A Syntactic Approach to Combining Functional Notation, Lazy Evaluation, and Higher-Order in LP Systems

Amadeo Casas[1], Daniel Cabeza[2], and Manuel V. Hermenegildo[1,2]

[1] Depts. of Comp. Science and Electr. and Comp. Eng., Univ. of New Mexico, USA
`{amadeo, herme}@cs.unm.edu`
[2] School of Computer Science, T.U. Madrid (UPM), Spain
`{dcabeza, herme}@fi.upm.es`

Abstract. Nondeterminism and partially instantiated data structures give logic programming expressive power beyond that of functional programming. However, functional programming often provides convenient syntactic features, such as having a designated implicit output argument, which allow function call nesting and sometimes results in more compact code. Functional programming also sometimes allows a more direct encoding of lazy evaluation, with its ability to deal with infinite data structures. We present a *syntactic* functional extension, used in the Ciao system, which can be implemented in ISO-standard Prolog systems and covers function application, predefined evaluable functors, functional definitions, quoting, and lazy evaluation. The extension is also composable with higher-order features and can be combined with other extensions to ISO-Prolog such as constraints. We also highlight the features of the Ciao system which help implementation and present some data on the overhead of using lazy evaluation with respect to eager evaluation.

Keywords: Declarative Languages; Logic, Functional, and Logic-Functional Programming; Lazy Evaluation; Higher Order.

1 Introduction

Logic Programming offers a number of features, such as nondeterminism and partially instantiated data structures, that give it expressive power beyond that of functional programming. However, certain aspects of functional programming provide in turn syntactic convenience. This includes for example having a syntactically designated output argument, which allows the usual form of function call nesting and sometimes results in more compact code. Also, lazy evaluation, which brings the ability to deal with infinite (non-recursive) data structures [1, 2], while subsumed operationally by logic programming features such as delay declarations, enjoys a more direct encoding in functional programming. Bringing this syntactic convenience to logic programming can result in a more compact program representation in certain cases and is therefore a desirable objective.

With this objective in mind, in this paper we present a design for an extensive functional syntactic layer for logic programing. While the idea of adding functional features to logic programming systems is clearly not new, and there are currently a good number of systems which integrate functions and higher-order programming into some form of logic programming, we feel that our proposal and its implementation offer a combination of features which make it interesting in itself (see Section 6 for a discussion of related work).

Our approach was inspired by some of the language extension capabilities of the Ciao system [3]: Ciao offers a complete ISO-Prolog system, but one of its most remarkable features is that, through a novel modular design [5], all ISO-Prolog features are library-based extensions to a simple declarative kernel. This allows on one hand not loading any (for example, impure) features from ISO-Prolog when not needed, and on the other hand adding many additional features at the source (Prolog) level, without modifying the compiler or the low-level machinery. The facilities that allow this (grouped under the Ciao *packages* concept [5]) are the same ones used for implementing the functional extensions proposed herein, and are also the mechanism by which other syntactic and semantic extensions are supported in the system. The latter include constraints, objects, feature terms/records, persistence, several control rules, etc., giving Ciao its multi-paradigm flavor.

However, while the Ciao extension mechanisms make implementation smoother and more orthogonal in our view,[1] a fundamental design objective and feature of our functional extensions is that they are to a very large extent directly applicable to (and also relatively straightforward to implement in) any modern (ISO-)Prolog system [6], and we hope to contribute in that way to their adoption in such systems. Thus, we will also discuss ISO-Prolog when describing the implementation of the proposed extensions.

The rest of the paper is organized as follows: first, we discuss in Section 2 our general approach to integrating functional notation. Section 3 presents how we implemented this approach in Ciao. Section 4 shows an example of the use of lazy evaluation, and how it is achieved by our implementation. Section 5 presents some experimental results. Finally, section 6 presents our conclusions and discusses related work.

2 Functional Notation in Ciao

Basic Concepts and Notation: Our notion of functional notation for logic programming departs in a number of ways from previous proposals. The fundamental one is that functional notation in principle simply provides *syntactic*

[1] As we will try to highlight with the upfront intention of motivating the adoption of the extension model by other logic programming systems.

sugar for defining and using predicates as if they were functions, but they can still retain the power of predicates. In this model, any function definition is in fact defining a predicate, and any predicate can be used as a function. The predicate associated with a function has the same name and one more argument, meant as the place holder for the result of the function. This argument is by default added to the right, i.e., it is the last argument, but this can be changed by using a declaration. The syntax extensions provided for functional notation are the following:

Function applications: Any term preceded by the `~` operator is a function application, as can be seen in the goal `write(~arg(1,T))`, which is strictly equivalent to the sequence `arg(1,T,A), write(A)`. To use a predicate argument other than the last as the return argument, a declaration like:
`:- fun_return functor(~,_,_).`
can be used, so that `~functor(f,2)` is evaluated to `f(_,_)` (where `functor/3` is the standard ISO-Prolog builtin). This definition of the return argument can also be done on the fly in each invocation in the following way: `~functor(~,f,2)`. Functors can be declared as *evaluable* (i.e., being in calls in functional syntax) by using the declaration `fun_eval/1`. This allows avoiding the need to use the `~` operator. Thus, "`:- fun_eval arg/2.`" allows writing `write(arg(1,T))` instead of `write(~arg(1,T))` as above. This declaration can also be used to change the default output argument:
`:- fun_eval functor(~,_,_).`
Note that all these declarations, as is customary in Ciao, are local to the module where they are included.

