UNIVERSITATEA DIN BACĂU
STUDII ŞI CERCETĂRI ŞTIINȚIFICE
Seria: MATEMATICĂ
Nr. 18 (2008), pag. 233 – 248

DIRECT MODULAR EVALUATION OF EXPRESSIONS USING THE MONADS AND TYPE CLASSES IN HASKELL

by
DAN POPA

Abstract. During last decade, the expression evaluators and the list monad had attracted both mathematicians (especially from the field of Category 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 tree-less expression evaluator, which can be build by importing modular components into a main Haskell program. In order to keep the parser of the DSL modular, parser combinators from ParseLib [Hutton G., Meijer E., (1998)] was used. In order to keep the source and the implicit syntax tree *modular* we have replace the data constructors by regular functions over the list monad, inspired by an idea from Haskell Report [Peyton Jones S. (editor) (2002)]: *data constructors are in fact just simple functions*. This gave us the idea of the replacement of data constructors with functions over monadic actions called by us *pseudoconstructors*. 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 some papers like [Sheard T.; Benaissa Z. ; Pasalic E. (1999)] and others. 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. 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 specification in contrast with the papers [Wadler P. (1992-1995)] .

Key words and phrases: Modular Monadic Interpreters, Type Classes, Haskell.
(2000) Mathematics Subject Classification: 68N20, 18-04.

Let's note the idea and definition of *pseudoconstructors* functions over monadic actions. The *pseudoconstructors* are replacing the data values constructors from the right side of a data declaration. The paper was accepted as a talk by Anglo Hasekell 2008 organizers.

1. Introduction

Bigger interpreters and/or compilers has to be serviced from time to time, as the language itself evolves by versioning. On the other hand, compilers and interpreters are a sort of strong connected systems [Zenger M. (2004), Odersky M, Zenger M (2005)], having their pieces strongly bind together. For decades, modular adaptable languages and compilers or interpreters were a sort of Saint Grail of the computer scientists community. The main problem was that we usually can not modify some parts of the system without the needs of rewriting other parts of it . Examples: If a fixed lookahead grammar is modified by adding a simple (but uninspired) rule, the so called "first" and "follow" sets have to be computed again and a monolithic parser have to be build again. The problem was finally solved by the introduction of monadic parser combinators in [Hutton G., Meijer E., (1998)] . The story of parser combinators is classified in [Hudak,P; Hughes,J; Peyton Jones S, Wadler, P, (2007)] as a successful story. Same problems were still encountered when dealing with the semantics. In order to modify the semantic of the language, all (or almost all) the recursive definitions of it have to be rewritten. No modularity here,too. The problem of modular semantics was solved by D.Espinosa in his PhD thesis, [Espinosa D. (1995)] using modular monadic semantics written in Scheme. The monad laws becomes the support of the do-notation (in Haskell) and connecting them with the papers of [Wadler P. (1992-1995)] the way of the monadic interpretation of trees in do-notation was open. Languages and DSL-s was implementing in this way, including the Perl 6. [Tang. A; (2005)]. But the syntax trees are still used and, because the data declaration in Haskell is not modular (as the instance declarations is) , the whole system is not completely modular.

To our knowledge, our work, in this paper, shows and marks the first modular monadic tree-less interpreter. Shockingly enough, although conventional interpreters and compilers writers states that syntax trees are a fundamental structure of a language we believe that a different approach is necessary. Now, we view the parsers from the parser monad as returning usual functions (over a monad) nor data constructors of some trees. As a consequence of laziness of the Haskell language, the dynamically produced structure of the functions calls will work similarly with the tree and will not be evaluated (it is still incomplete during the parsing phase) until the right moment comes. Therefore, we prove that the usual interpretation interp: Term -> Env -> Term is harmful (in terms of modularization) and can be replaced by *pseudoconstructors* defined on monadic values. Motivated by these observations, a smart methodology for modular tree-less interpreters and compilers building was developed. This is a direct result of the elimination of abstract syntax tree declared with data constructors. Next point, the disadvantage of this type of approach, however, is that some usual syntax trees processing as the tree optimization have to be embedded somehow in the *pseudoconstructors* or inserted somehow between the return of the parser and the use of our *pseudoconstructors*. Various kind of operations with terms can be plugged in the system by simply instantiating the required class of operators. And we had realized our objectives in a modular, monadic, tree-less way, therefore highly adaptable.

