Extensions: A Technique for Structuring Functional-Logic Programs

Rafael Caballero and Francisco J. López-Fraguas*

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid,
{rafa,fraguas}@sip.ucm.es

Abstract. Monads are a technique widely used in functional programming languages to address many different problems. This paper presents *extensions*, a functional-logic programming technique that constitutes an alternative to monads in several situations. Extensions permit the definition of easily reusable functions in the same way as monads, but are based on simpler concepts taken from logic programming, and hence they lead to more appealing and natural definitions of types and functions. Moreover, extensions are compatible with interesting features typical of logic programming, like multiple modes of use, while monads are not.

1 Introduction

Functional-Logic programming, FLP in short, aims to integrate of functional and logic programming, allowing the use of techniques from both paradigms into the same declarative framework (see [5] for a survey). Moreover, the combination of ideas of the two worlds gives rise to new features specific to FLP. This work should be seen as a contribution in this direction, for it presents a new technique, the *extensions*, that can be used as an alternative to the functional technique of *monads* when programming in a functional-logic language.

The concept of monad comes from category theory, and it has been widely used in functional programming to structure functions, pointing out the essence of the algorithms represented while concealing the data flow and the associated computations [13,14,15].

In several FLP frameworks such as *Escher* [9], *Curry* [6] or our working language, $\mathcal{TOY}$ [3,12], monads can be used directly, yielding the same benefits as in the case of functional programming. However, FLP has a wider range of programming mechanisms, including logical variables, and it should be questioned whether it is possible to define a specific FLP technique to address the same kind of problems from a different point of view. In the rest of the paper we describe such an alternative, the FLP *extensions*. Although lacking the theoretical background and wide range of applications of monads, extensions present some specific advantages, such as:

* Work was partially supported by the Spanish CICYT (project TIC98-0445-C03-02/97 "TREND") and the ESPRIT Working Group 22457 (CCL-II).

D. Bjørner, M. Broy, A. Zamulin (Eds.): PSI'99, LNCS 1755, pp. 297–310, 2000.

• Extensions can replace monads in several different situations, allowing the same expressiveness but using much simpler concepts.

• Multiple modes of use are allowed by extensions, which is not so easy to achieve when defining monads in an FLP context.

• In the case of adding new features to functions, monads enforce the evaluation of both the old and the new values simultaneously. Conversely, extensions can use the new feature only where it is required, thus avoiding unnecessary computations.

2 The FLP Framework: A Succinct Description of $\mathcal{TOY}$

All the programs in the next sections are written in the purely declarative functional-logic language $\mathcal{TOY}$, which is a concrete realization of *CRWL*, a theoretical framework for declarative programming (see [4]). We present here only the subset of the language relevant to this work. A more complete description and a number of representative examples can be found in [3].

A $\mathcal{TOY}$ program consists of *datatype*, *type alias*, *infix operator* definitions, and rules for defining *functions*. Syntax is mostly borrowed from Haskell [7], with the remarkable exception that variables begin with upper-case letters whereas constructor and function symbols use lower-case.

```
infixr 20 :/:
data expr = val real | expr :/: expr

eval:: expr → real
eval (val A) = A
eval (A :/: B) = (eval A)/(eval B)
```

Fig. 1. Monadic variations of the basic evaluator

Our first example of a program written in $\mathcal{TOY}$ may be seen in figure 1. This program is the $\mathcal{TOY}$ version of the evaluator for simple expressions presented by P. Wadler in his article [15], and will be our starting point in order to compare monads and extensions. The evaluator itself is represented by function `eval`, which takes an expression E as the only input parameter, and returns the real number resulting from evaluating E. An expression can be either a real number r, represented as `val r` or a quotient between expressions e_1 and e_2, represented as $e_1 :/: e_2$.

