On the Functional Interpretation of OCL

Daniel Calegari Marcos Viera

Instituto de Computación, Universidad de la República
Montevideo, Uruguay

OCL - October 2, 2016

What is the talk about?

- We explore the use of Haskell as an interpreter for OCL

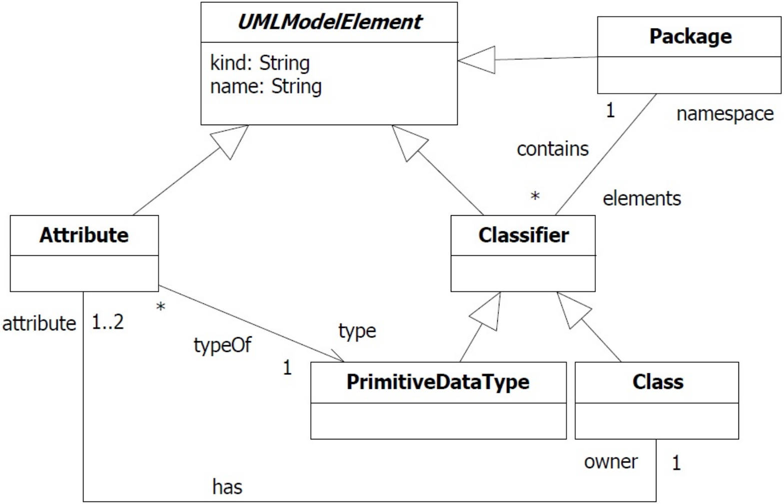
What is the talk about?

- We explore the use of Haskell as an interpreter for OCL
- We tackle with the functional representation of
  - model-oriented features in metamodels, models and OCL expressions
  - semantic interpretation of many OCL aspects

What is the talk about?

- We explore the use of Haskell as an interpreter for OCL
- We tackle with the functional representation of
  - model-oriented features in metamodels, models and OCL expressions
  - semantic interpretation of many OCL aspects
- What's next?

Running Example: UML Class Diagrams Metamodel



Representation of Metamodels

- Classes represented as datatypes with one constructor

Representation of Metamodels

- Classes represented as datatypes with one constructor

```
  -- UMLModelElement(oid:int, kind:String, name:String) + subtypes
data UMLModelElement = UMLModelElement Int String String
                                       UMLModelElementCh
```

Representation of Metamodels

- Classes represented as datatypes with one constructor

```
-- UMLModelElement(oid:int, kind:String, name:String) + subtypes
data UMLModelElement = UMLModelElement Int String String
                                       UMLModelElementCh
```

- Subclasses of a class *Class* represented by a field of type *ClassCh*

```
data UMLModelElementCh = UMLMECAtt Attribute
                       | UMLMECPck Package
                       | UMLMECCla Classifier
```

Representation of Metamodels

- Classes represented as datatypes with one constructor

```
-- UMLModelElement(oid:int, kind:String, name:String) + subtypes
data UMLModelElement = UMLModelElement Int String String
                                               UMLModelElementCh
```

- Subclasses of a class *Class* represented by a field of type *ClassCh*

```
data UMLModelElementCh = UMLMECAtt  Attribute
                       | UMLMECPck Package
                       | UMLMECCla  Classifier
```

- If the class is not abstract, the field is wrapped with *Maybe*

```
-- Classifier(namespace:Package) + subtypes
data Classifier = Classifier Int (Maybe ClassifierCh)
```

Representation of Metamodels (2)

- Properies are fields

Representation of Metamodels (2)

- Properies are fields
  - Primitive types mapped to their corresponding Haskell types

```
-- UMLModelElement(oid:int, kind:String, name:String) + subtypes
data UMLModelElement = UMLModelElement Int String String
                                        UMLModelElementCh
```

Representation of Metamodels (2)

- Properies are fields
  - Primitive types mapped to their corresponding Haskell types

```
-- UMLModelElement(oid:int, kind:String, name:String) + subtypes
data UMLModelElement = UMLModelElement Int String String
                                       UMLModelElementCh
```

  - In case of non-primitive types an integer represents the identifier of the element

```
-- Attribute(typ:Classifier, owner:Class)
data Attribute = Attribute Int Int
```

Representation of Metamodels (2)

- Properies are fields
  - Primitive types mapped to their corresponding Haskell types

```
  -- UMLModelElement(oid:int, kind:String, name:String) + subtypes
data UMLModelElement = UMLModelElement Int String String
                                                UMLModelElementCh
```

  - In case of non-primitive types an integer represents the identifier of the element

```
  -- Attribute(typ:Classifier, owner:Class)
data Attribute = Attribute Int Int
```

