A formal semantics for OCL 1.4

María Victoria Cengarle, Alexander Knapp

Angaben zur Veröffentlichung / Publication details:

Cengarle, María Victoria, and Alexander Knapp. 2001. "A formal semantics for OCL 1.4." *Lecture Notes in Computer Science* 2185: 118–33. https://doi.org/10.1007/3-540-45441-1_10.

A Formal Semantics for OCL 1.4

María Victoria Cengarle* and Alexander Knapp

Ludwig–Maximilians–Universität München
{cengarle,knapp}@informatik.uni-muenchen.de

Abstract. The OCL 1.4 specification introduces `let`-declarations for adding auxiliary class features in static structures of the UML. We provide a type inference system and a big-step operational semantics for the OCL 1.4 that treat UML static structures and UML object models abstractly and accommodate for additional declarations; the operational semantics satisfies a subject reduction property with respect to the type inference system. We also discuss an alternative, non-operational interpretation of `let`-declarations as constraints.

1 Introduction

The "Object Constraint Language" (OCL) allows to define constraints, like invariants and pre- and post-conditions, for models of the "Unified Modeling Language" (UML). The language has been extensively employed in the specification of the UML meta-model itself throughout UML 1.1. However, the meta-model constraints rely on the possibility of declaring auxiliary (meta-)class features, which was not provided for explicitly in OCL up to version 1.3. For example [12, p. 2-66], two additional features `parent` and `allParents` are declared for the meta-class `GeneralizableElement` in order to express the constraint that the inheritance relationship of a UML model must be acyclic. The OCL 1.4 specification [12, Ch. 6] defines the `let`-construct to introduce so-called pseudo-features for (meta-)classes, such that the acyclicity constraint could now be recast as:

```
context GeneralizableElement inv:
  let parent : Set(GeneralizableElement) =
        self.generalization.parent
  let allParents : Set(GeneralizableElement) =
        self.parent->union(self.parent.allParents)
  in not self.allParents->includes(self)
```

Though this `let`-construct is rather different from its conventional usage in functional languages like SML [10], the OCL specification does not provide a precise semantics, let alone for the whole language. In particular, it is not clear how `let`-declarations interfere with inheritance and whether arbitrary recursive defining expressions are allowed (cf. a comparable requirement on recursive post-conditions [12, p. 6-59]). Moreover [13], are additional features like `allParents` to be interpreted operationally (leading to non-termination when evaluated over a cyclic inheritance relationship) or should declarations of auxiliary features be regarded as a constraint?

* Current affiliation: Fraunhofer Institute for Experimental Software Engineering.

Several formal semantics for OCL have already been presented, in particular by Bickford and Guaspari [2], Hamie, Howse, and Kent [7], and Richters and Gogolla [14,16] for OCL 1.1 and by Clark [4] and the authors [3] for OCL 1.3. The `let`-construct of the previous OCL versions is neglected by all these semantics, with the exception of [3] where it is treated as an SML-style declaration. Moreover, these semantics show deficiencies in handling the OCL types `OclAny` and `OclType` [14,7], the OCL flattening rules [2,7], empty collections [14,4], undefined values [7,4], non-determinism [2,7,14], and overridden properties [2,7,14, 4]. Also, several OCL implementations have been provided, most noteworthy the Bremen USE tool [15], the Dresden OCL tool [8], and the pUML "Meta-Modelling Tool" (MMT [9]). These differ e.g. in their handling of collections and `oclAsType`; the USE tool does not include `let`, the Dresden tool and MMT also seem to handle the `let`-construct as an SML-style declaration.

We provide an improved and more comprehensive formal semantics of the OCL 1.4 including its main novelty, the possibility of declaring pseudo-features. We axiomatise UML static structures and UML object models such that the semantics is parametric in the treatment of declarations. We introduce a type inference and annotation system for OCL terms (Sect. 2) and define a big-step operational semantics that evaluates annotated terms (Sect. 3); the operational semantics satisfies a subject reduction property with respect to the type inference system. Finally, we discuss an alternative, non-operational interpretation of declarations as model constraints (Sect. 4). We conclude with some remarks on future work.

We assume a working knowledge of the OCL syntax and informal semantics. The concrete syntax of the OCL sub-language that we consider can be found in Table 1; in particular, we omit navigation to association classes and through qualified associations, templates, package pathnames, enumerations, the types `OclExpression` and `OclState`, the functions that can be defined by `iterate`, pre- and in-fix syntax, and pre- and post-conditions.

Table 1. OCL syntax fragment

Term ::= *Constr* | *Inv* | *Decl* | *Expr*
Constr ::= `context` *Type* `inv:` *Inv* {`inv:` *Inv*}
Inv ::= [*Decl* {*Decl*} `in`] *Expr*
Decl ::= `let` *Name* [`(` *Var* `:` *Type* {`,` *Var* `:` *Type*} `)`] `:` *Type* `=` *Expr*
Expr ::= *Literal* | `self` | *Var* | *Type* |
`(Set` | `Bag` | `Sequence)` `{` [*Expr* {`,` *Expr*}] `}` |
`if` *Expr* `then` *Expr* `else` *Expr* `endif` |
Expr `->` `iterate` `(` *Var* `:` *Type* `;` *Var* `:` *Type* `=` *Expr* `|` *Expr* `)` |
Expr `.` `oclAsType` `(` *Type* `)` | *Type* `.` `allInstances` `(` `)` |
Expr `.` `and` `(` *Expr* `)` | *Expr* `.` `or` `(` *Expr* `)` |
Expr `.` *Name* [`(` [*Expr* {`,` *Expr*}] `)`] | *Expr* `->` *Name* `(` [*Expr* {`,` *Expr*}] `)`
Literal ::= *IntegerLiteral* | *RealLiteral* | *BooleanLiteral* | *StringLiteral*
Type ::= *Name* | (`Set` | `Bag` | `Sequence` | `Collection`) `(` *Name* `)`
Var ::= *Name*

2 Type System

The type of an OCL term depends on information from an underlying UML static structure, its classifiers, structural and query behavioural features, generalisation relationship, opposite association ends, &c., and the built-in OCL types and properties. We abstractly axiomatise this information as a static basis which is parametric in the classifiers and the generalisation relationship and provides an extension mechanism by pseudo-features; the axiomatisation also captures the declaration retrieval of (overloaded) features and properties that is only vaguely described in the UML specification by full-descriptors [12, p. 2-75]. We present a type inference system for OCL terms over such a static basis that also annotates the terms for later evaluation of overloaded features and properties and pseudo-features; the type system entails unique annotations and types.

2.1 Static Bases

A *static basis* Ω defines types, a type hierarchy, functions for declaration retrieval, and an extension mechanism for declarations.

Types. The *(compile-time) types* T_Ω of a static basis Ω are defined as follows:

$$\begin{array}{ll} T_\Omega ::= \overline{A}_\Omega \mid \overline{S}\,\texttt{(}\,\overline{A}_\Omega\,\texttt{)} & B ::= \texttt{Integer} \mid \texttt{Real} \mid \texttt{Boolean} \mid \texttt{String} \\ \overline{A}_\Omega ::= A_\Omega \mid \texttt{OclType} & \overline{S} ::= S \mid \texttt{Collection} \\ A_\Omega ::= \texttt{Void} \mid B \mid C_\Omega \mid \texttt{OclAny} & S ::= \texttt{Set} \mid \texttt{Bag} \mid \texttt{Sequence} \end{array}$$

where the set parameter C_Ω represents a set of *classifiers*, that does not contain `Integer`, `Real`, `Boolean`, `String`, `Void`, `OclAny`, and `OclType`.

