A Structured Functional Programming Language

Nadia Nedjah and Luiza de Macedo Mourelle

Department of Systems Engineering and Computation, Faculty of Engineering,
State University of Rio de Janeiro,
Rio de Janeiro, Brazil
{nadia, ldmm}@eng.uerj.br
http://www.eng.uerj.br /~ldmm

Abstract. Declarative languages provide means of specifying problem solutions, without giving much attention to how the proposed solution is going to be effectively implemented. When only specification is provided, the system attempts execution using search techniques which may compromise efficiency. In this paper, we propose a new programming language that integrates functional and logic features together with procedural ones. In normal cases, this hybrid language should provide the programmer with advantages of both i.e., *efficiency* and *clarity*. Using this language, very rapid prototyping is possible.

1 Introduction

Purely functional and logic languages are ideal for developers of large-scale packages as they allow the programmers to easily prove the soundness of the built software. Declarative languages have many advantages, which include [7]: the soundness of declarative programs can be easily proven; the exploitation of parallelism in declarative programs is greatly simplified; the clarity of declarative programs is re-enforced. However, as purely functional and logic programs only specify the solution of a given problem without providing any details of the corresponding implementation, compilers of these programming languages are normally slow compared to compilers of imperative languages. Furthermore, when actually trying to write programs, it is often highly inconvenient to be limited to declarative features only. This is mainly due to efficiency of evaluation.

Programs are bound to have an algorithmic details and specifications. In all main line programming languages these are inextricably linked. This is as much true of logic and functional programming languages like Lisp, Prolog, Miranda [18], Haskell [5, 16], etc. as of procedural and object-oriented ones like Pascal, C, Java, C++ etc. By separating the two, it is possible to have

different algorithms implementing the same function definition with an automatic guarantee of partial correctness, rather than two obscure function implementations that still have to be verified with respect to a specification in a different language. In reality, this claim is a slight exaggeration, but it does point to an overall (unattainable) objective in a programming language, which is as far away in procedural languages as in declarative ones. The specifications, which a program uses cannot be truly abstract since functions defined using unbounded quantification may not terminate and so need not be executable. One needs to insist on computable definitions.

In this paper, we propose a programming language which is hybrid. It offers declarative and imperative features and thus yields the advantages of both: *clarity* and *efficiency*. The proposed language attempts to cover most of the life cycle of a program which is required to be both *verifiable* and *efficient*. It enables the systematic development of a program starting with the construction of a formal specification to which more and more details can be added and results in a correct and relatively fast implementation. With less effort, one can write programs which consist of executable code and neglect the abstract specification level by the inclusion of specific algorithmic information. In such a case, very rapid prototyping is possible. We will call our language *plog*.

We first describe the general characteristics of the proposed language as well as the overall structure of *plog* programs. As a new programming language is best conveyed through examples, we will do it through an illustrative example. After that, we detail the most essential features of *plog* while giving a glance at the syntax of the language. We then give some implementation issues of the language compiler. Finally, we conclude the paper.

2 The Language General Structure

A *plog* program is an extension of C to a functional programming language in which C is the command language. Thus a C program is a *plog* program. More specifically, a *plog* program is a C program in which various functions are defined not using C, but using a functional programming style to calculate their returned values. It is assumed that the reader has knowledge of C and that therefore the main interest is in the definition of the functions.

The operational semantics behind the functional programming paradigm is now well-known [3, 4, 7, 14, 17]. It consists of using a set of equations considered as left-to-right rewrite rules to simplify a given term, called the

subject term. Starting from this term, the evaluation process produces a sequence of expressions by repeatedly replacing instances of left sides of rules with their corresponding right sides until no further replacements are possible. An instance of a left side in the subject expression is called a ***redex***, and an expression with no redex is said to be in ***normal form***. The ***pattern-matching*** process provides a rule whose left side matches the expression considered [8-13]. When various redexes can be identified by the pattern-matching process, one redex has to be chosen to be reduced. This choice is made based on the reduction strategy in use. There exist several reduction strategies such as ***eager*** vs. ***lazy*** and ***bottom-up*** vs. ***top-down*** strategies [15]. The most commonly used strategy is called the ***outermost-leftmost strategy*** also called ***normal order***.

