Disambiguating Implicit Constructions in OCL

Kristofer Johannisson

Department of Computing Science
Chalmers University of Technology and Göteborg University
S-41296 Göteborg, Sweden,
krijo@cs.chalmers.se

Abstract. A rule system for type checking and semantic annotation of OCL is presented. Its main feature is the semantic annotation and disambiguation of syntax trees provided by an OCL parser, in particular for implicit property calls and implicit bound variables. It is intended as a component to be plugged in to other systems which handle OCL. An implementation of the system is available.

1 Introduction

A suitable structure for a computer program which handles the Object Constraint Language (OCL [7, 8]) is that of a compiler (cf. [10]) with at least three components:

1. a component which parses OCL specifications into syntax trees.
2. a component which performs some kind of semantic analysis of OCL syntax trees, e.g. type checking.
3. a component which does something interesting with the result of the semantical analysis, e.g. transforms it into a proof obligation in Dynamic Logic to be fed into a theorem prover [1], or generates assertions to be inserted into Java source code [10].

In this paper we focus on part 2: we present a rule system for type checking and semantic annotation of OCL. The input to the system is a syntax tree from an OCL parser, and a representation of the user UML model. The result is a syntax tree annotated with type information and other semantic distinctions which disambiguates the output of the parser. This makes the job of part 3 easier than if it would have to work directly with the parser output.

The disambiguation concerns in particular the "property call" syntactic structure, the semantics of which has many special cases: calls to properties defined in the user UML model or the OCL library, variable binding constructions (e.g. `forAll` or `collect`), and meta-level constructions (e.g. `allInstances`). With respect to property calls one must also consider various implicit or special forms: `self` can be left out, variable bindings can be left out, `collect` can be left out, and the return type of associations of multiplicity one (as opposed to other multiplicities) can be considered to be a collection type or a basic type. All these

cases share the same basic syntactic structure, which motivates a disambiguating step between part 1 and part 3.

Our system was originally developed as a part of a project for linking OCL specifications to informal specifications in natural language [9], which is in turn a part of the KeY project [1]. In this context, part 3 would be the translation to natural language. However, we here present our work as a standalone system.

1.1 Paper Outline

Some background, including related work, is given in Section 2. Then we define the language of annotated OCL in Section 3. Section 4 describes the annotating rule system itself. This is followed by brief discussion of the implementation in Section 5. Section 6 concludes and gives directions for future work. We assume that the reader is reasonably familiar with the OCL specification, version 1.x or 2.0 [7, 8].

2 Background

2.1 Implicit Features of OCL

In this paper we will use the term "implicit" for various features of OCL. We will make this use precise by the definition of annotated OCL and the annotating rules in the sections following, but we give some informal examples below to get started:

Implicit self refers to property calls to `self` where `self` has been left out ([7] Sect. 6.3.3). E.g. `attr` is an implicit form of `self.attr`.

Implicit bound variables refers to the use of variable binding collection operations of the OCL library where one does not explicitly bind variables ([7] Sect. 6.6.1). E.g. `collection->collect(attr)` is an implicit form of `collection->collect(x|x.attr)`.

Implicit property calls refers to property calls where either `self` or an implicit bound variable is "left out" ([7] Sect. 6.6.7). E.g. `collection->collect(attr)` is an implicit form of either `collection->collect(self.attr)` or `collection->collect(x|x.attr)` (and is, as noted in [7], potentially semantically ambiguous).

Implicit collect means that `collect` has been left out ([7] Sect. 6.6.2.1). E.g. `collection.attr` is an implicit form of `collection->collect(x|x.attr)`.

2.2 Related Work

Semantic analysis of OCL is of course in some sense performed in any OCL tool. Our rule system is in particular inspired by [4] and [3], which define OCL type derivation systems and operational semantics, and also prove various results such as subject reduction and type uniqueness. Of the two, the more recent [3]

handles more OCL constructions, and it also gives comparisons to many other systems.

In comparison to [3] we do not provide an operational semantics or proofs about our system. Our perspective is more that of giving a formal description of a concrete implementation which has to handle "real" OCL files. This means that we in our system must include constructions such as implicit `self`, implicit bound variables and collection operations which can be expressed using `iterate` — which are not included in [3]. Also, in contrast to our system, [3] does not employ any disambiguating annotations besides types. For instance, it handles implicit `collect` by normalization, which means an annotated implicit `collect` expression cannot be easily distinguished from an annotated expression using explicit `collect`.

2.3 From OCL to Natural Language

As a part of ongoing work of linking formal specifications to informal ones [9], we wish to translate OCL into Natural Language (NL), e.g. English or German.

The system in [9] is based on grammars in the Grammatical Framework (GF) formalism [12]. These grammars give an abstract syntax of specifications, which can be presented in NL (English and German) as well as OCL.

GF grammars are written in the same style as programs in typed functional languages. In our GF grammars, there are types and functions corresponding to the classes and properties from the OCL library and user UML model. The grammars have to be dynamically extended with new types and functions for each new user UML model. Since there is no subtyping in GF, we use explicit coercions (typecasts). Variable binding constructions are modelled as higher order functions.

