On the Functional Interpretation of OCL

Daniel Calegari[1] and Marcos Viera[1]

Universidad de la República, Uruguay
{dcalegar,mviera}@fing.edu.uy

Abstract. The Object Constraint Language (OCL) is defined as a side-effect-free language combining model-oriented and functional features. Its interpreters are mostly focused on model-oriented features, providing a direct representation of features like inheritance and properties navigation. However, in the last few years many other functional features were proposed, e.g. pattern matching, lambda expressions and lazy evaluation. In this work we explore the use of Haskell as an alternative for the functional interpretation of OCL in a direct and clear way. We also show its feasibility through the development of a case study.

Keywords: Object Constraint Language, functional paradigm, Haskell

1 Introduction

The Object Constraint Language (OCL, [1]) plays a central role in the Model-Driven Architecture (MDA, [2]) approach. The MetaObject Facility (MOF, [3]) is the standard language proposed for metamodeling. A metamodel captures the syntax and semantics of a modeling language and thus provides the context needed for expressing well-formed constraints for models (known as the conformance relation). If there are conditions (invariants) that cannot be captured by the structural rules of this language, the OCL is used to specify them. Moreover, OCL is used for constraining and computing object values in the definition of model transformation rules. In other contexts, OCL is also used for the description of pre- and post-conditions on operations, and the specification of guards in behavioral diagrams, among many other purposes.

The OCL is defined as a side-effect-free language combining model-oriented and functional features, e.g. type inheritance and functions composition, respectively. This combination is not easily interpreted, since there is a mismatch between both worlds: a functional programming language use algebraic datatypes and functions over these datatypes rather than types, inheritance and properties navigation. In this context, OCL interpreters (e.g. Eclipse OCL [4] and Dresden OCL [5]) are mostly focused on providing a direct representation of model-oriented features, which are useful in a wider model-driven environment.

In the last few years, many authors propose the inclusion of functional features in the language, e.g. pattern matching [6], lambda expressions [7] and lazy evaluation [8]. These concepts have a direct representation in functional programming languages (e.g. Haskell [9]), and could be not easily interpreted

following a model-oriented approach. Thus, a functional approach comes as a reasonable alternative for exploiting such features.

In this work we explore the use of Haskell as an interpreter for OCL with respect to its use for expressing invariant conditions in models. We tackle with the functional representation of model-oriented features in metamodels, models and OCL expressions, as well as with the semantic interpretation of many OCL aspects, e.g. its four-valued logic with the notion of truth, undefinedness and nullity. We also discuss the practical implications of this approach through the development of a case study[1]. This work tends to provide a different perspective on the interpretation of OCL. In particular, we claim that OCL can be interpreted in a direct and clear way, such that the needed functional infrastructure can be predefined and automatically generated. Moreover, the representation of OCL as an embedding of the language into Haskell could encourage functional programmers to get closer to the model-driven approach.

Although some knowledge of (Haskell) functional programming is needed to fully understand the code presented in this paper, we believe that the main ideas and solutions we propose can be understood by non-functional experts.

The remainder of the paper is structured as follows. In Section 2 we present related work on the interpretation of OCL in many semantic domains. In Section 3 we provide a brief introduction to some of the Haskell features used in this paper. In Section 4 we introduce how metamodel and models can be represented in Haskell. Then, in Section 5 we describe the functional interpretation of the main aspects of OCL, and in Section 6 we take a step further and introduce how a functional approach provides a direct and clear interpretation for many advanced OCL features proposed in the literature. Finally, in Section 7 we present some conclusions and an outline of future work.

2 Related Work

Some authors propose the inclusion of functional aspects in the syntax and semantics of OCL, e.g. lazy evaluation [8], pattern matching [6], functions definition [10], and lambda expressions [7]. They are expected to be implemented in any OCL interpreter, without a specific functional counterpart for its interpretation. In [11] the author proposes the use of monoid calculus (a.k.a. list comprehension in functional programming languages) for the interpretation of collection operations. These aspects will be subject of study in Section 6.

There are many proposals defining the semantics of MOF and OCL in terms of a shallow embedding of the language by providing a syntactic translation into another one, e.g. rewriting logic [12,13], and first-order logic [14]. These works are focused on providing semantics and a formal environment for verification. They neither consider functional aspects nor a functional interpretation of such aspects.

[1] Complete source code of our running example is available at https://www.fing.edu.uy/inco/grupos/coal/field.php/Research/ANII14

Haskell was used for the implementation of OCL parsers and type checkers [15,16]. These works use the functional abstract syntax tree to translate the language into others, without giving a functional interpretation of the language. In [17] we propose the representation of MDA elements using Attribute Grammars which are expressed as Haskell expressions. We addressed the inclusion of OCL expressions for structural and semantic conformance checking, but we did not exhaustively study its representation.