A ***plog*** program is an environment composed of a set of possibly nested ***modules*** followed by the ***main*** body of the ***plog*** program. Nesting of modules is achieved by importing them through the directives ***uses*** and ***refines***. Each module is the implementation of an abstract data type and is composed of two main parts: a ***specification*** of the abstract data type and the ***implementation*** of the corresponding functions.

- The specification part of a module is mainly a ***declaration*** part. The first part of a module specification includes a ***type name***, ***variables***, ***constructors*** and ***functions*** declarations. A constructor is a function that does not have any implementation. It is used to construct an object of a given abstract data type. For instance, ***empty*** and ***:*** in Example 2.1 are constructors. An object stack that contains the integers 1, 2 and 3 that were pushed in this order, would be represented as 3:2:1:empty. A function is declared by its name, the list of its parameter types and the type of the value returned.
- The implementation part of a module is composed of the ***procedures*** that implement the functions declared before. Each procedure can declare new objects i.e., types and variables. The body of the procedure is composed of a list of rewriting rules followed by a set of ***directives*** allowing the compiler to decide which rule to use next. To make the language more expressive conditional rules are allowed. Each rule of a given procedure is associated with a ***rule strategy*** that helps to reduce the work needed to the select the next redex. The directives may specify the reduction strategy to reduce terms or choose one of the known reduction [15].

A module can be parameterised as is illustrated in the module description of Example 2.1. This raises the degree of expressiveness of the proposed language.

Example 2.1: Consider a data type that describes the use of a stack. In Figure 2.2, we show the header of the module as well as its constructors and functions.

```
MODULE StackType(itemType) IS
     TYPE stack;
     CONSTRUCTORS
     stack empty;
     stack itemType:stack;
     FUNCTIONS
     stack      push(ItemType);
                itemType   pop(stack);
                Boolean    emptyStack(stack);
```

Figure 2.2. *Header and specification part of a module*.

Figures 2.3(a) and 2.3(b) describes the *StackType* module indicated in Figure 2.2 functions. For the operator, the user can suggest a reduction strategy. For instance, the strategy [1, 2, 0] in procedure *push* suggests the reduction process of a term *push*(*x*, *y*), where *x* and *y* are stacks should first reduce the subterm which is substituted for *x* then that which is substituted for *y* and then attempt to reduce the whole term. The implementation of the reduction strategy suggested by the programmer is described in the Section 3.

```
     PROCEDURE emptyStack;
     VARIABALES
          stack      s;
          itemType   i;
     RULES
          1: emptyStack(empty)  = true;;
          2: emptyStack(i:s))   = false;;
     {
          BottomUp;
     };
     PROCEDURE push [1,2,0];
     VARIABLE
          stack      s;
          itemType   i;
     RULES
          3:push(empty,i)= i:empty;
          4:push(s, i)    = i:s;;
     {
          Lazy;
     }
```

Figure 2.3(a). *Module function implementation - procedures emptyStack and push.*

```
PROCEDURE pop  [1,0];
     VARIABALES
          stack s, s1;
          itemType i, j;
     RULES
          5: pop(empty)= null;
          6: pop(i:s)  = i;;
          7: pop(i:empty)  = i;  [1,0];
          8: pop(push(s, i)= i; [0];
          9: pop(s)=IF(s == empty) null
                    ELSE j
                    WHERE (s== j:s1);;
     {};
 };
```

Figure 2.3(b). *Module function implementation - procedure pop.*

The main body of a *plog* program is a set of C instructions which include assignments, conditional instructions like an *if-then-else* and *switch*, constructs allowing iterations as *while*, *do*, and *for* and functions calls. The values returned by a function call should be computed using the rewrite rules specified in the body of the procedure implementing that function. For instance, Figure 2.4 describes how a C program could use the functions of module *StackType* described in Figure 2.2.

```
void main() {
     stackOfFloat  = newStackType(float);
     stackOfStacks = newStackType(StackOfFloat);
     stackOfFloat s1, s2, s3 = empty;
     stackOfStacks ss = empty;
     float x, y, z;
     for (int i=0; i<10; i++) {
          scanf("%f %f %f", x, y, z);
          s1 = push(s1, x);
          s2 = push(s2, y);
          s3 = push(s3, z);
          ss = push(s1:s2, s3);
     }
}
```

Figure 2.4. *Program main body*.

