In P. Fritzon (ed), Poster Proc. CC'96 - Int. Conf. Compiler Construction, TR 96-12, Comp. Dept., Linkoping Univ., 49-56, 1996.

On the Systematic Development of Compilers: A Case Study

Carlos H. C. Duarte

Department of Computing
Imperial College, London
cd7@doc.ic.ac.uk

Roberto Ierusalimschy

Departamento de Informática
PUC, Rio de Janeiro
roberto@inf.puc-rio.br

Abstract. As more and more computing platforms appear and the number of features embedded in programming languages grows, dimensions like complexity and size acquire a fundamental importance in compiler development. In order to treat these issues, this paper proposes a systematic method combining the object oriented and the transformational approaches for compiler development. This method is illustrated here by the development of a compiler for a functional language, showing that modular and extensible products can be obtained as the result of an organisation of the development process due to the use of these both approaches.

Keywords: Compiler construction, object oriented approach, program transformation, functional languages.

1 Introduction

Nowadays, as more and more computing platforms appear and the number of features embedded in each programming language grows, dimensions like complexity and size seem to acquire a determining importance for the successful development of new compilers. Without treating these dimensions, compilers become hard to develop and maintain.

Surprisingly, these dimensions have not received great attention. In general, correctness and efficiency have been considered the only relevant concerns in compiler design. However, as observed by Goguen [4], correctness formalisms alone led to several unsucessfull experiences. Regarding efficiency, it is believed that modularity and abstraction should be initially considered more important, as argued by Rothweel [12].

An important step in this direction has been made by the proposition of the object oriented approach for developing this kind of software, as initially introduced by Koskimies [10] and recently recalled by Crenshaw [2]. By using this approach, it is possible to treat the high number of programming language dependent characteristics exhibited by compilers through the definition of classes and methods. In this way, modularity and abstraction are supported directly by the constructs of the design approach. However, correctness and efficiency have not been an issue in these proposals.

In the present paper, these issues are treated by a systematic method for compiler development. Explicitly, by using a transformational approach for compilation [7], the problems concerning correctness can be handled in the proper formal setting without given out efficiency. Moreover, by providing an object oriented design derived from these specifications, in a style similar to those mentioned above, modularity and extensibility can be guaranteed. In fact, it appears that the deficiencies of each of these approaches are precisely supplemented by the advantages of using the other.

The proposed method has been validated by the development of a compiler for a high level functional programming language, based on the ideas initially proposed by Peyton-Jones [7]. In this case study, the full complexity of the functional languages family and most re-

lated problems have been treated satisfactorily.

The remainder of the paper is organised as follows: the next section describes the proposed method; then, some aspects of the developed compiler are presented; in the last section, some final comments are drawn.

2 A method for compiler development

The proposed method relies on the formal specification of the programming language in order to obtain the design and implementation of the compiler. Adequate data structures are derived from the language syntax, while compiler functionality is derived from the formal semantics. Rules are given in the sequel to guide these developments in a systematic manner.

2.1 Syntax

In the first place, the specification of a programming language grammar is used to construct a lexical and syntactic analyser. In the present work, it is required that these grammars are specified by sets of BNF rules without ambiguity, allowing the use of generators like *lex* and *yacc* [1].

Subsequently, the specification of the grammar is also used in the definition of the data structures which will represent the programs handled by the analyser. In [2], a basic rule is provided to derive such structures using the object oriented approach. There, it is suggested that it is only necessary to look at the syntactic equations to identify the objects required by the translator: every non-terminal would represent a class. However, such rule turns out to be correct only if it is required that the elements of the abstract syntax of the language are all represented as non-terminals.

An illustrative example is obtained by considering the following productions, which describe how lists may be organised in a functional programming language:

list → *string*
list → *head_tail_list*
list → *empty_list*
list → *enum_list*
list → *list_compr*

By using the rule proposed above, *list*, *string*, *head_tail_list*, *empty_list*, *enum_list* and *list_compr* should be represented at execution time by objects belonging to the classes *List*, *String*, *HeadTailList*, *EmptyList*, *EnumeratedList* and *ListComprehension* respectively.

