An Extended Type System for OCL Supporting Templates and Transformations[*]

Marcel Kyas

Christian-Albrechts-Universität zu Kiel, Germany
mky@informatik.uni-kiel.de

Abstract. Based on our experience in implementing a type-checker for the Object Constraint Language (OCL), we observed that OCL is not suitable for constraining a system under development, because changes in the underlying class diagram unnecessarily invalidate the type correctness of constraints, while their semantic value does not change. Furthermore, the type system of OCL does not support templates.
To alleviate these problems, we extended the type system of OCL with intersection and union types and bounded operator abstraction. The main advantage of our type system is that it allows more changes in the contextual class diagrams without adapting the OCL constraints.

1 Introduction

The Object Constraint Language (OCL) is a formal specification language that enables a developer to specify class invariants and pre- and postconditions for operations in UML models. It is designed to be a query language, like SQL, and specification language, like Z. Its latest version, OCL 2.0, is described in [1], to which we refer as *OCL 2.0 proposal*. It aims at a tight integration with the diagrammatic notations of UML 2.0, which are documented in [2] and [3].

In order to be used widely, OCL has to support the following:

1. A precise syntax which allows writing specifications in a concise and readable way, but which is also machine readable, and therefore also machine checkable.
2. A precise semantics which allows evaluation or verification of the model.
3. A type system which is compatible with the well-formedness constraints of UML 2.0 class diagrams.
4. A type system which is robust with respect to model transformations like refactoring or other changes in class diagrams.

Influenced by our experience in implementing a standard-conforming type-checker for OCL, we have come to the conclusion that OCL does not adequately implement these requirements so far:

[*] Part of this work has been financially supported by IST project Omega (IST-2001-33522) and NWO/DFG project Mobi-J (RO 1122/9-1, RO 1122/9-2)

The first item is not satisfied, because in the UML 2.0 and OCL 2.0 standards OCL constraints in different syntactical styles are used (compare the constraints in [2] to the ones in [1]).

The second item is not satisfied, because even though the semantics of OCL is precise enough for evaluating constraints [4, 5], it is not convenient for verification purposes, because the semantics of OCL is operational, and not declarative, as argued in [6], which leads, a.o., to the three-valuedness of the logic.

Items three and four represent the main problem of using OCL for writing constraints on models during different stages of design: the type system of OCL appears to be designed for languages which only use single inheritance and no templates (parameterized classes).[1] UML 2.0 introduces a new model for templates, which allows classifiers to be parameterized with classifier specifications, value specifications, and operation specifications. The OCL 2.0 proposal does not specify how those parameters can be used in constraints, how they can be constrained, or how these parameters are to be used in constraints and what their meaning is supposed to be. Furthermore, OCL constraints are fragile under the operations of refactoring of class diagrams, package inclusion and package merging. These operations, which often do not affect the semantic value of a constraint, can render constraints ill-typed. This essentially limits the use of OCL to a-posteriori specification of class diagrams.

To solve these problems, we have implemented a more expressive type system based on intersection types, union types, and bounded operator abstraction. Such type systems are already well-understood [7] and solve the problems we encountered elegantly. Our type system supports templates and is more robust under refactoring and package merging than the current type system.

The adaption of the type system to OCL was straight forward. The specification of the OCL standard library had to be changed to make use of the new type system. We have implemented this type system in a prototype tool and all constraints of the OCL 2.0 standard library have been shown to be well-typed with respect to our type system.

This paper is organized as follows: In Sect. 2, we survey the current type system for OCL. In Sect. 3, we describe our different extensions to the type system. In Sect. 4, we summarize the most important results. In Sect. 5 we compare our results with other results and draw some conclusions.

2 State of the Art

In this section, we recall the current type system used for OCL, which has been derived from the OCL 2.0 standard. It is similar to the one presented in [8].

[1] For example Java before version 5.0.

We start with a description of abstract OCL, a simple core language, into which almost all OCL expression can be translated. The grammar is defined by

$$\begin{aligned} t ::= {} & \mathit{true} \mid \mathit{false} \mid \cdots \mid -1 \mid 0 \mid 1 \mid \cdots \mid \mathit{self} \mid v \mid t.a \mid t.m(t_1, \ldots, t_n) \\ \mid {} & t \to m(t_1, \ldots, t_n) \mid t \to \mathit{iterate}(v_0 : T_0, \ldots, v_n : T_n; a = t \mid t_0) \\ \mid {} & t \to \mathit{flatten}(t) \mid \textbf{if } t \textbf{ then } t' \textbf{ else } t'' \textbf{ endif} \\ \mid {} & \mathit{let}\ v_0(v_{0,0}, \ldots, v_{0,m_0}) : T = t_0, \ldots, v_n(v_{n,0}, \ldots, v_{n,m_n}) : T = t_n\ \mathit{in}\ t \end{aligned}$$

We do not use @*pre* and qualifiers in our language, because these constructs do not add anything to the type system. We define the operation *flatten*, which flattens collections, as a primitive to OCL, like *iterate*. The reasons for this are discussed in Sect. 3.5.

