Understanding LOLITA: Experiences in Teaching Large Scale Functional Programming

Stephen A. Jarvis[1], Sanjay Poria[2] and Rick G. Morgan[2]

[1] Oxford University Computing Laboratory, Wolfson Building,
Parks Road, Oxford, England.
Stephen.Jarvis@comlab.ox.ac.uk
[2] Dept of Computer Science, Durham University,
South Road, Durham, England.
Sanjay.Poria@durham.ac.uk
R.G.Morgan@durham.ac.uk

Abstract. LOLITA is a large scale natural processing system written in the functional language Haskell. It consists of over 47,000 lines of code written over 180 different modules. There are currently 20 people working on the system, most of whom are Ph.D. students. The majority of research projects involve the development of an application which is written around a semantic network; the knowledge representation structure at the core of the system. Because of the type of various applications, developers often join the team with little or no functional programming experience. For this reason the task of teaching these developers to the level required to implement their respective applications, requires teaching at various levels of abstraction. The strategy chosen means that each researcher only needs to be taught at the particular level of abstraction at which they work. These abstractions give rise to the notion of a domain specific sublanguage; that is a programming style in which a different language is created for each desired level of abstraction. In this paper we show how functional languages provide the necessary framework to enable these sublanguages to be created.

1 Introduction

The LOLITA (Large-scale Object-based Linguistic Interactor Translator and Analyser) system [2] is a state of the art natural language processing system, able to grammatically parse, semantically and pragmatically analyse, reason about, and answer queries on complex texts, such as articles from the financial pages of quality newspapers. Written in the pure, lazy functional programming language Haskell [5], it consists of over 47,000 lines of source code [3] (excluding comments).

The semantic network, which is the system's central data structure, contains over 90,000 nodes, allowing more than 100,000 inflected word forms. Begun in 1986, when the language Miranda[3] was used, the system is being developed by the Laboratory for Natural Language Engineering at the University of Durham,

[3] Miranda is a trademark of Research Software Ltd.

currently involving a team of approximately twenty developers. In June 1993 the LOLITA system was demonstrated to the Royal Society in London.

LOLITA is an example of a large system which has been developed in a *lazy* functional language purely because it was felt that this was the most suitable type of language to use. It is important to note the distinction between this development, where the choice of lazy functional languages is incidental, and projects which are either initiated as *experiments* in lazy functional languages or have a vested interest in the use of lazy functional languages. There are many examples of the latter, the FLARE project [14], the Glasgow [11], and Chalmers [3] compilers; there are substantial examples of the former, the LOLITA project is one of the larger of these developments [3].

2 The LOLITA Natural Language Processing System

Many Natural Language Processing (NLP) systems have been built to solve specific problems. These systems are restricted, either in the particular task they perform or the domain in which they work. The aim with LOLITA is to produce a general, domain-independent knowledge representation and reasoning system.

LOLITA's core consists of a language independent representation for natural language in the form of an inheritance based *Sem*antic *Net*work; SemNet (see [8] for a more in-depth discussion). Statements are represented as a collection of nodes and arcs, signifying concepts and relationships respectively. This representation provides a rich and expressive formalism for any natural language sentence.

The core system can parse complex text, conduct semantic and pragmatic analyses of the resulting parse graphs/trees and add the relevant conclusions to SemNet. The system can also answer natural language (NL) interrogations about the knowledge held in the network by generating natural language from SemNet.

Built around this core are the various applications shown in Figure 1. An example of one of the applications is template generation. This involves the identification of relevant information contained within ordinary text such as newspaper articles. The relevant information is presented using a template. The template contains various slots which act as field headings, whose bodies are filled in according to the content of the original text. For example a suitable template for *meetings* might contain the slots: *participants*, *when* and *where*. LOLITA identifies the information to fill these slots.

3 Teaching Functional Languages

Most of the applications of the LOLITA system are Ph.D. research projects, and are built around the core system. The types of project are extremely diverse ranging from Chinese language tutoring to the generation of causal explanations

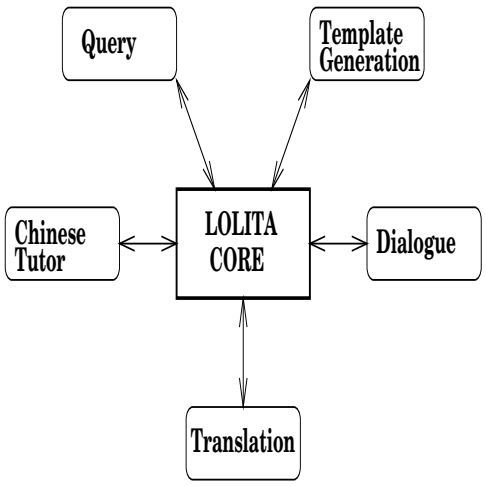

Fig. 1. A diagram showing the various applications built around the LOLITA core.

by abduction from the network. Consequently the backgrounds of researchers in the team varies greatly, from linguistics and mathematics to computer science and economics.

