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

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

{amadeo, herme}@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

ntroduction

- LP offers features, such as nondeterminism, partially instantiated data structures, and constraints providing high expressive power.
- FP provides syntactic convenience (because of designated output argument).
- FP also provides lazy evaluation: ability to deal with infinite data structures and save execution steps.
- LP provides more powerful (lazy) evaluation mechanism (delay declarations) but, again, FP brings syntactic convenience.
- Discuss the combination with higher order.
- We present a syntactic functional layer (combining functions, laziness, and HO) as implemented in the *Ciao* language
(but useful in general for LP-based systems).

s any of this new?

- Adding functional features to LP systems clearly not new:
  - A good number of systems integrate functions into some form of LP:
    NU-Prolog, Lambda-Prolog, HiLog/XSB, Oz, Mercury, HAL, . . .
  - Or perform a “native” integration of FP and LP (e.g., Babel, Curry, ...).
- Ciao design is contemporary to these (˜97). Its peculiarities make it interesting:
  - Functions can retain the power of predicates (it is just syntax!).
  - Functions *inherit all other Ciao features (assertions, properties, records, constraints, ...)* + *(analysis, optimization, verification, ACC, ...)*.
  - The system can support ISO-Prolog, and
    functions, laziness, (and hiord) can be used as a extension of it (or not).
- Also the implementation is different (library-based):
  - Exploits the Ciao extension/restriction facilities: Ciao **packages** concept.
  - Makes it independent from, and (partially) compositional with other extensions.
  - No compiler or abstract machine modification (all done at source level).

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).`

  - Declarations can be used to avoid the need to use the `~/1` operator:

`:- fun_eval arg/2.`	`write(arg(1, T)).`

  - Also possible to use arguments other than last for "return":

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

  - The following declaration combines the previous two:

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

Functional Notation in Ciao (II)

- Several functors are *evaluable* by default:
  - Special forms for disjunctive and conditional expressions: `|/2` and `?/2`.
    - `A | B | C`
    - `Cond1 ? V1`
    - `Cond1 ? V1 | V2`

    Precedence implies that: `Cond1 ? V1 | Cond2 ? V2 | V3`
    is parsed as: `Cond1 ? V1 | (Cond2 ? V2 | V3)`
  - All the functors understood by *is/2*, if the following declaration is used:

    ```
    :- fun_eval arith(true).
    ```

    Using `false` it can be (selectively) disabled.
- *Functional definitions:*

```
opposite(red) := green.               ≡ opposite(red,green).
addlast(X,L) := ~append(L,[X]).       ≡ addlast(X,L,R) :- append(L,[X],R).
```

Functional Notation in Ciao (III)

- Can also have a body (serves as a guard or as *where*):

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

- The declaration `:- fun_eval defined(true).`
  allows dropping the ~ within a function's definition:

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

  And, using conditional expressions:

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

- The translation:
  - Produces *steadfast* predicates (bindings after cuts).
  - Maintains tail recursion.

eriv and its Translation

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

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

Examples – Sugar for Append

- Some syntactic sugar for append:

```
:- fun_eval append/2.

mystring(X) := append("Hello",append(X,"world!")).
```

- Some more:

```
:- op(200,xfy,[++]).
:- fun_eval ++ /2.

A ++ B := ~append(A,B).

mystring(X) := "Hello" ++ X ++ "world!".
```

Examples – Array Access Syntax

- Assume multi-dimensional arrays such as (for 2x2): `A = a(a(_,_),a(_,_)).`
- We can now define the array access function with some syntactic sugar:

```
:- pred @(Array,Index,Elem) :: array * list(int) * int
   # "@var{Elem} is the @var{Index}-th element of @var{Array}.".

:- op(45, xfx, [@]).
:- fun_eval '@'/2 .

@(V,[I])     := ~arg(I,V).
@(V,[I|Js]) := @(~arg(I,V),Js).
```

- And use it: `?- M = ~array([2,2]), M@[2,1] = 3, display(M).`

  (for this the `op` and `function` declarations must be loaded in the top level also!)
- E.g., in a vector addition:

```
for(I,1,N) do V3@[I] = V1@[I] + V2@[I]
```

Functional Notation in Ciao (IV)

- *Quoting.* Evaluable functors can be prevented from being evaluated:

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

- *Scoping.* When innermost function application is not desired (e.g., for certain meta-predicates) a different scope can be determined with the `(^^)/1` operator:

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

translates to: `findall(X, (d(Y),f(Y,Z),T is Z+1,X=T), L).`
as opposed to: `f(Y,Z), T is Z+1, findall(X, (d(Y),X=T), L).`

- *Laziness.* Execution is suspended until the return value is needed:

```
:- lazy fun_eval nums_from/1.

nums_from(X) := [X | nums_from(X+1)].
```

(Can be done easily with `when`, `block`, `freeze`, etc. but proposed notation more compact for this special case. Also, `:- lazy pred_name/M`.)