We now define the abstract syntax of OCL types. We have essentially two kinds of types: elementary types and collection types. The elementary types are classifiers from the model and the elementary data types like *Boolean, Integer,* and so on. The collection types are types which are generic, i.e., they construct a type by applying the collection type to any other type.[2] This distinction is formalized with a kinding system (a type system for types). Kinds are defined by the language $K ::= \star \mid K \to K'$. The kind $\star$ denotes any type which does not take an argument. Type constructors have a kind $K \to K'$, which means that such a constructor maps each type of kind K to a type of kind K'. For example, the elementary data type *Integer* is of the kind $\star$. The collection type *Set* is of the kind $\star \to \star$ and the type *Set(Integer)* is of the kind $\star$. The language of types is defined as follows:

$$T ::= \mathit{type} \mid T(T_1) \mid T_0 \times \cdots \times T_n \to T$$

Here, a *type* is any classifier or template appearing in the contextual class diagram or the OCL standard library. The expression $T(T_1)$ expresses the type which results from instantiating a template parameter of the type T with T_1. The type $T_0 \times \cdots \times T_n \to T$ is used to express the type of properties. The type T_0 is the type of the classifier which defines the property, the types $T_1, \ldots, T_n$ are the types of the parameters of the property. We identify attributes with operations that do not define arguments.

Observe that our language of types does not contain constructs for operator abstraction or universal types. The reason for this is, that you cannot define *new* types in OCL. Instead all types are defined in class diagrams and are used like constants in the type system.

The kinding of a type states whether a type is an elementary type or a template and is formally defined by the system shown in Fig. 1. We write the rules in the usual style: a rule consists of an antecedent and a consequence, which are separated by a line. The antecedent contains the properties that need to be proved in order to apply the rule and conclude its consequence. Each rule has a name, which is stated right of the line in small capitals.

[2] In Sect. 4 we discuss the presence of *dependent types* in class diagrams.

$$T : \star \text{ For any type or property type } T \quad \text{K-ELEM}$$

$$T : \star \to \star \text{ For any parameterized class } T \quad \text{K-CONS} \qquad \frac{T : K \qquad S : K \to K'}{S(T) : K'} \text{ K-INST}$$

Fig. 1. Kinding System

It is an important property of the type system for OCL that new types cannot be defined through OCL expressions (except for tuples, which are out of the scope of this paper). This simplifies the type checking rules a lot. OCL expressions are checked in a context, which contains the information on variable bindings, operation declarations, and the subtype relation encoded in a class diagram. A context Γ maps variable names v to their type, or to undefined if that variable is not declared in this context. We write $\Gamma, v : T$ to denote the context extended by binding the variable v to T, provided that v does not occur in Γ. We write $\Gamma(v) = T$ to state that v has type T in context Γ. The context also contains the information on *type conformance*, i.e. clauses of the form $T \leq S$ derived from the generalization hierarchy, where T and S are types and $\leq$ denotes that T is a subtype of S. We write $\Gamma, T \leq S$ to extend a context with a statement that T is a subtype of S. Any context contains $T \leq T$ for every type T occurring in the model, since the conformance relation is reflexive. If the context Γ contains the declaration $T \leq S$ we denote this with $\Gamma \vdash T \leq S$. We write $\Gamma \vdash t : T$ to denote that t is a term of type T in the context Γ. Contexts which only differ in a different order of their declarations are considered equal. If the context is clear, we omit it in the example derivations. Finally, we assume that the context contains all declarations mandated by the OCL standard library. The subtype relation is transitive and function application is covariant in its arguments and contravariant in its result type. These rules are shown in Fig. 2. In rule S-COLL the notation $\Gamma \vdash C \leq \mathit{Collection}$ means that C ranges over every type which is a subtype of *Collection*. OCL defines *Bag*, *Set*, and *Sequence* as subtypes of *Collection*. Note that *Collection* is a parameterized type, but the rule S-COLL-2 is not sound for classes which define operations which change the contents of the collection [9]. In OCL operations defined on collections do not alter the content, but we cannot assume this in general, therefore we defined these particular assumptions. The typing rules for terms are presented in Fig. 3, except for the typing rule for *flatten*. The type of flatten is actually a dependent type, because it depends on the type of its argument. We present the rule in Sect. 3.5.

Rules T-TRUE, T-FALSE, and T-LIT assign to each literal their type. Especially, T-LIT is an axiom scheme assigning, e.g. the literal 1 the type *Integer* and the literal 1.5 the type *Real*. Rule T-COLL defines the type of a collection literal. The type of a collection is determined by the declared name C and the common supertype of all its members. Rule T-CALL states that if the arguments match the types of a method or a function, then the expression is well-typed and the

$$\frac{\mathit{\Gamma} \vdash C \leq \mathit{Collection}}{\mathit{\Gamma} \vdash C(T) \leq \mathit{Collection}(T)}\ \textsc{S-Coll} \qquad \frac{\mathit{\Gamma} \vdash C \leq \mathit{Collection} \quad \mathit{\Gamma} \vdash T \leq T'}{\mathit{\Gamma} \vdash C(T) \leq C(T')}\ \textsc{S-Coll-2}$$

$$\frac{\mathit{\Gamma} \vdash e : S \quad \mathit{\Gamma} \vdash S \leq T}{\mathit{\Gamma} \vdash e : T}\ \textsc{S-Sub} \qquad \frac{\mathit{\Gamma} \vdash S \leq T \quad \mathit{\Gamma} \vdash T \leq U}{\mathit{\Gamma} \vdash S \leq U}\ \textsc{S-Trans}$$

$$\frac{\mathit{\Gamma} \vdash T_0 \leq S_0, \mathit{\Gamma} \vdash S_1 \leq T_1, \cdots, \mathit{\Gamma} \vdash S_n \leq T_n \quad \mathit{\Gamma} \vdash T \leq S}{\mathit{\Gamma} \vdash T_0 \times T_1 \times \cdots \times T_n \to T \leq S_0 \times S_1 \times \cdots \times S_n \to S}\ \textsc{S-Arrow}$$

