Modular Evaluation and Interpreters
Using Monads
and Type Classes

by Dan Popa, Ro/Haskell Group
Dept. Comp. Sci. Univ. Bacău, Romania
popavdan@yahoo.com

based on papers provided by the Haskell community and some other resources.
Anglo Haskell 2008
Abstract

During the last decade, the expression evaluators (interpreters) and the list monad had attracted both mathematicians (especially from the field of Categories Theory) and computer scientists.
Abstract

During the last decade, the expression evaluators (interpreters) and the list monad had attracted both mathematicians (especially from the field of Categories Theory) and computer scientists.

For the last group, the main kind of applications comes from the field of DSL interpretation.
**Abstract**

During the last decade, the expression evaluators (interpreters) and the (list) monad had attracted both mathematicians (especially from the field of Categories Theory) and computer scientists.

For the last group, the main kind of applications comes from the field of DSL interpretation.

As a consequence of our research, we are able to introduce a new kind of modular interpreter or expression evaluator, which can be build by importing modular components into a main Haskell program.
Modularity, OK! But how to get it?:

1) Modular parser = ? Problem solved! Parser comb.

2) Modular trees = ? Nobody seems to try it!

3) Modular implementation of the interpreter = ?
   
   interpret :: Term -> Env -> M Value – not modular should be replaced by something else.
Modularity! Let's show how to get it:

1) In order to keep the parser of the DSL modular, parser combinators were used.
Modularity! Let's show how to get it:

1) In order to keep the parser of the DSL modular, parser combinators was used.
2) In order to keep the source (and the AST) modular we have replaced the data constructors by regular functions over the list monad, inspired by an idea of Simon P.J from the [Haskell Report]. He said that data constructors are in fact just simple functions.
Modularity! Let's show how to get it:

1) In order to keep the parser of the DSL modular, parser combinators were used.
2) In order to keep the source (and the AST) modular we have replaced the data constructors by regular functions over the list monad, inspired by an idea of Simon P.J from the [Haskell Report]. He said that data constructors are in fact just simple functions.
3) This gave us the general idea of the replacement of data constructors by functions over monadic actions, called by us "pseudoconstructors".
Modularity! Let's show how to get it:

The modular evaluator was written in do-notation, based on the idea that expressions should evaluate themselves nor by the help of an interpret-function as in [Tim Sheard and Abidine. et all].
Modularity! Let's show how to get it:

The modular evaluator was written in do-notation, on the idea that *expressions should evaluate them self* nor *by the help of an interpret-function* as in [Tim Sheard and Abidine. et al].

As a consequence, the useful data declarations which usually appears in DSL implementations are completely missing, shortening the source and reducing the work of the programmer.
1) Tree declarations like this are harmful (from the modularity point of view)

data Exp =  Constant Int
| Variable String
| Minus Exp Exp
| Greater Exp Exp
| Times Exp Exp

deriving Show
1') *Drop the declarations like this one, too!*

```haskell
data Com = Assign String Exp
           | Seq Com Com
           | Cond Exp Com Com
           | While Exp Com
           | Declare String Exp Com
           | Print Exp

deriving Show
```
2) **A new vision of monadic semantics**

A new vision of monadic semantics is now introduced. The semantics is not a function:

```
interp :: Term -> Environment -> Monad
```
A new vision of monadic semantics is now introduced. The semantics is not a function:

\[
\text{interp :: Term} \rightarrow \text{Environment} \rightarrow \text{Monad}
\]

but more likely a sort of

\[
\text{Monad} \rightarrow \text{Monad} \rightarrow \ldots \text{Monad}
\]

where the name is given by the pseudoconstructor itself.
2) A new vision of monadic semantics

Example:

A new vision of monadic semantics is now introduced. The semantics is not a function:

\[ \text{interp :: Term -> Environment -> Monad} \]

but more likely a sort of

\[ \text{Monad -> Monad -> ...Monad} \]

where the name is given by the pseudoconstructor itself.

\[ \text{Plus :: Exp -> Exp -> Exp} \]

will be replaced by a plus:
2) A new vision of monadic semantics

Example:

A new vision of monadic semantics is now introduced. The semantics is not a function:

```
interp :: Term -> Environment -> Monad
```

but more likely a sort of

```
Monad -> Monad -> ...Monad
```

where the name is given by the pseudoconstructor itself.

```
Plus :: Exp -> Exp -> Exp
```

will be replaced by a plus:

```
plus :: [ a] -> [ a] -> [ a] or a plus :: M a -> M a -> M a
```
2) A new vision of monadic semantics. Consequences:

1. The data declarations of the trees will be absent being replaced by a set of functions.

\[
\text{data Exp} = \text{Constant Int} \\
| \text{Variable String} \\
| \text{Minus Exp Exp} \\
| \text{Greater Exp Exp} \\
| \text{Times Exp Exp}
\]
2) A new vision of monadic semantics.
Consequences:

1. The data declarations of the trees will be absent being replaced by a set of functions.
   . . . are replaced by . . .

   constant :: Integer -> [Integer]
   variable :: String -> [Integer]
   minus :: [Integer] -> [Integer] -> [Integer]
   greater :: [Integer] -> [Integer] -> [Integer]
   times :: [Integer] -> [Integer] -> [Integer]
2) A new vision of monadic semantics.

