ARTICLE'S TITLE Adaptable Software – Modular extensible monadic evaluator and typechecker based on pseudoconstructors

Dan Popa
[*] "Vasile Alecsandri" University of Bacau, Romania

Abstract. This paper is investigating the use of pseudoconstructors (a monadic data structure which acts as an itself evaluator) to be simultaneously used for evaluation of terms and also for typechecking. This can lead us to the conclusion that we have a new software technique for modular DSLs building.

AMS Subject Classification (2000): 68Q45,68Q55,18C50
Key words and phrases: pseudoconstructor(s), modular monadic typechecker in Haskell
Keywords: pseudoconstructor(s), modular monadic typechecker in Haskell

1. Introduction

Modular building of domain small languages (DSLs) had attracted many scientists from decades. We may quote from papers like [6], [15] , including papers by Hutton and Meijer [3] [4],[5] to name only some authors. On the other side, the Haskell language (see Haskell Report[13]), a language widely used as a DSL for language building, also widely used for monadic computing (i.e. programming according to the monadic patterns) is providing the concept of modules [2] , [14] .

The first problem, when dealing with papers concerned modular DSL, we had noticed the amazing fact that the solutions was not modular (in sense of Haskell modules). We had also noticed that some advanced parser combinators [3],[4],[5] , and some from [9] was not so modular as we think they are, because they are producing trees which are declared using a non-modular data declaration [14] pp 43-45 (algebraic data declarations). The problem is closely related to Wadler's Expression Problem. The problem was solved by us in this – paper - [10] using *pseudoconstructors over monadic values* as defined by [8].

Usually, the ASTs (abstract syntax trees) was not modular – because Haskell did not allow us to spread a tree *data* declaration in many Haskell modules. The *pseudoconstructors over monadic values* (i.e. monadic structures which are simultaneously data structures and self evaluated modular interpreters, see [8]) had offered the solution and now the trees can be modularly defined even by spreading the declarations of pseudoconstructors (sse [8]) over the whole set of modules included by a big project.

Modular monadic semantics [7], [13] (in ML) when was (re)implemented using Haskell also failed to be modular in the sense of Haskell modules (and [14]), due to the data declarations, too. This leads us to a whole language implementation , partially introduced by a Ph.D Thesis [10], [11] based on modular monadic itself evaluators , where itself-evaluators are carrying a modular monadic entry pointless semantic (i.e a semantic without a specific *interpreter* function). This solution was used by the Rodin Project [12].

Because the demo-language Rodin had not included a typical typechecker (the modular monadic semantics isolates types very well as monadic values indexed by types, and also because Rodin was a monotype language with some type additions - ex. Strings to be printed) we had concentrated now on the task of building typecheckers and evaluators based on pseudoconstructors from [8], but in the same time.
The problem is necessary to be solved: when sticking to the idea of entry-pointless interpreters and evaluators or typecheckers, apparently the pseudoconstructors from [8] have only one semantic. No two different functions: *interpreter* and *typechecker* seems to mean – at first sight – no two different interpretations possibles. The problem is to put the same modular monadic structure to do bothe (or more than one) things. Notice, in this context of discussion, the code of a pseudoconstructor implementing an itself-evaluator: it did not have any *interpret* or *typecheck* function defined.

Let's examine it. Basically, such a module looks like here:

```
<structure>  arg1 …  arg n  =
  do  {   v1 <- arg1 ;
          …
          vi <- argi ;
          …
          vn <- argn ;
          return     f (v1 … vn) }
```

Also notice: in an e-mail from Simon Peyton Jones, such modular piece of software was considered to be carefully studied, being suspected of not being so usefull in some cases (for example in code optimization) but this task is still an open problem for us.

¶Empty space [Times New Roman 11p, single]

2.The as-instance implementation

The Haskell standard language is implementing the concept of type class and instance of a class. Various instances of the same class can implement the same interface using very different semantic functions. We are introducing the idea of implementing both the modular monadic evaluator and the typechecker as instances of the same type class.

Speaking of type checker, our starting point is gathered from [1] but a large set of semantics and books concerning semantics or lambda calculus may also be used as starting point.

