Verifying OCL specifications of UML models : tool support and compositionality

Kyas, M.

Chapter 3

Type Checking OCL

We describe the design and implementation of a type-checker for OCL. Based on the experience gained in implementing and using this type checker, we observed that OCL is not suitable for constraining systems under development, because changes in the underlying class diagram unnecessarily invalidate the type correctness of constraints, whereas the semantic value of these constraints do not change. A second observation was, that the type system specified in the OCL standard does not support all language features defined for UML class diagrams and part of the OCL standard library. OCL and UML support parameterised classes (see Section 2.1.3), for example, the *Set* type of OCL, but cannot take advantage of these type abstractions, not only during type checking.

To alleviate these problems, the type system of OCL has been extended with intersection types, union types, and bounded operator abstraction. This allows to take advantage of the universal polymorphism offered by parameterised classes, and makes the well-typedness of constraints more robust during the evolution of the contextual class diagram.

3.1 Introduction

In order to be used widely, OCL needs to have a type system that is compatible with the well-formedness constraints of UML 2.0 class diagrams and that integrates with this type system seamlessly. Furthermore, the type system has to be robust with respect to model transformations, for example, refactoring[1] or other changes in class diagrams.

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

[1] *Refactoring*, a term initially introduced by William F. Opdyke and Ralph E. Johnson in [119], is the process of rewriting a program to improve its readability or structure, with the explicit purpose of preserving its meaning and behaviour. The main purpose of refactoring is to improve the internal consistency, remove redundancy, and often to simplify the structure. Refactoring is often automated, using a catalogue of known refactoring methods. More information on refactoring can be found in Opdyke's thesis [118] and in the book of Fowler et al [55].

The type system of OCL appears to be designed for languages which only use single inheritance and no parameterised classes[2]. UML 2.0 introduces a new model for templates, which allows classifiers to be parameterised with classifier specifications, value specifications, and operation specifications [111]. The OCL 2.0 proposal [113] 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, as explained in Section 3.3. These operations, which often have no effect on the semantics 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 [126] 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 chapter is organised as follows: In Section 3.2, we survey the current type system for OCL. In Section 3.3, we describe our different extensions to the type system. In Section 3.4, we summarise the most important results. In Section 3.5 we compare our results with other results and draw some conclusions.

3.2 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 [25].

Recall that the abstract syntax of OCL has been defined in Section 2.3.

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, that is, they construct a type by applying the collection type to any other type.[3] This distinction is formalised 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

[2]For example Java before version 5.0.

[3]In Section 3.4 we discuss the presence of *dependent types* in class diagrams.

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:

$$\tau ::= type \mid \tau(\tau_1) \mid \tau_0 \times \cdots \times \tau_n \to \tau$$

Here, a *type* is any classifier or template appearing in the contextual class diagram or the OCL standard library. The expression $\tau(\tau_1)$ expresses the type which results from instantiating a template parameter of the type τ with τ_1. The type $\tau_0 \times \cdots \times \tau_n \to \tau$ is used to express the type of properties. The type τ_0 is the type of the classifier which defines the property, the types $\tau_1, \ldots, \tau_n$ are the types of the parameters of the property. We identify attributes with operations that do not define arguments. 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 Table 3.1.

We write the typing rules and proofs in (the usual) natural deduction 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.

$$\frac{\tau \text{ is a type or property type}}{\tau : \star}\ \text{K-Elem}$$

$$\frac{\tau \text{ is a parameterised class}}{\tau : \star \to \star}\ \text{K-Cons}$$

$$\frac{\tau_1 : K \qquad \tau : K \to K'}{\tau(\tau_1) : K'}\ \text{K-Inst}$$

Table 3.1: Kinding system

An important property of OCL is, that all types have to be defined in the contextual class diagram, with the only exception of tuple types. Tuple types (or structures) are data types. This simplifies the type checking rules a lot, because new types do not arise from the constraints to be checked. 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 $\Gamma, v : \tau$ to denote the context extended by binding the variable v to τ, provided that v does not occur in Γ. We write $\Gamma(v) = \tau$ to state that v has type τ in context Γ. The context also contains the information on *type conformance*, that is, clauses of the form $\tau \leq \zeta$ derived from the

generalisation hierarchy, where τ and ζ are types and $\leq$ denotes that τ is a subtype of ζ. We write $\Gamma, \tau \leq \zeta$ to extend a context with a statement that τ is a subtype of ζ. Any context contains $\tau \leq \tau$ for every type τ occurring in the model, since the conformance relation is reflexive. If the context Γ contains the declaration $\tau \leq \zeta$ we represent this with $\Gamma \vdash \tau \leq \zeta$. We write $\Gamma \vdash t : \tau$ to denote that t is a term of type τ 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 contra-variant in its result type. These rules are shown in Table 3.2. 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*.