Consequences:

1. The data declarations of the trees will be absent being replaced by a set of functions. . . . or even more generally . . .

   constant :: Integer -> M Integer
   variable :: String -> M Integer
   minus :: M Integer -> M Integer -> M Integer
   greater :: M Integer -> M Integer -> M Integer
   times :: M Integer -> M Integer -> M Integer

   ... M being an other monad, not only the list monad.
2) A new vision of monadic semantics.

Consequences:

1. The data declarations of the trees will be absent being replaced by a set of functions.

So: \[ \text{Minus (Variable "x") (Variable "y")} \]

will be replaced by a slightly different version:

\[ \text{minus (variable "x") (variable "y")} \quad (*) \]

where \text{minus}, \text{variable} and \text{so ...are called "pseudoconstructor"}.
2) A new vision of monadic semantics. Consequences:

1. The data declarations of the trees will be absent being replaced by a set of functions.

So: \[ \text{Minus ( Variable “x”) (Variable “y”)} \]

will be replaced by a slightly different version:

\[ \text{minus (variable “x”) (variable “y”) (*)} \]

where minus, variable and so ...are called “pseudoconstructors”.

Remark: The relation (*) are representing both syntax (being unevaluated) and semantics (when Haskel's lazy evaluation mechanism decides to compute the final value) in the same time!
2) A new vision of monadic semantics. Consequences:

1. The data declarations of the trees will be absent being replaced by a set of functions.
2. There is no needs for such functions to be together, in the same module.

We can describe / declare:

\[
\text{log} :: [\text{Float}] \rightarrow [\text{Float}] \rightarrow [\text{Float}] \quad \text{in a module and}
\]
\[
\text{plus} :: [\text{Float}] \rightarrow [\text{Float}] \rightarrow [\text{Float}] \quad \text{in an other module}
\]

and still be able to mix them in syntax and computations:

\[
(\text{plus} \ (\text{variable \ “x”}) \ (\text{log} \ (\text{constant} \ 2)\ (\text{variable \ “y”})))
\]
2) A new vision of monadic semantics. Consequences:

1. The data declarations of the trees will be absent being replaced by a set of functions.
2. There is no needs for such functions to be together, in the same module.

Or even more, we can describe / declare:

\[
\text{log :: } [\text{Float}] \rightarrow [\text{Float}] \rightarrow [\text{Float}]
\]
\[
\text{in a module and}
\]
\[
\text{plus :: } [\text{Float}] \rightarrow [\text{Float}] \rightarrow [\text{Float}]
\]
\[
\text{in an other module}
\]

and still be able to mix them in syntax and computations:

\[
(\text{plus (variable “x”)})(\text{log (constant 2)(variable “y”)})
\]
\[
\ldots \text{M being any other selected monad} \ldots
\]
2) A new vision of monadic semantics.

Consequences:

1. The data declarations of the trees will be absent, being replaced by a set of functions.
2. There is no need for such functions to be together, in the same module. We can spread such functions in different modules, providing modularity. And, last but not least, because of the monad:

3. We can use the do-notation in order to express computations:

```haskell
plus x y = do { vx <- x;
    vy <- y;
    return (vx + vy);
} :: [Float]
```
2) A new vision of monadic semantics. Consequences:

1. The data declarations of the trees will be absent, being replaced by a set of functions.  
2. There is no need for such functions to be together, in the same module.  
   We can spread such functions in different modules, providing modularity. And, last but not least, because of the monad:

3. Remember: The traditional solution was usually more complex and all those “do”-s were stick together in the same function.

```haskell
do { vx <- interp x env;
    vy <- interp y env;
    return (vx + vy); } :: M Float
```
2) A new vision of monadic semantics. Conclusions:

A new vision of monadic semantics is now introduced. The semantics is not a function:

\[
\text{interp :: Term -> Environment -> Monad}
\]

but more likely a sort of

\[
\text{Monad -> Monad -> ...Monad}
\]

specification in contrast with the papers [P.W.123] of Philip. Wadler.

Remember idea and definition of pseudoconstructor functions over monadic actions. The pseudoconstructor are replacing the data values constructors from the right side of a data declaration.
3) Where is the environment when we need it?

```haskell
plus x y = do {
  vx <- x;
  vy <- y;
  return (vx + vy);
} :: M Float
```

The code seems to have the environment hidden or no environment at all!

Idea: If an *environment* is needed (and usually it is !) the list monad may be replaced with an other state or writer monad. Anyway, for simple expressions using constants and operators the list monad is enough.
4) May we have overloaded functions?