  - Multiplicities that accept many elements are represented by lists

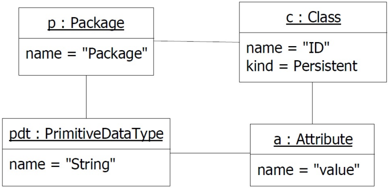
```
  -- Class(atts:Set(Attribute))
data Class = Class [Int]
```

Representation of Models

- Models are Haskell values of the type representing the Metamodel

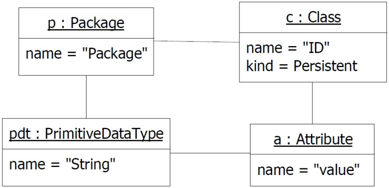
Representation of Models

- Models are Haskell values of the type representing the Metamodel



Representation of Models

- Models are Haskell values of the type representing the Metamodel



$$
\begin{array}{l}
\mathit{example} = \mathit{Model} \\
\quad [\mathit{UMLModelElement}\ 1\ \texttt{"Persistent"}\ \texttt{"Package"}\ (\mathit{UMLMECPck}\ \$\ \mathit{Package}\ [2,3]) \\
\quad , \mathit{UMLModelElement}\ 2\ \texttt{"Persistent"}\ \texttt{"String"}\ (\mathit{UMLMECCla}\ \$\ \mathit{Classifier}\ 1 \\
\qquad (\mathit{Just}\ \$\ \mathit{ClassifierChPri}\ \ \mathit{PrimitiveDataType})) \\
\quad , \mathit{UMLModelElement}\ 3\ \texttt{"Persistent"}\ \texttt{"ID"}\ (\mathit{UMLMECCla}\ \$\ \mathit{Classifier}\ 1 \\
\qquad (\mathit{Just}\ \$\ \mathit{ClassifierChCla}\ (\mathit{Class}\ [4]))) \\
\quad , \mathit{UMLModelElement}\ 4\ \texttt{"Persistent"}\ \texttt{"value"}\ (\mathit{UMLMECAtt}\ \$\ \mathit{Attribute}\ 2\ 3) \\
\quad]
\end{array}
$$

This representation can be automated by means of a model-to-text transformation

Representation of OCL

- The translation mimics the structure of the OCL invariants

Representation of OCL

- The translation mimics the structure of the OCL invariants

```
context Class inv:
  self.attribute->forAll ( a1 : Attribute; a2 : Attribute |
                           a1 <> a2 implies a1.name <> a2.n
```

Representation of OCL

- The translation mimics the structure of the OCL invariants

```
context Class inv:
  self.attribute->forAll ( a1 : Attribute; a2 : Attribute |
                           a1 <> a2 implies a1.name <> a2.n
```

$$
\begin{aligned}
&\mathit{chk1} \quad\;\; = \mathit{context}\ \_\mathit{Class}\ [\mathit{inv1}\,]\\
&\mathit{inv1}\ \mathit{self} = \mathit{ocl}\ \mathit{self}\ \mathtt{|.|}\ \mathit{atts}\ \mathtt{|\text{-}>|}\ \mathit{forAll}\ (\lambda(\mathit{Val}\ (\mathit{a1}, \mathit{a2})) \rightarrow\\
&\quad (\mathit{ocl}\ \mathit{a1}\ \mathtt{|<>|}\ \mathit{ocl}\ \mathit{a2})\ \mathtt{|==>|}\ ((\mathit{ocl}\ \mathit{a1}\ \mathtt{|.|}\ \mathit{name})\ \mathtt{|<>|}\ (\mathit{ocl}\ \mathit{a2}\ \mathtt{|.|}\ \mathit{name})))\\
&\quad .\ \mathit{cartesian}
\end{aligned}
$$

Representation of OCL

- The translation mimics the structure of the OCL invariants

```
context Class inv:
  self.attribute->forAll ( a1 : Attribute; a2 : Attribute |
                           a1 <> a2 implies a1.name <> a2.n
```

$$
\begin{array}{l}
\mathit{chk1} \quad = \mathit{context}\ \_\mathit{Class}\ [\mathit{inv1}] \\
\mathit{inv1}\ \mathit{self} = \mathit{ocl}\ \mathit{self}\ |.|\ \mathit{atts}\ |\text{->}|\ \mathit{forAll}\ (\lambda(\mathit{Val}\ (a1, a2)) \rightarrow \\
\quad (\mathit{ocl}\ a1\ |\text{<>}|\ \mathit{ocl}\ a2)\ |\text{==>}|\ ((\mathit{ocl}\ a1\ |.|\ \mathit{name})\ |\text{<>}|\ (\mathit{ocl}\ a2\ |.|\ \mathit{name}))) \\
\quad .\ \mathit{cartesian}
\end{array}
$$

