ROSE: A Practical Higher-Order Functional/Logic Language

Zeki O. Bayram
Department of Computer Engineering
Boğaziçi University
Bebek 80815
Istanbul
TURKEY

e-mail: bayram@boun.edu.tr

Barrett R. Bryant
Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35294-1170
U.S.A.

e-mail: bryant@cis.uab.edu

Ünal Altınay
Department of Computer Engineering
Boğaziçi University
Bebek 80815
Istanbul
TURKEY

e-mail: altinay@boun.edu.tr

Abstract

We present a higher-order functional/logic language, ROSE. The programs of ROSE are made up of conditional constructor based term rewriting systems. The conditions in the rules can optionally be committing. The operational semantics of the language is conditional narrowing, augmented to deal with committing conditions. The major innovation of the language is the use of committing guards and backtracking to make possible very practical, operationally oriented programs. We show many practical examples, one of which is the definition of the extra logical function *not,* which implements the *negation-as-finite-failure* rule in the context of functional/logic programming.

Keywords: committing guards, conditional narrowing, e-unification, term rewriting systems, non-determinism, functional/logic programming, practicality

1. Introduction

ROSE is a declarative functional/logic language designed to demonstrate that functional/logic programming need not be seen only as a topic of academic interest, but rather that actually useful programs can be written in this paradigm. ROSE can be likened in a sense to Prolog, which demonstrated that theorem proving can also be used for computation and statements in first order predicate calculus can be seen as programs to be executed. The success of Prolog came about largely because of its pragmatist approach, which is most apparent in the inclusion of *extra-logical* facilities, such as the built-in predicates **assert**, **retract**, and **cut** (which controls backtracking), as well as somewhat ad-hoc higher-order features, such as the **univ** and **call** predicates. Furthermore, Prolog's depth first search strategy of the resolution tree is

incomplete. All these have features take us away from a purely declarative reading of Prolog programs, but were necessary in order to obtain a usable programming language.

ROSE programs consist of *conditional constructor based term rewriting systems* (which are not necessarily confluent) augmented with *higher-order constructs* (nameless functions in the form of a kind of generalized lambda abstraction) and *committing conditions* (also called *guards*). Operational semantics is conditional narrowing, where the narrowing tree is traversed in a depth-first manner. It is possible to choose from an innermost or outermost narrowing strategy, and switch between the two during execution as necessary. It is the interplay between guards, non-confluent rewrite rules, non-determinism and backtracking which give ROSE programs their brevity and expressive power. Admittedly, in the presence of this combination of features, it is hard to talk about a declarative reading of ROSE programs (i.e. seeing rewrite rules as conditional equality theories). However, as will be seen in the examples to follow, the operational paradigm of computation that is the result of this interplay turns out to be a very elegant one, enough to justify our choice of extra-logical features in the language.

Many efforts have been made for the integration of logic and functional programming paradigms in a common framework (see, for example [1, 3, 5, 7, 10, 12, 13]). These languages are mostly *pure* in the sense they have no extra-logical features. Certainly, none of them have the *specific combination* of features present in ROSE. Other researchers (see, for example [4, 6, 8, 9]) have investigated the conditions under which narrowing and/or conditional narrowing is complete. Some sort of restriction is usually imposed on the set of rewrite rules to achieve the completeness results.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of ROSE. Section 3 contains a more detailed description of the language. Section 4 describes narrowing and conditional narrowing which is the operational semantics of the language. Section 5 demonstrates the practicality of the language through many examples. Section 6 describes some other proposed functional/logic languages, and Section 7 is the conclusion.

2. A Brief Description of ROSE

As was mentioned before, ROSE programs consist of *constructor based conditional term rewriting systems* where a condition can optionally be *committing*. The fact that we differentiate between *constructors* and other *function symbols* permits a clean distinction between *data* and *functions that operate on data.* This distinction not only makes implementation efficient, but in addition, a *normal form* of a term can be defined very naturally: a term is in *normal form* iff it contains only variables and constructors. So the *denotation* of a term **t** is the set (due to non-determinism and possible non-confluence of rules) of all terms in normal form **t** can be reduced to.

