Camila Revival: VDM++ meets Haskell

Alexandra Silva
xana@di.uminho.pt

Techn. Report DI-PURe-06.01.01
2006, January

PURE
Program Understanding and Re-engineering: Calculi and Applications
(Project POSI/ICHS/44304/2002)

Departamento de Informática da Universidade do Minho
Campus de Gualtar — Braga — Portugal

DI-PURe-06.01.01
Camila Revival: VDM++ *meets Haskell* by Alexandra Silva

Abstract

This report is a follow up of [16]. It shows how to model some of the key concepts of VDM++ in HASKELL. Classes, objects, operations and inheritance are encoded based on the recently developed OOHASKELL library.

1 Introduction

The CAMILA[1] innitiative explores how concepts from the VDM specification language and the functional programming language Haskell can be combined. This task, part of PURe project[2], is a revival of the original CAMILA system[3], initially developed in the 90's at University of Minho.

A careful review of the monadic modeling of VDM-SL features (Sets, Maps, datatype invariants and pre/post conditions, among others) can be found in [16].

Parameterized monads are used to control the switching among different modes of evaluation. The use of monads allows the integration of many features [18,17,5], just by changing the monad definition and with no need of re-writing the code. This is very useful in several situations, such as error control, and is very desirable for software maintenance [5].

VDM++ is a formal specification language intended to specify object oriented (OO) systems with parallel behaviour [6]. The use of classes and object concepts facilitates the development of object oriented formal specifications. In this report, we will analyse how to translate VDM++ code to HASKELL, in particular OO features such as classes, objects and instance variables. Parallelism issues are left for future research.

1.1 Structure of the report

Section 2 describes mutable variables in HASKELL and their use in implementing instance variables.

We describe HLIST and OOHASKELL libraries, in sections 3 and 4, that, respectively, model heterogeneous lists and OO features in Haskell.

A simple example is presented in section 5 as an illustration of the differences between OOHASKELL and the behaviour we intend to model and of how we intend to overcome them. A more complex example involving inheritance is discussed in section 6.

Conclusions and plans for future work are presented in section 7.

In appendix A, we explain how to get and use the source code which underlies the technical content of this report.

---

[1] http://wiki.di.uminho.pt/wiki/bin/view/PURe/Camila

[2] FCT under contract POSI/ICHS/44304/2002

[3] http://camila.di.uminho.pt

2 IORef

A mutable variable, ubiquitously used in imperative languages, is a very useful programming feature. In HASKELL it can be modeled by using the primitive `IORef` [8].

```
data IORef a
```

We can look at a value of type `IORef a` as a reference to a mutable cell of type `a`. Such references can be manipulated by the functions:

```
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
modifyIORef :: IORef a -> (a -> a) -> IO ()
```

which, respectively, allow to create, read and write a reference to a value `a`.
In order to better understand how we can easily use this concept we present the example which follows, which models a counter.

```
counter :: IORef Int
counter = unsafePerformIO $ newIORef 0

incCounter :: IO ()
incCounter = modifyIORef counter (+1)

inspectCounter :: IO ()
inspectCounter = do
                    c <- readIORef counter
                    putStrLn $ show c
```

The use of `unsafePerformIO :: IO a -> a` is not strictly necessary, but provides a easier way of defining the operations which access `counter`.

Using `ghci` it is possible to observe the following results:

```
*Counter> inspectCounter
0
*Counter> incCounter
*Counter> incCounter
*Counter> incCounter
*Counter> inspectCounter
3
```

3 The HLIST library

A heterogeneous collection can be regarded as a data structure that works as a repository for objects of different types and allows operations of update, iteration, lookup, etc. On a first approach we could define such a structure as:

```
type HetList = [Dynamic]
```

However, the type of each element is not precisely described. Type-level programming has been exploited by Kiselyov *et al.* [10] to model a more typeful heterogeneous list. The following declarations form the basis of the library:

```
data HNil      = HNil
data HCons e l = HCons e l

class HList l
instance HList HNil
instance HList l => HList (HCons e l)

myHList = HCons 1 (HCons True (HCons "foo" HNil))
```

The datatypes `HNil` and `HCons` represent empty and non-empty heterogeneous lists, respectively. The `HList` class, or type-level predicate, establishes a well-formedness condition on heterogeneous lists, *viz.* that they must be built from successive applications of the `HCons` constructor, terminated with `HNil`. Thus, heterogeneous lists follow the normal cons-list construction pattern on the type-level.

