Extending OCL with Map and Function Types

Kevin Lano, Shekoufeh Kolahdouz-Rahimi

Extending OCL with Map and Function Types

K. Lano[1] and S. Kolahdouz-Rahimi[2]

[1] Dept. of Informatics, King's College London
kevin.lano@kcl.ac.uk
[2] Dept. of Software Engineering, University of Isfahan, Iran
sh.rahimi@eng.ui.ac.ir

Abstract. Map and function types are of high utility in software specification and design, for example, maps can be used to represent configurations or caches, whilst function values can be used to enable genericity and reuse in a specification, and to support mechanisms such as callbacks or closures in an implementation. Map and function types have been incorporated into the leading programming languages, including Java, C++, Swift and Python.
The Object Constraint Language (OCL) specification notation lacks such types, and in this paper we make a proposal for a consistent extension of OCL with map and function types, and we identify modifications to OCL semantics to include these types. We also describe how map and function types are implemented using the Eclipse AgileUML toolset.

Keywords: Object Constraint Language (OCL) · Code generation · OCL semantics

1 Introduction

The Object Constraint Language (OCL) is the textual specification notation for class diagrams, metamodels and other UML models, including QVT transformations [19, 20]. The OCL originated in the work of Cook and others in the 1990's on semi-formal specification languages which could be used by general software practitioners [11]. Many of the concepts of OCL originated in the Z specification language [23], but with restrictions imposed in order to align the language more closely to computational systems. For example, only a finite powerset constructor $Set(X)$ is available in OCL, compared to the general powerset operator $\mathbb{P}(X)$ of Z. Z in turn is founded in Zermelo-Fraenkel set theory (ZFC) [4], which ensures the existence of the sets required by application of the type construction operators of Z.

Figure 1 shows the type system of the current OCL 2.4 standard. When OCL is used with a particular underlying model such as a class diagram or metamodel, the class types of the model are also available in OCL (as subtypes of *Class*), as are enumerated types (subtypes of *Enumeration*). This type system is based on the definitions of Annex A of [19], except that the Annex restricts *OclAny* to not be a supertype of collection or tuple types (Section A.2.7 of [19]); this restriction is removed in the main part of the OCL standard.

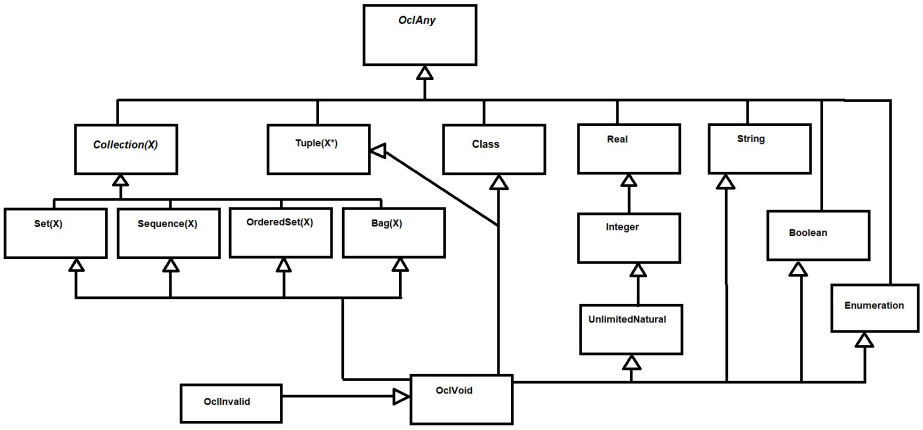

Fig. 1. Type system of OCL 2.4 standard [19]

Map and function types are now widely available in modern programming languages, and are part of the essential toolkit of software developers. They are also of high utility in software specifications, enabling concise and declarative specifications to be used for situations where associative data collections are needed, or functional genericity. Specification languages such as the EOL of Epsilon [9] and the expression languages of model transformation languages such as ATL [6] are related to OCL, and already include map types and operators. Map types were also present in Z and in the Z-based B language [13], and function types were available in Z.

Including map and function types in a specification language has advantages for code-generation from the language, enabling mechanisms such as operation caching and object indexing to be defined at a language-independent level, avoiding the need to define (in code-generators) language-specific schemes for these constructs in each target programming language. This potentially reduces the size and complexity of the code generators.