Fig. 2. Definition of Type Conformance

$$\mathit{\Gamma} \vdash \mathit{true} : \mathit{Boolean} \quad \textsc{T-True} \qquad \mathit{\Gamma} \vdash \mathit{false} : \mathit{Boolean} \quad \textsc{T-False}$$

$$\mathit{\Gamma} \vdash l : T_l \quad \textsc{T-Lit} \qquad \frac{\mathit{\Gamma}(v) = T}{\mathit{\Gamma} \vdash v : T}\ \textsc{T-Var}$$

$$\frac{\mathit{\Gamma} \vdash e_1 : T \quad \cdots \quad \mathit{\Gamma} \vdash e_n : T}{C\{e_1, \ldots, e_n\} : C(T)} \quad \text{if } C \in \{\mathit{Bag}, \mathit{Set}, \mathit{OrderedSet}, \mathit{Sequence}\}\ \textsc{T-Coll}$$

$$\frac{\begin{array}{c}\mathit{\Gamma} \vdash t_0 : S \quad \mathit{\Gamma} \vdash t_1 : S_1, \cdots, \mathit{\Gamma} \vdash t_n : S_n \\ \mathit{\Gamma} \vdash n : S \times S_1 \times \cdots \times S_n \to T \quad \mathit{\Gamma} \vdash S \not\leq \mathit{Collection}\end{array}}{\mathit{\Gamma} \vdash t_0.n(t_1, \ldots, t_n) : T}\ \textsc{T-Call}$$

$$\frac{\begin{array}{c}\mathit{\Gamma} \vdash t_0 : C(S) \quad \mathit{\Gamma} \vdash t_1 : S_1, \cdots, \mathit{\Gamma} \vdash t_n : S_n \\ \mathit{\Gamma} \vdash n : C(S) \times S_1 \times \cdots \times S_n \to T \quad \mathit{\Gamma} \vdash C \leq \mathit{Collection}\end{array}}{\mathit{\Gamma} \vdash t_0 \to n(t_1, \ldots, t_n) : T}\ \textsc{T-CCall}$$

$$\frac{\mathit{\Gamma} \vdash e_0 : \mathit{Boolean} \quad \mathit{\Gamma} \vdash e_1 : T \quad \mathit{\Gamma} \vdash e_2 : T}{\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{endif} : T}\ \textsc{T-Cond}$$

$$\frac{\mathit{\Gamma} \vdash t_0 : C(T) \quad \mathit{\Gamma} \vdash t : S \quad \mathit{\Gamma}, v : T, a : S \vdash e : S \quad \mathit{\Gamma} \vdash C \leq \mathit{Collection}}{\mathit{\Gamma} \vdash t_0 \to \mathit{iterate}(v; a = t \mid e) : S}\ \textsc{T-Iterate}$$

$$\frac{\mathit{\Gamma} \vdash t_0 : T_0 \qquad \mathit{\Gamma}, v_0 : T_0 \vdash t : T}{\mathit{\Gamma} \vdash \mathit{let}\ v_0 : T_0 = t_0\ \mathit{in}\ t\ \ : T}\ \textsc{T-Let}$$

$$\frac{\begin{array}{c}\mathit{\Gamma}' = \mathit{\Gamma}, v_0 : T_{0,1} \times \cdots \times T_{0,m_0} \to T_1, \ldots, v_n : T_{n,1} \times \cdots \times T_{n,m_n} \to T_n \\ \mathit{\Gamma}', v_{0,0} : T_{0,0}, \ldots, v_{0,m_0} : T_{0,m_0} \vdash t_0 : T_0 \\ \vdots \\ \mathit{\Gamma}', v_{n,0} : T_{n,0}, \ldots, v_{n,m_n} : T_{n,m_n} \vdash t_n : T_n \\ \mathit{\Gamma}' \vdash t : T\end{array}}{\mathit{\Gamma} \vdash \mathit{let}\ v_0(v_{0,0}, \ldots, v_{0,m_0}) : T_0 = t_0, \ldots, v_n(v_{n,0}, \ldots, v_{n,m_n}) : T = t_n\ \mathit{in}\ t\ \ : T}\ \textsc{T-Let}'$$

Fig. 3. Typing rules for OCL

result has the declared type. The antecedent $\Gamma \vdash C \nleq \mathit{Collection}$ denotes that in context Γ type C is not a subtype of *Collection*. A similar rule for collection calls is given by T-CCall. Recall that the antecedent $\Gamma \vdash C \leq \mathit{Collection}$ states that the type of t_0 has to be a collection type. Rule T-Cond defines the typing of a condition. If the condition e_0 has the type *Boolean* and the argument expressions e_1 and e_2 have a common supertype T, then the conditional expression has that type T. Rule T-Iterate gives the typing rule for an iterate expression. First, the expression we are iterating over has to be a collection. Then the accumulator has to be initialized with an expression of the same type. Finally, the expression we are iterating over has to be an expression of the accumulator variables type in the context which is extended by the iterator variable and the accumulator variable. Rule T-Let defines the rule for a let expression of the OCL 2.0 standard. Rule T-Let' allows a let-expression where the user can define functions and use mutual recursion. There we add all variables declared by the let expression to context Γ in order to obtain context Γ'. Each expression defined has to be well typed in the context extended by the formal parameters of the definition. Finally, the expression in which we use the definitions has to be well-typed in the context Γ'.

