Computer Science at Kent

Iguana:
A management support tool using Haskell and LDAP

Chris Ryder
cr24@ukc.ac.uk

Technical Report No: 6-01
Date: June 2001

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent CT2 7NF, UK.

Abstract

Haskell is widely used within research and academia but is less well used for "real world" projects. This paper describes a real world project using Haskell in a larger scale data processing application. The project was undertaken jointly by British Airways and the Computing Laboratory, University Of Kent.

1 Introduction

This paper describes a project undertaken between British Airways and the Computing Laboratory at the University of Kent. The project involved developing a system in Haskell for use by British Airways' Information Security Department.

1.1 Background to the Project

British Airways (hereafter known as BA) have an LDAP (Lightweight Directory Access Protocol) directory which contains an entry for every member of staff (approximately 100,000) and is used for many tasks from system authentication to the provision of an on-line telephone directory. Among the information stored about each employee are their name, employee number, userid, and the identity of their manager. From this information it is possible to build a model of the management hierarchy within BA.

A requirement in any large organisation is to measure how it is performing against a variety of criteria. Frequently this is done by measuring some aspect of each management unit and then aggregating the scores. Moreover, once management know aggregate scores they invariably want to "drill down" the organisation to understand which management units are performing well and which need their attention. For some years, the Information Security department at BA has been measuring the performance individual management units and, with the recent introduction of the LDAP directory, wished to produce aggregate reports based on the management structure represented within it. It was conjectured that Haskell, with its support for structures such as organisational trees would be well suited to produce such reports quickly, reliably and cheaply.

1.2 What is LDAP ?

A Lightweight Directory Access Protocol (LDAP) directory is a special form of database in which data is typically read many more times than it is written or modified. It is thus optimised for reading data.

All entries in an LDAP directory contain a *distinguished name* (DN), which is a unique identifier for a given entry. A DN is hierarchical, similar to path names in a file system or domain names in the Internet world. For example, a textual representation of a DN might look something like:

```
employeeNumber=123, ou=people, dc=baplc.com
```

This specifies the unique entry which has the *employee number* `123`, is part of the *organisational unit* (ou) "`people`" and is in the directory whose root is the *distinguished component* (dc) "`baplc.com`" which is guaranteed unique by Internet naming standards. Structuring DNs in such a way makes searching for a specific DN quicker, and also makes it easier to distribute the directory over multiple servers (in a similar way to domain name server distribution).

Entries may have other attributes as well as the DN, though they are not compulsory. Attributes consist of an *attribute type* and *attribute values*. These can be thought of as an attribute name and value. An entry may have only one instance of an attribute type ("name") but may have multiple values for that type, e.g, an entry cannot have multiple "`telephonenumber`" attributes, but may have a single "`telephonenumber`" attribute, containing two telephone numbers.

The LDAP protocol is a binary protocol and the data stored within an LDAP directory may be stored in a proprietary binary format. To aid moving data from one system to another, a textual representation of entries from an LDAP directory is often used. This textual form is called LDIF (LDAP Data Interchange Format), which looks like this

```
uid=12345, ou=People, dc=ldaptest
cn=Fake User
uid=12345
```

For this entry the `DN` is "`uid=12345, ou=People, dc=ldaptest`" and the entry has two attributes, a `cn` (common name) of "Fake User" and a `uid` (userid) of "12345". The entries in the BA LDAP directory have many more attributes, but for this project only those mentioned here are relevant.

Because all the attributes of an entry are not always required, it is possible to ask for just a subset of the available attributes to be returned from a query. For example :

```
ldapSearch "(uid=100)" ["cn","telephonenumber"]
```

would search for the entry with the `uid` of `100`, and return only the common name and telephone number.

There are numerous client-side libraries in several programming languages for accessing LDAP directories. For this project, a C library was used (See Section 3). Further information about LDAP and the LDAP libraries can be found in [1] and [2].

1.3 The Project

The purpose of the project was twofold :-

1. Each employee within BA is assigned a score in the range 0 to 100 inclusive. These scores are held in a flat file exported from a spreadsheet. The aim of the project was to implement a system that generates aggregate scores by taking an average of the individual's scores and the scores of their (immediate) subordinates. The information necessary to construct the management hierarchy is held within the LDAP directory. Figures 1 through 3 illustrate a particular example of the process in diagrammatic form. The aggregate information is output in the form of a collection of HTML files that let management view the information hierarchically. This allows management to quickly spot under-performing management units.