In the AgileUML toolset for UML [7], applications can be defined at a high level of abstraction using class diagrams, OCL and use cases. Consequently we divide code generation into two steps: (i) specification to design; (ii) design to code. The design is also expressed in UML, and is platform-independent, but explicit behaviour definitions are given for operations and use cases, using a pseudocode activity representation [16]. This common intermediate design stage helps to ensure semantic consistency of generated code in different target languages. Maps and function types are particularly useful at the design level, to define explicit semantics for specification-level concepts such as $\ll cached \gg$ operations and $\ll identity \gg$ (object identifier) attributes. They can be used to implement symbol tables for formally-specified software tools, and to store application preferences or other configuration property assignments. They can also be used to abstractly represent data structures such as CSV tables and JSON

values at the specification level. Object values can also be represented as maps from feature names to values.

Map types occur implicitly in UML as the type of qualified associations (Issue OCL25-202 [21]). Tuple types represent a restricted form of map type, and n-ary tuples for $n > 2$ could alternatively be expressed as n-element maps.

Since practitioners are now accustomed to using map types for many different purposes in software, it would seem reasonable to also provide this facility at the specification level. Otherwise, specifiers would be forced to use unintuitive constructions to achieve the effect that they want, and the gap between specification and implementation would be wider than necessary, complicating code generation and hindering code comprehension and verification. Thus there seems to be a strong case for adding map types and operators to OCL, and we present one approach in Section 2. The argument for function types is less clear, and the semantic complications are more severe than for maps, however we show in Section 3 how function types can also be consistently added to OCL, provided that the status of the 'universal type' *OclAny* is modified. In Section 4 we discuss related work, and in the Appendix we specify further map type operators, following the format of Chapter 11 of the OCL standard [19].

2 Map types

We propose to add a new type constructor $Map(,)$ of finite maps to OCL, such that $Map(K, T)$ denotes the type of finite maps from a domain K to a range T. K and T can be any OCL types from the type system of Figure 1, extended with *Map* types. The type $Map(K, T)$ could be regarded as a subtype of $Set(Tuple(key : K, value : T))$, however we will treat *Map* as an independent type constructor, as in EOL [9] and ATL [6].

A map is a (finite) aggregate data structure where the elements are indexed by key values. Elements may be members of the range of the map more than once, but key values must be unique. As with collections, the *invalid* value cannot be a member of a map, either as a key or range element. A Map m of type $Map(K, T)$ can be considered to have an underlying set $m \rightarrow asSet()$ of elements (pairs) of the type $Tuple(key : K, value : T)$. We use the simplified maplet notation $key \mapsto value$ for such pairs, which is the mathematical notation for map elements used in Z and B.

Following the same approach as adopted by the OCL standard for collections, we could write literal map values as $Map\{k1 \mapsto v1, ..., kn \mapsto vn\}$. An alternative notation[3] for this is $Map\{(k1, v1), ..., (kn, vn)\}$.

As discussed in the introduction, one useful application of map types is to define the semantics of caching of operation results. In particular, if a query operation

```
operation op(x : X) : Y
pre: P
```

[3] Used in the ATL extension of OCL.

```
post: Q
```

of class C has the stereotype $\ll cached \gg$, then in the specification-to-design step of code generation we can introduce a map-typed attribute

```
op_cache : Map(X,Y) := Map{}
```

of C, and define a platform-independent design of *op* as:

```
operation op(x : X) : Y
pre: P
activity:
  var result : Y ;
  if op_cache->includesKey(x)
  then
    return op_cache->at(x)
  else
    ( result := op_uncached(x) ;
      op_cache := op_cache->including(x,result) ;
      return result
    )
```

The operation *op_uncached* has the activity derived from Q, this is the version of the operation without caching. The above caching algorithm for *op* can then be translated into specific languages by design-to-code translators for the languages.

Similarly, if a class E has an $\ll identity \gg$ attribute $att : K$, then the design defines indexing maps

```
E_att_index : Map(K,E)
```

which enable efficient retrieval of E elements by key: $E\_att\_index \rightarrow at(k)$. This corresponds to the notation $E[k]$ in the specification. In implementations, map data structures can be used which have sub-linear time complexity for map application. This is therefore potentially more efficient than using a specification construct such as $E \rightarrow select(att = k) \rightarrow first()$ [3].

2.1 Map type semantics

The map type $Map(K, T)$ can be semantically represented by the mathematical type $K' \twoheadrightarrow T'$ of finite partial/total maps from the semantic representation K' of K to the representation T' of T [15, 23].

If T_1 is a subtype of T_2, then $Map(K, T_1)$ can be regarded as a subtype of $Map(K, T_2)$, because $T_1' \subseteq T_2'$, so that $m \in K' \twoheadrightarrow T_1'$ implies that $m \in K' \twoheadrightarrow T_2'$. Similarly, if K_1 is a subtype of K_2, then $Map(K_1, T)$ can be regarded as a subtype of $Map(K_2, T)$.

