Reengeneering of the PUReCamila interpreter

DRAFT VERSION

Alexandra Silva

2005, August

Abstract

This report describes the reengeneering of the PURe CAMILA interpreter. Some features that will provide more specification power, similar to VDM-SL and VDM++, and are not currently implemented in the CAMILA librarie, are also analysed.

1 Introduction

The CAMILA project explores how concepts from the VDM specification language and the functional programming language Haskell can be combined. This project, part of PURe project[1], is a revival of the original CAMILA system[2], initially developed in the 90's at University of Minho.
In the first phase of this project, J. Ferreira and A. Mendes[MF05, MFP05] developed an Haskell library that allows to express VDM-SL data types (maps, sets, sequences), data type invariants, pre- and post-conditions. To animate this concepts they built a small prototype interpreter.
The interpreter was built on top of `hsplugins` [Ste05] - a library that provides a mechanism to recompile haskell code at runtime. This indirection was introduced to overcome some difficulties in handling the interpreter state but introduces some efficiency problems.
This project aims to develop an alternative implementation that avoid the use of `hsplugins` to animate CAMILA objects and, instead, relies directly on the ghci IO monad interaction. Since there are several state changes involved – we must clearly set the difference between the object-state level and the interpreter-state level – it is required a careful review of the monads and monad-transformers used in the project.
A careful review of the monadic modeling of VDM-SL features of [MF05] is done in [VOB$^+$05]. Parametrized monads are used to control the switching among the different modes of evaluation.
To mantain the state of the objects we need a mutable variable. An approach to have mutable variables in HASKELL is described in 2. The

[1]FCT under contract POSI/ICHS/44304/2002
[2]http://camila.di.uminho.pt

storing of the several camila objects in the same structure implies the use of generic programming techniques, detailed in section 3.
VDM++ is a formal specification language intended to specify object ori- ented (OO) systems with parallel behaviour. The use of classes and object concepts facilitates the development of object oriented formal specifications. To model this behaviour in HASKELL we describe OOHASKELL in section 4, a librarie that models OO features in Haskell.

2 IORef

A mutable variable, often used in imperative languages, is a very useful feature. In HASKELL it can be modeled using the HASKELL primitive `IORef` [Jon05].

```
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 using the following functions:

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

that, respectively, allow to create, read and write a reference to a value `a`.
To better understand how we can easily use this concept it follows an example, that models a counter.

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

incCounter = readIORef counter >>= \c -> writeIORef counter (c+1)

inspectCounter :: IORef Int -> Int
inspectCounter =  unsafePerformIO . readIORef
```

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

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

3 Generic Programming

The next topics are intimately related. The set of ideas presented were result of a very richful interaction between three great researchers –

Simon Peyton Jones, Ralf Lammel and Joost Visser.
Although the three subjects can be seen as generic programming tools, some diferences bettween them justify the description of all in this report.

3.1 *Scrap your BoilerPlate*

This technique is very general (it supports arbitrary datatypes, including parametrised, nested and mutually dependent types) and makes the application program flexible to changes in the data types [LP03]. It saves time for several reasons:

- allows to write most of the boilerplate code only once;
- makes easier the updates (since there is less code to change, the probability of errors substantially decreases);
- reduces the testing time (specially the "looking for bugs" time), since the boilerplate code has only to be tested once.

The *Scrap your boilerplate* work was further developed to include generic functions that traverse generic data structures ([LP05]) and generic casts ([LP04]).

3.2 HList

An heterogeneous collection can be regarded as a data structure that works as a repository fr objects of different types and allows operations of update, iteration, lookup, etc ([KLS04]). On a first approach we can define such a structure as:

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

However, the type of each element is not precisely described. Using type level programming, Ralf Lammel *et al* have defined a more typeful heterogeneous list.

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

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 ([KLS04]):

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

The HList library developed can be used in the cases when the type of the sequence is not known statically.