2. To evaluate the suitability of Haskell as a language for implementing large systems involving interworking with non-Haskell systems in a real problem.

1.4 Tools Used

The bulk of the system was written using the Glasgow Haskell Compiler with a small part written in C. HaskellDirect was used to allow the Haskell code to call the C code. This is explained in more detail in Section 3. We additionally experimented with using Lambada to call Java code. This is touched on in Section 3. Overall, the tools worked well and proved to be useful, particularly in the case of HaskellDirect.

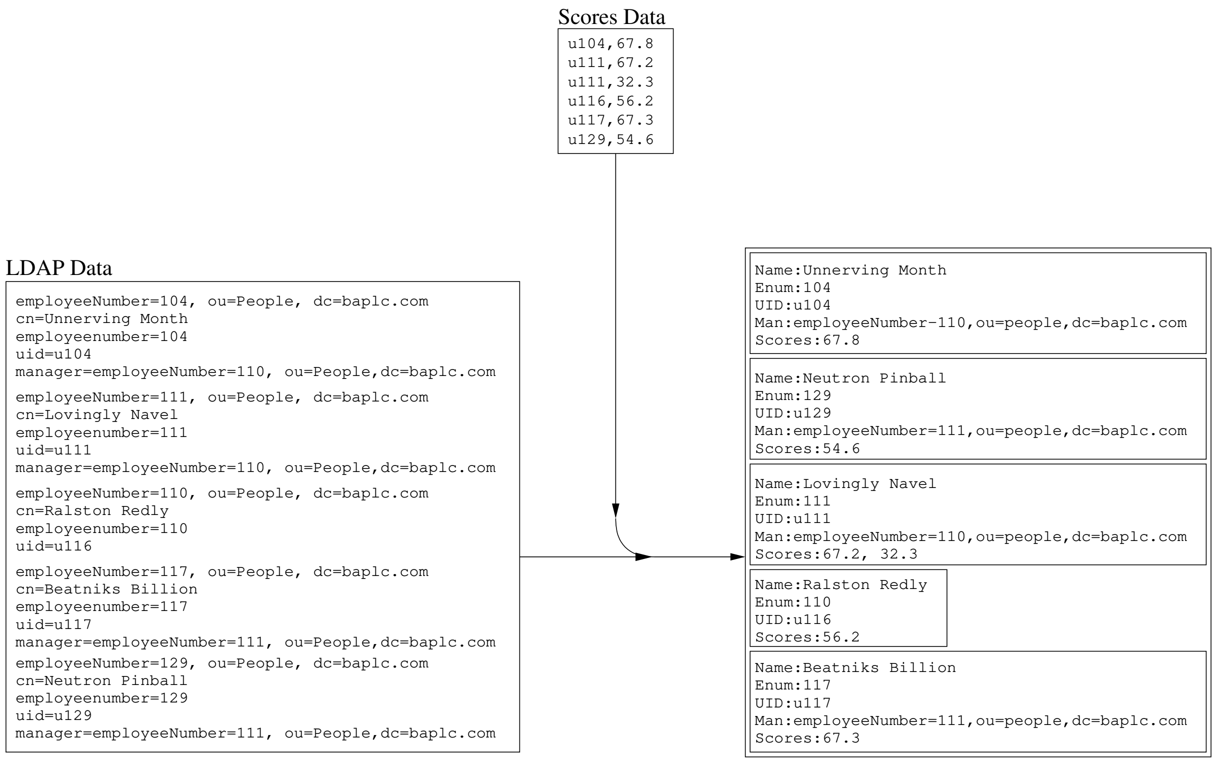

Figure 1: Data from the LDAP directory is combined with data from the scores file into a Haskell data type.

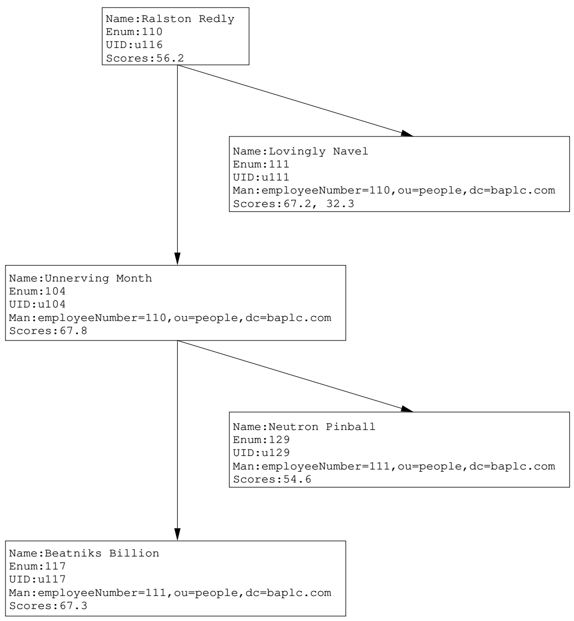

Figure 2: The Haskell data type is constructed into a tree.