Predefined evaluable functors: In addition to functors declared with the declaration `fun_eval/1`, several functors are evaluable, those being:

- The functors used for disjunctive and conditional expressions, `(|)/2` and `(?)/2`. A disjunctive expression has the form `(V1|V2)`, and its value when first evaluated is `V1`, and on backtracking `V2`. A conditional expression has the form `(Cond ? V1)`, or, more commonly, `(Cond ? V1 | V2)`. If the execution of `Cond` as a goal succeeds the return value is `V1`. Otherwise in the first form it causes backtracking, and in the second form its value is `V2`. Due to operator precedences, a nested expression
`(Cond1 ? V1 | Cond2 ? V2 | V3)`
is evaluated as `(Cond1 ? V1 | (Cond2 ? V2 | V3))`.
- If the declaration `:- fun_eval arith(true)` is used, all the functors understood by `is/2` are considered evaluable (they will be translated to a call to `is/2`). This is not active by default because several of those functors, like `(-)/2` or `(/)/2`, are traditionally used in Prolog for creating structures. Using `false` instead of `true` the declaration can be disabled.

Functional definitions: A functional definition is composed of one or more functional clauses. A functional clause is written using the binary operator `:=`, as in `opposite(red) := green.`
Functional clauses can also have a body, which is executed before the result value is computed. It can serve as a guard for the clause or to provide the equivalent of where-clauses in functional languages:

```
fact(0) := 1.
fact(N) := N * ~fact(--N) :- N > 0.
```

Note that guards can often be defined more compactly using conditional expressions:

```
fact(N) := N = 0 ? 1
         | N > 0 ? N * ~fact(--N).
```

If the declaration `:- fun_eval defined(true)` is active, the function defined in a functional clause does not need to be preceded by ~ (for example the `fact(--N)` calls above).

The translation of functional clauses has the following properties:

- The translation produces *steadfast* predicates [7], that is, output arguments are unified after possible cuts.
- Defining recursive predicates in functional style maintains the tail recursion of the original predicate, thus allowing the usual compiler optimizations.

Quoting functors: Functors (either in functional or predicate clauses) can be prevented from being evaluated by using the `(^)/1` prefix operator (read as "quote"), as in:
`pair(A,B) := ^(A-B).`
Note that this just prevents the evaluation of the principal functor of the enclosed term, not the possible occurrences of other evaluable functors inside.

Scoping: When using function applications inside the goal arguments of meta-predicates, there is an ambiguity as they could be evaluated either in the scope of the outer execution or in the scope of the inner execution. The default behavior is to evaluate function applications in the scope of the outer execution. If they should be evaluated in the inner scope the goal containing the function application needs to be escaped with the `(^^)/1` prefix operator, as in `findall(X, (d(Y), ^^(X = ~f(Y)+1)), L)` (which could also be written as `findall(X, ^^(d(Y), X = ~f(Y)+1), L)`), and whose expansion is `findall(X, (d(Y),f(Y,Z),T is Z+1,X=T), L)`. With no escaping the function application is evaluated in the scope of the outer execution, i.e., `f(Y,Z), T is Z+1, findall(X, (d(Y),X=T), L)`.

Laziness: Lazy evaluation is a program evaluation technique used particularly in functional languages. When using lazy evaluation, an expression is not evaluated as soon as it is assigned, but rather when the evaluator is forced to produce the value of the expression. The `when`, `freeze`, or `block` control primitives present in many modern logic programming systems are more powerful operationally than lazy evaluation. However, they lack the simplicity of use and cleaner semantics of functional lazy evaluation. In our design, a function (or predicate) can be declared as lazy via the declarations:
`:- lazy fun_eval function_name/N.`
(or, equivalently in predicate version, "`:- lazy pred_name/M.`", where $M = N + 1$). In order to achieve the intended behavior, the execution of each function declared as lazy is suspended until the return value of the function

is needed. Thus, lazy evaluation allows dealing with infinite data structures and also evaluating function arguments only when needed.

Definition of real functions: In the previous scheme, functions are (at least by default) not forced to provide a single solution for their result, and, furthermore, they can be partial, producing a failure when no solution can be found. A predicate defined as a function can be declared to behave as a real function using the declaration "`:- funct name/N.`". Such predicates are then converted automatically to real functions by adding pruning operators and a number of Ciao assertions [8] which pose (and check) additional restrictions such as determinacy, modedness, etc., so that the semantics will be the same as in traditional functional programming.

We now illustrate with examples the use of the functionality introduced above.

