Experimenting with Functional Features of the Object Constraint Language*

Daniel Calegari and Marcos Viera

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

Abstract. Although the Object Constraint Language (OCL) was significantly influenced by functional programming languages, most of its interpreters are based on the object-oriented paradigm, providing direct representation for model-oriented features like inheritance. In previous works, we introduced a Haskell-based sandbox providing a functional interpretation of OCL invariants. In this paper, we use this sandbox for experimenting with functional features proposed in the literature for OCL, showing the benefits of a functional interpretation and its limitations.

Keywords: Object Constraint Language, functional paradigm, Haskell

1 Introduction

The Object Constraint Language (OCL, [1]) However, it also supports for some functional features, e.g., functions composition, and many authors have proposed the inclusion of other ones, e.g., pattern matching [2], which have a direct representation in functional programming languages such as Haskell [3].

In previous works [4,5] we introduced a sandbox for experimentation with the interpretation of OCL invariants, tackling with the functional representation of model-oriented and functional features in metamodels, models and OCL expressions. We also introduced Haskell OCL[1], an Eclipse OCL-based tool, providing tool support for experimenting novel approaches from the functional perspective.

In this paper, we discuss advances in the experimentation with functional features proposed by the scientific community. We focus on OCL as an embedded domain-specific language (EDSL) in Haskell, to open the language to the functional programming community, to experiment with it. This means that we describe how OCL interpretation benefits from such an encoding and its corresponding limitations, neither focusing on proposing new features to the language itself nor on making any comparison at the level of technological support for it.

The rest of this paper is organized as follows. In Section 2 we introduce basic aspects of the functional paradigm used in the rest of the article, and in Section 3 we briefly present the sandbox and the functional encoding of OCL. In Section

* Copyright ©2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1] Haskell OCL: `https://gitlab.fing.edu.uy/open-coal/haskellOCL`

4 we use this encoding to evaluate the main advanced features we have identified, and in Section 5, we complement the analysis with other features and open issues. Finally, in Section 6 we present some conclusions and ideas for future work.

2 Haskell Preliminaries

Haskell [3] is a purely functional, lazy and static typed programming language.

Algebraic Datatypes introduce new types with a constructor for each element:

```
data Maybe a = Just a | Nothing
```

where `Maybe a` is a type representing the existence of an element of type `a` (`Just` constructor) or nothing (`Nothing` constructor). The constructors can have parameters; e.g., `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. `Maybe` is polymorphic on the type (`a`) of its elements, thus it can be instantiated with any type; e.g., integers (`Maybe Int`) and characters (`Maybe Char`). Constructors are used in pattern matching, e.g., a function stating if a maybe type has something or nothing is:

```
isJust :: Maybe a -> Bool
isJust Nothing  = False
isJust (Just _) = True
```

Type classes declare predicates over types; e.g., a class `Monad`, with methods `return` and (>>=), being the last one an infix operator:

```
class Monad m where
  return   :: a -> m a
  (>>=)    :: m a -> (a -> m b) -> m b
```

A type fulfills such predicate if the methods of the class are supported for this type. Out of the class declaration, the types of the methods include a constraint stating the membership to the class (i.e., `return :: Monad m => a -> m a`). When a function uses a method of a class it inherits its constraints, e.g.,

```
myReturn :: Monad m => a -> m a
myReturn x = return x
```

Monads structure computations in terms of values and sequences of (sub) computations that use these values (an imperative-style), allowing to incorporate side-effects and states without losing the pure nature of the language. Haskell monads follow the interface provided by the class `Monad` introduced before. If 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 (>>=) 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.

3 Functional Interpretation of OCL

Our sandbox provides a Haskell-based interpretation for OCL [4], and an Eclipse OCL-based tool (Haskell OCL) supporting such an approach [5]. The tool uses Eclipse OCL for modeling a metamodel together with their corresponding OCL invariants, and a model in which the invariants must be checked, it generates a Haskell representation of these elements through a model-to-text transformation, and runs the Haskell code for checking the OCL invariants, which uses a predefined and metamodel-independent functional OCL library.