As expected, the programming experience of people coming into the team varies greatly from those who have no programming experience, people who have experience of imperative languages to those who are proficient functional programmers. In addition some developers have no experience of simple mathematical notions.

Since most projects in the team involve implementing some application it is important to teach functional programming such that each person is adequately equipped to implement their component.

One possibility is that, all developers on the project are brought up to the same level of functional programming expertise. This would require teaching all aspects of functional languages. On a project this scale, such an approach has proved to be infeasible because of the problems caused by varying backgrounds and abilities of the various developers.

Instead the approach taken is to try and minimise the amount of teaching that needs to be done by teaching functional programming at various abstract levels. It will be shown that functional programming languages provide features which enable these levels of abstraction to be created.

3.1 Problems of Teaching Functional Programming

LOLITA is an example of an Artificial Intelligence system. Speed of response to the required queries is of importance. The production of efficient code is hence essential. This provides a difficulty since the traditional paradigm of functional

languages (also of logic based languages), is that the user declaratively describes *what* the function does but now *how* the function computes. Put in other words they try to abstract away from the more concrete representation of the various structures required to compute the result, to a more abstract notion of computation.

The problem which arises in teaching the various personnel is to find a balance between teaching a minimal amount such that the particular person may implement their application and teaching an adequate amount such that the implementation is efficient. Whereas there is a temptation to teach as little as possible and at a level of abstraction that the person will be working at, it is offset with the need to teach low level issues such as compiler details and models of reduction, to understand where inefficiencies typically occur. It is essential that the teaching must find the correct balance between the two on a large project. Consequently tools have been developed to enable this. One example of such a tool is a profiling tool [9] [7] which enables users working at the subsequent levels of abstraction to monitor the time efficiency of the code easily without requiring an in depth knowledge of other parts of LOLITA and Haskell system.

4 A Development Hierarchy for Functional Programmers

From our experience the natural solution to finding a balance between teaching at the right level of abstraction and detail of efficient implementation is to develop a hierarchy of programmers. This hierarchy has naturally developed into the levels shown in Figure 2.

The figure shows abstractions created by developers at various levels in the hierarchy, together with typical levels of expertise. People at each level will provide support to those higher up (either by creating abstractions or providing tools). An abstraction at the lower level in the hierarchy may hide developers from primitive I/O operations and access to other parts of the system implemented in `C` or `C++`.

In the rest of this paper we explain why certain features of functional languages are particularly suited to the creation of these levels of abstraction.

5 Suitability of Functional Languages

We use particular examples as a case studies, but stress that this abstraction approach can also be applied more generally. This section looks more specifically at aspects of functional languages which provide us with the desirable features mentioned above.

Consider the case of a typical level 2 to level 3 abstraction in our scheme (where level 1 is the lowest level in the table i.e. the Haskell language level). A developer at level 3 may be working on the parsing/grammar of natural language; that is, they are interested in the resulting parse graphs produced by variation of the grammar rules. A developer at this level is less interested in a specific