The type operator *Collection* refers to a parameterised class. The rule S-Coll-2 is not sound for classes which define operations which change the contents of the collection. 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-Coll-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); }
```

Using the rule S-Coll-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-Coll-2 is adequate.

In OCL operations defined on collections do not alter the contents, but we cannot assume this in general; therefore, we defined these particular assumptions. The typing rules for terms are presented in Tables 3.3 and 3.4, 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 Section 3.3.5.

Rules T-True, T-False, and T-Lit assign to each literal their type. Especially, T-Lit is an axiom scheme assigning, for example, the literal 2 the type *Integer* and the literal 2.5 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 result has the declared type. The antecedent $\Gamma \vdash C \not\leq \mathit{Collection}$ denotes that in context Γ type C is not a subtype of *Collection*.

Note that it is decidable whether a type is a subtype of another, provided the class diagram is finite. We have two cases to consider: If the two types are basic types, that is, not parameterised types. Then we can easily test whether a type is a subtype of another type. Otherwise we have to compare two *instantiations* of parameterised types.

$$\frac{\Gamma \vdash C \leq \mathit{Collection}}{\Gamma \vdash C(\tau) \leq \mathit{Collection}(\tau)}\ \text{S-Coll}$$

$$\frac{\Gamma \vdash C \leq \mathit{Collection} \quad \Gamma \vdash \tau \leq \tau'}{\Gamma \vdash C(\tau) \leq C(\tau')}\ \text{S-Coll-2}$$

$$\frac{\Gamma \vdash e : \zeta \quad \Gamma \vdash \zeta \leq \tau}{\Gamma \vdash e : \tau}\ \text{S-Sub}$$

$$\frac{\Gamma \vdash \zeta \leq \tau \quad \Gamma \vdash \tau \leq \upsilon}{\Gamma \vdash \zeta \leq \upsilon}\ \text{S-Trans}$$

$$\frac{\Gamma \vdash \tau_0 \leq \zeta_0, \Gamma \vdash \zeta_1 \leq \tau_1, \cdots, \Gamma \vdash \zeta_n \leq \tau_n \quad \Gamma \vdash \tau \leq \zeta}{\Gamma \vdash \tau_0 \times \tau_1 \times \cdots \times \tau_n \to \tau \leq \zeta_0 \times \zeta_1 \times \cdots \times \zeta_n \to \zeta}\ \text{S-Arrow}$$

Table 3.2: Definition of type conformance

Since these two are finite terms, we can recursively check whether the type arguments are subtypes of each other, and, if this succeeds, whether the type operators are related in the class diagram.

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 τ, then the conditional expression has that type τ. 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 initialised 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-RLet 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 [113], but we have omitted the typing rules for the boolean connectives, as they are given by Cengarle and Knapp in [23], because each expression using a boolean connective can be rewritten to an operation call expression, for example, *a and b* is equivalent to

$$\frac{}{\Gamma \vdash \text{true} : \mathit{Boolean}}\ \text{T-True}$$

$$\frac{}{\Gamma \vdash \text{false} : \mathit{Boolean}}\ \text{T-False}$$

$$\frac{}{\Gamma \vdash l : \tau_l}\ \text{T-Lit}$$

$$\frac{\Gamma(v) = \tau}{\Gamma \vdash v : \tau}\ \text{T-Var}$$

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_n : \tau_n}{\mathit{Tuple}\{a_1 = e_1, \ldots, a_n = e_n\} : \mathit{Tuple}\{a_1 : \tau_1, \ldots, a_n : \tau_n\}}\ \text{T-Tuple}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash t_0 : \zeta \quad \Gamma \vdash t_1 : \zeta_1, \cdots, \Gamma \vdash t_n : \zeta_n \\ \Gamma \vdash m : \zeta \times \zeta_1 \times \cdots \times \zeta_n \to \tau \quad \Gamma \vdash \zeta \not\leq \mathit{Collection}\end{array}}{\Gamma \vdash t_0.m(t_1, \ldots, t_n) : \tau}\ \text{T-Call}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_0 : C(\zeta) \quad \Gamma \vdash t_1 : \zeta_1, \cdots, \Gamma \vdash t_n : \zeta_n \\ \Gamma \vdash m : C(\zeta) \times \zeta_1 \times \cdots \times \zeta_n \to \tau \quad \Gamma \vdash C \leq \mathit{Collection}\end{array}}{\Gamma \vdash t_0 \to m(t_1, \ldots, t_n) : \tau}\ \text{T-CCall}$$

