RF-MAPLE: A LOGIC PROGRAMMING LANGUAGE WITH FUNCTIONS, TYPES, AND CONCURRENCY

by

Paul J. Voda and Benjamin Yu

Technical Report 84-8

April 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 compostion 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.

April 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

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. However, 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 true 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. Moreover, 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, 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 sacraficed. 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* where a predicate without output variables is called a *test* . We use the vertical bar to seperate the output arguments from the input arguments.

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

This generator generates the head of the list *lst* if the list is not nil, otherwise it returns F (false). Two new variables, *hd* and *tail* are declared in case the list *lst* is not nil. The declarative reading of this predicate is :

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

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 define 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 $A \Rightarrow B$ where part *A* always contains the operator !. For example:

(*A* **or** *B*)! => *A* ! **or** *B*
(*A* **orp** *B*)! => *A* ! **orp** *B* !
F ! **or** *B* => *B* !
T ! **or** *B* => **T** !
F ! ; *B* => **F** !
T ! ; *B* => *B* !

The first rewrite rule specifies that, for a sequential **or** , control is first passed to A. If A is false, then control will then pass on to B because of the rule **F** ! **or** *B* => *B* !. In the second rewrite rule, during a parallel *or* , control is passed to both A and B. That is, two processes are created to execute A and B simultaneously. 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* $(s, t \mid x)$. Add is a *functional* generator. In general, a non-functional generator $G(\mid x)!$ will be transformed into the form $x := a$! **or** $H(\mid x)$ where a is the first value generated, and $H(\mid x)$ 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 program is of the form

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

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

find x **in** $\{(x := a$! **or** $H(\mid x)); T(x)\} \Rightarrow$
find x **in** $\{(x := a \ !; T(x))$ **or** $(H(\mid x); T(x))\} \Rightarrow$
find x **in** $\{x := a \ !; T(x)\}$ **or find** x **in** $\{H(\mid x); T(x)\} \Rightarrow$
$T(a)!$ **or find** x **in** $\{H(\mid x); T(x)\}$ (1)

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

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

Should the test $T(a)$ in (1) fail, the control will fall back into the backtrack search employing $H(\mid x)$. 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 *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]}

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** $\{G(\mid x) \mid\mid T(x)\}$.

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 $x := s$! 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. The quantifier is then discharged by the rewriting rules:

find x **in** $\{x := s \ !; A(x)\} \Rightarrow A(s)!$
find x **in** $\{x := s \ ! \mid\mid A(x)\} \Rightarrow A(s)$

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 follow:

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-terminal on the left hand side of a *generating* production is an F-Maple type. 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 for the type *Numlist* denoting a number list containing elements 2, 4, and 6, whereas

42 +35 *i*

is a data value for the type *Complex* denoting a complex value with the obvious meaning. Thus the use of grammars at once specifies the data type and permits the concrete syntax to the 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 elimates 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.

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 => 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.

Number => *Number* + *Number*
Numlist => *append Numlist after Numlist*

Nonterminals on the right hand side specify the types of formal arguments while nonterminals 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*

Computation of F-Maple reduces this term to *head* 12 *and tail nil*. The computation rule for *append* may be specified as follows.

append Ls 1 after Ls 2 =
  case *Ls 2* **of**
    nil | *Ls 1*
    head H and tail T | *head H and tail append Ls 1 after T*

In the body of *append*, we use variables *Ls 1* and *Ls 2* which are automatically declared by the addition of two new variable productions:

Numlist => *Ls 1*
Numlist => *Ls 2*

Note that only types and variables are capitalized.

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 now declared.

Number => *H*
Numlist => *T*

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.

S => case T of α_1 | **S** α_2 | **S** $\cdots$ α_n | **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.

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. The type *Bool* is defined as follows.

Bool → *true*
Bool → *false*

Functions in applicative languages have all arguments input only. Therefore relations in a functional programming language are only 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 must 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 satisfying $G\ (x\ )$ 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 would have difficulties determining what number does the term $f\ (6,y\ ) + 3$ stands 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 *find, assignment*, parallel *and*, and parallel *or* constructs.

find is a schema of productions 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** $X := 5$

is a correct, if not particulary useful, term of type *Bool* because it uses the production *Bool* => **find** X: *Number* **in** *Bool*.

We would also like to extend RF-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 in the case in R-Maple.

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*

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.

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.

On the other hand, the use of functions instead of functional predicates should speed up the execution because there is no backtracking needed.

Another well known advantage of using functions over predicates is that they can be composed (nested) without the annoying auxiliary variables.

As the second example of combining functions and generators we present the RF-Maple implementation for 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.

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.

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