In general, each function `f` in $\mathcal{TOY}$ is defined by a set of conditional rules of the form

$$\texttt{f}\ t_1\ \ldots t_n\ \texttt{=}\ \texttt{e} \Longleftarrow e_1\ \texttt{==}\ e'_1,\ \ldots,\ e_k\ \texttt{==}\ e'_k$$

where $(t_1 \ldots t_n)$ forms a tuple of linear (i.e. with no repeated variable) constructor terms, and e, e_i, e'_i are *expressions*. No other conditions (except well-typedness) are imposed to function definitions. Rules have a conditional reading: `f` $t_1 \ldots t_n$ can be reduced to `e` if all the conditions $e_1 == e'_1, \ldots, e_k == e'_k$ are satisfied. The condition part is omitted if $k = 0$ (as in our previous example `eval`). The symbol `==` stands for *strict equality*, which is the suitable notion for equality when non-strict functions are considered. With this notion a condition `e == e'` can be read as: `e` and `e'` can be reduced to the same constructor term.

$\mathcal{TOY}$ can introduce non-deterministic computations by different means, but we only need one of them for this discussion, namely the occurrence of *extra* variables in the right side of the rules like in

```
z_list = [0|L]
```

Although in this case `z_list` reduces only to `[0|L]`, the free variable `L` can be later on instantiated to any list. Therefore, any list of integers is a possible value of `z_list`.

Computing in $\mathcal{TOY}$ means solving *goals*, which take the form

$$e_1\ \texttt{==}\ e'_1,\ \ldots,\ e_k\ \texttt{==}\ e'_k$$

giving as its result a substitution for the variables in the goal making it true. Evaluation of expressions (required for solving the conditions) is done by a variant of lazy narrowing based on a sophisticated strategy, called *demand driven strategy* which uses the so-called *definitional trees* [2] to guide unification with patterns in left-hand sides of rules (see [8]). For instance, using the evaluator defined above we may try the goal:

```
eval (val 16 :/: val 4 :/: val 1 :/: val 8) == R
```

which yields `R == 0.5`.

As an aside, we remark that the current version of our language does not incorporate *lambda* abstractions or *let* constructions. However, these syntactic facilities are usual in the functional programming literature, and we have included them in some of our examples in order to fairly represent the monadic approach. For testing the examples in the actual implementation, we have simply needed to 'lift' such constructions using well-known techniques [10].

3 Funcional-Logic Monads

In this section we present two variations of the basic evaluator, following the lines of Wadler's paper [15]. We also recall briefly some of the basic concepts concerned with monads, which will be useful when comparing monads and extensions. However, we will not delay very much at this point, assuming that the definition

and usefulness of monads are well-known, and referring to the cited article for a deeper discussion of these issues.

To convert a function *f::A* → *B* to monadic form we change its type to *f::A* → *m B*, meaning that function *f* accepts a parameter of type *A* and returns a value of type *B*, with an associated computation represented by *m*. The structure of the function will be based on the functions *unit:: A* → *m A* (also known as *result*) and *(*)::m A* → *(A* → *m B)* → *m B* (usually called *bind*) and indicates how the value *B* is constructed, avoiding any explicit reference to the computation *m*. Only *unit* and *** (and perhaps some auxiliary functions) will 'know' what *m* is actually, and how to deal with it. If we want to add some extra capabilities to the original code of *f* later, we only need to look for an appropriate data constructor *m'* that captures the essence of the modification. Then we redefine the type of the function to *f::A* → *m' B*, define the new versions of *** and *unit* and, perhaps, make a few local changes in the code of the function itself, but always keeping the same basic structure.