Apart from symbol names, there are other relevant information in the specification of every grammar. These are meant to represent the structure of symbols, and therefore should point out the implied class structure.

The definition of several productions having the same symbol on their left hand side means that this symbol can be used in a more general context to substitute those appearing on the right hand side. This characteristic is easily captured by the inheritance concept. In other words, the classes representing the strings on the right should be sub-classes of that class representing the symbol on the left in all these productions. In the previous example, *String*, *HeadTailList*, *EmptyList*, *EnumeratedList* and *ListComprehension* should be sub-classes of *List*.

Presumably, a production wherein symbols are defined by aggregation give rise to a class composed by the aggregation of the classes representing the constituent symbols. From the following example [1],

head_tail_list → *expression* CONS *expression*

a class *HeadTailList* should be created containing two occurrences of *Expression* to play the roles of head and tail of the list respectively.

Table 1 summarises these rules. In addition, other rules such as the generalisation of common components can be used to obtain a better class hierarchy, with less elements. For instance, grammar rules describing a list can be represented by a generic list class. Further details can be found in [3].

[1] Terminal symbols appear in uppercase letters.

Grammar	Representation
terminal	attribute
non-terminal	abstract class
production	concrete class
many rules for a symbol	sub-class
many non-terminals in a rule	aggregation

Table 1: Derivation rules using the grammar.

2.2 Semantics

For defining the formal specification of the programming language semantics, the transformational approach [7] is prescribed. Transformation schemes are used to describe the meaning of each symbol in terms of their translation into other symbols with a well known meaning (normally belonging to other language). The notation used here is similar to that presented in [7, 8], where each scheme has a signature and a definition in the following generic way:

scheme : $dom\_symbol \rightarrow img\_symbol$
scheme$[src\_pattern_1]$ $(param_1) \triangleq trg\_pattern_1$
$\vdots$
scheme$[src\_pattern_n]$ $(param_n) \triangleq trg\_pattern_n$

Patterns are composed by constants, variables or symbols and must satisfy the syntax of the domain and the image of each scheme. In addition to these, parameters and other scheme names may also appear in the definition of each target pattern.

Since there is no restriction over the derivation rules previously introduced concerning their language applicability, these rules can also be used to produce a representation of the target language of the translation process, or, in other words, the language of the target patterns above. In fact, this application results in the decomposition of the translation process into methods belonging to the classes of the source language, which show as their result objects of the classes representing the target language. In this way, it is possible to explore the potential of the polymorphism supported by the object oriented approach.

The main representation rule for semantics takes a scheme in the generic form above to define a new method in each of the classes representing the source patterns. Moreover, the scheme signature is used to define the interface of the respective method and the target pattern is used to define the method body. At execution time, if a sentence matches with some of the source classes, then the respective target pattern, coded as a method body, is used to define the resulting translation. This representation is particularly propitious, since one of the characteristics of transformation schemes is their polymorphism, and the same happens with methods organised by an inheritance hierarchy.

Figure 1 can be used to exemplify this representation rule. The figure shows the production and the schemes which specify the syntax and semantics of lists defined in terms of their elements. The meaning of the respective symbol is given by function **TE**. If a sentence satisfying this symbol is an enumeration, its translation will be given by the translation of its component called *contents_list* (due to the first rule in the definition of **TE**); if it is a list defined by properties of elements, then its meaning will be given by the scheme **TQ** (as defined by the second rule). In this example, **TE** should be represented by the distinct forms the method `TE` assumes in each of the classes *DefinedList*, *EnumeratedList* and *ListComprehension*, which produce as their result objects belonging to *LambdaExpr*.