The most related work is [18] in which the authors present a formalization using Isabelle/HOL (which can be considered a functional programming language) of a core part of OCL. This work is focused on a formal treatment of the key elements of the language rather than a complete implementation of it. It addresses many undesirable formal aspects, as those related with null and invalid values and provides detailed formal semantics for verification.

3 Haskell Preliminaries

Haskell [9] is a purely functional, lazy and static typed programming language. In what follows we briefly introduce some basic features used in this paper.

Algebraic Datatypes New types are introduced in Haskel using Algebraic Datatypes, where the shape of its belonging elements is specified. For example:

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Just}\ a \mid \mathit{Nothing}$$

where *Maybe* is a type representing the existence of an element (*Just* constructor) or nothing (*Nothing* constructor). The constructors can have parameters. In the example the constructor *Nothing* has no parameters, while *Just* receives an element of type a. We say that a type is polymorphic on the types represented by the variables occurring on the left-hand side of the definition. Thus, the type is polymorphic on the type (a) of its elements, and can be instantiated with any type, e.g.: integers (*Maybe Int*), and characters (*Maybe Char*). Constructors are used in pattern matching; for example a function that states if a maybe type has something or nothing can be defined using pattern matching as follows:

$$\begin{aligned}&\mathit{isJust} :: \mathit{Maybe}\ a \to \mathit{Bool}\\ &\mathit{isJust}\ \mathit{Nothing} = \mathit{False}\\ &\mathit{isJust}\ (\mathit{Just}\ \_) = \mathit{True}\end{aligned}$$

Type classes In Haskell type classes declare predicates over types, a type fulfills such predicate if the methods of the class are supported for this type. For example, the following definition declares a class *Monad*, with methods *return* and ($\gg\!=$), being the last one an infix operator.

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

Out of the class declaration, the types of the methods include a constraint stating the membership to the class (i.e. $return :: Monad\ m \Rightarrow a \rightarrow m\ a$). When a function uses a method of a class it inherits its constraints.

$myReturn :: Monad\ m \Rightarrow a \rightarrow m\ a$
$myReturn\ x = return\ x$

Monads Monads [19] structure computations in terms of values and sequences of (sub)computations that use these values. This provides a mechanism to structure computations in an imperative-style, allowing to incorporate side-effects and state without loosing the pure nature of the language. Haskell monads follow the interface provided by the class *Monad* presented before. Provided that a type constructor m is a monad, then a value of type $m\ a$ is a monadic computation that returns a value of type a. The function *return* is used to construct a computation from a given value. The bind function for monads ($\gg=$) defines a sequence of computations, given a computation that returns a value of type a and a function that creates a computation ($m\ b$) given a value of such type.

4 Representation of metamodels and models

The OCL is used as a supplementary language for guaranteeing the conformance relation between a model and a metamodel. A model needs to satisfy the structural rules defined by its metamodel and also the OCL invariant conditions to become a valid (well-formed) instance of the metamodel.

In what follows we introduce how metamodels and models can be represented in Haskell, providing a functional basis for the interpretation of OCL. The representation can be automated by means of a model-to-text transformation.

Basically, a metamodel defines classes which can belong to a hierarchical structure. Some of them may be defined as abstract. Any class has properties which can be attributes (named elements with an associated type: a primitive type or a class) and associations (relations between classes in which each class plays a role within the relation). Every property has a multiplicity constraining the number of elements that can be related through it, and it can be related with another (opposite) property if there is a bidirectional association.

Consider the metamodel in Figure 1 which defines UML class diagrams composed by `UMLModelElement` with a name (property `name`). Classifiers (classes and primitive types like integer) are contained in packages (property `namespace`). Classes contain attributes (property `attr`), whilst attributes have a type (property `typ`). A class contains only one or two attributes (multiplicity `1..2`), and the Classifier class is not abstract. We decided to handle these aspects differently from UML class diagrams in order to have a more complete example.