The set B contains all built-in simple OCL types, the *basic types*. The type `Void` is not required by the OCL specification; it denotes the empty type. The set A_Ω comprises `Void`, the basic types, the *classifier types*, and `OclAny`, which is the common super-type of all basic and classifier types. The type `OclType` is the type of all types (as used in impredicative polymorphism [11]). Finally, the set S defines the concrete collection type functions yielding, when applied to a type parameter, a concrete *collection type*; $\overline{S}$ adds the abstract collection type function `Collection` that yields the abstract collection type.

Each *Literal* l has a type, written as $\mathit{type}(l)$, such that $\mathit{type}(n) = \texttt{Integer}$ if n is an *IntegerLiteral*, &c. The type parameter of a collection type $\overline{\sigma}\texttt{(}\tau\texttt{)}$ may be recovered by $\mathit{base}(\overline{\sigma}\texttt{(}\tau\texttt{)}) = \tau$; for simplicity, we set $\mathit{base}(\tau) = \tau$ if τ is not a collection type.

Type hierarchy. The *subtype relation* $\leq_\Omega$ of a static basis Ω is defined as the least partial order that satisfies the following axioms:

1. for all $\tau \in T_\Omega$, $\texttt{Void} \leq_\Omega \tau$
2. for all $\alpha \in A_\Omega$, $\alpha \leq_\Omega \texttt{OclAny}$
3. $\texttt{Integer} \leq_\Omega \texttt{Real}$

4. for all $\zeta_1, \zeta_2 \in C_\Omega$, if $\zeta_1 \leq_{C_\Omega} \zeta_2$, then $\zeta_1 \leq_\Omega \zeta_2$
5. for all $\sigma \in S$ and $\overline{\alpha} \in \overline{A}_\Omega$, $\sigma(\overline{\alpha}) \leq_\Omega \texttt{Collection}(\overline{\alpha})$
6. for all $\overline{\sigma} \in \overline{S}$ and $\overline{\alpha}_1, \overline{\alpha}_2 \in \overline{A}_\Omega$, if $\overline{\alpha}_1 \leq_\Omega \overline{\alpha}_2$, then $\overline{\sigma}(\overline{\alpha}_1) \leq_\Omega \overline{\sigma}(\overline{\alpha}_2)$

where the partial order parameter $\leq_{C_\Omega}$ denotes the *generalisation hierarchy* on the classifier types C_Ω.

In particular, $\texttt{OclType} \not\leq_\Omega \texttt{OclAny}$ (in contrast to [9]) and $\sigma(\tau) \not\leq_\Omega \texttt{OclAny}$. According to [12, p. 6-54] collection types are basic types, following [12, pp. 6-75f.] these types are *not* basic types (cf. [1]). We choose the second definition (in contrast to [14]), avoiding the Russell paradox that could arise from $\texttt{Set(OclAny)} \leq_\Omega \texttt{OclAny}$ (see [2]); however, note, that thus none of the properties of `OclAny`, like inequality or `oclIsKindOf`, is immediately available for collections. Moreover, $\tau \leq_\Omega \mathit{base}(\tau)$ if, and only if $\tau \in \overline{A}_\Omega$.

We denote by $\bigsqcup_\Omega\{\tau_1, \ldots, \tau_n\}$ the least upper bound of types $\tau_1, \ldots, \tau_n$ with respect to $\leq_\Omega$; simultaneously, when writing $\bigsqcup_\Omega\{\tau_1, \ldots, \tau_n\}$ we assume this least upper bound to exist (which may not be the case in the presence of multiple inheritance). Note that $\bigsqcup_\Omega \emptyset = \texttt{Void}$.

Declaration retrieval. The retrieval of (overridden) properties, features, pseudo-features, and opposite association ends in a static basis Ω is defined by two suitably axiomatised maps yielding *declarations* in

$$D_\Omega ::= T_\Omega \ . \ \mathit{Name} : T_\Omega \mid T_\Omega \ . \ \mathit{Name} : T_\Omega^* \to T_\Omega \ .$$

Given a name a and a type τ, the partial function $\mathit{fd}_\Omega : \mathit{Name} \times T_\Omega \rightharpoonup D_\Omega$ yields, when defined, a declaration $\tau'.a : \tau''$ such that $\tau \leq_\Omega \tau'$. The type τ' represents a type that shows a structural (pseudo-)feature or an opposite association end with name a of type τ''. If $\mathit{fd}_\Omega(a, \tau)$ is defined, then $\mathit{fd}_\Omega(a, \tau')$ is defined for all $\tau' \leq_\Omega \tau$, i.e., a is inherited to all subtypes of τ.

Analogously, given a name o, a type τ, and a sequence of types $(\tau_i)_{1 \leq i \leq n}$ the partial function $\mathit{fd}_\Omega : \mathit{Name} \times T_\Omega \times T_\Omega^* \rightharpoonup D_\Omega$ yields, when defined, a declaration $\tau'.o : (\tau_i')_{1 \leq i \leq n} \to \tau_0'$ such that $\tau \leq_\Omega \tau'$ and $\tau_i \leq_\Omega \tau_i'$ for all $1 \leq i \leq n$. The type τ' represents a type that shows a query behavioural (pseudo-)feature or a property with name o, parameter types $\tau_1', \ldots, \tau_n'$, and return type τ_0'. If $\mathit{fd}_\Omega(o, \tau, (\tau_i)_{1 \leq i \leq n})$ is defined, then $\mathit{fd}_\Omega(o, \tau', (\tau_i)_{1 \leq i \leq n})$ is defined for all $\tau' \leq_\Omega \tau$.

Table 2 shows some sample axioms for OCL properties, where $\alpha, \alpha' \in A_\Omega$ and $\overline{\alpha}, \overline{\alpha}' \in \overline{A}_\Omega$; all other OCL properties [12, Sect. 6.8] may be added analogously. We require these axioms for all static bases Ω.

Extensions. We require that a static basis Ω be *extendable* by a declaration of a structural pseudo-feature $\zeta.a : \tau$ with $\zeta \in C_\Omega$, if $\mathit{fd}_\Omega(a, \zeta)$ is undefined, and by a declaration of a behavioural pseudo-feature $\zeta.o : (\tau_i)_{1 \leq i \leq n} \to \tau_0$ with $\zeta \in C_\Omega$, if $\mathit{fd}_\Omega(o, \zeta, (\tau_i)_{1 \leq i \leq n})$ is undefined. Such an extension Ω' of Ω must again be a static basis. For the extension by a declaration $\delta = \zeta.a : \tau$ we require that $\mathit{fd}_{\Omega'}(a, \zeta) = \delta$ and that $\mathit{fd}_{\Omega'}(a', \zeta')$ is the same as $\mathit{fd}_\Omega(a', \zeta')$ if $a' \neq a$; and by a declaration $\delta = \zeta.o : (\tau_i)_{1 \leq i \leq n} \to \tau_0$ that $\mathit{fd}_{\Omega'}(o, \zeta, (\tau_i)_{1 \leq i \leq n}) = \delta$ and that

Table 2. Typing of sample built-in OCL properties

$$fd_\Omega(\texttt{=}, \alpha, \alpha') = \alpha.\texttt{=} : \alpha' \to \texttt{Boolean}$$
$$fd_\Omega(\texttt{oclIsKindOf}, \alpha, \texttt{OclType}) = \alpha.\texttt{oclIsKindOf} : \texttt{OclType} \to \texttt{Boolean}$$
$$fd_\Omega(\texttt{first}, \texttt{Sequence}(\overline{\alpha})) = \texttt{Sequence}(\overline{\alpha}).\texttt{first} : \to \overline{\alpha}$$
$$fd_\Omega(\texttt{including}, \sigma(\overline{\alpha}), \overline{\alpha}') = \sigma(\overline{\alpha}).\texttt{including} : \overline{\alpha}' \to \sigma(\textstyle\bigsqcup_\Omega\{\overline{\alpha}, \overline{\alpha}'\})$$
$$fd_\Omega(\texttt{union}, \sigma(\overline{\alpha}), \sigma(\overline{\alpha}')) = \sigma(\overline{\alpha}).\texttt{union} : \sigma(\overline{\alpha}') \to \sigma(\textstyle\bigsqcup_\Omega\{\overline{\alpha}, \overline{\alpha}'\})$$

$fd_{\Omega'}(o', \zeta', (\tau_i')_{1\le i\le n})$ is the same as $fd_\Omega(o', \zeta', (\tau_i')_{1\le i\le n})$ if $o' \neq o$. Moreover, we must have $T_{\Omega'} = T_\Omega$ and $\le_{\Omega'} = \le_\Omega$.