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

Table 3.3: Typing rules for OCL

$$\frac{\Gamma \vdash t_0 : \tau_0 \quad \Gamma, v_0 : \tau_0 \vdash t : \tau}{\Gamma \vdash \textbf{let } v_0 : \tau_0 = t_0 \textbf{ in } t \ : \tau} \text{ T-Let}$$

$$\frac{\begin{array}{l}\Gamma' = \Gamma, \quad v_0 : \tau_{0,0} \times \cdots \times \tau_{0,m_0} \to \tau_0, \\ \qquad\quad \ldots, \\ \qquad\quad v_n : \tau_{n,0} \times \cdots \times \tau_{n,m_n} \to \tau_n \\ \Gamma', v_{0,0} : \tau_{0,0}, \ldots, v_{0,m_0} : \tau_{0,m_0} \vdash t_0 : \tau_0 \\ \qquad\qquad\qquad \vdots \\ \Gamma', v_{n,0} : \tau_{n,0}, \ldots, v_{n,m_n} : \tau_{n,m_n} \vdash t_n : \tau_n \\ \qquad\qquad \Gamma' \vdash t : \tau\end{array}}{\begin{array}{ll}\Gamma \vdash & \textbf{let} \quad v_0(v_{0,0}, \ldots, v_{0,m_0}) : \tau_0 = t_0, \\ & \qquad \ldots, \\ & \qquad v_n(v_{n,0}, \ldots, v_{n,m_n}) : \tau = t_n \\ & \textbf{in } t \ : \tau\end{array}} \text{ T-RLet}$$

Table 3.4: Typing rules for OCL (continued)

a.and(*b*). We do not have a rule for the undefined value, as presented in [23], because the OCL 2.0 proposal does not define a literal for undefined [113, pp. 48–50].[4]

Within the UML 2.0 standard [111] and the OCL 2.0 standard [113] methods are redefined co-variantly. 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 modelled with union-types in our system (see Section 3.3.2), as suggested in, for example, by Pierce [126, 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 Table 3.2. An example of where we use the largest common supertype can be found in Section 3.3.1. 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 and decidable.[5]

[4]Note that *OclInvalid* is the type of any undefined expression, which does not have a literal for its instances [113, p. 133]. Calling the property *oclIsUndefined*(), defined for any object, is preferred, because any other property call results in an instance of *OclInvalid*.

[5]For a type system with unique typing, first rules computing a normal form for each type have to be defined. These rules have to form a confluent rewriting system, otherwise the type system is not decidable. Then an equality rule for types is needed. Finally, a priority on the rules has to be declared and it has to be asserted that rules of the same priority have mutually exclusive antecedents. Details on this procedure can be found in [30] or in [145].

Proposition 3.1. *The type system is* adequate*, that is, for any OCL expression e we have: if* $e : \tau$ *can be derived in the type system, then e is evaluated to a result of a type conforming to* τ*.*

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

A proof of this proposition has been provided by Cengarle and Knapp in [23].

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

3.3 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, 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 with respect to 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 templates 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.

3.3.1 Intersection Types

Consider the following constraint of class *Obs* in Figure 3.1:

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

This is a simple constraint which asserts that the value returned by *m* 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 Section 3.2.

$$\dfrac{\dfrac{\dfrac{a : \mathit{Bag}(D) \quad b : \mathit{Bag}(C)}{a \to \mathit{union}(b) : \mathit{Bag}(A)}\ \text{T-CC}_{\text{ALL}}}{a \to \mathit{union}(b).m() : \mathit{Bag}(\mathit{Integer})}\ \text{T-C}_{\text{ALL}}}{a \to \mathit{union}(b).m() \to \mathit{forAll}(x \mid x > 1) : \mathit{Boolean}}\ \text{T-CC}_{\text{ALL}}$$

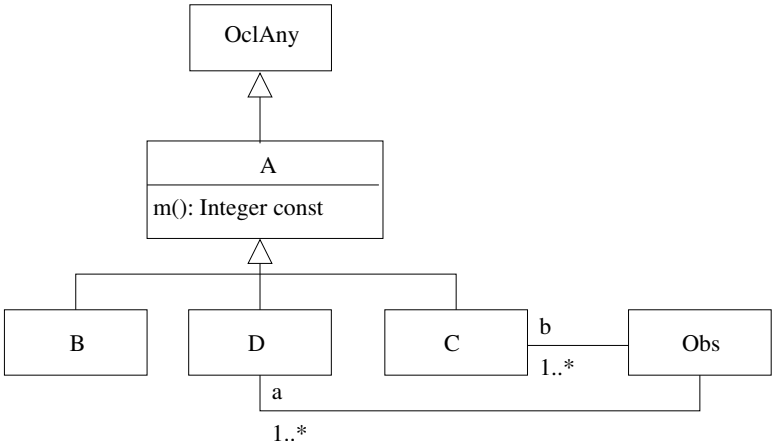

Figure 3.1: A simple initial class diagram

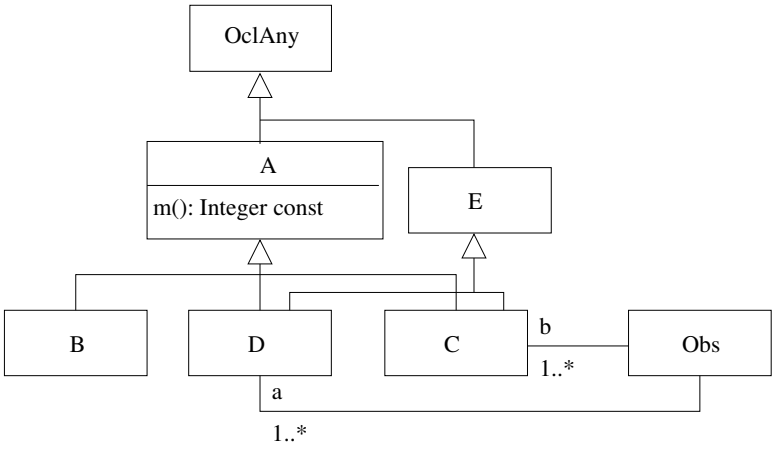

Figure 3.2: The same diagram after a change

Now consider the following question: What happens to the constraint if we change the class diagram to the one in Figure 3.2, 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 $\lightning$ is used to state that the type system is not able to derive a type for the expression.[6]