In other words, the GF syntax trees provide a typed and semantic representation of OCL specifications. Now, if we want to transform the syntax trees provided by a standard OCL parser into GF trees, it makes sense to do this in a modular way as described in the introduction: First we add types and disambiguating annotations to the syntax trees from the parser. Then it will be fairly straightforward to transform annotated syntax trees into GF syntax trees.

Although the system we present in this paper was originally a module in the OCL to NL translation, we think that it is general enough to be useful for other purposes, and therefore to be presented as something of its own.

3 Annotated OCL

We will give a grammar for *unannotated OCL*, based on OCL version 1.5 [7], and then define *annotated OCL* by extending this grammar — adding new categories, and new rules to existing categories.

There are a number of OCL 1.5 features that we do not support: replacing `self` with a named variable, named constraints, the `def` stereotype (i.e. "global let-definitions"), the types `OclExpression` and `OclState`, qualified associations,

associations classes, enumerations, and implicit flattening of collections. The main reason for these limitations, and for not supporting the new features in OCL 2.0, is practical: our system is intended as a part of the KeY [1] system, which has no yet moved beyond OCL 1.5.

In comparison to [3], the most important OCL features we add are implicit property calls and implicit bound variables. We also handle packages as well as pre- and postconditions, but we will not discuss them in this paper. On the other hand, [3] includes features we do not support, e.g. OCL 2.0 tuples and undefined values.

3.1 Unannotated OCL

The grammar for unannotated OCL, which describes the syntax trees provided by the OCL parser (for the sake of presentation the grammar used in the actual implemented parser has been somewhat simplified here).

```
OclPackage     ::= package PathName Constraint {Constraint} endpackage
PathName       ::= Ident{:: Ident}
Constraint     ::= context Context ConstrBody {ConstrBody}
Context        ::= Ident | Ident :: Ident ([FormalParam {,FormalParam}])
                   [: Class]
FormalParam    ::= Ident : Class
Class          ::= PathName | CollKind(Class)
CollKind       ::= Collection | Set | Bag | Sequence
ConstrBody     ::= (inv|pre|post): [LetExp {LetExp} in] Expr
LetExp         ::= let Ident [([FormalParam {,FormalParam}])] [: Class]
                    = Expr
Expr           ::= Expr InfixOp Expr |
                   PrefixOp Expr |
                   Literal |
                   if Expr then Expr else Expr endif |
                   PropCall |
                   Expr ApplOper PropCall
Literal        :::  IntLit | RealLit | StringLit | true | false |
                   CollKind {[CollItem{,CollItem}]}
CollItem       ::= Expr | Expr .. Expr
InfixOp        ::= + | - | / | * | = | <> | < | > | <= | >=
PrefixOp       ::= - | not
ApplOper       ::= . | ->
PropCall       ::= PathName[@pre][PropCallParams]
PropCallParams ::= ([Declarator |][Expr{,Expr}])
Declarator     ::= Ident{,Ident}[:PathName][;Ident:Class=Expr]
```

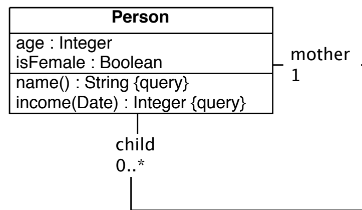
We leave the categories *IntLit*, *RealLit* and *StringLit* abstract. We will focus our disambiguation efforts on the implicit property call rule *Expr* ::= *PropCall*, and on the *PropCall* and *PropCallParams* categories.

3.2 Types

We will use the *Class* category to annotate expressions with their types. We also want to annotate properties (or property calls) with their types, and we therefore add a category *PropType*:

$$PropType ::= T \xrightarrow{A} T \mid T \xrightarrow{Q} \{T \rightarrow\}\ T \mid T \xrightarrow{St.} T$$

A property can be either an attribute (or association), a query, or a singleton (of multiplicity one) association . Consider the properties in this example class diagram:



These properties would be typed as follows:

$$\begin{array}{ll}
\texttt{age} & : \texttt{Person} \xrightarrow{A} \texttt{Integer} \\
\texttt{isFemale} & : \texttt{Person} \xrightarrow{A} \texttt{Boolean} \\
\texttt{name} & : \texttt{Person} \xrightarrow{Q} \texttt{String} \\
\texttt{income} & : \texttt{Person} \xrightarrow{Q} \texttt{Date} \longrightarrow \texttt{Integer} \\
\texttt{child} & : \texttt{Person} \xrightarrow{A} \texttt{Set(Person)} \\
\texttt{mother} & : \texttt{Person} \xrightarrow{St.} \texttt{Person}
\end{array}$$

3.3 Annotating Expressions

Aside from annotating an expression with its type, we also want to disambiguate the rule *Expr* ::= *PropCall* in the unannotated grammar. This syntactic structure could be a variable, a class literal, or an implicit property call to `self` or an implicit bound variable. We also introduce annotation for a singleton association considered as a set.

$$\begin{array}{rl}
Expr ::= & Expr\ :\ Class \mid \\
& \mathbf{Var}(Ident) \mid \\
& \mathbf{ClassLit}(Class) \mid \\
& \mathbf{implicit}((\texttt{self} \mid \underline{Ident}), PropCall) \mid \\
& \{Expr\}_{\mathbf{1}}
\end{array}$$

We use a **bold font** to distinguish semantic annotations from unannotated OCL. We use underline for indicating implicit bound variables.

