# An Implementation of Modular Monadic Semantics using Folds and Monadic Folds*

Jose Emilio Labra Gayo

Department of Computer Science
University of Oviedo
C/ Calvo Sotelo S/N, CP 33007, Oviedo, Spain

*e-mail*: labra@lsi.uniovi.es

## Abstract

Modular monadic semantics can be implemented using folds or catamorphisms over a functor that expresses the abstract syntax of the language. By composing several functors, it is possible to obtain modular interpreters from reusable components. Monadic folds structure programs that process recursively their input and control the side effects of the output they produce. We consider that the semantic specification of a programming language is a natural framework for monadic folds that improves the abstraction level and modularity. In this paper we use folds and monadic folds to develop the modular monadic semantics of a simple language.

## Introduction

Using monads to structure denotational semantics [20] facilitates more modular semantic specifications. While traditional denotational semantics maps terms to values, monadic semantics maps terms to computations where a computation is modelled with a monad. In general, it is not possible to compose two monads to obtain a new monad [13], but it is possible to transform a monad into a new monad using monad transformers [14]. Modular monadic semantics [15] is based on monad transformers and allows the development of modular specifications of programming languages.

---

* This research is partially supported by the Project PBP-TIC97-01 "*Sistema Integral Orientado a Objetos: Oviedo3*" from the "*II Plan Regional de Investigación del Principado de Asturias*"

Constructive algorithmics provided a theoretical basis for generic programming that favoured the development of generic functions parameterized by data types. Fold is one of these functions and its application to modular monadic semantics [3] resulted in more modular interpreters. Monadic folds generalise folds and have a great number of practical applications [19]. We consider that using folds and monadic folds in modular monadic semantics will improve the abstraction level and modularity of our interpreters.

As a running example, we present the semantic specification of a simple language of arithmetic expressions and we add boolean expressions in a modular way. Along the paper, we use Haskell syntax and assume that the system accepts multi-parameter type classes with overlapping instances.

Section 1 presents extensible union types, which are a cornerstone of this work because they allow extending not only the value domain, but also the semantic functors. Section 2 shows how to represent the abstract syntax as the fixed point of a functor and how to compose an abstract syntax from independent components. Section 3 introduces folds and section 4 describes the use of folds in modular monadic semantics. Finally, section 5 presents monadic folds and applies them to modular monadic semantics.

## 1 Extensible Union Types

The subtyping mechanism was introduced by S. Liang et. al. [14] in order to allow the future addition of new types to a given union type. By means of the following multi-parameter type class:

```
class SubType sub sup where
  inj::sub -> sup        -- injection
  prj::sup -> Maybe sub  -- projection
```

we can declare that the elements of a union type are subtypes of it

```
instance SubType a (Either a x)
 where inj       = inl
       prj       = either Just (const Nothing)

instance (SubType a b) => SubType a (Either x b)
 where inj       = inr . inj
       prj       = either (const Nothing) prj
```

As an example, if we want a semantic function `f` that returns a boolean value, we will write its type as: `f::(SubType Bool v) => v`. In this way, we will be able to add new types to the value domain without changing the function.

## 2 Modular Abstract Syntax

The abstract syntax of a programming language can be defined as the fixed point of a functor that captures the recursive shape of that language. In Haskell, the fixed point, `Fix f`, of a functor `f` can be defined as:

```
newtype Fix f = In (f (Fix f))
```

For example, the non-recursive functor `N` represents the recursive shape of a simple language of arithmetic expressions:

```
data N a =  Cons Int | Add a a | Dvd a a
```

and the abstract syntax of the language will be:

```
type Arith = Fix N
```

We are interested to compose the abstract syntax in a modular way. L. Duponcheel [3] presents a technique that we call extensible union functors to construct a new functor from two functors:

```
data SumF f g x = S (Either (f x) (g x))

unS (S x) = x

instance (Functor f, Functor g) => Functor (S f g)
 where map g = S . either (Left  . map g)
                          (Right . map g) . unS
```

Using that technique, it is very simple to compose the abstract syntax of a language from independent components. We can define, for example, a new language of arithmetic and boolean expressions as:

```
data B a =  BCons Bool | Not a | Less a a

type ArithBool = Fix (SumF N B)
```

## 3 Initial Algebra and Fold

In Haskell, an `f`-algebra for a functor `f` is a function of type `f a -> a` for some specific type `a`. We introduce the type synonym:

```
type Algebra = f a -> a
```