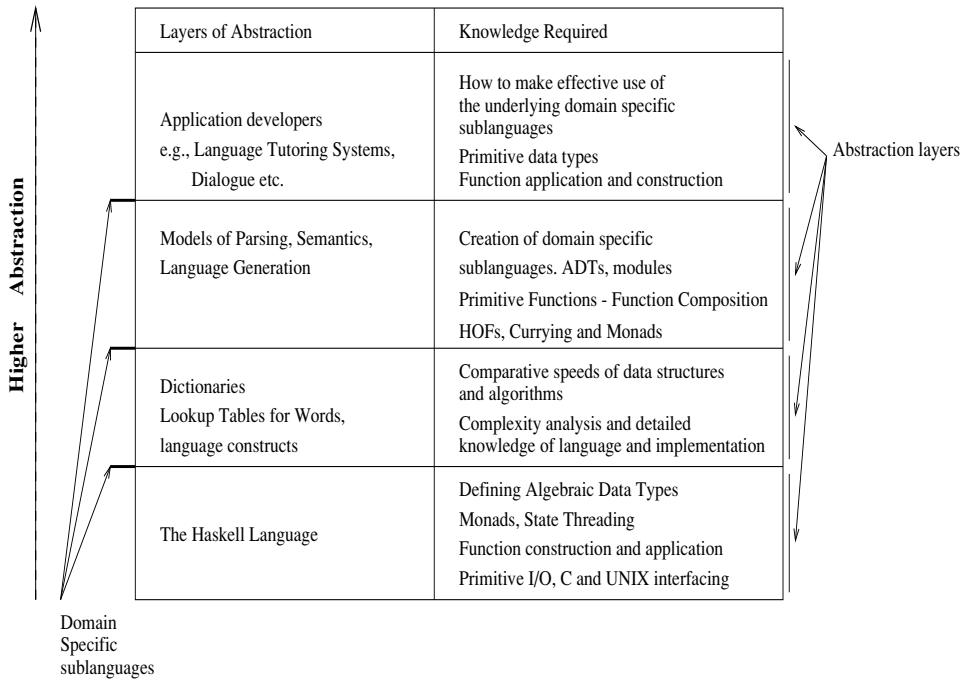

Fig. 2. A diagram showing the various levels of abstraction that exist. The column on the left shows the layers of abstraction, based on the Haskell language, at which applications are developed. People developing at lower layers support those at higher layers by providing tools and/or creating a level of abstraction. The second columns shows the knowledge that those working at each level require. Each level of abstraction can be bridged by the use of a DSS. The real power of Functional Languages lies in the way in which the boundaries between these levels of abstraction can be drawn.

types of detail (e.g. efficiency or how the grammar is used to actually produce a parse) but they are concerned with the effects of these grammar rule variations. For such a person we wish to create a different type of abstraction to a person who is working at the level of the natural language dictionary where efficient word lookup is paramount for the effective working on the system.

5.1 Domain Specific Sublanguages

Most people would accept that the choice of an ideal programming language for some task would greatly depend upon that task. That is, there is no notion of a universal programming language which is well suited to all programming tasks. We might therefore be left with a choice between designing and implementing a new language which is well suited to the particular domain in which we wish to

work, or using a language which does not allow us to express ourselves directly in terms of this domain, and has already been designed and implemented with wider goals in mind.

A solution which lies somewhere between these two extremes is provided by the use of what we call *Domain Specific Sublanguages* (DSS). We build such languages as collections of Haskell types, operators and functions, and so in a sense they are not a new language at all. However, they are designed in such a way that the programs written using them, not only look unlike "normal" haskell programs, but correspond closely to the important concepts in the domain in which our problem lies. So for example we provide a domain specific sublanguage for writing grammars which hides any details of how the grammar might be used to actually parse sentences.

We will illustrate with case studies that if well defined, a domain specific sublanguage can appear to the programmer to be a new language specially tailored to the required level of abstraction. We also try to show that lazy functional programming languages are ideal candidates to enable the creation of these sublanguages.

5.2 Case Study 1: The LOLITA Grammar

Perhaps the best example of a DSS currently in the LOLITA system is the NL grammar. This may be because DSS's are particularly suited to domains which rely on large numbers of rules which have similar structure. An example of one such grammar rule is shown in Figure 3.

```
> reported_sentence :: Parser
> reported_sentence
>     = prephrases
>       +++
>       sentence &? excl_mark     >> exclamN
>       +++
>       sentence & ques_mark      >> questN
```

Fig. 3. An example of one of LOLITA's grammar rules. `exclamN` and `questN` specify the labels to be placed on the resulting parse tree nodes, as well as the feature aspects (such as number and tense).

Although this rule is Haskell code it has a close correspondence with the standard formalisms for describing grammars. For example Figure 4 shows the grammar rule shown in Figure 3 in a more traditional BNF (Backus Naur Form).

As can be seen there exists a natural mapping of operators between the two formalisms.

```
reported_sentence
     =  prephrases |
        sentence ?excl_mark |
        sentence ques_mark
```

Fig. 4. The grammar rule in 3 in BNF form.

