A functional-logic alternative to monads

Rafael Caballero-Roldán, Francisco J. López-Fraguas [*][†]

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. This property further contributes to the employment of the same code to solve different problems.

1 Introduction

Functional-Logic programming, FLP in short, aims to the integration of functional and logic programming, allowing the use of techniques from both paradigms into the same declarative framework. Moreover, the combination of ideas of the two worlds gives rise to new features specific of 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 [Wad90, Wad92, Wad95]. This eases the problem of extending the capabilities of the functions, hence allowing the definition of reusable components. The skill of monads in hiding computations has been employed in a number of different areas, from the safe introduction of *imperative* characteristics (e.g. in-place updates) to the encapsulation of interactions with the real world, as performed by input/output operations [PW87]. Furthermore, the concept of monad has been also employed out of the scope of functional programming, like in the case of the logic programming language λProlog [BT95].

In several FLP frameworks such as *Escher* [Llo95], *Curry* [Han98] or our working language, $\mathcal{TOY}$ [CLS97], 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

[*]Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid.
e-mail: `rafacr@eucmos.sim.ucm.es`, `fraguas@eucmax.sim.ucm.es`

[†]Work partially supported by the Spanish CICYT (project TIC 95-0433-C03-01 "CPD") and the ESPRIT Working Group 22457 (CCL-II).

background and wide range of applications of monads, extensions present some specific advantages, such as:

• 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.

We begin in the following section by introducing our working language, $\mathcal{TOY}$. Only the features of the language that are needed for this discussion are described, setting a quite general FLP framework. Section 3 presents the examples of monadic variations that provide a suitable starting point for a comparison with extensions. The same examples are again discussed in section 4, but now adopting the technique of FLP extensions, which are defined formally. A first comparison between the two techniques is carried out in section 5, while section 6 discusses some specific benefits that we can expect from using extensions.

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 $\mathcal{TOY}$. We present here the subset of the language relevant to this work. A more complete description and a number of representative examples can be found in [CLS97].

A $\mathcal{TOY}$ program consists of *datatype*, *type alias* and *infix operator* definitions, and rules for defining *functions*. Syntax is mostly borrowed from Haskell [HAS97] (with the remarkable exception that variables begin with upper-case letters whereas constructor symbols use lower-case, as function symbols do). In particular, functions are *curried* and the usual conventions about associativity of application hold.

Datatype definitions like `data nat = zero | suc nat`, define new (possibly polymorphic) *constructed types* and determine a set of *data constructors* for each type. The set of all data constructor symbols will be noted as CS (CS^n for all constructors of arity n).

Types $\tau, \tau', \ldots$ can be constructed types, tuples $(\tau_1, \ldots, \tau_n)$, or functional types of the form $\tau \rightarrow \tau'$. As usual, $\rightarrow$ associates to the right. $\mathcal{TOY}$ provides predefined types such as `[A]` (the type of polymorphic lists, for which Prolog notation is used), `bool` (with constants `true` and `false`), `int` for integer numbers, or `char` (with constants `'a'`,`'b'`, ...). *Type alias* definitions like `type state = int` are also allowed. Type alias are simply abbreviations, but they are useful for writing more readable, self-documenting programs. Strings (for which we have the definition `type string = [char]`) can also be written with double quotes. For instance, `"sugar"` is the same as `['s','u','g','a','r']`.

The purpose of a $\mathcal{TOY}$ program is to define a set FS of functions. Each $f \in FS$ comes with a given *program arity* which expresses the number of arguments that must be given to f in order to make it reducible. We use FS^n for the set of function symbols with program arity n. Each $f \in FS^n$ has an associated principal type of the form $\tau_1 \rightarrow \ldots \rightarrow \tau_m \rightarrow \tau$ (where τ does not contain $\rightarrow$). Number m is called the *type arity* of f and well-typedness implies that $m \geq n$. As usual in functional programming, types are inferred and, optionally, can be declared in the program.

With the symbols in CS, FS, together with a set of variables $X, Y, \ldots$, we form more complex expressions. *Expressions* are of the form $e ::= X \mid c \mid f \mid (e_1, \ldots, e_n) \mid (e\ e')$, where $c \in CS$,

$f \in FS$. As usual, application associates to the left and parentheses can be omitted accordingly. Therefore $e\, e_1 \ldots e_n$ is the same as $(\,(\ldots(\,(e\, e_1)\, e_2\,)\ldots)\, e_n)$. Of course expressions are assumed to be well-typed. We call those expressions consisting only of variables and constructors *constructor terms*. Constructor terms are irreducible expressions and are meant to denote *values*.

