Adding equations to NU-Prolog

Lee Naish
(lee@cs.mu.OZ.AU)

Department of Computer Science
University of Melbourne
Parkville, 3052, Australia

Abstract

This paper describes an extension to NU-Prolog which allows evaluable functions to be defined using equations. We consider it to be the most pragmatic way of combining functional and relational programming. The implementation consists of several hundred lines of Prolog code and the underlying Prolog implementation was not modified at all. However, the system is reasonably efficient and supports coroutining, optional lazy evaluation, higher order functions and parallel execution. Efficiency is gained in several ways. First, we use some new implementation techniques. Second, we exploit some of the unique features of NU-Prolog, though these features are not essential to the implementation. Third, the language is designed so that we can take advantage of implicit mode and determinism information. Although we have not concentrated on the semantics of the language, we believe that our language design decisions and implementation techniques will be useful in the next generation of combined functional and relational languages.

1 Introduction

This is a brief report describing an extension of the NU-Prolog system which allows evaluable functions to be defined using equations. The work grew out of frustration concerning many published proposals for combining functional languages and relational languages such as Prolog. Many of these proposals seemed unduly complex, and simply transforming the function definitions into Prolog seemed a much better alternative. This lead to an implementation on top of NU-Prolog several years ago. It has gradually been extended over the years, incorporating some novel implementation techniques, and now supports higher order functions, lazy evaluation and parallel execution. The whole system consists of several hundred lines of Prolog code, and the underlying Prolog system was not modified at all. The system has been give some publicity on the computer networks and is available via anonymous ftp.

However, it was considered that a more formal description of the system was long overdue. This report describes the facilities provided, the implementation techniques used, some insight into issues such as parallelism, and some general conclusions about design of combined functional and relational languages.

2 Syntax

The syntax of NUE-Prolog is identical to NU-Prolog, but some clauses are treated specially. Clauses whose heads are of the form `LHS = EXPR` or `LHS == EXPR` are treated as equations. A group of such equations with the same atom on the left hand side are considered an evaluable function definition. The following code defines the functions `concat/2`, `sum_tree/1` and `+ /2`.

```
        % list concatenation (append)
concat([], A) = A.
concat(A.B, C) = A.concat(B, C).

        % sum integers in a tree
sum_tree(nil) = 0.
sum_tree(t(L, N, R)) = sum_tree(L) + N + sum_tree(R).

        % addition (== forces inline expansion)
A + B == C :- C is quote(A + B). % quote prevents function evaluation
```

There are several restrictions on function definitions (for reasons which will become clear). Function definitions using == can only have a single equation. Function definitions must also be *uniform* [Jon87]. This implies that the left hand sides of all equations must be mutually non-unifiable, which makes the order of equations irrelevant to the declarative semantics. The left hand sides of equations must not contain evaluable function symbols or repeated variables as arguments. All variables appearing on the right hand side must appear on the left hand side or in the body of the clause. Error messages are displayed if these conditions are violated.

Functional code can access Prolog code, such as builtin predicates, by having equations with non-empty clause bodies, as in the definition of + above. Prolog code called in this way should be deterministic and should always succeed (currently this is left up to the programmer to ensure). The system provides a builtin non-strict if-then-else function `(Goal ? TrueExpr : FalseExpr)`. The condition is a Prolog goal, which we feel is more natural than a function call. This provides a interface from functions to Prolog code which can succeed or fail. Because definitions must be uniform, if-then-else must sometimes be used instead of multiple conditional equations. Functions can also be called from NU-Prolog clauses and can be declared in one file but defined elsewhere. The following examples illustrate these points.

```
        % minimum defined elsewhere
?- function minimum/2.

        % maximum of two numbers
maximum(A, B) = (A >= B ? A : B).

        % sum of elements in tree T1 is greater than T2
gt_tree(T1, T2) :- sum_tree(T1) > sum_tree(T2).
```