Now, a function over this structure becomes a new class. As an example, the function for the concatenation of two heterogeneous lists can be defined as follows:

```
class HAppend l l' l'' | l l' -> l'' where
  hAppend :: l -> l' -> l''

instance HList l => HAppend HNil l l
    where hAppend HNil l = l
instance (HList l,HAppend l l' l'')
        => HAppend (HCons x l) l' (HCons x l'')
    where hAppend (HCons x l) l' = HCons x (hAppend l l')
```

Note the use of functional dependencies in the class definition. The clause `l l' -> l''` declares that the parameters `l` and `l'` uniquely determine `l''`. This dependency

is exploited by the compiler – when `l` and `l'` are instantiated, the instantiation of `l''` is automatically inferred.

These lists, in turn, are used to model extensible polymorphic records with first-class labels and subtyping, which will be used in the OOHASKELL library presented in section 4.

Records are modeled as heterogeneous lists of pairs of labels and values.

```
myRecord
 = Record (HCons (zero,"foo") (HCons (one,True) HNil))
one = succ zero
```

All labels of a record are required to be pairwise distinct on the type level. Type-level naturals are a simple candidate. A datatype constructor `Record` is used to distinguish lists that model records from other lists.

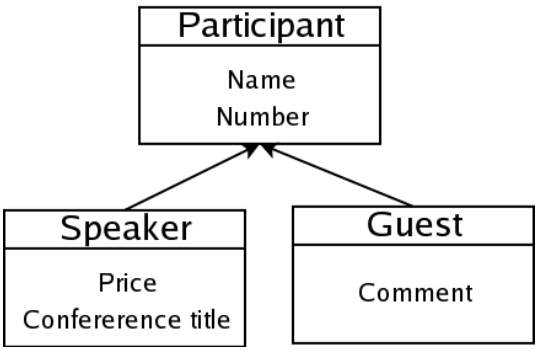
The library offers numerous operations on heterogeneous lists and records, such as append, zip, map or lookup.

4 OOHASKELL

Object oriented languages offer features such as encapsulation, mutable state, overriding, inheritance, and so on. Implementing these features in HASKELL is an issue that has been addressed for several years. There were some encodings proposed for OO idioms in HASKELL [7,14,3,9]. In this section, we will analyse the features of the recently developed library OOHASKELL (major release by September 2005). OOHASKELL is based in the extensible polymorphic records of HLIST and tries to overcome the insufficiencies of previous OO libraries in HASKELL and delivers an *amount of polymorphism and type inference that is unprecedented* when compared to other implementations [9]. The main design decisions of OOHASKELL are:

- Classes are represented as functions
- Objects are records with a component for each method
- State is maintained using mutable variables
- Methods are functions (monadic) that can access state and self

Let us analyse the following example modelling the participants in a conference:



There are three types of participant:

- Regular participants, who have name and are assigned a number;
- Speakers, who additionally to the participants information have the conference title and expenses;
- Guests, who additionally to the participants information have a comment (purpose to attend in the conference, etc)

To encode this example in OOHASKELL we start by defining unique identifiers for methods and instance variables, *e.g.*:

```
data GetName;       getName     = undefined :: GetName
data GetNumber;     getNumber   = undefined :: GetNumber
data SetName;       setName     = undefined :: SetName
data SetNumber;     setNumber   = undefined :: SetNumber
```

Concerning class `Participant`, which is modeled as a monadic function that takes constructor arguments and a *self* reference (needed for controling inheritance issues):

```
participant name_init number_init self
  = do
      name <- newIORef name_init
      number <- newIORef number_init
      returnIO $
           getName     .=. readIORef name
       .*. getNumber     .=. readIORef number
       .*. setName .=. (\newn -> writeIORef name newn)
       .*. setNumber .=. (\newnu -> writeIORef number newnu)
       .*. emptyRecord
```

The essence of inheritance is illustrated in the encoding of both `speaker` and `guest` classes. The *super-class* constructor is invoked and `self` is passed from

the sub to the super class.

```
speaker name number price conftitle self
  = do
      super <- participant name number self
      p <- newIORef price
      c <- newIORef conftitle
      returnIO $
           getPrice  .=. readIORef p
       .*. getConferenceTitle .=. readIORef c
       .*. setPrice  .=. (\newp -> writeIORef p newp)
       .*. setConferenceTitle .=. (\newc -> writeIORef c newc)
       .*. super

guest name number comment self
    = do
      super <- participant name number self
      c <- newIORef comment
      returnIO $
           getComment  .=. readIORef c
       .*. setComment  .=. (\newc -> writeIORef c newc)
       .*. super
```

Multiple inheritance, width and depth subtyping, among other OO topics are discussed in [9]. However, these issues go beyond the purpose of this report, and so they will not be dealt with in the sequel.