3 Implementation Issues

The philosophy behind the *plog* language is to clearly separate algorithmic details, memory management and specification so that each of these aspects of program generation may be tackled independently. The execution of programs in this language is performed with greater and greater efficiency according as information is added. If only the specification is given then the system attempts execution using search techniques, but termination is not guaranteed to the same extent as when specific algorithmic information is given to the runtime system in some form. The semantics of the language guarantees how it is executed given this additional information and also defines the model of computation. However, in the absence of any control whatsoever, i.e. when only a specification is provided, as in OBJ [2] for example, the algorithm which searches for solutions is not determined. This is left to the implementer of the language compiler as there is no algorithm which will in general solve any problem that can be defined in the language. So, various heuristics are required which may well depend on the use to which the system is put.

One of the great advantages of separating control, specification and data structures is that they may be implemented one at the time in the program enabling rapid prototyping and alternative choices of control to be tested.

The runtime system [8-13] is a theorem prover, which builds up a number of theorems that are held available as long as is necessary [7-9]. It is these theorems, deduced from the original rules, which form the system's memory. They have the form of additional rewrite rules which give values for functions needed in future computations. Thus, the memory holds natural values with very clear meaning: there are no obscure variables with a complex relationship between them.

The algorithmic control in the form of rules which are added to the program may disturb the logic of the functional programming language *plog*. This then requires a theorem-proving capability to verify the rules against the specification. Any implementation of the language is required to do such theorem-proving as it is necessary to ensure that the program meets its specification or report what still needs to be proved to enable verification. Normally, such theorem-proving capabilities will be interactive since proving theorems is rather a hard and undecidable process [6]. However, the compiler does have a switch by which the necessary verification, which is costly in time, may be switched off. Totally verified sections of code are marked as such and the proofs filed for future reference. For efficiency, code is date stamped so that future changes to code will only invalidate a minimal part of an already existent proof. Previously verified code which is thus invalidated uses the recorded proof in an attempt to re-verify the code. The need for

verification can almost be avoided by using rewrite rules for the algorithmic information and copying them for the specification. Only termination and consistency properties then have to be checked.

Operationally, the evaluation of a given term using the suggested reduction strategy can be described by the following rewriting rules:

$$
\begin{aligned}
&\textit{evaluate}(t) && = \textit{rewrite}(t, \textit{strategy}(t))\\
&\textit{rewrite}(t,\varnothing) && = t\\
&\textit{rewrite}(t,\textit{cons}(0,l) && = \text{if}(\textit{match}(t), \textit{evaluate}(\textit{template}(t)), \textit{rewrite}(t, l))\\
&\textit{rewrite}(t,\textit{cons}(x,l) && = \textit{rewrite}(\textit{substitute}(t, x, \textit{evaluate}(\textit{argument}(t, x))), l)
\end{aligned}
$$

where function ***evaluate*** rewrites a term to its normal form, function ***rewrite*** reduces the given term using its top operator strategy, function ***strategy*** returns the given term's top operator strategy, function ***match*** determines whether or not the given term is a reducible expression, function ***template*** returns the contractum of the given term, function ***substitute*** replaces a given subterm with another, and finally, function ***argument*** returns a given argument.

This straightforward implementation can be improved using the technique of ***memoisation***, which consists of bookkeeping already evaluated terms. Marking evaluated terms improves the evaluation time by a great deal (see [7] for more details).

The language *plog* has the structure of a program written in a *C*-like language. Functions are defined using a functional programming style to calculate their values. A *plog* function can be implemented by one or more procedures using rewrite rules. With this functional programming language, the user has the same algorithmic control as with imperative languages. This allows him to write programs which are as efficient in terms of memory space and execution time as if he were using an imperative programming language. The added control should increase efficiency by avoiding the process of pattern-matching that selects the rule to use and the process of redex selection that chooses the next subterm to rewrite. These two properties make the proposed language provably as efficient as an imperative language. In *plog*, functions are defined in an environment made up of a number of abstract data types. This means that the types needed for the function arguments and values are not defined explicitly by constructing them from the basic built-in types such as Boolean and integer. They are constructed implicitly using the functions which operate on them. This includes the use of constant functions, i.e. functions which have no parameters.