This example fully exploits the powerful abstraction mechanisms provided by functional languages by providing a DSS. The example above is a particularly good case because:

- The mapping from original BNF grammar to the domain specific language is particularly natural.
- The domain is totally enclosed. The user of such a language has not been given any opportunity to revert to the full complexity of Haskell.
- We have moved away from Haskell specific details like how to combine and apply functions to totally syntactic issues of how to combine terminal and non-terminal symbols. Combining such symbols requires only a simplified view of complex functional programming aspects such as the type system (see Section 8).

5.3 Case Study 2: The Semantic Parser

The semantic parser is a central feature of the LOLITA system (see Figure 5).

The input to the semantic parser is a syntactic parse tree built at a previous phase in the system. The output from the semantic parser is the corresponding semantic network structure. The fundamental task therefore is the transformation from the parse tree structure to the semantic network data type. Consider for example the parse tree representing the sentence "Roberto owns a motorbike", and its conversion to the corresponding piece of semantic network, shown in Figure 6.

Each node in the parse tree is labelled with its grammatical construct. For instance the root node of the parse tree is labelled with `sen`, representing the complete sentence structure. Each of these labels has a corresponding semantic rule which transforms the parse tree structure into the semantic network structure. Rather than coding these rules directly, we have defined a language which is used to specify these rules in an abstract way. This language has been modified as our comprehension of what is required by semantic analysis has changed and developed.

Two types of semantic rule are used, one for parse tree leaves and the other for branches. The parse tree leaf rules are represented by the abstract data type (ADT) 'leaf_rule' and are given as part of the definition of a function `meta_leaf` which maps parse tree labels onto the corresponding leaf rule. In a similar way

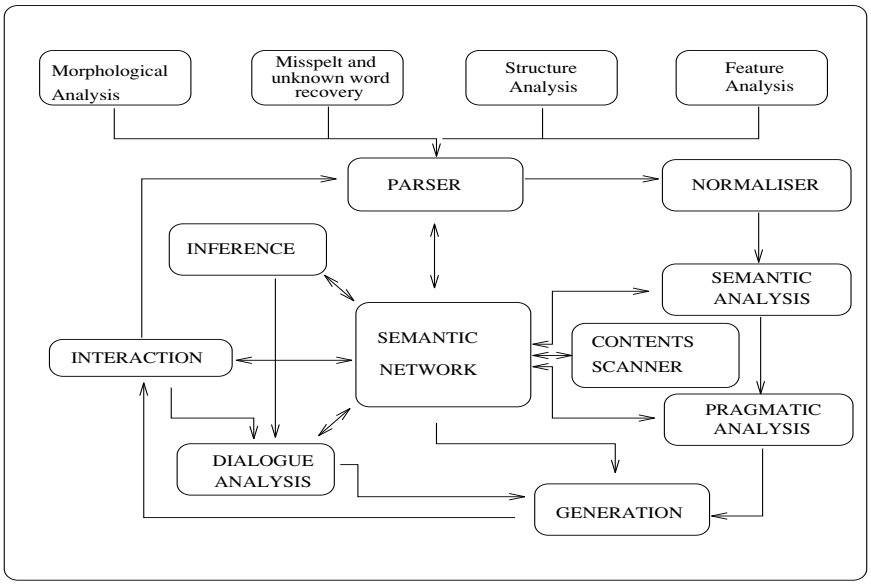

Fig. 5. A diagram showing the components of the LOLITA core.

the parse tree branch rules are represented by the ADT 'branch_rule' and are given as part of the definition of a function `meta_branch`.

```
meta_leaf :: ParseTreeLabel -> LeafRule
meta_branch :: ParseTreeLabel -> BranchRule
```

The semantic representation of a binary node in the parse tree is mainly determined compositionally according to its label and the semantics of the subtrees below it. Taking the `transvp` node of Figure 6 as an example, the left subtree produces the concept of ownership and the right subtree produces the concept of a particular (but unspecified) motorbike. The fact that these are linked by a `transvp` branch means that the 'ownership' must be an action and the motorbike must be an object. This rule is specified as follows:

```
> meta_branch "transvp"
>  = labelboth Act Obj
```

Although the rule for `transvp` can define the semantic representation for a node entirely in terms of the semantic representation of the subtrees, other rules must take into account contextual information such as the set of referents[4] that are available. It must also be possible to mark points in the parse tree which may correspond to new nodes in the semantic network as well as points which may

[4] Nodes which may be referred to in later pieces of text by pronouns (e.g. 'he' or 'it')

