PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

RF-Maple: A Logic Programming Language with Functions, Types, and Concurrency.

Paul J. Voda and Benjamin Yu.

Department of Computer Science, The University of British Columbia,
6356 Agricultural Road, Vancouver, B.C. Canada V6T 1W5

ABSTRACT

Currently there is a wide interest in the combination of functional programs with logic programs. The advantage is that both the composition of functions and non-determinism of relations can be obtained. The language RF-Maple is an attempt to combine logic programming style with functional programming style. "RF" stands for "Relational and Functional". It is a true union of a relational programming language R-Maple and a functional programming language F-Maple.

R-Maple is a concurrent relational logic programming language which tries to strike a balance between control and meaning. Sequential and parallel execution of programs can be specified in finer details than in Concurrent Prolog. R-Maple uses explicit quantifiers and has negation. As a result, the declarative reading of R-Maple programs is never compromised by the cuts and commits of both Prologs.

F-Maple is a very simple typed functional programming language (it has only four constructs) which was designed as an operating system at the same time. It is a syntactically extensible language where the syntax of types and functions is entirely under the programmer's control.

In combining the two concepts of R-Maple and F-Maple producing RF-Maple, the readability of programs and the speed of execution are improved. The latter is due to the fact that many relations are functional and therefore, do not require backtracking. We believe its power as well as its expressiveness and ease of use go a little beyond the possibilities of the currently available languages.

1. Introduction

Applicative programming languages are languages without side effects. They are either based on functions or predicates. The former are functional languages and the latter, logic programming languages. Since functions yield only one result, the expressive power and readability of functional programming languages come from the possibility of composition of functions. Composition of relations is not as readily available. Functional relations, such as $P\ (x,y)$ where there is exactly one y for each x, can be composed using the descriptions of Russell $R\ (\iota y P\ (x,y))$ [cf. Shoe]. In the case of proper relations, $P\ (x,y)$ need not be satisfied at all or it can be satisfied by many values of y. One can technically use the indeterminate descriptions of Hilbert $R\ (\epsilon y P\ (x,y))$ which can be read as " $R\ (y)$ is satisfied by a y such that $P\ (x,y)$ provided there is a such a y". Descriptions are, however, quite unreadable and one should introduce a function instead of a definite description and resort to an auxiliary variable $\exists y\ (R\ (y)\ \&\ P\ (x,y))$ instead of indeterminate descriptions. Note that the existential quantifier is only implicit in antecedents of clauses of Prolog [Kowa, Clar].

Relations, because of their nondeterminism, are often preferrable over functions. Yet many relations are functional and they should be replaced by functions in order to improve both the readability of programs and the speed of execution. The latter is possible because there is no overhead associated with backtracking. Due to the *or-nondeterminism* of relations, relation based programming languages can exhibit a wider scale of control behaviour than the functional languages. For these reasons there has been quite a few attempts recently to combine logic programming style with functional programming style [cf. Symp].

We believe that the programming language RF-Maple (RF is for Relational and Functional) blends nicely these two styles of programming. It is a union of two separately designed programming languages: R-Maple [Voda 1] and F-Maple [Voda 2].

R-Maple is a concurrent relational logic programming language which tries to strike a balance between control and meaning. Sequential and parallel execution of programs can be specified in finer details than in Concurrent Prolog [Shap]. R-Maple uses explicit quantifiers and has negation. As a result, the declarative reading of R-Maple programs is never compromised by the cuts and commits of both Prologs.

F-Maple is a very simple typed functional programming language (it has only four constructs) which was designed as an operating system at the same time. It is a syntactically extensible language where the syntax of types and functions is entirely under the programmer's control.

In combining the two concepts of R-Maple and F-Maple producing RF-Maple, we believe its power as well as its expressiveness and ease of use go a little beyond the possibilities of the currently available languages.

In this paper, we will first present the design principles of R-Maple in sections 2 to 3, and then in sections 4 to 5 , we will present the features of F-Maple, and finally in section 6, we will present the combination of the concepts of R-Maple with F-Maple to form RF-Maple. We have decided to discuss R-Maple and F-Maple separately because both of them have their own characteristics which are best explained independently. In combining the two languages we do not risk any collusion of concepts because RF-Maple is a true union of both languages.