A simple user interface is provided on top of NU-Prolog. This code is written in Prolog using the predicate `eval/2` provided by the system. It enables users to type an expression and have the result of evaluating the expressions printed:

```
?= concat([1, maximum(1,2), 1 + 2], [4]).
[1, 2, 3, 4]
```

3 Semantics and implementation

The informal meaning of equations is, we hope, quite clear. However, a formal definition of the semantics is also desirable. A theoretical framework in which functions and predicates are

distinct may well be preferable, especially when considering higher order functions and lazy evaluation. However, for the moment we use a simpler approach which is also very close to the implementation. We consider functions to be a special case of predicates and programs defining or using evaluable functions to be a shorthand for pure Prolog programs.

The transformation from functional code to predicate logic is known as the *flattening* transformation. Each definition of a function with `N` arguments is considered shorthand for a predicate definition with `N+1` arguments. Each call to an evaluable function is replaced with a new variable and an extra goal is added to compute the function and bind the new variable to the result. We use a slightly modified version of this transformation for functions defined by `==`. Rather than adding call to the predicate, the body of the predicate definition is unfolded inline. This can be used for defining constants in Prolog code. For example, the definition `radix == 10000` allows the constant `radix` to be used instead of `10000` throughout a Prolog program, without incurring any runtime overhead. The definitions above result in the following NU-Prolog code, which has well defined declarative semantics. NU-Prolog's if-then-else construct is implemented soundly; it generally suspends until the condition is ground.

```
concat([], A1, A1).
concat([A1|A2], A3, [A1|A]) :- concat(A2, A3, A).

sum_tree(nil, 0).
sum_tree(t(A1, A2, A3), A) :-
        sum_tree(A1, B), C is B + A2, % +(B, A2, C) unfolded
        sum_tree(A3, D), A is C + D.

+(A1, A2, A) :- A is A1 + A2.

maximum(A1, A2, A) :- (if A1 >= A2 then A = A1 else A = A2).

gt_tree(A, B) :- sum_tree(A, C), sum_tree(B, D), C > D.
```

The transformation is currently done by a preprocessor, though is could easily be incorporated into the NU-Prolog compiler. The preprocessor reads in a file, determines the evaluable functions (either defined or declared), then performs the flattening transformation on each clause and goal in the file and outputs the result. Extra goals are also output to define what evaluable functions there are. After preprocessing and compilation the files are loaded and these goals are executed, so that at runtime `eval` and other predicates can distinguish between evaluable functions and normal Prolog terms (constructors).

4 Coroutining and indexing

The computation rule of NU-Prolog supports coroutining. The default execution order is left to right, but *when declarations* can be used to force suspension of calls which are insufficiently instantiated. Along with the clauses derived for each function definition, the preprocessor outputs a set of when declarations. For each clause derived from an equation, a when declaration is generated. The head as the same input arguments as the clause head. This forces a call to suspend until its input arguments are subsumed by those in one of the clause heads. Effectively, function calls suspend until they are subsumed by the left hand side of an equations.

Suspending calls with insufficiently instantiated inputs reduces the flexibility of the Prolog code in some ways. Functions cannot be run "backwards", which would be possible in some cases otherwise. However, reversible and nondeterministic procedures can be defined using the normal Prolog syntax, so this is not a great loss. Furthermore, there are many advantages in reserving functional syntax for code which is only used to compute functions. First, we believe it enhances readability of programs. It is helpful to know that a definition is used in a particular mode and is a function, and functional syntax is not a particularly natural syntax for defining more general relations.

Second, suspension can actually make code more flexible rather than less flexible. It allows functions to be used easily with coroutining code. This gives the programmer more freedom in how the logic is expressed, and can greatly reduce the search space in some cases. The reason why control information can easily be generated for functions is that the definitions implicitly state the mode of use. Similarly, functions are implicitly deterministic and this can also be used as control information. In contrast, reasonable control information for predicates is much more difficult to determine, though some heuristics can be used [Nai86].