- We defined an Embedded Domain Specific Language

Representation of OCL (2)

```
context Class inv:
  Class.allInstances()->forAll (c : Class |
    c.attribute->iterate ( a:Attribute; result:Boolean=true |
                           result and
                           a.type.namespace = c.namespace))
```

Representation of OCL (2)

```
context Class inv:
  Class.allInstances()->forAll (c : Class |
    c.attribute->iterate ( a:Attribute; result:Boolean=true |
                           result and
                           a.type.namespace = c.namespace))
```

```
chk2   = context _Class [inv2]
inv2 _ = ocl _Class |.| allInstances |->| forAll (λc →
             ocl c |.| atts |->| iterate (λres a →
                                    ocl res |&&|
                                    (ocl a |.| typ |.| namespace) |==|
                                    (ocl c |.| namespace))
                                   (Val True))
```

Functional OCL Library

- Invariants have type *OCL Model* (*Val Bool*)

Functional OCL Library

- Invariants have type *OCL Model* (*Val Bool*)
  - OCL expression that applies to a MOF model (represented by the type *Model*) and returns a boolean value

Functional OCL Library

- Invariants have type *OCL Model* (*Val Bool*)
  - OCL expression that applies to a MOF model (represented by the type *Model*) and returns a boolean value

- The type *Val* represents the OCL four-valued logic

$$\textbf{data}\ \mathit{Val}\ a = \mathit{Null} \mid \mathit{Inv} \mid \mathit{Val}\ a$$

Functional OCL Library

- Invariants have type *OCL Model* (*Val Bool*)
  - OCL expression that applies to a MOF model (represented by the type *Model*) and returns a boolean value

- The type *Val* represents the OCL four-valued logic

```
data Val a = Null | Inv | Val a
```

- The type *OCL m a* is a *Reader* monad

```
type OCL m a = Reader m a
```

Functional OCL Library

- Invariants have type *OCL Model* (*Val Bool*)
  - OCL expression that applies to a MOF model (represented by the type *Model*) and returns a boolean value

- The type *Val* represents the OCL four-valued logic

$$\textbf{data}\ \mathit{Val}\ a = \mathit{Null} \mid \mathit{Inv} \mid \mathit{Val}\ a$$

- The type *OCL m a* is a *Reader* monad

$$\textbf{type}\ \mathit{OCL}\ m\ a = \mathit{Reader}\ m\ a$$

representing computations which read from a shared environment of type m and return a value of type a.

The OCL Monad

- Monads structure computations in terms of values and sequences of (sub)computations that use these values.
  - Allow to incorporate side-effects and state

The OCL Monad

- Monads structure computations in terms of values and sequences of (sub)computations that use these values.
  - Allow to incorporate side-effects and state

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m\ \textbf{where} \\
\quad \mathit{return} :: a \to m\ a \\
\quad (\ggg\!\!=)\ :: m\ a \to (a \to m\ b) \to m\ b
\end{array}
$$

The OCL Monad

- Monads structure computations in terms of values and sequences of (sub)computations that use these values.
  - Allow to incorporate side-effects and state

$$
\begin{aligned}
&\textbf{class } \mathit{Monad}\ m \textbf{ where}\\
&\quad \mathit{return} :: a \to m\ a\\
&\quad (\gg\!=)\ :: m\ a \to (a \to m\ b) \to m\ b
\end{aligned}
$$

- A sequence of computations describes the navigation through properties and functions

The OCL Monad

- Monads structure computations in terms of values and sequences of (sub)computations that use these values.
  - Allow to incorporate side-effects and state

$$
\begin{array}{l}
\textbf{class } \mathit{Monad}\ m \textbf{ where} \\
\quad \mathit{return} :: a \rightarrow m\ a \\
\quad (\ggg\!=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b
\end{array}
$$

- A sequence of computations describes the navigation through properties and functions
- The shared environment can be used to look up the elements referred by others.

The OCL Monad (2)

- We defined specialized versions of the monad operations to represent:

The OCL Monad (2)

- We defined specialized versions of the monad operations to represent:
  - Construction of OCL expressions from values.

$$ocl :: Val\ a \rightarrow OCL\ m\ (Val\ a)$$
$$ocl = return$$

The OCL Monad (2)

- We defined specialized versions of the monad operations to represent:
  - Construction of OCL expressions from values.

$$
\begin{array}{l}
ocl :: Val\ a \rightarrow OCL\ m\ (Val\ a) \\
ocl = return
\end{array}
$$

  - Object navigation

$$
\begin{array}{l}
(|.|) :: OCL\ m\ (Val\ a) \rightarrow (Val\ a \rightarrow OCL\ m\ (Val\ b)) \rightarrow OCL\ m\ (Val\ b) \\
(|.|) = (\gg\!=)
\end{array}
$$

The OCL Monad (2)

- We defined specialized versions of the monad operations to represent:
  - Construction of OCL expressions from values.