2. Model

The "Direct Modular Evaluator of Expressions Using The List Monad and Type Classes" (DMEEULMTC) relies on one public Haskell library outlined in the classic famous work [Hutton G., Meijer E., (1998)] in the field of monadic parsing (and, of course, the list monad). A new one, like Parsec, described in [Leijen D.; Meijer E.; (2001)] was also used by us with good results. The monad library is also included. Figure 1 diagrams the conceptual relationship between our semantic modules and the modular monadic parser which provide the data *pseudoconstructors* according with the syntax of the expression (term or program).

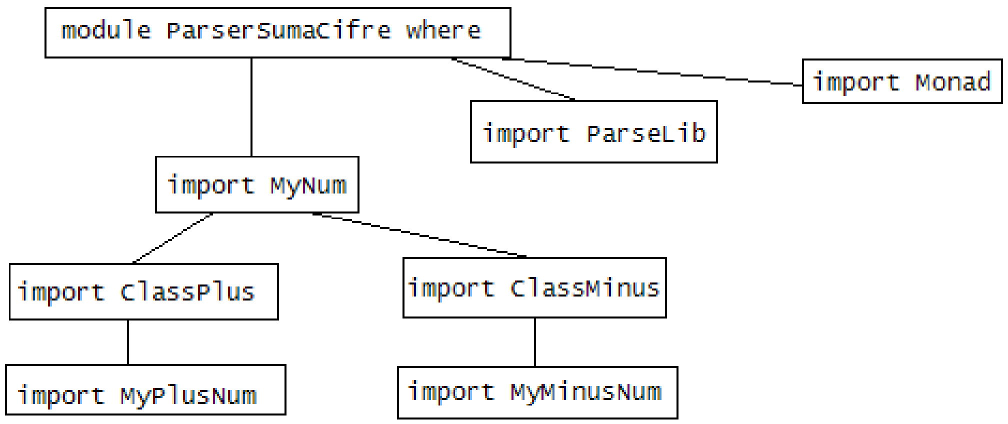


Figure 1: A simple DMEEULMTC tool.

The main module ParserSumaCifre which uses the Monad and the

ParseLib. The evaluated terms will consist of numbers (MyNum) and two different kind of operators (ClassPlus and ClassMinus). In every class we have declared a plug-in which is an instance of the class .The class is giving the signature of the operator. The instance provides the actual semantics of the operator. Overloading and type check are supported, both.

There is a huge set of references concerning interpreters and evaluators or virtual machines. The reader may want to check extra references. The list monad is also a well documented subject. Therefore, the architecture of a simple DMEEULMTC tool is still not far away of others similar (but not modular) tools. Any natural expression evaluator or interpretor used by a client-server technology will clearly require a syntax of the terms or/and programs and a parser. Our tool is not different from this point of view. We considered a parsing algorithm based on parser combinators [Hutton G., Meijer E., (1998)] or [Leijen D.; Meijer E. ; (2001)] which proved to be modular. As it is already known known, a parsing system does not require a syntax tree structure to parse correctly. Our system will produce a structure of *pseudoconstructors* (syntactically speaking they all looks like the similar tree constructors but - manipulates monadic values and – they are not written using a capital in the beginning of the identifier).

One of the problems was if we can add other types ? After few experiments we concluded that other types may be added by including other modules similarly with MyNum. The extensible architecture of a DMEEULMTC tool consists of independent semantic specifications for the operators belonging to one class, several independent classes of operators related to one (ore more) data-types like MyNum and all this semantic specification is connected to a modular parser - the application itself.