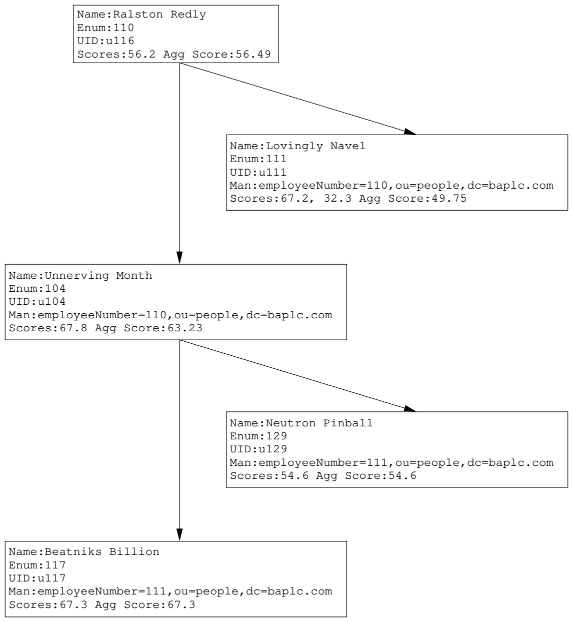

Figure 3: The aggregate scores are calculated.

1.5 Why Haskell ?

The project required the system to be developed quickly and cheaply. Since the system is intended to be used by only one department at BA, and is not operationally critical, it was felt that it was worthwhile for BA to experiment with Haskell for this project.

The project consists of an algorithmic part and an I/O part. Because the algorithmic part consisted of building a tree structure, it was thought that Haskell would be particularity suited to the task. Having decided on Haskell, we were left with two main choices for languages to use for the low-level I/O. These were Java and C. The original choice of Java as the interfacing language proved unworkable in practice (see Section 3) and so C was used.

1.6 Overview of Paper

The remainder of this paper is divided up thus : Section 2 explains the project task in greater detail. Section 3 details the software tools we used to complete the project. Section 4 shows how we finally implemented the system. Section 5 introduces the problems we encountered during the project, and how we solved them. Section 6 presents our conclusions from this project.

1.7 Acknowledgements

Acknowledgements go to BA, for suggesting and funding the project. Dominic Steinitz, BA, for suggest the problem and for general advice during the project. Simon Thompson, UKC, for help and advice during the project, and for editing this paper. Eric Meijer, for help and advice with Lambada. Claus Reinke, UKC, for support with understanding laziness. Graham Walter, BA, for advice on LDAP. Paul Barnett, BA, for system support at BA.

2 Deeper discussion of project

Solving the problem can be divided into three phases. The first is reading in the data to work with, the second is constructing the tree structure and the final phase is outputting the results.

2.1 Reading Input

The input to the program comes from two sources, the LDAP directory and the scores file. Reading the scores file is a trivial parsing exercise.

Reading of the data from the LDAP directory is more interesting. Unfortunately, there are no LDAP client libraries available for Haskell. Because of this it was necessary to use a C LDAP library and use HaskellDirect to interface to it. HaskellDirect provides code to convert between Haskell values and C values. This is called *marshalling*. The reverse process, converting C values into Haskell values is called *un-marshalling*. This introduces some overhead (illustrated in Figure 4) when requesting an item from the LDAP directory. For instance, in the example query in Section 1.2, the following steps must be taken to complete the query :

1. The parameters to the `ldapSearch` command must be marshalled into C values. This requires that they are copied into C variables, taking both time and space.