5 Simple Stack model

Let us start with a very simple example – the model for a stack. A stack can be represented (in a simplistic way) has a sequence of elements (in HASKELL `[Elem]`) that offers two basic operations – `push :: Elem -> ()` and `pop :: () -> Elem`.

We consider the VDM++ model of a Stack given in[13].

```
class stackObj
    types
      public Stack = seq of A ;
      public A = token ;
```

```
    instance variables
      stack : Stack := [];

    operations
      public PUSH : A ==> ()
      PUSH(a) == stack := [a] ^ stack;

      public POP : () ==> A
      POP() == def r = hd stack
               in ( stack := tl stack;
                    return r)
      pre s <> [];
end stackObj
```

In OOHaskell we directly translate the `StackObj` class to the following monadic function. (We omit the declaration of unique identifiers, cf. section 4):

```
stackObj
    = liftIO $
       do
        s <- newIORef []
        returnIO
               $ getStack .=. (liftIO $ readIORef s)
             .*. push .=. (pushop s)
             .*. pop  .=. (popop s)
             .*. emptyRecord
```

The needed auxiliary operations are as follow.

```
pushop :: (CamilaMonad m, MonadIO m) => IORef [a] -> a -> m ()
pushop s i = liftIO $ modifyIORef s (i:)

popop :: (CamilaMonad m, MonadIO m) => IORef [a] -> m a
popop s = do l <- liftIO $ readIORef s
             pre (not $ null l)
             liftIO $ modifyIORef s tail
             return (head l)
```

Note that we need all the computation to be performed inside a `CamilaMonad` which will allow different modes of evaluation using `runCamilaT` as described in [16]. Because we are using `IORef`'s, the computations are restricted to the `IO` monad. In order to *lift* the computation from this monad to a `CamilaMonad`,

in particular to `CamilaT mode m`, we define the latter as instance of `MonadIO`, class that provides the function `liftIO :: (MonadIO m) => IO a -> m a`.

```
instance MonadIO m => MonadIO (CamilaT mode m) where
  liftIO = CamilaT . liftIO
```

5.1 Hiding `liftIO`

To avoid confusing errors due to a missing `liftIO` invocation, we decided to encapsulate the `IORef` behaviour inside a generic monad.

```
class MonadRef m where
    readRef :: IORef a -> m a
    modifyRef :: IORef a -> (a -> a) -> m ()
    writeRef :: IORef a -> a -> m ()
    newRef :: a -> m (IORef a)
```

Then, to achieve the behaviour described in the last sections we just have to define an instance of `MonadRef` for `MonadIO`.

```
instance MonadIO m => MonadRef m where
    readRef = liftIO . readIORef
    modifyRef r f = liftIO $ modifyIORef r f
    writeRef r a = liftIO $ writeIORef r a
    newRef = liftIO . newIORef
```

Using this approach the new encoding for `StackObj` is as follows.

```
stackObj
    =
       do
        s <- newRef []
        return
               $ getStack .=. (readRef s)
             .*. push .=. (pushop s)
             .*. pop  .=. (popop s)
             .*. emptyRecord

pushop :: (CamilaMonad m, MonadRef m) => IORef [a] -> a -> m ()
pushop s i = modifyRef s (i:)

popop :: (CamilaMonad m, MonadRef m) => IORef [a] -> m a
popop s = do l <- readRef s
```

```
        pre (not $ null l)
        modifyRef s tail
        return (head l)
```

6 Folder Example

Let us now focus on an example that involves inheritance – the folder objectification example discussed in [4]. This example starts with two purely functional classes which model a stack and a folder – `StackAlg` and `FolderAlg` – including appropriate access methods. Two similar classes with state are then encoded – `StackObj` and `FolderObj` (objectification of the two purely functional classes). The hierarchy for these classes is depicted in figure 1.