Classes and hierarchies Each class is represented as a datatype with a constructor resulting from the translation of their properties. If the class does not have a superclass, then its constructor includes a field of type *Int* representing an unique identifier of any instance of such class[2]. Moreover, if the class has

[2] This is required since Haskell data values do not have an identity

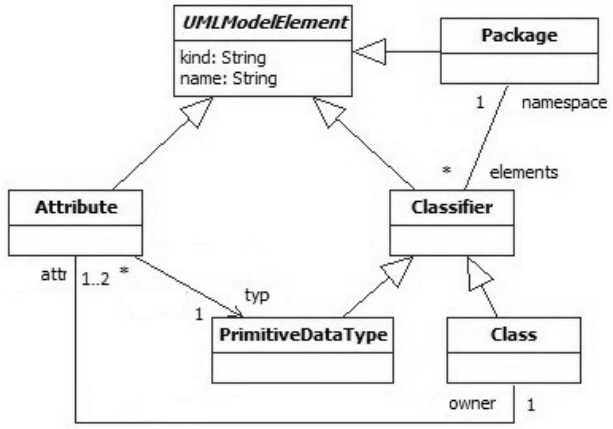

Fig. 1. UML class diagrams metamodel

subclasses, the production rule defines a field of type *ClassCh*, with *Class* the name of the class. This field defines one constructor for each subclass with its corresponding type. If the class is not abstract, then the child is wrapped with *Maybe*.

In Figure 2 we show the Haskell representation of the metamodel in Figure 1. Notice that for each class, there is a constructor for its corresponding properties. In the case of `UMLModelElement` at the top of the hierarchy, there is an *Int* field representing its identifier. Since `Classifer` is a class with non-abstract subclasses, it required the definition of *MaybeClassifierCh* and *ClassifierCh*.

Datatypes and enumerations Primitive types as string, boolean and integer are mapped to their corresponding Haskell types. In the case of user defined datatypes, they are translated as we do with classes. An enumeration is translated to a datatype with a choice of constructors corresponding to their values.

Properties and multiplicities Within the context of the constructor corresponding to the class who owns a property, we translate a property typed with a primitive type as a field of the translated type. Moreover, if the property is typed with a non-primitive type, we translate the property as a field of type *Int*, representing the identifier of the element that must be related through the property. If the multiplicity accepts many elements, the type of the field is a list of elements, and if it is 0..1, the type *Maybe* is used. More narrow multiplicities can be defined as OCL invariants as explained in Section 5.

In Figure 2 we can notice that within the constructor of `UMLModelElement` there are fields for their string properties `kind` and `name`, and in the case of `Class`, the property `atts` referencing their attributes is represented as a list of identifiers of those attributes. Since the multiplicity of `atts` is 1..2, we need to define an OCL invariant for checking it.

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

data UMLModelElementCh = UMLMECAtt  Attribute
                       | UMLMECPck Package
                       | UMLMECCla  Classifier
  -- Attribute(typ:Classifier, owner:Class)
data Attribute = Attribute Int Int
  -- Package(elements:Set(Classifier))
data Package = Package [Int]
  -- Classifier(namespace:Package) + subtypes
data Classifier = Classifier Int MaybeClassifierCh

type MaybeClassifierCh = Maybe ClassifierCh

data ClassifierCh = ClassifierChPri PrimDataType
                  | ClassifierChCla Class

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

Fig. 2. Haskell representation of the UML class diagrams metamodel

Models A model can be seen as a collection of interrelated elements. For this purpose we define a root element with a constructor having a field (of type list) for every other metamodel element on top of a hierarchy (isolated classes and datatypes are considered hierarchies of one element).

The Haskell value in Figure 3 represents a model which satisfies the structural rules of the metamodel in Figure 1. It is composed by a persistent class of name `ID` within a package of name `Package`. The class has an attribute of name `value` and type `String` which is a primitive type. The model is represented as a list of `UMLModelElement` since it is the only root element in the metamodel.

Navigation and inherited properties The above interpretation does not focus on navigation through properties (elements are represented with identifiers) and inherited properties by metamodel elements. Moreover, property names are ignored in this basic representation. These essential aspects are considered as part of the OCL interpretation in Section 5.

5 Haskell-based representation of OCL

We consider a basic but powerful subset of OCL serving as a proof of concepts. It allows representing and user defined types and values, boolean connectives (e.g. `and` and `or`), basic primitive type operators (e.g. `<` and `>` for integers), operators for equality `=` and inequality `<>`, if-then-else expressions, the built-in `allInstances`, navigation through properties (`.`) and functions over collections.

```
data Model = Model (ListUMLModelElement)
type ListUMLModelElement = [UMLModelElement]
  --
example = Model
  [UMLModelElement 1 "Persistent" "Package" (UMLMECPck $ Package [2,3])
  , UMLModelElement 2 "Persistent" "String" (UMLMECCla $ Classifier 1
    (Just $ ClassifierChPri PrimDataType))
  , UMLModelElement 3 "Persistent" "ID" (UMLMECCla $ Classifier 1
    (Just $ ClassifierChCla (Class [4])))
  , UMLModelElement 4 "Persistent" "value" (UMLMECAtt $ Attribute 2 3)
  ]
```