Third, the fact that a procedure will be used in a particular mode and is completely deterministic can be used to improve the implementation in a variety of ways. It is much easier to determine this information from the syntax than to analyse programs [DW86]. An example of the use of mode information is that the NU-Prolog compiler uses when declarations to guide the choice of clause indexing.

There are two important differences between indexing in functional languages and Prolog. First, the input-output mode of calls is not normally known in advance in Prolog, so full indexing is more expensive than in functional languages (and wasteful for arguments which are always output). Second, because Prolog must support nondeterminism in the form of backtracking, imperfect indexing can have more drastic consequences. If a call only matches with a single clause but indexing is insufficient for this to be determined, a *choice point* is created [War83]. Choice point creation takes time and space and, more importantly, can prevent tail recursion optimization and garbage collection. The equivalent of a functional computation which requires constant space may require non-constant space in Prolog.

Most Prolog implementations use the compromise of indexing on the top level functor of the first argument of each procedure. Procedures which need more complex indexing to avoid choice points can be recoded to use the indexing which is available. A less desirable alternative is to use cut to remove choice points instead of avoiding choice point creation in the first place. If appropiate when declarations are present, the NU-Prolog compiler will index on multiple arguments and on multiple subterms within an argument. With uniform definitions, efficient indexing can be generated so no choice point is ever created. This allows functions to be coded in a natural way without losing the benefits of indexing. The following example illustrates the generation of when declarations and the indexing which is achieved.

```
        % add two lists of numbers, pairwise
add_lists([], _) = [].
add_lists(_._, []) = [].
add_lists(A.As, B.Bs) = (A+B).add_lists(As, Bs).
```

```
        % transformed code
?- add_lists([], A, B) when ever.
?- add_lists([A|B], [], C) when ever.
?- add_lists([A|B], [C|D], E) when ever.
add_lists([], A1, []).
add_lists([A1|A2], [], []).
add_lists([A1|A2], [A3|A4], [A|B]) :- A is A1 + A3, add_lists(A2, A4, B).
```

Note that the first argument in the second equation is _._, rather than simply _. This makes the definition uniform, allowing more effective indexing. The compiled Prolog code initially checks the first argument of the call. If it is a variable, the call suspends. If it is [] the first clause is used. If it is a non-empty list then the second argument is checked. If the second argument is a variable then the call suspends; otherwise the second or third clauses are chosen. This is likely to significantly more efficient than a straightforward coding in Prolog and can be used with coroutines. These advantages come partly from the NU-Prolog implementation, but also from the implicit mode information in the function definition.

Functional code can be run if non-uniform definitions are allowed or a Prolog system which does not support such flexible indexing is used. However, there will be a substantially decrease in efficiency of programs for which the indexing is insufficient to eliminate choice points. Any benchmarking of functional programs transformed into Prolog, for example [CvER90], should therefore take indexing into consideration. Programmers should be made aware of how the code is transformed and what indexing is done by the system so efficient code can be written. For example, with the transformation we use and the standard first argument indexing, all function definitions should have distinct functors in the first argument of the head of each equation. The function above can be recoded using an auxillary function to achieve this.

5 Parallelism

Due to their declarative semantics, both relational and functional languages are promising candidates for parallel execution. The three main forms of parallelism which can be exploited in relational programs are *or-parallelism*, *independent and-parallelism* and *stream and-parallelism* [Gre87]. Or-parallelism exploits parallelism in nondeterministic programs and independent and-parallelism exploits parallelism in conjunctions of atoms which do not share any variables. Both these forms of parallelism have been exploited in conventional Prolog.

Stream and-parallelism exploits parallelism in conjunctions of atoms which may share variables. However, every binding to a shared variable must be deterministic; it cannot be undone at a later stage. This can be achieved by procedure calls suspending until they are sufficiently instantiated and by using pruning operators like Prolog's cut. At first, stream and-parallelism was only exploited by specialized "committed choice" logic programming languages such as Concurrent Prolog and Parlog [Gre87]. These languages require the programmer to supply information such as input/output modes to control when calls suspend.