These requirements only weakly characterise possible extension mechanisms for declarations. We assume that some scheme of extending static bases is fixed and we write Ω, δ for the extension of a static basis Ω by the declaration δ according to the chosen scheme.

2.2 Type Inference

The type inference system on the one hand allows to deduce the type of a given OCL term over a given static basis. On the other hand, the inference system produces a normalised and annotated OCL term adding type information on declarations and overridden properties for later evaluation.

The grammar for *annotated* OCL terms transforms the grammar in Table 1 by consistently replacing *Term*, *Constr*, *Inv*, *Decl*, and *Expr* by *A-Term*, *A-Constr*, *A-Inv*, *A-Decl*, and *A-Expr*, respectively; furthermore, the original clauses

> `let` *Name* [`(` *Var* `:` *Type* { `,` *Var* `:` *Type* } `)`] `:` *Type* `=` *Expr*
> *Expr* `.` *Name* [`(` [*Expr* { `,` *Expr* }] `)`] | *Expr* `->` *Name* `(` [*Expr* { `,` *Expr* }] `)`

are replaced by

> `let` *Name Type* [`(` *Var* `:` *Type* { `,` *Var* `:` *Type* } `)`] `:` *Type* `=` *A-Expr*
> *A-Expr* `.` *Name Type* [`(` [*A-Expr* { `,` *A-Expr* }] `)`] |
> *A-Expr* `->` *Name Type* `(` [*A-Expr* { `,` *A-Expr* }] `)`

The annotations by *Type* for `let` and $\tilde{e}.a$ record the defining class, the annotations for $\tilde{e}.o(\ldots)$ and $\tilde{e}\texttt{->}o(\ldots)$ the expected return type. Annotations are written as subscripts.

A *type environment* over a static basis Ω is a finite sequence Γ of variable typings of the form $x_1 : \tau_1, \ldots, x_n : \tau_n$ with $x_i \in \mathit{Var} \cup \{\texttt{self}\}$ and $\tau_i \in T_\Omega$ for all $1 \le i \le n$; we denote $\{x_1, \ldots, x_n\}$ by $\mathrm{dom}(\Gamma)$, and τ_i by $\Gamma(x_i)$ if $x_j \neq x_i$ for all $i < j \le n$. The empty type environment is denoted by $\emptyset$, concatenation of type environments Γ and Γ' by Γ, Γ'.

The type inference system consists of judgements of the form $\Omega; \Gamma \vdash t \triangleright \tilde{t} : \theta$ where Ω is a static basis, Γ is a type environment over Ω, t is a *Term*, $\tilde{t}$ is an *A-Term*, and $\theta \in T_\Omega \cup D_\Omega$. When writing such a judgement, we assume that

`self`, `oclAsType`, `allInstances`, `and`, and `or` are reserved names and that *Var* and $T_\Omega \subseteq$ *Type* are disjoint. The empty type environment may be omitted.

The judgement relation $\vdash$ is defined by the rules in Tables 3–4; a rule may only be applied if all its constituents are well-defined. The meta-variables that are used in the rules and which may be variously decorated range as follows: $l \in$ *Literal*; $\alpha \in A_\Omega$, $\overline{\alpha} \in \overline{A}_\Omega$, $\zeta \in C_\Omega$, $\sigma \in S$, $\overline{\sigma} \in \overline{S}$, $\tau \in T_\Omega$, $\delta \in D_\Omega$; $x \in$ *Var*; $a, o \in$ *Name*; $e \in$ *Expr*, $\tilde{e} \in$ *A-Expr*, $d \in$ *Decl*, $\tilde{d} \in$ *A-Decl*, $p \in$ *Inv*, $\tilde{p} \in$ *A-Inv*.

The rules follow the OCL specification [12, Ch. 6] as closely as possible. The (Inv^τ) and (Decl_1^τ–Decl_2^τ) rules in Table 3 treat `let`-declarations as being simultaneous; dependent declarations may be easily introduced (cf. [10]). The rule (Coll^τ) provides a unique type for the empty concrete collections (in contrast to [4]) and for "flattening" nested collections; for a motivation see [12, p. 6-67], though the least upper bound is not directly justified by the specification (in particular, Schürr [17] suggests to employ union-types instead; [4,15] require homogenous collections; the typing rules of [8] depend on the expression order; [9] shows no flattening). The type of a conditional expression, as given by the (Cond^τ) rule, differs from what is stated in [12, p. 6-83]: there, independently of the type of e_2, the (evaluation) type of e_1 is assumed to be the type of the whole expression ([14] requires comparable types, [4] a single type). For the casting rule (Cast^τ) see [12, pp. 6-56, 6-63f., 6-77] ([14] requires that the new type is smaller than the original type; not present in [7,4]); note, however, that this rule does not allow for arbitrary expressions resulting in a type as the argument for `oclAsType`, since this would imply term-dependent types as, for example, in

```
5.oclAsType(if 1.=(2) then Real else Integer endif) .
```

By the same argument, (Inst^τ) does not allow `allInstances` to be called on arbitrary expressions, but only type literals. The annotation in (Feat_1^τ) in Table 4 accounts for the retrieval of an overridden structural feature or opposite association end [12, pp. 6-63f.] (not present in [14,4]); the annotations in (Feat_2^τ) and (Prop^τ) are necessary, since we do not require any return type restrictions for query behavioural features and properties (in contrast to [4]). The rules (Sing_1^τ–Sing_2^τ) in Table 4 are the so-called "singleton" rules, see [12, pp. 6-60f.], allowing to apply collection properties to non-collection expressions (not present in [14, 4]). The (Short_1^τ–Short_2^τ) rules define the shorthand notation for features on members of collections [12, p. 6-71] combined with flattening [12, p. 6-67] (not present in [2,7,14,4]). There is no subsumption rule; such a rule would interfere with the overriding of properties and features (cf. e.g. [5]).

The type inference system entails unique annotations and types:

Proposition 1. *Let Ω be a static basis, Γ a type environment, and t a Term. If $\Omega; \Gamma \vdash t \triangleright \tilde{t} : \tau$ and $\Omega; \Gamma \vdash t \triangleright \tilde{t}' : \tau'$ for some A-Term's $\tilde{t}$ and $\tilde{t}'$ and types τ and τ', then $\tilde{t} = \tilde{t}'$ and $\tau = \tau'$.*

Proof. By induction on the term t.

We also write $\Omega; \Gamma \vdash \tilde{t} : \tau$ if $\Omega; \Gamma \vdash t \triangleright \tilde{t} : \tau$ where $t \in$ *Term* is obtained from $\tilde{t} \in$ *A-Term* by erasing the annotations; t and $\tilde{t}$ are called *well-typed*.