ROSE has higher-order features, such as the passing of functions as arguments and returning them as values. To facilitate this, *nameless functions* in the form of *conditional lambda abstractions* are part of ROSE syntax. This is the counterpart of lambda abstractions in functional languages. *apply* is a built-in function of the language, which applies a nameless function to one or more arguments. Formal parameters of *conditional lambda abstractions* are terms in normal form, and application means *e-unification* of the formulas with actuals, and applying the substitution generated to the body of the abstraction (actually, the process is a little more complicated due to the condition, but this level of description is appropriate for now).

The operational semantics makes use of two narrowing strategies, as specified by the programmer: the default strategy can be *innermost*, which results in an *eager* evaluation of expressions, or *outermost* which results in a *lazy* evaluation of expressions and allows the

handling of infinite data structures. Individual subterms of a term can be marked for execution using either strategy, which gives the programmer a lot of flexibility in determining the strategy to be used in evaluating expressions.

The *e-unification/narrowing* algorithm employed can solve equations using either the *(leftmost) innermost* or *(leftmost) outermost* narrowing strategies. Although incomplete in general (not all possible subterms are tried in a narrowing step), this kind of controlled e-unification is necessary due to efficiency considerations. Besides, the eager strategy in general coincides with the intuitions of the programmer, and the lazy strategy is called for when infinite data structures are required. Thus, not all the answers dictated by a declarative reading of the programs shall be generated by the interpreter (in the case where the rules are confluent, and a declarative reading is possible). But again, programmers seldom think in terms of proving theorems when they write programs, and they usually have an operationally-oriented mindset. Similar considerations lie behind the design choices of Prolog [2].

3. ROSE In Detail

In *constructor based term rewriting systems,* there are two kinds of function symbols: *data* constructors, or just *constructors*, and *defined function symbols*, also called *function names*. Constructors are used to represent data, much in the same way of the list constructor "." in Lisp. Zero-ary constructors are also called *constants*. Below we give a recursive definition of *term* as it applies to ROSE (in this presentation the terms *term* and *expression* are used interchangeably).

- A *constant* (zero-ary constructor) is a term.
- A *variable* is a term. Variables in ROSE have the same meaning and notation as in Prolog.
- If $s_1, \ldots, s_n$ are terms and f is a function name, then $f(s_1, \ldots, s_n)$ is a term, also called a *function application*.
- If $s_1, \ldots, s_n$ are terms and c is a constructor, then $c(s_1, \ldots, s_n)$ is a term, also called a *constructed expression*.
- (*lambda*,$[s_1, \ldots, s_n]$, *condition*, *body*) is a term, called a *conditional lambda abstraction* (or just *lambda abstraction*) if $s_1, \ldots, s_n$ are terms in normal form (see below), and *condition*, and *body* are terms. $s_1, \ldots, s_n$ are the *formal parameters* of the lambda abstraction.
- *apply*$(s_1, \ldots, s_n)$ is a term, called and *apply expression*, if $s_1, \ldots, s_n$ are terms. s_1 should be an expression that evaluates to a lambda abstraction and $s_2, \ldots, s_n$ are the actual parameters s_1 will be applied to.

The *scope* of the variables in $s_1, \ldots, s_n$ of a lambda abstraction (*lambda*,$[s_1, \ldots, s_n]$, *condition*, *body*) is the *condition* and *body*. All such variables in *body* or *condition* are said to be *bound*. If a variable is not bound, then it is said to be *free*. A term is said to be in *normal form* if it consists *only* of constructors, variables and lambda abstractions of the form *(lambda*,[$s_1, \ldots, s_n$],*guard,body)*. A ROSE program consists of an ordered set of conditional rewrite rules of the form $f(s_1, \ldots, s_n)$:*condition* $\rightarrow$ *body* or $f(s_1, \ldots, s_n)$#*condition* $\rightarrow$ *body* where $s_1, \ldots, s_n$ are all in terms in normal form, *f* is a function symbol, and *condition* and *body* are terms. The first definition above is a *non-committing* definition, whereas the second one is a *committing* definition. Note that unconditional rewrite rules are equivalent to conditional ones where the condition is the constant *true*.