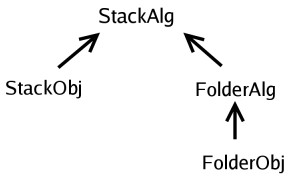


Fig. 1. Class hierarchy for the Folder example

Class `StackAlg` is purely functional, providing an API for Stack's.

```
type Stack = [Elem]
type Elem = Int

stackAlg :: CamilaMonad m => s ->
            m (Record (
                Empty :=: Stack
            :*: IsEmpty :=: (Stack -> Bool)
            :*: Push :=: (Elem -> Stack -> Stack)
            :*: Pop  :=: (Stack -> Stack)
            :*: Top :=: (Stack -> Elem)
            :*: HNil))


stackAlg self
    =
      return
```

```
              $ empty .=. []
            .*. isempty .=. (==[])
            .*. push .=. (:)
            .*. pop .=. tail
            .*. top .=. head
            .*. emptyRecord
```

Class `StackObj` is a subclass of `StackAlg` and offers an *object oriented* API for the outside [4] where all the function signatures lose the `Stack` parameter when compared to the corresponding ones in the superclass.

```
stackObj self
    =
       do
        super <- stackAlg self
        s <- newRef []
        return
              $ getStack .=. (readRef s)
            .*. opempty .=. (emptyop s super)
            .*. opisempty .=. (isemptyop s super)
            .*. oppush .=. (pushop super s)
            .*. oppop  .=. (popop s super)
            .*. optop .=. (topop s super)
            .*. super
```

The auxiliary functions are defined as follows.

```
-- push operation
pushop super s i = modifyRef s ((super # push) i)

popop s super = do l <- readRef s
                   pre (not $ null l)
                   modifyRef s (super # pop)

topop s super
        = do l <- readRef s
             pre (not $ null l)
             return ((super # top) l)

emptyop s super = modifyRef s ((!) (super # empty))

isemptyop s super = do  x <- readRef s
                        return $ (super # isempty) x
```

The functional core for a Folder is represented in the `FolderAlg` class. A folder is modelled *via* two stacks and provides methods to turn pages forward and backwards:

```
folderAlg self =
          do
             super <- stackAlg self
             return $
                  new .=. ((super # empty, super # empty))
              .*. insert .=. (\e -> id >< ((super # push) e))
              .*. remove .=. (id >< (super # pop))
              .*. backward .=. (backwardop super)
              .*. forward .=. (forwardop super)
              .*. emptyRecord
```

Methods to turn pages are simply defined by inheriting stack methods as follows:

```
backwardop super f = let e = (super # top) $ fst f
                      in  ((super # pop) >< ((super # push) e)) f

forwardop super f = let e = (super # top) $ snd f
                    in  (((super # push) e) >< (super # pop)) f
```

The subclass `FolderObj` of `FolderAlg`, which offers the functionality to the outside world is defined similarly to `FolderAlg`. It contains an internal state which is altered by the methods provided.

```
folderObj self
    =
        do
         super <- folderAlg self
         f <- newRef (super # new)
         return
               $
                  opnew .=. (modifyRef f $ const (super # new))
              .*. opinsert .=. (\e -> modifyRef f $ (super # insert) e)
              .*. opremove .=. (modifyRef f (super # remove))
              .*. opbackward .=. (modifyRef f (super # backward))
              .*. opforward .=. (modifyRef f (super # forward))
              .*. super
```

6.1 Checking pre-conditions

Similarly to [16], let us show that the pre/post conditions checking using `runCamilaT` is still working in the new encoding.

As a test case, let us define a function which checks what happens when we try to pop an element from an empty stack.

```
testPopEmptyStack () =
   do
    s <- mfix (stackObj)
    n <- s # oppop
    return n
```

The object construction is done with `mfix` function, that ties the recursive knot for the self references. Thus, we can see `mfix` as the Java operator `new`.

As expected when invoking `runCamilaT`, we get no warnings with mode **freefall**; exception, using **Fail**; error, using **ErrorMode** and a warning, using **Warn** mode.

```
*Camila.Examples.StackOO> runCamilaT $ freeFall $ testPopEmptyStack ()
*Camila.Examples.StackOO> runCamilaT $ fatal $ testPopEmptyStack ()
*** Exception: PreConditionViolation
*Camila.Examples.StackOO> runCamilaT $ errorMode $ testPopEmptyStack
()
*** Exception: user error (PreConditionViolation)
*Camila.Examples.StackOO> runCamilaT $ warn $ testPopEmptyStack ()
PreConditionViolation
```

7 Conclusion

The representation of classes, objects and state in HASKELL was deeply studied in [9]. The translation of such features from VDM++ to HASKELL appeared to be straightforward.

A major obstacle was the monad used to contain the class/object. OOHASKELL (for simplicity) uses the IO monad, since `IORef`'s are used to model mutable variables.

This approach was found unsuitable for our purposes, because of the need to check invariants and pre/post conditions in different modes. To offer this behaviour all the computations must be performed inside a `CamilaMonad`. Therefore, we have to lift the `IO` monad to a `CamilaMonad`. This was done through an intermediate class – `MonadRef` – which allows the encapsulation of `IORef` operations in a more general monad.