$$\dfrac{\dfrac{a : Bag(D) \quad b : Bag(C)}{a \to union(b) : Bag(A)}\,\text{T-CC}_{\text{ALL}}}{a \to union(b).m() : \lightning}$$

The problem is, that the OCL type system chooses the unique and most precise supertype of a and b to type the elements of $a \to union(b)$, 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 $\tau \wedge \tau'$ for types τ and τ' states that an object is of type τ and τ'. Because $\wedge$ is both an associative and commutative operator, we introduce the generalised intersection $\bigwedge_{\tau \in \mathcal{T}} \tau$. In this chapter $\mathcal{T}$ is always a *finite* set of types.

Recall, that the generalisation hierarchy is a partially ordered set. Intersection types have been defined, such that they are compatible with generalisation. It can be easily shown, that the set of types and their intersection types form a lattice with the empty intersection type $\bigwedge \emptyset$ as the top type (cf. Pierce [126]).

The empty intersection $\bigwedge \emptyset$ is a subtype of any other type and does not have any instances; therefore, it is equivalent to *OclVoid*, which also does not have any instances and is a subtype of any other type.

Intersection types are very useful to explain multiple inheritance, a relation that has been investigated by Cardelli and Wegener [21], Pierce [125], and Compagnoni [31].

We add the rules of Table 3.5 to the type system, which introduces intersection types into the type hierarchy. The rule S-InterLB and S-Inter formalise the notion that a type

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

τ 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

$$\frac{\tau \in \mathcal{T}}{\Gamma \vdash \bigwedge_{\tau' \in \mathcal{T}} \tau' \leq \tau} \text{ S-InterLB}$$

$$\frac{\Gamma \vdash \tau \leq \tau' \text{ for all } \tau' \in \mathcal{T}}{\Gamma \vdash \tau \leq \bigwedge_{\tau' \in \mathcal{T}} \tau'} \text{ S-Inter}$$

$$\frac{}{\Gamma \vdash \left(\bigwedge_{\tau' \in \mathcal{T}} (\tau \to \tau') \right) \leq \left(\tau \to \bigwedge_{\tau' \in \mathcal{T}} \tau' \right)} \text{ S-InterA}$$

Table 3.5: Intersection types

extension of the type system already solves the problem raised for the OCL constraint in the context of Figure 3.2, as this derivation demonstrates:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{a : Bag(D) \quad D \leq A \quad D \leq E}{a : Bag(A \wedge E)} \text{ S-Inter} \qquad \dfrac{b : Bag(C) \quad C \leq A \quad C \leq E}{b : Bag(A \wedge E)} \text{ S-Inter}}{a \to union(b) : Bag(A \wedge E)} \text{ T-CCall}}{a \to union(b) : Bag(A)} \text{ S-InterLB}}{a \to union(b).m() : Bag(Integer)} \text{ T-Call}}{a \to union(b).m() \to forAll(x \mid x > 1) : Boolean} \text{ T-CCall}$$

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.

3.3.2 Union Types

Union types are dual to intersection types, and can be used to address type-checking of overloaded operators. Union types also solve type checking problems for collection literals and the union operation of collections in OCL. We explain this by the class diagram in Figure 3.3. Consider the expression **context** C **inv** : $Set\{self.a, self.b\}.m() \to$

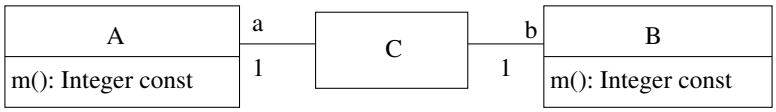

Figure 3.3: A simple example class diagram

$size() > 1$ on this class diagram.[7] Assuming that the multiplicities of the associations are 1, we have:

$$\dfrac{\dfrac{a : A \quad b : B}{Set\{a, b\} : Set(OclAny)}\,\text{T-Coll}}{Set\{a, b\}.m() : \lightning}$$

even though both types A and B define the property $m(x : Integer) : Integer$. 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 generalised union $\bigvee_{\tau \in \mathcal{T}} T$. The type $\bigvee \emptyset$ is the universal type, a supertype of *OclAny*, of which any object is an instance. Union types are characterised by the rules in Table 3.6. Rules S-UnionUB and S-Union formalise the fact that a union type is the least upper

$$\frac{\tau \in \mathcal{T}}{\tau \leq \bigvee_{\tau' \in \mathcal{T}} \tau'}\,\text{S-UnionUB}$$