This type system is a faithful representation of the type system given in [1], but we have omitted the typing rules for the boolean connectives, as they are given by Cengarle and Knapp in [5], because each expression using a boolean connective can be rewritten to an operation call expression, e.g., $a\ and\ b$ is equivalent to $a.and(b)$. We do not have a rule for the undefined value, as presented in [5], because the OCL 2.0 proposal does not define a literal for undefined [1, pp. 48–50].[3]

Within the UML 2.0 standard [3] and the OCL 2.0 standard [1] methods are redefined covariantly. We assume that some kind of multi-method semantics for calls of these methods is intended. These redefinitions are not explicitly treated in the OCL 2.0 type system, they can, however, be treated as overloading a method, and hence, be modeled with union-types in our system (see Sec. 3.2), as suggested in, e.g., [7, p. 340].

Also note that our type system makes use of the largest common supertype only implicitly, whereas it is explicitly used in other papers. It is hidden in the type conformance rules of Fig. 2. An example of where we use the largest common supertype can be found in Sect. 3.1. The rules presented here are not designed for a type-checking algorithms, but for deriving well-typedness. Therefore, the type system presented here lacks the unique typing property, but it is adequate with respect to the operational semantics defined in [1] and decidable.

Proposition 1. *The type system is* adequate, *i.e.for any OCL expression e if $e : T$ can be derived in the type system, then e is evaluated to a result of a type conforming to T.*

[3] Note that *OclUndefined* is the semantic value of any undefined expression and the "only instance of OclVoid", and there not part of the concrete syntax [1, p. 133]. Calling the property *oclIsUndefined()*, defined for any object, is preferred, because any other property call results in OclUndefined.

The type system is decidable*, i.e. there exists an algorithm which either derives a type T for any OCL expression e or reports that no type can be derived for e.*

We use the definitions of this section for the discussion of its limitations in the following sections.

3 Extensions

In this section, we propose various extensions to the type system of OCL which help to use OCL earlier in the development of a system and to write more expressive constraints. We introduce intersection types, union types, operator abstraction, and bounded operator abstraction to the type system of OCL. Intersection types, which express that an object is an instance of all components of the intersection type, are more robust w.r.t. transformations of the contextual class diagram. Union types, which express that an object is an instance of at least one component of the union type, admit more constraints that have a meaning in OCL to be well-typed. Parametric polymorphism extends OCL to admit constraints on template without requiring that the template parameter is bound. Bounded parametric polymorphism allows one to specify assumptions on a template parameter. Together, our extensions result in a more flexible type system which admits more OCL constraints to be well typed without sacrificing adequacy or decidability.

3.1 Intersection Types

Consider the following constraint of class *Obs* in Fig. 4:

$$\textbf{context}\ \mathit{Obs}\ \mathit{inv} : a \rightarrow \mathit{union}(b).m1() \rightarrow \mathit{forAll}(x \mid x > 1)$$

This is a simple constraint which asserts that the value returned by $m1$ for each element in the collection of a and b is always greater than 1. We show that it is well-typed in OCL using the type system of Sect. 2.

$$\cfrac{\cfrac{\cfrac{a : \mathit{Bag}(D) \qquad b : \mathit{Bag}(C)}{a \rightarrow \mathit{union}(b) : \mathit{Bag}(A)}}{a \rightarrow \mathit{union}(b).m1() : \mathit{Bag}(\mathit{Integer})}}{a \rightarrow \mathit{union}(b).m1() \rightarrow \mathit{forAll}(x \mid x > 1) : \mathit{Boolean}}$$

Now consider the following question: What happens to the constraint if we change the class diagram to the one in Fig. 5, which introduces a new class E that implements the common functions of classes C and D? The meaning of the constraint is not affected by this change. However, the OCL constraint is not well-typed anymore, as this derivation shows, where the type annotation error is

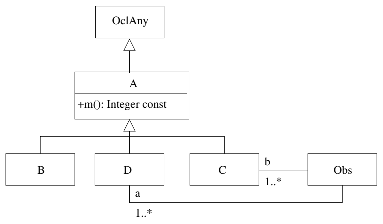

Fig. 4. A simple initial class diagram.

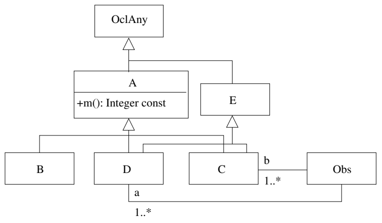

Fig. 5. The same diagram after a change.

used to state that the type system is not able to derive a type for the expression.[4]

$$\cfrac{\cfrac{a : Bag(D) \quad b : Bag(C)}{a \rightarrow union(b) : Bag(OclAny)}}{a \rightarrow union(b).m1() : \mathsf{error}}$$

The problem is, that the OCL type system chooses the unique and most precise supertype of a and b to type the elements of $a \rightarrow union(b)$, which now is *OclAny*, because we now have to choose one of A, E, and *OclAny*, which are the supertypes of C and D. Neither A nor E are feasible, because the type of the expression has to be chosen now. Hence, we are forced to choose *OclAny*. To avoid this problem constraints should be written once the contextual class diagram does not change anymore. Otherwise all constraints have to be updated, which if it is done by hand is a time consuming and error prone task.

The mentioned insufficiency of the type system can be solved in two ways: We can implement a transformation which updates all constraints automatically after such a change, or we introduce a more permissive type system for OCL. Because an automatic update of all constraints entails an analysis of all constraints in *the same way* as performed by the more permissive type system, therefore we extended the type system and leave the constraints unchanged.