The correct semantic behavior of a DMEEULMTC tool and the semantic errors reported depends on the carefully implementations of the operators in the class instances (like MyPlusNum). The syntax error reports, on the other side, depends on the parser combinator library used (better with Parsec).

We have ran a set of tests confirming that our architecture is feasible. Excluding the libraries (Monad and parser combinator library) the architecture for our system consists of four independent components:

- The main program including the modular parser.
- The set of data types: MyNum, MyFloat, MyChar, My Bool and so...
- A class describing the signature (types and result, monadic packed) for any kind of operators. (ClassPlus, ClassMinus, ClassMult,ClassDiv end so...)
- Monadic Semantics modules written as instances of the classes above.

3. From the Idea to the Implementation

3.1 History of aproaching modularity

1) Modular parser = ? The Problem was solved by .... Parser combinators are a real success story.
2) Modular trees = ? Nobody seems to try it ! Here is the place were we can work.
3) Modular implementation of the interpreter = ? Usually (in the papers by ...) the interpreter is defined as a function working on Terms and Environment and producing monadic Values.

```
interpret :: Term -> Env -> M Value
```

Such a function is not modular (it can not be decomposed and spread into different Haskell modules!)

So, it should be replaced by something else.

3.2. How to obtain the modularity

1) In order to keep the parser of the DSL modular, parser combinators was used, being a tested solution.
2) In order to keep the source (and the abstract syntax tree) 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.

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 all].

As a consequence, the useful data declarations which usually appears in all DSL implementations are completely missing, shortening the source and reducing the work of the programmer.

3.3. Tree declarations like this - below - 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

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

Adding new variants means to rewrite such declarations, in a way that it is not modular.

3.4. 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
```

but more likely a sort of Monad -> Monad -> ... -> Monad

where the name is given by the pseudoconstructor itself. Let's see an example:

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

will be replaced by a plus:

plus :: [a] -> [a] -> [a] or a plus :: M a -> M a -> M a (M being *any* Monad !)

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

Let's see how, using a bigger example:

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

This data declarations becomes a set of functions having this set of signatures:

```
constant :: Integer -> [Integer]
variable :: String  -> [Integer]
minus :: [Integer] -> [Integer] -> [Integer]
greater :: [Integer] -> [Integer] -> [Integer]
times :: [Integer] -> [Integer] -> [Integer]
```

So: Minus (Variable “x”) (Variable “y”)

will be replaced by a different version:

minus (variable “x”) (variable “y”)   (*)

where minus, variable and so ...are called *pseudoconstructors*.

Remark 1: The relation (*) are representing both syntax (being unevaluated) and semantics (when Haskel's lazy evaluation mechanism decides to compute the final semantic value) in the same time!
Remark 2: There is no needs for such functions to be together, in the same module. We can describe / declare, for example,:

```
log :: [Float] -> [Float] -> [Float ]          in a module and
plus :: [Float] -> [Float] -> [Float]       in an other module
```

and still be able to mix them in syntax and computations by using something like this, which is not a tree as you might expect:

(plus (variable “x”) (log (constant 2)(variable “y”))

Notice the non capitals letters from the beginning of the identifiers.

We are using the do-notation in order to express computations:

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

but we have to specify the type of the monadic action which is the result. Remember: The traditional solution was usually more complex and all those "do"-s were stick together in the same function. Here is one of them:

```
  do {  vx <- interp x env;
        vy <- interp y env;
        return  (vx +  vy); }    :: M Float
```

Remark: Our specifications are in contrast with the papers [Wadler P. (1992-1995)]. Remember the idea and the definition of *pseudoconstructors* as functions over monadic actions: The *pseudoconstructors* are replacing the data values constructors from the right side of a data declaration.

3.6. Where is the environment when we need it ?

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

As you may notice: This code – above - 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 (or even the identity monad) is enough.

3.7. May we have overloaded functions ? Usually, some arithmetic operators are overloaded:

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

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

Or, more generaly:

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

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

Can we use two or more kind of “plus” in different modules? After some experiments the answer was: Yes, using multi-parameter type classes, which is a common extension of Haskell 98. (i.e. the program must be run using the “-98” switch with Hugs.)