The output Haskell file is available to experiment directly with it, and since OCL is defined as an EDSL, in this work we focus on the Haskell encoding, its potential, and limitations. In this context, Eclipse OCL is used only as a reference of value for the community in terms of coverage of basic language constructions. In what follows we present the basics of the OCL functional encoding.

3.1 Representation of Metamodels & Models

Each class of a metamodel is represented as a Haskell datatype with a constructor resulting from the translation of their properties (attributes and associations). The primitive type of an attribute is represented with its corresponding Haskell type, and associations are represented with an `Int` parameter (or list of integers) identifying the referenced elements. If a `Class` has subclasses, a field of type `ClassChild` is added, defining one constructor (`ClassCh`) for each subclass (`Class`) with its corresponding type. For non-abstract classes this field is wrapped with a `Maybe` type. We also define a class `ModelElement`, which is the superclass of all the orphan classes, defining a unique identifier other elements can refer to. A model is thus represented as a list of `ModelElement` values.

The metamodel depicted in Figure 1 (from [6]) represents teams and team meetings, such that each meeting has a certain number of participants and a moderator from the same team. Its functional representation is depicted in Figure 2. There is a root `ModelElement` from which a `Person` (and the other orphans) inherits. A `Person`, and its `Teammember` subclass, are represented as types together with their properties (associations are represented with lists of integers). Since not every `Person` is a team member, its last parameter is wrapped with a `Maybe`. There is also an access function to the property `name` from an element of type `Person` or any of its subclasses. The function goes upwards (`upCast`) in the hierarchy until it finds a `Person`, and then returns the value representing the name.

3.2 Representation of OCL Invariants

We defined `OCL m a`, a `Reader` monad representing computations in a shared environment of type `m` (the model), that returns a value `a` with respect to the OCL four-valued logic with the notion of truth, undefinedness and nullity.

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

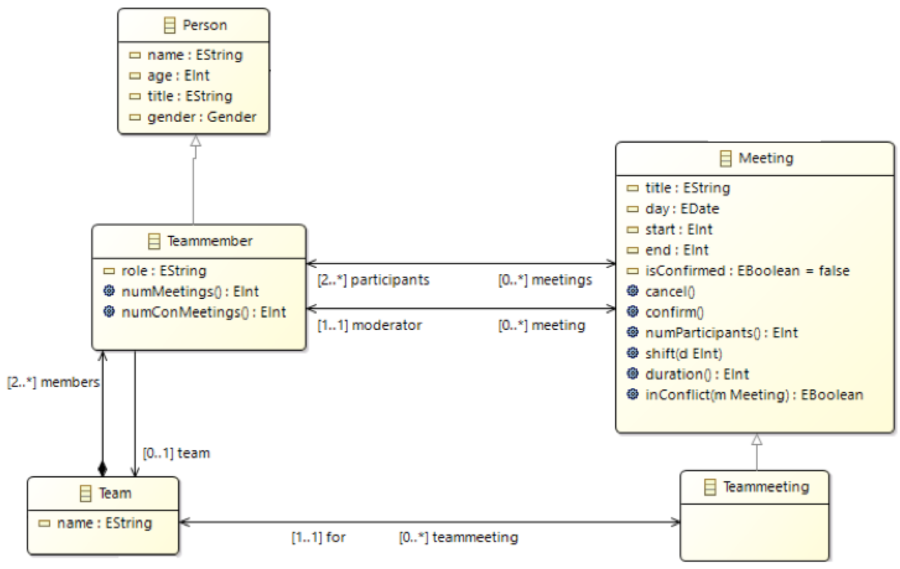

Fig. 1. Team Meeting metamodel

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. We defined specialized versions of bind (>>=) for better representing the object navigation operator (`|.|`) and the collection navigation operator (`|->|`). Primitive types and collection types have their corresponding Haskell representation. Collection operators were defined in terms of `iterate` which is almost directly translated to the `fold` recursion scheme in Haskell (a monadic version of it), e.g.,

```
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)

pureOCL f (Val x) = f x
pureOCL _ Inv     = oclInv
pureOCL _ Null    = oclNull
```