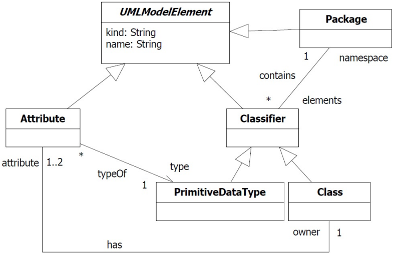
$$
\begin{aligned}
&\mathit{ocl} :: \mathit{Val}\ a \rightarrow \mathit{OCL}\ m\ (\mathit{Val}\ a)\\
&\mathit{ocl} = \mathit{return}
\end{aligned}
$$

  - Object navigation

$$
\begin{aligned}
&(|.|) :: \mathit{OCL}\ m\ (\mathit{Val}\ a) \rightarrow (\mathit{Val}\ a \rightarrow \mathit{OCL}\ m\ (\mathit{Val}\ b)) \rightarrow \mathit{OCL}\ m\ (\mathit{Val}\ b)\\
&(|.|) = (\ggg)
\end{aligned}
$$

  - Collection navigation

$$
\begin{aligned}
(|\text{->}|) ::\ & \mathit{OCL}\ m\ (\mathit{Val}\ [\mathit{Val}\ a]) \rightarrow (\mathit{Val}\ [\mathit{Val}\ a] \rightarrow \mathit{OCL}\ m\ (\mathit{Val}\ b))\\
& \rightarrow \mathit{OCL}\ m\ (\mathit{Val}\ b)\\
(|\text{->}|) =\ & (\ggg)
\end{aligned}
$$




```
chk2    = context _Class [inv2]
inv2 _ = ocl _Class |.| allInstances |->| forAll (λc →
                ocl c |.| atts |->| iterate (λres a →
                                        ocl res |&&|
                                        (ocl a |.| typ |.| namespace) |==|
                                        (ocl c |.| namespace))
                                       (Val True))
```

Collection Operators

- The `iterate` OCL operator is almost directly translated to the *fold* recursion scheme of Haskell.

$$
\begin{aligned}
&iterate :: (Val\ b \rightarrow Val\ a \rightarrow OCL\ m\ (Val\ b)) \rightarrow Val\ b \rightarrow Val\ [Val\ a]\\
&\qquad\qquad \rightarrow OCL\ m\ (Val\ b)\\
&iterate\ f\ \ b = pureOCL\ (foldM\ f\ \ b)
\end{aligned}
$$

Collection Operators

- The `iterate` OCL operator is almost directly translated to the *fold* recursion scheme of Haskell.

$$
\begin{aligned}
&\mathit{iterate} :: (\mathit{Val}\ b \to \mathit{Val}\ a \to \mathit{OCL}\ m\ (\mathit{Val}\ b)) \to \mathit{Val}\ b \to \mathit{Val}\ [\mathit{Val}\ a] \\
&\qquad\quad \to \mathit{OCL}\ m\ (\mathit{Val}\ b) \\
&\mathit{iterate}\ f\ b = \mathit{pureOCL}\ (\mathit{foldM}\ f\ b)
\end{aligned}
$$

- Most collection operators correspond to well-known functional programming abstractions, like *map* and *filter*.

The functional representation of OCL invariants can be automatically generated.

The functional representation of OCL invariants can be automatically generated.

But can also be directly defined in Haskell.

Conclusions

- We have explored a functional approach to support the construction of an OCL interpreter.

Conclusions

- We have explored a functional approach to support the construction of an OCL interpreter.
- The functional infrastructure can be automatically generated

What's next?

- We can provide an interpretation for many advanced OCL features proposed in the literature.

What's next?

- We can provide an interpretation for many advanced OCL features proposed in the literature.
  - Lambda abstraction for collection operators
  - Polymorphic collection types
  - Reflection
  - Safe navigation
  - Pattern Matching
  - Lazy Evaluation

What's next?

- We can provide an interpretation for many advanced OCL features proposed in the literature.
  - Lambda abstraction for collection operators
  - Polymorphic collection types
  - Reflection
  - Safe navigation
  - Pattern Matching
  - Lazy Evaluation
- We can put functional programming abstractions to work on MDE problems.

Thank you