Camila Revival: VDM meets Haskell

Overture Workshop
18 July 2005, Newcastle upon Tyne, UK
Co-located with FM’05

Joost Visser, J.N. Oliveira, L.S. Barbosa, J.F. Ferreira, and A. Mendes

DI/U.Minho Braga, Portugal
CAMILA Revival

FM tools at Minho

- **CAMILA** software (1986-1997)
- **VDMTools** (1998-2005)

What next?

- **CAMILA Revival** (*Haskell* based)
- **Overture** (*Eclipse* based)

Why *Haskell*?
CAMILA Revival

Objectives

- FM perspective: exploit Haskell’s advanced type system and extensive suite of libraries for specification purposes.
- FP perspective: bring VDM features, such as constrained datatypes and partial functions, into the functional programmer’s reach.

So far

- Capture VDM operations in Haskell libraries (CPrelude)
- Implement VDM interpreter in Haskell (iCamila)
- Model VDM state features monadically
- Model VDM partiality features monadically (current paper)
<table>
<thead>
<tr>
<th>VDM versus Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Specification</strong></td>
</tr>
<tr>
<td><strong>Set-theoretic</strong></td>
</tr>
<tr>
<td>Numerous built-in operators</td>
</tr>
<tr>
<td><strong>Strict</strong></td>
</tr>
<tr>
<td><strong>Implicit functions</strong></td>
</tr>
<tr>
<td><strong>Datatype invariants</strong></td>
</tr>
<tr>
<td><strong>Pre / post conditions</strong></td>
</tr>
<tr>
<td><strong>State</strong></td>
</tr>
<tr>
<td><strong>Programming</strong></td>
</tr>
<tr>
<td><strong>Type-theoretic</strong></td>
</tr>
<tr>
<td>Numerous library functions</td>
</tr>
<tr>
<td><strong>Lazy</strong></td>
</tr>
<tr>
<td>?</td>
</tr>
<tr>
<td>?</td>
</tr>
<tr>
<td>?</td>
</tr>
<tr>
<td>?</td>
</tr>
</tbody>
</table>
Why Haskell?

**Component-oriented design relies on compositionality** — the true basis of software construction — for instance

Recall

- Unix pipes $g | f$
- Functional composition, $\lambda x. f(g(x))$
- etc
Why Haskell?

Ideal world:

\[
\left[ \begin{array}{c}
g \\ \rightarrow \\ f \\ \rightarrow \\ \end{array} \right] = [f] \cdot [g]
\]
Why Haskell?

Ideal world:

\[
\begin{bmatrix}
g \\
f
\end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix}
\]

Real world!

pre-\(g\)  \quad \text{pre-}\(f\)  \quad \text{Internal state}
Why Haskell?

Ideal world:

\[
\begin{bmatrix}
g \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
f \\
\end{bmatrix} = [f] \cdot [g]
\]

Semantics of real world?

\[
\begin{bmatrix}
g \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
f \\
\end{bmatrix} = [f] \circ [g]
\]

Claim: just write (monadic) instead of [Overture — 07/18 – p.5/32]
Why Haskell?

Ideal world:

\[
\begin{bmatrix}
g \\
\end{bmatrix} \rightarrow \begin{bmatrix}
f \\
\end{bmatrix} = [f] \cdot [g]
\]

Semantics of real world?

\[
\begin{bmatrix}
A \\
\text{pre} \cdot g \\
\end{bmatrix} \rightarrow \begin{bmatrix}
g \\
\text{inv} \cdot B \\
B \\
\text{pre} \cdot f \\
\end{bmatrix} \rightarrow \begin{bmatrix}
f \\
\text{inv} \cdot C \\
C \\
\end{bmatrix} = [f] \text{ ?} [g]
\]

Claim: just write (monadic) \([f] \cdot [g]\) instead of \([f] \cdot [g]\).
Why monads

Compare:

\[(f \cdot g)a = \text{let } b = g(a) \text{ in } f(b)\]

with

\[(f \cdot! g)a = \text{do } \{ b \leftarrow g(a); f(b) \} \]
Why monads

Compare:

\[(f \cdot g)a = \text{let } b = g(a) \text{ in } f(b)\]

with

\[(f \cdot ! g)a = \text{do } \{ \text{ b <- g(a); } f(b) \} \]

where types are, in the second case, as follows

\[A \xrightarrow{g} M B\]

\[B \xrightarrow{f} M C\]
Why monads

Compare:

\[(f \cdot g)a = \text{let } b = g(a) \text{ in } f(b)\]

with

\[(f .! g)a = \text{do } \{ b <- g(a); f(b) \}\]