More recently, the NU-Prolog implementation has been modified so that deterministic code can be run in parallel [Nai88]. Special declarations were introduced in PNU-Prolog so that programmers can provide determinism and mode information with their predicates. The code is transformed and when declarations are added so that the resulting code is guaranteed to be completely deterministic.

The code produced by transforming equations has exactly the same properties as the PNU-

Prolog code and hence can be run in parallel. There is no reason why the output of the preprocessor could not be Parlog code. Again, this is made possible by the implicit mode and determinism information and the decision to restrict the mode of use of evaluable functions. If equations have clause bodies it is vital that the Prolog code is completely deterministic. The bodies of equations could be restricted to only call deterministic builtin predicates or PNU-Prolog procedures which are declared to be deterministic. If such procedures cannot be compiled into deterministic code, an error message is given.

The `sum_tree` function given earlier is a good example of how parallelism can be exploited. A sequential, innermost, left to right functional execution corresponds to a left to right Prolog execution of the flattened definitions. In a parallel functional language, we would expect the two recursive calls to proceed in parallel and when the results are returned the final addition would be done. The same effect can be achieved in Prolog by exploiting independent and-parallelism in the flattened code.

This method does not exploit all potential parallelism however. In an expression such as `sum_tree(build_tree(...))`, the tree is built completely before being summed. Using *dataflow parallelism* in a functional language allows the tree to be incrementally constructed and summed at the same time. This is equivalent to using stream and-parallelism in the flattened code. In the Prolog context, a partially constructed tree is simply a term containing variables. If the summing process proceeds faster than the building process, `sum_tree` will be called with a variable in the first argument. The when declarations will then force the process to suspend until the tree building process catches up and instantiated the variable.

6 Higher order functions

One of the attractive features of functional programming languages is the support of higher order functions. This allows high level descriptions of algorithms to be coded very concisely. Our implementation supports higher order facilities by using Prolog `call` to implement *apply*.

A call to `apply(F, A)` first checks to see if `F` with an additional argument is an evaluable function. If it is not, a term is constructed by adding the additional argument to the function without further evaluation; otherwise the corresponding predicate (with two additional arguments) is called to evaluate the function. For example, if `+ /2` is defined as before, `apply(+, 1)` simply returns the value `+(1)`, whereas `apply(+(1), 2)` calls `+(1, 2, X)` and returns the value 3. The way `apply` treats non-evaluable functions allows us to think in terms of *curried* functions to a limited extent.

We can think of `+` as a function mapping a number to a function mapping a number to a number. Applying `+ /0` to 1 returns a function mapping a number to its successor. Whenever the result of `apply` is a function, a term which represents that function is returned, rather than any evaluation taking place. An advantage of currying which is not currently supported by our system is more concise definitions of functions. For example, `plus` can be defined to be the same function as `+` by the equation `plus = +`, rather than `plus(A, B) = A + B`. We are considering extending the system to allow such definitions with additional declarations which specify how many additional arguments are needed. If type declarations were supported, this would provide enough information.

The definition of apply is currently written mostly in Prolog, rather than the functional syntax. An alternative approach, due to Warren [War82] [CvER90], is to generate a definition of `apply` when transforming the function definitions. For each function with N arguments, N clause for `apply` are generated. An example using `+` is given below. This method results in a

faster system, though it requires more space and is less convenient when the function definitions are spread across several files.

```
apply(+, A, +(A)).
apply(+(A), B, C) :- +(A, B, C).
```