Annex A.2 of [19] can also be extended in a direct manner to include map types. For example, following the approach of the annex, the interpretation function for map types is:

$$I(Map(s,t)) = (I(s) \twoheadrightarrow I(t)) \cup \{\bot\}$$

where $\perp$ is the semantic interpretation of *invalid*. In the same manner, constructor and other operations for maps can be formally defined.

The role of *OclAny* needs to be examined with regard to the $Map(,)$ constructor. Because the constructor is finitary, the cardinality $card(Map(K, T))$ of a map type cannot be greater than that of K or T. The situation is therefore the same as with the existing collection types, eg., $card(Set(Real)) = card(Real)$. This holds because finite sets of real numbers can be themselves represented in a 1-1 manner as real numbers, and likewise for finite maps from *Real* to *Real*. Thus, in principle $Map(K, T)$ can be considered a subtype of *OclAny* for any K, T. However the 1-1 embeddings of $Map(K, T)$ or $Set(T)$ into *OclAny* are non-trivial (and not unique) when K or T themselves involve *OclAny*, so that subtyping in this sense cannot be simply regarded as subsetting.

There is inconsistency in [19] regarding the relation of *OclAny* to other OCL types. Sections 8.2 and 11 of [19] state that it is a supertype of all other OCL types, including collection and tuple types, as in Figure 1, but Annex A excludes collection/tuple types from inclusion in *OclAny* (Section A.2.6). In the absence of function types it is semantically consistent for *OclAny* to generalise collection, map and tuple types. If *OclAny* can itself be used as a type argument of any parameterised type, then $Map(OclAny, OclAny)$ is embeddable in *OclAny*, and the above issue of the choice of embedding arises. This extended type system for OCL is adopted by EOL and ATL. However, in the UML-RSDS notation of AgileUML there is no universal *OclAny* type [16]. Likewise, in Z and B there is no universal type [23]. If this approach was adopted the type system would appear as Figure 1 without the *OclAny* type.

The underlying mathematical theory of ZFC asserts that if sets a, b exist, then so does the Cartesian product $a \times b$ (axiom of pairing), powerset $\mathbb{P}(a)$, and distributed union $\bigcup(a)$. Moreover, $\mathbb{Z}$ exists (the axiom of infinity asserts the existence of $\mathbb{N}$), and sets can be formed using $\rightarrow select$ (by the axiom schema of separation), and $\rightarrow collect$ (by the axiom schema of replacement). The axioms also imply the existence of $\mathbb{R}$ (based on $\mathbb{P}(\mathbb{N})$). Countable sets isomorphic to $\mathbb{N}$ could be used as the semantic denotations of object types, with class extents (for $C.allInstances()$) being finite subsets of these. OCL maps from K to T can be based on finite sets of tuples in ZFC, ie., on elements of $\mathbb{F}(K \times T)$. We could also weaken ZFC to ZFC^F by using $\mathbb{F}(a)$ in the powerset axiom instead of $\mathbb{P}(a)$. In this case the existence of $\mathbb{R}$ would need to be additionally asserted.

2.2 Operations on maps

Some of the main operations on maps are given in the following and in the Appendix.

keys() : Set(K) The set of keys in the map, ie., its domain:

```
post:
  result = self->asSet()->collect( p | p.key )->asSet()
```

Notice that $\rightarrow collect$ returns a Bag in general, hence the second $\rightarrow asSet$.

The unique key constraint of maps is expressed by:

$$m \rightarrow asSet() \rightarrow size() = m \rightarrow keys() \rightarrow size()$$

values() : Bag(T) The bag of values in the map, ie., its range:

```
post:
  result = self->asSet()->collect( p | p.value )
```

=(c : Map(K,T)) : Boolean c and *self* are equal when both are maps of the same key and range types, and $c \rightarrow asSet() = self \rightarrow asSet()$.

<>(c : Map(K,T)) : Boolean The negation of =.

at(k : K) : T The value to which *self* maps k, *invalid* if k is not in $self \rightarrow keys()$:

```
post:
  (self->keys()->excludes(k) implies result = invalid) and
  (self->keys()->includes(k) implies
    result = self->restrict(Set{k})->values()->any())
```

2.3 Implementation

Implementations of the map type and map operators can be given in Java, Swift, C#, C++, Python and C. We have defined translations in the code generators for AgileUML [7], which can be accessed from [1] or viewed in the OCL libraries at http://www.nms.kcl.ac.uk/kevin.lano/libraries. Eg., in ocl.py for Python. Tables 1, 2 give some examples of the map type and operation mappings in different languages. `X` denotes the interpretation in each language of X in OCL. In C we define a custom data structure *ocltnode* based on binary search trees to represent maps. Other languages already have map types inbuilt or in standard libraries. The application of a map to a key is usually a constant-time or log-time operation in terms of the map size.