Table 3. Type inference system I

$$(\mathrm{Ctxt}^{\tau}) \quad \frac{(\Omega; \Gamma, \texttt{self} : \zeta \vdash p_i \triangleright \tilde{p}_i : \texttt{Boolean})_{1 \le i \le n}}{\begin{array}{l}\Omega; \Gamma \vdash \texttt{context}\ \zeta\ (\texttt{inv:}\ p_i)_{1 \le i \le n} \triangleright \\ \qquad \texttt{context}\ \zeta\ (\texttt{inv:}\ \tilde{p}_i)_{1 \le i \le n} : \texttt{Boolean}\end{array}}$$

$$(\mathrm{Inv}^{\tau}) \quad \frac{\begin{array}{c}(\Omega, (\delta_j)_{1 \le j \le n}; \Gamma \vdash d_i \triangleright \tilde{d}_i : \delta_i)_{1 \le i \le n} \\ \Omega, (\delta_j)_{1 \le j \le n}; \Gamma \vdash e \triangleright \tilde{e} : \tau\end{array}}{\begin{array}{l}\Omega; \Gamma \vdash (d_i)_{1 \le i \le n}\ \texttt{in}\ e \triangleright (\tilde{d}_i)_{1 \le i \le n}\ \texttt{in}\ \tilde{e} : \tau\end{array}}$$

$$(\mathrm{Decl}_1^{\tau}) \quad \frac{\Omega; \Gamma \vdash e \triangleright \tilde{e} : \tau'}{\Omega; \Gamma \vdash \texttt{let}\ x\ \texttt{:}\ \tau\ \texttt{=}\ e \triangleright \texttt{let}\ x_\zeta\ \texttt{:}\ \tau\ \texttt{=}\ \tilde{e} : \zeta.x : \tau}$$

if $\tau' \le_\Omega \tau$ and where $\zeta = \Gamma(\texttt{self})$

$$(\mathrm{Decl}_2^{\tau}) \quad \frac{\Omega; \Gamma, (x_i : \tau_i)_{1 \le i \le n} \vdash e \triangleright \tilde{e} : \tau'}{\begin{array}{l}\Omega; \Gamma \vdash \texttt{let}\ x(x_1 : \tau_1, \ldots, x_n : \tau_n)\ \texttt{:}\ \tau\ \texttt{=}\ e \triangleright \\ \qquad \texttt{let}\ x_\zeta(x_1 : \tau_1, \ldots, x_n : \tau_n)\ \texttt{:}\ \tau\ \texttt{=}\ \tilde{e} : \zeta.x : (\tau_i)_{1 \le i \le n} \to \tau\end{array}}$$

if $\tau' \le_\Omega \tau$ and where $\zeta = \Gamma(\texttt{self})$

(Lit^{τ}) $\Omega; \Gamma \vdash l \triangleright l : \mathit{type}(l)$ $\qquad$ (Self^{τ}) $\Omega; \Gamma \vdash \texttt{self} \triangleright \texttt{self} : \Gamma(\texttt{self})$

(Var^{τ}) $\Omega; \Gamma \vdash x \triangleright x : \Gamma(x)$ $\qquad$ (Type^{τ}) $\Omega; \Gamma \vdash \tau \triangleright \tau : \texttt{OclType}$

$$(\mathrm{Coll}^{\tau}) \quad \frac{(\Omega; \Gamma \vdash e_i \triangleright \tilde{e}_i : \tau_i)_{1 \le i \le n}}{\begin{array}{l}\Omega; \Gamma \vdash \sigma\{e_1, \ldots, e_n\} \triangleright \\ \qquad \sigma\{\tilde{e}_1, \ldots, \tilde{e}_n\} : \sigma(\alpha)\end{array}} \quad \text{where } \alpha = \bigsqcup\nolimits_\Omega \{\mathit{base}(\tau_i) \mid 1 \le i \le n\}$$

$$(\mathrm{Cond}^{\tau}) \quad \frac{\begin{array}{c}\Omega; \Gamma \vdash e \triangleright \tilde{e} : \texttt{Boolean} \\ (\Omega; \Gamma \vdash e_i \triangleright \tilde{e}_i : \tau_i)_{1 \le i \le 2}\end{array}}{\begin{array}{l}\Omega; \Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{endif} \triangleright \\ \qquad \texttt{if}\ \tilde{e}\ \texttt{then}\ \tilde{e}_1\ \texttt{else}\ \tilde{e}_2\ \texttt{endif} : \tau\end{array}} \quad \text{where } \tau = \bigsqcup\nolimits_\Omega \{\tau_1, \tau_2\}$$

$$(\mathrm{Iter}^{\tau}) \quad \frac{\begin{array}{c}\Omega; \Gamma \vdash e \triangleright \tilde{e} : \overline{\sigma}(\overline{\alpha}') \qquad \Omega; \Gamma \vdash e' \triangleright \tilde{e}' : \tau' \\ \Omega; \Gamma, x : \overline{\alpha}, x' : \tau \vdash e'' \triangleright \tilde{e}'' : \tau''\end{array}}{\begin{array}{l}\Omega; \Gamma \vdash e\texttt{->iterate(}x : \overline{\alpha}\texttt{; } x' : \tau\ \texttt{=}\ e' \texttt{ | } e''\texttt{)} \triangleright \\ \qquad \tilde{e}\texttt{->iterate(}x : \overline{\alpha}\texttt{; } x' : \tau\ \texttt{=}\ \tilde{e}' \texttt{ | } \tilde{e}''\texttt{)} : \tau\end{array}} \quad \begin{array}{l}\text{if } \overline{\alpha}' \le_\Omega \overline{\alpha} \text{ and} \\ \tau', \tau'' \le_\Omega \tau\end{array}$$

$$(\mathrm{Cast}^{\tau}) \quad \frac{\Omega; \Gamma \vdash e \triangleright \tilde{e} : \tau}{\Omega; \Gamma \vdash e\texttt{.oclAsType(}\tau'\texttt{)} \triangleright \tilde{e}\texttt{.oclAsType(}\tau'\texttt{)} : \tau'} \quad \text{if } \tau \le_\Omega \tau' \text{ or } \tau' \le_\Omega \tau$$

(Inst^{τ}) $\Omega; \Gamma \vdash \tau\texttt{.allInstances()} \triangleright \tau\texttt{.allInstances()} : \texttt{Set(}\mathit{base}(\tau)\texttt{)}$

$$(\mathrm{And}^{\tau}) \quad \frac{(\Omega; \Gamma \vdash e_i \triangleright \tilde{e}_i : \texttt{Boolean})_{1 \le i \le 2}}{\begin{array}{l}\Omega; \Gamma \vdash e_1\texttt{.and(}e_2\texttt{)} \triangleright \\ \qquad \tilde{e}_1\texttt{.and(}\tilde{e}_2\texttt{)} : \texttt{Boolean}\end{array}} \qquad (\mathrm{Or}^{\tau}) \quad \frac{(\Omega; \Gamma \vdash e_i \triangleright \tilde{e}_i : \texttt{Boolean})_{1 \le i \le 2}}{\begin{array}{l}\Omega; \Gamma \vdash e_1\texttt{.or(}e_2\texttt{)} \triangleright \\ \qquad \tilde{e}_1\texttt{.or(}\tilde{e}_2\texttt{)} : \texttt{Boolean}\end{array}}$$