Apply can be used as the basis for more complex higher order functions, as the example below illustrates. Such definitions are expanded in the normal way; the additional complexity occurs at runtime when `apply` is called. The use of curried functions incurs the most overhead, since a new structure representing a function must be returned. We believe the most promising way of reducing the overheads is partial evaluation of programs (Warren's method is very similar to partial evaluation of just the `apply` predicate). It seems simplest to do this at the functional level, though the Prolog level would be more general.

```
map(F, []) = [].
map(F, A.B) = apply(F, A).map(F, B).
```

7 Lazy evaluation

Another important feature of many functional programming languages is lazy evaluation. This allows manipulation of infinite objects and definition of a somewhat simpler semantics for programs. Although coroutining can reduce the search space of Prolog for certain kinds of programs in a similar way to lazy evaluation, it is less powerful than lazy evaluation for deterministic (including functional) programs. Coroutining can only speed up deterministic computations by early detection of failure. For successful deterministic computations, coroutining has no positive effect. In contrast, lazy evaluation can turn an infinite (deterministic) computation into a finite one.

Our system does support optional lazy evaluation however. It is done by using a more complex transformation into Prolog. When declarations are used for the purposes mentioned previously, but the code can still be run on a conventional Prolog system with a left to right computation rule. Lazy evaluation can be invoked my declaring one or more functions to be lazy, as follows.

```
        % generates (lazily) an infinite list of ones
?- lazy ones/0.
ones = 1.ones.
```

7.1 Implementation of lazy evaluation

If any function is declared lazy then all functions are transformed in a more complex way, and the way functional code and Prolog code interacts is more complex. These additional overheads are the reason why we have made lazy evaluation optional. Without lazy evaluation the code is as efficient as straightforward Prolog code, and often more efficient. This is not the case when lazy evaluation is used. With sophisticated dataflow and strictness analysis, it may well

be possible to make all functions lazy with very little overhead. However, this was considered beyond the scope of our relatively simple preprocessor.

When predicates derived from lazy functions are called initially, they simply return a *closure* containing enough information to compute the value if required. If the Prolog call is `p(A,B,C)` then the output variable, `C`, will be bound to the closure `$lazy($lazy$p(A, B, C1), C1)`. The first argument of the closure is a goal which will compute the answer, and the second argument is a variable appearing in the goal which will be bound to the answer. For each lazy function an additional predicate is generated in the same way as non-lazy functions. Typically, this partly instantiates the answer (the top-most functor, for example) then recursively calls lazy functions to compute the rest of the answer. Thus a partial answer which may contain closures is returned. The `ones` function above is translated as follows:

```
ones($lazy($lazy$ones(A), A)). % just returns a closure

?- $lazy$ones(A) when ever.
$lazy$ones([1|A]) :- % produces head of list
        ones(A). % returns a closure for tail of list
```

Returning lazy closures can cause two potential problems. First, the closures can be passed as arguments of functions and not match any of the left hand sides, resulting in failure. Second, they can be passed to Prolog code, including builtin predicates, which may cause failure or error messages. To solve the first problem, equations with structures in the arguments of the left hand side generate additional clauses to evaluate lazy closures. To solve the second problem, wherever the result of a function is passed to Prolog code, full evaluation is forced (by the `evalLazy` predicate, which is written in Prolog). The `add_lists` function given previously is translated as follows (we discuss the algorithm in more detail later):

```
?- add_lists($lazy(B, A), C, D) when ever.
?- add_lists([], A, B) when ever.
?- add_lists([C|D], $lazy(B, A), E) when ever.
?- add_lists([A|B], [], C) when ever.
?- add_lists([A|B], [C|D], E) when ever.
add_lists($lazy(B, A), C, D) :- call(B), add_lists(A, C, D).
add_lists([], A1, []).
add_lists([C|D], $lazy(B, A), E) :- call(B), add_lists([C|D], A, E).
add_lists([A1|A2], [], []).
add_lists([A1|A2], [A3|A4], [A|B]) :-
        evalLazy(A1, C), % fully evaluate A1
        evalLazy(A3, D), % and A3
        A is C + D, % before calling Prolog builtin
        add_lists(A2, A4, B).
```