ROSE also has a module system with *public* and *private* parts in each module. Let *m* be the name of a module. The functions defined in a public part of *m* are visible to other modules that *use m*, whereas the functions in the private part of *m* are visible only within *m*. Any function in the public section of a module can be invoked from any module by prefixing it with

the module it belongs to. For example, if *f(...)* is defined in the module *m*, *m*::f(...) invokes f(...) in module *m*. A function *f* is *accessible* in a module *n* if f is defined in *n*, if *f* is in a module that *n* uses, or if *f* is *accessible* in some module *n'* and *n* uses *n'* (i.e. accessibility is transitive).

4. Operational Semantics of ROSE

The operational semantics of ROSE is based on *conditional narrowing*. First, we give some terminology and notations that will be used in the remainder of the paper. We denote a *substitution* by $\{t_1 / X_1,..., t_n / X_n\}$ where X_i , $1 \le i \le n$, are called the replaced variables, and t_i, $1 \le i \le n$, are called the replacing terms. An application of a substitution δ to a term **t**, denoted by **(t)** δ, is the term obtained by simultaneously replacing all unbound variables **X** in **t** with **v** such that **v/X** $\in \delta$. *Composition* of two substitutions δ_1 and δ_2, denoted by $(\delta_1 \text{ o } \delta_2)$, is defined to be a substitution θ such that for any term **t**, $(t)\,\theta = (t)(\,(\delta_1 \text{ o } \delta_2) = ((t)\,\delta_1\,)\delta_2$.

Let **W** be a set of variables, and δ a substitution. The *restriction* of δ to **W**, δ **[W]**, is defined as $\{ t / X \mid X \in W \text{ and } t/X \in \delta \}$. VAR_t is the set of all variables in the term **t**,

t[u] denotes the subterm of t at occurrence u (**u** can be seen as the address of the subterm). A *redex* is a subterm of the form **f(...)** where **f** is a *function name*. An *innermost redex* is a redex that does not have a proper subterm that is also a redex (t[u] is a *proper subterm* of a term t if t[u]≠ t).

4.1 Conditional Narrowing

Given a term *E* and a set of (conditional) rewrite rules, the algorithm in Figure 1 generates a *conditional narrowing* tree for *E*. This algorithm performs conditional narrowing essentially equivalent to that described in [8], but generates the narrowing tree explicitly. Note that *a forest* of conditional narrowing trees is generated during the execution of the algorithm due to the recursive calls for evaluating the conditions. For two expressions *E* and *E'* that are in the conditional narrowing tree for some expression, if *E'* is a *direct descendant* of *E* (i.e. there are no other nodes between *E* and *E'*) and the arc between them is labeled with δ we say that *E* and *E'* belong to the *one step conditional narrowing relation* and denote this fact as $E >>>_\delta E'$.

Notice that conditional narrowing subsumes *standard* or *unconditional* narrowing in that all unconditional rewrite rules are trivially conditional, where the condition is the constant *true*. Note also that conditional narrowing is a semantically much more complex operation than simple narrowing, which is what gives the conditional rewrite rules their expressive power.

1. Create a tree T with only one node, *E* (the expression to be narrowed).
2. Let L be a leaf in the narrowing tree generated so far.
- If L is in *normal form*, label it *success*.
- For any rule **r** *lhs : cond* $\rightarrow$ *rhs* and any subterm **t** of L at occurrence u: if substitution δ is a *most general unifier* of **t** and *lhs,* the conditional narrowing tree for *(cond)*δ has a path from the root to a leaf node that is the constant *true* with the composition of substitutions along the path being θ (note the recursion!), create a child L' of L such that L'= (L[u $\leftarrow$ rhs]) $\delta\theta$), and label the arc between *L* and *L'* as (δ o θ) (this is a one step narrowing operation applied at every possible occurrence of L using every possible rule). If there is no such rule **r**, label the node *failure*.
3. Repeat 2 until all leaves marked, or forever!