As an example, the tag function `In` is an `f`-algebra of any functor `f`. That function is special among all the `f`-algebras in the sense that it is the initial algebra in the category of `f`-algebras. This means that for any other `f`-algebra `g` over `a`, there exists a unique function `(fold g)` that can be defined as:

```
fold::(Functor f) => Algebra f a -> Fix f -> a
fold g = g . map (fold g) . out

out (In x) = x
```

## 4 Modular Monadic Semantics using fold

Modular Monadic Semantics [15] represents a monadic approach to structured denotational semantics. It is composed in two parts:

- Semantic building blocks: which define the semantics of individual source language features. They are independent of each other and are defined using monad transformers.

- Monad Transformers: which define the abstract notion of programming language features like continuations, state, exceptions, etc. Multiple monad transformers can be composed to form the underlying monad used by all the semantic building blocks.

We do not go into the details of monad transformers and their composition, which can be consulted in [14], [15] and [16]. With regard to semantic building blocks, L. Duponcheel [3] presents a modular way to compose different blocks using folds. We present his approach to define the semantics of our simple language.

First, we need a monad supports exceptions (to handle division by zero), so we use `ErrMonad` defined by the following type class:

```
class Monad m => ErrMonad m
          where raise::String -> m a
```

The semantics of arithmetic expressions is defined as an `N`-algebra:

```
phiN::(ErrMonad m, SubType Int v) => Algebra N (m v)
phiN (Cons n)     = return (inj n)
phiN (Add mx my) = do { vx <- mx;
                        vy <- my;
                        x::Int <- checkType vx;
                        y::Int <- checkType vy;
                        return (inj (x + y)) }
```

```
phiN (Dvd mx my) = do { vx <- mx;
                        vy <- my;
                        x::Int <- checkType vx;
                        y::Int <- checkType vy;
                        if y == 0
                        then err "Divide by zero"
                        else return (inj (x `div` y)) }
```

`phiN` only assumes that the value domain contains the type `Int` and that the monad of computations is an instance of `ErrMonad`. It will not change if we add more types to the value domain or new features to the monad.

We used the function `checkType` that takes an extensible union type and tries to project it over one of its subtypes.

```
checkType::(ErrMonad m, SubType a b) => b -> m a
checkType = maybe (err "Type Error!") return . prj
```

The semantics of the arithmetic language is easily defined as a fold over the `N`-algebra `phiN`:

```
semN::(ErrMonad m) => Sums -> m Value
semN = fold phiN
```

## 5 Monadic Folds

E. Meijer et al. [19] introduced monadic folds as a technique that enables the combination of programs that abstract from the recursive processing of their input as well as from the side-effects of computing the output. In this sense, we consider that programming language semantics seems a natural framework for using monadic folds since the input is usually a recursive abstract syntax and the output is expressed as a monad of computations.

The generalisation of folds to monadic folds has been studied in [8] and [6] and several restrictions have been stated to obtain them for arbitrary monads and data types. We explore an implementation of their ideas for modular monadic semantics without taking into account the different restrictions that must be imposed for the monads in order to apply it.

Given a monad `M`, we define a monadic function (also called M-resultric) as a function `f` of type `f::a -> M b`.

The composition `(@@)` of monadic functions can be defined as:

```
(@@)::(Monad m) => (b -> m c) -> (a -> m b) -> a -> m c
f @@ g = \x -> g x >>= f
```

From the structure of a polynomial functor `F`, it is possible to derive the definition of a new function `mMap::(a -> M b) -> F a -> M (F b)`. Similar to the `Functor` type class, we define a `MFunctor` type class:

```
class (Monad m, Functor f) => MFunctor f m where
 mMap::(a -> m b) -> f a -> m (f b)
```

An instance of that class could be:

```
instance (Monad m) => MFunctor N m where
 mMap f (Cons n)  = return (Cons n)
 mMap f (Add x y) = do {v <- f x; w <- f y; return (Add v w)}
 mMap f (Dvd x y) = do {v <- f x; w <- f y; return (Dvd v w)}
```

Note that we need to code its definition explicitly, but that it could have been induced from the structure of the functor [8]. Note also that the definition specifies an order of evaluation (which could be different for different monads).