We will also annotate expressions with explicit coercions (typecasts) by the following rule:

$$Expr ::= [Expr]^{Class}_{Class}$$

If expression e has type T_1, and we know that T_1 is a subtype of T_2, then $[e]^{T_1}_{T_2}$ represent the expression e coerced (upcasted) into type T_2. We will slightly abuse notation by considering $[e]^{T_1}_{T_2}$ to be the same as just e when T_1 is the same class as T_2.

3.4 Annotating Property Calls

Property calls will be annotated with their types, i.e. with a *PropType*. We also make some disambiguation: a property call can be a "normal" property call (in which case we use no annotation except the *PropType*), a variable binding operation (e.g. `iterate` or `select`) or an implicit collect:

$$\begin{aligned} PropCall ::= \; & PropCall \;: PropType \mid \\ & (\textbf{iterate} \mid \textbf{forAll} \mid \textbf{exists} \mid \textbf{one} \mid \textbf{isUnique} \mid \textbf{any} \mid \textbf{sortedBy} \mid \\ & \textbf{select} \mid \textbf{reject} \mid \textbf{collect}) \; [\texttt{@pre}] \; PropCallParams \mid \\ & \textbf{implCollect}(PropCall) \end{aligned}$$

In the unannotated grammar, there is no reserved word for the OCL library `iterate` construction. The identifier `iterate` could refer either to the OCL library construction, or to a user-defined property. In the annotated grammar, **iterate** will refer to the OCL library construction, and `iterate` to a user-defined property (if there is such a property in the UML model).

We also extend *Declarator* to enable implicit binding of variables (again using underlining to distinguish explicit from implicit binding):

$$Declarator ::= \underline{Ident \;: Class}$$

3.5 Annotated Example

This unannotated OCL constraint says that the age of a person must be non-negative:

```
context Person inv: age >= 0
```

Using the rule system in Sect. 4, we add type annotations on the property `age` and the integer literal `0`. Also, we disambiguate the property call `age` by an implicit `self` annotation. Finally, since `age` and `0` both have type `Integer`, but the comparison operator `>=` works on the supertype `Real`, we add explicit coercions from `Integer` to `Real`:

$$\begin{aligned} & \texttt{context Person inv:} \\ & [\textbf{implicit}(\texttt{self} \;: \texttt{Person}, \texttt{age} \;: \texttt{Person} \xrightarrow{A} \texttt{Integer}) \;: \texttt{Integer}]^{\texttt{Integer}}_{\texttt{Real}} \\ & \;\texttt{>=} \\ & [\texttt{0} \;: \texttt{Integer}]^{\texttt{Integer}}_{\texttt{Real}} \end{aligned}$$

Since we add annotation in almost every node of the syntax tree, even this small example becomes unwieldy when annotated. Of course, the annotated syntax is intended as input to programs, not as a convenient notation to be read or written manually.

3.6 Semantics of Annotated OCL

While we have not defined a formal semantics for our annotated OCL language, we hope that the informal semantics is clear (as far as the semantics in [7, 8] is clear). However, one way of achieving a formal semantics would be to drop the annotations in a systematic way, with the goal of ending up in the unannotated OCL fragment of [3], which has an operational semantics. From this perspective, we have then given a semantics to the OCL fragment of [3] extended with implicit property calls and implicit bound variables.

An informal outline of how to de-annotate annotated OCL is the following:

- the type annotations of *Expr* with *Class* and *PropCall* with *PropType* are simply dropped, and so are explicit coercions
- the annotations **implCollect** and $\{\dots\}_{\mathbf{1}}$ are also just dropped, since implicit `collect` and singleton associations are handled by [3]
- **Var** and **ClassLit** expressions should be transformed into their counterpart in [3]
- The **implicit** annotation has to be normalized. E.g. **implicit**(`self`, `age`) should be changed into `self.age`, and **implicit**($\underline{\texttt{x}}$, `age`) into $\underline{\texttt{x}}$`.age`.
- **iterate** is just changed into `iterate`. The other variable-binding collection operations (**forAll**, **exists**, . . .) have to be rewritten in terms of `iterate`, according to the definitions in [7, 8].
- Implicit bound variables and declarators are made into normal variables and declarators (i.e. just remove the underlining). Before doing this the implicit bound variables should be renamed to avoid name clashes.

Essentially we remove all annotations except the ones for implicit property calls and implicit bound variables — which are not handled by [3]. These are instead replaced with the corresponding "explicit" constructions.

4 Rule System

The task of the rule system is to take a syntax tree produced by an OCL parser, typecheck it, and annotate it with semantic information. To do this, information about the user UML model is also required. Inspired by [3], the system will use judgements of the form

$$E \vdash t \triangleright t'$$

where E is an environment, t is a syntax tree, and t' is an typechecked, annotated syntax tree. Note that the rule system *annotates* implicit constructions, it does not replace them with explicit ones, as is done with e.g. implicit `collect` in [3].