A function `context` defines a boolean computation that verifies a list of invariants in a 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)
```

```
data ModelElement      = ModelElement Int ModelElementChild
data ModelElementChild = PersonCh  Person | ...
data Person            = Person String Int String String (Maybe PersonCh)
data PersonCh          = TeammemberCh Teammember
data Teammember        = Teammember String [Int] [Int] [Int]
...
name :: Cast Model Person_ a => Val a -> OCL Model (Val String)
name a = upCast _Person a >>= pureOCL (\(Person x _ _ _ _) -> return (Val x))
```

Fig. 2. Haskell representation of the Team Meeting metamodel

In Figure 3 there is an OCL invariant and its Haskell OCL version. It specifies that a team meeting has to be organized for a whole team.

```
-- context Teammeeting inv:
--     self.participants -> forAll(team = self.for)

invariant = context _TeamMeeting [inv]
inv self = ocl self |.| participants |->|
           forAll (\a -> ocl a |.| team |==| ocl self |.| for)
```

Fig. 3. Team Meeting invariant

Some type classes of our library provide functions to navigate through models, and instances of such classes have to be provided for any data type representing a model element within a given metamodel. Boilerplate code is defined to navigate through a model uniformly, e.g., to implement the `oclAsType` operation.

As summarized in Table 1, our Haskell OCL proposal currently provides an almost complete representation of OCL for invariants and queries. We take Eclipse Oxygen OCL 6.3 as a reference and use a semaphore-like notation, where: green, yellow and red means that the OCL aspect is fully, partially or not supported, respectively. Eclipse OCL supports parsing of other OCL aspects as pre/postconditions but it does not provides any interpretation for them since it is focused on Essential OCL (as we are), which provides the core capabilities for expressing invariants on models. Our biggest limitation is not the support of OCL itself, but the support of other constructs not focused (or not commonly used) on metamodels, e.g., operations on types and association classes.

4 Interpreting Advanced Features

The functional paradigm provides a purely functional, declarative and concurrent programming environment with static typing and type inference. These features are strongly connected with the OCL language. In what follows we analyze three main functional features proposed to be incorporated into OCL.

Table 1. OCL compliance (excerpt from [5])

	Haskell OCL	**Eclipse OCL**
OCL Constructs & Expressions		
`context` / `inv` (invariant condition and its context)	✓	✓
`init` (initial value of an attribute or association role)	✓	✓
`derive` (derived attribute or association role)	✗	✓
`def` (new attribute or query operation)	✓	✓
`package` (package to which OCL expressions belong)	~[a]	✓
`self` (contextual instance)	✓	✓
`if-then-else` / `let-in` expressions	✓	✓
Navigation (through attributes, association ends, etc.)	~[b]	✓
OCL Standard Library (types and operations)		
`Boolean/Integer/Real/String` types	✓	✓
`UnlimitedNatural` type	✗	✓
`OCLAny` type (supertype of all OCL types)	~[c]	~
`OCLVoid` type (one single instance `undefined`)	~[d]	✓
`Tuple` type	✓	✓
`Collection` types (`Set`, `OrderedSet`, `Bag`, and `Sequence`)	✓	✓
`Collection` operations	~[e]	✓

[a] Syntactic sugar, can be easily supported.
[b] Association classes and qualified associations are not supported.
[c] Not defined, but its operations are implemented for any type; `oclIsInState`, `oclIsNew`, `oclType` are not supported (only the last one in Eclipse OCL).
[d] Not defined as a type, but `undefined` considered as a value during evaluation.
[e] `flatten`, `sortedBy` and `collectNested` are not supported.

4.1 Functions Everywhere

Functions are first-class citizens in Haskell. It allows higher-order functions taking other functions as arguments or return types. OCL defines something like functions (e.g., when defining a let expression) with some restrictions since it is not possible to define higher-order functions, except in the case of collection operations. In [7] the authors take these problems and propose to extend OCL to improve the language abstraction and modularity capabilities, as well as providing collection operations based on primitive collection operators and recursive functions. Our sandbox completely addresses this aspect. As an example, collection operators already use lambda abstractions, e.g., in Figure 3 the invariant takes a parameter `a` to apply the expression of the `forAll` (`\a −>` ...) . This allows to define functions in terms of other functions, as is the case of `reject` in Figure 4, which applies the `select` operation negating the expression condition `p` by using the operator `notOCL` (the boolean operator also considering invalid cases). This can be extended for any OCL expression, e.g., those in a `let−in` context, as in the first invariant in Figure 4.