Each function $f \in FS^n$ is defined by a set of conditional rules of the form

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

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 $k = 0$.

The symbol $==$ stands for *strict equality*, which is the suitable notion (see e.g. [Han94]) 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 constructed term. When used in the condition of a rule, $==$ is better understood as a constraint (if it is not satisfiable, the computation fails), but the language contemplates also another use of $==$ as a function, returning the value true in the case described above, but false when a clash of constructors is detected while reducing both sides. Both uses of $==$ are distinguishable by the context.

As a syntactic facility, $\mathcal{TOY}$ allows repeating variables in the head of rules, but in this case repetitions are removed by introducing new variables and strict equations in the condition of the rule. As an example, the rule f X X = 0 would be transformed into f X Y = 0 $\Longleftarrow$ X == Y.

In addition to $==$, $\mathcal{TOY}$ incorporates other predefined functions like the arithmetic functions +,*, ..., or the functions if_then and if_then_else, for which the more usual syntax if _ then _ and if _ then _ else _ is allowed. Symbols ==,+,* are all examples of *infix operators*. New operators can be defined in $\mathcal{TOY}$ by means of *infix* declarations, like infixr 50 ++ which introduces ++ (used for list concatenation, with standard definition) as a right associative operator with priority 50. Operators for data constructors must begin with ':', like in the declaration infix 40 :/:. *Sections*, or partial applications of infix operators like (==3) or (3==) are also allowed.

$\mathcal{TOY}$ 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`.

Our language adopts the so called *call-time choice* semantics for non-deterministic functions, following [Hus93, GH+96]. Call-time choice has the following intuitive meaning: given a function call *(f $e_1 \ldots e_n$)*, one chooses some fixed value for each of the e_i before applying the rules for *f*. As an example, if we consider the function double X = X+X, then the expression (double coin) can be reduced to 0 and 2, but not to 1. As it is shown in [GH+96], call-time choice is perfectly compatible with non-strict semantics and lazy evaluation, provided that *sharing* is performed for all the occurrences of a variable in the right-hand side of a rule.

Computing in $\mathcal{TOY}$ means solving *goals*, which take the form $e_1 == e'_1, \ldots, e_k == e'_k$, 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* in [LLR93] and *needed narrowing* in [AEH94], which uses the so-called *definitional trees* [Ant92] to guide unification with patterns in left-hand sides of rules (see [LLR93]). With respect to higher-order functions, a first order translation following [Gon93] is performed.

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 [Pey87].

Before ending this section we introduce the first example of a program written in $\mathcal{TOY}$. This program is the $\mathcal{TOY}$ version of the evaluator for simple expressions presented by P. Wadler in his article [Wad95], and will be our starting point in order to compare monads and extensions.

The evaluator 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. The definition of the data type expr, together with the definition of the evaluator eval is

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

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

For instance, we may try the goal:

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

which yields R == 0.5.

3 Funcional-Logic monads

In this section we present three variations of the basic evaluator, following the lines of Wadler's paper [Wad95]. 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 $f::A \rightarrow B$ to monadic form we change its type to $f::A \rightarrow m\ B$, 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 $unit::\ A \rightarrow m\ A$ (also known as *result*) and $(*)::m\ A \rightarrow (A \rightarrow m\ B) \rightarrow m\ B$ (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 $f::A \rightarrow m'\ B$, 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 1 shows the three variations of the original evaluator. The first one include exceptions to ensure that a division by 0 is never performed, but instead a suitable error message is displayed. For example the goal

```
type exception = string

data m A = error exception
         | return A

unit:: A → m A
unit A = return A

infixr 30 *
(*)::m A → (A → m B) → m B
(error A) * _ = raise A
(return A) * K = K A

raise::exception → m A
raise E = error E

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

eval (A :/: B) =
     eval A * λR1. eval B * λR2.
     if R2 == 0
     then (raise "divide by zero")
     else unit (R1/R2)
```

```
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)
```

Figure 1: Monadic variations of the basic evaluator

eval (val 3 :/: val 4 :/: val 0 :/: val 7) == R

yields R == (error "divide by zero") while

eval (val 3 :/: val 4 :/: val 5 :/: val 7) == R

succeeds with R == return 1.037037037037037 . The second variation is based on the very useful *state monad* , which is used in this case to count the total number of divisions. Finally, the third variation produces a trace of the operations performed while evaluating the expression. 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 FLP 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 $f{::}A \rightarrow B$. Then, all we need to do is to *extend* the type of the function to $f{::}A \rightarrow B \rightarrow C$, 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 may 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.