In detail:

\[
\begin{align*}
A \xrightarrow{g} M & \xrightarrow{f} M(M(C')) \xrightarrow{\mu} M(C) \\
B \xrightarrow{f} M(C)
\end{align*}
\]
Why monads

Compare:

\[(f \cdot g)a = \text{let } b = g(a) \text{ in } f(b)\]

with

\[(f \cdot ! g)a = \text{do } \{ b \leftarrow g(a); f(b) \} \]

Example (list monad):

\[
\begin{align*}
  A & \xrightarrow{g} [B] & \xrightarrow{\text{map } f} [[C]] & \xrightarrow{\text{concat}} [C] \\
  B & \xrightarrow{f} [C]
\end{align*}
\]
Standard definition

\[(f .! g)a = g(a) \gg= f\]

where

```haskell
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    fail :: String -> m a
```
Partiality and the **Error** monad

Which monad M? A popular choice for handling partiality is

- datatype

```haskell
data Error a = Err String | Ok a
```

- that is, monad

```haskell
instance Monad Error where
    return b = Ok b
    (Err e) >>= f = Err e
    (Ok a)  >>= f = f a
```
First experiment

“Monadify” normal functions,

\[ [f]a = \text{Ok}(f \ a) \]

and convert conditions and invariants to monadic partial identities, eg.

\[ [\text{inv}]\ a = \text{if} \ (\text{inv} \ a) \]
\[ \quad \text{then} \ \text{Ok} \ a \]
\[ \quad \text{else} \ \text{Err} \ "\text{Invariant violation}" \]

(So \ [\text{inv}] \ :: \ a \to \text{Error} \ a \ \text{while} \ \text{inv} \ :: \ a \to \text{Bool} \)
Back to the real world

In this way, we get a very simple, “pipelined” approach to composition

where the arrows are \( \text{Error-monadic} \) — think of \( \cdot \) ! instead of \( \cdot \) — that is

\[
\begin{align*}
\text{do } & \{ \text{pre-}g \ a; \\
b & \leftarrow g \ a; \\
\text{inv-}B \ b; \ & \text{pre-}f \ b; \\
c & \leftarrow f \ b; \\
\text{inv-}C \ c \\
\}\n\end{align*}
\]
Monadic invariant example

Invariant associated to a relational table $t$ with schema $s$ in a RDB system:

```haskell
inv (Rel s t) = do {
  m <- mfoldS munion' (Ok emptyPf) (nmap (id *-> valType) t)
  `otherwise_` "Tuple schemas are not mutually compatible" ;
  check (relSchemaOk m) (Rel s t)
  "At least one tuple type does not match relation schema" ;
  check fdpOk (Rel s t)
  "The key-property is not valid in the relation"
}
where
  relSchemaOk m r = m <= (id *-> (valType . defaultV)) (schema r)
  fdpOk (Rel s t) = fdp(nmap (tnest (getKeyAtts s)) t)
```

(Excerpt of Necco’s Haskell model of a relation in a RDB system. Note the successively contextualized error messages interspersed with the monadic code.)
Why this not enough

- We are stuck to a single monad (*Error*) and a single evaluation mode *(fail)*

We would like to be able to switch among

- **free fall** —no checking performed whatsoever.
- **warn** —when invariants and conditions are found violated, a warning will be issued, but computation proceeds as if nothing happened.
- **fail** —invariant and conditions checked, and when found violated a run-time error is forced immediately.
- **error** —invariants and conditions are checked, and when found violated an error or exception will be thrown.
Running example (VDM)

VDM model of stacks of odd integers — (partial) datatype

Stack = seq of int
    inv s = forall a in set elems s & odd(a);

and (partial) functions

empty : Stack -> bool
    empty(s) == s = [];

pop : Stack -> Stack
    pop(s) == tl s
    pre not empty(s);

top : Stack -> int
    top(s) == hd s
    pre not empty(s);

push : int * Stack -> Stack
    push(p,s) == [p] ^ s
    pre odd(p) ;
Constrained datatypes (Haskell)