UML/OCL	Java 4/5/6	Swift 5	C#	C++
$Map(S, T)$	Map	Dictionary<S,T>	Dictionary	map<S,T> $*$
$m \rightarrow at(x)$	((T) m.get(x))	m[x]	m[x]	(*m)[x]
$m[x] = y$	m.put(x,y);	m[x] = y	m.Add(x,y);	(*m)[x] = y;

Table 1. Mappings from extended OCL to Java, Swift, C#, C++

In the directory *oclmapexamples.zip* on [1] we give examples of specifications using maps, together with the corresponding generated code in Java, C, Swift and Python.

UML/OCL	Java 7/8	Python	C
$Map(S, T)$	HashMap<S,T>	dict	struct ocltnode*
$m \rightarrow at(x)$	m.get(x)	m[x]	(T) lookupInMap(m, x)
$m[x] = y$	m.put(x,y);	m[x] = y	insertIntoMap(m,x,y);

Table 2. Mappings from extended OCL to Java 7+, C and Python

3 Function types

Being able to use functions as values was a key innovation of functional programming languages such as LISP and ML, and this facility was also provided in some procedural programming languages such as C, where a function name could be used as a pointer to the address of the function code in memory, and passed as an argument to other functions. Function types were definable in Z [23] and at the abstract specification level of B.

The two key operations on functions are *function abstraction* and *function application*. A mechanism to define function values by abstraction is now present in most modern programming languages, with different terminologies (lambda expressions in Python or Java, closures in Swift, delegates in C#, etc). These enable a function value to be defined by specifying the values of its applications, via expressions within different contexts. Both Z and B provide a function abstraction operator, written using the λ symbol.

Function types are used implicitly in the OCL 2.4 specification, for example in the definition of the *iterate* operator [24, 25]. We propose an extension of OCL with explicitly defined function types $S \rightarrow T$ (also denoted $Function(S, T)$) for types S and T, and function abstraction *lambda* $x : S$ *in* e. However, because this construction introduces the possibility of types of unbounded cardinality (Section 3.1), the *OclAny* type cannot be used in S, otherwise semantic contradictions appear [5]. Moreover, since equality of functions is generally undecidable, collections of functions will have undecidable membership relations.

An important use of function types is to simplify the code generation of expressions such as $E \rightarrow select(x \mid P)$, $E \rightarrow reject(x \mid P)$, $E \rightarrow collect(x \mid e)$, which depend upon other expressions P, e of arbitrary complexity. If function types are available, then the above constructs can be given uniform designs as higher-order functions independent of the specific P, e expressions. For example, *select* on E-sets could be defined by a single higher-order operation:

```
operation selectE(col : Set(E), f : E -> Boolean) : Set(E)
activity:
  var res: Set(E) := Set{} ;
  for x in col
  do
    if f(x)
    then res := res->including(x)
    else skip ;
  return res
```

An occurrence $col{\rightarrow}select(x \mid P)$ for set $col : Set(E)$ can then be interpreted as $selectE(col, lambda\ x : E\ in\ P)$. This avoids the need to define *select* operations for each specific P. In the same manner, the OCL $s{\rightarrow}sortedBy(e)$ operator can be expressed at the design stage in terms of a function parameter $lessThan : T \rightarrow (T \rightarrow Integer)$ on the type T of e, and a sorting algorithm such as merge sort.

Within specifications, function types are useful to provide genericity in cases where a uniform algorithm should be applied to a wide range of different functions. For example, an optimisation procedure such as bisection or secant can be applied to any total continuous function $f : Real \rightarrow Real$:

```
operation secant(rn : Real, rprev : Real, fprev : Real, tol : Real,
    f : Real -> Real) : Real
pre: tol > 0
post:
  let fn : Real = f(rn) in
    if abs(fn) < tol
    then result = rn
    else
      result = secant(rn - fn*((rn-rprev)/(fn-fprev)),rn,fn,tol,f)
    endif
```

A similar situation arises with genetic and other evolutionary algorithms which can be parameterised by different fitness functions.

The introduction of function types would also enable a functional programming language style of specification in OCL, in terms of functions and higher-order functions. This specification style is particularly useful for application areas such as finance and machine learning.