2. Description of R-Maple.

Imperative programming languages are concerned mostly with control and complicated meaning functions are required to give meaning to programs. On the other hand, logic programs [Kowa], at least in theory, being fomulas of predicate calculus, direcly express the meaning, but like Prolog and Concurrent Prolog (hereafter referred to as C-Prolog), has limited control over the execution sequence [Shap 1, Clar]. R-Maple strikes a balance between these two ends of the scale by allowing sequential and parallel execution of predicates but still maintains that a program is closely related by a meaning function to formulas of predicate calculus. Like C-Prolog, R-Maple synchronizes parallel processes by distinguishing between the input and output variables. This turns out to be essential for the synchronization of concurrent processes as confirmed by C-Prolog. Thus the symmetry of some of the relations of Prolog is sacrificed. However, unlike Prolog, R-Maple has quantifiers and logical connectives. Quantifiers eliminate the need for cuts and commits, while connectives allow negation. A simple example is *Genm* (*lst* | *x*) which generates all elements *x* of the list *lst*. A predicate such as *Genm* with output variables is called a *generator* whereas a predicate without output variables is called a *test*. We use the vertical bar to separate the output arguments from the input arguments.

Genm (*lst* | *x*) **is**
    case *lst* **of**
        nil | **F**
    [*hd* ,*tail*] | *x* := *hd* **or** *Genm* (*tail* | *x*)

When *lst* is nil, then the generator fails (returns false). Otherwise, the head of the *lst* (i.e. *hd*) is 'assigned' to the output variable *x*. This assignment will be propagated by computation rules as explained below. Should the value *hd* be rejected, the execution will backtrack into the execution of *Genm* (*tail* | *x*) to generate successive values. This will become clearer in later sections. The declarative reading of this predicate is :

$$Genm\ (lst, x) \leftrightarrow \exists hd\ tail\ \ ([hd, tail] = lst\ \&\ (x = hd\ \ \vee\ Genm\ (tail, x)))$$

Note that the declarative reading of the assignment is just the logical identity $x = hd$ and the declarative reading of the sequential disjunction **or** is the logical disjunction $\vee$. R-Maple also provides for parallel disjunction **orp** with the same declarative reading. Similarly, both sequential and parallel conjunctions, ; and ||, have the declarative (logical) reading & . Although the parallel and sequential connectives have the same logical meaning, their behaviour is different since the operational rules of R-Maple are given by different transformation rules. Another example of generator is *Add* (*s* ,*t* | *x*) which has the declarative reading $s + t = x$.

3. Computations in R-Maple.

Before we describe how the control directs the execution of a R-Maple program, we first introduce a postfix operator !. When a program *C* is to be computed, it is placed into the scope of the operator ! which is called a process. *C* ! will then indicate a process that is ready to be executed. Computation is performed by applications of rewriting rules of the form $A \Longrightarrow B$ where both sides contain the operator !. For example, some of the transformation rules for logical connectives are :

$(A$ **or** $B)! \Longrightarrow A\ !$ **or** B
$(A$ **orp** $B)! \Longrightarrow A\ !$ **orp** $B\ !$
$\mathbf{F}\ !$ **or** $B \Longrightarrow B\ !$
$\mathbf{T}\ !$ **or** $B \Longrightarrow \mathbf{T}\ !$
B **orp** $\mathbf{F}\ ! \Longrightarrow B$
B **orp** $\mathbf{T}\ ! \Longrightarrow \mathbf{T}\ !$
$\mathbf{F}\ !\ ;\ B \Longrightarrow \mathbf{F}\ !$
$\mathbf{T}\ !\ ;\ B \Longrightarrow B\ !$

The first rewrite rule specifies that, for a sequential **or** , control is first passed to A. If A is reduced to false, the control will then pass on to B because of the third rule $\mathbf{F}\ !$ **or** $B \Longrightarrow B\ !$. In the second rewrite rule, during a parallel or, **orp** , control is passed to both A and B. That is, two processes are created to execute A and B simultaneously. The behaviour of **orp** is explained in the fifth and sixth rules. When one of the arguments is reduced to **F** !, it is deleted. (Note that there must be at least one process inside of *B* because of the second rule.) If one of the arguments reduces to **T** !, then the other argument is simply discarded, thus killing all the processes inside. The same principles apply to conjunctions. The rules of R-Maple are designed in such a way that there is at most one rule applicable for each process in the computed formula.

The rewriting continues until the program is transformed into the form where no rewriting rules are applicable. This can either fail to terminate, terminate normally (in the form **T** !), or remain deadlocked. We should note that the executing machine is not a full theorem prover and that if the program never terminates, it does not mean that the original program was not a theorem. (For instance: *P* **or** 3=3 will never terminate if *P* does not terminate although the declarative reading of the formula is true. But since the sequential **or** is used, the executing machine will try to compute *P* before starting to compute 3=3 and therefore the whole program will never terminate.)