Furthermore, the new version of eval needs to satisfy two apparently converse requirements. First of all, it must be able to work with values of type C in order to return the new associated value. But we prefer not to make explicit any value of type C, as we intend to define a reusable version of eval, i.e. the basic structure of the function should not be modified any time we change C by a new C'. The two aims can be accomplished by defining the combination of elements of type C through a suitable function (*)::C → C → C. When later we change C by C' we only need to change the definition of *.

At this point, we know that the second rule for eval needs to 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 *.

However, we still need to associate the value R1/R2 to the result of the evaluation R. This will be performed by function unit, which also should accomplish two goals:

1. It must 'identify' R and R1/R2. In order to generalize the definition to other situations, we may say that unit needs to 'identify' its two input parameters.
2. 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.

```
type exception = string

data status = ok | error exception

unit:: A → A → status
unit A A = ok

infixr 30 *
(*)::status → status → status
ok * K = K
(error A) * _ = error A

raise::exception → status
raise E = error E

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

eval (A :/: B) R =
      eval A R1 * eval B R2 *
      if R2 == 0
      then (raise "divide by zero")
      else unit (R1/R2) R )
```

```
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)
```

Figure 2: Extensions of the basic evaluator

4.2 Extensions of the basic evaluator

The 'extension counterpart' of the monadic variations presented in the previous section may be seen in figure 2. The type C of our discussion is represented respectively by the types **status**, **trans**, **output**, while the unit elements are in the same order **ok**, **if** and " ", where the standard function **id** is defined as usual:

$$\text{id X} = \text{X}$$

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.

Now it can be proved easily that the variations of figure 2 are actually extensions. For example, the pair (" ",++) used in the output extension is known to satisfy the properties of monoids. The proofs for the other two cases are 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 [Wad95]. However, deeper insight into the algebraic properties of extensions and interesting consequences of them should be a matter of further research.

4.4 The do notation

Before ending this section, we introduce a easily definable useful combinator based on * and unit.

The construction do has been introduced in functional languages such as Haskell [HAS97], as a 'syntactic sugar' that gives rise to a more appealing structure of functions, specially when combining a great number of monads in sequence. In our approach we do not need any kind of syntactic modification, for it is possible to define directly the following generalization of *:

$$\mathsf{do} = \mathsf{foldr1}\ (*)$$

and a suitable function ($\longleftarrow$):

$$\mathsf{R} \longleftarrow \mathsf{F} = \mathsf{F}\ \mathsf{R}$$

which permits definitions such as

$$\mathsf{eval}\ (\mathsf{A}\ :/:\ \mathsf{B})\ \mathsf{R} = \mathsf{do}\ [\ \mathsf{R1}{\longleftarrow}\mathsf{eval}\ \mathsf{A},\ \mathsf{R2}{\longleftarrow}\mathsf{eval}\ \mathsf{B},\ \mathsf{R}{\longleftarrow}\ \mathsf{unit}\ (\mathsf{R1}/\mathsf{R2}),\ \mathsf{tick}\ ]$$

for the second rule of the extension of states.

5 A comparative survey

So far we have presented the three 'classical' variations of the basic evaluator, using both extensions and monads. Now we can establish 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

$$\mathsf{type\ output} = \mathsf{string}$$

instead of

$$\mathsf{m\ A} = (\mathsf{A},\ \mathsf{output})$$

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

$$\begin{array}{l} (*) :: \text{output} \rightarrow \text{output} \rightarrow \text{output} \\ \text{M} * \text{K} = \text{M} ++ \text{K} \end{array}$$

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:

$$\begin{array}{l} (*) :: \mathsf{m\ A} \rightarrow (\mathsf{A} \rightarrow \mathsf{m\ B}) \rightarrow \mathsf{m\ B} \\ (\mathsf{X},\mathsf{A}) * \mathsf{K} = \quad \mathsf{let}\ (\mathsf{Y},\mathsf{B}) = \mathsf{K\ A\ in}\ (\mathsf{X}{+}{+}\mathsf{Y},\mathsf{B}) \end{array}$$

• 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:

$$\mathsf{eval\ (A :/: B) = eval\ A\ *\ \lambda R1.\ eval\ B\ *\ \lambda R2.\ out\ (line\ (A :/: B)\ (R1/R2))\ *\ \lambda().\ unit\ (R1/R2)}$$