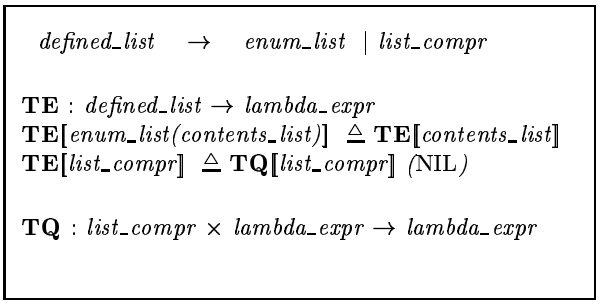

Figure 1: Specification of element defined lists

There are cases in which a transformation scheme needs to use other information, apart from the components of the sentence on which it has been applied. This is the case of **TQ**

in Figure 1, which is applied on a *list_compr* element and also on a *lambda_expr* element resulting from a previous application of the same scheme. As it can be seen in the figure, this recursive process of applying **TQ** starts by using the empty list. This scheme should be represented by a method with parameters: `TQ` is a method of *ListComprehension* receiving objects of *LambdaExpr*.

In defining translation schemes, decisions should be ultimately made by the designer. For instance, if it requires the specification of an scheme acting on a set of symbols that are not directly related by any rule of the language grammar, the representation of this scheme will be dissociated from any class of the derived hierarchy. Cases like this are also treated by the method and will be presented in Section 3.1.

3 A functional language compiler

Recently, functional programming languages that are higher order, lazy and polymorphic have attracted great attention to themselves and their compilation process. In this class are Miranda[2] [14], Haskell [9], A [11] and others. For these reasons, the development of a compiler for **Carmen**, a variant of Miranda, was chosen as a case study.

The overall organization of the compiler is detailed in Figure 2. In the picture, it is possible to note that programs handled during the translation process can be represented in two forms: textual or as an object tree. The second representation can be obtained from the first through the syntactic analysis of the program text, while the opposite transformation is performed by pretty-printing the corresponding tree.

Each program given as an input to the compiler is translated consecutively in other equivalent representations written in lower level languages. They are called **Carmen**, **Lamb**, **Comb** and **Gcode**, respectively. Each of these languages is represented as a class which aggregates the hierarchy of classes representing the language structure as derived in the previous section. At the end of this translation process, the obtained program can be executed by the proper interpreter called **GMachine**.

[2]Miranda is a trademark of Research Software Limited.

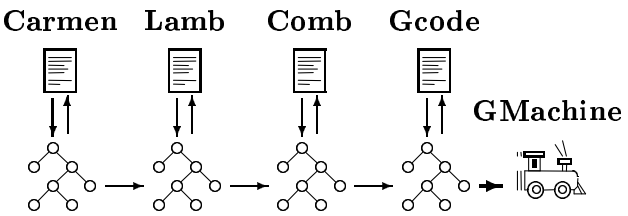

Figure 2: Organizational view of the compiler

3.1 Carmen and its compilation into Lamb

A functional program is normally called a *script*, and consists of a list of statements in the form of equations. Each equation partially specifies a function by providing a sub-set of the domain and by defining how the resulting image is to be computed for values in this set. A typical script is presented in Figure 3.

```
norm (x,y) = sqr ((sq(x) + sq(y)) / 2)
fat 0 = 1
fat n = n * fat (n - 1), n > 0
mod x = norm (x,x)
main = fat (mod (-4))
```

Figure 3: **Carmen** example program

Patterns are used to define the application domain of each function. They are composed in a way similar to that described in the Section 2.2. When patterns exist, they are declared after the function name. In Figure 3, after the symbol `=` in each equation, many so called "guarded terms" may appear, composed by a single expression followed by an optional condition called guard. For example, `n * (n - 1)` has `n > 0` as its guard. When a guard is evaluated as true, the respective equation is used to compute the value of the function.

The scheme **TF** presented in Figure 4 defines the semantics of statement lists (*stmt_list*). It establishes that the translation process of a certain script into a **Lamb** object tree is given by the application of **TS**, **TF** and **TR** on the components of the program. The representation of this scheme is the method `TF` of the class *StatementList*.