```
reject :: (Val a -> OCL m (Val Bool)) -> Val (Collection a)
                                      -> OCL m (Val (Collection a))
reject p = select (notOCL . p)

-- context Meeting inv:
-- let noConflict : Boolean = participants.meetings->forAll(m || m<> self and
--                                 m.isConfirmed implies not self.noConflict(m))
-- in isConfirmed implies noConflict

invariant = context _Meeting [inv]
inv self = let noConflict = ocl self |.| participants |.| meetings ...
           in ocl self |.| isConfirmed |==>| noConflict

-- context Meeting inv:
-- let priority(Set(Teammember)) : Teammember = ... ,
--     getModerator(m:Meeting, p:(Set(Teammember) -> Teammember)) : Teammember
--                                                    = p(m.participants)
-- in self.moderator = getModerator(self,priority)

invariant = context _Meeting [inv]
inv self = let priority = ...
               getModerator m p = p (m |.| participants)
           in ocl self |.| moderator |==| getModerator (ocl self) priority
```

Fig. 4. Functional abstractions examples

We can also support higher-order functions, e.g., `iterate` in Section 3, which is defined as a higher-order function based on `foldM`. Assume, by abuse of notation, that OCL can express the second invariant in Figure 4, stating that the moderator is the one with highest priority. In such case, we can define a function `priority` for selecting the member with highest priority, and a higher-order function `getModerator` that takes a `Meeting`, a priority function and returns a `Teammember`. This representation is nowadays supported in our sandbox.

Even though the use of functions benefits OCL, there are some problematic aspects. For example, the function `flatten`, which returns a collection containing all elements of self recursively flattened, is not easily supported when there is more than one level of recursion, or there are elements in the collection with a different type. A recursive definition of this function can be defined with type `flatten :: Collection (Collection a) -> Collection b`. However, it has a typing problem since it is not possible to define a generic relation between types `a` and `b`. To find a solution, we need to add more information within the representation of metamodels and to generate boilerplate code, which deserves further analysis. A derivation of this problem is that since the operation `collect` is based on `flatten`, we provided a limited version in which the OCL expression is applied to every element in the collection but not recursively on collections of collections.

4.2 Lazy Evaluation

Lazy evaluation is Haskell's default evaluation strategy. Since lazy functions do not evaluate their arguments until their values are needed, lazy evaluation for OCL [8] was proposed as a way of optimizing queries, in particular when dealing with large, or even infinite, models. The authors focused on achieving three objectives: a performance increase delaying the access to source model elements when needed, enabling the use of infinite data structures, and improve the reusability of OCL libraries. Let us say that the third aspect is not only related to laziness but also functional abstractions and composition, as analyzed before. The other two aspects truly benefit from a lazy evaluation. As an example, take the definition of `allInstances` which implies returning every element of a given type. Its strict semantics prevent processing infinite models, e.g., in the example in Figure 5 the first `allInstances` would never terminate on a state machine with infinite states. A lazy evaluation semantics allows the query to terminate if a non-final state containing a self-transition was found (and there is no invalid evaluation [9]). This is what happens by default in our sandbox (also in Figure 5), even when using monads.