Consider what occurs in the evaluation of an expression such as `add_lists([1,2], ones)`. The first step is to transform the expression into a Prolog goal: `ones(A), add_lists([1,2], A, B)`. `Ones` initially binds A to `$lazy($lazy$ones(A1), A1)`. The call to `add_lists` matches with the third clause. This is a special clause which attempts to evaluate the closure then recursively calls `add_lists` with the result. The closure is evaluated by calling `$lazy$ones(A1)`, which

binds A1 to [1 | $lazy($lazy$ones(A2), A2)]. This term is passed to the recursive call to add_lists, which now matches with the last clause. This step is repeated, evaluating the second element of the list of ones, then the next recursive call to add_lists matches with the second clause. The closure in the second argument is ignored and the computation terminates with the answer B = [2,3].

Our implementation of lazy functions is very similar to some other lazy functional language implementations. The main difference is that a Prolog term is used to represent the closure containing essentially a reference to some code (the $lazy$ procedure name), a set of argument registers (the arguments to the procedure call) and an address for the returned value (the output variable).

The interface between lazy functional code Prolog is straightforward. Any variable which occurs on the left hand side of an equation and also in the body of a clause or the condition of an if-then-else, must be replaced by two variables and be processed by evalLazy. This does introduce extra overheads, especially for relatively fast operations such as arithmetic. It could be avoided in many cases by analysing the dataflow within the program or having a more sophisticated exception handling mechanism for Prolog builtins.

The clause head matching is very efficient, even in the presence of additional clauses for handling closures. In many cases (add_lists, for example), exactly the same WAM instructions are executed as in the non-lazy case, due to the clause indexing. The algorithm for generating the extra clauses is closely related to the indexing algorithm. With the when declarations produced by the preprocessor, the indexing algorithm constructs a tree which includes nodes for each non-variable subterm in the clause heads. An extra case is added to each of these nodes, to deal with $lazy/2. From this expanded tree, the clause heads for the additional clause can be extracted. Our current implementation is actually simpler than this and makes some assumptions about argument ordering.

7.2 Avoiding repeated evaluation

Most lazy functional language implementations avoid repeated evaluation of duplicated variables. When using a Prolog system with a left to right computation rule this can also be achieved by a minor change to the extra clauses we introduce to evaluate the lazy closures. Rather than calling the goal immediately, the output variable can be checked. If it is already instantiated, calling the goal is unnecessary (it has already been executed). In a coroutining Prolog system, we must be more careful because functions can be called and delay before instantiating the output variable, and output arguments can be instantiated by Prolog code before the function is called. However, with a slightly more complex representation for closures, the problem can be avoided. We can include an extra argument to $lazy which is a flag to indicate whether the goal has been called yet:

```
p(...$lazy(G, V, F)...)  :- (var(F) -> F = 1, call(G) ; true), p(...V...).
```

7.3 Further optimization

The most significant gains to be made in the implementation of lazy evaluation probably come from program analysis (for example, strictness), which lead to certain overheads being removed completely from most code. Much of this high level analysis is independent of the low level details of the implementation. Also, any faster Prolog can be used to speed up our system, since

we can compile to standard Prolog. Recent efforts have resulted in Prolog systems which are faster than C for small recursive programs.

There are also some optimization issues specific to our implementation on top of Prolog. The most important consideration is the indexing produced by the compiler. Whether lazy evaluation is used or not, full indexing of input arguments is very desirable. Second, `call/1` is used for lazy code and `call/3` is used for higher order functions; they should be implemented efficiently. Standard versions of `call` must deal with cut and other system constructs, increasing complexity. A specialised version which only has to deal with certain user-defined predicates can generally be implemented more efficiently. This could be done at the system level or the preprocessor could define such a predicate in a similar manner to Warren's implementation of `apply`. Third, if repeated evaluation is to be avoided, the conditional `(var(V) -> ...)` must also be implemented efficiently, without creating a choice point. All these points are useful for Prolog code also. If the Prolog implementation is to be modified for executing lazy functional code it may be desirable to have a separate tag, and perhaps representation, for the `$lazy` functor (as is currently done with cons). This could make indexing a little faster in functions which used other complex terms, and could save a little space. This is also suggested in the K-LEAF system, discussed later. A special construct for conditionally evaluating a closure could also be provided.