Table 4. Type inference system II

$$(\text{Feat}_1^\tau)\quad \frac{\Omega;\Gamma \vdash e \triangleright \tilde{e} : \tau}{\Omega;\Gamma \vdash e.a \triangleright \tilde{e}.a_{\tau'} : \tau''} \quad \text{if } fd_\Omega(a,\tau) = \tau'.a : \tau''$$

$$(\text{Feat}_2^\tau)\quad \frac{\begin{array}{c}\Omega;\Gamma \vdash e \triangleright \tilde{e} : \tau \\ (\Omega;\Gamma \vdash e_i \triangleright \tilde{e}_i : \tau_i)_{1\le i\le n}\end{array}}{\begin{array}{l}\Omega;\Gamma \vdash e.o(e_1, \ldots, e_n) \triangleright \\ \quad \tilde{e}.o_{\tau'_0}(\tilde{e}_1, \ldots, \tilde{e}_n) : \tau'_0\end{array}} \quad \begin{array}{l}\text{if } fd_\Omega(o,\tau,(\tau_i)_{1\le i\le n}) = \\ \quad \tau'.o : (\tau'_i)_{1\le i\le n} \to \tau'_0\end{array}$$

$$(\text{Prop}^\tau)\quad \frac{\begin{array}{c}\Omega;\Gamma \vdash e \triangleright \tilde{e} : \overline{\sigma}(\overline{\alpha}) \\ (\Omega;\Gamma \vdash e_i \triangleright \tilde{e}_i : \tau_i)_{1\le i\le n}\end{array}}{\begin{array}{l}\Omega;\Gamma \vdash e\texttt{->}o(e_1, \ldots, e_n) \triangleright \\ \quad \tilde{e}\texttt{->}o_{\tau'_0}(\tilde{e}_1, \ldots, \tilde{e}_n) : \tau'_0\end{array}} \quad \begin{array}{l}\text{if } fd_\Omega(o,\overline{\sigma}(\overline{\alpha}),(\tau_i)_{1\le i\le n}) = \\ \quad \tau'.o : (\tau'_i)_{1\le i\le n} \to \tau'_0\end{array}$$

$$(\text{Sing}_1^\tau)\quad \frac{\begin{array}{c}\Omega;\Gamma \vdash e \triangleright \tilde{e} : \overline{\alpha}' \qquad \Omega;\Gamma \vdash e' \triangleright \tilde{e}' : \tau' \\ \Omega;\Gamma, x : \overline{\alpha}, x' : \tau \vdash e'' \triangleright \tilde{e}'' : \tau''\end{array}}{\begin{array}{l}\Omega;\Gamma \vdash e\texttt{->iterate(}x : \overline{\alpha}\texttt{; } x' : \tau \texttt{ = } e' \texttt{ | } e''\texttt{)} \triangleright \\ \quad \texttt{Set\{}\tilde{e}\texttt{\}->iterate(}x : \overline{\alpha}\texttt{; } x' : \tau \texttt{ = } \tilde{e}' \texttt{ | } \tilde{e}''\texttt{)} : \tau\end{array}} \quad \begin{array}{l}\text{if } \overline{\alpha}' \le_\Omega \overline{\alpha} \text{ and} \\ \quad \tau', \tau'' \le_\Omega \tau\end{array}$$

$$(\text{Sing}_2^\tau)\quad \frac{\begin{array}{c}\Omega;\Gamma \vdash e \triangleright \tilde{e} : \overline{\alpha} \\ (\Omega;\Gamma \vdash e_i \triangleright \tilde{e}_i : \tau_i)_{1\le i\le n}\end{array}}{\begin{array}{l}\Omega;\Gamma \vdash e\texttt{->}o(e_1, \ldots, e_n) \triangleright \\ \quad \texttt{Set\{}\tilde{e}\texttt{\}->}o_{\tau'_0}(\tilde{e}_1, \ldots, \tilde{e}_n) : \tau'_0\end{array}} \quad \begin{array}{l}\text{if } fd_\Omega(o,\texttt{Set}(\overline{\alpha}),(\tau_i)_{1\le i\le n}) = \\ \quad \tau'.o : (\tau'_i)_{1\le i\le n} \to \tau'_0\end{array}$$

$$(\text{Short}_1^\tau)\quad \frac{\Omega;\Gamma \vdash e \triangleright \tilde{e} : \overline{\sigma}(\overline{\alpha})}{\begin{array}{l}\Omega;\Gamma \vdash e.a \triangleright \tilde{e}\texttt{->iterate(i : } \overline{\alpha}\texttt{; a : } \sigma(\overline{\alpha}') \texttt{ = } \sigma\texttt{\{\} |} \\ \qquad\qquad \texttt{a->union}_{\sigma(\overline{\alpha}')}(\sigma\texttt{\{i.}a_\tau\texttt{\}))} : \sigma(\overline{\alpha}')\end{array}}$$

if $fd_\Omega(a,\overline{\alpha}) = \tau.a : \overline{\alpha}'$ and
$(\overline{\sigma} \neq \texttt{Sequence}, \sigma = \texttt{Bag})$ or $(\sigma = \overline{\sigma} = \texttt{Sequence})$
or $fd_\Omega(a,\overline{\alpha}) = \tau.a : \sigma(\overline{\alpha}')$

$$(\text{Short}_2^\tau)\quad \frac{\Omega;\Gamma \vdash e \triangleright \tilde{e} : \overline{\sigma}(\overline{\alpha}) \qquad (\Omega;\Gamma \vdash e_i \triangleright \tilde{e}_i : \tau_i)_{1\le i\le n}}{\begin{array}{l}\Omega;\Gamma \vdash e.o(e_1, \ldots, e_n) \triangleright \\ \quad \tilde{e}\texttt{->iterate(i : } \overline{\alpha}\texttt{; a : } \sigma(\overline{\alpha}') \texttt{ = } \sigma\texttt{\{\} |} \\ \qquad\qquad \texttt{a->union}_{\sigma(\overline{\alpha}')}(\sigma\texttt{\{i.}o_{\tau'_0}(\tilde{e}_1, \ldots, \tilde{e}_n)\texttt{\})}) : \sigma(\overline{\alpha}')\end{array}}$$

if $fd_\Omega(o,\overline{\alpha},(\tau_i)_{1\le i\le n}) = \tau.o : (\tau'_i)_{1\le i\le n} \to \overline{\alpha}' = \tau'_0$ and
$(\overline{\sigma} \neq \texttt{Sequence}$ and $\sigma = \texttt{Bag})$ or $(\sigma = \overline{\sigma} = \texttt{Sequence})$
or $fd_\Omega(a,\overline{\alpha},(\tau_i)_{1\le i\le n}) = \tau.o : (\tau'_i)_{1\le i\le n} \to \sigma(\overline{\alpha}') = \tau'_0$

3 Operational Semantics

The result of an OCL term depends on information from an underlying UML object model, the instances and their types, the values of structural features, and the implementations of query behavioural features of instances, as well as the implementations of the built-in OCL properties. We abstractly summarise this information in a dynamic basis which is the dynamic counterpart of static bases and is axiomatised analogously. We define a big-step operational semantics for annotated OCL terms and a conformance relation between static and dynamic bases, such that the semantics satisfies a subject reduction property with respect to the type system of the previous section.

3.1 Dynamic Bases

A *dynamic basis* ω defines values, results, a typing relation, implementation retrieval functions, and an extension mechanism for implementations.

Values and results. The values V_ω and results R_ω of a dynamic basis ω are defined as follows:

$$\begin{aligned} R_\omega &::= V_\omega \mid \bot \\ V_\omega &::= N_\omega \mid (\texttt{Set} \mid \texttt{Bag} \mid \texttt{Sequence})\,\texttt{\{}\,[N_\omega\,\{\texttt{,}\,N_\omega\}]\,\texttt{\}} \\ N_\omega &::= \mathit{Literal} \mid T_\omega \mid O_\omega \end{aligned}$$

The *(run-time) types* T_ω are defined as T_Ω for a static basis Ω in Sect. 2.1, but replacing C_Ω by a set parameter C_ω, again representing classifiers. The finite set parameter O_ω represents the *instances* in a model, disjoint from *Literal* and T_ω.