2. The appropriate C function is called.

3. The LDAP query is transmitted over the network. This creates a time delay.

4. The result(s) of the query are also sent over the network, creating further delay.

5. The returned results must be un-marshalled into Haskell values. This again requires that they values are copied.

6. The result of the query is of type

   ```
   [(AttributeName,[AttributeValue])]
   ```

   where both `AttributeName` and `AttributeValue` are of type `String`. This is not an easy type to manipulate so it is likely to be parsed into a more useful Haskell data type. e.g, for the example query :

   ```
   Person Name PhoneNo
   ```

There are a few things we can do to minimise these overheads. First, making one LDAP request that returns several results is more efficient in terms of marshalling overhead than sending lots of small queries. This is because marshalling only occurs once, when the query is initiated; there is still an un-marshalling overhead for every entry returned, but there is little that can be done about this. An additional method to minimise both the time and space overhead is to use asynchronous communication with the LDAP directory. The time overhead is reduced because it is possible to un-marshall one result while the next result is being sent across the network.

The reason for the decreased space usage is less obvious. When a synchronous request is sent to an LDAP directory, the request blocks until all the results have been returned. Hence, all those results must be stored somewhere until the request is complete. The result is that a large chunk of memory is used to buffer all the results in the LDAP library before they are then un-marshalled.

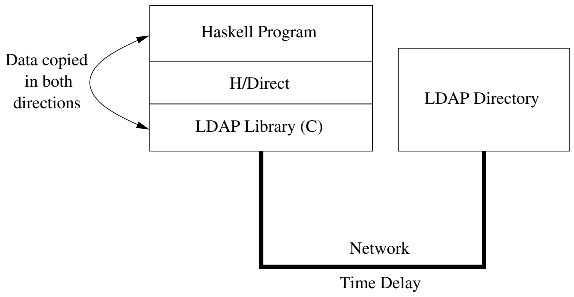

Figure 4: Overhead associated with an LDAP query

Conversely, using an asynchronous query means the search request returns immediately, but the function to retrieve a result blocks until either a result is returned or the query ends. Hence, if the program can un-marshal and process the results more quickly than they are returned from LDAP, only a very small number of results will ever be stored in the buffer of the LDAP library prior to processing.

2.2 Building the tree

To construct the management hierarchy from the data retrieved from LDAP it is necessary to consider two points.

First, the only information within the LDAP directory from which to build the management hierarchy is the `manager` attribute of the entries. Thus, it is necessary to build the tree in a "bottom-up" manner from the `manager` back pointers. This should not be a problem for a functional language such as Haskell.

Secondly, the consistency of the data within the LDAP directory is not guaranteed. Entries may not have a `manager` attribute (in which case they are "top-level" managers) or entries may have a `manager` attribute, but it might contain the `DN` of an entry that no longer exists. Such entries become "top-level" entries but must be distinguished from those entries with no `manager` attribute.

It is worth noting that it is possible to calculate aggregate scores in two ways. They can be calculated, from the bottom up, as the tree is constructed, or they can be calculated by walking over the tree once it has been constructed.

2.3 Outputting the results

The output of the tree is relatively trivial. There are a number of choices for the format of the output. Our original intention was to produce a Comma Separated Variable (CSV) file, but this resulted in a file that was too long to load into the Microsoft Excel spreadsheet. It would be possible to output other formats such as XML, however we eventually settled on using HTML.

HTML was chosen as the output format for a number of reasons. It is simple to produce, which reduced the amount of time spent on the output module of the program. HTML also offers easy ways to represent hierarchical data. This, along with the wide availability of web browsers for many platforms, made HTML a good choice.

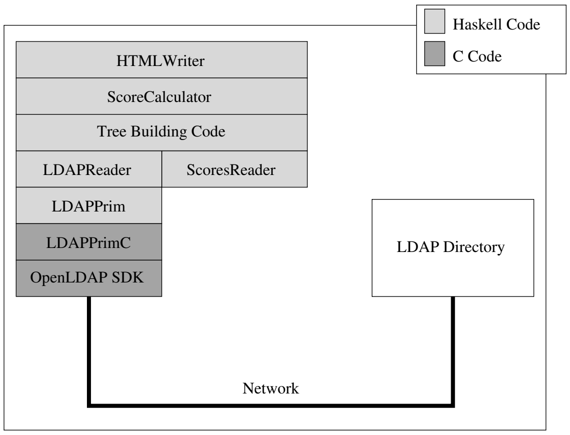

Figure 5: Simplified model of the program architecture.

3 More About The Tools Used

The program was initially developed using GHC 4.04 and HDirect 0.16, which seemed to be the only compatible versions at the time the project was started in July 2000. Later on we moved to GHC 4.08 when HDirect 0.17 was released.

HaskellDirect [3] [4] is an IDL (Interface Definition Language) to Haskell compiler. It allows you to write descriptions of C libraries in IDL and generate Haskell code that will allow you to call the library from Haskell and vice versa.