TF : *stmt_list* → *lambda_expr*

$$\mathbf{TF}\left[\!\left[\begin{array}{l} \textit{stmt\_list1} \\ \textit{stmt(}\mathbf{main}\ \textit{guard\_list stmt\_list2)} \end{array}\right]\!\right] \triangleq$$

TDA[.] *(***TS[***stmt_list1***]** *,*
TR[*guard_list***]** *(***TF[***stmt_list2***]** *))*

Figure 4: **TF** translation scheme

In some cases, methods representing transformation schemes dissociated from any derived class need to be interleaved in between other method applications. This is the case of **TDA** in the previous example, which is represented as a method of **Lamb**. This is used to perform a dependency analysis on the terms that are to constitute a definition with delimited scope. The result of such an analysis is an expression in which the given terms are positioned only where they will really be necessary.

The translation of functions defined by cases may also be represented by such kind of method. The need for this translation appears because functions defined by cases may be programmed in two different manners: through guarded expressions or by a set of equations having constant patterns. A typical example is the factorial function, which can be programmed in the following ways:

```
fat n = 1, n = 0
      = n * fat (n - 1), otherwise

fat 0 = 1
fat n = n * fat (n - 1), n > 0
```

Since both programs have the same meaning and the first form is required by subsequent translations, constant patterns are removed from case-based definitions of functions.

```
let norm = ^_2.
    let _5 = (SELECT (1, 2) (1, 2)) _2
     in let _4 = (SELECT (1, 2) (1, 1)) _2
         in sqr / + sq _4 sq _5 (1, 2)
 in letrec fat = ^_0.
           IF ~ == _0 (1, 0)
              * _0 fat - _0 (1, 1)
              IF == _0 (1, 0)
                 (1, 1)
                 FAIL
        in let mod = ^_1.norm ((0, 2) _1 _1)
            in fat mod - (1, 4)
```

Figure 5: Example program written in **Lamb**

Another method of **Carmen**, called `Match`, is used in the translation of each script in order to perform pattern matching compilation. This creates a lambda expression to encompass all the equations defining the same function. The resulting expression is able to determine which equation will be used to compute the image of the actual function, based on the matching between call values and formal parameters. A detailed definition of the respective translation scheme is described in [7].

3.2 Compiling Lamb into Comb

Lamb is an extension of the lambda calculus. In this language, programs are composed by abstractions (functions), represented by the symbol `^`, and also by variables, constants and applications. Statements with a delimited scope can also be created using either `let`s for non recursive definitions or `letrec`s. The previous **Carmen** script translated into this language is shown in Figure 5.

Both in **Lamb** and in lower level languages, constants are represented as pairs, where the first component stores the type of the value and the second component the value itself. In Figure 5, `(1, 4)` represents the integer 4. Type constructors are also represented in this way, as `(0, 2)` which denotes a tuple type constructor with arity equal to 2. This uniformity avoids a problem identified in [8]: "Compilers sometimes implement the built-in types in special "magic" ways, and the programmer pays

a performance penalty for user-defined types".

The translation of **Lamb** programs uses a technique called *lambda lift* in order to remove the free variables from each expression, so that an efficient code can be subsequently generated. For each abstraction, a new global function is defined having as parameters, apart from the abstracted variables, those which appeared free in the abstraction body. In the places where it had been used, such an abstraction is substituted by the application of the resulting function on the variables that have become parameters. Using this technique, after translating `norm` into function `$3`, the abstraction `^_1.norm ((0, 2) _1 _1)` is translated into

```
$2 _1 = $3 ((0, 2) _1 _1)
```

The **TLL** translation scheme specifies an enhanced lambda lifting. Not only free variables but also entire sub-expressions that do not have bound variables are removed from each abstraction. In this way, waste of time and memory is avoided by evaluating each expression only when this is really required. This characterises the described compiler as a so called "fully lazy". By applying the respective method to the abstraction `_2`, a function with three parameters is created, where `(SELECT (1, 2) (1, 2))` and `(SELECT (1, 2) (1, 1))` have also been substituted by variables.

During expression evaluation, it may happen that definition bodies inside `let(rec)`s have to be re-computed unnecessarily, implying in the loss of the fully lazy characteristic. To avoid this, another technique known as *float out* is used, removing from definition bodies the statements that are not essential where they appear enclosed. This technique is applied before lambda lifting, and it is specified by the scheme **TFO**. While the internal structure of only one `let` definition is treated by **TDA**, the scheme **TFO** treats in a recursive manner all kinds of lambda expressions.