```
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]
```

A more general form is:

```
module MyPlusFloat where
import MyFloat
import ClassPlus
instance Plus Float Float Float where
 plus x y  =   do { vx <-  x;   vy <-  y;
                    return  (vx +  vy); } :: M Float
```

M being an arbitrary monad.

Exercise for the ambitious reader: Write similars modules: MyPlusInt, MyPlusChar, MyPlusComplex, ...

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

```
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 ClassPluss will be provided by an instantiation of that class, as we saw.

You are free to use any traditionally used monad, for example the StOut monad from the paper of [Sheard T.; Benaissa Z. ; Pasalic E. (1999)], or any other monad, built using monad transformers.

```
--}
```

3.8 But how are the numbers defined?

Let's see the module which is used to define numbers:

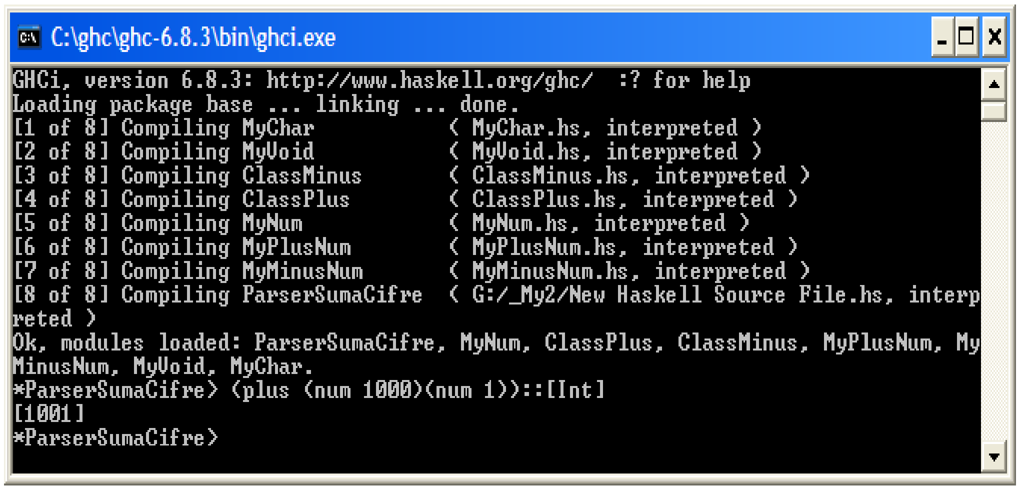
```
module MyNum where
--- Modular evaluator for Integers producing monadic values [Integer] in the
list monad.

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

---The pseudoconstructor is producing monadic values, in this case lists
having exactly one element.

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

... well, we will not discuss optimization, yet!

When an evaluator / interpreter is build all the required modules are used and nothing more:

```
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 3: Other parser combinators (like Parsec) may be used instead of ParseLib, or we can work only with *pseudoconstructors*, as you see below.



```
GHCi, version 6.8.3: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
[1 of 8] Compiling MyChar           ( MyChar.hs, interpreted )
[2 of 8] Compiling MyVoid           ( MyVoid.hs, interpreted )
[3 of 8] Compiling ClassMinus       ( ClassMinus.hs, interpreted )
[4 of 8] Compiling ClassPlus        ( ClassPlus.hs, interpreted )
[5 of 8] Compiling MyNum            ( MyNum.hs, interpreted )
[6 of 8] Compiling MyPlusNum        ( MyPlusNum.hs, interpreted )
[7 of 8] Compiling MyMinusNum       ( MyMinusNum.hs, interpreted )
[8 of 8] Compiling ParserSumaCifre  ( G:/_My2/New Haskell Source File.hs, interp
reted )
Ok, modules loaded: ParserSumaCifre, MyNum, ClassPlus, ClassMinus, MyPlusNum, My
MinusNum, MyVoid, MyChar.
*ParserSumaCifre> (plus (num 1000)(num 1))::[Int]
[1001]
*ParserSumaCifre>
```