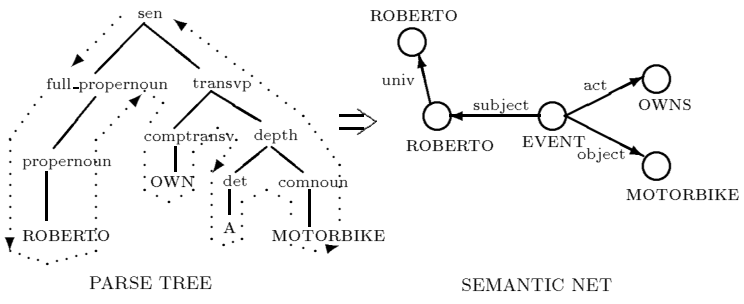

Fig. 6. An example of the task of the semantic parser part of LOLITA. The input (a parse tree) is transformed into a section of semantic network (the knowledge representation structure).

be referred to later in the text. To provide these facilities, additional operators and functions are provided.

The operator `compose` combines two rules and applies the first rule to the result obtained from the application of the second (it is the semantic rule equivalent of function composition). The following example creates a branch-rule for proper nouns by combining three smaller rules: `labelLeft`, `newnode` and `addref`.

```
> meta_branch "full_propernoun"
>  = addref           `compose`
>    newnode object   `compose`
>    labelLeft Univ
```

This rule specifies the semantics for `full_propernoun` as a unique new object node related to the semantics of the left subtree by the universal link (as indicated in Figure 6, `full_propernoun` branches contain no right subtree). It also ensures that this new node is available as a referent (this would be used in the sentence "Roberto owns a motorbike and *he* cleans it almost every day"). The new node is necessary here to distinguish between the unique 'ROBERTO' being talked about in the example and the universal concept of 'ROBERTO'.

This short example illustrates how we use domain specific sublanguages to create a level of abstraction.

The developer who is working on transforming parse trees to corresponding semantic network fragments need not have internal knowledge of the representation of `BranchRule`'s and `LeafRule`'s and how they are further used to actually build the fragment of semantic network (this is hidden in an ADT). The person is actually working at a conceptual *meta-level*. The developer only needs to worry about *what* the rules at each level in the parse tree should be. Also note that

although `BranchRule` and `LeafRule` are implemented as functions, the semantic rule writer, need have no knowledge of the higher order aspects of Haskell needed to implement `` `compose` ``.

Abstract types therefore provide a framework for creating domain specific sublanguages and as we see in this example, there may be one or many abstract types needed to implement a domain specific sublanguage.

6 Supporting Domain Specific Sublanguages

Although declarative languages provide an abstraction away from the machine representation this is precisely what is needed for some tasks in a diverse system such as LOLITA which has many components.

The user is shielded from the complexity of the solution. For example, for a long time it was thought that the grammar of the system was adequately written. However, it was not until we obtained a graphical tool to display parse graphs (a graph of all possible parse trees of a sentence) that it was realised that parsing contributed a significant amount to the space problem that was being encountered. It was later found to be a problem caused by the way in which the rules had been written.

It was soon realised that people were required to support the development of the system at the lowest level indicated in the hierarchy. This has lead to the development of debugging[4] and profiling tools [9] as Ph.D. projects. These developers need to be aware of all the details of functional languages including graph reduction techniques, program optimisations and transformations carried out by the compiler. Part of their object is to shield people who develop components higher up in the programming hierarchy from implementation issues, and also to present debugging and profiling information at the required level of abstraction, (rather then at lower levels). Without these techniques it is not possible to monitor the complexity behaviour of the programs at higher levels of abstraction without detailed knowledge, hence defeating the object of creating DSS's. These projects are looked at in more detail.

6.1 Debugging Domain Specific Sublanguages

Since each developer works at a particular level of abstraction and is not expected to know details of the mechanism underlying the particular DSS, aspects such as debugging code can present a problem. In the case of an error one of two possible cases has occured:

1. Either, the error has occured at the abstract level at which the developer is working (e.g. a DSS function applied to illegal arguments), or,
2. the error has occured at some lower level.

What needs to be determined is which of these two cases has occured. To aid in this search a debugging tool has been developed [4].