Figure 2 shows two 'classical' variations of the original evaluator.

```
type state = int

type m A = state → (A,state)

unit:: A → m A
unit A X = (A,X)

infixr 30 *
(*)::m A → (A → m B) → m B
(*) M K S = let (A,S2) = M S
            in K A S2

tick :: m ()
tick X = ((),X+1)

eval:: expr → m real
eval (val A) = unit A

eval (A :/: B) =
    eval A * λR1. eval B * λR2.
    tick * λ(). unit (R1/R2)
```

```
type output = string

type m A = (output,A)

unit:: A → m A
unit A = ("",A)

infixr 30 *
(*)::m A → (A → m B) → m B
(X,A) * K = let (Y,B) = K A
            in (X++Y,B)

out::output → m ()
out X = (X,())

eval:: expr → m real
eval (val A) = out(line(val A) A)
             * λ().unit A
eval (A :/: B) =
    eval A * λR1. eval B * λR2.
    out (line (A :/: B) (R1/R2))
     * λ().unit (R1/R2)
```

Fig. 2. Monadic variations of the basic evaluator

The first variation, is based on the very useful *state monad*, taken from [15] and adapted to $\mathcal{TOY}$ syntax, which is used to count the total number of divisions performed while evaluating the expression. The second variation produces a trace of the evaluation. This last variation uses a function `line` which produces a step of the trace and may be defined as:

```
line T R = "eval(" ++  showterm T ++  ")⇐= " ++
           number_to_string R ++  "\n"
```

assuming suitable definitions for `showterm` and `number_to_string`. The infix operator `++` is the standard function for concatenation of lists. It can be seen that the basic structure of `eval` is kept almost unaffected. If we had modified the initial code directly, this would have been more difficult to achieve.

4 Functional-Logic Extensions

In the previous section we have sketched how the monadic approach can be adopted in $\mathcal{TOY}$. Now it is time to present the alternative provided by our FLP extensions.

4.1 An Informal Introduction to Extensions

The idea of FLP extensions is quite simple, and constitutes itself a good example of mixing the resources of logic and functional programming:

Suppose we would like to add a new capability of type C to a given function $f::A \to B$. Then, all we need to do is to *extend* the type of the function to $f::A \to B \to C$, meaning that the old returned value is now an *output parameter*, while the new value is introduced as the *result* of the function.

Consider the initial basic evaluator and suppose we want to enrich the capabilities of the function

```
eval::expr → real
```

by associating a new value of type `C` to the currently returned real number. Then, we extend the function with the new feature, changing its type to

```
eval::expr → real → C
```

Of course the definition of `eval` also needs to be modified, acknowledging that the result of the evaluation is no longer the result of the function, but an output parameter.

In order to hide the way the values of type `C` are composed we define a combinator

```
(*)::C → C → C
```

Hence the second rule for `eval` will have the shape

```
eval (A :/: B) R = eval A R1 * eval B R2 ...
```

with the values `R, R1, R2` standing for the result of the evaluation of `A :/: B`, `A` and `B` respectively. The problem of constructing the new result of the function seems to be solved: `eval A R1` and `eval B R1` are actual values of type `C` related to the 'old' values `R1` and `R2`, and therefore can be combined by using `*`. If later we change `C` by `C'` we only need to change the definition of `*` but not the basic structure of `eval`.

However, we still need to associate the value `R1/R2` with the result of the evaluation `R`. This will be performed by function `unit`, which must 'identify' `R` and `R1/R2`. In order to generalize the definition to other situations, both values `R` and `R1/R2` will be input parameters of `unit`. The logical way of adding `unit` to the definition of `eval` is simply by using `*`:

```
eval (A :/: B) R = eval A R1 * eval B R2 * unit (R1/R2) R
```

This means that `unit` should return a value of type C and, since we said above that the result of the functions was already properly constructed by `eval A R1 * eval B R2` , the value of `unit` must be a truly *unit value* with respect to the operation `*`. Therefore given a *unit element* `e` of type `C`, we can define `unit` as

```
unit::real → real → C
unit A A = e
```

where the repeated variable is just a 'syntactic sugar' of

```
unit A B = e ⇐= A==B
```

That is, `unit` returns `e` if the strict equality `A==B` succeeds. This produces the desired identification between the result `R` and `R1/R2`.