Figure 2. Pseudoconstructors in action

3.9. Optimizing a module using monad's laws.

An optimized module may look like the next one, where both sources are shown, the old one being commented.

```
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 one of the monad's law  =>

----New implementation of the pseudoconstructor
char ::Char -> [Char]
char x = [x]
```

4. Evaluation of our solution

Our evaluation seeks to prove four hypotheses:
(1) that the increment of the RAM space used is approx. 5% or less when adding modularity on this way
(2) that missing syntax trees have no important impact in system design; and finally
(3) optimizing the code using the monad rules is a good improvement

4.1 Hardware and Software Configuration

The programs was tested using the Hugs 2002 interpreter included in The Mandrake 10.0 Linux distribution. The OS was upgraded to the new Mandriva 2007 Spring Free Edition running on a 3.4 GHz Pentium D. The smallest usable configuration seems to be an IBM dual processor (2x133Mhz) with 80MB Ram and a 4GB SCSI HDD running Mandrake Linux 8.2.

4.2 Experiments and Results

We have ran six experiments using two different kind of terms:
(1) we had measured the spaced used by the evaluation of a simple expression (no paranthesis, 10 numbers) using a classic evaluator *with trees but no lists* .
(2) we had measured the spaced used by the evaluation of a simple expression (no paranthesis, 10 numbers) using a classic modified evaluator with trees and *numbers represented by lists,* in order to determine the overloading introduced by the *lists.*
(3) we had measured the spaced used by the evaluation of a simple expression (no paranthesis, 10 numbers) using a our new modular evaluator *with trees and the list monad*, in order to see the overloading introduced by *the monad structure* .
(4) we had measured the spaced used by the evaluation of a simple expression (more paranthesis, 10 numbers) using a classic evaluator *with trees but no lists* .
(5) we had measured the spaced used by the evaluation of a simple expression (more paranthesis, 10 numbers) using a classic modified evaluator with trees and *numbers represented by lists,* in order to see the overloading introduced by the *lists.*
(6) we had measured the spaced used by the evaluation of a simple expression (more paranthesis, 10 numbers) using our new modular evaluator *with trees and the list monad,* in order to see the overloading introduced by the monad structure. The three last tests was repeated.

The space was computed by Hugs, using the ":set +s" command. All sources had included the same modular parser written using ParseLib, in order to evaluate the impact of our modular evaluator on a complete (minimal) system. The source of the modular system was optimized using the monad rules, in order to reduce the consumed space. We have just described out evaluation setup; now, let's talk about our results.