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

$$\mathsf{eval\ (A :/: B) = eval\ A\ *\ \lambda R1.\ eval\ B\ *\ \lambda R2.\ unit\ (R1/R2)\ *\lambda R.\ out\ (line\ (A :/: B)\ R)}$$

avoiding the unnecessary repeated calculation of $R1/R2$, but this is not possible without changing the definition of out. However the definition of * for extensions allows us to write:

$$\mathsf{eval\ (A :/: B)\ R = eval\ A\ R1\ *\ eval\ B\ R2\ *\ unit\ (R1/R2)\ R\ *\ out\ (line\ (A :/: B)\ R)}$$

where $R1/R2$ is computed only once.
• The separation between the old and the new values also benefits to the definitions of auxiliary functions such as raise, tick or out. For example, as tick must increase the state we only need to write

```
tick::trans
tick = (1+)
```

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

Of course, extensions also 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 the 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, defining unit as

$$\mathsf{unit\ A = [A]}$$

and * as

$$\begin{array}{rcccl} [\,] & * & \mathsf{K} & = & [\,] \\ [A\,|\,X] & * & \mathsf{K} & = & \mathsf{K\ A\ ++\ (X*K)} \end{array}$$

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.

5.1 Efficiency

To compare the efficiency of both techniques in the same FLP setting, we tried the three variations of the evaluator, each one with three different expressions[1] . The two first variations, exceptions and state, were tested with expressions which had 1000, 5000 and 10000 quotients, while the output variation required smaller expressions, namely those with 10, 20 and 30 quotients, due to the great length of the strings returned. The results, in milliseconds, may be seen in table 5.1, and they indicate that the differences between monads and extensions are not great. Only the state monad is clearly faster than the state extension, but not in a very significant factor.

	Exception			**State**			**Output**		
	1000	**5000**	**10000**	**1000**	**5000**	**10000**	**10**	**20**	**30**
Monads	430	2200	4720	560	2950	5570	580	2090	5670
Extensions	480	2950	5740	940	4100	8310	570	2180	5260

Table 1: A time comparative between monads and extensions

We must remark that $\mathcal{TOY}$'s implementation is based on compilation to Prolog, i.e. a $\mathcal{TOY}$ program is translated into a Prolog program which incorporates all that is needed for performing lazy narrowing and equality solving, together with the functions of the source program. Therefore we cannot claim $\mathcal{TOY}$ to be *very* efficient. We think that the interesting point in relation to efficiency in this work is to compare monads and extensions within $\mathcal{TOY}$ itself. However, we have compared results for the monadic variations with the the well-known implementation of the functional language Haskell, *Hugs 1.4* [JP97]. Indeed, the results in *Hugs* are displayed almost immediately, but it cannot deal with expressions of 6000 quotients (in the case of the exception monad) and 3000 (in the case of state monad), while in our implementation the exception monad can cope with expressions up to 40000 quotients and the state monad with expressions of 25000 quotients.

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.

[1]All the results obtained with $\mathcal{TOY}$ 3.5 running over SICStus Prolog 3.6. The machine was a Sun SPARC station 2 with 32 Mb of RAM.

type state = [real]	type state = [real]
tick :: real → m ()	tick :: real → trans
tick A S = ((), insert A S)	tick A = insert A
eval:: expr → m real	eval:: expr → real → trans
eval (val A) = tick A * λ(). unit A	eval (val A) R = tick A * unit A R
eval (A :/: B) =	eval (A :/: B) R =
eval A * λR1. eval B * λR2. unit (R1/R2)	eval A R1 * eval B R2 * unit (R1/R2) R

Figure 3: Evaluator yielding an ordered list, using monads (left side) and extensions (right side)

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, even when only the old value of the function is required. This situation 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 3, 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 1 and 2). 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
```

which returns

```
R == 4
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 following 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 $\Longleftarrow$ 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 none of them.

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 an evaluator of boolean expressions defined as

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

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

evalb:: expr → boolean
evalb (A : \/ : B) = (evalb A) 'or' (evalb B)
evalb (A : /\ : B) = (evalb A) 'and' (evalb B)
```

where functions or and and are defined as usual in functional programming.

Suppose that we decide to add a new feature to evalb, returning not only the result of the evaluation, but also a suitable representation of the expression. The modification may be seen in the figure 4, using monads (left side) and using extensions (right side), and is a simple application of the output feature presented before.

Function convert may be easily defined as

```
convert true   =  "T"
convert 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)

evalb:: expr → m boolean
 	
evalb (val A) = out (convert 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 → boolean → output
	
	evalb (val A) R = out (convert 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 ")"

Figure 4: Boolean evaluator with output, using monads (left side) and extensions (right side)