The proposed extension is the introduction of intersection types. An intersection type, written $T \wedge T'$ for types T and T' states that an object is of type T and T'. Because $\wedge$ is both an associative and commutative operator, we introduce the generalized intersection $\bigwedge_{T \in \mathcal{T}} T$. In this paper $\mathcal{T}$ is always a *finite* set of types. The empty intersection type $\bigwedge \emptyset$ is the top type, which does not have any instances (and therefore is equivalent to *OclVoid*). Intersection types are useful to explain multiple inheritance [10–12].

We add the rules of Fig. 6 to the type system, which introduces intersection types into the type hierarchy. The rule S-InterLB and S-Inter formalize the notion that a type T belongs to both types, and that $\wedge$ corresponds to the order-theoretic meet. The rule S-InterA allows for a convenient interaction with operation calls and functions. This extension of the type system already solves the problem raised for the OCL constraint in the context of Fig. 5, as the derivation in Fig. 7 demonstrates.

[4] Recall, that we do not explicitly write the context in examples if it is clear.

$$\Gamma \vdash \bigwedge_{T' \in \mathcal{T}} T' \leq T \text{ for any } T \in \mathcal{T} \quad \text{S-InterLB}$$

$$\frac{\Gamma \vdash T \leq T' \text{ for all } T' \in \mathcal{T}}{\Gamma \vdash T \leq \bigwedge_{T' \in \mathcal{T}} T'} \ \text{S-Inter} \qquad \Gamma \vdash \bigwedge_{T' \in \mathcal{T}} (T \to T') \leq T \to \bigwedge_{T' \in \mathcal{T}} T' \quad \text{S-InterA}$$

Fig. 6. Intersection Types

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{a : Bag(D) \quad D \leq A \quad D \leq E}{a : Bag(A \wedge E)} \qquad \cfrac{b : Bag(C) \quad C \leq A \quad C \leq E}{b : Bag(A \wedge E)}}{a \to union(b) : Bag(A \wedge E)}}{a \to union(b) : Bag(A)}}{a \to union(b).m1() : Bag(Integer)}}{a \to union(b).m1() \to forAll(x \mid x > 1) : Boolean}$$

Fig. 7. Type checking with intersection types.

The extension of the OCL type system with intersection types is sufficient to deal with transformations which change the class hierarchy by moving common code of a class into a new super-class. This extension is also safe, and does not change the decidability of the type system.

3.2 Union Types

Union types are dual to intersection types. They are not as useful as intersection types, because they do not address a fundamental language concept like multiple inheritance. They can be used to address overloading of operators, and they do solve type checking problems for collection literals and the union operation of collections in OCL. We explain this by the class diagram in Fig. 8. Consider

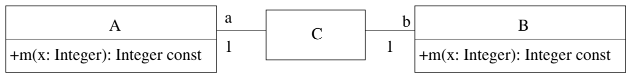

Fig. 8. A simple example class diagram.

the expression *context C inv* : $Set\{self.a, self.b\}.m(1)$ on this class diagram.[5] Assuming that the multiplicities of the associations are 1, we have the derivation

$$\cfrac{\cfrac{a : A \qquad b : B}{Set\{a, b\} : Set(OclAny)}}{Set\{a, b\}.m(1) : \mathsf{error}}$$

[5] Note that $a : A$ and $b : B$, and both classes define a method $m()$ returning an *Integer*. However, in this case the intended meaning of the constraint is $Set\{self.a.m(1), self.b.m(1)\}$, which is well-defined.

even though both types A and B define the property $m(x : \mathit{Integer}) : \mathit{Integer}$. Here, it is desirable to admit the constraint as well-typed, because it also has a meaning in OCL. Using intersection types does not help here, because stating that a and b have the type $A \wedge B$ is not adequate.

Instead, we want to judge that a and b have the type A *or* B. For this purpose we propose to introduce the union type $A \vee B$. A union type states, that an object is of type A or it is of type B. Again, $\vee$ is associative and commutative, so we introduce the generalized union $\bigvee_{T \in \mathcal{T}} T$. The type $\bigvee \emptyset$ is the universal type, a supertype of *OclAny*, of which any object is an instance. Union types are characterized by the rules in Fig. 9. Rules S-UnionUB and S-Union formalize

$$T \leq \bigvee_{T' \in \mathcal{T}} T' \text{ for any } T \in \mathcal{T}. \quad \text{S-UnionUB}$$

$$\frac{T' \leq T \text{ for any } T' \in \mathcal{T}}{\bigvee_{T' \in \mathcal{T}} T' \leq T} \text{ S-Union} \qquad \bigwedge_{T' \in \mathcal{T}} (T' \to T) \leq \left(\bigvee_{T' \in \mathcal{T}} T'\right) \to T \quad \text{S-UnionA}$$

Fig. 9. Rules for union types.

the fact that a union type is the least upper bound of two types. Note that it only makes sense to use a property on objects of $A \vee B$ that are defined for A and B. This is stated by the rule S-UnionA.

Using our extended type system, we can indeed derive that our example has the expected type.

$$\frac{\dfrac{a : A \quad b : B}{\mathit{Set}\{a, b\} : \mathit{Set}(A \vee B)} \qquad \dfrac{m : A \to \mathit{Integer} \quad m : B \to \mathit{Integer}}{m : A \vee B \to \mathit{Integer}}}{\mathit{Set}\{a, b\} \to \mathit{collect}(m()) : \mathit{Bag}(\mathit{Integer})}$$

3.3 Parametric Polymorphism

UML 2.0 provides the user with templates[6] (see [3, Sect. 17.5, pp. 541ff.]), which are functions from types or values to types, i.e., they take a type as an argument and return a new type. We first consider the case where the parameter of a class ranges over types. This form of parametric polymorphism is highly useful, as shown, e.g., in [13] and [14]. Adequate support for parametric polymorphism in the specification language is again highly useful, as the proof of a property of a template carries over to all its instantiations [15]. The OCL standard library contains the collection types, which are indeed examples of generic classes.