We do not give a complete and fully formal description of our system — we focus mostly on the the disambiguation of constructions involving *PropCall*.

In the rules we will use variables ranging as follows: $x, y, i \in \mathit{Ident}, e \in \mathit{Expr}, T \in \mathit{Class}, PT \in \mathit{PropType}, p \in \mathit{PathName}, C \in \mathit{CollKind}$

4.1 Environments

An environment may contain a signature and a context. Depending on the syntactic category we are annotating, there may also be other components. A signature Σ contains class names, operations, properties, and a subtyping relation, i.e. information from the user UML model and the definition of the OCL library classes. A context Γ contains typing of variables and properties defined in let-definitions, and the (name of) the current package. We omit the many details of how theories and contexts are represented, but use the constructions below to access and update their respective components:

$T_1 <:_\Sigma T_2$	type T_1 conforms to T_2 in signature Σ
$\sqcup_\Sigma \{T_1, \ldots, T_n\}$	the least common supertype of types $T_1, \ldots, T_n$ in Σ
$T \in \text{classes}_\Sigma$	T is a class in Σ
$(x\ : T) \in \Gamma$	x has type T in context Γ
$\Gamma, (x\ : T)$	update type of x in Γ to T
package_Γ	current package in Γ

Since a context Γ contains the current package, we always assume that an unqualified, non-OCL-library class T belongs to the current package in a lookup $(x\ : T) \in \Gamma$ or update $\Gamma, (x\ : T)$.

To find out what property an identifier refers to in a given environment, we use the following partial functions ([*Class*] is the type of lists of *Class*):

$$\begin{aligned} \text{lookupAttr}_{\Sigma,\Gamma} &: \mathit{Ident} \to \mathit{Class} \to \mathit{PropType} \\ \text{lookupProp}_{\Sigma,\Gamma} &: \mathit{Ident} \to \mathit{Class} \to [\mathit{Class}] \to \mathit{PropType} \end{aligned}$$

Again, we omit all details on how these functions are defined, but note that they are partial: there might not be a matching property for a given identifier, or the result might be ambiguous (e.g. in case of multiple inheritance).

We define two functions recType and retType on *PropType*:

$$\begin{aligned} \text{recType}\ (T_1\ (\stackrel{A}{\to} \mid \stackrel{Q}{\to} \mid \stackrel{St.}{\to}) \ldots T_2) &= T_1 \\ \text{retType}\ (T_1\ (\stackrel{A}{\to} \mid \stackrel{Q}{\to} \mid \stackrel{St.}{\to}) \ldots T_2) &= T_2 \end{aligned}$$

4.2 Property Call Parameters

We go bottom-up and start with the category *PropCallParams*, where we handle variable bindings. The environment $\Sigma; \Gamma; T; B$ consists of signature and context, a type T, and a boolean B. T is used for typing bound variables (if there are any), if B is true then we insert an implicit bound variable in case there are no explicit ones.

If there are no bound variables, and B is false, then we just annotate the parameters:

$$\frac{\Sigma; \Gamma \vdash e_1 \triangleright e_1' \ : T_1 \ \cdots \ \Sigma; \Gamma \vdash e_1 \triangleright e_n' \ : T_n}{\Sigma; \Gamma; T; \text{False} \vdash (e_1, \ldots, e_n) \triangleright (e_1' \ : T_1, \ldots, e_n' \ : T_n)}$$

When there is a declarator (or an implicit bound variable), there must be exactly one parameter. There may or may not be typings on the bound variables.

$$\frac{\Sigma; \Gamma, (x_1 \ : T), \ldots, (x_n \ : T) \vdash e \triangleright e' \ : T_1}{\Sigma; \Gamma; C(T); B \vdash (x_1, \ldots, x_n \,|\, e) \triangleright (x_1, \ldots, x_n \,|\, e' \ : T_1)}$$

$$\frac{\begin{array}{c} T <:_\Sigma T' \\ \Sigma; \Gamma, (x_1 \ : T'), \ldots, (x_n \ : T') \vdash e \triangleright e' \ : T_1 \end{array}}{\Sigma; \Gamma; C(T); B \vdash (x_1, \ldots, x_n \ : T' \,|\, e) \triangleright (x_1, \ldots, x_n \,|\, e' \ : T_1)}$$

$$\frac{\Sigma; \Gamma, (\underline{x} \ : T) \vdash e \triangleright e' \ : T_1}{\Sigma; \Gamma; C(T); \text{True} \vdash (e) \triangleright (\underline{x \ : T \,|}\ e' \ : T_1)}$$

Side condition: $\underline{x}$ is fresh.