4.2 Extensions of the Basic Evaluator

The 'extension counterpart' of the monadic variations presented in the previous section may be seen in figure 3. The type `C` of our discussion is represented respectively by the types `trans` and `output`, while the unit elements are `id` and `" "`, where the standard function `id` is defined as usual:

```
id X = X
```

Further details about these examples may be found in section 5.

4.3 Definition of Extension

A *FLP extension* is a tuple *(b, unit, *)* where b is an specific type, *unit* is a function of type $A \to A \to b$ and definition *unit A A = e*, $e \in b$, and where $*$ is a function of type $b \to b \to b$ such as (e, *) is a monoid.

```
type state = int
type trans = state → state

unit:: A → A → trans
unit A A = id

infixr 30 *
(*)::trans → trans → trans
(*) M K S = K S2 ⇐= M S == S2

tick :: trans
tick = (1+)

eval:: expr → real → trans
eval (val A) R = unit A R

eval (A :/: B) R =
    eval A R1 * eval B R2 *
    unit (R1/R2) R * tick
```

```
type output = string

unit:: A → A → output
unit A A = ""

infixr 30 *
(*)::output → output → output
M * K = M ++ K

out::output
out = id

eval:: expr → real → output
eval (val A) R = unit A R *
    out (line (val A) A)
eval (A :/: B) R =
    eval A R1 * eval B R2 *
    unit (R1 / R2) R *
    out (line (A :/: B) R)
```

Fig. 3. Extensions of the basic evaluator

Now it can be proved easily that the variations of figure 3 are actually extensions. For example, the pair ("",++) used in the output extension is known to satisfy the properties of monoids. The proof for the other case is quite straightforward. Although this definitions lacks the theoretic background of the definition of monad, the structure of monoid is enough to prove some simple assertions about the functions defined using `*` and `unit` in the same line as that of [15].

5 A Comparative Survey

So far we have presented two 'classical' variations of the basic evaluator, using both extensions and monads. Now we can present a first comparative study of the two techniques. In the following points we show some of the advantages of using extensions that can be checked directly in the examples.

• The definitions of types for extensions are simpler than in the case of monads. Indeed, we do not need to worry about how to combine the old and the new value, while monads need to define a suitable type constructor `m`. For example, in order to add the output trace to the basic evaluator, we have defined the type

```
type output = string
```

while the monadic version needs also define

```
m A = (A, output)
```

• As a consequence of the previous point, functions `unit` and `*` admit simpler definitions. For instance

```
(*):: output → output → output
M * K = M ++K
```

indicates that the result of combining two outputs is the concatenation of both of them. Observe, in particular, the symmetrical aspect of the type of `(*)`. This definition seems more readable than the monadic variation:

```
(*):: m A → (A → m B) → m B
(X,A) * K =   let (Y,B) = K A in (X++Y,B)
```

• The symmetrical definition of `*` also entails some practical consequences, as it allows the programmer to change the order of the combined values. Thus we do not need to end the sequence with a `unit` expression, as in the case of monads. For instance, take the second rule for `eval` in the output monad:

```
eval (A :/: B) = eval A * λR1. eval B *
                 λR2. out (line (A :/: B) (R1/R2)) *
                 λ(). unit (R1/R2)
```

It would better to change the order of `unit` and `out`, writing instead

```
eval (A :/: B) = eval A * λR1. eval B * λR2. unit (R1/R2)
                 *λR. out (line (A :/: B) R)
```

avoiding the unnecessary repeated calculation of $R1/R2$ and separating the *side effect* from the main computation, but this is not possible without changing the definition of `out`. However the definition of `*` for extensions allows us to write

```
eval (A :/: B) R = eval A R1 * eval B R2 * unit (R1/R2) R
                   * out (line (A :/: B) R)
```

where $R1/R2$ is computed only once.