On the other side, the evaluator is inspired by those used by the Rodin Project [12] and the chap. 9 from [11] – Ph.D Thesis.

¶Empty space [Times New Roman 11p, single]

3.Types and values

For typechecking we had used the following type, representing various results which can be obtained by the process of typechecking a term built by (as seen in [8]) pseudoconstructors. To simplify, only on simple type, MyInt was used, but more can be similarly added:

```
data Type  =      MyInt
                 | TypeError
                 | Pair  Type Type
                 deriving (Show, Eq)
```

Note that this is not modular, being an usual data declaration so it should be defined from the beginning.

¶Empty space [Times New Roman 11p, single]

4.Values as support for the evaluation

Inspired by [15] we have used the folowing type, as implementation of the values provided by evaluation of the terms. The type below is different than those used by Prof. Wadler, but the idea behind it's use in computations is the same:

```
data  Value =      Wrong
                 |  Num Int
                 |  PairV Value Value
```

A custom instance of the Show class is used to print such results of the process of computing the values of the terms built by pseudoconstructors , from [8]. We discovered that Wadler's evaluation mechanism works fine on terms build by pseudoconstructors. Back to the instance of Show, we had defined:

```
instance Show Value where
  show Wrong      = " Wrong value"
  show (Num I)    = show i
  show (PairV x y)= show (x,y)
```

Remark: The paired values designed as (PairV x y) are printed in a common manner, as a t-uple where t=2, using round brackets, as in the Haskell language itself.

Also, in order to compare values, which is a normal operation whenusing for example, an *if*, the above values was also made an instance of the Eq class:

```
instance Eq Value where
  Wrong == Wrong               =  True
  Num a == Num b               =  a == b
  PairV a1 a2 == PairV b1 b2 =  a1==b1 && a2 == b2
  _ == _         = False
```

This will allow, for example, to check if the value of the first sub-term of an if expression is providing a value which is equal with 1 - as it is conventionaly used by C-like languages.

5.The "Computing with errors" Monad

In the above quoted paper, [15] a monad is used to capture the essence of computing in the presence of errors , and we will use it in our example, as model of computation for the *values* of the terms. As a remark, nowadays The Haskell Platform is including the Maybe monad which can also be used to model the computations with errors. Using the Maybe Monad here, is also a possibility. The name of the type is M, and it has a single type parameter.

```
data M a = Success a | Error String

instance Monad M where
  return a          = Success a
  (Success a) >>= k = k a
  (Error s)   >>=k  = Error s
error s = Error s
```

The meaning is usual: a finally computed value *a*, injected into the monad by *return* becomes a *Success a*. This is the result of a successful computation.

The *bind* (>>==) between *Success a* and a second computation k is defined as k applied to a. So, we can compute the result by simply extracting a from (Success a)

and apply the (notice: monadic action) k. On the other side, an error produced by a computation should propagate and finally produce an error, as result of the whole computation.
The result of computations is also made show-able, using also a custom instance of the Show class:

```
instance (Show a)  => Show (M a) where
  show (Success a) = "Success: " ++ show a
  show (Error s)     = "Error :" ++ s
```

6. The dual semantics

Modular monadic interpretation can be implemented as a type class, which can also be instantiated as a typechecker. The class we had defined, (called "interpretare") is including the following functions: *if0*, *operator*, *constant*, *prj*, *pair*, *variable*. Here it is, and it is based on the idea that semantics should not necesarry be a function from Values to Values than is usual, or a function from Values to M Values – as the functions are implemented in [15]. We considered that modular monadic semantics for the terms build by pseudoconstructors ([8]) should usually be a sort of function from *m values* to *m values* semantics – or usually a function from *m types* to *m types* (next column):

```
class Interpretare  typ where
  if0:: (Monad m) =>  m typ-> m typ-> m typ-> m typ
  operator :: (Monad m)
    => (Int -> Int -> Int) -> m typ-> m typ-> m typ
  constant :: (Monad m) => Int -> m typ
  prj :: (Num t, Monad m) => t -> m typ-> m typ
  pair :: (Monad m) => m typ-> m typ-> m typ
  variable :: (Monad m) => String -> m typ
```

Of course, in some special cases, like the simplest terms packaging other kind of data, the signature will not include only *m type*-s. If needed, the interface of the class can be extended. Here, we have just put a minimal collection. The word "typ" can be read as "type" which is a keyword in Haskell.