If there is an accumulator, there must be exactly one bound variable (because accumulators are only used with `iterate`):

$$\frac{\begin{array}{c} \Sigma; \Gamma \vdash e_y \triangleright e_y' \ : T_2 \\ \Sigma \vdash T_2 <:_\Sigma T_y \\ \Sigma; \Gamma, (x \ : T), (y \ : T_y) \vdash e \triangleright e' \ : T_1 \end{array}}{\Sigma; \Gamma; C(T); B \vdash (x; y \ : T_y = e_y \,|\, e) \triangleright (x; y \ : T_y = [e_y' \ : T_2]^{T_2}_{T_y} \,|\, e' \ : T_1)}$$

$$\frac{\begin{array}{c} T <:_\Sigma T' \\ \Sigma; \Gamma \vdash e_y \triangleright e_y' \ : T_2 \\ \Sigma \vdash T_2 <:_\Sigma T_y \\ \Sigma; \Gamma, (x \ : T'), (y \ : T_y) \vdash e \triangleright e' \ : T_1 \end{array}}{\Sigma; \Gamma; C(T); B \vdash (x \ : T'; y \ : T_y = e_y \,|\, e) \triangleright (x \ : T'; y \ : T_y = [e_y' \ : T_2]^{T_2}_{T_y} \,|\, e' \ : T_1)}$$

4.3 Property Calls

A property call (*PropCall*) *pc* is annotated in an environment $\Sigma; \Gamma; T; e; (\texttt{.}|\texttt{->})$, meaning that it occurred in an expression context $e(\texttt{.}|\texttt{->})pc$ where $e \ : T$. According to the grammar, every property has an optional `@pre`. This does not affect the typing or annotation, however, so in this section we give all rules without `@pre`.

We distinguish between normal property calls (attributes, associations and properties from the user UML model or the OCL library), variable binding

constructions (e.g. `iterate`), implicit `collect`, and meta-level operations (e.g. `allInstances`). All property calls are annotated with their *PropType*, variable binding constructions and implicit `collect` are also annotated as special constructions. The variable binding constructions could be seen as higher order functions, but in the *PropType* annotations we ignore this. We have chosen not to give the meta-level operations special annotation, but they do require special rules for type checking.

In variable binding constructions, there may or may not be variables in the *Declarator*, and the variables may or may not have typings. These cases are handled in the annotation of *PropCallParams* above, and make no difference in the *PropCall* rules of this section. Therefore, we give the rules only for when there are variables explicitly given, and the variables have typings.

Attributes/associations

$$\frac{\text{lookup}_{\Sigma;\Gamma}(a,T) = T_1 \xrightarrow{A} T_2}{\Sigma;\Gamma;T;e;\,.\vdash a \triangleright a\ \ : T_1 \xrightarrow{A} T_2}$$

For singleton associations we have the same rule except that $\xrightarrow{A}$ is replaced with $\xrightarrow{St.}$.

Queries

$$\frac{\begin{array}{c}\Sigma;\Gamma;T;\text{False} \vdash \texttt{(}e_1,\ldots,e_n\texttt{)} \triangleright \texttt{(}e'_1\ \ : T_1,\ldots,e'_n\ \ : T_n\texttt{)}\\ \text{lookup}_{\Sigma;\Gamma}(q,T,[T_1,\ldots,T_n]) = T' \xrightarrow{Q} T'_1 \to \cdots \to T'_{n+1}\end{array}}{\begin{array}{c}\Sigma;\Gamma;T;e;ao \vdash q\texttt{(}e_1,\ldots,e_n\texttt{)} \triangleright\\ q\texttt{(}[e'_1\ \ : T_1]^{T_1}_{T'_1},\ldots,[e'_n\ \ : T_n]^{T_n}_{T'_n}\texttt{)}\ \ : T' \xrightarrow{Q} T'_1 \to \cdots \to T'_{n+1}\end{array}}$$

Note that n might be 0 here.

Iterate

$$\frac{\Sigma;\Gamma;\texttt{Collection(}T\texttt{)};\text{False} \vdash \texttt{(}x\texttt{:}T_1\texttt{;}acc\texttt{:}T_2\texttt{=}e_2\texttt{|}e_3\texttt{)} \triangleright \texttt{(}x\texttt{:}T_1\texttt{;}acc\texttt{:}T_2\texttt{=}e'_2\texttt{|}e'_3\texttt{)}}{\begin{array}{c}\Sigma;\Gamma;\texttt{Collection(}T\texttt{)};e_1;\texttt{->} \vdash \texttt{iterate(}x\texttt{:}T_1\texttt{;}acc\texttt{:}T_2\texttt{=}e_2\texttt{|}e_3\texttt{)} \triangleright\\ \textbf{iterate}\texttt{(}x\texttt{:}T_1\texttt{;}acc\texttt{:}T_2\texttt{=}e'_2\texttt{|}e'_3\texttt{)}\ \ : \texttt{Collection(}T\texttt{)} \xrightarrow{Q} T_2 \to T_2\end{array}}$$

Other variable binding collection operations The rules for `forAll`, `exists`, `one`, `isUnique`, `any`, `sortedBy`, `select`, `reject` and `collect` are quite similar. We here give the ones for `forAll` and `collect` to exemplify.