Example 1. The following example defines a simple unary function `der(X)` which returns the derivative of a polynomial arithmetic expression:

```
der(x)       := 1.
der(C)       := 0                  :- number(C).
der(A + B)   := der(A) + der(B).
der(C * A)   := C * der(A)         :- number(C).
der(x ** N)  := N * x ** ~(N - 1)  :- integer(N), N > 0.
```

Note that if we include the directive mentioned before which makes arithmetic functors evaluable then we would have to write the program in the following (clearly, less pleasant and more obfuscated) way:

```
:- fun_eval(arith(true)).
der(x)          := 1.
der(C)          := 0                        :- number(C).
der(^(A + B))   := ^(der(A) + der(B)).
der(^(C * A))   := ^(C * der(A))            :- number(C).
der(^(x ** N))  := ^(N * ^(x ** (N - 1)))  :- integer(N), N > 0.
```

Both of the previous code fragments translate to the following code:

```
der(x, 1).                    der(C * A, C * X) :-
der(C, 0) :-                            number(C),
          number(C).                    der(A, X).
der(A + B, X + Y) :-          der(x ** N, N * x ** N1) :-
          der(A, X),                    integer(N),
          der(B, Y).                    N > 0,
                                        N1 is N - 1.
```

Note that in all cases the programmer may use `der/2` as a function or as a predicate indistinctly.

Example 2. Functional notation interacts well with other language extensions. For example, it provides compact and familiar notation for regular types and other properties (assume `fun_eval` declarations for them):

```
color       := red | blue | green.
list        := [] | [_ | list].
list_of(T) := [] | [~T | list_of(T)].
```

which are equivalent to (note the use of higher-order in the third example):

```
color(red). color(blue). color(green).
list([]).
list([_|T]) :- list(T).
list_of(_, []).
list_of(T, [X|Xs]) :- T(X), list_of(T, Xs).
```

Such types and properties are then admissible in Ciao-style assertions [8], such as the following, and which can be added to the corresponding definitions and checked by the preprocessor or turned into run-time tests [9]:

```
:- pred append/3 :: list * list * list.
:- pred color_value/2 :: list(color) * int.
```

Example 3. The combination of functional syntax and user-defined operators brings significant flexibility, as can be seen in the following definition of a list concatenation (`append`) operator:[2]

```
:- op(600, xfy, (.)).
:- op(650, xfy, (++)).
:- fun_eval (++)/2.
[]   ++ L := L.
X.Xs ++ L := X.(Xs ++ L).
```

This definition will be compiled exactly to the standard definition of `append` in Prolog (and, thus, will be reversible). The functional syntax and user-defined operators allow writing for example `write("Hello" ++ Spc ++ "world!")` instead of the equivalent forms `write( append("Hello", append(Spc, "world!")))` (if `append/2` is defined as evaluable) or `append(Spc, "world!", T1), append("Hello", T1, T2), write(T2)`.

Example 4. As another example, we define an array indexing operator for multi-dimensional arrays. Assume that arrays are built using nested structures whose main functor is '`a`' and whose arities are determined by the specified dimensions, i.e., a two-dimensional array A of dimensions $[N, M]$ will be represented by the nested structure `a(a(`A_{11}`,...,`A_{1M}`), a(`A_{21}`,..,`A_{2M}`), ..., a(`A_{N1}`,...,` A_{NM}`))`, where A_{11}`,...` A_{NM} may be arbitrary terms.[3] The following recursive definition defines the property `array/2` and also the array access operator `@`:

[2] This operator, as well of other conveniences to be able to program in a more functional-flavored style, are defined in an additional Ciao package.

[3] We ignore for simplicity possible arity limitations, solved in any case typically by further nesting with logarithmic access time (as in Warren/Pereira's classical library).

```
array([N],A) :-
        functor(A,a,N).
array([N|Ms],A) :-
        functor(A,a,N),
        rows(N,Ms,A).

:- op(55, xfx, '@').
:- fun_eval (@)/2.
V@[I]    := ~arg(I,V).       %% Or: V@[] := V.
V@[I|Js] := ~arg(I,V)@Js.
```

```
rows(0,_,_).
rows(N,Ms,A) :-
        N > 0,
        arg(N,A,Arg),
        array(Ms,Arg),
        rows(N-1,Ms,A).
```

This allows writing, e.g., `M = array([2,2]), M@[2,1] = 3` (which could also be expressed as `array([2,2])@[2,1] = 3`), where the call to the `array` property generates an empty 2×2 array M and `M@[2,1] = 3` puts 3 in $M[2,1]$. Another example would be: `A3@[N+1,M] = A1@[N-1,M] + A2@[N,M+2]`.