8 Related work

A great deal of work has been done on combining functional and relational languages. Due to lack of space we are only able to breifly discuss those proposals most similar to our own. In [Nai90] we give some more discussion and references, but a full paper is really needed to give justice to this topic.

The most common approach to combining functional and relational languages is to add features of relational languages (nondeterminism, multi-mode relations, logical variables) to a functional language. Functional syntax is used to define more general predicates. Most of these languages are based on narrowing [Red85] [GM85] [JD86] [BCM88] though other methods are also used [DFP85] [Fri85]. An alternative is to include some of the features of functional programs (lazy evaluation, higher order functions) into a relational language, or develop programming techniques to support these styles of programming [Nar86] [She90]. Both these approaches forego the many benefits of knowing what things are functions. A third approach, the one we take, is to support predicates and functions and restrict functional syntax to functions.

The most similar language to that we have proposed is Funlog [SY85]. However, Funlog is implemented by an interpreter written in Prolog, so it is much less efficient. When comparing high level implementation details, the most similar proposal to ours is LOG(F) [Nar88]. LOG(F) is implemented by translation into standard Prolog. However, the way lazy evaluation is implemented introduces one more level of level of functor nesting in the heads of clauses than our scheme. The standard method of indexing cannot be used to distinguish between different cases in a function definition, so choice points cannot be avoided. For simple deterministic code LOG(F) is five to ten times slower than Prolog, and uses much more space. In contrast, our system avoids choice points even with the standard indexing method if functions are coded carefully. Similar code can be obtained by partially evaluating the code produced by the LOG(F) translator. We have also exploited the better indexing of NU-Prolog and made lazy evaluation optional, allowing the convenience of functions with, if anything, increased performance.

From a low level implementation viewpoint, the most similar system to ours is K-LEAF [BCM88]. K-LEAF uses flattening plus other optimizations and is compiled into K-WAM code.

The K-WAM has special support for *prodvars* (closures) and the instructions that involve dereferencing variables (including indexing instructions) are modified. The closures are actually represented as Prolog terms, similar to those in our implementation. The main differences between the K-LEAF and NUE-Prolog implementations are that the K-LEAF compiler relies on a specialized abstract machine and includes additional optimizations, whereas NUE-Prolog is translated into Prolog and the NU-Prolog compiler does better indexing. K-LEAF also supports parallelism [BCM+90]. However, stream and-parallelism is not exploited because K-LEAF functions can be used nondeterministically. An advantage of K-LEAF is its well developed semantics.

9 Conclusions

Both relational and functional styles of programming have many merits. We believe NUE-Prolog provides a useful combination of the two, providing full NU-Prolog plus a functional language which has higher order features and (optional) lazy evaluation. We have used a pragmatic, implementation guided approach to combining the relational and functional paradigms. Our implementation takes advantage of implicit information in function definitions to provide control information, efficient indexing and parallel execution. Our method of implementing lazy evaluation entirely within standard Prolog is also new. We take advantage of the unique features of NU-Prolog, but these features are not necessary for our implementation techniques. We can transform functions into standard Prolog code and thus take advantage of faster Prolog implementations. Further work is needed on programming environment support for evaluable functions and on the semantics of NUE-Prolog. Alternatively, the NUE-Prolog implementation techniques could be applied to languages for which well defined semantics has already been established.