In the informal descriptions in [7] (Sect. 6.6.3) and [8] (Sect. 7.6.3) it is explicitly said that for `forAll`, binding of several variables at once in the declarator is

allowed, and there is no mention that other operations would allow it. This is apparently contradicted by [8] (Sect. 11.9) which lists certain constructions as only allowing one bound variable (the others would then implicitly allow several).

$$\frac{\Sigma;\Gamma;\texttt{Collection}(T);\text{True}\vdash \texttt{(}x_1,\ldots,x_n\ :T_1\,\texttt{|}\,e_2\texttt{)}\rhd\texttt{(}x_1,\ldots,x_n\ :T_1\,\texttt{|}\,e_2'\ :T_2\texttt{)}\quad \Sigma\vdash T_2 <:_\Sigma \texttt{Boolean}}{\begin{array}{c}\Sigma;\Gamma;\texttt{Collection}(T);e_1;\texttt{->}\vdash \texttt{forAll(}x_1,\ldots,x_n\ :T_1\,\texttt{|}\,e_2\texttt{)}\rhd\\ \textbf{forAll}\texttt{(}x_1,\ldots,x_n\ :T_1\,\texttt{|}\,[e_2'\ :T_2]^{T_2}_{\texttt{Boolean}}\texttt{)}\ :\\ \texttt{Collection}(T)\xrightarrow{Q}\texttt{Boolean}\to\texttt{Boolean}\end{array}}$$

The `collect` construction is defined for `Set`, `Bag`, and `Sequence`, but not for `Collection`, so in this rule $C\in\{\texttt{Set},\texttt{Bag},\texttt{Sequence}\}$. If C is `Sequence`, D is `Sequence`, otherwise D is `Bag`.

$$\frac{\Sigma;\Gamma;C(T);\text{True}\vdash \texttt{(}x\ :T_1\,\texttt{|}\,e_2\texttt{)}\rhd\texttt{(}x\ :T_1\,\texttt{|}\,e_2'\ :T_2\texttt{)}}{\begin{array}{c}\Sigma;\Gamma;C(T);e_1;\texttt{->}\vdash \texttt{collect(}x\ :T_1\,\texttt{|}\,e_2\texttt{)}\rhd\\ \textbf{collect}\texttt{(}x\ :T_1\,\texttt{|}\,e_2'\ :T_2\texttt{)}\ :C(T)\xrightarrow{Q}T_2\to D(T_2)\end{array}}$$

Implicit collect An expression of collection type followed by a dot and a property call might be an implicit `collect` property call. As with `collect`, $C\in\{\texttt{Set},\texttt{Bag},\texttt{Sequence}\}$:

$$\frac{\Sigma;\Gamma;C(T);e_1;\texttt{->}\vdash \texttt{collect(}pc\texttt{)}\rhd\textbf{collect}\texttt{(}\underline{i}\ :T\,\texttt{|}\,\underline{i}\texttt{.}pc'\texttt{)}\ :T_2}{\Sigma;\Gamma;C(T);e_1;\texttt{.}\vdash pc\rhd\textbf{implCollect}(pc')\ :T_2}$$

Meta-level operations The meta-level operations involve `OclType` and class literals.

$$\frac{\Sigma;\Gamma;\texttt{OclAny};\text{False}\vdash\texttt{(}e_1\texttt{)}\rhd\texttt{(}\textbf{ClassLit}(T)\ :\texttt{OclType}\texttt{)}}{\begin{array}{c}\Sigma;\Gamma;\texttt{OclAny};e;\texttt{.}\vdash\texttt{oclAsType(}e_1\texttt{)}\rhd\\ \texttt{oclAsType(}\textbf{ClassLit}(T)\ :\texttt{OclType}\texttt{)}\ :\texttt{OclAny}\xrightarrow{Q}\texttt{OclType}\to T\end{array}}$$

$$\frac{}{\begin{array}{c}\Sigma;\Gamma;\texttt{OclType};\textbf{ClassLit}(T);\texttt{.}\vdash\texttt{allInstances()}\rhd\\ \texttt{allInstances()}\ :\texttt{OclType}\xrightarrow{Q}\texttt{Set}(T)\end{array}}$$

$$\frac{}{\begin{array}{c}\Sigma;\Gamma;\texttt{OclType};e;\texttt{.}\vdash\texttt{allSupertypes()}\rhd\\ \texttt{allSupertypes()}\ :\texttt{OclType}\xrightarrow{Q}\texttt{Set(OclType)}\end{array}}$$

4.4 Expressions: Implicit property calls

The rule *Expr* ::= *PropCall* has to be disambiguated. It represents either a variable, a class literal or an implicit property call.

Variables Mark variables as being variables.

$$\frac{x \;: T \in \Gamma}{\Sigma; \Gamma \vdash x \triangleright \mathbf{Var}(x) \;: T}$$

Class literals Mark class literals as being class literals.

$$\frac{\begin{array}{c}\text{package}_\Gamma = p \\ p\texttt{::}i \in \text{classes}_\Sigma\end{array}}{\Sigma; \Gamma \vdash i \triangleright \mathbf{ClassLit}(i) \;: \texttt{OclType}}$$

$$\frac{i_1\texttt{::}\ldots\texttt{::}i_n \in \text{classes}_\Sigma}{\Sigma; \Gamma \vdash i_1\texttt{::}\ldots\texttt{::}i_n \triangleright \mathbf{ClassLit}(i_1\texttt{::}\ldots\texttt{::}i_n) \;: \texttt{OclType}}$$

Implicit property call to `self` or bound variables Here we try putting a `self` or an implicit bound variable in front of the property call. Implicit property calls can be ambiguous if `self` and an implicit bound variable (or two implicit bound variables) conform to the same type, in which case typechecking should fail. This is expressed as side-conditions on the two rules.

$$\frac{\Sigma; \Gamma \vdash \texttt{self}\,\texttt{.}\,pc \triangleright e\,\texttt{.}\,pc' \;: T}{\Sigma; \Gamma \vdash pc \triangleright \mathbf{implicit}(\texttt{self}, pc') \;: T}$$

Side condition: There are no $\underline{x}, pc', T$ such that $\Sigma; \Gamma \vdash pc \triangleright \mathbf{implicit}(\underline{x}, pc') \;: T$.