Example 5. As a simple example of the use of *lazy evaluation* consider the following definition of a function which returns the (potentially) infinite list of integers starting with a given one:

```
:- lazy fun_eval nums_from/1.
nums_from(X) := [ X | nums_from(X+1) ].
```

Ciao provides in its standard library the **`hiord`** package, which supports a form of higher-order untyped logic programming with predicate abstractions [10, 11, 12]. Predicate abstractions are Ciao's translation to logic programming of the lambda expressions of functional programming: they define unnamed predicates which will be ultimately executed by a higher-order call, unifying its arguments appropriately.[4] A function abstraction is provided as functional syntactic sugar for predicate abstractions:

Predicate abstraction	⇒	Function abstraction
`{''(X,Y) :- p(X,Z), q(Z,Y)}`	⇒	`{''(X) := ~q(~p(X))}`

and function application is syntactic sugar over predicate application:

Predicate application	⇒	Function application
`..., P(X,Y), ...`	⇒	`..., Y = ~P(X), ...`

The combination of this `hiord` package with the `fsyntax` and `lazy` packages (and, optionally, the type inference and checking provided by the Ciao preprocessor [9]) basically provide the functionality present in modern functional languages,[5] as well as some of the functionality of higher-order logic programming.

Example 6. This `map` example illustrates the combination of functional syntax and higher-order logic programming:

[4] A similar concept has been developed independently for Mercury, but there higher-order predicate terms have to be moded.

[5] Currying is not syntactically implemented, but its results can be obtained by deriving higher-order data from any other higher-order data (see [11]).

```
:- fun_eval map/2.
map([], _)      := [].
map([X|Xs], P) := [P(X) | map(Xs, P)].
```

With this definition, after calling:

```
["helloworld", "byeworld"] = map(["hello", "bye"], ++(X)).
```

(where `(++)/2` corresponds to the above definition of `append`) `X` will be bound to `"world"`, which is the only solution to the equation. Also, when calling:

```
map(L, ++(X), ["hello.", "bye."]).
```

several values for `L` and `X` are returned through backtracking:

```
L = ["hello","bye"],   X = "." ? ;
L = ["hello.","bye."], X = [] ?
```

3 Implementation Details

As mentioned previously, certain Ciao features have simplified the proposed extension to handle functional notation. In the following we introduce the features of Ciao that were used and how they were applied in this particular application.

Code Translations in Ciao. Traditionally, Prolog systems have included the possibility of changing the syntax of the source code through the use of the `op/3` builtin/directive. Furthermore, in many Prolog systems it is also possible to define *expansions* of the source code (essentially, a very rich form of "macros") by allowing the user to define (or extend) a predicate typically called `term_expansion/2` [13, 14]. This is usually how, e.g., definite clause grammars (DCG's) are implemented.

However, these features, in their original form, pose many problems for modular compilation or even for creating sensible standalone executables. First, the definitions of the operators and, specially, expansions are often global, affecting a number of files. Furthermore, it is not possible to determine statically which files are affected, because these features are implemented as a side-effect, rather than a declaration: they become active immediately after being read by the code processor (top-level, compiler, etc.) and remain active from then on. As a result, it is impossible just by looking at a source code file to know if it will be affected by expansions or definitions of operators, which may completely change what the compiler really sees, since those may be activated by the load of other, possibly unrelated, files.

In order to solve these problems, the syntactic extension facilities were redesigned in Ciao, so that it is still possible to define source translations and operators, but such translations are local to the module or user file defining them [5]. Also, these features are implemented in a way that has a well-defined behavior in the context of a standalone compiler, separate compilation, and global analysis (and this behavior is implemented in the Ciao compiler, `ciaoc` [15]). In particular, the `load_compilation_module/1` directive allows separating code that will be used at compilation time (e.g., the code used for program transformations) from code which will be used at run-time. It loads the module defined by its argument *into the compiler.*

In addition, in order to make the task of writing source translations easier, the effects usually achieved through `term_expansion/2` can be obtained in Ciao by means of four different, more specialized directives, which, again, *affect only the current module* and *are (by default) only active at compile-time.* The proposed functional syntax is implemented in Ciao using these source translations. In particular, we have used the `add_sentence_trans/1` and `add_goal_trans/1` directives. A sentence translation is a predicate which will be called by the compiler to possibly convert each *term* (clause, fact, directive, input, etc.) read by the compiler to a new term, which will be used in place of the original term. A goal translation is a predicate which will be called by the compiler to possibly convert each *goal* present in each clause of the current text to another goal which replaces the original one. The proposed model can be implemented in Prolog systems similarly using the traditional `term_expansion/2` and operator declarations, but having operators and syntactic transformation predicates local to modules is the key to making the approach scalable and amenable to combination with other packages and syntactic extensions in the same application.

Ciao Packages. Packages in Ciao are libraries which define extensions to the language, and have a well defined and repetitive structure. These libraries typically consist of a main source file which defines only some declarations (operator declarations, declarations loading other modules into the compiler or the module using the extension, etc.). This file is meant to be *included* as part of the file using the library, since, because of their local effect, such directives must be part of the code of the module which uses the library. Any auxiliary code needed at compile-time (e.g., translations) is included in a separate module which is to be loaded into the compiler via a `load_compilation_module/1` directive placed in the main file. Also, any auxiliary code to be used at run-time is placed in another module, and the corresponding `use_module` declaration is also placed in the include file.

In our implementation of functional notation in Ciao we have provided two packages: one for the bare function features without lazy evaluation, and an additional one to provide the lazy evaluation features. The reason for this is that in many cases the lazy evaluation features are not needed and thus the translation procedure is simplified.