A class of run-time error which was found to be occurring frequently in developers code was the *exception error* type. An exception error is one which results in termination of the program and the printing of an error message. Examples of this type of error in Haskell are

```
Fail: head{PreludeList}: head []
```

which results from passing the list head function an empty list and

```
Fail: (!!){PreludeList}: index too large
```

which occurs when the list indexing operator is passed a subscript which is outside the bounds of the list. These exception errors give the applications programmer no indication of whereabouts in the program the error occurred. In particular the errors are displayed with reference to the lowest level in the system; the list function defined in the standard prelude.

This is a problem as functions such as `head` and the list indexing operator are used many times in many different parts of LOLITA. Previously, this problem had been approached by providing a customised version of each function capable of generating an exception error for each module. This new version of the function would report the name of the module when it generated an exception error. However simply knowing the name of the module in which the exception was generated is not sufficient—the error causing the exception error to be generated may be in a function which called the exception-generating one, or even some way back in a chain of functions each calling the other with the exception-generating function at the very end of the chain. Reporting errors in this way is of no use in the determination of errors at the level of a domain specific sublanguage.

The *distinguished path debugging tool* allows the display of chains of functions, termed *distinguished paths*. The path displayed is the route taken through the dependency graph of the functions in the program. The tool works by transforming each function to take an extra parameter, a representation of the distinguished path, which is built up from one function call to the next. When an exception error is encountered, the value of this extra parameter is displayed. Unlike the previous method of debugging such errors, which involved altering the source code by hand, the tool works automatically by transforming each source module.

Using such a tool it is possible to display the cause of errors at a higher level in the programming hierarchy. Though the error may have been caused by a list index out of range, this is of little help to a developer working on spell checking input text. However, it may be more useful if they are able to see that one of their functions (case 1 above) caused such an index out of range, (in which case it is the developers job to investigate the error) or the error is further down the chain, below the DSS (a case 2 error). In this latter case the developer may have to consult the person who works at this lower level so it can be fixed.

6.2 Profiling Domain Specific Sublanguages

During the development of the LOLITA system attention has been paid to the efficiency of the code. A large amount of time has been spent profiling the system with both the Glasgow cost centre profiler [10], part of GHC, and also the York heap profiler [12] [13], supplied with the Chalmers haskell compiler. Improvements to the overall system achieved through the use of the profilers have been in the order of 35%.

The profiling task requires the parts of the code that the programmer is interested in to be identified. This process can take place automatically, the compiler will select functions, modules or constructors to profile at compile time, or alternatively this selection can be made by hand, by annotating the code.

Using hand annotated code to identify functions for profiling it is possible to profile the functions of the domain specific sublanguage abstract data types. Perhaps more importantly it is possible for the applications programmer to shield themselves from the lower level functions in the system by profiling just the modules in which they program.

The analysis of the profiling results may be limited if the programmer is unaware of precise details relating to the functional language. For instance they may not understand why a space leak manifests itself or why functions are not evaluated lazily when this is the aim. However, the profiling results do allow them to compare and contrast different implementations of their application functions and to see which appears to be more efficient.

Work has been done at Durham to develop a profiling tool which provides the programmer with detailed profiling information in an interactive post-processing environment[5]. A modified version of the Glasgow Haskell Compiler produces profiling results based on *stacks* of functions. These stacks are used to recreate the function call-graph. The programmer can interact with functions in this call-graph to gather costs at different levels in the program. Since the profiling costs are recorded in this stack form, the inheritance of costs to high levels in the call-graph is accurate; no statistical averaging is used for shared functions.

6.3 Optimising Domain Specific Sublanguages

One of the drawbacks with the domain specific sublanguage approach is the lack of any facilities to provide off-line processing of the sublanguage. Such a scheme would become particularly important in situations where the code in which the domain specific sublanguage is written could be optimised and transformed at compile time. The development of the grammatical analysis offers a typical example of where this would have been useful.

In order to achieve efficient parsing of natural language, a number of transformations were performed on the original grammar to make it largely deterministic [1]. The grammar was originally coded in its deterministic form but this was found to destroy the structure of the grammar and it became difficult to

[5] The *cost-centre stack* profiler, [9].

maintain. It was therefore decided that the transformations should be applied to the grammar.

The transformations were were performed by a three stage process. First, the original Haskell form of the grammar was parsed, secondly the transformations were applied, and finally the results of the transformations were built using the parsing ADT with additional deterministic constructs contained within it. Using the ADT to code the source of the grammar meant that the grammar could be tested without having to wait for the transformations to be applied. Given that the transformations typically took three hours, this made the maintenance of the grammar far more practical. The efficiency of the untransformed grammar meant that it could not be applied to long and complex sentences; however it could be applied to sufficiently small sub-parts.