We saw earlier an example of the generator *Add* (*s* ,*t* | *x*). For example, an invocation *Add* (3,5 | *x*) ! will be rewritten as $x := 8$!. Add is a *functional* generator. In general, a non-functional generator *G* (| *x*)! will be transformed into the form $x := a$! **or** *H* (| *x*) where *a* is the first value generated, and *H* (| *x*) is a generator for the rest of the values in case backtracking is required (i.e. when *a* is later *rejected*).

A typical setup for a generator is of the form

find x **in** $\{G\ (\ |\ x\ );\ T(x)\}$

This program has a declarative reading $\exists x\ \ (G\ (x)\ \&\ T(x))$. $G\ (\ |\ x\ )$ could be a functional generator, in which case, we obtain **find** x **in** $\{x := a\ !;\ T(x)\}$, and eventually $T(a)\,!$ because of tautology $\exists x\ \ (x = a\ \&\ T(x)) \leftrightarrow T(a)$. In case $G\ (\ |\ x\ )$ is a relational generator, we successively obtain

find x **in** $\{(x := a\ !$ **or** $H\ (\ |\ x\ ));\ T(x)\} \Longrightarrow$
  find x **in** $\{(x := a\ !;\ T(x))$ **or**
      $(H\ (\ |\ x\ );\ T(x))\} \Longrightarrow$ (1)
    find x **in** $\{x := a\ !;\ T(x)\}$ **or**
      find x **in** $\{H\ (\ |\ x\ );\ T(x)\} \Longrightarrow$ (2)
    $T(a)\,!$ **or find** x **in** $\{H\ (\ |\ x\ );\ T(x)\}$ (3)

That is, backtracking is done using computational rules only. These rewritings are justified by the distributivity of conjuction applied in (1), and by the quantifier splitting tautology

$\exists x \ (A \lor B) \leftrightarrow \exists x \ A \lor \exists x \ B$

applied in (2). Should the test $T(a)$ in (3) fail, the control will fall back into the backtrack search employing $H(\mid x)$. This should be obvious from the transformation rules for disjunction explained above. On the other hand, if the test $T(a)$! is satisfied the whole program is transformed to **T**! automatically erasing the backtrack program. Another example is the generator *Append* (*lst* 1,*lst* 2 | *result*) which appends list *lst* 1 to *lst* 2 to form the output *list* in result.

Append (*lst* 1,*lst* 2 | *result*) **is**
  case *lst* 1 **of**
    nil | *result* := *lst* 2
    [*hd* ,*tl*] |
      find *res* 1 **in**
        Append (*tl* ,*lst* 2 | *res* 1); *result* := [*hd* ,*res* 1]   (4)

R-Maple is more flexible in expressing parallel execution than C-Prolog. To execute the generator and the test in **find** x **in** $\{G(\mid x); T(x)\}$ in parallel, we can use the same expression with only one minor change; i.e. **find** x **in** $\{G(\mid x) \parallel T(x)\}$. Lazy evaluation can also be obtained with partially uninstantiated data structures, by switching the assignment and the recursive invocation of *Append* around in (4).

Computations of R-Maple are *invariant* to the declarative reading of programs. This is because each rewriting rule is justified by a logical tautology. In case of tests, computation employs the truth tables of logical connectives. In case of generators, an assignment $x := a$! reached by the control is propelled backwards through its enclosing connectives and quantifiers by relying on the associativity and distributivity of conjunctions and disjunctions until it reaches its associated quantifier. Some of the corresponding rewrite rules are as follows:

$(((x := a)!\,; A) \textbf{ or } B) \textbf{ or } C) \Rightarrow$
  $((x := a)!\,; A) \textbf{ or } (B \textbf{ or } C)$

$(((x := a)!\,; A) \textbf{ or } B); C) \Rightarrow$
  $((x := a)!\,; (A; C)) \textbf{ or } (B; C)$

find $y\,\{(x := a)!; A \textbf{ or } B\} \Rightarrow$
  $\{(x := a)!; \textbf{find } y \textbf{ in } A\} \textbf{ or find } y \textbf{ in } B.$

The last is possible only if y does not occur in the term a. Should y occur in a, the scope of the quantifier binding the variable y will be extended by pushing the quantifier back beyond the quantifier binding the variable x. The last is possible because of the tautologies:

$\exists y \ (x = a \ \& \ A \lor B) \ \& \ C \leftrightarrow$
  $\exists y \ \{x = a \ \& \ (A \ \& \ C) \lor (B \ \& \ C)\}$

$\exists y \ (x = a \ \& \ A \lor B) \lor C \leftrightarrow$
  $\exists y \ \{x = a \ \& \ A \lor (B \lor C)\}.$

provided y does not occur in C. If y occurs in C, it must be systematically changed to a different variable. There are more rules like these catering to all the possible combinations of sequential and parallel conjuntions and disjunctions. When the assignment reaches the quantifier, it is discharged by the following rules:

find x **in** $\{x := a\,!; A(x)\} \Rightarrow A(a)!$
find x **in** $\{x := a\,! \parallel A(x)\} \Rightarrow A(a)$

In the second case, the substitution of a for x will probably awake a process blocked on the execution of the statement:

case x ! **of** $\cdots$

Note that this blocking in **case** plays the same role as the use of read-only variables in C-Prolog.