The Ciao Implementation of Functional Extensions. To translate the functional definitions, we have used as mentioned above the `add_sentence_trans/1` directive to provide a translation procedure which transforms each functional clause to a predicate clause, adding to the function head the output argument, in order to convert it to the predicate head. This translation procedure also deals with functional applications in heads, as well as with `fun_eval` directives. Furthermore, all function applications are translated to an internal normal form.

On the other hand, we have used the `add_goal_trans/1` directive to provide a translation procedure for dealing with function applications in bodies (which were previously translated to a normal form). The rationale for using a goal translation is that each function application inside a goal will be replaced by a variable, and the goal will be preceded by a call to the predicate which implements the function in order to provide a value for that variable. A simple recursive application of this rule achieves the desired effect.

An additional sentence translation is provided to handle the `lazy` directives. The translation of a lazy function into a predicate is done in two steps. First, the function is converted into a predicate using the procedure sketched above. Then, the resulting predicate is transformed in order to suspend its execution until the value of the output variable is needed. We explain the transformation in terms of the `freeze/1` control primitive that many modern logic programming systems implement quite efficiently [16], since it is the most widespread (but obviously `when` [17] or, specially, the more efficient `block` [16] declarations can also be used). This transformation renames the original predicate to an internal name and add a *bridge predicate* with the original name which invokes the internal predicate through a call to `freeze/2`, with the last argument (the output of the function) as suspension variable. This will delay the execution of the internal predicate until its result is required, which will be detected as a binding (i.e., demand) of its output variable. The following section will provide a detailed example of the translation of a lazy function. The implementation with `block` is even simpler since no bridge predicate is needed.

We show below, for reference, the main files for the Ciao library packages `fsyntax`:

```
% fsyntax.pl
:- include(library('fsyntax/ops')).  %% Operator definitions
:- load_compilation_module(library('fsyntax/functionstr')).
:- add_sentence_trans(defunc/3).
:- add_goal_trans(defunc_goal/3).
```

and `lazy` (which will usually be used in conjunction with the first one):

```
% lazy.pl
:- include(library('lazy/ops')). %% Operator definitions
:- use_module(library(freeze)).
:- load_compilation_module(library('lazy/lazytr')).
:- add_sentence_trans(lazy_sentence_translation/3).
```

These files will be *included* in any file that uses the package. The Ciao system source provides the actual detailed code, which follows the our description.

4 Lazy Functions: An Example

In this section we show an example of the use of lazy evaluation, and how a lazy function is translated by our Ciao package. Figure 1 shows in the first row the definition of a lazy function which returns the infinite list of Fibonacci numbers, in the second row its translation into a lazy predicate[6] (by the `fsyntax` package)

[6] The `:- lazy fun_eval fiblist/0.` declaration is converted into a `:- lazy fiblist/1.` declaration.

```
:- lazy fun_eval fiblist/0.
fiblist := [0, 1 | ~zipWith(+, FibL, ~tail(FibL))]
        :- FibL = fiblist.
```

```
:- lazy fiblist/1.
fiblist([0, 1 | Rest]) :-
          fiblist(FibL),
          tail(FibL, T),
          zipWith(+, FibL, T, Rest).
```

```
fiblist(X) :-
          freeze(X, fiblist_lazy_$$$(X)).

fiblist_lazy_$$$([0, 1 | Rest]) :-
          fiblist(FibL),
          tail(FibL, T),
          zipWith(+, FibL, T, Rest).
```

Fig. 1. Code translation for a Fibonacci function, to be evaluated lazily

and in the third row the expansion of that predicate to emulate lazy evaluation (where `fiblist_lazy$$$` stands for a fresh predicate name).

In the `fiblist` function defined, any element in the resulting *infinite* list of Fibonacci numbers can be referenced, as, for example, `nth(X, ~fiblist, Value)`. The other functions used in the definition are `tail/2`, which is defined as lazy and returns the tail of a list; `zipWith/3`, which is also defined as lazy and returns a list whose elements are computed by a function having as arguments the successive elements in the lists provided as second and third argument;[7] and `(+)/2` which is defined as by the rule `+(X, Y) := Z :- Z is X + Y`.

Note that the `zipWith/3` function (respectively the `zipWith/4` predicate) is in fact a *higher-order* function (resp. predicate).

5 Some Performance Measurements

Since the functional extensions proposed simply provide a syntactic bridge between functions and predicates, there are only a limited number of performance issues worth discussing. For the case of *real* functions, it is well known that performance gains can be obtained from the knowledge that the corresponding predicate is moded (all input arguments are ground and the "designated output" will be ground on output), determinate, non-failing, etc. [18, 20]. In Ciao this information can in general (i.e., for any predicate or function) be inferred by the Ciao preprocessor or declared with Ciao assertions [9, 8]. As mentioned

[7] It has the same semantics as the `zipWith` function in Haskell.

```
:- fun_eval nat/1.
nat(N) := ~take(N, nums_from(0)).

:- lazy fun_eval nums_from/1.
nums_from(X) :=
     [X | nums_from(X+1)].
```

```
:- fun_eval nat/1.
:- fun_eval nats/2.
nat(X) := nats(0, X).
nats(X, Max) := X > Max ? []
             | [X | nats(X+1, Max)].
```

Fig. 2. Lazy and eager versions of function `nat(X)`

before, for declared "real" (`func`) functions, the corresponding information is added automatically. Some (preliminary) results on current Ciao performance when this information is available are presented in [20].