$$\frac{\tau' \leq \tau \text{ for all } \tau' \in \mathcal{T}}{\bigvee_{\tau' \in \mathcal{T}} \tau' \leq \tau}\,\text{S-Union}$$

$$\frac{}{\left(\bigwedge_{\tau' \in \mathcal{T}} (\tau' \to \tau)\right) \leq \left((\bigvee_{\tau' \in \mathcal{T}} \tau') \to \tau\right)}\,\text{S-UnionA}$$

Table 3.6: Rules for union types

bound of two types. Note that it only makes sense to use a property on objects of $A \vee B$

[7] 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 $Set\{self.a.m(), self.b.m()\} \to size() > 1$, which is well-defined.

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.

$$\dfrac{\dfrac{\dfrac{\dfrac{a : A \quad b : B}{Set\{a, b\} : Set(A \vee B)}\,\text{S-UnionUB} \qquad \dfrac{m : A \to Integer \quad m : B \to Integer}{m : A \vee B \to Integer}\,\text{S-UnionA}}{Set\{a, b\}.m() : Bag(Integer)}\,\text{T-Call}}{Set\{a, b\}.m() \to size() : Integer}\,\text{T-CCall}}{Set\{a, b\}.m() \to size() > 0 : Boolean}\,\text{T-Call}$$

3.3.3 Parametric Polymorphism

UML 2.0 provides the user with templates (see [111, Section 17.5, pp. 541ff.]), which are also called generics or parameterised classes, which are functions from types or values to types, that is, 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. 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, which is explained in Wadler's beautiful paper "Theorems for Free!" [152]. The OCL standard library contains the collection types, which are indeed examples of generic classes.

The type of a generic class, which is indeed a type operator, that is a function which operates on types, is written as $\Lambda\tau \leq \zeta : \upsilon$. A type operator of this form creates a type $\upsilon[\tau/\tau']$ if it is applied to a type τ', which has to be a subtype of ζ. Therefore, one speaks of bounded operator abstraction (see also Section 3.3.4). Because parameterised classes define operations and methods, the methods are terms which are parameterised in the same way as their class, that is, the body b of a method is parameterised by τ. Therefore, we speak of parametric polymorphism.

Example 3.1. Consider the OCL type operator $\Lambda\tau \leq OclAny : Set(\tau)$ (we have made the type abstraction explicit in the type name). Independent of the concrete type instantiated for τ, we can easily reason about the behaviour of each operation defined for $\Lambda\tau \leq OclAny : Set(\tau)$, because, for example, the definition of *count* given in Section 2.3.4 does not depend on any property of the objects contained in an instance of an instance of $\Lambda\tau \leq OclAny : Set(\tau)$. ♦

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 initialised in order to parse expressions according to the rules of Chapter 4 of [113]. For example, consider the following constraint:

$$\begin{aligned}&\textbf{context } Sequence :: excluding(o : \tau) : Sequence(\tau)\\ &\textbf{post} : result = self \to iterate(e; a : Sequence(\tau) = Sequence\{\} \mid \\ &\quad \textbf{if } e = o \textbf{ then } a \textbf{ else } a \to append(o) \textbf{ endif})\end{aligned} \tag{3.1}$$

To what does τ refer to? Currently, τ is not part of the type environment, because it is neither a classifier nor a state but an instance of *TemplateParameter* in the UML meta-model. This constraint is, therefore, not well-typed. But it is worthwhile to admit constraints like (3.1), because this constraint is valid for any instantiation of the parameter τ.

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 chapter, we only consider parameters ranging over classifiers.

We propose to extend the environment such that Γ contains the kinding judgement $\tau \in \star$ if τ 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 $\star \to \cdots \to \star$, depending on the number of type parameters. Additionally, we give the following type checking rules for templates in Table 3.7. These rules generalises the conforms-to relation previously defined for collection types only.