Figure 1: Narrowing Algorithm (Using Conditional Rules)

Now, the algorithm in Figure 1 works only in the absence of conditional lambda abstractions. To deal with abstractions, a conditional lambda abstraction can be temporarily given a unique name, and a rewrite rule defined automatically for the abstraction. No subterm of a lambda abstraction should be used in any narrowing step. If a leaf of the narrowing tree contains *apply*((*lambda*,*[s_1, ... ,s_n], condition*, *body*)),t_1,...,t_n), we generate a new rewrite rule *f(s_1, ... ,s_n):condition* $\rightarrow$ *body* where *f* is a previously unused function symbol, and replace *apply*((*lambda*,*[s_1, ... ,s_n],condition*, *body*)),t_1,...,t_n) with *f(t_1,...,t_n)*. Surely, this modification preserves the intended semantics of the original program.

Furthermore, no provision is made for committing guards. That is a question of how and when to prune the narrowing tree, and is explained in section 4.4.

4.2 Narrowing Strategies

A *narrowing strategy* determines which of the redexes of a leaf node in a narrowing tree will be used in a narrowing step. Two obvious strategies are the *innermost* strategy, where an innermost redex is selected, and the *outermost* strategy. In the outermost strategy, let L be a leaf node, and L[u] be the redex selected. Then, it must be the case that there is no redex L[w] of L such that L[u] is a *proper* subterm of L[w] *and* L[w] can participate in a narrowing step. In other words, the inner subterms are evaluated *just enough* so that outer redexes can participate in a narrowing step.

4.3 E-Unification

Narrowing can also be used to solve *equations*, an operation called *e-unification*. Suppose for two terms t_1 and t_2, we want to find a substitution δ such that $(t_1)\delta = (t_2)\ \delta$ can be shown to be true using the rewrite rules (interpreted as conditional equality between terms) in a program. To find such a δ we need a rule $X=X \rightarrow true$ to be part of the program. We then generate the narrowing tree for the expression $t_1=t_2$. If there is a leaf in this tree which is the constant *true*, the composition of substitutions on the path leading from the root to this success node, say θ, gives us a substitution that is *more general* than δ, i.e. $\delta = \theta\ o\ \phi$ for some ϕ [14].

4.4 Committing Guards

Committing guards are meaningful only if the narrowing tree for an expression is generated in a *left-to-right*, *depth-first* fashion and the rules in a program have an ordering imposed on them by their physical location in a program. Let E_1 and E_2 be related with the *one step*

conditional narrowing relation, E_2 being the child of E_1. If *f(...)#guard → body* is the rule used in the narrowing tree to get from E_1 to E_2, the semantics of the committing guard dictates that there will be no other child of E_1 to the right of the E_2 that is obtained through the use of another definition of *f* and the same subterm of E_1 that was used in obtaining E_2 This restriction prunes the full conditional narrowing tree of some of its branches, because the programmer has given further information by using committing guards that those branches are not useful in finding a solution.

The committing guard allows the programmer to specify control information in a similar way to the *cut* operator in Prolog by allowing him to provide information about when remaining rules defining a function need/should not be used once the guard in one rule evaluates to *true*. The effect and convenience of this can be seen in the definition of various functions in the section on examples. Because of the committing guard, however, it is possible to write programs that are logically wrong, but which operationally exhibit correct behavior. An example of this is given in Figure 2.

smaller(X,Y)# X<Y → true.
smaller(X,Y) → false.

Figure 2: Defining A Logically Wrong Function That Exhibits Correct Behavior

4.5 Using Conditional Guards To Define The Function not

In Horn *clause logic programming*, since no negative information can be represented [11], negative information has to be deduced from the unprovability of positive information. Thus, if *p(a)* cannot be shown to be *true*, then it is assumed to be *false*. This has been termed the *negation as (finite) failure rule*. The same idea can be implemented in the context of functional/logic programming using the idea of committing guards. Figure 4 depicts a definition of the *not* function using committing guards.

not(X)# X → false.
not(X) → true.