The result $\bot$ represents "undefined". Values of the form $\sigma\texttt{\{}\ldots\texttt{\}}$ with $\sigma \in S = \{\texttt{Set}, \texttt{Bag}, \texttt{Sequence}\}$ are *collection values*, the values in N_ω are *simple values*. We assume suitably axiomatised arithmetical, boolean, &c. functions and relations on values such that, e.g., $\texttt{1} + \texttt{1} = \texttt{2}$, $\texttt{false} \wedge \texttt{true} = \texttt{false}$, $\texttt{Set\{ 1, 2 \}} = \texttt{Set\{ 2, 1, 1 \}}$, $\texttt{1} \leq \texttt{2}$, &c.

For collection values, we use a function $\mathit{flatten}_\omega : V_\omega \to N_\omega^*$ to sequences of simple values, stipulating that $\mathit{flatten}_\omega(\sigma\texttt{\{}v_1\texttt{, }\ldots\texttt{, }v_n\texttt{\}}) = v_1 \cdots v_n$ and that $\mathit{flatten}_\omega(v) = v$, if $v \in N_\omega$. Collection values are constructed by a map $\mathit{make}_\omega : S \times V_\omega^* \to V_\omega$ such that $\mathit{make}_\omega(\sigma, v_1 \cdots v_n) = \sigma\texttt{\{}v_1\texttt{, }\ldots\texttt{, }v_n\texttt{\}}$ if $\sigma \in S$ and $v_i \in N_\omega$ for all $1 \leq i \leq n$, and, if $v_i \in V_\omega \setminus N_\omega$ for some $1 \leq i \leq n$ then $\mathit{make}_\omega(\sigma, v_1 \cdots v_n) = \mathit{make}_\omega(\sigma, \mathit{flatten}_\omega(v_1) \cdots \mathit{flatten}_\omega(v_n))$; if $\sigma = \texttt{Set}$, repetitions in $\mathit{flatten}_\omega(v_1) \cdots \mathit{flatten}_\omega(v_n)$ are discarded, such that only the leftmost occurrence of a value remains. If $n = 0$, we write $\mathit{make}_\omega(\sigma, \emptyset)$; more generally, for a set $M = \{v_1, \ldots, v_n\}$, we let $\mathit{make}_\omega(\sigma, M)$ denote $\mathit{make}_\omega(\sigma, v_1 \cdots v_n)$.

A collection value $v = \sigma\texttt{\{}v_1\texttt{, }\ldots\texttt{, }v_n\texttt{\}}$ has a *sequence value representation* v', written as $v \rightsquigarrow v'$, if $\mathit{make}_\omega(\texttt{Sequence}, \mathit{flatten}_\omega(\mathit{make}_\omega(\sigma, \mathit{flatten}_\omega(v'')))) = v'$ for some $v'' = v$. In general, a collection value has several different sequence value representations; e.g. $\texttt{Set\{1, 2\}} \rightsquigarrow \texttt{Sequence\{1, 2\}}$ and also $\texttt{Set\{1, 2\}} \rightsquigarrow \texttt{Sequence\{2, 1\}}$; but not $\texttt{Set\{1, 2\}} \rightsquigarrow \texttt{Sequence\{1, 1, 2\}}$.

Typing relation. We require a relation $:_\omega \subseteq V_\omega \times T_\omega$ between values and types defined by the least left-total relation satisfying the following axioms:

1. For $v \in \mathit{Literal}$, $v :_\omega \mathit{type}(v)$
2. For $v \in T_\omega$, $v :_\omega \texttt{OclType}$
3. For $v \in O_\omega$, if $v :_{O_\omega} \tau$ then $v :_\omega \tau$
4. $\sigma\texttt{\{\}} :_\omega \sigma\texttt{(Void)}$
5. If $v :_\omega \texttt{Integer}$ then $v :_\omega \texttt{Real}$
6. If $v_i :_\omega \overline{\alpha} \in \overline{A}_\omega$ for $1 \leq i \leq n$, then $\sigma\texttt{\{}v_1\texttt{, ..., }v_n\texttt{\}} :_\omega \sigma(\overline{\alpha})$
7. If $v :_\omega \sigma(\overline{\alpha})$ then $v :_\omega \texttt{Collection}(\overline{\alpha})$
8. If $v :_\omega \alpha \in A_\omega$ then $v :_\omega \texttt{OclAny}$

where the left-total relation parameter $:_{O_\omega} \subseteq O_\omega \times C_\omega$ denotes the *typing relation* between instances and classifiers.

In particular, by rule (6), we have that if $v :_\omega \tau$ and $\mathit{flatten}_\omega(v) = v_1 \cdots v_n$ then $v_i :_\Omega \mathit{base}(\tau)$ for all $1 \leq i \leq n$. Note, however, that there is no $v \in V_\omega$ with $v :_\omega \texttt{Void}$ and no type $\tau \in T_\omega$ such that $\texttt{Set\{1, Boolean\}} :_\omega \tau$.

We write $\tau \leq_\omega \tau'$ if, and only if $v :_\omega \tau$ implies $v :_\omega \tau'$ for all $v \in V_\omega$. Given a type $\tau \in C_\omega \cup \{\texttt{Void}, \texttt{Boolean}, \texttt{OclType}\}$ we denote the finite set $\{v \in V_\omega \mid v :_\omega \tau\}$ by $\omega(\tau)$; for all other types τ, $\omega(\tau)$ is undefined.

Implementation retrieval. The retrieval of implementations of (overridden) properties, features, pseudo-features, and opposite association ends in a dynamic basis ω is defined by two partial maps yielding *implementations* in

$$I_\omega ::= T_\omega \,.\, \mathit{Name} \equiv (\mathit{A\text{-}Expr} \mid R_\omega) \;\mid\; T_\omega \,.\, \mathit{Name}(\ \mathit{Var}^*\ ) \equiv (\mathit{A\text{-}Expr} \mid R_\omega) : T_\omega \ .$$

Given a name a, a type τ (the annotation), and a value v, the partial function $\mathit{impl}_\omega : \mathit{Name} \times T_\omega \times V_\omega \rightharpoonup I_\omega$ yields, when defined, an implementation $\tau.a \equiv \psi$ with $v :_\omega \tau$, representing the implementation of a structural (pseudo-)feature or an opposite association end with name a, defined in τ, as required by the annotation. If $\psi \in \mathit{A\text{-}Expr} \setminus R_\omega$, then $\tau \in C_\omega$, accounting for class pseudo-features. If $\mathit{impl}_\omega(a, \tau, v)$ is defined, then $\mathit{impl}_\omega(a, \tau, v')$ is defined for all v' such that $v :_\omega \tau'$ implies $v' :_\omega \tau'$, i.e., a is present for all values with the same types as v.

Analogously, given a name o, a type τ (the annotation), a value v, and a sequence of values $(v_i)_{1 \leq i \leq n}$, the partial function $\mathit{impl}_\omega : \mathit{Name} \times T_\omega \times V_\omega \times V_\Omega^* \rightharpoonup I_\omega$ yields, when defined, an implementation $\tau'.o((x_i)_{1 \leq i \leq n}) \equiv \psi : \tau$ with $v :_\omega \tau'$, representing the implementation of a query behavioural (pseudo-)feature or a property with name o, defined with a return type as required by the annotation. If $\psi \in \mathit{A\text{-}Expr} \setminus R_\omega$, then $\tau' \in C_\omega$. If $\mathit{impl}_\omega(o, \tau, v, (v_i)_{1 \leq i \leq n})$ is defined, then $\mathit{impl}_\omega(o, \tau, v', (v_i)_{1 \leq i \leq n})$ is defined for all v' such that $v :_\omega \tau'$ implies $v' :_\omega \tau'$.