3.1 Function type semantics

The OCL function type $S \rightarrow T$ can be given a semantics as the set of partial functions $S' \nrightarrow T'$ in [15, 14]. In the case that f of type $S \rightarrow T$ in OCL maps $s : S$ to *invalid*, s is considered to be outside the domain of the semantic representation f' of f: $s' \notin \mathrm{dom}(f')$. If S_1 conforms to S_2 and T_1 conforms to T_2, then $S_1 \rightarrow T_1$ conforms to $S_2 \rightarrow T_2$: every function from S_1 to T_1 can also be treated as a (partial) function from S_2 to T_2, with *invalid* value on elements of S_2 not in S_1.

The semantics of function types can also be expressed within the framework of Annex A of [19]:

$$I(s \rightarrow t) = (I(s) \nrightarrow I(t)) \cup \{\bot\}$$

In contrast to maps, functions are immutable. They can also be non-finite and have non-finite domains and ranges. Thus the domains and ranges cannot be expressed as OCL collections in general, nor can $\rightarrow asSet()$ be computed for functions. The $S \rightarrow T$ type constructor can produce types with non-countable

sets of elements, for example $Integer \rightarrow Integer$. Moreover, $S \rightarrow Boolean$ always has higher cardinality than S:

$$card(S \rightarrow Boolean) > card(S)$$

This has the implication that if *OclAny* is regarded as a universal type extending all OCL types, then it cannot occur as the first argument of a function type, otherwise the contradiction

$$card(OclAny) \geq card(OclAny \rightarrow Boolean) > card(OclAny)$$

would arise.

Even with this restriction, the construction of a universal type *OclAny* including all other forms of function type would require the application of powerful transfinite axioms of ZFC, ie., to construct a set containing all elements of $\mathbb{N}$, $\mathbb{P}(\mathbb{N})$, $\mathbb{P}(\mathbb{P}(\mathbb{N}))$, ... For this reason we consider that it is preferable to either remove the concept of a universal type from the OCL type system, as in [16, 15, 14], and languages such as Z and B, or to exclude function and other parameterised types from being subtypes of the universal type.

Adopting the 'no *OclAny*' approach, the revised OCL type system incorporating map and function types would be as shown in Figure 2.

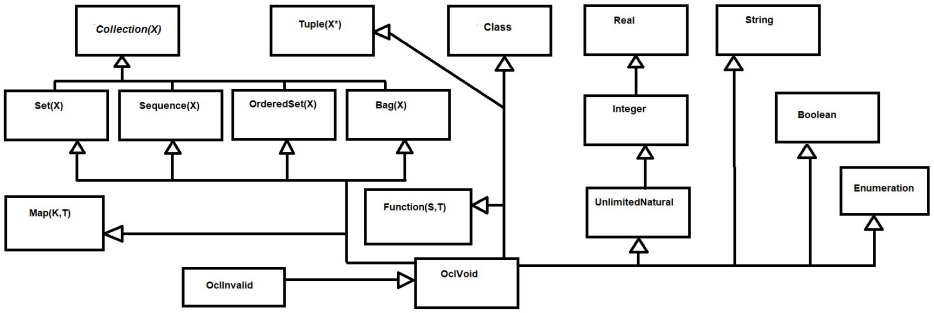

Fig. 2. Revised OCL type system

The semantics of function application is described in Section A.3.1.2 of [19].

3.2 Operations on functions

Some of the key operations on functions are given in the following. While theoretical definitions of equality, inequality and definedness of functions can be given, these cannot be effectively computed in all cases, and hence should not be used in specifications. Only abstraction and application are valid operators for all functions.

Function application Denoted $f(x)$ for $f : S \to T$ and $x : S$, this is an element of T, or is *invalid*, if x is not in the domain of f.

Function abstraction *lambda* $x : S$ *in* E The function of type $S \to T$ defined by evaluating $E : T$ on elements of S.

S ::
$$(lambda\ x : S\ in\ E)(self)\ =\ E[self/x]$$

where $P[e/v]$ denotes the substitution of e for v in P. E should be side-effect free, and x should not occur bound in E.

An alternative notation for abstraction could be *let* $x : S$ *in* E, however this may introduce confusion with the existing OCL *let* operator. Indeed, *let* and *lambda* are related by $(let\ x : S = v\ in\ E) = (lambda\ x : S\ in\ E)(v)$ for side-effect free E. The notation could be used with multiple arguments: *lambda* $x1 : S1, ..., xn : Sn$ *in* E to form anonymous functions with multiple parameters. The Currying transformation of this to *lambda* $x1 : S1$ *in* $(lambda\ x2 : S2, ..., xn : Sn\ in\ E)$ is also naturally expressed in this notation.