We should mention here that there are no rewriting rules for guiding an assignment through a negation. This is because there is no good declarative reading for such a transformation. A program that attempts this will result in a deadlock. Moreover, there is no need for this in logic programs as the practice of Prolog confirms.

Thus R-Maple is a simple, purely declarative, logic programming language with explicit control over sequencing and parallelism. By the employment of logical connectives, the use of explicit quantifiers (*find*) coupled with the use of *case* statements, all the cuts and commits of Prologs can be eliminated. Moreover, a wide scale of control behaviours is now possible without compromising the declarative reading of programs.

4. Description of F-Maple.

F-Maple (F stands for Functional) is typed and provides, not only for semantic extensibility (new types and functions), but also for syntactic extensibility. The grammar of data types and functions is completely under the user's control. Schemes for data types specified by grammars have been proposed, among others, by [Kand] and [Malu]. F-Maple generalizes this approach by providing grammars for the specification of functions as well. Moreover, only four constructs are all that is needed, making F-Maple a simple but powerful functional programming language.

The basic types of F-Maple are *Number* and *String*. From these basic types, a user can define new data types by means of productions. For example, we can define the data type *Complex* which defines all complex numbers as follows:

Complex → *Number* + *Number* *i*

Similarly, to define the type for a list of numbers *Numlist*, we can express this new data type by:

Numlist → *nil*
Numlist → *head Number and tail Numlist*

Such productions are called the *generating* productions. The non-terminals occuring in *generating* productions are F-Maple types. Sentences produced from a non-terminal are values of the data type. For example:

head 2 *and tail* (*head* 4 *and tail* (*head* 6 *and nil*))

is a data value of type *Numlist* denoting a number list containing elements 2, 4, and 6. To improve readability, we allow the use of parentheses in data values to indicate grouping. They do not play any role either in syntax or semantics. Consequently, they should not be used as terminals. The sentence:

42 +35 *i*

is a data value of type *Complex* denoting a complex value with the obvious meaning. Enumerated types can be defined by productions which do not contain any non-terminals on the right hand side. For example, the type Bool specified by.

Bool → *true*
Bool → *false*

defines a type with just two values.

We have seen that the use of grammars at once specifies the data type and permits the concrete syntax to the type constructors. The user has complete control over the syntax. Ambiguous grammars are permitted in F-Maple. Rather than attempt to parse the basic values or terms specifying bodies of functions, we use an interactive *structure editor* to prompt the user for the value of the type needed at any moment. This also eliminates the need for the user to type in the long descriptive names as terminals because he simply enters the needed value to the production that he selects from the menu. It is apparent that the use of a grammar (or productions) gives the user a very powerful syntactically and semantically extensible tool for constructing types and their values.

Although the original F-Maple does not allow parameterized types, they can be easily added. We give one example here. The generic type constructor Bintree (T) which is the type for a binary tree whose nodes are of type T, can be defined by the following productions:

$$\begin{array}{l} Bintree\ (T) \rightarrow empty \\ Bintree\ (T) \rightarrow \\ \quad node\ \ T\ left\ Bintree\ (T)\ and\ right\ Bintree\ (T) \end{array}$$

T in the above production acts as a variable ranging over types. Different types can be sustituted for T. The type for a binary tree containing numbers, (eg. *Bintree* (*Number*)), will be automatically defined as follows :

$$\begin{array}{l} Bintree\,(Number\,) \rightarrow empty \\ Bintree\,(Number\,) \rightarrow \\ \quad node\ \ Number \\ \quad left\ Bintree\,(Number\,) \\ \quad and\ right\ Bintree\,(Number\,). \end{array}$$

The value:

$$\begin{array}{l} node\ \ 5 \\ left\ \ (\ node\ \ 2\ left\ empty\ \ and\ right\ empty\ \ ) \\ and\ right\ empty \end{array}$$

can be derived from Bintree (Number). Similarly, the value:

$$node\ \ (\ head\ \ 7\ and\ tail\ \ nil\ \ )\ left\ empty\ \ and\ right\ empty$$

can be derived from Bintree (Numlist).

