each character of the string, and if it encounters a character that needs escaping, it then returns a function that concatenates the escaped version (usually multiple characters), otherwise it returns a function that performs a cons (:) using only the character itself. This is applied recursively to itself until it gets to an empty list.

```
unEntity [] = []
unEntity (c:cs) = (if c `elem` "<>&\f\r"
                   then (entify c ++)
                   else (c:)) $ unEntity cs

entify '<' = "&lt;"
entify '>' = "&gt;"
entify '&' = "&amp;"
entify '\f' = ""
entify '\r' = ""
```

Bibliography

[AK03] Norm Aleks and Brian Knittel. *All about the IBM 1130 Computing System.* http://www.ibm1130.org/, 2003. 4

[CKK$^+$04a] J. Carette, W. Kahl, R. Khedri, M. Lawford, K. Sartipi, J. Sun, and A. Wassyng. Requirements for reverse engineering tools. preprint, available through SQRL, 2004. 5, 12

[CKK$^+$04b] J. Carette, W. Kahl, R. Khedri, M. Lawford, K. Sartipi, and A. Wassyng. Procedure for reverse engineering of high-level requirements from assembly code. preprint, available through SQRL, 2004. 6

[EKW00] J. Ebert, B. Kullbach, and A. Winter. GraX: Graph eXchange Format. Technical report, 2000. 16

[GKN02] Emden Gansner, Eleftherios Koutsofios, and Stephen C. North. *Drawing graphs with dot.* Murray Hill, NJ, 2002. 42

[HWS00] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Towards a Standard Exchange Format. Technical Report 1–2000, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2000. 9, 16

[IBM65a] IBM. *IBM 1130/1800 Assembler Manual*, 1965. 34

[IBM65b] IBM, Murray Hill, NJ. *IBM 1800 Functional Characteristics*, 1965. 34

[JPP94] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: computing control regions in linear time. In *Proceedings* *of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 171–185. ACM Press, 1994. 49

[KL95] David J. King and John Launchbury. Structuring depth-first search algorithms in haskell. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.*, pages 344–354, New York, NY, 1995. 48, 73

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984. 15

[QRY04] Daniel Quinlin, Paul 'Rusty' Russell, and Christopher Yeoh. Filesystem Hierarchy Standard. electronically available via: `http://www.pathname.com/fhs/`, 2004. 52

[Thi03] Peter Thiemann. An embedded domain-specific language for type-safe server-side web-scripting. electronically available via: `http://www.informatik.uni-freiburg.de/~thiemann/haskell/WASH/draft.pdf`, 2003. 52

[Win01] Andreas Winter. Exchanging Graphs with GXL. Technical Report 9–2001, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2001. 16

[Wu04] Jun Wu. Formalization of GXL in Z notation. Master's thesis, McMaster University, Department of Computing and Software, 2004. 16, 49