By definition the property of β-reduction holds for the *lambda* operator; α-conversion also holds in the sense that functions *lambda* $x : S$ *in* E and *lambda* $y : S$ *in* $E[y/x]$ are equivalent if neither y or x occur as bound variables in E. The expression E may involve data identifiers other than x, and the formed lambda function is then implicitly a function of these additional variables, and can only be evaluated in contexts where they are defined. This property is termed *scope capture*.

3.3 Implementation

Table 3 gives some examples of the implementation of function types and function application and abstraction in different programming languages. Function abstraction is not supported in C or in earlier versions of Java. Instead, in C, we introduce new global operations `T op_i(S x) { return e; }` for each occurrence *lambda* $x : S$ *in* e of a lambda expression, and translate the expression as `op_i`. This approach does not support scope capture: e must depend only on x.

UML/OCL	C++	Swift 5	Java 8	Python
$S \to T$	T (*)(S)	(S) -> T	Evaluation<S,T>	–
$f(x)$	(*f)(x)	f(x)	f.evaluate(x) f(x)	f(x)
lambda $x : S$ *in* E	[](S x) -> T { return E; }	{ (x : S) -> T in E }	(x) -> { return E; }	lambda x : E

Table 3. Mappings from extended OCL to Java, Swift, Python, C++

In the directory *oclfunctionexamples.zip* on [1] we give an example specification using function types, together with the corresponding generated code in Java, Swift, C and Python. Currently in AgileUML version 2.0, lambda expressions may only occur in the arguments of operation calls.

4 Related work

There are two main semantic approaches for OCL: (i) via translation to a classical or extended logic and set theory based on ZFC (eg., [14, 15] or Annex A of [19]), or (ii) metamodelling semantics [12], whereby OCL syntax and value domains are defined via metamodels, and expression evaluation defined in terms of instances of these models. Either approach could be used to provide a semantics for the extensions which we propose, and we have indicated above the necessary modifications for the translational semantic approach. While the semantic revisions required for function types are more severe than for map types, these can also be consistently defined, provided that the concept of a universal type (*OclAny*) is modified.

In summary, map types can be added to OCL, retaining the existing type hierarchy (Figure 1), and with $Map(K, T)$ a subtype of *OclAny* for any K, T, including parameter types that involve *OclAny*. However, as with the subtyping of $Set(OclAny)$ with respect to *OclAny*, such subtype relations cannot be expressed as subset relationships. It would also be possible to have a type system without *OclAny* (Figure 2).

In contrast, for function types there are three main choices:

1. Retain *OclAny* as the supertype of all types, and forbid the use of *OclAny* in the domain argument of $Function(D, R)$.
   This suffers from the flaw that subtyping of collection, tuple map and function types wrt *OclAny* will not correspond to subsetting, as above. In addition *OclAny* would need to have a high transfinite cardinality.
2. Retain *OclAny* only as a supertype of non-parameterised types (*Class*, *Real*, *String*, *Enumeration*, *Boolean*). In this case, subtyping wrt to *OclAny* could be represented as subsetting. The cardinality of *OclAny* would be $card(Real)$.
3. Remove *OclAny* from the type system, as in Figure 2. This prevents the formation of heterogeneous collections such as sets including a mix of strings, numbers, objects, maps, etc.

In AgileUML we adopt the third option.

OCL was the subject of intensive research in the initial years of the UML standard and MDE [5, 10, 11, 22], however subsequently there was slow progress on the development of the language, due mainly to language engineering factors and interdependence between the standard and other OMG standards [2, 24, 25]. The projected revision to OCL 2.5 was abandoned, however a long-term revision and rationalisation of the language to Version 3.0 is proposed in [25]. Function types are one possible extension suggested by [25], together with OCL libraries.

A map type and operators were added to ATL from Version 0.7 onwards [6], and these are defined in a style consistent with OCL collection operators. The Map type and operators in EOL [9] are however closely aligned with the implementation of maps in Java. We initially raised the suggestion of a map type and operators for OCL in [17]. The present paper develops this proposal based on feedback, and on experience of implementing the map type and operators in a range of programming languages. A related proposal is in [26].

The Eclipse OCL version [8] provides a map type with a similar set of operators to those proposed here. Table 4 compares the AgileUML and Eclipse OCL Map operators.