Figure 3: Defining The not Function Using Committing Guards

4.6 Implementation of ROSE

Prolog has been used as the implementation language of ROSE. The intrinsic unification facilities of Prolog and the fact that all the *data* we are dealing with, including the rules defining functions, can be coded as first order terms, the native data structure of Prolog, have been a big convenience.

5. Sample Programs in ROSE

In this section, we give some function definitions to highlight the main features and capabilities of the language. We start out with functions that demonstrate the equation solving capabilities, higher-order features, and lazy evaluation facilities of the language that permit the handling of infinite data structures. (Note that even though in ROSE all functions must belong to some module, here we have left out the declarations for modules for clarity and brevity.)

We have used outermost (lazy) narrowing as the default mode in executing the functions given in the examples. However, no matter which strategy we are using as the default mode,

the built-in functions (labels) *lazy* and *eager* allow us to override the default strategy and execute any (sub)expression in a lazy or eager way. *lazy(exp)* evaluates *exp* utilizing the outermost narrowing strategy, and *eager(exp)* evaluates *exp* utilizing the innermost strategy. *eq* and *lazy_eq* are boolean valued built-in binary functions that e-unify their two arguments, *eq* utilizing the innermost narrowing strategy, and *lazy_eq* utilizing the outermost narrowing strategy.

5.1 Simple List Functions

In Figure 4 definitions of *app(end)*, *reverse* and *member* are given. *palindrome* and *revstrings* demonstrate the equation solving capability of ROSE. *palindrome* tests to see whether its argument is a palindrome, i.e. if it reads the same both backwards and forwards. However, when given a partially instantiated argument, it can generate the missing values in the sequence given to it.

```
app([],X) →X.
app([H|T],L) →[H|app(T,L)].
reverse([])→[].
reverse([H|T]) →app(reverse(T),[H]).
member(X,[X|Y])# true →true.
member(X,[H|T])# true →member(X,T).
member(X,Y) →false.
palindrome(L)# L lazy_eq reverse(L) →true.
palindrome(L) →false.
revstrings(A,B) : A lazy_eq app(X,app([H|T],Z))  and
                  B lazy_eq app(XX,app(reverse([H|T]),ZZ))
                  → [H|T].
```

Figure 4: Some List Functions

Figure 5 shows a terminal session, calling the function palindrome with an argument that is not fully instantiated. (The underscore "_" has the same meaning as in Prolog: a variable whose value will not be used elsewhere.) *revstrings* is also called and finds all substrings occurring in its first argument that occur in its second argument in reverse order.

```
>> palindrome([1,2,_,3,_,_]).
palindrome([1,2,3,3,2,1]) ⇒true
another solution ? y
palindrome([1,2,_62,3,_66,_68]) has no other solution

>> revstrings([1,2,3,4],[5,2,1,7]).
revstrings([1,2,3,4],[5,2,1,7]) => [1]
another solution ? y
revstrings([1,2,3,4],[5,2,1,7]) => [1,2]
another solution ? y
revstrings([1,2,3,4],[5,2,1,7]) => [2]
another solution ? y
revstrings([1,2,3,4],[5,2,1,7]) has no other solution
```

Figure 5: Sample Query

5.2 Higher-Order Features

The *foldr* function (adapted from [1]) is defined in Figure 6 and demonstrates the higher-order capabilities of ROSE. *foldr* takes three arguments: a binary operation OP, the identity element under the operation, and a list of items belonging to the domain of the binary operation. *foldr*(OP,IDENTITY,[a_1,...,a_n]) returns *OP*(a_1, *OP*(a_2,... *OP*($a_{(n-1)}$, *OP*(a_n ,*IDENTITY*))...)). *foldr* is thus implemented as a higher-order function.