In the case of lazy evaluation of functions, the main goal of the technique presented herein is not really any increase in performance, but achieving new functionality and convenience through the use of code translations and delay declarations. However, while there have also been some studies of the overhead introduced by delay declarations and their optimization (see, e.g., [21]), it is interesting to see how this overhead affects our implementation of lazy evaluation by observing its performance. Consider the `nat/2` function in Figure 2, a simple function which returns a list with the first N numbers from an (infinite) list of natural numbers.

Function `take/2` in turn returns the list of the first N elements in the input list. This `nat(N)` function cannot be directly executed eagerly due to the infinite list provided by the `nums_from(X)` function, so that, in order to compare time and memory results between lazy and eager evaluation, an equivalent version of that function is provided.

Table 1 reflects the time and memory overhead of the lazy evaluation version of `nat(X)` and that of the equivalent version executed eagerly. As a further example, Table 2 shows the results for a quicksort function executed lazily in comparison to the eager version of this algorithm. All the results were obtained by averaging ten runs on a medium-loaded Pentium IV Xeon 2.0Ghz, 4Gb of RAM memory, running Fedora Core 2.0, with the simple translation of Figure 1, and compiled to traditional bytecode (no global optimizations or native code).

We can observe in both tables that there is certainly an impact on the execution time when functions are evaluated lazily, but even with this version the results are quite acceptable if we take into account that the execution of the predicate does really suspend. Related to memory consumption we show heap sizes, without garbage collection (in order to observe the raw memory consumption rate). Lazy evaluation implies as expected some memory overhead due to the need to copy (freeze) program goals into the heap. Also, while comparing with standard lazy functional programming implementations is beyond the scope of this paper, some simple tests done for sanity check purposes (with HUGS) show that the results are comparable, our implementation being for example slower on `nat` but faster on `qsort`, presumably due to the different optimizations being performed by the compilers.

Table 1. Performance for `nat/2` (time in ms. and heap sizes in bytes)

	Lazy Evaluation		**Eager Evaluation**	
List	**Time**	**Heap**	**Time**	**Heap**
10 elements	0.030	1503.2	0.002	491.2
100 elements	0.276	10863.2	0.016	1211.2
1000 elements	3.584	104463.0	0.149	8411.2
2000 elements	6.105	208463.2	0.297	16411.2
5000 elements	17.836	520463.0	0.749	40411.2
10000 elements	33.698	1040463.0	1.277	80411.2

Table 2. Performance for `qsort/2` (time in ms. and heap sizes in bytes)

	Lazy Evaluation		**Eager Evaluation**	
List	**Time**	**Heap**	**Time**	**Heap**
10 elements	0.091	3680.0	0.032	1640.0
100 elements	0.946	37420.0	0.322	17090.0
1000 elements	13.303	459420.0	5.032	253330.0
5000 elements	58.369	2525990.0	31.291	1600530.0
15000 elements	229.756	8273340.0	107.193	5436780.0
20000 elements	311.833	11344800.0	146.160	7395100.0

An example when lazy evaluation can be a better option than eager evaluation in terms of performance (and not only convenience) can be found in a concurrent or distributed system environment (such as, e.g., [22]), and in the case of Ciao also within the active modules framework [3, 23]. The example in Figure 3 uses a function, defined in an active module, which returns a big amount of data. Function `test/0` in module `module1` needs to execute function `squares/1`, in (active, i.e., remote) module `module2`, which will return a very long list (which could be infinite for our purposes). If `squares/1` were executed eagerly then the entire list would be returned, to immediately execute the `takeWhile/2` function with the entire list. `takeWhile/2` returns the first elements of a (possibly infinite) list while the specified condition is true. But creating the entire initial list is very wasteful in terms of time and memory requirements. In order to solve this problem, the `squares/1` function could be moved to module `module1` and merged with `takeWhile/2` (or, also, they could exchange a size parameter). But rearranging the program is not always possible and it may also perhaps complicate other aspects of the overall design.