We have not found yet how the parameter of a template, which is defined in the class diagram, is integrated into OCL's type system. In fact, it is not defined in the proposal how the environment has to be initialized in order to parse

[6] Also called generics or parameterized classes

expressions according to the rules of Chapter 4 of [1]. For example, consider the following constraint:

$$
\begin{array}{ll}
& \textbf{context}\ \mathit{Sequence} :: \mathit{excluding}(\mathit{object} : T) : \mathit{Sequence}(T) \\
(1) & \mathit{post} : \mathit{result} = \mathit{self} \rightarrow \mathit{iterate}(\mathit{elem}; \mathit{acc} : \mathit{Sequence}(T) = \mathit{Sequence}\{\} \mid \\
& \quad \textbf{if}\ \mathit{elem} = \mathit{object}\ \textbf{then}\ \mathit{acc}\ \textbf{else}\ \mathit{acc} \rightarrow \mathit{append}(\mathit{object})\ \textbf{endif})
\end{array}
$$

To what does T refer to? Currently, T is not part of the type environment, because it is neither a classifier nor a state but an instance of *TemplateParameter* in the UML metamodel. This constraint is, therefore, not well-typed. But it is worthwhile to admit constraints like (1), because this constraint is valid for any instantiation of the parameter T.

UML 2.0 allows different kinds of template parameters: parameters ranging over classifiers, parameters ranging over value specifications, and parameters ranging over features (properties and operations). In this paper, we only consider parameters ranging over classifiers.

We propose to extend the environment such that Γ contains the kinding judgment $T \in \star$ if T is the parameter of a template. This states that the parameter of a template is a type. Also note that the name of the template classifier alone is not of the kind $\star$ but of some kind $\star \rightarrow \cdots \rightarrow \star$, depending on the number of type parameters. Additionally, we give the following type checking rules for templates in Fig. 10. These rules generalizes the conforms-to relation previously defined for collection types only. The rule S-InstSub states that if a template

$$
\frac{\Gamma \vdash T : K \rightarrow K' \quad \Gamma \vdash T' : K \rightarrow K' \quad \Gamma \vdash T'' : K \quad \Gamma \vdash T \leq T'}{\Gamma \vdash T(T'') \leq T'(T'')}\ \textsc{S-InstSub}
$$

$$
\frac{\Gamma \vdash T : K \rightarrow K' \quad \Gamma \vdash T' : K \quad \Gamma \vdash T'' : K \quad \Gamma \vdash T' \leq T''}{\Gamma \vdash T(T') \leq T(T'')}\ \textsc{S-InstSub-2}
$$

Fig. 10. Subtyping rules for parametric polymorphism

class T is a subtype of another template class T', then T remains a subtype of T' for any class T'' bound to the parameter. This rule is always adequate. The rule T-InstSub-2 states that for any template class T and any types T' and T'' such that T' is a subtype of T'', then binding T' and T'' to the type parameter in T preserves this relation. Note that rule S-InstSub-2 is not always safe. The absence of side-effects in OCL expressions are a fundamental property for the validity of the rule S-Arrow and therefore also for S-InstSub-2. The following counter-example illustrates the importance of the absence of side-effects for the type system. Consider the following fragment of C++ code:

```
class C { public: void m(double *&a) { a[0] = 1.5; } }
void main(void) { int *v = new int[1]; C *c = new C(); c->m(v); }
```

If we allow the S-InstSub-2, then the call `c->m(v)` is valid, because `int` is a subtype of `double`. But within the body of `m` the assignment `a[0] = 1.5` would store a double value into an array of integers, which is not allowed. However, since we assume that each expression is free of side-effects, rule S-InstSub-2 is adequate.

3.4 Bounded Operator Abstraction

While parametric polymorphism in the form of templates is useful in itself, certain properties still cannot be expressed directly as types but have to be expressed in natural language. For example, in the OCL standard the collection property `sum()` of set has the following specification:

> The addition of all elements in *self*. Elements must be of a type supporting the + operation. The + operation must take one parameter of type T and be both associative: $(a + b) + c = a + (b + c)$, and commutative: $a + b = b + a$. Integer and Real fulfill this condition.

Formally, the post condition of sum does not type check, because a type checker has no means to deduce that T indeed implements the property + as specified. The information can be provided in terms of bounded polymorphism, where the type variable is bounded by a super type. The properties of + can be specified in an abstract class (or interface), say *Sum*, and the following constraints:

$$\begin{array}{l} \textbf{context}\ \mathit{Sum} \\ \mathit{inv} : \mathit{self}.\mathit{typeOf}().\mathit{allInstances}() \rightarrow \mathit{forAll}(a, b, c \mid \\ \qquad a + (b + c) = (a + b) + c) \\ \mathit{inv} : \mathit{self}.\mathit{typeOf}().\mathit{allInstances}() \rightarrow \mathit{forAll}(a, b \mid a + b = b + a) \end{array} \tag{2}$$

In Eq. (2) the property *typeOf*() is supposed to return the run-time type of the *self* object. It is important to note that we cannot write *Sum.allInstances*, because the type implementing *Sum* need not provide an implementation of + which work uniformly on all types implementing *Sum*. For example, we can define + on *Real* and on *Vectors* of *Reals*, but it may not make sense to implement an addition operation of vectors to real which returns a real. So we do not want to force the modeler to do this. The purpose of *Sum* is to specify that a classifier provides an addition which is both associative and commutative.