The current situation could be considerably improved if the compiler were able to provide a mechanism to perform the stages of grammar transformations as part of the compilation process. The grammar operators could then be implemented as constructors which would build a data structure representing the grammar (this contrasts with the present implementation in which grammar operators are functions). Supposing this was represented by the type NewParser:

```
> data NewParser = NewParser :+++ NewParser |
>    NewParser :>> ParserName |

etc.
```

The `parse` function which actually performs the parsing would then have the type:

```
> parse:: NewParser -> Input -> Output
```

and would be implemented as

```
> parse p i = (transformGrammar p) i
```

The function `transformGrammar` would produce the actual parsing function, and would be evaluated at compile time.

This type of facility would then ensure that such off-line preprocessing could be performed more easily than at present, preventing the need to parse the original Haskell code and generate and compile new Haskell code. It is interesting to note that many Prolog implementations provide a facility to transform the source code as it is loaded or compiled into the Prolog system. Although this facility can be used to perform the type of off-line transformations discussed here, it has the disadvantage that it can change the semantics of the initial source program. Using compile time evaluation would provide a means of performing such off-line transformations, but without changing the semantics of the program.

7 Support for the Construction of Domain Specific Sublanguages

It is certainly possible to use this domain specific sublanguage approach in other languages. Most modern languages provide facilities for the creation of abstraction data types – these are essential to the use of DSS's as they prevent the user of the sublanguage from accessing the implementation of the types used in the sublanguage directly. The ability to define operators with specified precedence and associativity is also provided in other languages.

However, certain features of functional languages make them particularly suitable for the implementation of DSS's. These are:

Higher-Order Functions. In the use of domain specific sublanguages, the programmer will often apply a function to a value without realising that these values are also functions. For example in Figure 3, `+++` and `>>` are thus higher order functions as they take values of type `Parser` as parameters. The result of using this form of representation is to make functions written using these abstract types easier to write and clearer to read. They enable a more natural mapping between the original rules the user may wish to enter and the form that Haskell the created Haskell sublanguage will accept.

Lazy Evaluation. Once an abstraction into a DSS is created it needs to be interfaced to the rest of the system. Lazy evaluation provides an essential mechanism that enables this integration [6]. For example, consider the conditional function `cond` taken from the Generator DSS of LOLITA

```
> cond :: GenCond -> Gen -> Gen -> Gen
> sayEvent
>   = cond hasObject
>          sayObject
>          sayNothing
```

The function `cond` evaluates either its second or third argument depending upon the value of the first. In a strict language an interface to this function is not practical because it requires the evaluation of both the *then* and *else* parts of the conditional.

Another example can be found in the parser. A typical piece of grammar will look like:

```
> paragraph = sentence & (empty +++ paragraph)   >> parLabel
```

If the operators `+++, &` and `>>` are all strict in both arguments, any attempt to evaluate `paragraph` leads to non termination (or a "black hole" error) since it requires an evaluation of `paragraph` to return a result. This in practise is avoided because the operator `+++` is non strict in its second argument.

8 Teaching and Domain Specific Sublanguages

We have presented a methodology for the development of large scale functional systems. This consists of:

- The various Domain Specific Sublanguages which make up the system; the case studies presented show examples of these.
- Tools which make these abstractions practical. These include the profiling, debugging and potentially the optimisation of domain specific sublanguages.

Consequently, this presents a structured paradigm for teaching of functional programming. Developers are taught an application specific DSS, Typically this means teaching a greatly simplified functional language typically consisting of the functions available at the application specific DSS (with relevant types), together with:

1. basic function definitions: (un)guarded expressions. . . ,
2. associated typing rules: (`f::t`, simplified view of type system excluding aspects such as currying,
3. function application,
4. simple module interfaces - `module X (`$f_1 \ldots f_n$`) where import...`

This avoids teaching advanced aspects of functional programming such as algebraic data types, higher order functions, monads and interfacing to other languages. . . .

Also we gain the normal benefits given by abstract data types. Common functional programming ideas such as list comprehension and currying can be hidden from the application programmer. The amount of actual Haskell that needs to be taught is therefore only a small subset of the actual language.

At an optimal level in the system a developer need only know the functions of the supporting domain specific sublanguage and Haskell operations such as function application which enables them to combine these functions for their own development. Even powerful ideas like higher order functions can be hidden from the programmer because they are explicitly built into the domain specific sublanguage.