Imagine 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)) , 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 4 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. Function evalb is used here to generate values of type expr, looking for the one that produces as result "(F and (F or T))". At the same time the expression is evaluated, and therefore we can define an evaluator evalb' that works with string representations:

evalb'::string → bool
evalb' Expr = R ⇐= evalb _ R == Expr

It is worth noticing that, although at a first sight the *generate & test* mechanism we have used may seem rather naive and awkward, it is actually a quite efficient technique. Consider, for example, the parsing of "((T or F) and (F or T))" above. To solve the goal, the strict equality tries to match the string with evalb Expr R, but using laziness, i.e. comparing first the outer constructor of both values, then the next one and so on. This avoids the evaluation of the whole terms if they are not equal.

In our case this means firstly to look for a rule of evalb that produces a symbol '('. Then, following with the same rule, try to produce 'F' and continue searching the expression and the string at the same time. If the strict equality arrives at a point where coincidence is no longer possible, it fails, and the built-in backtracking mechanism takes care of looking for an available alternative. 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. To understand why the free variable R spoils the *generate & test* parsing, we must recall that strict equality does a 'careful matching' as we showed before. Therefore, it first tries to reduce the left hand-side to a pair whose first component begins with '('. It can be done by using the first rule for evalb. That is, evalb Expr is now substituted by (['(' | T] , R'), where the T and R' parts are tried to be generated using the 'or' rule for evalb, or the 'and' if the first possibility is used up. Now, the strict equality tries to get the outer constructor of R', if possible, in order to compare it with R. Of course it is possible, but this necessitates generating a whole expression, in order to determine its result, true or false. And, by using the first 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.

Therefore, we have showed an example where extensions are reusable in two different ways: in the same sense as monads – easily modifiable code – and also in a 'logic sense' due to the multiple modes of use.

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 adventage is that the state extension allows one to dismiss the stateful computations whenever they are not interesting, hence saving both time and space.

Future work should apply extensions to other areas where monads have been successfully employed, such as input/output and the introduction of safe imperative features. We also should study the algebraic properties of extensions and their consequences.

References

[AEH94] S. Antoy , R. Echahed, M. Hanus. *A Needed Narrowing Strategy.* 21st ACM Symp. on Principles of Programming Languages, 268–279, Portland 1994.

[Ant92] S. Antoy. *Definitional Trees*, In Proc. ALP'92, Springer LNCS 632, 1992, 143–157.

[BT95] Y. Bekkers, P. Tarau. *Logic Programming with Monads and Comprehensions*, Proceedings of JFLP'95, 1995.

[CLS97] 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`

[GH+96] J.C. González-Moreno, T. Hortalá-González, F.J. López-Fraguas, M. Rodríguez-Artalejo. *A Rewriting Logic for Declarative Programming.* Procs. of ESOP'96, Springer LNCS 1058, 156–172, 1996.

[Gon93] J.C. González-Moreno. *A Correctness Proof for Warren's HO into FO Translation.* Procs. of GULP'93, 569–585, 1993.

[Han94] 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.

[Han98] 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`

[HAS97] *Report on the Programming Language Haskell: a Non-strict, Purely Functional Language.* Version 1.4, Peterson J. and Hammond K. (eds.), January 1997.

[Hus93] H. Hussmann. *Non-determinism in Algebraic Specifications and Algebraic Programs.* Birkhäuser, 1993.

[JP97] M.P. Jones, J.C. Peterson. *Hugs 1.4 The Nottingham and Yale Haskell User's System. Users Manual.* The University of Nottingham, Technical Report NOTTCS-TR-97-1, 1997.

[LLR93] 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.

[Llo95] Lloys, J.W. *Declarative Programming in Escher.* Technical Report CSTR-95-013, Departament of Computer Science, University of Bristol, June 1995.

[Pey87] S.L. Peyton-Jones. *The implementation of functional languages*, Prentice Hall, 1987.

[PW87] S.L. Peyton-Jones, P. Wadler. *Imperative functional programming*, 20 Annual Symposium on Principles of Programming Languages, Charleston, South Carolina, 1993.

[Wad90] P. Wadler. *Comprehending Monads*, Proc. ACM Conf. on Lisp and Functional Programming, 1990.

[Wad92] P. Wadler. *The essence of functional programming*, Proc. ACM conference on the Principles of Programming Languages, pages 1-14, 1992.

[Wad95] P. Wadler. *Monads for functional programming.* In J. Jeuring and E. Meijer editors, Lecture Notes on Advanced Functional Programming Techniques, Springer LNCS 925. 1995