A monadic `f`-Algebra is a function `f::f a -> m a` for some specific monad `m` and type `a`:

```
type MAlgebra m f a = f a -> m a
```

There is an initial monadic `f`-Algebra, which allows the definition of monadic fold as:

```
mFold::(MFunctor f m) => MAlgebra m f a -> Fix f -> m a
mFold phi = phi @@ mMap (mCata phi) @@ (return . out)
```

The semantics of arithmetic expressions can now be expressed as:

```
mPhiN::(ErrMonad m, SubType Int v) => MAlgebra m N v
mPhiN (Cons n)    = return (inj n)
mPhiN (Add vx vy) = do { x::Int <- checkType vx;
                         y::Int <- checkType vy;
                         return (inj (x + y)) }
mPhiN (Dvd vx vy) = do { x::Int <- checkType vx;
                         y::Int <- checkType vy;
                         if y == 0 then err "Divide by zero"
                         else return (inj (x `div` y)) }

semN'::(ErrMonad m) => Arith -> m Value
semN' = mFold mPhiN
```

The main advantage of monadic folds is to separate the evaluation of sub-expressions (which could have been obtained automatically from the datatype declaration) from the semantic specification.

We can also describe the semantics of boolean expressions:

```
instance MFunctor B where
 mMap f (BCons n) = return (BCons n)
 mMap f (Not x)   = do{ v <- f x; return (Not v) }
 mMap f (Less x y)= do{ v <- f x; w <- f y; return (Less w w)}

mPhiB::(ErrMonad m,
        SubType Int v, SubType Bool v)=> MAlgebra m B v
mPhiB (BCons b)    = return (inj b)
mPhiB (Not  vx)    = do { x::Bool <- checkType vx
                        ; return (inj (not x)) }
mPhiB (Less vx vy) = do { x::Int <- checkType vx
                        ; y::Int <- checkType vy
                        ; return (inj (x < y))
                        }
```

Finally, we can join both semantics using extensible union functors:

```
sem::(ErrMonad m) => ArithBool -> m Value
sem = mFold (either mPhiN mPhiB . unS)
```

## Related Work

The use of monads to structure denotational semantics was anticipated by E. Moggi [20]. P. Wadler [25], [26] popularized monads for functional programming. Apart of other applications, he presented the incremental development of an interpreter using monads. Inspired in that work, there was a special interest in monad-based modular interpreters. Several attempts were made by Steele [24], Espinosa [4] and finally Liang et al. [14] who developed modular monadic semantics based on monad transformers.

On the other hand, E. Meijer [17] used folds to develop the semantics of programming languages and obtain compilers by calculation. A number of categorical techniques were applied for calculating and transforming programs using fold for arbitrary data types in [5], [22] and [18]. Recently, PolyP [12] presents an implementation of these theoretical developments and includes a library [11] of polytipic functions (including fold as a standard example).

With regard to monadic folds, Fokkinga [6] and Hu [8] developed the theoretical background to adapt the work made with folds to monadic folds, and Meijer et al. [19] advocated their use for practical applications.

Recently, G. Hutton [10] presents the relationship between denotational semantics (described using folds) and operational semantics (using unfolds) and alludes the future work on monadic folds.

Duponcheel [3] applied folds to the development of more modular monadic semantics, which inspired the present work.

Recent advances on modular monadic semantics are the connection between modular monadic semantics and action semantics [27] and the development of modular compilers from monad transformers [7].

## Conclusions and Future Work

Using folds to structure modular monadic semantics allows the development of more modular interpreters. Monadic folds adapt naturally to this framework and facilitate the abstraction of sub-component evaluation from semantic specification. However this separation does not work for all kinds of monads. Some constructs, like different call mechanisms, need an explicit control of sub-component evaluation, which jeopardise the applicability of this approach. A lot of work remains to be done on the theoretical side, especially to consider the interaction between different monad transformers and monadic folds.

Although we have only presented the interpreter of a very small language, we have developed a more complete interpreter using folds or monadic folds depending on the semantic feature modelled. The interpreter is assembled from independent components using extensible union functors. We could even develop a modular parser implementing an algebra for each functor over the Parser monad [9]. Nevertheless, our parser is far from elegant because we need extra parenthesis to allow the modular nature. Another research direction is the modular development of parsers from modular abstract syntax. In this sense, the dual notion of monadic folds, monadic unfolds, could be used to obtain the abstract syntax tree from the input string.

Fold and monadic folds are polytipic functions [12]. PolyP is a Haskell extension that supports polytipic functions and its adoption seems a natural choice that we want to explore in the future.

