Functional Notation and Lazy Evaluation in Ciao

Amadeo Casas[1] Daniel Cabeza[2] Manuel Hermenegildo[1,2]

amadeo@cs.unm.edu ,
{dcabeza, herme }@fi.upm.es

[1]Depts. of Comp. Science and Electr. and Comp. Eng., Univ. of New Mexico, Albuquerque, NM, USA.

[2]School of Computer Science, T. U. Madrid (UPM), Madrid, Spain

CLIP Group

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.
- Functional Programming provides syntactic convenience, by having a syntactically designated output argument.
- Functional Programming also provides the ability to deal with infinite data structures by means of lazy evaluation.
- We present a design for an extensive functional layer for logic programs and its implementation in the *Ciao* system.

Why this is new

- Adding functional features to LP systems is clearly not new:
  - Work by Bella, Levi, et al.
  - Naish: equations in NU-Prolog.
- A good number of systems which integrate functions into some form of L.P.:
  Oz, Mercury, HAL, Lambda-Prolog, . . .
- Or perform a full integration of Functional and Logic Programming (e.g., Curry).
- Our proposal and its implementation has peculiarities which make it interesting:
  - The system supports ISO-Prolog.
  - The Language is extensible (and restrictable).
  - Functional features added at the source (Prolog) level.
  - Using Ciao *packages* (no compiler or machinery modification).
  - Functions retain the power of predicates (it's just notation).

Functional Notation in Ciao (I)

- Function applications:
  - Any term preceded by the ~/1 operator is a function application:

```
write(~arg(1, T)).
```

```
arg(1, T, A), write(A).
```

  - The next declaration avoids the need to use the ~/1 operator:

```
:- function arg/2.
```

```
write(arg(1, T)).
```

  - It is possible to use a predicate argument other than the last as the return argument:

```
:- fun_return functor(~, _,_).
```

```
~functor(~, f, 2).
```

  - The following declaration combines the previous two:

```
:- function functor(~, _, _).
```

```
:- fun_return functor(~, _, _).
:- function functor/2.
```

Functional Notation in Ciao (II)

- Predefined evaluable functors: several functors are evaluable by default:
  - All the functors understood by *is/2*. Can be disabled by a declaration:

    ```
    :- function arith(false).      % reverted by using true.
    ```

  - Functors used for disjunctive and conditional expressions:

    ```
    (Cond1 ? V1 | (Cond2 ? V2 | V3)).
    ```

- Functional definitions:

```
fact(0)  := 1.                              % Using body guards
fact(N)  := N * fact(--N)   :- N > 0.

fac(N)  := N = 0 ? 1                        % using conditional expressions
         | N > 0 ? N * fac(--N).
```

The translation of functional clauses defining recursive predicates maintains the tail recursion of the equivalent predicate.

Functional Notation in Ciao (III)

- Quoting functors: functors can be prevented from being evaluated:

```
pair(A,B)    := ^(A-B).
```

- Scoping: function applications evaluated 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` operator:

```
findall(X,    (d(Y),   ^^(X  = ~f(Y)+1)),    L).
```

- Laziness: 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:

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

Functional Notation in Ciao (IV)

- Definition of real functions: functions not forced to provide a single solution for their result. In order to declare a function as a real function:

```
:- funct name/N.
```

  which adds pruning operators and Ciao *assertions* to add restrictions as determinacy and modedness.

- Functional notation really useful to write regular types in a very compact way:

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

Example of Functional Notation in Ciao

```
:- function(arith(false)).
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.
```

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

Higher-Order

- Not topic of this paper, but combines well with these syntactic extensions.
- Adding functions to higher-order in Ciao:
  - Predicate abstraction ⇒ Function abstraction
    `{''(X,Y) :- p(X,Z), q(Z,Y)}` ⇒ `{''(X) := ~q(~p(X))}`
  - Predicate application ⇒ Function application
    `..., P(X,Y), ...` ⇒ `..., Y = ~P(X), ...`
- The integration is at the predicate level.

Implementation Details

- Functional features provided by two Ciao *packages*.
- Packages in Ciao are libraries which define extensions to the language.
- Packages are based in the redesigning of the traditional *term expansions* and operator definitions to make them more well-behaved and local to the module.
- Two packages: one for the bare function features without lazy evaluation, and an additional one to provide the lazy evaluation features.
- Basic functional features are translated using the well-known technique of adding a goal for each function application.

Lazy Functions Implementation

- Translation of a lazy function into a predicate is done in two steps:
  - First, the function is converted into a predicate by the bare functions package.
  - The predicate is transformed to suspend its execution until the value of the output variable is needed, by the use of the `freeze/2` control primitive.
- The translation will rename 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/1` .

Example of Lazy Functions

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

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

```
fiblist(X) :-
        freeze(X, 'fiblist_$$lazy$$'(X)).

'fiblist_$$lazy$$'([0, 1 | Rest]) :-
        fiblist(FibL),
        tail(FibL, T),
        zipWith(add, FibL, T, Rest).
```

Performance Measurements (I)

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

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

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

Performance Measurements (II)

	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

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

```
:- lazy function qsort/1.
qsort(X) := qsort_(X, []).

:- lazy function qsort_/2.
qsort_([], Acc) := Acc.
qsort_([], Acc) := Acc.
qsort_([X|T], Acc) := qsort_(S, [X|qsort_(G, Acc)])
                   :- (S, G) = partition(T, X).
```

```
:- lazy function partition/3.
partition([], _) := ([], []).
partition([X|T], Y) := (S, [X|G]) :-
                       Y < X,
                       !,
                       (S,G) = partition(T, Y).
partition([X|T], Y) := ([X|S], G) :-
                       !,
                       (S,G) = partition(T, Y).
```

Lazy Evaluation vs. Eager Evaluation

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

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

:- function takeWhile/2.
takeWhile(P, [H|T]) := P(H) ? [H | takeWhile(P, T)]
                            | [].
:- function test/0.
test := takeWhile( { ''(X) := X < 10000 }, squares).
```

```
:- module(module2, [squares/1], [functions, lazy, hiord]).

:- lazy function squares/0.
squares := map_lazy(take(1000000, nums_from(0)), { ''(X) := X * X }).

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

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

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

Conclusions

- We have presented a functional extension of Prolog, which includes the possibility of evaluating functions lazily.
- The proposed approach has been implemented in *Ciao* and is used now throughout the libraries and other system code as well as in a number of applications written by the users of the system.
- The performance of the package has been tested with several examples. As expected, evaluating functions lazily implies some time and memory overhead with respect to eager evaluation.
- The main advantage of lazy evaluation is to make it easy to work with infinite data structures in the manner that is familiar to functional programmers.