We go back to invariants as Boolean functions and define class

```haskell
class CData a where
  inv :: a -> Bool
  inv a = True -- default
```

so that invariants propagate dynamically, eg. listwise

```haskell
instance CData a => CData [a] where
  inv = all inv
```

eg. pairwise

```haskell
instance (CData a, CData b) => CData (a,b) where
  inv (a,b) = (inv a) && (inv b)
```

etc
Semantics of VDM type Stack

\[
\text{Stack} = \text{seq of int} \\
\quad \text{inv } s = \text{forall } a \text{ in set elems } s \& \text{ odd}(a);
\]

\[
\text{newtype Stack} = \text{Stack} \{ \text{theStack} :: [\text{Int}] \}
\]

\[
\text{instance CData Stack where} \\
\quad \text{inv } s = \text{all odd (theStack } s)\]

- In general, VDM partial types such as \text{Stack} are mapped into \text{CData} instances.
- What about (partial) functionality?
Define **CamilaMonad**, a subclass of **Monad**

```haskell
class Monad m => CamilaMonad m where
    -- | Check precondition
    pre :: Bool -> m ()
    -- | Check postcondition
    post :: Bool -> m ()
    -- | Check inv before returning data in monad
    returnInv :: CData a => a -> a -> m a
```

which cares about pre-/post-conditions and invariants.
Monadic VDM translation

Example, showing *genericity* of the translation—for any *CamilaMonad* m,

\[
\begin{pmatrix}
\text{top : Stack} \rightarrow \text{int} \\
\text{top(s)} == \text{hd s} \\
\text{pre not empty(s)};
\end{pmatrix}
\quad = 
\begin{cases}
\text{top :: CamilaMonad m => Stack} \rightarrow \text{m Int} \\
\text{top s =} \\
\quad \text{do } \{ \\
\quad \text{pre (not (empty s))}; \\
\quad \text{return (head (theStack s))}
\}\;
\end{cases}
\]

Note the difference: our first approach was bound to

\[
\text{top :: Stack} \rightarrow \text{Error Int}
\]

How is this to work?
CamilaT monad transformer

We need a family of monads, one per evaluation mode. So, we define

```haskell
data CamilaT mode m a =
   CamilaT {runCamilaT :: m a}
```

**NB:**

- **CamilaT mode m** is isomorphic to **m**:
  ```haskell
  instance Monad m => Monad (CamilaT mode m) where
      return   = CamilaT . return
      ma >>= f = CamilaT (runCamilaT ma >>=
                           runCamilaT . f)
  ```

- **CamilaT** will add checking effects to a given base monad, depending on the phantom **mode** argument (**type-indexed family of monads**);
Free fall mode

Define type

```
data FreeFall
```

and then instantiate `CamilaMonad` as follows:

```
instance Monad m =>
    CamilaMonad (CamilaT FreeFall m) where
    pre p = return ()
    post p = return ()
    returnInv = return
```

Thus

- pre-/post-conditions \( p \) are simply ignored
- the invariant-aware return simply does not check it
Example (free fall mode)

Taking top of an empty stack

testTopEmptyStack :: CamilaMonad m => m Int
testTopEmptyStack = do { 
    s <- initStack ; -- create empty stack 
    n <- top s ;
    return n 
}

In free-fall mode we get

> runCamilaT $ freeFall testTopEmptyStack
*** Exception: Prelude.head: empty list

as expected.
Fail mode

Define type

```haskell
data Fail

and then instantiate `CamilaMonad` as follows:

```haskell
instance Monad m => CamilaMonad (CamilaT Fail m) where
  pre p = if p then return ()
    else fail "Pre-condition violation"
  post p = if p then return ()
    else fail "Post-condition violation"
  returnInv a = if (inv a) then return a
    else fail "Invariant violation"
```

Thus, when violations are detected, the standard `fail` function is used to force an immediate `fatal` error.
Running example (fail mode)

Taking top of an empty stack in fail mode will yield

```
> runCamilaT $ fatal testTopEmptyStack
*** Exception: Pre-condition violation
```

as expected.
Warn mode

Define type

data Warn

To enable reporting, we need a monad with writing capabilities, eg the standard IO monad:

instance MonadIO m => CamilaMonad (CamilaT Warn m) where
   pre p = unless p $ liftIO $ putErr "Pre-condition violation"
   post p = unless p $ liftIO $ putErr "Post-condition violation"
   returnInv a = do
      unless (inv a) $ liftIO $ putErr "Invariant violation"
      return a

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

(The `unless` combinator runs its monadic argument conditionally on its boolean argument.)
Running example (warn mode)

Taking top of an empty stack in warn mode will yield