If on the other hand `squares/1` is evaluated lazily, it is possible to keep the definitions unchanged and in different modules, so that there will be a smaller time and memory penalty for generating and storing the intermediate result. As more values are needed by the `takeWhile/2` function, more values in the list returned by `squares/1` are built (in this example, only while the new generated value is less than 10000), considerably reducing the time and memory consumption that the eager evaluation would take.

```
:- module(module1, [test/1], [fsyntax, lazy, hiord, actmods]).
:- use_module(library('actmods/webbased_locate')).

:- use_active_module(module2, [squares/2]).

:- fun_eval takeWhile/2.
takeWhile(P, [H|T]) := P(H) ? [H | takeWhile(P, T)]
                            | [].
:- fun_eval test/0.
test := takeWhile(condition, squares).
condition(X) :- X < 10000.
```

```
\vspace*{6mm}
:- module(module2, [squares/1], [fsyntax, lazy, hiord]).

:- lazy fun_eval squares/0.
squares := map_lazy(take(1000000, nums_from(0)), square).

:- lazy fun_eval map_lazy/2.
map_lazy([], _)     := [].
map_lazy([X|Xs], P) := [~P(X) | map_lazy(Xs, P)].

:- fun_eval take/2.
take(0, _)     := [].
take(X, [H|T]) := [H | take(X-1, T)] :- X > 0.

:- lazy fun_eval nums_from/1.
nums_from(X) := [X | nums_from(X+1)].

:- fun_eval square/1.
square(X) := X * X.
```

Fig. 3. A distributed (active module) application using lazy evaluation

6 Conclusions and Related Work

As mentioned in the introduction, the idea of adding functional features to logic programming systems is clearly not new [24, 25, 17] and there are currently a good number of systems which integrate functions and higher-order programming into some form of logic programming. However, we feel that our proposal and its implementation offer a combination of features which make it interesting in itself. More concretely, the approach is completely syntactic, functions can be limited or retain the power of predicates, any predicate can be called through functional syntax, and lazy evaluation is supported both for functions and predicates. Furthermore, functional syntax can be combined with numerous (Ciao) syntactic and semantic extensions such as higher-order, assertions, records, constraints, objects, persistence, other control rules, etc., without any modification to the compiler or abstract machine. Finally, and perhaps most importantly, and again because of the syntactic nature of the extensions, they can be the target of analysis, optimization, static checking, and verification (of types, modes, determinacy, nonfailure, cost, etc.), as performed by, e.g., the Ciao preprocessor [9]. Finally, another important characteristic of our approach is that most of it can be applied directly (or with minor changes) to any ISO-standard Prolog system.

The original version of the functional extensions was first distributed in Ciao 0.2 [4] and later used as an example in [5]. The full description presented herein includes some minor changes with respect to the currently distributed version [3] which will be available in the next release. The performance of the package for lazy evaluation was tested in this system with several examples. As expected, lazy evaluation implies time and memory overhead, which justifies making lazy evaluation optional via a declaration.

Returning to the issue of related work, Lambda Prolog [26] offers a highly expressive language with extensive higher-order programming features and lambda-term (pattern) unification. On the other hand it pays in performance the price of being "higher order by default," and is not backwards compatible with traditional Prolog systems. It would be clearly interesting to support pattern unification, but we propose to do it as a further (and optional) extension, and some work is in progress along these lines. HiLog [27] is a very interesting logic programming system (extending XSB-Prolog) which allows using higher-order syntax, but it does not address the issue of supporting functional syntax or lazyness. Functional-logic systems such as Curry or Babel [31, 32] perform a full integration of functional and logic programming, with higher-order support. On the other hand, their design starts from a lazy functional syntax and semantics, and is strongly typed. However, it may also be interesting to explore supporting narrowing as another optional extension. Mercury [28] offers functional and higher-order extensions based on Prolog-like syntax, but they are an integral part of the language (as opposed to an optional extension) and, because of the need for type and mode declarations, the design is less appropriate for non strongly-typed, unmoded systems. As mentioned above, in our design type and mode declarations are optional and handled separately through the assertion mechanism. Also, Mercury's language design includes a number restrictions with respect to Prolog-like systems which bring a number of implementation simplifications. In particular, the modedness (no unification) of Mercury programs brings them much closer to the functional case. As a result of these restrictions, Mercury always performs the optimizations pointed out when discussing our `funct` declaration (or when that type of information is inferred by CiaoPP).[8] Oz [30] also allows functional and (a restricted form of) logic programming, and supports higher-order in an untyped setting, but its syntax and semantics are quite different from those of LP systems. BIM Prolog offered similar functionality to our `~/2` operator but, again, by default and as a builtin.

[8] However, recent extensions to support constraints [29] recover unification, including the related implementation overheads and mechanisms (such as the trail), and will require analysis for optimization, moving Mercury arguably closer to Ciao in design.