3.3 Strafunski

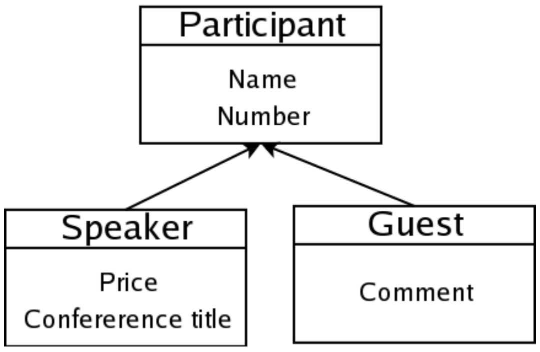
[LV03] Strafunski 2
[LV02a] Strafunski 1
[LV02b] Strafunski

4 OOHaskell

Object oriented languages offer features such as encapsulation, mutable state, overriding, inheritance, ... Implementing this features in HASKELL is an issue that has been addressed for several years. There were some encodings proposed for OO idioms in HASKELL ([HS95, Bay05, KL05]). In this section, we will analyse the features of the lybrary OOHASKELL recently developed (major realease in September 2005), that tries to overcome the insufficiencies of previous OO libraries in HASKELL and delivers an *amount of polymorphism and type inference that is unprecedented* ([KL05]). Several comparisons of OOHASKELL to other implementations are done in [KL05]. The refered library is based in the extensible polymorphic records of HLIST (section 3.2).

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

Let us analyse the following example that tries to model the participants in a conference.



There are three types of participants:

- Regular participants, that have name and number;
- Speakers, that aditionally to the participants information have the conference title and the expenses;
- Guests, that aditionally to the participants information have a comment (reason of attending the conference, etc)

We can encode this example using OOHASKELL as follows.

```
-- The class Participant

-- First, declare the labels.
-- Using proxies as of HList/Label4.hs

data GetName;       getName     = proxy::Proxy GetName
data GetNumber;     getNumber   = proxy::Proxy GetNumber
data SetName;       setName     = proxy::Proxy SetName
data SetNumber;     setNumber   = proxy::Proxy SetNumber


-- This is the actual definition of class participant

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
```

As can be seen in the definition of the class `Participant` , classes are function that take constructor arguments and a *self* reference.

```
-- Speaker: inherits from Participant

data GetPrice;             getPrice = proxy::Proxy GetPrice
data GetConferenceTitle;   getConferenceTitle = proxy::Proxy GetConferenceTitle
data SetPrice;             setPrice = proxy::Proxy SetPrice
data SetConferenceTitle;   setConferenceTitle = proxy::Proxy SetConferenceTitle

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

-- SpecialGuest: inherits from Participant

data GetComment; getComment = proxy:: Proxy GetComment
data SetComment; setComment = proxy:: Proxy SetComment

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

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.
The following function encodes an example that uses the classes defined above.

```
myOOP = do
        -- Insertion of new participants
        p1 <- mfix (participant "Xana" (10::Int))
        p2 <- mfix (participant "Bacelar" (11::Int))
        p3 <- mfix (participant "Joost" (12::Int))
        p4 <- mfix (speaker "Gibbons" (1::Int) (100::Int) "Folds")
        p5 <- mfix (guest "Jones" (1::Int) "")
        let l=consLub p1 (consLub p2 (consLub p3 (consLub p4 (consLub p5 nilLub))))
        mapM_ (\p -> do
                      n <- p # getNumber
                      m <- p # getName
                      putStrLn ("Participant number " ++ (show n) ++ "--" ++ (show m)
```

The results of invoking this function in `ghci` are presented below.

```
*Participant> myOOP
Participant number 10--"Xana"
Participant number 11--"Bacelar"
Participant number 12--"Joost"
Participant number 1--"Gibbons"
Participant number 1--"Jones"
```

5 Developed work

The module `State.hs`, described in [MF05], simultaneously defines a state for Camila objects and for the interpreter. This concepts should be explicitly divided. So, a new module `InterpState` was defined to contain the data types and operations concerning the interpreter state. To construct the store for the camila elements used by the user in the interpreter, I defined a weakly typed collection, that should be replaced by a strongly typed collection (3.2), as described in 7. There has been an attempt to define the store strongly typed but after some type checking problems (and lack of time) I ended up with this simpler solution.

```
data CamilaState = CamilaState {
                    store :: Store
                    }

type Store = [CamilaElement]
data CamilaElement = forall b. Typeable b => CamilaElement (String,b)
```

The initial state of the interpreter is:

```
initialSt :: Store
initialSt = []

-- | The initial state of the interpreter.
camilaInitial :: CamilaState
camilaInitial = CamilaState { store = initialSt}
```

The `Store` data structure has defined some funcionalities, such as insertion and lookup, as HASKELL classes, using the same principle described in section 3.2. The classes `AddToStore` and `HLookup` implement, respectively, the insertion and the lookup operations.