Both **TFO** and **TLL** have a complex and extensive definition by cases, for each of the symbols of **Lamb**. However, the inherent complexity of these schemes is divided and treated

```
$2 _1 = $3 ((0, 2) _1 _1)
$1 _0 = IF ~ == _0 (1, 0)
           * _0 $1 - _0 (1, 1)
           IF == _0 (1, 0)
              (1, 1)
              FAIL
$3 = $0 (SELECT (1, 2) (1, 2))
        (SELECT (1, 2) (1, 1))
$0 _2, #0, #1 = let _5 = (#0 _2)
                    in let _4 = #1 _2
                        in sqr / + sq _4
                                   sq _5 (1, 2)
main = $1 $2 - (1, 4)
```

Figure 6: Example program written in **Comb**

by using the polymorphism of the representing methods. The method `TLL` in an object of *Application* only needs to self-apply itself on each of its two own components, without the need for knowing who they are. Further details concerning these schemes and their representation can be found in [3].

3.3 Comb and GCODE

Comb is a language analogous to **Lamb**, wherein declaration of super combinators is permitted. According to the definition in [7], a super combinator $S is a lambda expression in the form $\lambda x_1.\lambda x_2....\lambda x_n.E$ such that E is not an abstraction, $S does not have free variables, the expression E is a super combinator and $n \geq 0$.

Super combinators provide another representation for functions, eliminating from the language the explicit use of abstractions. The previous example program appears in Figure 6 translated into **Comb**, where super combinators are represented as

```
$S x1 x2 ... xn = E
```

The use of super combinators allows functional programs to be compiled into an efficient code, based on the reduction, at execution time, of the graphs representing the functions computed by the program. These reductions are performed by a stack machine called **GMachine**, which interprets the corresponding language **GCODE**.

F : *combinator* → *g_program*
F[*code_node var_expr_list lambda_expr*] ≜
  globalstart *code_node*
    Length[.] *(var_expr_list);*
  R[*lambda_expr*] *(var_expr_list)*

Figure 7: **F** translation scheme

The translation of each super combinator is specified by scheme **F** in Figure 7. It defines a context for the execution of each combinator associating stack positions to parameter names. From this point, methods representing other schemes like **R** are applied recursively in a polymorphic way. At the end of this process, the previous function `fat` is translated into a **GCODE** program with 43 instructions.

3.4 Implementing the compiler

Implementing the compiler becomes straightforward by using the proposed method. Each rule in the specification of the language gramar, which is to be given as input to an *yacc* compatible generator, should give rise to a rule following one of the patterns below:

```
non_terminal : component_1 ... component_n
   { $$ = new NonTerminal ( $1, ..., $n ); }

non_terminal1 : non_terminal2
   { $$ = $1; }
   | ...
```

By using C++ [13] as the programming language, in the first case above each semantic action of the parser specification should be defined as a `new` followed by the constructor of the respective class. C++ also makes trivial to implement the classes derived using the method. To illustrate this fact, the interface of the class *ListComprehension* introduced in Section 2 is presented in Figure 8.

```
class ListComprehension : public DefinedList {
   ExpressionList *qualifier;
   public:
   ListComprehension (ListContentsList *c,
                      ExpressionList *q);
   int GetId (void);
   void Save (FILE *f);
   int Print (FILE *f, int indent, char *sep);
   static ProgramSymbol *Load (FILE *f);
   LambdaExpr* TE (void);
   LambdaExpr* TQ (LambdaExpr *L);
};
```

Figure 8: Interface of *ListComprehension* class

4 Final Remarks

Recently, growing concerns have been observed regarding the organizational aspects of compiler development. Even when useful techniques are applied alone, as the transformational specification of programming languages, problems have appeared because of size and complexity issues. It appears to exist a consensus in trying to organize this kind of product in an improved manner; this is confirmed by [8, 11, 12]. On the other hand, the object oriented approach alone as proposed in [2, 10] provides a satisfactory treatment for these issues but sometimes lacks in correctness. It was to conciliate these views that the method described in the present paper was proposed.