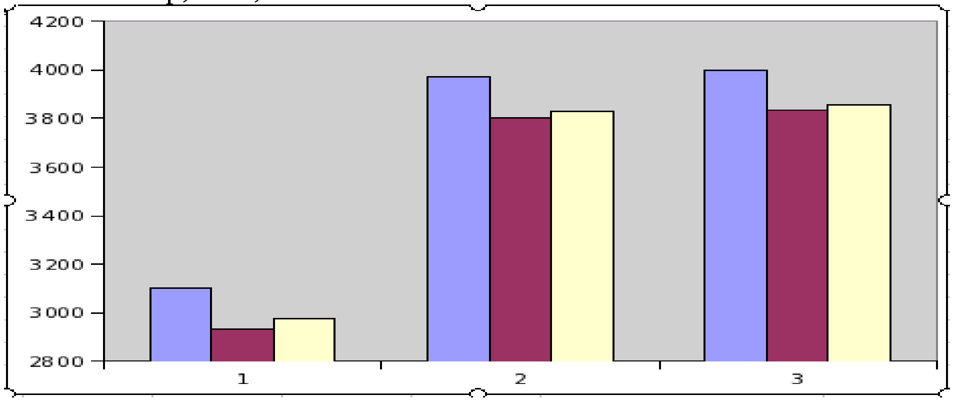


Figure 3: Three solutions was compared: Standard evaluator, Modified standard evaluator and The New modular monadic evaluator
Cyclamen = Standard evaluator: Parser , Trees, Integer
Yellow = Modified std. evaluator: Parser, Trees, [Integer], Lists to see how much overload is got by lists
Magenta = New monadic evaluator: Parser, no Trees, Modularity, [Integer], The List Monad
(all three had ran on the same P4 3.40 GHz Intel PC).

5. Conclusions

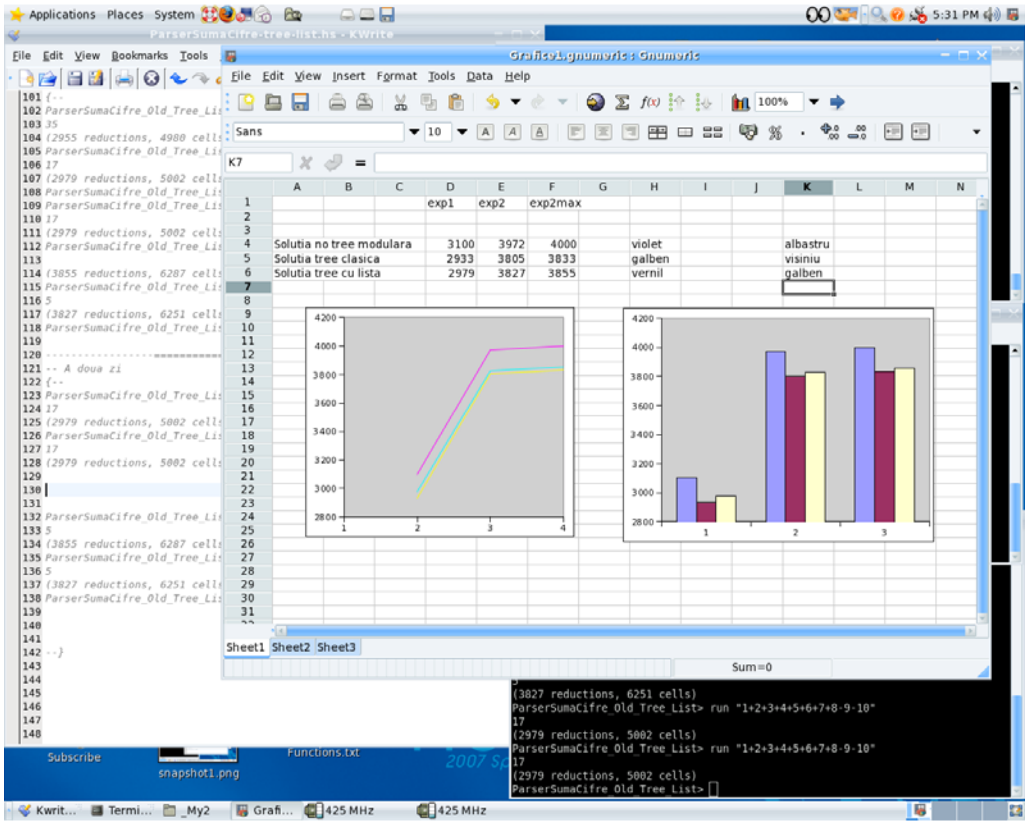
Space consumed adding lists and modularization: Adding lists to a standard evaluator (i.e. storing values as lists with only one element instead of elements) increases space with aprox 4% (considering a whole system, including the parser). Adding modularity increases space again with aprox 2%. Adding both we get a value a little big than 5% but far less than 10%. We consider that increasing the memory even with 10% is still a good price for modularity of a monadic interpreter or evaluator. So, such kind of modularity is affordable and we are able to build evaluators, interpreters and even compilers by simply including the requested (reusable) modules in the main program. In fact the modularity itself overloads the space with aprox 2%.

