Adaptable Software - Modular Extensible Monadic Entry-pointless Type Checker in Haskell

by Dan Popa
Ro/Haskell Group, Univ. “V.Alecsandri”, Bacau

Abstract: The goal of this paper is to investigate the use of a software technology – the pseudoconstructors over monadic values (structures capable of simultaneously representing both syntax and semantics) – to build modular entry-pointless type checkers using the VHLL Haskell. A template used by almost all the modules of the modular monadic entry-pointless type checker is also revealed.

The actual situation

The *pseudoconstructors* over monadic values was used by The Rodin Project [Rodin] in order to build modular entry-pointless monadic interpreters and evaluators . The notion comes from some papers like [Popa 2008] and it also have been introduced by a dedicated page of The Haskell Community's Website [Pseudoconstructors] and included – as a way to promote the concept – in the English Version of Wikipedia.
What is known: the pseudoconstructors (a sort of functions used to replace the usual data declarations in Haskell and provide modularity) offered a way of solving Wadlers's Expression Problem. The fact was checked and confirmed py Prof. Philip Wadler, by e-mail. Some questions concerning their properties and applications was raised by Prof. Simon Peyton Jones, related to code generation and optimization in a private mail, confirming the existence of such branch in the domain of functional interpretation and monadic semantics . I wish to thank you, all, for the time you had spend checking my papers, notices and e-mails.
As a result, in an actually defended Ph.D thesis [Popa – 2010] a supplementary chapter (Chap 9, pp 140 – 155) was added, also concerning pseudoconstructors over monadic values (The small modules of the interpreter are called, self-evaluators, there) . Because further studies appears to be possible and due to the fact that Rodin is in fact a small mono-type modular language, the abilities of pseudoconstructors to be used as software components for adaptable modular monadic type-checkers have interested us.
Some points should be remind, because, in fact, the advantages of the pseudoconstructors are:
1. They are *simultaneously* syntactic structures of the terms and modular adaptable monadic representations of term's semantics.
2. Due to the missing of (we say “unimportant”) *interpret* function – the common entry point of any interpreter like those presented in [SHJ 1995] the pseudoconstructors can be distributed across various modules of the project, so providing modularity which is crucial,for example, in natural language processing. [Popa - 2008] [Popa – 2010].
3. Examining the do-notation as it used (see also below, in the case study or in the template area) , the monad used by semantic implementation is not fixed, remaining variable, and the programmers may use any monad to develop the system (including one produced by composition of Monad Transformers). The idea of monad replacement - in order to produce various semantics – was implemented in some papers by prof. P.Wadler, important being, for us [Wadler]
4. Pseudoconstructors may be used as implementation for modular trees (which are not declarable in Haskell using *data* declarations) .The fact that *data* declarations in Haskell are not modular can be found reading any Haskell manual, including the Haskell Report. [Report] As a consequence, Haskell programmers are forced to declare the whole syntax tree in one place, which is not modular.
So, building a modular monadic typechecker (suitable for modular syntax typechecking) in Haskell is in fact a real challenge.

Starting point

The type-checker is in fact a recursively defined semantic function of terms. We had started from one semantic function which can be found in [Guillemette & Monier 2007] but other books and papers can also serve as a starting point. Because the semantic itself becomes modular, some (coherent) subsets of semantic rules can be good starting points too.

The template

We had succeeded in providing a standard Haskell form for such a type checking semantics, which can be used. Here it is (this is not Haskell code):

```
structure :: (Monad m) => m Type -> …  -> m Type-> m Type
structure e1 e2 … en   = do { t1   <- e1 ;
                              t2   <- e2 ;
                              …
                              tn   <- en ;
                              return ( proceed t1 t2  … tn ) }
                               where
                                 proceed t1 t2  … tn | p1 && p2 … && pn   = f t1 t2 … tn
                                 proceed t1 t2  … tn | p1' && p2' … && pn' = f' t1 t2 … tn
                                 proceed _ _ _ … _                        = TypeError
```