$$\frac{\Gamma \vdash \tau : K \to K' \quad \Gamma \vdash \tau' : K \to K' \quad \Gamma \vdash \tau'' : K \quad \Gamma \vdash \tau \leq \tau'}{\Gamma \vdash \tau(\tau'') \leq \tau'(\tau'')}\ \text{S-InstSub}$$

$$\frac{\Gamma \vdash \tau : K \to K' \quad \Gamma \vdash \tau' : K \quad \Gamma \vdash \tau'' : K \quad \Gamma \vdash \tau' \leq \tau''}{\Gamma \vdash \tau(\tau') \leq \tau(\tau'')}\ \text{S-InstSub-2}$$

Table 3.7: Subtyping rules for parametric polymorphism

The rule S-InstSub states that if a template class τ is a subtype of another template class τ', then τ remains a subtype of τ' for any class τ'' bound to the parameter. This rule is always adequate. The rule T-InstSub-2 states that for any template class τ and any types τ' and τ'' such that τ' is a subtype of τ'', then binding τ' and τ'' to the type parameter in τ preserves this relation. Both rules generalise S-Coll and S-Coll-2 to all parameterised classes.

Because rule S-InstSub-2 generalises rule S-Coll-2, it is not always safe, for the same reasons that have been described in Section 3.2.

3.3.4 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*.[8] Elements must be of a type supporting the + operation. The + operation must take one parameter of type τ and be both associative: $(a + b) + c = a + (b + c)$, and commutative: $a + b = b + a$. Integer and Real fulfil 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:

$$\begin{aligned}&\textbf{context } \mathit{Sum}\\&\textbf{inv} : \mathit{self}.\mathit{typeOf}().\mathit{allInstances}()\rightarrow\mathit{forAll}(a, b, c \mid\\&\qquad a + (b + c) = (a + b) + c)\\&\textbf{inv} : \mathit{self}.\mathit{typeOf}().\mathit{allInstances}()\rightarrow\mathit{forAll}(a, b \mid a + b = b + a)\end{aligned} \tag{3.2}$$

In Equation (3.2) the property *typeOf*() is supposed to return the run-time type of the object represented by *self*. It is important to observe 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 *Real*s, 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 modeller 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 τ, and we have a collection of instances of τ, then we also know that the property *sum* is defined for this classifier. So *Sum* is a lower bound of the types of τ. Indeed, the signature of *Collection* :: *sum* can be specified by $\mathit{Collection}(T \leq \mathit{Sum}) :: \mathit{sum}() : T$, which expresses the requirements on τ.

Syntactically, we express a bounded template using the notation $C(\tau \leq \zeta)$, defining a type of kind $\Pi\tau \leq \zeta \rightarrow \star$, provided that ζ is of kind $\star$. Here the new kind $\Pi\tau \leq \zeta$ states that τ has to be a subtype of ζ to construct a new type, otherwise, the type is not well-kinded.

Bounded operator abstraction is highly useful in designing object-oriented programs. They enable to express assumptions about the interfaces of types, from which the implementation of the class abstracts using parametric polymorphism, in defining a bound. Many examples of how to design systems using bounded operator abstraction have been described in Bertrand Meyers "Object-Oriented Software Construction" [98]. This form of bounded operator abstraction has been implemented, for example, in the Eiffel programming language [97].

[8] Recall that OCL views the first operand of an operation as the receiver of a message. Here *self* represents the collection, whose elements are to be summed up.

Observe that abstracted operators are not comparable using the subtype relation, that is, we have neither

$$\Lambda T \preceq OclAny : Collection(T) \preceq \Lambda T \preceq OclAny : Set(T)$$

nor

$$\Lambda T \preceq OclAny : Set(T) \preceq \Lambda T \preceq OclAny : Collection(T)$$

Therefore, no additional rules have to be introduced.

3.3.5 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.[9] The provided mechanisms are not sufficient, as the specification of the *flatten()* operation shows (see [113]):

$$\begin{array}{l}\textbf{context } Set :: flatten() : Set(\tau_2)\\ \textbf{post} : result = \textbf{if } self.type.elementType.oclIsKindOf(\text{CollectionType})\\ \quad \textbf{then } self \rightarrow iterate(c; a : Set() = Set\{\} \mid a \rightarrow union(c \rightarrow asSet()))\\ \quad \textbf{else } self\\ \quad \textbf{endif}\end{array} \tag{3.3}$$