Fig. 3. Conforming model for the example

```
-- Inv1
context Class inv:
  self.attribute->forAll ( a1 : Attribute; a2 : Attribute |
                           a1 <> a2 implies a1.name <> a2.name)
-- Inv2
context Class inv:
  Class.allInstances()->forAll (c : Class |
    c.attribute->iterate ( a:Attribute; result:Boolean=true |
                           result and a.type.namespace = c.namespace))
-- Inv3
context Classifier inv:
  if self.oclIsTypeOf(Class) then
     self.oclAsType(Class).attribute->size() > 0
  else
     True
  endif
```

Fig. 4. OCL invariants

As an example, consider the invariants in Figure 4 over the metamodel in Figure 1: (`Inv1`) there cannot be two attributes with the same name within the same class, (`Inv2`) a class and its attributes belong to the same package, and (`Inv3`) for every classifier, if it is a class it must have at least one attribute (multiplicity constraint).

In Figure 5 we show how these invariants can be translated into Haskell. Without delving into details yet, it can be seen that the translation mimics the structure of the OCL invariants. In fact, it can be automated. This is achieved due to the functional nature of OCL and the use of Haskell features like higher-order functions, infix operators, monads and type classes.

In what follows we show the main aspects of a functional OCL library, and the translation procedure from OCL invariants to our functional settings. The library is predefined and the OCL invariants can be automatically translated.

```
  -- Inv1
chk1      = context _Class [inv1]
inv1 self = ocl self |.| atts |->| forAll (λ(Val (a1, a2)) →
  (ocl a1 |<>| ocl a2) |==>| ((ocl a1 |.| name) |<>| (ocl a2 |.| name))) ∘ cartesian
  -- Inv2
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))
  -- Inv3
chk3      = context _Classifier [inv3]
inv3 self = oclIf (ocl self |.| oclIsTypeOf _Class)
              ((ocl self |.| oclAsType _Class |.| atts |->| size) |>| ocl (Val 0))
              (ocl (Val True))
```

Fig. 5. Invariants in Haskell

We also show how to define the functional infrastructure for accessing properties and navigating through them, which can also be automated.

5.1 Functional OCL library

Invariants *chk1*, *chk2* and *chk3* have type *OCL Model* (*Val Bool*), which can be read as an OCL expression that applies to a MOF model (represented by the type *Model*) and returns a boolean value. The type *OCL m a* is a *Reader* monad, representing computations which read from a shared environment of type *m* (e.g. *Model*), and return a value of type *a* (e.g. *Val Bool*). A sequence of computations describes the navigation through properties and functions, and the shared environment (which is the model itself) can be used by the computations to look up the elements referred by others.

```
type OCL m a = Reader m a
data Val a   = Null | Inv | Val a
```

In order to represent the OCL four-valued logic with the notion of truth, undefinedness and nullity, we define the type *Val a* for OCL values. An OCL value can be null (*Null*), invalid (*Inv*) or a value (*Val*) of some type *a*. The value *Val a* allows both representing a boolean value, and any other typed value which can be useful for using OCL as a query language.

We define a specialized version of *return*, called *ocl* to construct OCL expressions from values.

```
ocl :: Val a → OCL m (Val a)
ocl = return
```

For example, in the third invariant of Figure 5 this function is used to construct the boolean and integer sub-expressions *ocl* (*Val True*) and *ocl* (*Val* 0).

We defined specialized versions of bind for better representing the object navigation operator (|.|) and the collection navigation operator (|->|).

```
infixl 8 |->|, |.|
(|.|)  :: OCL m (Val a) → (Val a → OCL m (Val b)) → OCL m (Val b)
(|.|)  = (>>=)
(|->|) :: OCL m (Val [Val a]) → (Val [Val a] → OCL m (Val b)) → OCL m (Val b)
(|->|) = (>>=)
```

In the first invariant of Figure 5, the fragment (*ocl a1* |.| *name*) defines an OCL computation that, having an *Attribute a1* and the function *name*, that given an *Attribute* defines an OCL computation that returns the name of such *Attribute*, applies *name* to *a1*. Larger sequences can be composed, like this fragment of the second invariant (*ocl a* |.| *typ* |.| *namespace*), that returns the namespace of the type of a given attribute. In this example we can notice the role of the *Reader* monad. In the representation of the metamodel (Figure 2) the elements only keep the identifiers of the other elements they are related to through properties. Then, for example, the function *typ* needs to access to the entire model in order to lookup the *Classifier* referred by the index provided by the *Attribute*. The *Reader* monad "silently" passes the given model through the sequences of computations.