4 Conclusion

In this paper, we proposed a new programming language that integrates functional and logic features together with procedural ones without detailing all the features available to the programmer. Normally, this hybrid language should provide the programmer with two advantages: efficiency due to the imperative constructs and clarity and provability due to the declarative features.

The compiler of the *plog* language is being implemented. We intend to evaluate the performance of such a compiler in comparison with purely declarative systems like the OBJ3 system [2] and a purely imperative language as Pascal.

References

1. Ehrig, H. and Mahr, B., *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*, Springer-Verlag, 1985.
2. Goguen, J. A. and Winkler, T., *Introducing OBJ3*, Technical report, SRI-CSL-88-9, Computer Science Laboratory in SRI International, August 1988.
3. C.M. Hoffman and M.J. O'Donnell, 'Pattern-matching in trees', *Journal of ACM*, pp. 68-95, January 1982.
4. C.M. Hoffman and M.J. O'Donnell, 'Programming with equations', *ACM TOPLAS*, pp. 83-112, January 1982.
5. Hudak, P. and Wadler, P., *Report on the Functional Programming Language Haskell*, Technical report, YALEU/DCS/RR656, Department of Computer Science, Yale University, 1988.
6. Klop, J. W., *Term Rewriting Systems From Church-Rosser to Knuth-Bendix*, International Colloquium on Automata, languages and Programming, vol. 443, pp. 350-369, Lecture Notes in Computer Science, Springer-Verlag, 1990.
7. Nedjah, N. *Pattern-matching automata for efficient evaluation in equation programming*, Ph. D. thesis, UMIST, University of Manchester Institute of Science and Technology, September 1997.
8. Nedjah, N., Walter, C.D. and Eldridge, S.E., *More efficient pattern-matching automata for overlapping patterns*, Ninth International Workshop on Implementation of Functional Languages, St. Andrews, Scotland, UK, pp. 341-350, August 1997.

9. Nedjah, N., Walter, C.D. and Eldridge, S.E., *Optimal left-to-right pattern-matching automata*, Sixth Conference on Algebraic and Logic Programming, Southampton, UK, Lecture Notes in Computer Science, Springer-Verlag Editors, vol. 1298, pp. 273-281, September 1997.

10. Nedjah, N. *Minimal deterministic left-to-right pattern-matching automata*, ACM Sigplan Notices, vol. 33, no. 1, pp. 40-47, January 1998.

11. Nedjah, N. and Mourelle, L.M. *Very Efficient pattern-matching for overlapping patterns*, Fifth International Workshop on Languages, Logic, Information and Computation, São Paulo, Brazil, July 1998.

12. Nedjah, N., Walter, C.D. and Eldridge, S.E., *Efficient automata-driven pattern-matching for equational programming*, Software-Practice and Experience, vol. 29, no. 8, 1999.

13. Nedjah, N. and Mourelle, L.M. *Adaptive pattern-matching for overlapping patterns*, Fifteenth International Symposium on Computer and Information Systems, Istambul, Turkey, October 2000.

14. Nedjah, N. and Mourelle, L., *Improving Space, Time and Termination in Rewriting-Based Programming*, The Forteenth International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems, Budapest, Hungry, Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence, Springer-Verlag Editors, to appear, June 2001.

15. Peyton-Jones, S. L., *The Implementation of Functional Programming Languages*, Prentice-hall, 1987.

16. Peyton-Jones, S. L., Hall, G., Hammond, K., et al., *The Glasgow Haskell Compiler: a Technical Overview*, Joint Framework for Information Technology, 1993.

17. R.I. Strandh, Compiling equational programs into efficient code, *Ph.D. Thesis*, The Johns Hopkins University, 1988.

18. Turner, D. A., *Miranda: a Non Strict Functional Language with Polymorphic Types*, ACM Conference on Lisp and Functional Languages, vol. 20. pp. 1-16, 1985.