Some notations should be explained:

structure - is the name of the syntactic structure, it may be simple or complex. Examples: constant, variable, operator, if0, lam, app, pair, prj, etc.

m - is the type variable left free for the use of the required monad.

e1,..,en - substructures of the syntactic structure

ti <- ei - entry-pointless monadic evaluators. Notice the missing of the *interp* (or *interpret*) function, which usually looks like:

type <- interpret ei context

proceed - the semantic rule provided for typechecking

p1,...,pn,
p1',...,pn' - predicates expressed in terms of types

f, f' … - auxiliary functions, sometimesneeded, sometimes not needed

This design was carefully chosen, many attempts was made before. The reader may eventually want to reimplement such semantics in order to struggle against some Haskell's limitation. A part of this problems are included in the case study below:

The case study

The semantic taken from [Guillemette & Monier 2007] was the subject of our experiment and building procedure, following (as much as possible) according the above described template. Because the is a recursively defined set of terms, the semantic is also built starting from some simple cases, coresponding with the simplest terms. But, first of all, we have to define the set of type values:

The type values

Actually, the values resulted from typechecking are also forming a union type , which is declared using an usual *data* declaration.

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

which means we have had in mind the following situations:

1. Usual integer values.
2. Incorrect expressions. Note that an explanations, as a String, can also be needed and added.
3. A functional type, used in process of computing the type of lambda abstractions.
4. A product of types, used for pairs of expressions.
5. A special, auxiliary type used to simplify the implementation of the typescheme associated with binary operators.

The type values declared above will be comparable using the "==" operator, which is a need of the type-checking procedure. And all this values are declared *"showable"* ; this will help the debugging, because we wish to use an interactive Haskell system as Hugs or GHCi to show the results of evaluations. Generally speaking, fort the programmer, is a good choice to create showable types because any results of the functions returning such types will be printable, even without a special *printer* procedure. Of course, a commercial implementation may have a different set of requirements and may add a custom made printing - i.e. *show* – function for such types.

Constant's type-checking

The rules required by constants was first of all implemented as:

```
constant :: (Monad m) => Int -> m Type
constant j = return MyInt
```

Also, variants of those rules can be considered. The above one leads to the following evaluation, for example, using the list monad as support for the do-notation:

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

Basicaly, the rule is implementing the fact that any constant j will produce the MyInt type, if j belongs to Int.

Variable's type-checking

Both for small languages like Rodin [Rodin] having a single simple type, Int or for typed lambada calculus systems we may want to use a rule like:

```
variable :: (Monad m) => t -> m Type
variable _ = return MyInt
```

This is allowing both kind of definitions (variable 'x') and (variable “x”) to work fine. Of course, the next step will be to use a complete environment and a lookup function. But to simplify the example, this simple above definition is enough. So, we can evaluate:

```
Main>  (variable 'x') ::[Type]
[MyInt]
```

Remark, a formula like:

```
  variable :: (Monad m) => String -> m Type
  variable s = return MyInt
```

can only work for variable having identifiers expressed as elements of the String type. The drawback of the previous solution is tha fact that it can evaluate even strange sequences like: (variable (constant 1)). That is why t is replaced by String.

Composed structures: the *if*

Considering a sort of if accepting an Int instead of a bool (using the classic C language convention: 1 means True, 0 and others means False). This if, called if0 is also used by [Guillemette & Monier 2007]. It's implementation using monadic type is:

```
if0:: (Monad m) => m Type -> m Type -> m Type -> m Type
if0 e1 e2 e3   = do { tcond <- e1 ;
                      tau   <- e2 ;
                      tau'  <- e3 ;
                      return ( proceed tcond tau tau' ) }
                         where
                              proceed MyInt t1 t2 | t1 == t2 = t1
                              proceed _ _ _ = TypeError
```

The “proceed” function was a bit modified, starting from the template. According to the template, it may be :

```
                    proceed t1 t2 t3 | t2 == t3 && t1==MyInt  = t2
                    proceed _ _ _  = TypeError
```