```
-- State.allInstances()->select(s | not s.kind = 'final')
--                          ->exists(s | s.outgoing->exists(t | t.target = s))

invariant = context _State [inv]
inv self = ocl self |.| allInstances |->| select (\s -> ... )
                                     |->| exists |->| (\s -> ...)
```

Fig. 5. Lazy evaluation example

Although laziness can improve performance, it also adds memory overhead since the compiler has to record the expression in the heap in case it is evaluated later. Haskell provides strictness analysis and explicit strict evaluation features, which may improve this aspect. Further studies are required in this way.

4.3 Pattern Matching

OCL pattern matching [2] could provide more concise specifications based on the definition of patterns over object structures instead of the use of repeated navigation expressions. The authors do not propose full pattern matching but a special case: typesafe if, which allows reducing the number of `oclIsTypeOf/oclAsType` uses, e.g., instead of `if self.oclIsTypeOf(Class) then self.oclAsType(Class).f` we can express `if c:Class = self then c.f`. This notation can be defined as a new Haskell operator `oclIf`. Pattern matching is a basic construct in Haskell so it could be further explored to support more complex expressions. We indeed use pattern matching, as depicted in Figure 2 for accessing the property `name`. Unlike functional pattern matching, in which we access parameters of a constructor by their position, in OCL we want to access properties through their names.

Inspired by the syntax in [2], we show how a more complete and simple pattern matching extension to OCL can be supported. The purpose here is to provide better basics for the support of the original proposal, and not to suggest just another syntax. In Figure 6 we show an example of the use of this extension and its translation to our library. The semantics is the following: in case the `Person` is a `Teammember` with role “Chief”, then he or she has to be an “Engineer” older than 23; in the case is not a `Teammember` or is not a “Chief” but has title “Engineer”, then he or she has to be older than 22; otherwise, the `Person` has to be older than 18. Note that although it is a bit more verbose, the Haskell structure matches directly with the original OCL.

```
--  context Person inv:
--     case self {
--        (Teammeber)[age = a,  role = "Chief",  title = t] -> (a > 23) || t == "Engineer";
--        (Person)[age = a,  title = "Engineer"] -> a > 22;
--        (Person)[age = a]                       -> a > 18
--     }

invariant = context _Person [inv]
inv self = ocl self |.|
  ((caseOCL (_Teammember <::> age <:> role <=> "Chief" <:> title <:> NilP)
            (\(a,(t,())) -> (oclVal a |>| oclInt 23) ||||
                            (oclVal t |==| oclVal "Engineer") )) <||>
   (caseOCL (_Person <::> age <:> title <=> "Engineer" <:> NilP)
            (\(a,()) -> oclVal a |>| oclInt 22))               <||>
   (caseOCL (_Person <::> age  <:> NilP)
            (\(a,()) -> oclVal a |>| oclInt 18)) )
```

Fig. 6. Pattern Matching example

The implementation of the whole pattern matching mechanism is very simple. We first define a couple of datatypes to represent patterns:

```
data Pattern m e p as where
  Pattern :: Val p -> PList m e p as -> Pattern m e p as

data PList m e p as where
  NilP   :: PList m e p ()
  VarP   :: (Cast m e p => Val e -> OCL m (Val a)) -> PList m e p as
              -> PList m e p (a,as)
  LitP   :: Eq a => (Cast m e p => Val e -> OCL m (Val a)) -> a
              -> PList m e p as -> PList m e p as
```

The combinators `<::>`, `<:>` and `<=>` used in Figure 6 are just smart constructors of such types. For example, the first pattern is equivalent to:

```
   (Pattern _Teammember (VarP age (LitP role "Chief" (VarP title NilP))))
```

A pattern consists of a reference to a class (`Val p`) and a list of *sub-patterns* referring to the attributes of the class. A sub-pattern can bind the matched element to a variable (`VarP`) or compare it to a literal (`LitP`). We could have defined sub-patterns recursively as patterns, but for simplicity we decided to have only two-level patterns. Notice that the `as` index of the types is increased every time a `VarP` is included, constructing a nested cartesian product, that ends with () at `NilP` (e.g., the type `as` for our example pattern is `(Int,(String,()))`). The first parameter of both `VarP` and `LitP` is a function that knows how to extract some information from an element representing a class `e` that can be casted to the referenced class `p`. Thus, `age`, `role` and `title` in Figure 6 are attribute accessor functions defined in the same way we defined `name` in Figure 2.

The function `caseOCL` implements a case branch. It takes a pattern `p`, a function `f` that takes the values bound by the pattern and returns an OCL computation (with a value of type `Val a`), and the value `self` to inspect, and returns the OCL computation. It basically evaluates the pattern to produce the cartesian product needed to apply to the function `f`. If the pattern fails it results in `Inv`.

```
caseOCL :: Cast m e p => Pattern m e p as -> (as -> OCL m (Val a))
                                          -> Val e -> OCL m (Val a)
caseOCL p f self = evalPattern p self >>= pureOCL f
```

Pattern evaluation needs to downcast the given element to the class the pattern refers. If it succeed, we go through the list of sub-patterns, applying the access function to the elements and constructing the cartesian product in the case of `VarP`, or checking the match in the case of `LitP` (returning `Inv` if it fails).

```
evalPattern :: Cast m e p => Pattern m e p as -> Val e -> OCL m (Val as)
evalPattern (Pattern p plist) e = do p' <- downCast p e
                                     case p' of Inv -> return Inv
                                                _   -> evalPList plist e

evalPList :: Cast m e p => PList m e p as -> Val e -> OCL m (Val as)
evalPList NilP _ = return $ Val ()
evalPList (VarP f   ps) e = f e >>= pureOCL (\x -> evalPList ps e >>=
                                    pureOCL (\xs -> return (Val (x,xs))))
evalPList (LitP f l ps) e = f e >>= pureOCL (\x -> if x == l
                                     then evalPList ps e else return Inv)
```

5 Other OCL Features

Packages and Package Invariants. Haskell already provides modules for packaging definitions and the use of qualified names. In Figure 7 there is a module `UML` defining the metamodel and there is a qualified import for expressing invariants in another module `OCL`. Moreover, in [10] the authors propose to define invariants for packages, e.g., `package UML inv: forAll(m in Meeting | ...)`, avoiding the need of expressing the OCL expression for every element of some type, e.g., `context Meeting inv: Meeting.allInstances()->forAll(...)`. As expressed in Figure 7, it does not change the way an expression is defined.

Generic Collection Types. Haskell functions can be polymorphic based on type variables. As proposed in [10], our OCL collection functions are all polymorphic, and based on polymorphic functions, e.g., `foldM` used for defining `iterate`. As depicted in Figure 7, we have defined a polymorphic `Collection` type with four constructors (i.e., Bag, Set, Sequence and OrderedSet), and collection operators over this type. Their behavior is determined depending on the type of collection using pattern matching, e.g., `select`. As analyzed in Section 4.1, many problems arise from hierarchical typing.

Safe Navigation. The existence of the `null` object is troublesome since it introduces potential navigation failures. Safe navigation is proposed in [11] 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. In our sandbox, safe navigation can be supported by the definition of new operators, e.g., (`|?.|`)) in Figure 7 (`pureOCL` is defined in Section 3.2).

```
-- Packages and Package Invariants
module UML where data Meeting = ...

module OCL where import qualified UML as UML

context UML._Meeting [inv]
inv self = ocl self |.| allInstances |->| forAll ...