When *Sum* is a base class of a classifier T, and we have a collection of instances of T, then we also know that the property *sum* is defined for this classifier. So *Sum* is a lower bound of the types of T. Indeed, the signature of $\mathit{Collection} :: \mathit{sum}$ can be specified by $\mathit{Collection}(T \leq \mathit{Sum}) :: \mathit{sum}() : T$, which expresses the requirements on T.

Syntactically, we express the type of a bounded template using the notation $\Lambda\tau.C(\tau \leq S)$. For this new type constructor we have to define a new kind $\Pi\tau \leq S \to \star$, where S is of kind $\star$. This new kind $\Pi\tau \leq S$ states that τ has to be a subtype of S to construct a new type, otherwise the type is not well-kinded.

Observe that type operators are not comparable using the subtype relation. Therefore, bounded operator abstraction does not introduce new rules into the typing system.

3.5 Flattening and accessing the run-time type of objects

Quite often it is necessary to obtain the type of an object and compare it. OCL provides some functions which allow the inspection and manipulation of the run-time type of objects. To test the type of an object it provides the operations *oclIsTypeOf()* and *oclIsKindOf()*, and to *cast* or coerce an object to another type it provides *oclAsType()*. In OCL, we also have the type *OclType*, of which the values are the names of all classifiers appearing in the contextual class diagrams.[7] The provided mechanisms are not sufficient, as the specification of the *flatten()* operation shows (see [1]):

$$\begin{aligned}
&\textbf{context } \mathit{Set} :: \mathit{flatten}() : \mathit{Set}(T2)\\
&\mathit{post} : \mathit{result} = \textbf{if } \mathit{self.type.elementType.oclIsKindOf}(\mathit{CollectionType})\\
&\quad \textbf{then } \mathit{self} \rightarrow \mathit{iterate}(c; \mathit{acc} : \mathit{Set}() = \mathit{Set}\{\} \mid \mathit{acc} \rightarrow \mathit{union}(c \rightarrow \mathit{asSet}()))\\
&\quad \textbf{else } \mathit{self}\\
&\quad \textbf{endif}
\end{aligned} \tag{3}$$

This constraint contains many errors. First, the type variable *T2* is not bound in the model (see Sect. 3.3 for the meaning of binding), so it is ambiguous whether *T2* is a classifier appearing in the model or a type variable. Next, self is an instance of a collection kind, so the meaning of *self.type* is actually a shorthand for *self→collect(type)*, and there is no guarantee that each instance of the collection defines the property *type*. Of course, the intended meaning of this sub-expression is to obtain the element-type of the members of *self*, but one cannot access the environment of a variable from OCL. Next, the type of the accumulator in the iterate expression is not valid, *Set* requires an argument, denoting the type of the elements of the accumulator set (one could use *T2* as the argument).

The obvious solution, to allow the type of an expression depending on the type of other expressions, poses a serious danger: If the language or the type system is too permissive in what is allowed as a type, we cannot algorithmically decide, whether a constraint is well-typed or not. But decidability is a desirable property of a type-system. Instead, we propose to treat the *flatten()* operation as a kind of literal, like *iterate* is treated. For *flatten*, we introduce the following two rules:

$$\frac{e : C(T) \qquad C \leq \mathit{Collection} \qquad T \leq \mathit{Collection}(\tau')}{e \rightarrow \mathit{flatten}() : C(\tau')}\ \text{T-F}\textsc{lat}$$

$$\frac{e : C(T) \qquad C \leq \mathit{Collection} \qquad T \not\leq \mathit{Collection}(\tau')}{e \rightarrow \mathit{flatten}() : C(T)}\ \text{T-NF}\textsc{lat}$$

The rule T-Flat covers the case where we may flatten a collection, because its element type conforms to a collection type with element type T'. In this case, T' is the new collection type. The rule T-NFlat covers the case where the collection e does not contain any other collections. In this case, the result type of *flatten* is the type of collection e.

[7] The type *OclType* will be removed but still occurs in the proposal.

These rules encode the following idea: For each collection type we define an *overloaded* version of flatten. As written in Sec. 3.2, we are able to define the type of any overloaded operation using a union type. However, using this scheme directly yields infinitary union types, because the number of types for which we have to define a flatten operation is not bounded. The price for this extension is decidability [16].

The drawback of this extension is that the meaning of the collection cannot be expressed in OCL, because we have no way to define T' in OCL. The advantage is, that the decidability of the type system extended in this way is not affected.

4 Adequacy and Decidability

In this section, we summarize the most important results concerning the extended type system. This means that if the type system concludes that an OCL-expression has type T, then the result of evaluating the expression yields a value of a type that conforms to T. The type system is adequate and decidable. For the (operational) semantics of OCL we use the one defined in [4, 1].

Theorem 1. *Let Γ be a context, e an OCL expression, and T a type such that $\Gamma \vdash e : T$. Then the value of e conforms to T.*

Proof. Similar to the one presented in [5] and in [17]. □

Theorem 2. *Let Γ be a context and e be an OCL expression. Then there exists an algorithm which computes a type T such that $\Gamma \vdash e : T$ or returns an* error *if no such type can be found by the type system.*

Proof. Follows from [17] and [9], since our type system is a special case of the type systems used there. □

Our type-checking algorithm is based on [17] and [9] but handles union types. It is simpler than the cited ones, because we only have a form of bounded operator abstraction, where type abstractions are not comparable, except for collection types, which is crucial in the proof of decidability. Furthermore, the kind of polymorphism in our type system is ML-like, where type variables (the template parameters) are universally quantified.