Eclipse OCL	*AgileUML OCL*
$=$, $<>$, $\rightarrow at$	$=$, $<>$, $\rightarrow at$
$m \rightarrow excludes(k)$	$m \rightarrow excludesKey(k)$
$m \rightarrow excludes(k, v)$	$m \rightarrow excludesAll(Map\{k \mapsto v\})$
$m \rightarrow excludesAll(ks)$	$m \rightarrow keys() \rightarrow intersection(ks) \rightarrow isEmpty()$
$m \rightarrow excludesMap(m1)$	$m \rightarrow excludesAll(m1)$
$m \rightarrow excludesValue(v)$	$m \rightarrow excludesValue(v)$
$m \rightarrow excluding(k)$	$m \rightarrow restrict(m \rightarrow keys() \rightarrow excluding(k))$
$m \rightarrow excluding(k, v)$	$m - Map\{k \mapsto v\}$
$m \rightarrow excludingAll(ks)$	$m \rightarrow restrict(m \rightarrow keys() - ks)$
$m \rightarrow excludingMap(m1)$	$m - m1$
$m \rightarrow includes(k)$	$m \rightarrow includesKey(k)$
$m \rightarrow includes(k, v)$	$m \rightarrow includesAll(Map\{k \mapsto v\})$
$m \rightarrow includesAll(ks)$	$m \rightarrow keys() \rightarrow includesAll(ks)$
$m \rightarrow includesMap(m1)$	$m \rightarrow includesAll(m1)$
$m \rightarrow includesValue(v)$	$m \rightarrow includesValue(v)$
$m \rightarrow including(k, v)$	$m \rightarrow including(k, v)$
$m \rightarrow includingMap(m1)$	$m \rightarrow union(m1)$
$\rightarrow isEmpty$, $\rightarrow notEmpty$, $\rightarrow size$	$\rightarrow isEmpty$, $\rightarrow notEmpty$, $\rightarrow size$
$\rightarrow keys$, $\rightarrow values$	$\rightarrow keys$, $\rightarrow values$

Table 4. Comparison of Eclipse OCL and AgileUML map operators

The main difference is that [8] uses the terminology *excludes*/*includes* for operators relating a map and key, whilst we use the more explicit *excludesKey*/ *includesKey*. We use the collection/collection operator names *includesAll*, *excludesAll* for the analogous map/map operators, instead of for map/key set operators. Additionally, we provide a literal map notation, and further map operators such as *count*, *select*, etc by analogy with corresponding collection operators. Our objective in this respect is to align the map operators closely with collection operators, in order to enhance the usability of the extended OCL notation.

Conclusions

We have described possible extensions of OCL with map and function types, and associated operators. A semantics has been given for these extensions, and a rationale provided, in terms of benefits for specification and code generation.

References

1. AgileUML repository, https://github.com/eclipse/agileuml/, 2020.
2. M. Belaunde, *Evolution of the OCL OMG specification*, OCL 2010.
3. J. Cuadrado et al., *Optimisation patterns for OCL-based model transformations*, MODELS 2009, LNCS Vol. 5421, 2009, pp. 273–284.
4. K. Ciesielski, *Set theory for the working mathematician*, Cambridge University Press, 1997.
5. S. Cook, A. Kleppe, R. Michell, B. Rumpe, J. Warmer, A. Wills, *The Amsterdam manifesto on OCL*, in Object Modeling with the OCL, LNCS vol. 2263, 2002, pp. 115–149.
6. Eclipse, *ATL user guide*, eclipse.org, 2019.
7. Eclipse AgileUML project, https://projects.eclipse.org/projects/modeling.agileuml, 2020.
8. Eclipse OCL Version 6.4.0, https://projects.eclipse.org/projects/modeling.mdt.ocl, 2021.
9. The Epsilon Object Language, https://www.eclipse.org/epsilon/doc/eol, 2020.
10. R. Hennicker, H. Hussmann, M. Bidoit, *On the Precise Meaning of OCL Constraints*, Object Modeling with the OCL 2002: pp. 69–84.
11. A. Kleppe, J. Warmer, S. Cook, *Informal formality? The OCL and its application in the UML metamodel*, UML '98 conference, Springer-Verlag, 1999, pp. 148–161.
12. A. Kleppe, *Object Constraint Language: Metamodelling semantics*, Chapter 7 of *UML 2 Semantics and Applications*, Wiley, 2009.
13. K. Lano, *The B Language and Method*, Springer-Verlag, 1996.
14. K. Lano, *A Compositional Semantics of UML-RSDS*, Software and Systems Modelling, Vol. 8, No. 1, February, 2009, pp. 85–116.
15. K. Lano, T. Clark, S. Kolahdouz-Rahimi, *A Framework for Model Transformation Verification*, FACS, vol. 27, 2015, pp. 193–235.
16. K. Lano, *Agile model-based development using UML-RSDS*, CRC Press, 2016.
17. K. Lano, *Map type support in OCL?*, https://www.eclipse.org/forums/index.php/t/1096077/, November 2018.
18. J. Monk, *Mathematical Logic*, Springer-Verlag, 1976.
19. OMG, *Object Constraint Language 2.4 Specification*, 2014.
20. OMG, *MOF2 Query/View/Transformation v1.3*, 2016.
21. OMG, https://issues.omg.org/issues/spec/OCL/2.4, 2020.
22. M. Richters, M. Gogolla, *On Formalizing the UML Object Constraint Language OCL*, Proc. 17th Int. Conf. Conceptual Modeling (ER 98), Springer LNCS 1507, pp. 449–464, 1998.
23. J. Spivey, *The Z Notation*, Prentice Hall, 1989.
24. E. Willink, *OCL omissions and contradictions*, OMG ADTF, 2012.
25. E. Willink, *Reflections on OCL 2*, Journal of Object Technology, Vol. 19, No. 3, 2020.
26. E. Willink *An OCL Map Type*, OCL '19, 2019.