The `iterate` OCL operator is almost directly translated to the *fold* recursion scheme of Haskell. We represent collections as Haskell lists.

```
iterate :: (Val b → Val a → OCL m (Val b)) → Val b → Val [Val a] → OCL m (Val b)
iterate f b = pureOCL (foldM f b)
```

Since we are dealing with monadic computations we have to use *foldM*, the monadic version of *fold*. We also need to take care of the cases were the collection we want to iterate is either an invalid or null value. This is expressed in the function *pureOCL*, which calls a given function only if a valid value is received. In other cases it just creates a computation returning an invalid or null value.

```
pureOCL :: (a → OCL m (Val b)) → Val a → OCL m (Val b)
pureOCL f (Val x) = f x
pureOCL _ Inv     = ocl Inv
pureOCL _ Null    = ocl Null
```

The rest of the collection operators can be implemented in terms of iterate. Although we decided to provide specific implementations, given that they all correspond to well-known functional programming abstractions, like *map* and *filter*. This also allows representing specific semantic aspects with respect to null and invalid values in each collection operator. For example, *collect* is almost directly translated to a monadic *map*.

```
collect :: (Val a → OCL m (Val b)) → Val [Val a] → OCL m (Val [Val b])
collect f = pureOCL (λl → mapM f l >>= ocl ∘ Val)
```

The function *context* defines a computation that verifies a list of invariants in a given context. This is done by applying all the invariants to every instance of the given context (*self*).

```
context :: (OCLModel m e, Cast m e a)
        ⇒ Val a → [Val a → OCL m (Val Bool)] → OCL m (Val Bool)
context self invs = ocl self |.| allInstances |->| forAll (mapInvs invs)
```

The function *mapInvs* applies all the invariants *invs* to a given element, and creates a computation that returns true if all of them are satisfied. The invariants (e.g. those in Figure 5) are represented as functions from a given context (which we call *self* to keep the OCL terminology) to a boolean OCL computation.

In order to be able to declare invariants, monadic versions of the operators on the basic types had to be defined for the OCL computations.

```
(|&&|) :: OCL m (Val Bool) → OCL m (Val Bool) → OCL m (Val Bool)
e1 |&&| e2 = liftM2 (&&&) e1 e2

Val False &&& _         = Val False
_         &&& Val False = Val False
Val True  &&& Val True  = Val True
_         &&& _         = Inv
```

Notice that (`&&&`) complies with the semantic rule: *False* AND-ed with anything (even invalid or null values) is *False*, satisfying a specific semantic aspect.

5.2 Accessing model elements

The following type classes provide functions to navigate through models. Instances of such classes have to be provided for any data type m, that represents a model, with top-level elements e, as shown in the Section 5.3.

```
class OCLModel m e | m → e where
  elems :: m → [e]
```

The function *elems* of the class *OCLModel*, returns a list with the elements of a given model.

```
class Cast m a b where
  downCast :: Val b → Val a → OCL m (Val b)
  upCast   :: Val a → Val b → OCL m (Val a)
```

The type class *Cast m a b* provides functions for the model m, to downcast an element of a base class (represented by the type a) to one of its derived classes (represented by the type a), and upcast an element of a derived class to one of

its supertypes. In the casting functions, the first parameter is used to determine the type to which the element has to be casted and the second parameter is the element to cast. The results is an OCL computation that returns the casted value or invalid if the cast is not possible.

The casting functions are used, for example, to implement the `oclAsType` operation, that casts an element to a given type, if possible.

```
oclAsType :: (Cast m a b, Cast m b a) ⇒ Val a → Val b → OCL m (Val a)
oclAsType t e = do c ← downCast t e
                   case c of
                     Val _ → ocl c
                     _     → upCast t e
```

We first try downcasting, if it returns a valid value the result is a computation returning this value, otherwise we return the result of upcasting (possibly an undefined value). These definitions perform specific type checkings preventing from undesirable outcomes.

To implement *allInstances*, that returns a collection with all the instances of a given class t in a given model, we first obtain the model m from the monad (using the *Reader* monad operation *ask*), then we get its list of elements and try to downcast all of them to t. Finally we return a collection with the elements that could be downcasted (i.e. downcast resulted in a valid value).

```
allInstances :: (OCLModel m e, Cast m e a) ⇒ Val a → OCL m (Val [Val a])
allInstances t = do m ← ask
                    es ← mapM (downCast t ∘ Val) (elems m)
                    return (Val [Val e | Val e ← es])
```

5.3 Defining Instances for a given Metamodel

In order to apply the OCL library in a specific metamodel, we have to define the instances of *OCLModel* and *Cast* for the datatypes that represent the metamodel. We also need to define functions to access element properties.