Remark 4: The lazy evaluation system based in fact on Hugs semantics caused sometimes dual experimental results, due to the different ways of lazy evaluation. That is why we have to evaluate the second expression two times. The biggest values are shown as Experiment 3:

	Exp.1	Exp.2	Exp. 3		Colour
Modular Monadic Ev.	3100	3972	4000		violet
Standard Evaluator	2933	3805	3833		galben
Modified Std. Evaluator	2979	3827	3855		vernil

Figure 4-5: The effective values of space used in our first experiments, (captured from Gnumeric).

Relative differences between solutions are less than 6%. The first line is the relative difference between the New Modular Monadic Evaluator and The Standard Tree Evaluator. The main part of the difference is introduced by the lists nor by the modularization (see the second line) – aprox 4% of it.



```
{--
ParserSumaCifre_Old_Tree_Lis
35
(2955 reductions, 4980 cells
ParserSumaCifre_Old_Tree_Lis
17
(2979 reductions, 5002 cells
ParserSumaCifre_Old_Tree_Lis
ParserSumaCifre_Old_Tree_Lis
17
(2979 reductions, 5002 cells
ParserSumaCifre_Old_Tree_Lis

(3855 reductions, 6287 cells
ParserSumaCifre_Old_Tree_Lis
5
(3827 reductions, 6251 cells
ParserSumaCifre_Old_Tree_Lis

-------------------==========
-- A doua zi
{--
ParserSumaCifre_Old_Tree_Lis
17
(2979 reductions, 5002 cells
ParserSumaCifre_Old_Tree_Lis
17
(2979 reductions, 5002 cells


ParserSumaCifre_Old_Tree_Lis
5
(3855 reductions, 6287 cells
ParserSumaCifre_Old_Tree_Lis
5
(3827 reductions, 6251 cells
ParserSumaCifre_Old_Tree_Lis



--}
```

	A	B	C	D	E	F	G	H	I	J	K
1				exp1	exp2	exp2max					
2											
3											
4	Solutia no tree modulara			3100	3972	4000		violet			albastru
5	Solutia tree clasica			2933	3805	3833		galben			visiniu
6	Solutia tree cu lista			2979	3827	3855		vernil			galben

```
(3827 reductions, 6251 cells)
ParserSumaCifre_Old_Tree_List> run "1+2+3+4+5+6+7+8-9-10"
17
(2979 reductions, 5002 cells)
ParserSumaCifre_Old_Tree_List> run "1+2+3+4+5+6+7+8-9-10"
17
(2979 reductions, 5002 cells)
ParserSumaCifre_Old_Tree_List> 
```

On the other side, *pseudoconstructors* over monadic actions – introduced here – had proved to be a good tool for the modularization of interpreters, compilers or evaluators. Of course, modular parsers have to produce values expressed by *pseudoconstructors* and not by usual data constructors.

The syntax trees proved harmful in terms of modularity.

6. Future and In-Progress Work

Replacing the Parser Combinators from ParseLib with a modern library as Parsec or one of his successors leads to a better control and report of the syntax errors made by the parser. Replacing the list monad with an State and IO monad can be a simple way of building modular systems like web-pages generators, and, therefore, we will be able to dynamically serve clients with pages produced by a modular, easy updated system. Replacing the list monad with a more complex monad, having IO, state, errors, context and so, is a way of building modular imperative (local, national) programming languages as Rodin. [Dan Popa, (2008)] In this Project, the *pseudoconstructors* are also used as a replacement of normal data constructors used by the data declarations of The Haskell Language.