A Additional map type operators

The following map operators can also be formalised in our proposed extension of OCL.

size() : Integer

```
post: result = self->asSet()->size()
```

This is equal to $self \rightarrow keys() \rightarrow size()$.

includesValue(object : T) : Boolean True if the *object* is an element of the map range, false otherwise:

```
post:
  result = self->values()->includes(object)
```

Similarly for $\rightarrow includesKey()$, $\rightarrow excludesValue()$, $\rightarrow excludesKey()$.

count(object : T) : Integer The number of times the *object* occurs as an element of the map range (a bag):

```
post:
  result = self->values()->count(object)
```

includesAll(c2 : Map(K,T)) : Boolean True if $c2$ is a map, and the set of pairs of *self* contains all those of $c2$, false otherwise:

```
post:
  result = self->asSet()->includesAll(c2->asSet())
```

Similarly for $\rightarrow excludesAll()$.

isEmpty() : Boolean, notEmpty() : Boolean Defined based on $self \rightarrow asSet()$.

max() : T, min() : T, sum() : T Defined as the corresponding operations on $self \rightarrow values()$.

asSet() : Set(Tuple(key : K, value : T)) The set of pairs of elements in the map. Since duplicate keys are not permitted, this has the same size as $self \rightarrow keys()$.

restrict(ks : Set(K)) : Map(K,T) Domain restriction $ks \lhd self$. The map restricted to the keys in ks. Its elements are the pairs of *self* whose key is in ks:

```
post:
  result->asSet() =
    self->asSet()->select(ks->includes(key))
```

Range restriction is provided via the $\rightarrow select$ operator.

-(m: Map(K,T)) : Map(K,T) Map subtraction: the elements of *self* that are not in m.

```
post:
  result->asSet() =
      self->asSet() - m->asSet()
```

union(m : Map(K,T)) : Map(K,T) Map override, the operation $self \oplus m$ or $self <+ m$ in mathematical notation. This consists of the pairs of *self* which do not conflict with pairs of m, together with all pairs of m:

```
post:
  result->asSet() =
    m->asSet()->union(
      self->asSet()->select(p | m->keys()->excludes(p.key)))
```

This is the same as *putAll* in EOL [9]. Unlike set union, it is not commutative.

intersection(m : Map(K,T)) : Map(K,T) The pairs of *self* which are also in m:

```
post:
  result->asSet() =
    m->asSet()->intersection(self->asSet())
```

This is associative and commutative.

including(k : K, v : T) : Map(K,T) The pairs of *self*, with the additional/overriding mapping of k to v:

$$self{\rightarrow}including(k, v) \;\widehat{=}\; self{\rightarrow}union(Map\{k \mapsto v\})$$

We also use the abbreviated notation $m[k] = v$ for $m = m@pre{\rightarrow}including(k, v)$.

excluding(k : K, v : T) : Map(K,T) The pairs of *self*, with any mapping of k to v removed:

$$self{\rightarrow}excluding(k, v) \;\widehat{=}\; self - Map\{k \mapsto v\}$$

any Defined as

$$m{\rightarrow}any(x \mid P) \;\widehat{=}\; m{\rightarrow}values(){\rightarrow}any(x \mid P)$$

Likewise for *forAll*, *exists*, *one*.

select The map formed by restricting to range elements which satisfy the *select* condition:

$$m{\rightarrow}select(x \mid P(x)) \;\widehat{=}\; m{\rightarrow}restrict(m{\rightarrow}keys(){\rightarrow}select(k \mid P(m{\rightarrow}at(k))))$$

Similarly for *reject*.