Another goal for future research is to obtain compilers from monadic specifications using partial evaluation or staging ([1], [7], [23]).

Finally, we want to mention that our goal is the semantic specification of several abstract machines that are being built as part of our work on the Oviedo3 Project [21].

## References


1. O. Danvy, J. Koslowski, K. Malmkjaer, Compiling Monads. Technical Report.*CIS-92-3* (Kansas State University, 1991).

2. L. Duponcheel, E. Meijer, On the expressive power of Constructor Classes, Glasgow Functional Programming Workshop, Ayr, Scotland (Springer-Verlag, 1994).

3. L. Duponcheel, Writing modular interpreters using Catamorphisms, subtypes and monad transformers. *Research Report (draft)* (Utrecht University, 1995).

4. D. Espinosa, Semantic Lego, PhD. Thesis. (Columbia University, 1995).

5. L. Fegaras, T. Sheard, Revisiting Catamorphisms over Datatypes with Embedded Functions, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida (1996).

6. M. Fokkinga, Monadic Maps and Folds for Arbitrary Datatypes. Technical Report.(Dept. INF, University of Twente, 1994).

7. W. L. Harrison, S. N. Kamin, Modular Compilers Based on Monad Transformers, IEEE Computer Society International Conference on Computer Languages, Loyola University, Chicago (1998).

8. Z. Hu, H. Iwasaki, Promotional Transformation of Monadic Programs, Fiji International Workshop on Functional and Logic Programming, Susono (World Scientific, 1995).

9. G. Hutton, E. Meijer, Monadic Parser Combinators. Technical Report.*NOTTCS-TR-96-4* (University of Nottingham, 1996).

10. G. Hutton, Fold and Unfold for Program Semantics, 3rd ACM International Conference on Functional Programming, Baltimore, Maryland (1998).

11. P. Jansson, J. Jeuring, PolyLib - a library of polytipic functions. Unpublished Manuscript.(Chalmers University of Technology, 1998).

12. J. Jeuring, P. Jansson, in *Advanced Functional Programming* J. Launchbury, E. Meijer, T. Sheard, Eds. (Springer-Verlag, 1996), LNCS 1129, pp. 68-114.

13. M. P. Jones, L. Duponcheel, Composing monads. Research Report.*YALEU/DCS/RR-1004* (Dept. of Computer Science, Yale University, 1993).

14. S. Liang, P. Hudak, M. Jones, Monad transformers and modular interpreters, POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California (1995).

15. S. Liang, P. Hudak, Modular denotational semantics for compiler construction, ESOP'96: 6th European Symposium on Programming, Linkoping, Sweden (Springer-Verlag, 1996).

16. S. Liang, Modular Monadic Semantics and Compilation, PhD. Thesis. Computer Science, Yale (Yale University, 1998).

17. E. Meijer, More Advice on Proving a Compiler Correct: Improve a Correct Compiler, Phoenix Seminar and Workshop on Declarative Programming (Springer-Verlag, 1992).

18. E. Meijer, G. Hutton, Bananas in Space: Extending fold and unfold to exponential types, Functional Programming Languages and Computer Architecture, La Jolla, California (1995).

19. E. Meijer, J. Jeuring, Merging monads and folds for functional programming, First International Spring School on Advanced Functional Programming Techniques (Springer-Verlag, 1995).

20. E. Moggi, An abstract view of programming languages. Technical Report.*ECS-LFCS-90-113* (University of Edinburgh, 1989).

21. Oviedo3 Project. URL: http://www.uniovi.es/~oviedo3

22. T. Sheard, L. Fegaras, A fold for all seasons, Proc. Conference on Functional Programming Languages and Computer Architecture, Copenhagen (1993).

23. T. Sheard, Z. Benaissa, From Interpreter to Compiler using Staging and Monads (Pacific Software Research Center, Oregon Graduate Institute, 1998).

24. G. L. Steele, Building Interpreters by composing Monads, POPL' 94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon (ACM Press, 1994).

25. P. Wadler, The Essence of Functional Programming, 19th ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico (ACM, 1992).

26. P. Wadler, Monads for Functional Programming, J. Jeuring, E. Meijer, Eds., Bastad Spring School on Advanced Functional Programming, Bastad (Springer Verlag, 1995).

27. K. Wansbrough, J. Hamer, A Modular Monadic Action Semantics, Domain Specific Languages, Santa Barbara, California (The USENIX Association, 1997).