In order to navigate up and down in the hierarchy we used an approach inspired by the Zipper [20] structure, where we couple the element in focus and its context. In our case the context includes the information to obtain the immediate supertype. We define types to represent such "navigable elements" based in the inheritance relations described in Figure 1.

```
data Top = Top
type UMLModelElement_ = (UMLModelElement, Top)
type Attribute_       = (Attribute       , UMLModelElement_)
type Package_         = (Package         , UMLModelElement_)
type Classifier_      = (Classifier      , UMLModelElement_)
```

```
type PrimitiveDataType_ = (PrimDataType      , Classifier_)
type Class_             = (Class             , Classifier_)
```

Thus, in the instance of *OCLModel* for the UML *Model* defined in Figure 2, the type of the top-level elements is *UMLModelElement_*; i.e. navigable *UMLModelElement*s.

```
instance OCLModel Model UMLModelElement_ where
elems (Model l) = map (λe → (e, Top)) l
```

To downcast to an immediate child in this representation, we return the child coupled with the actual (navigable) element, while to upcast to an immediate supertype we only need to return the second component of the pair. We show this in the instance of *Cast* for *Classifier_* and *Class_*.

```
instance Cast Model Classifier_ Class_ where
  downCast _ (Val e @(Classifier _ (Just (ClassifierChCla x)), _)) = ocl (Val (x, e))
  downCast _ _ = ocl Inv
  upCast _ (Val (_, e)) = ocl (Val e)
```

If we try to downcast a `Classifier` element as a `Class` which is not (e.g. a `PrimitiveDatatype`), then an invalid value is returned.

When casting to classes further in the inheritance path, we use the casting functions of the inmmediate parent or child. For example in the instance for *UMLModelElement_* and *Class_*, to downcast from *UMLModelElement_* to *Class_*, we obtain the *Classifier_* and then downcast it to *Class_*.

```
instance Cast Model Classifier_ Class_ ⇒ Cast Model UMLModelElement_ Class_ where
  downCast t (Val e @(UMLModelElement _ _ _ (UMLMECCla c), Top))
             = downCast t (Val (c, e))
  downCast _ _ = ocl Inv
  upCast t (Val (_, e)) = upCast t (Val e)
```

Notice that the first parameter of the casting functions is only used to determine the type to which we want to cast, and thus, due to lazy evaluation, it is never evaluated. For all the classes of the metamodel, we define a dummy value, which can be used just to provide its type information.

```
_UMLModelElement = ⊥ :: Val UMLModelElement_
_Attribute       = ⊥ :: Val Attribute_
_Package         = ⊥ :: Val Package_
_Classifier      = ⊥ :: Val Classifier_
_Class           = ⊥ :: Val Class_
_Primitivedatatype = ⊥ :: Val PrimitiveDataType_
```

Such values have been used in Figure 5 to refer to the contexts of the invariants, and as arguments of the functions *allInstances*, *oclIsTypeOf* and *oclAsType*.

Finally, the functions to access to the properties are implemented. For example, `oid` is a property of `UMLModelElement`, thus it is a property of every class that inherits from it. This is implemented by the function *oid*, that given an element of a given type a, if this type can be upcasted to *UMLModelElement_* then the value is upcasted and the property is obtained from the returned *UMLModelElement*.

```
oid :: Cast Model UMLModelElement_ a ⇒ Val a → OCL Model (Val Int)
oid a = upCast _UMLModelElement a ≫=
        pureOCL (λ(UMLModelElement x _ _ _, _) → return (Val x))
```

In the cases where the property is a reference to another element of the model, for example the `Class` owner of an `Attribute`, the element is looked up in the model and downcasted to the desider type. This is done by the function *lookupM*, which takes a value, representing the type to which we want to downcast, and an identifier, and searchs (and downcasts) the element in the list of elements of the model obtained from the monad.

```
owner :: Cast Model Attribute_ a ⇒ Val a → OCL Model (Val Class_)
owner a = upCast _Attribute a ≫=
            pureOCL (λ(Attribute _ x, _) → lookupM _Class x)
```

Properties can return collections of elements. In this case the *lookupM* function has to be mapped to all the indices.

```
atts :: Cast Model Class_ a ⇒ Val a → OCL Model (Val [Val Attribute_])
atts a = upCast _Class a ≫=
         pureOCL (λ(Class x, _) → mapM (lookupM _Attribute) x ≫= ocl ∘ Val)
```

6 Supporting OCL advanced features

Based on the functional setting defined in Section 5 we can provide an interpretation for many advanced OCL features proposed in the literature.

The first point of discussion is according the real OCL 2.5 plans as presented in [7]. The new version of OCL tends to rewrite many aspects of the specification as well as to improve it semantic basis and language constructs. In the current specification the expressions used within collections are textual macros, e.g. `result and a.type.namespace = c.namespace` within the iteration in example (`Inv2`), which poses certain restrictions. In the new version there will be a lambda type which is in the basis of functional programming languages for denoting an anonymous function abstraction. In our example `iterate` already defines a lambda abstraction with two parameters *res* and a: *iterate* (λ*res a* → *ocl res* `|&&|` ((*ocl a*...) `|==|` (*ocl c*...))).

