Industrial Experience of Lazy Functional Programming

Paul Sanders
BT Laboratories
Martlesham Heath
IP5 7RE*

Iain Checkland
University of York
York
YO1 5DD†

1 Introduction

Current trends in software engineering point to the development of *higher–functionality* programming languages as the key to solving the problems of producing highly complex systems. These languages abstract away from the machine–level view of programming prevalent in today's popular programming languages. The claimed advantage is that, freed from the constraints of the underlying hardware, programmers may concentrate on producing better algorithms, more reusable code and, possibly, more rigorously developed software.

One example of these higher–functionality languages is Haskell [1], an emerging standard in *functional programming languages* [2, 3]. There have been many claims made of functional programming languages by the academic community but take–up by industry has been very slow. The DTI–funded FLARE project, managed by BT, has the aim of evaluating these claims on real applications:

> *"The effectiveness of functional programming has been amply demonstrated on a laboratory scale and it is now appropriate to scale up the experiments with a view to encouraging the wider community of applications developers to embrace this technology"* — FLARE Project Proposal.

This paper presents some of the results of a case study into the use of Haskell to implement a workforce management system. In order to obtain a balanced view of its suitability a problem specification was implemented independently by a novice Haskell programmer and an experienced Haskell programmer. This enabled a study of two important angles:

- the features of Haskell that make it suitable or unsuitable for the problem task;
- the problems that a person new to functional programming might suffer, and how an expert might solve them.

This paper is organised as follows. Section 2 briefly describes the case study application and its salient features. Section 3 gives some examples of how the features of Haskell were used to

*e-mail: **sanders_p_m@bt-web.bt.co.uk**
†e-mail: **iain@minster.york.ac.uk**

solve the problem and how they relate to current software engineering paradigms. Section 4 compares the results obtained by the novice and the experienced programmer. Section 5 briefly comments on the tool support available for the program development. Section 6 summarises the results of the study.

2 The Workforce Management System

The task of this application is to assign a set of jobs to a set of technicians satisfying a number of constraints concerning their skills, locations and workload. We require that technicians are occupied to their capacity and that travelling time is minimised. The problem decomposes into two distinct phases.

- Phase I – allocating jobs to technicians. To ensure that technicians are occupied to their capacity, we must fully occupy any who already have a partial agenda in preference to allocating a job to a technician with an empty agenda;
- Phase II – producing routes. To ensure that time spent travelling is minimised, we must order the jobs in each technician's agenda to find a least cost route between them.

Phase I can be viewed as a classic constraint–handling problem, resulting in a set of candidate agendas, and Phase II as a best–first search to find the least cost solution.

3 Using Haskell to Solve the Problem

In this section we present some of the Haskell features that were used to implement the program and highlight the benefits that these brought.

3.1 Information Hiding

It has been recognised for over two decades that information hiding is an important principle in developing modular software [4]. The implementation of a module or object should be hidden and access to its operations possible only through a published interface. This makes the operations secure from abuse by the programmer. Further, the implementation can change without affecting any code which uses the module.

In Haskell, information hiding is achieved using the module system restricting the functions that are exported. For example, a module providing operations on jobs might be declared as:

```
> module Job (JobType, jobNo, period, duration, visitStore, location)
> where
> data JobType = Job Int Period Int Bool Location
> jobNo :: JobType -> Int
> jobNo (Job n _ _ _ _) = n
>         ...
```

This creates an abstract data type for the job type, `JobType`. Outside the module, operations on the values of the type are restricted to those exported in the module declaration, given in the brackets. The interface to the module is given by the type signatures of the exported functions. Although `jobType` appears in the export list, its implementation, the constructors, do not. Its implementation is therefore hidden from its users. This corresponds exactly to the encapsulation and abstraction paradigms found in object oriented languages.

3.2 Type Classes

Haskell is unique among functional languages in its provision of *type classes*. These allow types that have common operations to be grouped together and are very similar to the classes and methods concept in object–oriented programming. Haskell and most OOPLs differ, however, since type classes in Haskell are securely typed. An example class is `Eq` in which all types have values that can be compared for equality. The existence of an equality operator is guaranteed on any value of a type belonging to the `Eq` class; the full security of polymorphism is preserved and the runtime system will determine which definition to use.