```
class AddToStore l b where
    add :: l -> b -> l

instance AddToStore Store CamilaElement where
   add [] c = [c]
   add  (a@(CamilaElement (y,b)):rs)
         e@(CamilaElement (x,v)) | (typeOf b == typeOf v) && (x==y) = (CamilaElement
                                 | (x==y) = (CamilaElement (x,b) : add rs (CamilaElem
                                 | otherwise = a : (add rs e)

class HLookup l b where
  hLookup :: String -> l -> Maybe b

-- | Definition of Lookup for Store
instance Typeable b => HLookup Store b where
    hLookup _ [] = Nothing
    hLookup x (a@(CamilaElement (y,b)):rs) | x==y = case (cast b) of
                                Nothing -> hLookup x rs
                                Just k -> Just k
                                           | otherwise = hLookup x rs
```

The interpreter must have information about the state of the camila objects.

```
-- | State of the interpreter

state :: IORef (CamilaState)
state = unsafePerformIO $ newIORef $ initial

-- | Initial State of the Interpreter
```

```
initial :: CamilaState
initial = camilaInitial
```

The operations defined for the camila interpreter are:

- creation of a new variable – if the variable has alrady a value of the same type we are trying to assign this is replaced; otherwise, a new item in the `Store` is created;

```
-- | Definition of a variable value

(.=) :: (Typeable b, Show b) => String -> b -> IO ()
(.=) = ccreate

ccreate id h = modifyIORef state (\s -> (s { store = add (store s) (CamilaElemen
```

- lookup for the value of a variable (of a certain type);

```
-- | Checking if a variable of type b with a certain label
-- | is already defined and what value it has
-- | It is a generic lookup for the Interpreter store

(.>) :: (Typeable b) => String -> Maybe b
(.>)=look

look id = (hLookup id (store $ unsafePerformIO $ readIORef state))
```

- apply a function to a variable;

```
-- | Function Application to a defined variable
(.$) :: (Typeable b) => (b -> a) -> String -> Maybe a
(.$) = apply

apply f id = do x <- look id
                return (f x)
```

5.1 Examples

Loading the interpreter to `ghci` we can use the features described above. Some simple examples of creating and lookup values of variables as well as applying a function to a variable follow.

```
*Camila.Interpreter> ccreate "bool" True
*Camila.Interpreter> ccreate "str" "Mfp2"
*Camila.Interpreter> ((.>) "str")::Maybe String
Just "Mfp2"
*Camila.Interpreter> ((.>) "a")::Maybe String
```

	Features	Camila
Vdm-sl	invariants	√
	pre/post conditions	√
	state	√
Vdm++	class	—

```
Nothing
*Camila.Interpreter> ((.>) "bool")::Maybe String
Nothing
*Camila.Interpreter> ((.>) "bool")::Maybe Bool
Just True
*Camila.Interpreter> apply (\s->s++"x") "str"
Just "Mfp2x"
*Camila.Interpreter> apply (==True) "bool"
Just True
*Camila.Interpreter> apply (==False) "bool"
Just False
```

We can have different values store in the same variable (if they have different types) and still apply functions to the variable, as we can see in the following example.

```
*Camila.Interpreter> "x" .= "abcd"
*Camila.Interpreter> "x" .= True
*Camila.Interpreter> apply (==False) "x"
Just False
*Camila.Interpreter> apply ("123"++) "x"
Just "123abcd"
```

6 General overview

7 Conclusions and Future Work

References

[Bay05] A. Bayley. Functional programming vs object oriented programming. June 2005.

[HS95] Jonh Hughes and Jan Sparud. Haskell++ : An object oriented extension of Haskell. April 1995.

[Jon05] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions and foreign-language calls in Haskell. May 2005.

[KL05] Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. 2005.

[KLS04] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceed-*

ings of the ACM SIGPLAN workshop on Haskell, pages 96–107. ACM Press, 2004.

[LP03] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[LP04] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 244–255. ACM Press, 2004.

[LP05] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages ??–?? ACM Press, September 2005.

[LV02a] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, January 2002.

[LV02b] Ralf Lämmel and Joost Visser. Design Patterns for Functional Strategic Programming. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, October5 2002. ACM Press. 14 pages.

[LV03] R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.

[MF05] Alexandra Mendes and João Ferreira. Pure camila – a system for software development using formal methods. Technical report, University of Minho, 2005.

[MFP05] Alexandra Mendes, João Ferreira, and José Proença. Camila prelude. Technical report, University of Minho, 2005.

[Ste05] Don Stewart. hs-plugins – dynamically loaded HASKELL modules. July 2005.

[VOB$^+$05] Joost Visser, J.Nuno Oliveira, Luís Soares Barbosa, João Ferreira, and Alexandra Mendes. Camila: Vdm meets haskell. 2005.