-- Generic Collection Types
data Collection a = Bag [Val a] | Sequence [Val a] | ...

select :: (Val a -> OCL m (Val Bool)) -> Val (Collection a)
                                      -> OCL m (Val (Collection a))
select p (Val (Bag xs))    = ...

-- Safe Navigation
(|?.|) :: OCL m (Val a) -> (Val a -> OCL m (Val b)) -> OCL m (Val b)
x |?.| f = x >>= pureOCL f
```

Fig. 7. Other OCL Features

Exploiting Monads. As introduced in Section 3.2, our sandbox uses a `Reader` monad which "silently" passes a given model through the sequences of computations. The context definition captures the boolean result of the evaluation of an invariant. However, the monad do not depends on a concrete value type, thus monadic computations allow any return type, not just booleans, e.g., `iterate` (also in Section 3.2) is defined for a collection containing any type. This lifts OCL to a navigation language, which could be useful for supporting operations and model transformations, among other uses.

By exploiting monads, we can get other interesting features. In [12] the authors address several shortcomings of the OCL language (many tools already provide language extensions for dealing with them, e.g., Eclipse OCL), such as it does not support specifying user messages and there is no support for repairing inconsistencies in a model. We can use an `Error` monad, which represents computations which may fail or throw exceptions, as presented in Figure 8. In such example, we use the `ErrorT` monad transformer that adds error handling to another monad (our original `Reader` monad) and allows throwing errors on any computation, e.g., when an invariant fails. It could also be possible to use a `State` monad for consuming a state (e.g., a given model) and produce both a result (e.g., an invariant check) and an updated state (e.g., a repaired model). In summary, the use of combined monads allows adding effects to OCL expressions in a modular way.

```
type OCLError m a = ErrorT String (OCL m a)

invariant1 = context _Meeting [inv2]
inv2 self = do res <- ocl self |.| participants ...
               if res |==| oclVal True
               then return res
               else throwError "There␣was␣an␣error"
```

Fig. 8. Using the `ErrorT` monad transformer

Operations & Pre/postconditions. The navigation language we provide can be used to the derivation of properties in terms of other properties or the specification of the body of an operation that does not change the state of the system. In Figure 9 there is an example of a derived property (`size`) whose value (the size of the team) is computed when needed, and the body of an operation (`getMeetingTitles`) which returns the titles of the meetings that a team member attends. Both aspects can be currently supported as any other property introduced in Section 3. Pre/postconditions support rely on the same settings but with some additional features. In particular, we must consider a pair or `pre` and `post` models where the pre/postconditions of an operation specification must hold. As an example, take the pre/postconditions of the operation `shift` in Figure 9. Supporting preconditions is straightforward, since they are invariants that must hold in the `pre` model. In the case of a postcondition, it is also an invariant check, but we need to define a new context in which we can shift from one model to another by using a function `atPre` in order to get the value of any expression marked with `@pre` from the `pre` model. The example in this last case is just explanatory since it requires further development.

6 Conclusions & Future Work

In this paper, we discussed advances in the experimentation with functional features proposed for OCL and provided a different perspective on the interpretation of OCL as a Haskell EDSL.

```
-- context Team::size:Integer
-- derive: members->size()

size :: Cast Model Team_ a => Val a -> OCL Model (Val Int)
size self = (ocl self) |.| members |->| size

-- context Teammember::getMeetingTitles():Bag(String)
-- body: meetings->collect(title)

getMeetingTitles :: Cast Model Teammember_ a => Val a
                                 -> OCL Model (Val (Bag String))
getMeetingTitles self = (ocl self) |.| meetings |->|
                                      collect(\m -> ((ocl m) |.| title))