It must be stressed that although the strategy is to teach a minimal amount (i.e. the DSS and those constructs listed above only), often developers will teach themselves a greater range of features. However, examples where developers have only learned the provided sublanguage do exist.

9 Conclusion

In this paper we have presented a different paradigm for the teaching of functional programming for large scale systems such as LOLITA. That is the creation of problem dependent abstraction levels via sublanguages where required. This paradigm of development has advantages over other development processes,

where the full complexity of the language is used at all levels in the code. Hence our developers do not have to show proficiency at all levels in programming techniques. Although it may seem desirable to have all developers showing proficiency in programming techniques at all levels of functional languages, it has been found to be infeasible in large project such as the ours. This arises mainly as a result of the scale of the project, tight time constraints on teaching, and the varying abilities and backgrounds of the members joining the team.

In choosing our own natural syntax and hiding levels of detail in an ADT we are able to create a domain specific sublanguage for a particular task. The approach we have developed has a number of well established traits:

- **Flexibility** — By using a sublanguage of Haskell, the syntax and semantics of the language may be changed simply. This is something that has frequently occurred within the development of the LOLITA system.
- **Power** — These domain specific sublanguages are extremely useful when we can find an appropriate set of constructions that cover all of rules we want to describe without becoming overly complex. However, we do find in a substantial rule set that there are often some rules which need special treatment. In our approach it is easy to revert to the full power of Haskell (and lose some abstraction at isolated points). In the external language approach, this would be far more difficult to achieve, and either requires the special construction of "one off" primitives, or some facility to interface with a more powerful language.
- **Scale** — The overheads in setting up new language tools mean that the external approach is really only feasible for substantial rule sets. The internal approach has a very small overhead and is thus applicable to much smaller rule sets.

The development of the LOLITA system over a number of years, by a number of people with mixed programming experience, has given us a large amount of functional programming experience. The significance of this work highlights not only how, features of functional languages alleviate the burden of teaching a wide spectrum of functional techniques to developers of large-scale real world system, but also, with the need for developing advanced programming tools, where such a scheme fails.

Acknowledgements

Our thanks to Dave Nettleton and Paul Callaghan for reading earlier drafts of this paper.

References

1. N. Ellis, R. Garigliano, and R. Morgan. A new transformation into deterministically parsableform for natural language grammars. In *Proceedings of 3rd International Workshop on Parsing Technologies*, Tilburg, Netherlands, 1993.

2. R. Garigliano. LOLITA: Progress report 1. Technical Report 12/92, School of Engineering and Computer Science, University of Durham, 1992.
3. A. Gill and P. Wadler. Real world applications of functional programs. http://www.dcs.gla.ac.uk/fp/realworld.html.
4. J. Hazan and R. Morgan. The location of errors in functional programs. *Lecture Notes in Computer Science*, 749:135–152, 1993.
5. P. Hudak, S. Peyton Jones, P. Wadler, et al. *Report on the Programming Language Haskell Version 1.2*, 1992.
6. J. Hughes. Why fuctional programming matters. *The Computer Journal*, 32, 1989.
7. S. Jarvis. Profiling large-scale lazy functional programs. PhD thesis, Durham University, *forthcoming*.
8. D. Long and R. Garigliano. *Reasoning by Analogy and Causality: A model and application*. Artificial Intelligence. Ellis Horwood, 1994.
9. R. G. Morgan and S. A. Jarvis. Profiling large-scale lazy functional programs. In *Proceedings of the Conference on High Performance Functional Computing*, Denver, USA., April 1995.
10. S. L. Peyton Jones. Implementing lazy functional programs on stock hardware: the spineless tagless g-machine. *Journal of Functional Programming*, 2:127–202, 1992.
11. S. L. Peyton Jones et al. The Glasgow Haskell Compiler: a technical overview. In *Framework for Information Technology Technical Conference*, Keele, 1993.
12. C. Runciman and D. Wakeling. Heap profiling of a lazy functional compiler. In J. Launchbury and P. Sansom, editors, *Functional Programming*. Springer-Verlag, 1992.
13. C. Runciman and D. Wakeling. Heap profiling for lazy functional programs. *Journal of Functional Programming*, 3, April 1993.
14. C. Runciman and D. Wakeling. *Applications of Functional Programming*. UCL Press, 1995.

This article was processed using the LaTeX macro package with LLNCS style