This use of type classes for overloading proved to be very useful in the development of the case study. Part of the application's task is to parse the data from input files into the appropriate internal representation. The class system allows us to define a general parsing class:

```
> class Parse a where
>       parseFile :: String -> [a]
>       parseFile string = map parseLine (lines' string)
>       parseLine :: String -> a
>       parseLine = fst . parse
>       parse :: String -> (a, String)
>       parse = parseType . whiteSpace
>       parseType :: String -> (a, String)
```

For each type we wish to parse we need only define `parseType`, the other parsing functions are *inherited* from the class definition. For example, the data files containing the job data used integers to represent the time of day. We can define a parser for the `Period` type as:

```
> instance Parse Period where
>          parseType string = case string of
>                                 ('0' : ss) -> (Early, ss)
>                                 ('1' : ss) -> (Morning, ss)
>                                 ('2' : ss) -> (Afternoon, ss)
>                                 ('3' : ss) -> (Anytime, ss)
>                                 ('4' : ss) -> (Last, ss)
```

If similar definitions are made for the other internal types one may simply write the following to parse the job data file.

```
> jobData = parseFile jobFile
```

The ability to overload functions and operators is an important aid to program readability. Haskell makes overloading very straightforward but in current implementations there is some performance penalty to pay in using classes since part of the work to resolve overloading is done at run–time. It is therefore important that definitions are not overloaded when not necessary.

3.3 Higher–Order Functions

The best–first search performed in Phase II of the algorithm is a common computational pattern and is an obvious component for reuse. This can be easily achieved using higher–order functions.

There are three parameters to the search: (i) how to recognise a solution, (ii) how to extract a result from a solution state, and (iii) how to obtain a successor state. These are functions passed as arguments. The search can be defined as:

```
> bestFirst :: Ord state => (state -> Bool) -> (state -> result) ->
>                    (state -> [state]) -> [state] -> [result]
> bestFirst finished result successors = bf
>    where
>    bf [] = error ``No solution found''
>    bf (st:sts) | finished st = result st
>                | otherwise   = (bf . foldr insert st . successors) state
```

This may be used in many applications and is therefore highly reusable. Further, this is a natural solution. No extra effort has been expended in trying to make the function general: its generality is a consequence of the availability of higher–order functions of polymorphic type.

Since the functional parameters are first class values in their own right it is possible to isolate each component of the search for independent testing. This makes testing this function much easier than if it had been written with all components intertwined as might be the case in an imperative language solution.

3.4 Lazy Evaluation

Haskell is a *lazy* functional language. The evaluation strategy used to execute lazy programs has two important features. First, only needed computation is performed and, second, common expressions may share the same evaluation graph and so will only be evaluated once, if at all.

We can use lazy evaluation to define a natural, and efficient, function to allocate a job to a technician. The function closely matches the specification: *the job is allocated to the first technician into whose agenda it fits where only technicians with the appropriate skills are considered and are sorted by travel distance.*

```
> allocateJob :: JobType -> [TechType] -> Status TechType
> allocateJob job = first . map (fits (period job) job) .
>                      sort . map (\x -> (travel x, x) .
>                         collect techNo (skilled job)
>       where travel tech = distance (location job) (base tech)
```

If a non–lazy evaluation strategy is used computation will be wasted performing a complete sort of the technicians and trying to allocate the job to *each and every one* before returning only the first success. A more efficient implementation might be obtained by rewriting the function but the resulting code is unlikely to be so clear and easy to understand.

When searching for a solution state in Phase II states may be tested many times and, if so, their cost will be recalculated many times. In an imperative language this might be avoided by destructively updating some variable in a record the first time the cost is calculated; subsequent demands can then check to see whether they need perform the calculation or not. This has to be explicitly coded by the programmer and is therefore open to error. In Haskell, using lazy evaluation, the checking is done for free.

In order to calculate the cost of a tour we need to calculate the cost of travel between pairs of jobs in the tour, and this is represented by the `Link` type.

```
> data Link = Arc Int JobType JobType
```

When an `Arc` is built between two jobs an expression of the form

```
Arc (distance job1 job2) job1 job2
```

will be used. Lazy evaluation ensures that this distance will not be calculated until something demands its value; when, at some subsequent point, the value of `cost arc` is needed where

```
> cost (Arc c _ _) = c
```

More importantly, because the result is stored as part of the value, subsequent calls of `cost` will find the result already computed. This *memoisation* of the cost will bring large savings if these values are demanded many times.

Lazy evaluation is a valuable tool for producing concise code and minimising the amount of work needed to compute a result, but it is not without cost. Each unevaluated expression occupies memory. If there are many such expressions then garbage collection may occur frequently, negating some (or all) of the time savings. A tool to analyse space usage is briefly mentioned in Section 5.

4 The Experiences of the Novice Programmer

To obtain a balanced view of the benefits of using Haskell for this application, one solution was written by a novice functional programmer and another by an expert. No time for formal training was available to the novice and so training was on–line, from books and nearby experts. The difficulties encountered and the solution obtained was then compared with the experiences of the expert.

The program produced by the novice suffered problems caused by:

- using a sequential style rather than a declarative one;
- not exploiting the power of higher–order functions;
- not giving enough consideration to types;
- failing to take advantage of lazy evaluation.

The unfamiliarity with declarative programming led to an attempt to program procedurally in a functional language. This had the effect of making the code more cluttered and less readable than that of the expert.