7.The typechecker

Starting from the class above, the implementation of the typechecker is strightforward. As a general case, all the subterms of a term, are itself-evaluated and the results are used by a fucntion (called f,here) to produce the final result, in this instance the computed type of the complete term.

---------- Code of the type-checker----------

```
instance Interpretare Type where
  if0 e1 e2 e3   = do { tau1  <- e1 ;
                        tau2  <- e2 ;
                        tau3  <- e3 ;
                        return ( f tau1 tau2 tau3 )}
                          where
                              f tau1 tau2 tau3 =      if tau2==tau3 && tau1 == MyInt
                                                      then tau2 else TypeError
                              f _ _ _ = TypeError
  constant i = return MyInt
  variable s = return MyInt
  operator p e1 e2 = do {    MyInt <- e1 ;
                             MyInt <- e2 ;
                             return MyInt }
  prj 1 e = do { (Pair tau1 tau2) <- e ; return tau1 }
  prj 2 e = do { (Pair tau1  tau2) <- e ;    return tau2 }
  pair e1 e2 = do {          tau1 <- e1;
                             tau2 <- e2;
```

As a result of this definition we are able to evaluate the types of terms by simply typing the terms in the interactive mode of hugs or ghci, and specifying That we are interested in types. In the folowings line, the list monad is used as support for the do-notations but other monads can also be used (try the Maybe Monad – for example).

8.The evaluator

A standard evaluator, in this case using a dummy environment – but it can be replaced by a real one - is also writable as an instance of the previous class. Notice (by missing) the absence of interpreter or typechecker nominal functions.

```
instance Interpretare Value where
 constant i = return (Num i)
 variable v = return (g v)            -- dummy lookup function
              where
                g "x" = (Num 1)
                g "y" = (Num 2)
 if0 e1 e2 e3= do{v0  <- (constant 1);
                  v1  <- e1 ;
                  v2  <- e2 ;
                  v3  <- e3 ;
                  return ( f v0 v1 v2 v3 )}
                  where
                    f v0 v1 v2 v3 = if v1==v0
                                    then v2  else v3
 prj 1 e = do {(PairV i1 i2) <- e ; return i1 }
 prj 2 e = do {     (PairV i1 i2) <- e ; return i2 }
 pair e1 e2 = do {          i1 <- e1;
                            i2 <- e2;
                            return (PairV i1 i2)}
 operator p e1 e2 = do {a <- e1 ;
                        b <- e2 ;
                        return (lift p  a  b );
                        }
                    where lift op (Num x)  (Num y) =
                                    Num $ op x y
```

9.The results

As a result of this dual definitions, the terms based on pseudoconstructors (like in [8]) can be easily typechecked and evaluated, simply writing them in interactive mode or using them in programs. There is a supplementary need: due to the polymorphism involved. to specify the type of the terms is a must. Also the monad which is selected to be used with the do-notations is needed. The system is so flexible – being based on do-notation, that almost any monad can be used. In the following examples, the list monad is used, but it is just a simple possibility, which is available in the system without suplementary programming. The reader is encouraged to use the Maybe monad, too.

10.Examples

Let's evaluate and type check some terms, to see the dual modular monadic entry -pointless semantic at work for terms based on pseudoconstructors. For the following examples, we had asked a Hugs system (but also a ghc or ghci can be used) to compute the value (packed a s a list) and the type (also packed as a list) of some terms which are built based on pseudoconstructors. Also notice how the terms looks like, and how simple can they be written comparing with some other solutions.