To do this it uses the Foreign Function Interface[5] built into newer Haskell compilers, and provides some libraries of its own that provide marshalling and un-marshalling functions. [6] and [7] have more information on the subject.

HaskellDirect also provides facilities for calling Java from Haskell using Lambada [8]. Lambada is still in the early stages of development. Because of this, it proved to be tricky to compile and was also rather buggy in use. It has great potential, but requires more development.

GHC can be tricky to compile from source, due to bootstrapping problems (as it is written in Haskell), and eventually we used a pre-compiled binary. Once up and running, GHC is very stable. A useful feature of GHC is its ability to generate Makefile dependencies for a given program.

We had some problems finding a compatible combination of HaskellDirect and GHC versions, due to changes in the Foreign Function Interface. Once compatible version had been found, HaskellDirect worked well for straight forward marshalling/un-marshalling but some problems were encountered when un-marshalling the result of a function that returned `char**` (an array of strings). HaskellDirect un-marshalled this into `[Ptr]`, instead of `[String]`. This was fixed by modifying the marshalling code by hand.

4 Implementation Overview

This section offers a brief overview of the final implementation. It is not an exact description of the implementation, but offers a general idea of how the program works.

The program has a layered structure. Each layer is generally self contained in a single source file or module. The only exception to this is the tree-building code which is actually in several files. Some of the layers are written in C rather than Haskell. The structure is illustrated in Figure 5.

The layers below the tree building code are concerned only with providing data to the higher levels. They could easily be replaced or modified to import data from some other source.

The tree building code works by creating a large `IOArray` (a mutable array) which is large enough to hold all the entries from the LDAP. Each element of the array holds a `Person`. A `Person` consists of the following data.

- Data for the person from the LDAP directory.
- Data for the person from the scores file (initially empty).
- Indices of all subordinates entries in the array (initially empty).
- The aggregate score for this entry (initially empty).

The data from the LDAP directory is read directly into this array. Next, a hash table is created that maps a person's `DN` to their index in the array. This is done by walking over the array adding each entry's index and `DN` into the hash table.

Once the hash table is created, it is then possible to construct the hierarchy tree. This is done by walking over the array adding each `Person`'s index into their manager's subordinate list. To do this, it is necessary to find an entry in the array from a `DN`. This is the purpose of the hash table.

So, after two passes over the array, the hierarchy tree is constructed. The next step is to read in the scores data from the scores file. Unfortunately, the scores file uses the `UID` attribute of an entry as the key. To be able to read in the scores data it is necessary to replace the hash table with one that maps the `UID` attribute (userid) of an entry to its index. This is done with another pass over the array. This makes it possible to read each line of the scores file and fill in the appropriate elements in the array.

From this point it is possible to walk over the tree using a simple recursive algorithm to calculate the aggregate scores. This gives a completed tree, which can then be displayed.

The display is handled by the `HTMLWriter` module. This uses a recursive algorithm to walk over the tree and generates a directory containing HTML files. It would be easy to modify or replace this module to output the results in a different format.

5 Problems Faced

There were two main areas where problems occurred during this project. The first area, interfacing to LDAP, was mainly caused by bugs in software and incompatibilities between versions as described in Section 3.

The second and bigger area where problems occurred was in memory usage. At the minimum it is necessary to store a person's DN, their manager's DN, their user ID and their scores; we also store their CN (common name) for pretty output. A DN string is, on average, about 50-60 characters long. With the large number of entries that need to be worked on this memory usage can grow to be quite large.

Early attempts at storing the results and building the hierarchy tree resulted in programs that worked fine for small test directories (approximately 2000 entries), but used large (greater than 400MB) amounts of memory on realistic directories of approximately 100,000 entries. Since the program was intended to run on a computer with 160MB RAM this was unacceptable.

From the experiments we made to reduce memory usage it appears that `String`s in Haskell can be expensive in memory usage. For this reason, the two DNs that were stored for each person were converted to MD5 checksums [9]. MD5 is a method of generating a message digest (a "fingerprint") from a given string. It is conjectured that it is computationally infeasible to produce two strings with the same message digest. Using MD5 checksums significantly reduced the memory usage because an MD5 checksum is 128 bits long. This is represented in Haskell as a `String` of sixteen characters. There is a small risk that two DNs may be encoded to the same checksum value, but this was considered an acceptable risk. Although reducing the DNs to checksums reduced the memory usage it was still unacceptably large.