```
> runCamilaT $ warn testTopEmptyStack
Pre-condition violation
*** Exception: Prelude.head: empty list
```

It signals out **Pre-condition violation** but carries on, later to crash as in the free-fall mode.
Running example (error mode)

(See paper for details on the `CamilaMonad` instance for this mode)

Taking top of an empty stack in error mode will yield

```haskell
> runCamilaT $ errorMode testTopEmptyStack
*** Exception: user error Pre-condition violation
```

So, an exception is raised, but the text `user error` in the message indicates that this exception is actually catchable, and not necessarily fatal.
Fatal versus error modes

Difference between fail mode and error mode becomes clear when we try to catch the generated exceptions: compare

```haskell
> (runCamilaT $ fatal testTopEmptyStack)
  'catchError' \_ -> putStrLn "CAUGHT" >> return 42
*** Exception: Pre-condition violation
```

with

```haskell
> (runCamilaT $ errorMode testTopEmptyStack)
  'catchError' \_ -> putStrLn "CAUGHT" >> return 42
CAUGHT
```

Thus, exceptions that occur in error mode can be caught, higher in the call chain, while in fail mode the exception always gets propagated to the top level.
Details on elegance of solution

Clever use of the identity function’s polymorphism:

```haskell
freeFall :: CamilaT FreeFall m a -> CamilaT FreeFall m a
freeFall = id

warn :: CamilaT Warn m a -> CamilaT Warn m a
warn = id
```

etc (= let the type system do work — type level programming !)
newtype Stack = Stack { theStack :: [Int] }
instance CData Stack where inv s = all odd (theStack s)

empty :: Stack -> Bool
empty s = theStack s == []

push :: CamilaMonad m => Int -> Stack -> m Stack
push n s = do {
    pre (odd n) ;
    returnInv $ Stack (n : theStack s)
}

pop :: CamilaMonad m => Stack -> m Stack
pop s = do {
    pre (not $ empty s) ;
    returnInv $ Stack $ tail $ theStack s
}

top :: CamilaMonad m => Stack -> m Int
top s = do {
    pre (not $ empty s) ;
    return (head $ theStack s)
}
Summary and current work

- Formal model animation has to do with rapid-prototyping (= early testing).
- Animation prepares model for proof obligation discharge (proofs become free of stupid errors)
- “Animatographer” (=interpreter) should be as flexible as possible — thus our evaluation modes (new ones can be invented, cf. eg. error logging)
- Different modes can be used (simultaneously) for different parts of the same model
- Example — switch component testing to free-fall as soon as proof obligations have been discharged for such a component, while keeping protecting the others’ animation
- Warn mode suited for testing via fault-injection
Closely related work

- **VDM conversion into Gofer** (Paul Mukherjee, FME’97) — comprehensive translation strategy is based on the (fixed) *state* and *error* monads.

- **VDMTools** (IFAD) — debugging and dynamic checking of invariants and pre-/post-conditions can be turned on and off individually.

- **VDM conversion into Lazy ML** (Borba & Meira, JSS 1993) — monads are not used; invariants are checked at input parameter passing time (rather than at value return time).

- **Irish VDM** (see A. Butterfield’s home page) — Haskell libraries, including QuickCheck support; concern for proof obligations.
Other related work

- **Programatica** — This is a system for the development of high-confidence software systems. Assertions are type-checked to ensure a base level of consistency with executable portions of the program and annotated with certificates that provide evidence of validity.

- **JCL** (Jakarta Commons Logging) — The Jakarta project of the Apache Software Foundation offers logging support in the form of a LogFactory class and a Log interface which offers methods like fatal, error, and warn to emit messages to consoles and/or log files.
Relevance for Overture

- Software **architecture** above all — with Haskell’s help
- Our monadic model for **VDM** property checking provides an answer to how such checking may be understood semantically.
- When compiling to **Java**, for instance, our monadic model so far suggests to consider using class parameters (possibly using a model of monads in Java?)
- We hope the outcome of our experiments may lead to **inspiration** for future developments in projects such as **Overture**.
- Haskell versus Java: **Scala** (F + OO)?