There is a possible relation between our research and The Expression Problem, raised by [Wadler (1998)]. But they are different problems with different solutions. The relation between them, anyway, should be investigated.

Also, a parallel between our work and [Swierstra, W. (2008)] should be made, even there are different ideas. (Functors versus Monadic Pseudoconstructors).

References

[1] Leijen D.; Meijer E. ; (2001) *Parsec: A practical parser library,* Electronics Notes in Theoretical Computer Science 41, No. 1

[2] Espinosa D. ;(1995) *Semantic Lego*, Ph.D. thesis, Columbia University,

[3] Leijen D.;(2001) *Parsec a fast combinator parser library,* Univ. of Utrecht, Dept. Of Computer Science, Utrecht, The Nederlands,

[4] Leijen D., Meijer E., (2001) *Parsec: Direct Style Monadic Parser Combinators For The Real World*, DRAFT, Oct.4,

[5] Hudak,P; Hughes,J; Peyton Jones S, Wadler, P, (2007) *A History of Haskell: Being Lazy With Class* , Third ACM Sigplan History of Programming Languages Conference (HOPL – III) San Diego, CA, April 16, 2007

[6] Hutton G., Meijer E., (1998) *ParseLib - A library of monadic parser combinators*, Included in Mandriva Linux 10.0

[7] Odersky M, Zenger M (2005) – *Scalable Component Abstractions*, OOPSLA'05, Oct. 16-20, San Diego, California, USA, ACM

[8] Peyton Jones S; Hughes J, (1999) *Report on the programming language Haskell 98, a non strict purely functional programming language,*

Technical report, www.Haskell.org/onlinereport
[9] Peyton Jones S. (editor) (2002) *Haskell 98 Language and Libraries, The Revised Report*, Cambridge , www.haskell.org/definition/haskell98-report.pdf
[10] Sheard T.; Benaissa Z.; Pasalic E. (1999) *DSL Implementations Using Staging And Monads*, Proceedings of DSL'99: The 2nd Conference on Domain-Specific Languages, Austin, Texas, USA, October 3–6,. p 81-94 www.usenix.org/events/dsl99/full_papers/sheard/sheard.pdf
[11] Popa D., (2007) *Introducere in Haskell 98 prin exemple*, (*Introduction to Haskell by Examples*) Bacau, EduSoft,
[12] Popa, D. (2008) *Rodin* Language Website – in prepairing http://www.haskell.org/haskellwiki/Rodin
[13] Monad laws http://www.haskell.org/haskellwiki/Monad_laws
[14] Tang. A; (2005) *PUGS, Bootstrapping Perl 6 with Haskell*, ACM Haskell, sep 30, 2005, Talin, Estonia
[15] Wadler. P (1992) – *Monads for functional programming*, Broy Manfred (ed.) Proc. Marktoberdorf Summer School on program design calculi, Springer Verlag,
[16] Wadler. P (1992) – *The essence of functional programming*, The 19'th Symposium on Principles of Programming Languages (Albuquerque), New Mexico, ACM ,
[17] Wadler. P (1992) – *Comprehending Monads*. Mathematical Structures in Computer Science, 2,
[18] Wadler P. (1995) *How to declare an imperative*, International Logic Programming Symposium, MIT Press,
[19] Wadler P. (1998) The Expression Problem. Accessible at: http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt
[20] Zenger M. (2004) *Programming Language Abstractions for Extensible Software Components – Doctoral Thesis*, EPFL – Switzerland, EPFL Lausanne http://zenger.org/papers/thesis.pdf
[21] Swierstra, W. (2008), *Data types `a la carte*, In JFP 18(4), Cambridge University Press, pp 423-436.
[22] Extra readings: interpreters evaluators and virtual machines; the list monad

University of Bacau,
Bacau, 600114, Romania, danvpopa@ub.ro
Ro/Haskell Group, popavdan@yahoo.com