Using *foldr* we can define other functions such as *list_append, list_and* and *list_or* by instantiating the first two arguments of foldr to a previously defined function, and an identity element under that function. Thus, *list_append* is a function that takes one argument, a list of lists, and appends all the lists together. *list_and* and *list_or* are defined similarly.

```
twice(F,A) →apply(F,apply(F,A)).
map(F,[])# true →[].
map(F,[H|T]) →[apply(F,H)|map(F,T)].
foldr(OP,IDENTITY,[])# true →IDENTITY .
foldr(OP,IDENTITY,[H|T]) →apply(OP,H,foldr(OP,IDENTITY,T)).
list_append →foldr(app, []).
list_and →foldr(and,true).
list_or → foldr(or,false).
true and true # true →true.
X and Y →false.
false or false # true  →false.
X or Y →true.
```

Figure 6: Higher-Order Functions

map and *twice* are also higher-order functions: *map* applies its first argument to all the elements of its second argument, and *twice* applies its first argument to its second argument twice. Figure 7 depicts some function calls using interesting combinations of the above functions.

```
>> list_append([[1],[2],[3]] ).
list_append([[1],[2],[3]]) ⇒ [1,2,3]
another solution ? y
list_append([[1],[2],[3]]) has no other solution
>> map(twice(app([a])),[[b],[c],[d]]).
map(twice(app([a])),[[b],[c],[d]]) ⇒[[a,a,b],[a,a,c],[a,a,d]]
another solution ? y
map(twice(app([a])),[[b],[c],[d]]) has no other solution
>> list\_append( map( twice(app([a])), [[c],[d,e],[f]] )).
list_append(map(twice(app([a])),[[c],[d,e],[f]])) ⇒ [a,a,c,a,a,d,e,a,a,f]
another solution ? y
list_append(map(twice(app([a])),[[c],[d,e],[f]])) has no other solution
```

Figure 7: Query For Some Higher-Order Functions

Note that when a function is to be passed as an argument, if that function has more than one defining rule, lazy narrowing must be used, so that the name of the function, and not the individual rewrite rules defining the function, are passed into the formal arguments.

5.3 Infinite Data Structures and Lazy Evaluation

Finally, we give the definitions of functions for generating the list of *lucky* numbers (again adapted from [1]) which demonstrates the use of infinite data structures. The lucky numbers are generated as follows: We start out with 1,3,5,7,9,11,13,15,17,19,21,... and remove from the list every third item, which gives us 1,3,7,9,13,15,19,21,... Now, we remove from the resulting list every seventh item, and keep going. The numbers remaining in this sequence are the lucky numbers. Obviously, the list of lucky numbers is an infinite sequence. contains an implementation of the lucky numbers function in ROSE, using an infinite list to hold the resulting numbers. Note that we have used the built in function eager to evaluate eagerly what does not need to be evaluated lazily, thus saving redundant recomputation caused by lazy evaluation.

```
lucky → [1|lucky2(2,odds(1))] .
odds(N) →[N|odds(eager(N+2))].
lucky2(N,List): Y eq ith(N, lazy List) →
                [Y|lucky2(eager(N+1),knock_out(1,Y,List))].
ith(1,[H|T])# true →H.
ith(N,[H|T]) →ith(N-1,lazy T).
knock_out(X,Y,[H|T])# X eq Y →knock_out(1,Y,T).
knock_out(X,Y,[H|T]) →[H|knock_out(eager(X+1),Y,T)].
first_n_items(0,[H|T])# true →[].
first_n_items(N,[H|T]) →[H|first_n_items(eager(N-1),T)].
```

Figure 8: Function Definitions Demonstrating Infinite Lists

Figure 9 depicts a query for generating the first 3 lucky numbers.

```
>> first_n_items(3,lucky).
first_n_items(3,lucky) =>  [1,3,7]
another solution ? y
first_n_items(3,lucky) has no other solution
```

Figure 9: Query For Generating The First Three *lucky* Numbers

6. Related Work

IDEAL, described in [1], is a language which combines type checking, higher-order objects, lazy evaluation, function invertibility, and non-determinism. However, no mention is made of equation solving, and the language adopted is much more complicated than conditional rewrite rules with committing guards.