We assume here that implicit bound variables have non-collection types, hence we can safely use a dot (and not an arrow) when annotating the the property call.

$$\frac{\begin{array}{c}\underline{x} \;: T_1 \in \Gamma \\ \Sigma; \Gamma \vdash \underline{x}\,\texttt{.}\,pc \triangleright e\,\texttt{.}\,pc' \;: T_2\end{array}}{\Sigma; \Gamma \vdash pc \triangleright \mathbf{implicit}(\underline{x}, pc') \;: T_2}$$

Side conditions: There are no pc', T such that $\Sigma; \Gamma \vdash pc \triangleright \mathbf{implicit}(\texttt{self}, pc') \;: T$, and there is exactly one $\underline{x} \in \Gamma$ such that there exists e, pc', T such that $\Sigma; \Gamma \vdash \underline{x}\,\texttt{.}\,pc \triangleright e\,\texttt{.}\,pc' \;: T$

4.5 Expressions: Explicit property calls

We did annotation of the category *PropCall* separately in in Sect. 4.3. This means that we annotate an explicit property call expression $e(\texttt{.}|\texttt{->})pc$ by annotating the property call pc, and then explicitly coercing e into the correct type. Note that when annotating pc in the context $e(\texttt{.}|\texttt{->})pc$, the environment is extended with the type of e, e itself, and also the `.` or `->`. Here $ao \in \{\texttt{.}, \texttt{->}\}$.

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash e \triangleright e' \;: T \\ \Sigma; \Gamma; T, e', ao \vdash pc \triangleright pc' : PT\end{array}}{\Sigma; \Gamma \vdash e\;ao\;pc \triangleright [e' \;: T]^T_{\text{recType}(PT)}\;ao\;pc' \;: \text{retType}(PT)}$$

Singleton associations The only special case is e`->`pc when e is a singleton association property call, in which case we need to treat e as a singleton set ($a, a' \in \mathit{PropCall}$):

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash e \triangleright e' \ : T \\ e' = e''.a \text{ or } e' = \mathbf{implicit}(e'', a) \\ a = a' \ : T_1 \overset{St.}{\to} T_2 \\ \Sigma; \Gamma; \texttt{Set(}T\texttt{)}; \{e'\}_1, \texttt{->} \vdash pc \triangleright pc' \ : PT\end{array}}{\Sigma; \Gamma \vdash e \texttt{ -> } pc \triangleright [\{e'\}_1 \ : \texttt{Set(}T\texttt{)}]^{\texttt{Set(}T\texttt{)}}_{\mathrm{recType}(PT)} \texttt{ -> } pc' \ : \mathrm{retType}(PT)}$$

In contrast to [3], we do not allow any non-collection expression to be considered as a set, but only direct property calls to singleton associations.

4.6 Expressions: other cases

The rules for infix, prefix, literal and if-then-else expression should be unsurprising. We give only the rule for if-then-else here, as an example.

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash e_1 \triangleright e_1' \ : T_1 \\ \Sigma \vdash T_1 <:_\Sigma \texttt{Boolean} \\ \Sigma; \Gamma \vdash e_2 \triangleright e_2' \ : T_2 \\ \Sigma; \Gamma \vdash e_3 \triangleright e_3' \ : T_3 \\ \sqcup_\Sigma \{T_2, T_3\} = T_4\end{array}}{\begin{array}{c}\Sigma; \Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \texttt{ endif} \triangleright \\ \texttt{if } [e_1' \ : T_1]^{T_1}_{\texttt{Boolean}} \texttt{ then } [e_2' \ : T_2]^{T_2}_{T_4} \texttt{ else } [e_3' \ : T_3]^{T_3}_{T_4} \texttt{ endif} \ : T_4\end{array}}$$

4.7 Other Categories

We do not present the rules for the categories *LetExp*, *ConstrBody*, *Constraint* or *OclPackage*. These rules mainly deal with adding let-definitions, `self`, `result`, formal parameters and the name of the current package to the context.

4.8 KeY extensions

Our system is developed as part of the KeY system [1], in which OCL is used for specifications of JavaCard programs. To handle the JavaCard concepts of null values and exceptions, the constructions `null` and `excThrown` are used in KeY OCL specifications.