But both are equivalent, being evaluated to TypeError excepting the case when t2 == t3 and t1==MyInt. So, if it is written as below, it will match the template:

```
if0:: (Monad m) => m Type -> m Type -> m Type -> m Type
if0 e1 e2 e3   = do { tcond <- e1 ;
                      tau   <- e2 ;
                      tau'  <- e3 ;
                      return ( proceed tcond tau tau' ) }
                         where
                              proceed t1 t2 t3 | t2 == t3 && t1==MyInt  = t2
                              proceed _ _ _  = TypeError
```

Using any of this definitions we can evaluate:

```
Main> (if0 (plus) (constant 1) (constant 2)):: [Type]
[TypeError]

Main> (if0 (variable 'x') (constant 1) (constant 2)) :: [Type]
[MyInt]
```

The binary operators

Various binary operations may require a specific treatment (at least because usual lambda calculus functions are single argument functions). The rule is implemented as:

```
operator :: (Monad m) => m Type -> m Type -> m Type -> m Type
operator p e1 e2 = do { tau1 <- e1 ;
                        tau2 <- e2 ;
                        tbop <- p;
                        return (proceed tau1 tau2 tbop) }
                          where
                            proceed tau1 tau2 (BOperator t1 t2 trez) | tau1 == t1 && tau2 == t2 = trez
                            proceed _ _ _ =  TypeError
```

This is also becomes similar with the template, considering the proceed function being:

```
   proceed tau1 tau2 tbop | tau1 == f1( tbop) &&  tau2 == f2(tbop)  = f3 (tbop)
   proceed _ _ _ =  TypeError
```

f1,f2,f3 being the projections used to decompose the (Boperator a b c) structure on it's components. Also some operators should added , like:

```
plus :: (Monad m) => m Type
plus = return (BOperator MyInt MyInt MyInt)
```

As part of the research we also wanted to have the operator's type as a function from monadic values to monadic values (m Type -> m Type) but the Haskell language did not allow us to use something like:

```
plus :: (Monad m) => m Type -> m Type -> m Type
plus (return MyInt) (return MyInt) (return MyInt) = (return MyInt)
```

This is not allowed because the monad's return is not accepted as part in a pattern matching, because *return* is not a data constructor. As a consequence we have decided to use a special type for binary operators. Now we are able to evaluate something like:

```
Main> ( operator plus (constant 1) (constant 2)) :: [Type]
[MyInt]
```

Of course, ore complex terms representing expressions can be type-checked. Also note that a solution using separate operators and no “operator” rule is also possible, leading us to the possibilitry of evaluating something like:

```
Main> (plus (constant 1) (constant 2)) :: [Type]
```

[MyInt]
Because we had previously used something like this in [Popa 2008] we did not insist on this case.

Type-checking pairs

Pairs are an important part of the theory of computing, especially lambda calculus. See, for example the course [Gordon] by Prof. Mike Gordon , freely available resource, on the internet.

Pairs can be composed and decomposed. So, projections can also be neeeded to extract both parts of the pairs. The main rule will be accompanied by others.

```
pair :: (Monad m) => m Type -> m Type -> m Type
pair e1 e2 = do { tau1 <- e1;
                  tau2 <- e2;
                  return ( Pair tau1 tau2 )}
```

This simple rule used the fact that every two types can be paired, so we don't need predicates at all. But in order to match the template a “proceed” function can be defined:

```
  proceed tau1 tau2 = Pair tau1 tau2
```

or simply:

```
  proceed = Pair
```

and the rule can also be written as:

```
pair :: (Monad m) => m Type -> m Type -> m Type
pair e1 e2 = do { tau1 <- e1;
                  tau2 <- e2;
                  return ( proceed tau1 tau2 )}
                    where
                       proceed tau1 tau2 = Pair tau1 tau2
```