References

1. Narain, S.: Lazy evaluation in logic programming. In: Proc. 1990 Int. Conference on Computer Languages. (1990) 218–227
2. Antoy, S.: Lazy evaluation in logic. In: Symp. on Progr. Language Impl. and Logic Progr (PLILP'91), Springer Verlag (1991) 371–382 LNCS 528.
3. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., (Eds.), G.P.: The Ciao System. Reference Manual (v1.10). The ciao system documentation series–TR, School of Computer Science, Technical University of Madrid (UPM) (2004) System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.
4. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., Puebla, G.: The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM) (1997) System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.
5. Cabeza, D., Hermenegildo, M.: A New Module System for Prolog. In: International Conference on Computational Logic, CL2000. Number 1861 in LNAI, Springer-Verlag (2000) 131–148
6. Deransart, P., Ed-Dbali, A., Cervoni, L.: Prolog: The Standard. Springer-Verlag (1996)
7. O'Keefe, R.: The Craft of Prolog. MIT Press (1990)
8. Puebla, G., Bueno, F., Hermenegildo, M.: An Assertion Language for Constraint Logic Programs. In Deransart, P., Hermenegildo, M., Maluszynski, J., eds.: Analysis and Visualization Tools for Constraint Programming. Number 1870 in LNCS. Springer-Verlag (2000) 23–61
9. Hermenegildo, M.V., Puebla, G., Bueno, F., López-García, P.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Computer Programming **58** (2005) 115–140
10. Cabeza, D., Hermenegildo, M.: Higher-order Logic Programming in Ciao. Technical Report CLIP7/99.0, Facultad de Informática, UPM (1999)
11. Cabeza, D.: An Extensible, Global Analysis Friendly Logic Programming System. PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain (2004)
12. Cabeza, D., Hermenegildo, M., Lipton, J.: Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction. In: Ninth Asian Computing Science Conference (ASIAN'04). Number 3321 in LNCS, Springer-Verlag (2004) 93–108
13. Quintus Computer Systems Inc. Mountain View CA 94041: Quintus Prolog User's Guide and Reference Manual—Version 6. (1986)
14. Carlsson, M., Widen, J.: Sicstus Prolog User's Manual, Po Box 1263, S-16313 Spanga, Sweden. (1994)
15. Cabeza, D., Hermenegildo, M.: The Ciao Modular Compiler and Its Generic Program Processing Library. In: ICLP'99 WS on Parallelism and Implementation of (C)LP Systems, N.M. State U. (1999) 147–164
16. Carlsson, M.: Freeze, Indexing, and Other Implementation Issues in the Wam. In: Fourth International Conference on Logic Programming, University of Melbourne, MIT Press (1987) 40–58

17. Naish, L.: Adding equations to NU-Prolog. In: Proceedings of The Third International Symposium on Programming Language Implementation and Logic Programming (PLILP'91). Number 528 in Lecture Notes in Computer Science, Passau, Germany, Springer-Verlag (1991) 15–26
18. Van Roy, P.: 1983-1993: The Wonder Years of Sequential Prolog Implementation. Journal of Logic Programming **19/20** (1994) 385–441
19. Henderson et al., F.: (The Mercury Language Reference Manual) URL: `http://www.cs.mu.oz.au/research/mercury/information/doc/reference_manual_toc.html`.
20. Morales, J., Carro, M., Hermenegildo, M.: Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In: Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages. Number 3507 in LNCS, Heidelberg, Germany, Springer-Verlag (2004) 86–103
21. Marriott, K., de la Banda, M.G., Hermenegildo, M.: Analyzing Logic Programs with Dynamic Scheduling. In: 20th. Annual ACM Conf. on Principles of Programming Languages, ACM (1994) 240–254
22. Carro, M., Hermenegildo, M.: A simple approach to distributed objects in prolog. In: Colloquium on Implementation of Constraint and LOgic Programming Systems (ICLP associated workshop), Copenhagen (2002)
23. Cabeza, D., Hermenegildo, M.: Distributed Concurrent Constraint Execution in the CIAO System. In: Proc. of the 1995 COMPULOG-NET Workshop on Parallelism and Implementation Technologies, Utrecht, NL, U. Utrecht / T.U. Madrid (1995) Available from http://www.clip.dia.fi.upm.es/.
24. Barbuti, R., Bellia, M., Levi, G., Martelli, M.: On the integration of logic programming and functional programming. In: International Symposium on Logic Programming, Atlantic City, NJ, IEEE Computer Society (1984) 160–168
25. Bellia, M., Levi, G.: The relation between logic and functional languages. Journal of Logic Programming **3** (1986) 217–236
26. Nadathur, G., Miller, D.: An overview of λprolog. In: Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle), MIT Press (1988) 810–827
27. Chen, W., Kifer, M., Warren, D.: HiLog: A foundation for higher order logic programming. Journal of Logic Programming **15** (1993) 187–230
28. Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of Mercury: an efficient purely declarative logic programming language. JLP **29** (1996)
29. Becket, R., de la Banda, M.G., Marriott, K., Somogyi, Z., Stuckey, P.J., Wallace, M.: Adding constraint solving to mercury. In: Eight International Symposium on Practical Aspects of Declarative Languages. Number 2819 in LNCS, Springer-Verlag (2006) 118–133
30. Haridi, S., Franzén, N.: The Oz Tutorial. DFKI. (2000) Available from http://www.mozart-oz.org.
31. Hanus et al, M.: Curry: An Integrated Functional Logic Language. (`http://www.informatik.uni-kiel.de/~mh/curry/report.html`)
32. Moreno Navarro, J., Rodríguez-Artalejo, M.: BABEL: A functional and logic programming language based on constructor discipline and narrowing. In: Conf. on Algebraic and Logic Programming (ALP). LNCS 343 (1989) 223–232