5. Terms over F-Maple Types.

Terms over the types of F-Maple are used to specify functions operating on the data types. They are obtained by adjoining to the generating productions three new kinds of productions. These are called *function*, *case*, and *variable* productions. To distinguish them from the generating productions we will write them with $\Rightarrow$ as the *produces* symbol. Each term in F-Maple has a type. Terms stand for the *basic* values. Basic values are constructed from generating productions only and terms are *reduced* by computations to basic values.

Examples of function productions may be the following ones.

$$\begin{array}{l} Number \Rightarrow Number\ +\ Number \\ Numlist \Rightarrow append\ \ Numlist\ \ after\ \ Numlist \end{array}$$

Non-terminals on the right hand side specify the types of formal arguments while non-terminals on the left hand side specify the types of the function result. Thus the first function takes two values of type *Number* and yields a *Number* again. Addition is a predefined F-Maple function. On the other hand, the two-argument function *append* operating over the type *Numlist* must be defined at the same time as its production is adjoined to the grammar of F-Maple.

The above function productions combined with generating productions are used to produce the following term from *Numlist*.

$$append\ \ nil\ \ after\ \ (\ head\ \ 5+7\ and\ tail\ \ nil\ \ )$$

Since this term is produced from the non-terminal Numlist, it denotes (stands for) a data value of type Numlist. The computation of F-Maple reduces this term to the basic term *head* 12 *and tail nil* which is produced only by the generating productions. Computation of F-Maple transforms F-Maple terms in such a way that the use of all but the generating productions are removed. A term produced by generating production cannot be further reduced. The computation rule for *append* may be specified as follows.

$$\begin{array}{l} append\ \ Ls\,1\ after\ \ Ls\,2 = \\ \quad \mathbf{case}\ Ls\,2\ \mathbf{of} \\ \quad\quad nil\ \mid Ls\,1 \\ \quad\quad head\ \ H\ \ and\ tail\ \ T \mid \\ \quad\quad\quad head\ \ H\ \ and\ tail\ \ (\ append\ \ Ls\,1\ after\ \ T) \end{array}$$

In the body of *append*, we use variables $Ls\,1$ and $Ls\,2$ to denote the two arguments of append. The variables are automatically declared by the addition of two new variable productions:

$$\begin{array}{l} Numlist \Rightarrow Ls\,1 \\ Numlist \Rightarrow Ls\,2 \end{array}$$

Note that types and variables are capitalized for readability purposes.

When *append* is invoked, $Ls\,2$ will be bound to the actual argument of type *Numlist*. There are two generating productions for the type *Numlist* thus there are two possible forms for $Ls\,2$. If $Ls\,2$ is nil, the first case is executed. The result of this function is just the value of $Ls\,1$, otherwise, $Ls\,2$ must be a list consisting a head and a tail. In the latter case, the head of the list $Ls\,2$ is given the name H, and the tail T. Now these variables can be used in the body of the production of this second case. This is because two new variables are declared in the second clause of **case** .

$$\begin{array}{l} Number \Rightarrow H \\ Numlist \Rightarrow T \end{array}$$

The result of the function would be combining the head of $Ls\,2$ with the result of appending $Ls\,1$ after the tail of $Ls\,2$.

Generally, case productions are of the following form.

$$\mathbf{S} \Rightarrow \mathbf{case\ T\ of}\ \alpha_1 \mid \mathbf{S}\ \alpha_2 \mid \mathbf{S} \cdots \alpha_n \mid \mathbf{S}$$

where each α_i is called a **case** *label*. This case production is legal iff the case labels correspond exactly to all the generating productions for the type **T**. The user may adjoin a case production for any combinations of types **S** and **T** using his own variable names in the case labels.

The searching function of an ordered binary tree is an example of a generic Boolean function.

$$Bool \Rightarrow search\ \ Bintree\ (T)\ \mathrm{for}\ T$$

Its body can be defined as :

$$\begin{array}{l} search\ \ Tree\ \ \mathrm{for}\ \ Value\ = \\ \quad \mathbf{case}\ \ Tree\ \ \mathbf{of} \\ \quad\quad empty\ \mid false \\ \quad\quad node\ \ V\ \ left\ Lt\ \ and\ right\ Rt\ \mid \\ \quad\quad\quad \mathbf{case}\ \ V < Value\ \ \mathbf{of} \\ \quad\quad\quad\quad true\ \mid search\ \ Rt\ \ \mathrm{for}\ \ Value \\ \quad\quad\quad\quad false\ \mid \\ \quad\quad\quad\quad \mathbf{case}\ \ Value < V\ \ \mathbf{of} \\ \quad\quad\quad\quad\quad true\ \mid search\ \ Lt\ \ \mathrm{for}\ \ Value \\ \quad\quad\quad\quad\quad false\ \mid true \end{array}$$