Table 5 contains some sample axioms for the retrieval of the implementation of built-in OCL properties. Generally, we write the types for impl_ω as subscripts and omit the types and name for implementations that do not show an annotated expression.

Extensions. We require that a dynamic basis ω be *extendable* by an implementation $\zeta.a \equiv \psi$ with $\zeta \in C_\omega$, if $\mathit{impl}_\omega(a_\zeta, v)$ is undefined for all $v :_\omega \zeta$, and by an implementation $\zeta.o((x_i)_{1 \leq i \leq n}) = \psi : \tau$ if $\mathit{impl}_\omega(o_\tau, v, (v_i)_{1 \leq i \leq n})$ is undefined for all $v :_\omega \zeta$ and all $v_1, \ldots, v_n \in V_\omega$. Such an extension ω' of ω must again

Table 5. Semantics of sample built-in OCL properties

$$impl^{\omega}_{\Omega}(\texttt{=}_{\texttt{Boolean}}, v, v') = (v = v')$$
$$impl^{\omega}_{\Omega}(\texttt{oclIsKindOf}_{\texttt{Boolean}}, v, \tau) = v :_{\omega} \tau$$
$$impl^{\omega}_{\Omega}(\texttt{first}_{\texttt{Sequence}(\overline{\alpha})}, \texttt{Sequence\{}v_1\texttt{, ..., }v_n\texttt{\}}) = v_1$$
$$impl^{\omega}_{\Omega}(\texttt{including}_{\sigma(\overline{\alpha})}, v, v') = make_{\omega}(\sigma, v\ v')$$
$$impl^{\omega}_{\Omega}(\texttt{union}_{\sigma(\overline{\alpha})}, v, v') = make_{\omega}(\sigma, v\ v')$$

be a dynamic basis. For the extension by an implementation $\iota = \zeta.a \equiv \psi$ we require that $impl_{\omega'}(a_{\zeta}, v) = \iota$ for all $v :_{\omega} \zeta$ and that $impl_{\omega'}(a'_{\zeta'}, v)$ is the same as $impl_{\omega}(a'_{\zeta'}, v)$ if $a' \neq a$; and by an implementation $\iota = \zeta.o((x_i)_{1\leq i\leq n}) \equiv \psi : \tau$ that $impl_{\omega'}(o_{\tau}, v, (v_i)_{1\leq i\leq n}) = \iota$ for all $v :_{\omega} \zeta$ and some $v_1, \ldots, v_n \in V_{\omega}$ and that $impl_{\omega'}(o'_{\tau'}, v', (v_i)_{1\leq i\leq n})$ is the same as $impl_{\omega}(o'_{\tau'}, v', (v_i)_{1\leq i\leq n})$ if $o' \neq o$. Moreover, we must have $T_{\omega'} = T_{\omega}$ and $:_{\omega'} = :_{\omega}$.

As the requirements for extensions of static bases, the constraints on extensions of dynamic bases only weakly characterise possible extension mechanisms for implementations. We assume that some scheme of extending dynamic bases is fixed and we write ω, ι for the extension of a dynamic basis ω by the implementation ι according to this scheme.

3.2 Operational Rules

The operational semantics evaluates annotated OCL terms in the context of a dynamic bases and some variable assignments.

A *variable environment* over a dynamic basis ω is a finite sequence γ of variable assignments of the form $x_1 \mapsto v_1, \ldots, x_n \mapsto v_n$ with $x_i \in Var \cup \{\texttt{self}\}$ and $v_i \in V_{\omega}$ for all $1 \leq i \leq n$; we denote $\{x_1, \ldots, x_n\}$ by $\mathrm{dom}(\gamma)$ and v_i by $\gamma(x_i)$ if $x_i \neq x_j$ for all $i < j \leq n$. The empty variable environment is denoted by $\emptyset$, concatenation of variable environments γ and γ' by γ, γ'.

The operational semantics consists of judgements of the form $\omega; \gamma \vdash \tilde{t} \downarrow \rho$ where ω is a dynamic basis, γ is a variable environment over ω, $\tilde{t}$ is an *A-Term*, and $\rho \in R_{\omega} \cup I_{\omega}$. The empty variable environment may be omitted.

The judgement relation $\vdash$ is defined by the rules in Tables 6–7; a rule may only be applied if all its constituents are well-defined. The meta-variables range as follows: $l \in Literal$; $\alpha \in A_{\omega}$, $\zeta \in C_{\omega}$, $\sigma \in S$, $\tau \in T_{\omega}$, $\iota \in I_{\Omega}$; $x \in Var$; $a, o \in Name$; $v \in V_{\omega}$, $\overline{v} \in R_{\omega}$; $\tilde{e} \in A\text{-}Expr$, $\tilde{d} \in A\text{-}Decl$, $\tilde{p} \in A\text{-}Inv$.

We additionally adopt the following general *strictness convention* that applies to all rules with the single exception of the rules ($\mathrm{And}^{\downarrow}_1$–$\mathrm{And}^{\downarrow}_3$) and ($\mathrm{Or}^{\downarrow}_1$–$\mathrm{Or}^{\downarrow}_3$) in Table 6: if $\bot$ occurs as a result in a judgement of a premise of some rule, the whole term evaluates to $\bot$.

The operational rules are presented in close correspondence to the typing rules in Tables 3–4. All rules, except the rules ($\mathrm{And}^{\downarrow}_2$–$\mathrm{And}^{\downarrow}_3$) and ($\mathrm{Or}^{\downarrow}_2$–$\mathrm{Or}^{\downarrow}_3$) in Table 6 require of all sub-terms to be fully evaluated and to result in a value

Table 6. Operational semantics I

$$\text{(Ctxt}^{\downarrow}\text{)}\quad \frac{(\omega;\gamma,\texttt{self}\mapsto v\vdash \tilde{p}_i\downarrow v_{i,v})_{1\le i\le n,v\in\omega(\zeta)}}{\omega;\gamma\vdash \texttt{context}\ \zeta\ (\texttt{inv:}\ \tilde{p}_i)_{1\le i\le n}\downarrow \bigwedge_{i,v} v_{i,v}}$$

$$\text{(Inv}^{\downarrow}\text{)}\quad \frac{(\omega;\gamma\vdash \tilde{d}_i\downarrow \iota_i)_{1\le i\le n}\quad \omega,(\iota_i)_{1\le i\le n};\gamma\vdash \tilde{e}\downarrow v}{\omega;\gamma\vdash (d_i)_{1\le i\le n}\ \texttt{in}\ \tilde{e}\downarrow v}$$

$$\text{(Decl}_1^{\downarrow}\text{)}\quad \omega;\gamma\vdash \texttt{let}\ x_\zeta\ \texttt{:}\ \tau\ \texttt{=}\ \tilde{e}\downarrow \zeta.x=\tilde{e}$$

$$\text{(Decl}_2^{\downarrow}\text{)}\quad \omega;\gamma\vdash \texttt{let}\ x_\zeta(x_1\ \texttt{:}\ \tau_1\texttt{,}\ \ldots\texttt{,}\ x_n\ \texttt{:}\ \tau_n)\ \texttt{:}\ \tau\ \texttt{=}\ \tilde{e}\downarrow \zeta.x((x_i)_{1\le i\le n})=\tilde{e}:\tau$$