• The separation between the old and the new values also benefits the definitions of auxiliary functions such as `tick` or `out`. For example, as `tick` must increase the state we need only write

```
tick = (1+)
```

instead of the monadic definition

```
tick X = ((),X+1)
```

These straightforward definitions also avoid the useless dummy variables and values () that appear in the monadic definitions.

Of course, extensions have some disadvantages like any other programming technique. We can point out the following drawbacks:

• Monads are a more abstract technique. They are based upon deep theoretical results and can be applied to a number of different areas beyond programming, such as type inference or semantics, while extensions are hitherto just a specific methodology of FLP.

• Some monads cannot be thought of in terms of extensions, because they are not meant to add new values to a previously given function. For instance, *lists* may be seen as a monad (see [15]), while they cannot be defined in terms of extensions.

Therefore, extensions cannot be applied to the same situations as monads. And, can monads substitute extensions? In Section 6 we will present some applications of extensions that cannot be accomplished by monads, hence showing that neither of both techniques may be subsumed into the other one.

6 Other Features of Extensions

Extensions and monads look quite similar, but actually they can be used to solve different problems. We have pointed out in Section 5 some limitations of extensions. Now we are going to show how extensions can be used in two situations where monads cannot be readily applied.

6.1 Avoiding Unnecessary Computations

Monads (as well as extensions) allow one to increase the capabilities of functions while keeping their basic structures unaffected. Of course, these extra features also entail extra computation time. The efficiency of the two techniques is quite similar (both in time and space) when the extra features are computed. However the situation changes remarkably in the points of the program where still only the old value of the function is required. This may be specially extreme when dealing with the state monad (or extension).

Imagine for example that we need a variation of the evaluator of expressions that not only computes the resulting real number but also maintains an ordered list with the numbers that appear in the expression. Such variation may be seen in figure 4 using monads and extensions
with the function `insert` defined as usual. Functions `*`, `unit` and types `m A` and `trans` have not been included for they are those of the state variations we showed before (figures 2 and 3). Here function `tick` is used to insert an element in the ordered list, while the initial state is the empty list. For example, using extensions we may try

```
eval (val 8 :/: val 4 :/: val 2) R [] == L
```

```
type state = [real]

tick :: real → m ()
tick A S = ( (), insert A S)

eval:: expr → m real
eval (val A) = tick A * λ().unit A
eval (A :/: B) = eval A *
                 λR1.eval B *
                 λR2. unit (R1/R2)
```

```
type state = [real]

tick :: real → trans
tick A = insert A

eval:: expr → real → trans
eval (val A) R = tick A * unit A R
eval (A :/: B) R = eval A R1 *
                   eval B R2 *
                   unit (R1/R2) R
```

Fig. 4. Evaluator yielding an ordered list, using monads (left) and extensions (right)

which returns the values

$$\texttt{R == 4}$$

$$\texttt{L == } [2, 4, 8]$$

However, it is possible that we might still need to evaluate expressions just to get the result, dismissing the list. In this case, the insertion of all the elements in the list is an unnecessary overweight that should be avoided. Using extensions this can be done by simply not providing the initial state [] to the goal. Then the result of evaluating the expression is computed as usual, but the state is returned as a 'chain of actions' not evaluated yet, as is witnessed by the goal

```
eval (val 8 :/: val 4 :/: val 2) R == L
```

that returns

```
R == 4

L == (insert 8 * id) * ((insert 4 *id) * (insert 2 *id) *id) *id
```

Thus the actual insertion in the list is not carried out, and we can define a function `eval'` as

```
eval' Expr = R ⇐= eval Expr R == _
```

Note that this cannot be done by using monads, because the two values, the numeric result and the list are actually parts of a single value. Effectively, if we do not provide the initial state to the monadic variation, a goal like

```
eval (val 8 :/: val 4) == L
```

yields an expression of the shape

```
L == (tick 8 * λ().unit 8) * λR1.(tick 4 * λ().unit 4) *
     λR2.unit(R1/R2)
```

because functions `tick`, `unit` and `*` cannot be reduced until a initial state is provided. Thus we can either compute both the result and the ordered list, or neither.