dan@device:/media/disk/2011-HOAS-7feb$ hugs interpretare2.hs

```
Main> (constant 1) :: [Type]
[MyInt]
Main> (constant 1) :: [Value]
[1]

Main> (variable "x") :: [Type]
[MyInt]
Main> (variable "x") :: [Value]
[1]

Main> (operator (+) (constant 1000)
(constant 1)) :: [Type]
[MyInt]
Main> (operator (+) (constant 1000)
(constant 1)) :: [Value]
[1001]
Main> (operator (+) (constant 1000)
(variable "x")) :: [Value]
[1001]
Main> (operator (+) (constant 1000)
(variable "x")) :: [Type]
[MyInt]

Main> (pair (constant 1000) (variable
"x")) :: [Type]
[Pair MyInt MyInt]
Main> (pair (constant 1000) (variable
"x")) :: [Value]
[(1000,1)]
Main> prj 1 (pair (constant 1000)
(variable "x")) :: [Value]
[1000]
```

```
Main> (if0 (constant 1) (variable "x")
(variable "y")) ::[Type]
[MyInt]
```

The reader is invited to use ghc instead of hugs, to try other terms and even give a complete formal proof that this implementation can type and evaluate all of the terms built by this set of pseudoconstructors. Also he or she should notice: the monad of Successful and Errors computations is not necessary needed by this kind of solutions, but can be included as a part of it.

References

[1] Louis Julien Guillemette, Stefan Monier, *Type Safe Code Transformation in Haskell*, Univ. of Montreal, Electronic Notes in Theoretical Computer Science, Elsevier Science, 174 (2007) 23-39

[2] Hudak Paul, Peterson John, Fasel Joseph – A Gentle Introduction to Haskell 98, Yale University, Los Alamos Laboratory, 1999

[3] Hutton, Graham; Meijer, Errik; Monadic parsing in Haskell; Journal of Functional Programming 1 (1): 1-000, 1993, Cambridge University Press

[4] Hutton, Graham; Meijer, Errik; Monadic Parser Combinators - "Technical report NOTTCS-TR-96-4 Dept. Comp Sci Univ,. Nottingham – 1996
http://www.cs.nott.ac.uk/Department/Staff/gmh/monparsing.ps

[5] Hutton, Graham; Meijer, Errik; Monadic parsing in Haskell; Journal of Functional Programming 8(4): 437-444,Jully 1998
http://www.cs.nott.ac.uk/Department/Staff/gmh/bib.html#pearl

[6] Ivanovik, Mirjana; Kuncak, Viktor; Modular Language Speciffications in Haskell; Institute of Mathematics, University of Novi Sad, Yugoslavia, 2000

[7] Liang S , Hudak P., Jones M. Monade transformers and modular interpreters, POPL'95. ACM Press,1995

[8] Pseudoconstructors over monadic values
http://www.haskell.org/haskellwiki/Pseudoconstructors_over_monadic_values

[9]Popa Dan; *Practica Interpretarii Monadice*, Matrix Rom Publishing House, 2008

[10] Dan Popa, *Modular evaluation and interpreters using monads and type classes in Haskell* , Studii si Cercetări Ştiinţifice, Seria Matematică, Univ. Bacău, (18) 2008, pp pag. 233 – 248
http://www.haskell.org/wikiupload/7/7d/POPA_D.pdf

[11] Popa, Dan ; *Metode si tehnici de realizare a interpretoarelor adaptabile*, Univ. "Al.I.Cuza" Iasi, 2010

[12] The Rodin Project
http://www.haskell.org/haskellwiki/Rodin
http://www.haskell.org/haskellwiki/RodinEn

[13] Zine-el-Abidine Benaisa, Emir Pasalic; DSL Implementation using staging and monads, Pacific Software Research Center, Proceedings of DSL'99: The 2nd Conference on Domain-Specific Languages , Austin, Texas, USA, October 3–6, 1999
http://www.usenix.org/events/dsl99/full_papers/sheard/sheard.pdf

[14] Peyton Jones, Simon (editor); Haskell 98 Language and Libraries The Revised Report, Cambridge University Press (May 5, 2003)
http://haskell.org/definition/haskell98-report.pdf

[15] Wadler, Philip; *The essence of functional programming*, The 19th Symposium on Principles of Programming Languages,ACM, Albuquerque, New Mexico, 1992