However, the type system is incomplete. By this we mean that if e is an OCL expression the type system will not compute the most precise type of e, but one of its supertypes. One reason for incompleteness is the following: If e is a constraint whose evaluation does not terminate, its most specific type is *OclVoid*. But we cannot decide whether the evaluation of a constraint will always terminate.

Indeed, the type system presented in this paper covers a usable set of features and it is still decidable. If we, e.g., also add checking for value specifications of method specifications of templates to the type system, it would become undecidable [18]. Such type systems indeed form the theoretical foundation of interactive theorem provers.

5 Related Work and Conclusions

A type system for OCL has been presented by Clark in [8], by Richter and Gogolla in [19], and by Cengarle and Knapp in [5]. In Sect. 2 we summarized these results and give a formal basis for our proposal.

A. Schürr has described an extension to the type system of OCL [20], where the type system is based on set approximations of types. These approximations are indeed another encoding of intersection and union types. His algorithm does not work with parameterized types and bounded polymorphism, because the normal forms of types required for the proof of Theorem 2 cannot be expressed as finite set approximations. We extended OCL's type system to also include polymorphic specifications for OCL constraints, which is not done by Schürr.

Our type system is a special case of the calculus $F^{\omega}_{\wedge}$. This system is analyzed in [17], where a type checking algorithm is given. This calculus is a conservative extension of $F^{\omega}_{\leq}$. M. Steffen has described a type checking algorithm for $F^{\omega}_{\leq}$ with polarity information [9]. Our type system does not allow type abstractions in expressions and assumes that all type variables are universally quantified in prenex form.

We have presented extensions to the type system for OCL, which admits a larger class of OCL constraints to be well-typed. Furthermore, we have introduced extensions to OCL, which allow to write polymorphic constraints.

The use of intersection types simplifies the treatment of multiple inheritance. This extension makes OCL constraints robust to changes in the underlying class diagram, e.g., refactoring by moving common code into a superclass. Intersection types are therefore very useful for type-checking algorithms for OCL. Union types simplify the treatment of collection literals, model operator overloading elegantly, and provides unnamed supertypes for collections and objects. Parametric polymorphism as introduced by UML 2.0's templates is useful for modeling. We described how polymorphism may be integrated into OCL's type system and provided a formal basis in type checking algorithms. Bounded parametric polymorphism is even more useful, because it provides the linguistic means to specify assumptions on the type of the type parameters.

We have proposed typing rules for certain functions which can not be formally expressed in OCL. We have shown that this type system is sound, adequate, and decidable.

References

1. Boldsoft and Rational Software Corporation and IONA and Adaptive Ltd.: UML 2.0 OCL Specification. (2003) `http://www.omg.org/cgi-bin/doc?ptc/2003-10-14`.
2. Object Management Group: UML 2.0 Infrastructure Specification. (2003) `http://www.omg.org/cgi-bin/doc?ptc/2003-09-15`.
3. Object Management Group: UML 2.0 Superstructure Specification. (2004) `http://www.omg.org/cgi-bin/doc?ptc/2004-10-02`.

4. Richters, M.: A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis, Universität Bremen (2002) Logos Verlag, Berlin, BISS Monographs, No. 14.
5. Cengarle, M.V., Knapp, A.: OCL 1.4/5 vs. 2.0 expressions: Formal semantics and expressiveness. Software and Systems Modeling (2003)
6. Kyas, M., Fecher, H., de Boer, F.S., van der Zwaag, M., Hooman, J., Arons, T., Kugler, H.: Formalizing UML models and OCL constraints in PVS. In Lüttgen, G., Madrid, N.M., Mendler, M., eds.: Proc. SFEDL 2004. Volume 115 of ENTCS., Elsevier (2005) 39–47
7. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
8. Clark, A.: Typechecking UML static models. In France, R.B., Rumpe, B., eds.: Proc. UML'99. Number 1723 in LNCS, Springer-Verlag (1999) 503–517
9. Steffen, M.: Polarized Higher-Order Subtyping. PhD thesis, Technische Fakutät, Friedrich-Alexander-Universität Erlangen-Nürnberg (1997)
10. Cardelli, L., Wegener, P.: On understanding types, data abstraction, and polymorphism. ACM Computing Surveys **17** (1985) 471–522
11. Pierce, B.C.: Programming with Intersection Types and Bounded Polymorphism. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 (1991)
12. Compagnoni, A.B., Pierce, B.C.: Intersection types and multiple inheritance. Mathematical Structures in Computer Science **6** (1996) 469–501
13. Meyer, B.: Eiffel: The Language. Prentice Hall (1992)
14. Meyer, B.: Object-Oriented Software Construction. 2nd edn. Prentice Hall (1997)
15. Wadler, P.L.: Theorems for free! In: Fourth International Conference on Functional Programming Languages and Computer Architecture, ACM Press (1989) 347–359
16. Bonsangue, M.M., Kok, J.N.: Infinite intersection types. Information and Computation **186** (2003) 285–318
17. Compagnoni, A.B.: Higher-order subtyping and its decidability. Information and Computation **191** (2004) 41–113
18. Coquand, T., Huet, G.: The calculus of constructions. Information and Computation **76** (1988) 95–120
19. Richters, M., Gogolla, M.: OCL: Syntax, semantics, and tools. [21] 42–68
20. Schürr, A.: A new type checking approach for OCL 2.0? [21] 21–40
21. Clark, T., Warmer, J., eds.: Object Modelling with the OCL. In Clark, T., Warmer, J., eds.: Object Modelling with the OCL. Number 2263 in LNCS, Springer-Verlag (2002)