The above definition presupposes the comparison functions, $<$, for each concrete type T used. For instance, for Bintree (Number), we need :

$$Bool \Rightarrow Number\ <\ Number$$

As mentioned above, the scale of possible control behaviours of functional programs is very limited. We did not attempt to include any explicit control mechanism in F-Maple. The computation is by lazy evaluation.

6. Description of RF-Maple.

In combining functions and relations together, we have a choice of introducing functions in a relational environment, or introducing relations in a functional environment. In the first case we obtain the standard predicate logic with functions in terms. The second case leads to a logic without formulas but only with terms. This kind of logic, although not as common as the first one, is perfectly legal from the logical point of view and is called *term* logic. Actually it is slightly simpler than the traditional presentation of predicate logic because the sometimes superfluous distinction between formulas and terms dissappears.

In the design of RF-Maple we have opted for the *term* logic. Relations are simply functions with Boolean values. Functions in applicative languages have all arguments input only. Therefore relations in a functional programming language are equivalent to tests of R-Maple. The power of logic programming comes from generators, that is Boolean functions with output arguments. Thus any extension of a functional programming language to a relational one should permit Boolean functions with output arguments.

One has to be careful to limit Boolean functions as the only kind of functions that can generate output. It is easy to give the declarative reading $\exists x\ (G\ (x\ )\ \&\ T(x\ ))$ to the program **find** x **in** $G\ (x\ );\ T(x\ )$ no matter how many values satisfy $G\ (x\ )$ where $G\ (x\ )$ is a generator. On the other hand, if we allow the integer function $f\ (x\ ,y\ )$ with y being the output argument, we could have difficulties determining what number does the term $f\ (6,y\ )+3$ stand for.

The computation of RF-Maple is taken over from the component languages without any changes. Functions are computed by the lazy evaluation of F-Maple. Generators are computed by the rewriting rules of R-Maple. The latter computation is necessarily slower because it must cater to the backtracking. Functions execute without this overhead.

RF-Maple has, in addition to the four basic constructs of F-Maple, four new ones. These are the parallel *and*, parallel *or*, *assignment*, and *find* constructs.

We would like to extend F-Maple to include the control structures of R-Maple. This includes both parallel and sequential *and* and *or*. Sequential *and* and sequential *or* can be predefined using the **case** construct as follows :

Bool ⟹ *Bool* ; *Bool*

$A\,;B$ = **case** A **of** *true* | B *false* | *false*

and

Bool ⟹ *Bool* **or** *Bool*

A **or** B = **case** A **of** *true* | *true* *false* | B

For parallel *and* and parallel *or*, we introduce two new productions:

Bool ⟹ *Bool* || *Bool*
Bool ⟹ *Bool* **orp** *Bool*

Control will be passed on to the two bodies as is the case in R-Maple.

Assignments are of the form :

Bool ⟹ α := **T**

where α is an identifier declared as **T** ⟹ α for any type **T**.

Boolean functions can have output arguments. These are called *generators*. Generators can contain the *find*, *assignment*, the parallel *and* and parallel *or* constructs as well as calls to another generators. Thus an example is:

Bool ⟹ *generate Number from Numlist*

generate X from Lst =
  case *Lst* **of**
    empty | *false* ;
    head H and tail T |
      X := *H* **or** *generate X from T*

All *find* productions are of the form :

Bool ⟹ **find** α: **T** **in** *Bool*

where α is an identifier and **T** is a type. For each *find* production, two more productions are automatically added. These are the variable production **T** ⟹ α and the assignment production *Bool* ⟹ α:=**T**. The productions may be used in the body of **find** . For example:

find *X* : *Number* **in**
  generate X from
    head 2 *and tail head* 8 *and tail head* 5 *and tail nil* ;
  7 <*X*

is a correct term of type *Bool* because it uses the production *Bool* ⟹ **find** *X*: *Number* **in** *Bool*. This term reduces to true after one backtrack to obtain the value 8, thus satisfying the test 7 <*X*.

By mixing all eight kinds of productions, we can create arbitrarily complicated terms over our types.

Let us give as an example for the RF-Maple implementation of parallel Quicksort. It is a generator of type *Bool*.