Functional Notation in Ciao (V)

- Functional notation really useful, e.g., to write regular types in a compact way:

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

Which translate to:

```
color(red). color(blue). color(green).

list([]).
list([_|T]) :- list(T).

list_of(_, []).
list_of(T, [X|Xs]) :- T(X), list_of(T, Xs).
```

And can then of course be used in Ciao assertions:

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

Functional Notation in Ciao (VI)

- *Definition of "real" functions:* `:- funct name/N.`
  adds pruning operators and Ciao *assertions* to add functional restrictions: determinacy, modedness, etc.
- E.g.:

```
:- funct nrev/1.
nrev( [] )     := [].
nrev( [H|T] ) := ~conc( nrev(T),[H] ).
```

Is translated to (simplified):

```
:- pred nrev(A,B,C)
      : (ground(A), ground(B), var(C))
     => (ground(A), ground(B), ground(C))
      + is_det,mut_exclusive,covered,no_fail.

nrev( [],    Y ) :- !, Y = [].
nrev( [H|L],R ) :- !, nrev(L,RL), conc(RL,[H],R).
```

Combining with Constraints, etc.

- Combining with constraints, some syntactic sugar, assertions:

```
:- module(_,_,[assertions,fsyntax,clpq]).

:- fun_eval .=. /1.
:- op(700,fx,[.=.]).
:- fun_eval fact/1.

:- pred fact(+int,-int) + is_det.
:- pred fact(-int,-int) + non_det.

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

- Sample query:

```
?- 24 = ~fact(X).
X = 4
```

Combining Higher-Order with Functional Notation

- HO not topic of the paper, but combines well with these syntactic extensions.
- Combining function application (~) and HO:
  - Predicate application ⇒ Function application
    `..., P(X,Y), ...` ⇒ `..., Y = ~P(X), ...`
- Function abstraction:
  - Predicate abstraction ⇒ Function abstraction
    `{''(X,Y) :- p(X,Z), q(Z,Y)}` ⇒ `{''(X) := ~q(~p(X))}`
- The integration is at the predicate level.

Combining Higher-Order with Functional Notation

- Some common examples:

```
:- meta_predicate map(_,pred(2),_).
map([], _) := [].
map([X|Xs], P) := [~P(X)|~map(Xs,P)].

:- meta_predicate foldl(_,_,pred(3),_).
foldl([], Seed, _Op) := Seed.
foldl([X|Xs], Seed, Op) := ~Op(X,~foldl(Xs,Seed,Op)).
```

- More uses of `map/3` (using functional notation):

```
?- L = ~map([1,2,3], ( _(X,Y):- Y = f(X) ) ).
L = [f(1),f(2),f(3)]

?- [f(1),f(2),f(3)] = ~map(L, ( _(X,f(X)) :- true ) ).
L = [1,2,3]
```

Combining with Ciao's Abstract Interp.-based Assertion Checking

Combining with Ciao's Certificates (Abstraction Carrying Code)

mplementation Details

- All syntactic effects are local to the modules that use these packages (as usual in Ciao):
  - Packages in Ciao are libraries which define extensions to the language.
  - Packages are based on the redesign of the traditional *term expansions* and operator definitions to make them more well-behaved and module-local.
- Functional features provided by Ciao *packages*:
  - One provides the bare function features without lazy evaluation,
  - An additional one provides the lazy evaluation features.
- Functional features are implemented by translation using the well-known technique of adding a goal for each function application.

mplementation Details

- Translation of a lazy function into a predicate is done in two steps:
  - First, the function is converted into a predicate by the standard functions package.
  - The predicate is then transformed to *suspend its execution until the value of the output variable is needed*, by means of the `freeze/2` or `block` family of control primitives.
- (For `freeze/2` 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 and Translation (stylized)

```
:- lazy fun_eval 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).

```
:- 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)].
```

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 fun_eval qsort/1.
qsort(X) := qsort_(X, []).

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

```
:- lazy fun_eval 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 (I)

```
- 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( { ''(X) := X < 10000 }, squares).
```

Lazy Evaluation vs. Eager Evaluation (II)

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

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

- 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)].
```

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.
- Current work w/Gopalan Nadathur's team on HO-unification – λ-Prolog.