Usually, some arithmetic operators are overloaded:

\[
\text{plus } x \ y = \ do \{ \ vx <- x; \\
vy <- y; \\
return (vx + vy); \}
:: [\text{Float}]
\]

\[
\text{plus } x \ y = \ do \{ \ vx <- x; \\
vy <- y; \\
return (vx + vy); \}
:: [\text{Integer}]
\]

Can we use two or more kind of plus in different modules?
4) May we have overloaded functions?
Answer:
YES, using multiparameter type classes

module MyPlusFloat where
import MyFloat
import ClassPlus

instance Plus Float Float Float where
    plus x y = do
        vx <- x;
        vy <- y;
        return (vx + vy);
    :: [Float]

Exercise: Write similars modules: MyPlusInt,
          MyPlusChar, MyPlusComplex, ...
4) May we have overloaded functions?
Answer: 
YES, using multiparameter type classes

module MyPlusFloat where
import MyFloat
import ClassPlus

instance Plus Float Float Float where
  plus x y = do { vx <- x;
                 vy <- y;
                 return (vx + vy); }
  :: [Float]

-- Example: modular specification for an
overloaded "plus" using a multiparameter
type class: ClassPlus. It looks like...
4) **Example: modular specification for an overloaded “plus” using a multiparameter type class: ClassPlus. It looks like...**

```haskell
module ClassPlus where

class Plus a b c where
    plus :: [a] -> [b] -> [c]

{------------------------------------------
A triple of types a b c belongs to the Plus Class “ClassPlus” if (and only if)
there exist a function “plus” having the signature as above.
The hypothesis that three types belongs (as a triple) to the ClassPlus will be
provided by an instantiation of that class ...Pleas go back to see it again !!!
--}
```
4) May we have overloaded functions? 
YES, even with a different monad, M.

module ClassPlus where

class Plus a b c where
    plus :: M a -> M b -> M c

{------------------------------------------
You are free to use any traditionally used
monad, for example the StOut monad from the
paper of [Tim Sheared], or any other monad
built by help of transformers.

--}
4) But how are the numbers defined?
4) **But how are the numbers defined?**

**First solution:**

```haskell
module MyNum where

--- Modular evaluator for Integers producing monadic values [Integer] in the list monad.

evalnum :: Integer -> [Integer]
evalnum x = [x]

--- The pseudoconstructor is producing monadic values, in this case (one element) lists.

constant :: Integer -> [Integer]
constant x    = do { vx <- evalnum x ;
                     return vx ; }

...well, we will not discuss optimization, yet!
```
4) When an evaluator / interpreter is build all the required modules are used:

module ParserSumaCifre where  --main prg.
import Monad                   --use monads,
import ParseLib                --parsers,
import MyNum                   --numbers,
import ClassPlus               --plus,
import ClassMinus              --minus:
import MyPlusNum               --one plus
import MyMinusNum              --one minus

-- Remark: Other parser combinators (like Parsec) may be used instead of ParseLib, or we can work only with pseudoconstructors:
4') Run an evaluation: pseudoconstructor and overloading specification
4") Optimizing a module using monad's laws:

module MyChar where

evalchar :: Char -> [Char]
evalchar x = [x]

----Old implementation of the pseudoconstructor
--char :: Char -> [Char]
--char x = do { vx <- evalchar x;
--            return vx; } ----Applying monad's law =>

----New implementation of the pseudoconstructor
char :: Char -> [Char]
char x = [x]
5) *Have we lost space, gaining modularity?*

Three solutions was compared:

**Cyclam** = Standard evaluator:  
Parser, Trees, Integer

**Yellow** = Modified std. evaluator:  
Parser, Trees, [Integer], Lists  
--to see how much overload is got by lists

**Magenta** = New monadic evaluator:  
Parser, no Trees, Modularity, [Integer], ListMonad
5) Space consumed adding lists and modularization: Conclusions

Adding lists increases space with aprox 2.5%
Adding modularity increases space again with aprox 2-3%
5') Final conclusion: +10% space is an acceptable price for the modularity of the languages

Diagram of our small example:
6) Anexa: Traditional evaluator

Usually, an evaluator receive an expression, a context and produces a result stored by a monadic “capsule”.

```haskell
eval1 :: Exp -> Index -> M Int
eval1 exp index = case exp of
  Constant n -> return n
  Variable x -> let loc = position x index
               in getfrom loc
  Minus x y -> do { a <- eval1 x index ; b <- eval1 y index ; return (a-b) }
```
6) Anexa: Traditional evaluator (cont.)

Greater x y -> do { a <- eval1 x index ;
    b <- eval1 y index ;
    return (if a > b
    then 1
    else 0) }

Times x y -> do { a <- eval1 x index ;
    b <- eval1 y index ;
    return ( a * b ) }
6) **Selective Bibliography:**

References, names, papers, books, sites used:

- Leijen Daan – a lot of papers concerning Parsec
- Tim Sheard and Abidine, DSL implementation using staging and monads...
- Hutton Graham; Meijer Erik - a lot of papers on monadic parsing
- Peyton Jones Simon - The History of Haskell
- haskell org – pages including those of monad laws
- Autrijus/Audrey Tang- all about Perl 6
- Philip Wadler - a lot of papers concerning monadic interpreters
- Zenger Matthias – his Ph.D Thesis

*Extra readings: -- interpreters evaluators and virtual machines, the list monad... sorry if somebody else is missing...*