We add a class `Null` to the OCL library package, an expression `null` of type `Null`, and let `Null` be a subtype of all other types. We consider `null` as a reserved word in both unannotated and annotated OCL.

$$\frac{}{\Sigma; \Gamma \vdash \texttt{null} \triangleright \texttt{null} \ : \texttt{Null}}$$

`Null` and `null` seem to work just as `Void` and `undef` in [3].

We handle `excThrown` as a new property of the OCL library class `OclAny`, taking an argument of type `OclType`, and returning `Boolean`. Assuming a user UML model containing JavaCard packages, `excThrown(java::lang::Exception)` can be used as a predicate stating that an exception of type `java::lang::Exception` has been thrown. This is an implicit `self` property call, and since `self` always conforms to `OclAny` it can be used in any context. We use the following rule for the property call `excThrown`:

$$\frac{\begin{array}{c}\Sigma;\Gamma \vdash e_1 \triangleright \mathbf{ClassLit}(T_1) \ : \texttt{OclType} \\ \Sigma \vdash T_1 <:_{\Sigma} \texttt{java::lang::Exception}\end{array}}{\begin{array}{c}\Sigma;\Gamma;T;e;\,.\, \vdash \texttt{excThrown}(e_1) \triangleright \\ \texttt{excThrown}(\mathbf{ClassLit}(T_1) \ : \texttt{OclType}) \ : \texttt{OclAny} \xrightarrow{Q} \texttt{OclType} \rightarrow \texttt{Boolean}\end{array}}$$

5 Implementation

The system for annotating OCL syntax trees is implemented in the functional language Haskell, hence it can be used as a component of Haskell programs. We also provide a command line interface taking as input two text files: one containing OCL and another containing the UML model (for which we have defined a simple custom format). The output is a textual representation of the typechecked and annotated OCL file, or a message giving a type error.

The textual representation used in the output of the command line interface is by default normal OCL syntax, but extended with our annotations. To use our implementation with another, non-Haskell program, a format which is more suitable for parsing might be preferred. We have defined an example of such a simple interchange format, which is supported by the command line tool and for which we provide a Java parser.

The parsers used in our system are generated from context free grammars using the BNF converter (BNFC, [6]), a front-end to standard lexer and parser generators for Haskell, Java, C++, and C. The grammar used for generating the OCL parser is based on the grammars given in [7] and [5]. Using the LALR grammar in [2] instead is being investigated.

Using BNFC makes it simple to experiment with e.g. adjustments to the OCL grammar, or to define tailor-made interchange formats for the command line interface between our Haskell based system and other systems in Java, C++ or C.

The implementation is available on the web [11].

6 Conclusion

We have presented a rule system for type checking and semantic annotation of OCL, with disambiguation of implicit property calls and implicit bound variables as the main feature. It is intended to function as a component of a larger system, sitting in between an OCL parser and some other component performing

transformations on OCL specifications. So far it has only been tested in ongoing work in the KeY project [1] with translating OCL specifications to Natural Language. Besides adding features of OCL 1.5 and 2.0 currently missing from the system, this is the most important line of future work: to try to link it to other parts of the KeY project which handle OCL, e.g. the transformation of OCL to Dynamic Logic, and ongoing work with partial evaluation of OCL.

Acknowledgements: We thank Aarne Ranta for discussions on drafts of this paper, Daniel Larsson for feedback on the implementation, and the anonymous reviewers for suggestions on improving the paper.

References

1. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 2004. Online First issue, to appear in print.
2. David Akehurst and Octavian Patrascoiu. OCL: Implementing the Standard. In *OCL2.0-"Industry standard or scientific playground?" - Proceedings of the UML'03 workshop*, page 19. Electronic Notes in Theoretical Computer Science, November 2003.
3. María Victoria Cengarle and Alexander Knapp. Ocl 1.4/1.5 vs. 2.0 expressions: Formal semantics and expressiveness. In *Softw. Syst. Model.*, 2004.
4. Tony Clark. Type checking UML static diagrams. In Robert France and Bernhard Rumpe, editors, *UML'99—The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 503–517. Springer, 1999.
5. Frank Finger. Dresden OCL toolkit homepage, 2002. `http://dresden-ocl.sourceforge.net/`.
6. Markus Forsberg and Aarne Ranta. The BNF converter: A high-level tool for implementing well-behaved programming languages. In *NWPT'02 proceedings, Proceedings of the Estonian Academy of Sciences*, 2003.
7. Object Managment Group. OCL 1.5 specification, 2003. `http://www.omg.org/cgi-bin/apps/doc?formal/03-03-13.pdf`.
8. Object Managment Group. OCL 2.0 specification, 2003. `http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-14.pdf`.
9. Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta. An authoring tool for informal and formal requirements specifications. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering*, number 2306 in LNCS, 2002.
10. Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting OCL. In Andy Evans, Stuart Kent, and Bran Selic, editors, *Proc. 3rd Int. Conf. on the Unified Modeling Language*, LNCS 1939, pages 278–293. Springer, 2000.
11. Kristofer Johannisson. OCL tool implementation homepage, 2004. `http://www.cs.chalmers.se/~krijo/ocltc/`.
12. Aarne Ranta. Grammatical Framework: A Type-theoretical Grammar Formalism. *The Journal of Functional Programming*, 14(2):145–189, 2004.