The use of the function `eval'` whenever the list is not required can speed up the program considerably. Checked with a expression of 300 numbers, we have found out that the differences of time between `eval'` and `eval` using extensions, can vary from $0'38s$ to $5'10s$. And, despite the big chain of `insert` and `id` functions that `eval'` must construct, the space required is also less than in the case of actually performing the insertions with `eval`.

6.2 A Parser for Free

Consider the boolean expressions defined as

```
infixr 20 : /\ :
infixr 15 : \/ :

data expr = val bool | expr : /\ : expr | expr : \/ : expr
```

Suppose that we decide to define a evaluator `evalb` for this expressions, returning not only the result of the evaluation, but also a suitable representation of the expression. The code for such function may be seen in the figure 5, using monads (left side) and using extensions (right side), and is a simple application of the `output` feature presented before.

Functions `or` and `and` are defined as usual in functional programming, while function `conv` may be easily defined as

```
conv true  = "T"
conv false = "F"
```

For example, using the monadic variation, we may try

```
evalb (val true : /\ : (val false : \/ : val true)) == R
```

which returns

```
R == ( "(T and (F or T))" , true )
```

Suppose now that, after evaluating a few expressions using the new variation, we decide that representations like `"(T and (F or T))"` are definitely nicer and more readable than

```
evalb (val true : /\ : (val false : \/ : val true))
```

```
evalb:: expr → m bool

evalb (val A) = out (conv A) *
     λ _. unit A

evalb (A :\/: B) =
     out "(" * λ(). evalb A *
     λ R1. out " or " * λ().
     evalb B * λR2. out ")" *
     λ(). unit (R1 `or` R2)

evalb (A :/\: B) =
     out "(" * λ(). evalb A *
     λ R1. out " and " * λ().
     evalb B * λR2. out ")" *
     λ(). unit (R1 `and` R2)
```

```
evalb:: expr → bool → output

evalb (val A) R = out (conv A) *
     unit A R

evalb (A :\/: B) R =
     out "(" * evalb A R1 *
     out " or " *
     evalb B R2 *
     unit (R1 `or` R2) R * out ")"

evalb (A :/\: B) R =
     out "(" * evalb A R1 *
     out " and " *
     evalb B R2 *
     unit (R1 `and` R2) R * out ")"
```

Fig. 5. Boolean evaluator with output, using monads and extensions

and that we would like to define a version of `evalb` accepting strings representing expressions as input parameter. Does it mean that now we need to define a parser for boolean expressions? The answer is *no, if we use extensions*. Indeed, the extension of the boolean evaluator showed in the figure 5 can be used as a parser without making any changes, as witnessed by the goal

```
evalb Expr R == "(F and (F or T))"
```

which succeeds with

```
Expr == val false :/\: (val false :\/: val true)
R == false
```

This nice outcome of extensions is an example of the *generate & test* techniques, very usual in logic programming. Therefore, ours is actually a recursive top-down parser of the grammar rules expressed in `evalb` by means of `output` (for terminals) and recursive calls of `evalb` (for non-terminals).

But, why is it not possible to use the monadic variation in this case? It is due to the combination of the string representation and the output value, which is a free variable. For example, the goal

```
evalb Expr == ("(F and T)",R)
```

loops. We must recall that strict equality does a 'careful matching' as we showed before. In the example, this means generating the outer constructor of both `"(F`

`and T)"` and `R` by means of `evalb Expr`. But getting an outer constructor for `R` entails generating a whole expression, and by using the second rule of `evalb`, infinite expressions may be generated. These expressions, all of which have an `or` in their representations, when finally compared with `(F and T)`, fail.