This constraint contains many errors. First, the type variable τ_2 is not bound in the model (see Section 3.3.3 for the meaning of binding), so it is ambiguous whether τ_2 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 $self \rightarrow collect(type)$, 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 τ_2 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

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

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:

$$\frac{e : C(\tau) \qquad C \leq \mathit{Collection} \qquad \tau \leq \mathit{Collection}(\tau')}{e \to \mathit{flatten}() : C(\tau')}\ \text{T-FLAT}$$

$$\frac{e : C(\tau) \qquad C \leq \mathit{Collection} \qquad \tau \not\leq \mathit{Collection}(\tau')}{e \to \mathit{flatten}() : C(\tau)}\ \text{T-NFLAT}$$

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 τ'. In this case, τ' 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.

These rules encode the following idea: For each collection type we define an *overloaded* version of flatten. As written in Section 3.3.2, 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 [10].

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

3.4 Adequacy and Decidability

In this section, we summarise the most important results concerning the extended type system. This means that if the type system concludes that an OCL-expression has type τ, then the result of evaluating the expression yields a value of a type that conforms to τ. The type system is adequate and decidable. For the (operational) semantics of OCL we use the one defined in Chapter 2.

Theorem 3.2 (Adequacy)**.** *Let Γ be a context, e an OCL expression, and τ a type of kind $\star$ such that $\Gamma \vdash e : \tau$. Then the value of e is either undefined or it conforms to τ.*

Proof. The proof is by structural induction on the construction of e We only treat some example cases.

1. Assume $e = \ell$ for some literal and $e : \mathit{Real}$. By the literal axiom of Definition 2.19 it follows that $\mathrm{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(\ell) = \ell \in \mathbb{R}$ for all $D, \sigma, \overleftarrow{\sigma}, \Gamma$.

2. Assume $e = \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ endif} : \tau$ and the induction hypothesis is true for e_0, e_1, and e_3. By rule T-COND and the induction hypothesis we have $e_0 : \mathit{Boolean}$ and $\mathrm{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0) \in \mathbb{B}_{\mathcal{J}_{\mathbb{B}}}$.

a) If $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0) = \text{true}$, and, by induction hypothesis $\Gamma \vdash e_1 : T$, then $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e) = \text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_1)$, which is a value that conforms to T.

b) If $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0) = \text{false}$, and, by induction hypothesis $\Gamma \vdash e_2 : T$, then $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e) = \text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_2)$, which is a value that conforms to T.

c) If $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0) = \mathcal{J}_{\mathbb{B}}$, then $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e) = \mathcal{J}_T$.

3. Assume $e = e_0 \to \mathit{iterate}(v; a = e_1 \mid e_2)$, and as an induction hypothesis, that we have $\Gamma \vdash e_0 : \mathit{Collection}(T)$, $v : T$, $a : T'$, $\Gamma \vdash e_1 : T'$ and $\Gamma, v : T, a : T' \vdash e_2 : T'$. Also, as an induction hypothesis, assume, that $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0) \in \mathfrak{D}(\mathit{Collection}(T))$ and $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_1) \in \mathfrak{D}(T')$, and it holds that

$$\lambda x.\lambda y.\, \text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma\{v \mapsto x, a \mapsto y\})(e_2) \in \mathfrak{D}(T \to T' \to T') \quad .$$

Then we distinguish three cases:

a) If $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0)$ is not finite, then $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0) = \mathcal{J}'_T$.

b) If $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0)$ is empty, then

$$\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e) = \text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_1) \quad .$$

c) Else, the induction hypothesis $\lambda x.\lambda y.\, \text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma\{v \mapsto x, a \mapsto y\})(e_2) \in \mathfrak{D}(T \to T' \to T')$ implies

$$(\lambda x.\lambda y.\, \text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma\{v \mapsto x, a \mapsto y\})(e_2))(x)(y) \in \mathfrak{D}(T')$$

for all $x \in \text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0)$ and $y = \text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_1)$.

The proofs for the other cases have a very similar structure. □

The proof presented here is very similar to the one presented Cengarle and Knapp in [23], but Cengarle and Knapp use a slightly different type system and a quite different semantics.

Theorem 3.3 (Decidability of Type Checking)**.** *For any context* Γ *and any OCL expression* e *and type* τ *of kind* $\star$*, it is decidable whether* $\Gamma \vdash e : \tau$.

The proof of this theorem is a consequence of the theorems by Compagnoni [30] and Steffen [145]. The key components of the proof are: Infer a minimal type T' for e in Γ using a decidable procedure, and check whether $\Gamma \vdash T' \leq T$, which has to be decidable.

We have used the type inference algorithm of Definition 4.81 in [30], which has been proved decidable in this paper. For the case of checking subtyping, we point out

that OCL and UML only allow types of kind $\star$ to be compared. This reduces our type system to a special case of [30].

Constraints for methods defined in parameterised classes are handled by instantiating an arbitrary type T for the type variable τ and by adding the axiom T is a subtype of the lower bound defined in the abstraction.

We do not give the proof of Theorem 3.3 here, because our proof does not provide any new insight over the proof given by Compagnoni, and her proof is about 50 pages.

Our algorithm is an adaption of [30] and [145] to handle union types. It is simpler, because we only have a form of bounded operator abstraction, where type abstractions are not comparable, which simplifies the subtyping problem, and polymorphism is ML-like.

Another result is concerning incompleteness of the type checking procedure. 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. The 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 chapter addresses many features of UML and OCL: multiple inheritance, operator overloading, parameterised classifiers, and bounded operator abstraction. And the type-checking problem is still decidable for this type system. If we, for example, also add checking for value specifications of method specifications of parameterised classes to the type system, the type theory would correspond to the calculus of constructions, which is undecidable, as described by Coquand [35]. Such type systems indeed form the theoretical foundation of interactive theorem provers.

3.5 Related Work and Conclusions

A type system for OCL has been presented by Clark in [25], by Richter and Gogolla in [135], and by Cengarle and Knapp in [23]. In Section 3.2 we summarised these results and give a formal basis for our proposal.

A. Schürr has described an extension to the type system of OCL [141], 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 parameterised types and bounded polymorphism, because the normal forms of types required for the proof of Theorem 3.3 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 analysed in [30], 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 infor-

mation [145]. 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, for example, 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 modelling. 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.