References

[BCM88] P.G. Bosco, C. Cecchi, and C. Moiso. Exploiting the full power of logic plus functional programming. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, pages 3–17, Seattle, Washington, August 1988.

[BCM+90] P.G. Bosco, C. Cecchi, C. Moiso, M. Porta, and G. Sofi. Logic and functional programming on distributed memory architectures. In *Proceedings of the Seventh International Conference on Logic Programming*, Jerusalem, Israel, June 1990.

[CvER90] M.H.M. Cheng, M.H. van Emden, and B.E. Richards. On warren's method for functional programming in logic. In *Proceedings of the Seventh International Conference on Logic Programming*, Jerusalem, Israel, June 1990.

[DFP85] J. Darlington, A.J. Field, and H. Pull. The unification of functional and logic languages. In Doug DeGroot and Gary Lindstrom, editors, *Logic programming: relations, functions, and equations*, pages 37–70. Prentice-Hall, 1985.

[DW86] Saumya K. Debray and David S. Warren. Detection and optimisation of functional computations in prolog. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 490–504, London, England, July 1986. published as Lecture Notes in Computer Science 225 by Springer-Verlag.

[Fri85] Laurent Fribourg. SLOG: a logic programming language interpreter based on clausal superposition and rewriting. In *Proceedings of the Second IEEE Symposium on Logic Programming*, pages 172–184, Boston, Massachusetts, July 1985.

[GM85] Joseph A. Goguen and Jose Meseguer. EQLOG: equality, types, and generic modules for logic programming. In Doug DeGroot and Gary Lindstrom, editors, *Logic programming: relations, functions, and equations*, pages 295–363. Prentice-Hall, 1985.

[Gre87] Steve Gregory. *Parallel logic programming in parlog.* Addison-Wesley, Wokingham, England, 1987.

[JD86] Alan Josephson and Nachum Dershowitz. An implementation of narrowing the RITE way. In *Proceedings of the Third IEEE Symposium on Logic Programming*, pages 187–197, Salt Lake City, Utah, September 1986.

[Jon87] S. Peyton Jones. *The implementation of functional programming languages.* Prentice Hall International series in computer science. Prentice Hall, London, 1987.

[Nai86] Lee Naish. *Negation and control in Prolog.* Number 238 in Lecture Notes in Computer Science. Springer-Verlag, New York, 1986.

[Nai88] Lee Naish. Parallelizing NU-Prolog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, pages 1546–1564, Seattle, Washington, August 1988.

[Nai90] Lee Naish. Adding equations to NU-prolog. Technical Report 91/2, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1990.

[Nar86] Sanjai Narain. A technique for doing lazy evaluation in logic. *Journal of Logic Programming*, 3(3):259–276, October 1986.

[Nar88] Sanjai Narain. *LOG(F): An optimal combination of logic programming, rewriting and lazy evaluation.* Ph.d. thesis, Dept. of computer science, UCLA, Los Angeles, CA, 1988.

[Red85] Uday S. Reddy. Narrowing as the operational semantics of functional languages. In *Proceedings of the Second IEEE Symposium on Logic Programming*, pages 138–151, Boston, Massachusetts, July 1985.

[She90] Yeh-Heng Sheng. HIFUNLOG: logic programming with higher-order relational functions. In *Proceedings of the Seventh International Conference on Logic Programming*, Jerusalem, Israel, June 1990.

[SY85] P.A. Subrahmanyam and Jia-Huai You. FUNLOG: a computational model integrating logic programming and functional programming. In Doug DeGroot and Gary Lindstrom, editors, *Logic programming: relations, functions, and equations*, pages 157–198. Prentice-Hall, 1985.

[War82] David H.D. Warren. Higher-order extensions to prolog: are they needed? In J.E. Hayes, Donald Michie, and Y-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd., Chicester, England, 1982.

[War83] David H.D. Warren. An abstract Prolog instruction set. Tecnical Note 309, SRI International, Menlo Park, California, October 1983.