Collection types are currently defined using a generic type `T`. In the new version it represents a type template parameter. Haskell functions can be polymorphic based on type variables (for primitive types and other functions). Our

OCL collection functions are all polymorphic, and they are based on basic polymorphic functions, e.g. the function *foldM* used for defining *iterate* which has the following type: $foldM :: Monad\ m \Rightarrow (a \to b \to m\ a) \to a \to [b] \to m\ a$.

Reflection is of special interest, not only for the definition of OCL (e.g. for resolving `oclIsTypeOf`) but also for the discovery and manipulation of metaobjects and metadata in metamodels, as introduced in MOF. This aspect requires further study. In this sense, Template Haskell [21], i.e. an extension to Haskell that adds compile-time metaprogramming facilities, needs to be analyzed.

Syntactic sugar issues are introduced in [7] to make OCL expressions more clear. Two specific proposals are of special interest: safe navigation and pattern matching. The existence of the *null* object is troublesome since it introduces potential navigation failures. Safe navigation [22] is proposed through the safe object navigation operator `?` and the safe collection navigation operator `?->`. These operators ensure that the result is the expected value or null; no invalid failure. As an example, the operators allow `a1?.name` instead of `if a1 <> null then a1.name else null endif`. Safe navigation can be easily supported by the definition of new operators (e.g. (|?.|)) and the corresponding handling of the *Val* values, or by defining more restrictive types.

OCL pattern matching was proposed in [6] in order to provide more concise specifications based on the definition of patterns over object structures instead of the use of repeated navigation expressions. The new version of OCL does not propose full pattern matching but a special case: typesafe if, which allows reducing the number of `oclIsTypeOf/oclAsType` uses. As an example, in (`Inv3`) instead of expressing `if self.oclIsTypeOf(Class) then self.oclAsType(Class).f` we can express `if c : Class = self then c.f`. In our functional setting this notation can be defined as a new operator *oclIf*. Nevertheless, pattern matching is a basic construct in Haskell so it could be further explored to support more complex expressions.

In [10] the authors propose to extend OCL with functional abstractions (possible higher-order functions) so the language may improve its abstraction and modularity capabilities, as well as providing a collection operations definition based on primitive collection operations and recursive functions. This is straightforward in Haskell which provides means for the definition of (higher-order) functions, lambda abstractions (as explained before) and a standard library of collection functions as *fold* and *map*. Haskell also provides a `let` and `where` declaration expression constructs. As an example, recall the definition of `iterate` which is defined as a higher-order function based on *foldM*. The definition of collection operations based on the use of monoid calculus (list comprehension in Haskell), as proposed in [11], is straightforward. In fact, Haskell already provides an implementation for Set, OrderedSet, Sequence and Bag collection types.

In [10] the authors also propose to allow implicit strict downcast in OCL collection operations, e.g. using the expression `s->forAll(c:Class | ...)` instead of `s->select(oclIsTypeOf(Class))->forAll(c:Class | ...)`. As with the case of safe navigation, this can be supported by the definition of new collection operators with the corresponding handling of the *Val* values.

Finally, in [8] the authors propose a lazy evaluation strategy for OCL which could be beneficial when processing large or infinite models. Lazy evaluation is Haskell's default evaluation strategy. Moreover, lazy evaluation provides means for processing infinite structures defined through functions or list comprehension. As an example, the expression `Bag{1..}` is equivalent to [1..].

7 Conclusions & Future Work

In this paper we have explored a functional approach to support the construction of an OCL interpreter. We have presented a Haskell-based representation of the main aspects of OCL and we have discussed how it provides a direct and clear interpretation for advanced OCL features proposed in the literature.

Although the functional infrastructure could be not easily readable for a inexperienced user, it can be predefined (the whole OCL library) and automatically generated (e.g. operations for accessing model properties). There is also a direct representation of the OCL invariants mimicking its structure. Moreover, the —Reader— monad allows a clean handling of errors and a precise definition of the four-valued semantics of OCL.

This approach is exploratory, thus it still needs further work to be put into real action. We need to continue developing the OCL library, e.g. considering other kind of collections and primitive types, and to fine tuning its semantics. In particular, it could be desirable to examine the relation between our definitions and the Isabelle/HOL-based semantics defined in [18]. Some OCL aspects could be not easily represented (e.g. tuples without a fixed length and heterogeneous collections) so they deserve further analysis. We also need to consider other OCL uses, e.g. for expressing pre- and post-conditions on operations, and to study the use of Template Haskell [21] for providing metaprogramming capabilities.