Another cause of high memory usage was due to the way data from the scores file needed to be merged with the data from the LDAP directory. The LDAP directory uses DNs to identify individual people, and so DNs were used as the unique identifiers for building the hierarchy tree. However the scores file uses a person's userid to identify people. Because of this it was necessary to have some way of mapping userids to DNs. Experiments were made with balanced trees (both implemented by hand, and using Haskell's `FiniteMap`) but this made a large contribution to the memory usage.

The early version of the program stored data from the LDAP directory in a list of type `[Person]`. It then converted this list into a balanced tree that mapped a userid to a `Person`. At that point the scores were read from the scores file into the entries in the balanced tree. This merged the scores and LDAP data together. Next, the balanced tree was modified so that it was possible to search for the entry with a specific manager attribute. The program then walked over the balanced tree finding all the entries who had incorrect or missing manager attributes. These entries formed the root nodes of the hierarchy tree (actually a forest). From these entries, the whole hierarchy was constructed in a top-down fashion using the following data type.

```
data Tree = Manager Person [Tree]
          | Sub Person
```

The main users of memory in this process were the original (large) list, and the two balanced trees (although the final hierarchy tree also used a big chunk).

To combat this memory usage we eventually used a single mutable array (`IOArray`) and made several passes over this array. We used indices stored with the data in the array to build the hierarchy tree within the array, rather than building an explicit tree. Although it would have been nice to only make passes over the array, it is still necessary to have some mapping between DNs and indices and also between userids and indices (although not at the same time). For this purpose, a second mutable array was used to create a hash table (using chaining for handling collisions). This used significantly less memory than the previous approach which built balanced trees and an explicit hierarchy tree.

Although this improved version was closer to running in an acceptable amount of memory it still required more memory than was available. After further investigation it appeared that parts of the program were being lazy in an unforeseen way. Experimenting with forced evaluation in parts of the program (particularly the parsers) resulted in a dramatic reduction in memory usage. After this result, further parts of the program were modified to force evaluation, before it was settled on which parts of the program benefited from lazy evaluation and which parts didn't. At this point the program was running in a satisfactory amount of memory.

6 Conclusions

In this section we will try to summarise our observations during this project. The biggest problem faced in the project was controlling memory usage. It proved to be quite difficult to monitor and understand memory usage in the program. Profiling tools helped but we experienced some difficulty profiling programs using the FFI. This appeared to be fixed in GHC 4.08. Additionally, there appears to be little solid documentation on how to interpret the output of the profiling tools. Such documentation would greatly increase the usability of the profiling tools. Because of these problems, the memory usage was mostly controlled through trial and error. This approach worked because this is a relatively small program but such an approach would not be possible on a larger scale. It is also interesting to note that, in this instance, lazy evaluation was more of a hindrance than a help. Indeed, the program ended up with large parts of the code having an imperative style to them. This was not our original intention but was forced upon us by the need to reduce memory usage.

It was pleasing to see how well Haskell interacted with other languages, although it was not always straightforward to generate correct marshalling code. However, once the marshalling code was correctly written the interaction between the two languages was perfect. It is the opinion of the author that this is an important area of the language. Haskell cannot "stand alone"; it must be able to interact well with other languages. HaskellDirect provides a good base from which to do this.

Using Haskell in this project enabled us to quickly build a working system and as such showed it is possible to build systems cheaply with Haskell. Unfortunately, there is not enough experience of using Haskell on large scale problems. It is also unfortunate that understanding the memory usage of lazy Haskell problems can be very tricky. This was where a large amount of time was lost during the project.

All in all, Haskell shows great promise for this kind of application. We believe that Haskell will be a suitable language given time and extra tools to ease the understanding of memory usage.

References

[1] RFC2251, `http://www.ietf.org/rfc/rfc2251.txt`

[2] Howes, T., Smith, M. and Good, G. *Understanding And Deploying LDAP Directory Services*

[3] `http://www.haskell.org/hdirect`

[4] Finne, S., Leijn, D., Meijer, E., and Peyton Jones, S. *H/Direct: a binary foreign language interface for Haskell.*

[5] Finne, S. *A foreign function interface for Haskell*

[6] Finne, S., Leijn, D., Meijer, E., and Peyton Jones, S. *Calling hell from heaven and heaven from hell.*

[7] Peyton Jones, S. *Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*

[8] Meijer, E. and Finne, S. *Lambada, Haskell as a better Java.*

[9] RFC1321, `http://www.ietf.org/rfc/rfc1321.txt`