The absence of higher–order functions in the novice's code meant there were very few reusable functions. A consequence of this was that testing some of the functions was more difficult than in the expert case and this made the development even harder for the novice.

The novice was used to a typeless language and experienced particular problems with Haskell's use of strong typing. This manifested itself in two ways. First, many type errors were made which, because of the unfamiliar style of the type errors from the compiler, the novice found hard to understand and correct. This increased the development time for the program. Second, the practice of specifying the type of each function with its definition was not followed. In the expert program, specifying types was found to speed up the development process by clarifying the programmer's thought and by producing better type errors from the compiler when they occurred

Since they had no previous experience with lazy evaluation it is not surprising that the novice programmer did not exploit this feature. However, the effect of this was to make the program very inefficient and contain some space leaks.

Measuring execution times reveals that on typical sample data the novice code executed in 19 minutes and 18 seconds and the expert code ran in just 13.8 seconds. Experience in the use of functional programming and laziness prompted the use of appropriate techniques to be used to improve efficiency. As in any language these tricks come with experience and so were not employed by the novice.

Few of the benefits discussed in Section 3 were enjoyed by the novice programmer. This shows that, as with any language, it is possible to use the language in such a way as to negate its benefit. This highlights the need for comprehensive training before the benefits of functional programming can be achieved in routine software development.

5 Development Tools

This section briefly discusses the tool support that was available when developing the application. When compared to the number of tools that are on the market for traditional languages, those available for Haskell look quite inadequate.

Three tools were used to support the program development:

- the hbc compiler and interpreter from Chalmers University, Gothenburg, Sweden;
- the space profiler from the University of York.

The former was used to compile and test the Haskell program. The latter being used to identify and improve the space usage of the program.

Most program errors were caught by the type checker and no great need was felt for a debugger/tracer. In fact none was available[1]. The ease with which components can be isolated meant that logical program errors that were not caught by the type checker could be identified and removed using the interpreter to test expressions interactively.

The space profiler [5] is a tool for analysing the space usage of a program. It was used only by the expert developer and proved invaluable for improving the efficiency of the final program; some large space leaks being found and eradicated. Some knowledge of how lazy functional programs are executed is needed when fixing space leaks and so novice functional programmers may not obtain the full benefits from using the profiler. Rewriting functions to remove space leaks can affect the clarity of the functional program solution if an algorithm needs to be rewritten to remove a space problem. The efficiency benefits are likely to outweigh this problem, however, and comments in the code or design document should be sufficient to make clear what is going on.

Other common support tools such as language–directed editors and design tools that support Haskell are not available, nor is it clear that they are needed. As Haskell develops, and if it gets taken–up by industry more widely, then support tools will appear but these are likely to be in the area of program transformation (e.g. [6]) and support for parallel functional program design (e.g. [7]).

6 Conclusions

The key features of the development in Haskell of a workforce management system have been presented. A solution by an expert has been contrasted with another implemented by a novice Haskell programmer. Haskell was found to be an adequate language for this application but experience is needed to exploit functional style fully. In particular, the use of Haskell supports many current software development paradigms more easily than currently popular programming languages:

- algebraic data types make code clear and easy to understand;
- encapsulation and abstraction are supported naturally in the language;
- secure code is promoted by the use of higher–order functions with polymorphic types;
- laziness can be used to improve the efficiency of a solution.

[1] Experimental facilities for debugging and tracing are available in the latest release of hbc available from animal.cs.chalmers.se:/pub/haskell/chalmers/lml-0.998.5.*

There are a number of disadvantages incurred using Haskell:

- some of the useful features are not yet efficiently implemented;
- tool support is still in its infancy.

The inexperienced use of the language manifests itself in

- failure to program in a result–driven declarative style;
- not exploiting the power of functions as first–class objects;
- giving inadequate consideration to types of functions;
- not appreciating the role of lazy evaluation.

Training and experience in (lazy) functional programming is essential before programmers can use it well.

7 Acknowledgements

We are grateful to Dr. Colin Runciman at the University of York for his many helpful comments in reviewing an earlier draft of this paper.

References

[1] P Hudak, S Peyton-Jones, and P Wadler (Editors). Report on the programming language Haskell, Version 1.2. *ACM SIGPLAN Notices*, 27(5), 1992.

[2] J Hughes. Why functional programming matters. *Computer Journal*, 32(2), April 1989.

[3] R Bird and P Wadler. *Introduction to Functional Programming.* Prentice Hall International, 1988.

[4] D Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15, 1972.

[5] C Runciman and D Wakeling. Heap profiling of lazy functional programs. Technical Report YCS172, University of York - Department of Computer Science, (To appear in the Journal of Functional Programming) 1992.

[6] M Firth. *A Fold/Unfold Transformation System for a Non-Strict Language.* PhD thesis, Department of Computer Science, YCST 90/09, University of York, 1990.

[7] J Darlington et al. Structured parallel functional programming. In *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, 1991.