7 Conclusions

We have shown throughout this paper that extensions are a suitable mechanism to solve a number of problems when working in a functional-logic language. Although lacking the deep theoretical background of monads, extensions can be used as an alternative to define easily reusable code. The concepts used are simple, and were already known in each declarative paradigm, such as the use of arguments in logic programming to return output values, or the definition of higher order combinators (e.g. `*`) in order to connect different computations in sequence. The novelty of our approach is that it combines techniques of both main declarative streams, yielding a new mechanism that allows us to address problems, as the addition of new features to functions, in a simple and appealing way. Specifically, extensions avoid the necessity of lambda abstractions, provide a more symmetric definition of the combinator `*` – from the point of view of types – and lead to nicer and more natural definitions of types and auxiliary functions.

In spite of all the resemblances, extensions and monads are different techniques, each one with its own particularities and limitations. An advantage of extensions is that they provide functions with the possibility of multiple modes of use, therefore defining functions that can be reused in a wider sense than in the case of monads. Another advantage is that the state extension allows one to dismiss the stateful computations whenever they are not interesting, hence saving both time and space.

References

1. S. Antoy , R. Echahed, M. Hanus. *A Needed Narrowing Strategy.* 21st ACM Symp. on Principles of Programming Languages, 268–279, Portland 1994.
2. S. Antoy. *Definitional Trees*, In Proc. ALP'92, Springer LNCS 632, 1992, 143–157.
3. R. Caballero-Roldán, F.J. López-Fraguas and J. Sánchez-Hernández. *User's Manual For $\mathcal{TOY}$* . Technical Report D.I.A. 57/97, Univ. Complutense de Madrid 1997. The system is available at `http://mozart.sip.ucm.es/incoming/toy.html`
4. J.C. González-Moreno, T. Hortalá-González, F.J. López-Fraguas, M. Rodríguez-Artalejo. *An Approach to Declarative Programming Based on a Rewriting Logic.* Journal of Logic Programming, Vol 40(1), July 1999, pp 47–87.
5. M. Hanus. *The Integration of Functions into Logic Programming: A Survey.* J. of Logic Programming 19-20. Special issue "*Ten Years of Logic Programming*", 583–628, 1994.
6. M. Hanus (ed.). *Curry, an Integrated Functional Logic Language*, Draft, February 1998. Available at `http://www-ir.informatik.rwth-aachen.de/ hanus/curry/report.htlm`

7. *Report on the Programming Language Haskell: a Non-strict, Purely Functional Language*. Version 1.4, Peterson J. and Hammond K. (eds.), January 1997.
8. R. Loogen, F.J. López-Fraguas,M. Rodríguez-Artalejo. *A Demand Driven Computation Strategy for Lazy Narrowing*. Procs. of PLILP'93, Springer LNCS 714, 184–200, 1993.
9. Lloyd, J.W. *Declarative Programming in Escher*. Technical Report CSTR-95-013, Departament of Computer Science, University of Bristol, June 1995.
10. S.L. Peyton-Jones. *The implementation of functional languages*, Prentice Hall, 1987.
11. S.L. Peyton-Jones, P. Wadler. *Imperative functional programming*, 20 Annual Symposium on Principles of Programming Languages, Charleston, South Carolina, 1993.
12. F.J. López-Fraguas, J. Sánchez-Hérnandez. *TOY: A Multiparadigm Declarative System*. Proc. RTA'99, Springer LNCS 1631, 244–247, 1999.
13. P. Wadler. *Comprehending Monads*, Proc. ACM Conf. on Lisp and Functional Programming, 1990.
14. P. Wadler. *The essence of functional programming*, Proc. ACM conference on the Principles of Programming Languages, pages 1-14, 1992.
15. P. Wadler. *Monads for functional programming*. In J. Jeuring and E. Meijer editors, Lecture Notes on Advanced Functional Programming Techniques, Springer LNCS 925. 1995