-- context Meeting::shift(d:Integer)
-- pre: self.isConfirmed = false and d > 0
-- post: start = start@pre + d and end = end@pre + d

context _Teammeeting [inv]
inv self d = (((ocl self) |.| isConfirmed) |==| (oclVal False)) |&&|
               ((oclInt d) |>| (oclInt 0))

post self d = (((ocl self) |.| start)) |==|
                 (ocl self) |.| atPre start) |+| (oclInt d)) |&&|
              (((ocl self) |.| end)) |==|
                 (ocl self) |.| atPre end) |+| (oclInt d))
```

Fig. 9. Operations & Pre/postconditions

Hierarchical typing and identities introduce the main mismatch problems. Once some functional boilerplate is generated to take care of this, the resulting expressions and their interpretation are more modular, abstract and extensible. Nevertheless, there are still some challenges, e.g., the definition of recursive multi-typed collection operators and reflection capabilities, as with other open issues, as discussed in Section 5, that require further studies.

To our knowledge, the most related work to ours is Sigma [13], an OCL EDSL implemented in Scala. Since Scala is both functional and object-oriented, their embedding does not have to deal with the hierarchy representation mismatch problem, and its syntax looks closer to OCL than ours. However, Sigma OCL expressions are not effect-free, and formal reasoning is much more difficult than in our purely functional approach. Another shallow embedding of OCL, in this case into Isabelle/HOL, is presented in [14]. It is intended to be a proposal for the standardization process of OCL 2.5, and it deserves further analysis to examine the relationship between our definitions and it.

Considering that in some cases OCL could be benefited from the introduction of some kind of controlled side effects, e.g., for expressing error messages, an interesting research line could be to analyze how to define a modular effects mechanism for OCL, e.g., inspired by monads and monad transformers.

As a complementary perspective, we are currently addressing a benchmark comparison between Haskell OCL and Eclipse OCL, dealing with performance concerns (time and memory) on some (larger) real-world examples.

Acknowledgements

This work was partially funded by Fondo Carlos Vaz Ferreira 2017, Dirección de Innovación, Ciencia y Tecnología (DICYT), MEC, Uruguay.

References

1. OMG: Object Constraint Language. Spec. V2.4, Object Management Group (2014)
2. Clark, T.: OCL pattern matching. In: Proc. OCL Workshop. Volume 1092 of CEUR Workshop Proceedings., CEUR-WS.org (2013) 33–42
3. Jones, S.P., ed.: Haskell 98 Language and Libraries: The Revised Report. http://haskell.org/ (September 2002)
4. Calegari, D., Viera, M.: On the functional interpretation of OCL. In: Proc. of the 16th Intl. Workshop on OCL and Textual Modelling. Volume 1756 of CEUR Workshop Proceedings., CEUR-WS.org (2016) 33–48
5. Sintas, G., Lutz, L.V., Calegari, D., Viera, M.: Model-driven development of an interpreter for the object constraint language. In: XLIV Latin American Computer Conference CLEI, IEEE (2018) 120–128
6. Demuth, B.: OCL (Object Constraint Language) by example. Lecture at MINE Summer School (2009)
7. 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., CEUR-WS.org (2014) 83–99
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., CEUR-WS.org (2015) 46–61
9. Willink, E.D.: Deterministic lazy mutable OCL collections. In: Proc. STAF 2017 Collocated Workshops. Volume 10748 of LNCS., Springer (2018) 340–355
10. Willink, E.: Ocl 2.5 plans. Presentation in the 14th Intl. Workshop on OCL and Textual Modelling, 2014.
11. Willink, E.D.: Safe navigation in OCL. In: Proc. 15th Intl. Workshop on OCL and Textual Modeling. Volume 1512 of CEUR Workshop Proceedings., CEUR-WS.org (2015) 81–88
12. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: On the evolution of OCL for capturing structural constraints in modelling languages. In: Rigorous Methods for Software Construction and Analysis. Volume 5115 of LNCS., Springer (2009) 204–218
13. Krikava, F., Collet, P.: On the use of an internal DSL for enriching EMF models. In: Proc. of 12th Workshop on OCL and Textual Modelling, ACM (2012) 25–30
14. 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)