In the language described in [5], infinite data structures are available not as a result of the innate execution mechanism, but by modifying the programs in a not-so-obvious way.

SLOG, described in [7], employs innermost superposition as its operational semantics and does not have any notion of infinite data structures or higher-order functions.

K-LEAF, described in [10], is a language based on Horn Clause Logic with equality, and also has no notion of higher-order functions.

The language described in [13] tries to unify the two paradigms at a semantic level using domain theory as a common basis for functional and logic programming and uses set abstraction in a predominantly functional language to provide logic programming capability.

The field of functional/logic programming has reached a certain maturation point, and there is indeed quite a collection functional/logic programming languages already in existence, each with a different subset of features associated with functional and logic programming. We could not hope to list all of them here. ROSE is different from rest because it offers for the first time the facilities for controlling the execution of functional/logic programs and pruning the narrowing tree, where the pruning information is implicit in the rules in the form of guards. In a sense the combination of the specific features of ROSE result in a somewhat different and arguably more useful paradigm of computation then *equational programming*, which is what functional/logic programming is usually taken to be.

7. Conclusion

We have presented a higher-order functional/logic programming language, ROSE, that permits guards (committing conditions) in the conditional rewrite rules making up its programs. Higher-order facilities are available in the form of *conditional lambda abstractions* and the built-in *apply* function. The operational semantics, which is *conditional narrowing* augmented to deal with committing guards, can be done *lazily* using the outermost narrowing strategy (making possible infinite data structures) , or *eagerly* using the *innermost* narrowing strategy.

The major innovation in ROSE is the new paradigm of computation that results from the interplay between guards in the rules, non-confluent rewrite rules and backtracking (due to non-determinism). Other powerful features, such as the logical variable, and higher order features are an integral part of the language. Consequently, ROSE programs turn out to be concise and easily understandable.

References

[1] P.G. Bosco and E. Giovanetti. IDEAL: An Ideal Deductive Applicative Language. In *Proceedings the 1986 Symposium on Logic Programming*, pages 89-94, 1986.

[2] William F. Clocksin and Cristopher S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, 1987.
[3] D. DeGroot and G.Lindstrom, editors. *Logic Programming : Functions, Equations, and Relations.* Prentice-Hall, 1985.
[4] N. Dershowitz and M. Okada. Conditional equational programming and the theory of conditional term rewriting. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems,* pages 337-346, 1988.
[5] N. Dershowitz and D.Plaisted. Logic programming cum applicative programming. In *Proceedings of the 1985 Symposium on Logic Programming,* Pages 54-67, 1985
[6] M.Fay . First order unification in an equational theory. In *Proceedings of the 4th Workshop on Automated Deduction,* Pages 161-167, 1979.
[7] L.Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *Proceedings of the !985 Symposium on Logic Programming,* Pages 172-184, 1985
[8] E. Giovannetti and C. Moiso. A completeness result for e-unification algorithms based on conditional narrowing. In *Foundations of Logic and Functional programming Workshop,* pages 157-167. Springer-Verlag LNCS 306,1986
[9] J. M. Hullot. Canonical forms and unification. In *Proceedings of the 5. Conference on Automated Deduction,* pages 318-334, Berlin, 1980. Sringer-Verlag LNCS 87.
[10]Giorgio Levi et al. A complete semantic characterization of K-LEAF, a logic language with partial functions. In *Proceedings of the 1987 Symposium on Logic Programming,* pages 318-327, 1987.
[11]W. L. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1984.
[12]Uday S. Reddy. Narrowing as the operational semantics of functional languages. In *Proceedings of the 1985 Symposium on Logic Programming,* pages 138-151, 1985.
[13]Frank S. Silberman and Baharat Jayaraman. A domain-theoretic approach to functional and logic programming. Technical report TUTR 91-109, Tulane University, 1991.
[14]Akihiro Yamamoto. Completeness of extended unification based on basic narrowing. In *Proceedings of the Logic Programming Conference 88,* pages 19-28, 1988.