7.1 Future Work

As future work, and to bring all relevant VDM++ features to PURe Camila, we believe it is important to investigate how to implement parallel behaviour. Several research work in how to implement parallelism and concurrency in HASKELL has been done [15,12,1]. The `MonadRef` definition could be improved so that the monad used for implementing mutable references could change. To do so `MonadRef` would have an extra argument. Then, we could easily use `STRef` [11] instead of `IORef` to model mutable variables. It would be enough to declare a new instance of `MonadRef` for `STRef`. A hint in how to encode this approach follows.

```
class MonadRef m r where
    readRef :: r a -> m a
    modifyRef :: r a -> (a -> a) -> m ()
    writeRef :: r a -> a -> m ()
    newRef :: a -> m (r a)

instance MonadRef m IORef where
...

instance MonadRef m (STRef s) where
...
```

It would also be interesting to work on object communication semantics [2].

References

1. Shail Aditya, Jan-Willem Maessen, and Lennart Augustsson. Semantics of ph: A parallel dialect of haskell. Technical Report Computation Structures Group Memo 377-1, MIT, June 1995.
2. M. Barbosa and L. S. Barbosa. A relational model for component interconnection. *Journal of Universal Computer Science*, 10(7):808–823, 2004.
3. A. Bayley. Functional programming vs object oriented programming, June 2005. http://www.haskell.org/tmrwiki/FpVsOo.
4. A. Cruz, L. Barbosa, and J. Oliveira. From algebras to objects: Generation and composition. *Journal of Universal Computer Science*, 11(10):1580–1612, 2005.
5. Martin Erwig and Deling Ren. Monadification of functional programs. *Sci. Comput. Program.*, 52(1-3):101–129, 2004.
6. John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag Telos, 2005.
7. John Hughes and Jan Sparud. Haskell++: An object-oriented extension of haskell. In *Haskell Workshop 1995*, 1995.
8. Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions and foreign-language calls in HASKELL. May 2005.

9. Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. 2005.
10. Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.
11. John Launchbury and Simon L. Peyton Jones. State in haskell. *Lisp Symb. Comput.*, 8(4):293–341, 1995.
12. H-W. Loidl, P.W. Trinder, K. Hammond, S.B. Junaidu, R.G. Morgan, and S.L. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience*, 11:701–752, 1999.
13. J. Nuno Oliveira and Luis Soares Barbosa. Transposing partial components- an exercise on coalgebraic refinement. Technical report, U. Minho, Sept. 2005.
14. Mark Shields and Simon Peyton Jones. Object-oriented style overloading for Haskell. In *First Workshop on Multi-language Inferastructure and Interoperability (BABEL'01), Firenze, Italy*, September 2001.
15. P.W. Trinder, H-W. Loidl, E. Barry Jr., K. Hammond, U. Klusik, S.L. Peyton Jones, and Á.J. Rebón Portillo. The Multi-Architecture Performance of the Parallel Functional Language GPH. In A. Bode, T. Ludwig, and R. Wismüller, editors, *Euro-Par 2000 — Parallel Processing*, LNCS, Munich, Germany, 29.8.-1.9., 2000. Springer-Verlag.
16. Joost Visser, J. Nuno Oliveira, Luis Soares Barbosa, Joao Ferreira, and Alexandra Mendes. Camila: Vdm meets HASKELL. 2005.
17. Philip Wadler. Monads for functional programming. In *Marktoberdorf Summer School on Program Design Calculi*, August 1992.
18. Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4(1):1–32, 2003.

A Demo

All the code used in this report is available under PURe CVS repository. To run the examples you just have to follow the instructions described below.

A.1 Getting the code

First , you need an environment variable `CVS_RSH` set to `ssh`, such that cvs will make a connection with the server on which the repository resides via `ssh`.

Then, you can do checkout as follows.

```
$ cvs -d :pserver:username@haskell.di.uminho.pt:/mnt/ds/cvsroot login
$ cvs -d :pserver:username@haskell.di.uminho.pt:/mnt/ds/cvsroot checkout PURe
```

If you do not have an username for the repository then, for read only access, you can use the username `anonymous` (password `anonymous`).

A.2 Running the examples

In the directory `libraries`, inside `PURe` (`PURe/software/haskell/libraries`), just type:

```
$ make top=Camila/Examples ghci
```

Inside `ghci` you can test the defined examples.

```
*Camila.Examples.InfSet> :m +Camila.Examples.StackObjOO
*Camila.Examples.InfSet Camila.Examples.StackObjOO> runCamilaT $ warn $ testPopEmptyStack ()
PreConditionViolation
```