From a practical point of view, we need to focus on parsing and type checking issues. In particular, we are working on the automatic transformation of models, metamodels and OCL invariants into Haskell and its connection with the Eclipse Modeling Framework for simplifying these aspects. A benchmark comparison between our interpreter and others could also be of interest, since lazy evaluation has a main drawback which is that memory usage becomes hard to predict. Moreover, it will be desirable to essay an extension of the —Reader— monad in order to capture more expressive error messages.

Finally, we should evaluate the use of this approach together with model transformations in three directions: considering an extension of OCL for expressing model transformations, as in [23], continuing our previous work [17] on the use of Attribute Grammars for the same purpose, and exploring the functional definition of bidirectional model transformations.

Acknowledgements

This work has been partially funded by the Agencia Nacional de Investigación e Innovación (ANII, Uruguay).

References

1. OMG: Object Constraint Language. Spec. V2.4, Object Management Group (2014)
2. OMG: Model Driven Architecture. Spec. V2.0, Object Management Group (2014)
3. OMG: Meta Object Facility. Spec. V2.0, Object Management Group (2003)
4. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison-Wesley Professional, Boston, Massachusetts (2008)
5. Wilke, C., Thiele, M., Freitag, B.: Dresden OCL - manual for installation use and development. Technical report, TU Dresden (2009-2011)
6. Clark, T.: OCL pattern matching. In: Proc. OCL Workshop. Volume 1092 of CEUR Workshop Proceedings. (2013) 33–42
7. Willink, E.: Ocl 2.5 plans. Presentation in the 14th Intl. Workshop on OCL and Textual Modelling, 2014.
8. Tisi, M., Douence, R., Wagelaar, D.: Lazy evaluation for OCL. In: Proc. 15th Intl. Workshop on OCL and Textual Modeling. Volume 1512 of CEUR Workshop Proceedings. (2015) 46–61
9. Jones, S.P., ed.: Haskell 98 Language and Libraries: The Revised Report. http://haskell.org/ (September 2002)
10. Brucker, A.D., Clark, T., Dania, C., Georg, G., Gogolla, M., Jouault, F., Teniente, E., Wolff, B.: Panel discussion: Proposals for improving OCL. In: Proc. of 14th Intl. Workshop on OCL and Textual Modelling. Volume 1285 of CEUR Workshop Proceedings. (2014) 83–99
11. Garcia, M.: Efficient integrity checking for essential MOF + OCL in software repositories. Journal of Object Technology **7**(6) (2008) 101–119
12. Boronat, A., Meseguer, J.: Algebraic semantics of OCL-constrained metamodel specifications. In: TOOLS (47). Volume 33 of LNBIP., Springer (2009) 96–115
13. Rivera, J.E., Durán, F., Vallecillo, A.: Formal specification and analysis of domain specific models using Maude. Simulation **85**(11-12) (2009) 778–792
14. Shan, L., Zhu, H.: Semantics of metamodels in UML. In: TASE, IEEE Computer Society (2009) 55–62
15. Liduan, F.: Design and implementation of a uml/ocl compiler. Master thesis, Utrecht University (2004)
16. Burke, D.A., Johannisson, K.: Translating formal software specifications to natural language. In: Proc. 5th Intl. Conf. Logical Aspects of Computational Linguistics. Volume 3492 of LNCS., Springer (2005) 51–66
17. Calegari, D., Viera, M.: Model-driven engineering based on attribute grammars. In: Proc. 19th Brazilian Symposium Programming Languages. Volume 9325 of LNCS., Springer (2015) 112–127
18. Brucker, A.D., Tuong, F., Wolff, B.: Featherweight OCL: A proposal for a machine-checked formal semantics for OCL 2.5. Archive of Formal Proofs **2014** (2014)
19. Wadler, P.: Monads for functional programming. In: 1st Intl. School on Adv. Functional Programming Techniques, London, UK, Springer-Verlag (1995) 24–52
20. Huet, G.: The zipper. J. Funct. Program. **7**(5) (September 1997) 549–554
21. Sheard, T., Jones, S.L.P.: Template meta-programming for haskell. SIGPLAN Notices **37**(12) (2002) 60–75
22. Willink, E.D.: Safe navigation in OCL. In: Proc. 15th Intl. Workshop on OCL and Textual Modeling. Volume 1512 of CEUR Workshop Proceedings. (2015) 81–88
23. Jouault, F., Beaudoux, O., Brun, M., Clavreul, M., Savaton, G.: Towards functional model transformations with OCL. In: Proc. 8th Intl. Conf. Theory and Practice of Model Transformations. Volume 9152 of LNCS., Springer (2015) 111–120