Bool ⟹ *sort Numlist into Numlist*

sort Il into Ol =
  append already sorted (*nil*) *after Il giving sorted Ol*

The body of *sort* calls another generator *append*. At this point we urge the reader to reflect on how the syntactic extensibility of RF-Maple self-describes the intended effect of both generators down to the level of indicating the output variables. This can be contrasted with the quite cryptic Prolog counterpart (especially if difference lists are used).

The definition of the generator *append* is a recursive one.

Bool ⟹
  append already sorted Numlist
    after Numlist giving sorted Numlist

append already sorted Sl after Ul giving sorted Ol =
  case *Ul* **of**
    nil | *Ol* :=*Sl*
    head N and tail T |
      case *partition T by N* **of**
        small Sml and large Lrg |
          find *X*: *Numlist* **in**
            append already sorted Sl
              after Lrg giving sorted X ||
            append already sorted (*head N and tail X*)
              after Sml giving sorted Ol

Two partioned sublists *Sml* and *Lrg* are sorted in parallel. We use the speeded up version of Quicksort where the concatenation of the two sorted sublists is done on the fly.

Both predicates above are generators. However, there is no need to program *partition* as a predicate. Partition is, then, simply a function yielding two lists. The relevant definitions are as follows.

```
Pair → small Numlist and large Numlist
Pair => partition Numlist by Number

partition Nl by Num =
  case Nl of
    nil | small nil and large nil
    head H and tail T |
      case partition T by Num of
        small S and large L |
          case Num < H of
            true |
              small S and large ( head H and tail L )
            false |
              small ( head H and tail S ) and large L
```

If the reader finds such a style of programming too Cobol-like let us note that

a) the syntax of constructs is entirely under the control of the programmer. If the user prefers the terse Prolog-like style, he just has to define the types, predicates and functions accordingly,

b) bodies of functions are not entered by a programmer. A structured editor is used. The editor knows from the given context what type and what kind of productions are available and the programmer needs only to select from a menu listing all the productions available at the moment.

As the second example of combining functions and generators we present the RF-Maple implementation of the eight queens problem. Solutions are obtained by the invocation of the generator

```
give a solution S to 8 queens
```

Should the correct solution S of the problem turn out to be unacceptable for some reasons later, the generator will be backtracked to produce the next solution by the standard methods of R-Maple computations.

The solution S is encoded as a list of column positions of queens. The i-th element of S is the column position of the queen in the row i.

We need two auxiliary functions

```
Bool =>
  queen in column Number
    is compatible with solution Numlist
Numlist =>
  attach new position Number
    at the end of solution Numlist
```

The first one is a test verifying the compatibility of the next position of a queen with a partial solution. Note that although it is a predicate, it behaves, and indeed is, an ordinary F-Maple function which can be executed faster than a generator. The second function yields an extended solution from an accepted new position and a partial solution. We do not give the bodies of functions here as they are quite straight-forward.

The main generator is defined as follows.

```
Bool => give a solution Numlist to Number queens

give a solution S to N queens =
  case N=0 of
    true | S:=nil
    false |
      find X: Numlist in
        give a solution X to N-1 queens ;
          find C: Number in
            C:=1 or C:=2 or C:=3 or C:=4 or
            C:=5 or C:=6 or C:=7 or C:=8;
            case queen in column C
              is compatible with solution X of
              true |
                S:= attach new position C
                      at the end of solution X
              false | false
```

This generator is quite simple. After finding the partial solution X the eight candidates C are tried. In the case of an acceptable candidate the partial solution X is extended to the required length by generating the solution S. In the case that all candidates are rejected the recursive invocation of the generator is reentered to generate a new partial solution X.

The next example illustrates cooperation of concurrent processes using partially instantiated streams as described in [Shap 2]. We shall present the RF-Maple version of the queue manager.

The streams by which processes communicate are represented using lists. Messages are sent and kept in queues. Therefore, we redefine the type *list* as a type constructor as follows :

```
List (T) → nil
List (T) → head T and tail List (T)
```

Note that the type Numlist is the same as List (Number).