So, we can evaluate and check pairs:

```
Main> (pair  (constant 1) (constant 2) )::[Type]
[Pair MyInt MyInt]
```

but also more complex pairs can be checked.
To implement the projections we need two other rules:

```
prj :: (Num t, Monad m) => t -> m Type -> m Type
prj 1 e = do {  t1 <- e ;
                return (proceed t1) }
                  where
                     proceed (Pair tau1 tau2) = tau1
                     proceed _ = TypeError
prj 2 e = do {  t1 <- e ;
                return (proceed t1) }
                  where
                     proceed (Pair tau1 tau2) = tau2
                     proceed _ = TypeError
```

Using the above rules we ca check pairs:

```
Main> (prj 1 (pair (constant 1)(constant 2)))::[Type]
[MyInt]

Main> (prj 1 (constant 1))::[Type]
[TypeError]
```

Also more complex pairs can be checked.
Remark: The deviation from the template is just apparent, and is produced by the way of identifying the first and the second part of a pair, using numbers. But the template can be strictly followed using:

```
first :: (Monad m) => m Type -> m Type
first e = do {  t1 <- e ;
                return (proceed t1) }
                  where
                     proceed (Pair tau1 tau2) = tau1
                     proceed _ = TypeError

second :: (Monad m) =>  m Type -> m Type
second e = do { t1 <- e ;
                 return (proceed t1) }
                   where
                     proceed (Pair tau1 tau2) = tau2
                     proceed _ = TypeError
```

Both this rules can be inserted in the system together with the old rules, and make evaluations like this possible:

```
Main> (first (pair (constant 1) (constant 2))) :: [Type]
[MyInt]
Main> (second (pair (constant 1) (constant 2))) :: [Type]
[MyInt]
```

Type-checking abstractions and applications

Important parts of a typed lambda calculus system, abstractions and applications can and should be checked. The rules are:

```
lam :: (Monad m) => m Type -> m Type -> m Type
lam x e1 = do { tau1 <- x;
                tau2 <- e1;
                return (Arrow tau1 tau2) }
```

In theory, functions can link any type with any type and produce a new type. A special value, Arrow is used as type of the abstractions. The applications are using an other rule:

```
app :: (Monad m) => m Type -> m Type -> m Type
app e1 e2  = do { t1 <- e1   ;
                  arg1 <- e2   ;
                  return (proceed t1 arg1) }
                    where
                      proceed (Arrow tau1 tau2) arg1 | tau1 == arg1 = tau2
                      proceed _ _ = TypeError
```

Using *lam* and *app* we can check terms like this:

```
Main> (lam (variable 'x') (operator plus (variable 'x') (constant 1))) ::[Type]
[Arrow MyInt MyInt]

Main> (app (lam (variable 'x') (operator plus (variable 'x') (constant 1))) (constant 1) )::[Type]
[MyInt]

Main> (app (constant 1) (constant 2))::[Type]
[TypeError]
```

Conclusion

The use of the pseudoconstructors over monadic values as part of a modular monadic type-checker is possible. We have also identified a template matching all rules involved, excepting the simplest cases. This kind of type-checker ca be used for the implementation of adaptable modular languages. From our point of view it is an important contribution to the set of adaptable tools available for language constructions and – open problem - for natural language processing.

References

[Gordon] Gordon, Mike ; *Introduction to Functional Programming*, 1996
http://www.cl.cam.ac.uk/users/mjcg
http://www.haskell.org/wikiupload/a/a5/Notes_Functional_programming.pdf
http://www.haskell.org/wikiupload/2/2a/MikeGordon.pdf

[Guillemette & Monier 2007]
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

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

[Popa 2008]
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

[Popa 2010]
Popa, Dan ; *Metode si tehnici de realizare a interpretoarelor adaptabile*, Univ. “Al.I.cuza” Iasi, 2010
first published as
Popa Dan; *Practica Interpretarii Monadice*, Matrix Rom Publishing House, 2008

[SHJ 1995]
Sheng Liang,Paul Hudak,Mark Jones; *Monad Transformers and Modular Interpreters*, Yale University, Department of Computer Science, New Haven, Conference Record of POPL’95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, CA, January 1995.

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

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