Generally speaking, the method produced very satisfactory results in application cases, organizing the development process and showing that it is adapted to common practices. From the BNF specification of the each grammar, class hierarchies organized by inheritance and aggregation are created. Transformation schemes are used in the derivation of methods for each of these classes, exploring the polymorphism of both design and implementation languages. These derivations follow precise rules and guide the creation of all language dependent components of the compiler. To provide language independent functionality, a reusable framework was defined.

To validate this method, the development of a functional compiler was chosen. In this tool, several techniques for program transformation have been integrated, such as dependency analysis, pattern matching and fully lazy lambda lifting. The implementation of

this compiler (and the **GCODE** interpreter) has nearly 16000 lines of code written using C++ and the specification languages of *lex* and *yacc*. This is a portable product that can be executed using MS-DOS[3] and UNIX[4]. Such a product is as efficient as other implementations based on the same compilation techniques, nevertheless it is organised in a more modular and organized manner, preserving the full clarity of the operational specifications provided. Initial work using denotational specifications in an automatized manner is described in [5].

To evaluate the extensibility of the products created using this method, the incorporation of new functionality to each compiler should be considered. In our example, though techniques as type checking or strictness analysis were not mentioned, they could be easily introduced by annotating the respective trees with proper information [3]. For the former, the functionality needed is orthogonal to translation and can thus be defined using complimentary approaches like [6]. However, this is not the case for strictness analysis: it would result in additional transformation schemes. Consequently, our framework offers support for compiler extension through the connection of new functionality derived either using the method described or sometimes not. We believe that facts like this, in addition to the high modularity reached, reinforces the advantages of using an object oriented approach in a rigorous manner, in particular for compiler development.

Acknowledgements: This work has been partially supported by CNPq, The Brazilian Research Council. The paper was written while the first author was visiting Hewlett Packard Laboratories, Bristol, U.K.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] J. W. Crenshaw. A perfect marriage. *Computer Language*, 8(6):44–55, June 1991.

[3] C. H. C. Duarte. The development of a Miranda compiler using an object-oriented method. Master's thesis, Department of Informatics, Pontificial Catholic University of Rio de Janeiro, Brazil, July 1994. In Portuguese.

[4] J. A. Goguen and M. Moriconi. Formalisation in programming environments. *IEEE Computer*, pages 55–64, November 1987.

[5] L. C. C. Guedes, E. H. Haeusler, and J. L. Rangel. Object-oriented semantics-directed compiler generation – A prototype. In *6^{th} Conference on the Theory and Practice of Software Development (TAPSOFT'95)*, 1995.

[6] R. Ierusalimschy. A denotational approach for type-checking in object-oriented programming languages. *Computer Languages*, 19(1):19–40, 1993.

[7] S. L. P. Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[8] S. L. P. Jones. Implementing lazy functional languages on stock hardware: The Spineless-Tagless G-machine: Version 2.5. Department of Computing Science, University of Glasgow, July 1992.

[9] S. L. P. Jones and P. Wadler. A static semantics for HASKELL. Department of Computing Science, University of Glasgow, May 1991.

[10] K. Koskimies. Software engineering aspects in language implementation. In *Proc. 2^{nd} CCHSC Workshop*, volume 371 of *Lecture Notes in Computer Science*, pages 39–51. Springer Verlag, October 1988.

[11] S. L. Meira, M. A. Musicante, and A. Santos. The design and implementation of language A. In *Proc. V Brazilian Symposium on Software Engineering*, pages 237–256, October 1991. In Portuguese.

[12] N. Rothwell. Functional compilation from the standard ML core language to lambda calculus. Technical Report ECS-LFCS-92-235, Department of Computer Science, University of Edimburg, September 1992.

[13] B. Stroustrup. *The C++ Programming Language*. Prentice-Hall, 2^{nd} edition, 1992.

[14] D. Turner. An overview of Miranda. *ACM SIGPLAN Notices*, 21(12):158–166, December 1986.

[3] MS-DOS is a trademark of Microsoft Corporation.
[4] UNIX is a trademark of AT&T Bell Laboratories.