The queue manager accepts two kinds of messages. They are:

```
Message (T) → enqueue T
Message (T) → dequeue T
```

for putting and retrieving elements of type T from the queue. As in Shapiro's program, the queue manager is communicating with two users via a merge process. The relevant types are :

```
Bool => user 1 with stream List (Message (T))
Bool => user 2 with stream List (Message (T))
Bool =>
  merge List (T) with
    List (T) yielding List (T)
Bool =>
  queue with front List (T) end List (T)
    and messages List (Message (T))
```

These are invoked by :

```
find S1, S2, S3 : List (Message (T)) in
  user 1 with stream S1 ||
  user 2 with stream S2 ||
  merge S1 with S2 yielding S3 ||
  find Q : List (T) in
    queue with front Q end Q and messages S3
```

We do not give the predicate bodies for *user* 1 and *user* 2. *Merge* must be a primitive in RF-Maple. The body for *queue* is as follows :

```
queue with front H end T and messages M ==
  case M of
    nil | true
    head Hm and tail Tm |
      case Hm of
        enqueue V |
          find Nt in
            T:= head V and tail Nt ||
            queue with front H end Nt and messages Tm
        dequeue V |
          V:= head of H ||
          queue with front (tail of H )
            end T and messages Tm
```

The first case terminates the cooperation of processes when the list of messages is exhausted. Otherwise, the head of the list of messages contains the message *enqueue* or *dequeue*. In the first case, the end of the queue is partially instantiated with the pair composed of the value to be enqueued and the rest as yet not instantiated. The queue manager is then recursively invoked with the rest of the messages. In the latter case, V should be instantiated with the value at the head of the front. The queue manager then proceeds recursively with the rest of the messages. *Head* and *tail* are projection functions. *Head* is defined as follows :

```
T => head of List (T)
head of L ==
  case L of
    nil | 0
    head A and tail B | A
```

Similarly for *tail*. We suppose that for each T used there is a production

$T \rightarrow 0$

giving a distinguished constant 0 of type T. Note also that if the queue is empty when a dequeue message comes, H, and thus *head of* H, will not be instantiated. Therefore, the *user* process trying to use V will be delayed when trying to determine the structure of V in a **case** construct. It will be allowed to proceed when an enqueue message from the second *user* arrives. Hence, it is necessary to restart the queue manager in parallel.

7. Conclusion.

In the process of combining the power of a relational logic programming language with a typed extensible functional programming language, we find that RF-Maple offers a solution to a wide variety of applications. We have a syntactically extensible programming language with a fine scale of control behaviour. Moreover, the declarative reading is not compromised by any operational aspects. The declarative reading of RF-Maple programs specifies only the partial correctness. Programs may still fail to terminate. But if they terminate, the declarative reading has been achieved. Cuts of Prolog are not invariant to the declarative reading.

Finally we should say a few words on the current state of the languages. We have a running pilot implementation of R-Maple done by the second author. There is an almost running implementation of F-Maple done by the first author. Almost running is because there is a lot more than a mere interpreter to F-Maple. F-Maple has been designed as its own operating system with a structure editor and a virtual file system. A function is not aware whether the arguments come from another function, from a file, or from the input. In the last case we reenter the structure editor and the user constructs the value of the requested type via menus of applicable generating productions. Thus there is never a need for a program to parse the input from the characters.

RF-Maple is a true superset of F-Maple. One needs a separate interpreter for the execution of generators in addition to the changes in the structure editor. This interpreter will be added to the F-Maple system as soon as F-Maple becomes operational. With the capability to sequence the execution of a program sequentially or in parallel, and the power of both functional and relational programming, RF-Maple goes a little beyond the possibilities of the currently available languages without compromising the declarative reading of programs by cuts and commits.

References

[Malu] Maluszynski J., Nilsson J., A Notion of Grammatical Unification Applicable to Logic Programming Languages, Department of Computer Science, Technical University of Denmark, Doc. ID 967, August 1981.

[Clar] Clark K.L., McCabe F.G., Gregory S., IC-Prolog Reference Manual; Research Report Imperial College, London 1981.

[Kand] Kanda A., Abramson H., Syrotiuk V., A Functional Programming Language Based on Data Types as Context Free Grammars; (submitted for publication), December 1983.

[Kowa] Kowalski R., Logic for Problem Solving; North Holland, Amsterdam 1979.

[Shap 1] Shapiro E., A Subset of Concurrent Prolog and its Interpreter, TR3 Institute for New Generation Computer Technology, Jan 1983.

[Shap 2] Shapiro E., Takeuchi Akikazu, Object Oriented Programming In Concurrent Prolog; Journal of New Generation Computing, Volume 1, Number 1, 1983.

[Shoe] Shoenfield J., Mathematical Logic, Addison-Wesley, 1967.

[Symp] 1984 International Symposium on Logic Programming, Feb. 6-9, 1984.

[Voda 1] Voda P. J., R-Maple: A Concurrent Programming Language Based on Predicate Logic, Part I: Syntax and Computation; Technical Report of Dept. Comp. Science UBC, Vancouver August 1983.

[Voda 2] Voda P. J., F-Maple: A Simple Typed Extensible Functional Programming Language Designed as an Operating System (in preparation).