$$\text{(Lit}^{\downarrow}\text{)}\quad \omega;\gamma\vdash l\downarrow l \qquad \text{(Self}^{\downarrow}\text{)}\quad \omega;\gamma\vdash \texttt{self}\downarrow \gamma(\texttt{self})$$

$$\text{(Var}^{\downarrow}\text{)}\quad \omega;\gamma\vdash x\downarrow \gamma(x) \qquad \text{(Type}^{\downarrow}\text{)}\quad \omega;\gamma\vdash \tau\downarrow \tau$$

$$\text{(Coll}^{\downarrow}\text{)}\quad \frac{(\omega;\gamma\vdash \tilde{e}_i\downarrow v_i)_{1\le i\le n}}{\omega;\gamma\vdash \sigma\{\tilde{e}_1\texttt{,}\ \ldots\texttt{,}\ \tilde{e}_n\}\downarrow make_\omega(\sigma,v_1\cdots v_n)}$$

$$\text{(Cond}^{\downarrow}\text{)}\quad \frac{\omega;\gamma\vdash \tilde{e}\downarrow v\quad (\omega;\gamma\vdash \tilde{e}_i\downarrow v_i)_{1\le i\le 2}}{\omega;\gamma\vdash \texttt{if}\ \tilde{e}\ \texttt{then}\ \tilde{e}_1\ \texttt{else}\ \tilde{e}_2\ \texttt{endif}\downarrow v'}\quad \begin{array}{l}\text{if } v=\texttt{true} \text{ and } v'=v_1 \text{ or}\\ v=\texttt{false} \text{ and } v'=v_2\end{array}$$

$$\text{(Iter}^{\downarrow}\text{)}\quad \frac{\omega;\gamma\vdash \tilde{e}\downarrow v\qquad \omega;\gamma\vdash \tilde{e}'\downarrow v_0'\quad (\omega;\gamma,x\mapsto v_i,x'\mapsto v_{i-1}'\vdash \tilde{e}''\downarrow v_i')_{1\le i\le n}}{\omega;\gamma\vdash \tilde{e}\texttt{->iterate(}x\ \texttt{:}\ \alpha\texttt{;}\ x'\ \texttt{:}\ \tau\ \texttt{=}\ \tilde{e}'\ \texttt{|}\ \tilde{e}''\texttt{)}\downarrow v_n'}\quad \text{if } v\rightsquigarrow \texttt{Sequence\{}v_1\texttt{,}\ \ldots\texttt{,}\ v_n\texttt{\}}$$

$$\text{(Cast}^{\downarrow}\text{)}\quad \frac{\omega;\gamma\vdash \tilde{e}\downarrow v}{\omega;\gamma\vdash \tilde{e}\texttt{.oclAsType(}\tau\texttt{)}\downarrow \overline{v}}\quad \begin{array}{l}\text{if } v:_\Omega \tau \text{ and } \overline{v}=v \text{ or}\\ v\not:_\Omega \tau \text{ and } \overline{v}=\bot\end{array}$$

$$\text{(Inst}^{\tau}\text{)}\quad \omega;\gamma\vdash \tau\texttt{.allInstances()}\downarrow make_\omega(\texttt{Set},\omega(base(\tau)))$$

$$\text{(And}_1^{\downarrow}\text{)}\quad \frac{(\omega;\gamma\vdash \tilde{e}_i\downarrow v_i)_{1\le i\le 2}}{\omega;\gamma\vdash \tilde{e}_1\texttt{.and(}\tilde{e}_2\texttt{)}\downarrow v_1\wedge v_2} \qquad \text{(Or}_1^{\downarrow}\text{)}\quad \frac{(\omega;\gamma\vdash \tilde{e}_i\downarrow v_i)_{1\le i\le 2}}{\omega;\gamma\vdash \tilde{e}_1\texttt{.or(}\tilde{e}_2\texttt{)}\downarrow v_1\vee v_2}$$

$$\text{(And}_2^{\downarrow}\text{)}\quad \frac{\omega;\gamma\vdash \tilde{e}_i\downarrow \texttt{false}}{\omega;\gamma\vdash \tilde{e}_1\texttt{.and(}\tilde{e}_2\texttt{)}\downarrow \texttt{false}}\quad \text{where } i=1 \text{ or } i=2 \qquad \text{(Or}_2^{\downarrow}\text{)}\quad \frac{\omega;\gamma\vdash \tilde{e}_i\downarrow \texttt{true}}{\omega;\gamma\vdash \tilde{e}_1\texttt{.or(}\tilde{e}_2\texttt{)}\downarrow \texttt{true}}\quad \text{where } i=1 \text{ or } i=2$$

$$\text{(And}_3^{\downarrow}\text{)}\quad \frac{(\omega;\gamma\vdash \tilde{e}_i\downarrow \overline{v}_i)_{1\le i\le 2}}{\omega;\gamma\vdash \tilde{e}_1\texttt{.and(}\tilde{e}_2\texttt{)}\downarrow \bot}\quad \begin{array}{l}\text{if } \overline{v}_1\neq\texttt{false} \text{ and } \overline{v}_2=\bot \text{ or}\\ \overline{v}_1=\bot \text{ and } \overline{v}_2\neq\texttt{false}\end{array} \qquad \text{(Or}_3^{\downarrow}\text{)}\quad \frac{(\omega;\gamma\vdash \tilde{e}_i\downarrow \overline{v}_i)_{1\le i\le 2}}{\omega;\gamma\vdash \tilde{e}_1\texttt{.or(}\tilde{e}_2\texttt{)}\downarrow \bot}\quad \begin{array}{l}\text{if } \overline{v}_1\neq\texttt{true} \text{ and } \overline{v}_2=\bot \text{ or}\\ \overline{v}_1=\bot \text{ and } \overline{v}_2\neq\texttt{true}\end{array}$$

Table 7. Operational semantics II

$$(\text{Feat}_1^{\downarrow})\quad \frac{\omega;\gamma \vdash \tilde{e} \downarrow v}{\omega;\gamma \vdash \tilde{e}.a_\tau \downarrow \overline{v}'} \quad \text{if } \mathit{impl}_\omega(a_\tau, v) = \overline{v}'$$

$$(\text{Feat}_2^{\downarrow})\quad \frac{\omega;\gamma \vdash \tilde{e} \downarrow v \qquad \omega;\gamma, \texttt{self} \mapsto v \vdash \tilde{e}' \downarrow \overline{v}'}{\omega;\gamma \vdash \tilde{e}.a_\tau \downarrow \overline{v}'} \quad \text{if } \mathit{impl}_\omega(a_\tau, v) = \zeta.a \equiv \tilde{e}'$$

$$(\text{Feat}_3^{\downarrow})\quad \frac{\omega;\gamma \vdash \tilde{e} \downarrow v \qquad (\omega;\gamma \vdash \tilde{e}_i \downarrow v_i)_{1\le i\le n}}{\omega;\gamma \vdash \tilde{e}.o_\tau(\tilde{e}_1, \ldots, \tilde{e}_n) \downarrow \overline{v}'} \quad \text{if } \mathit{impl}_\omega(o_\tau, v, (v_i)_{1\le i\le n}) = \overline{v}'$$

$$(\text{Feat}_4^{\downarrow})\quad \frac{\omega;\gamma \vdash \tilde{e} \downarrow v \qquad (\omega;\gamma \vdash \tilde{e}_i \downarrow v_i)_{1\le i\le n} \qquad \omega;\gamma, \texttt{self} \mapsto v, (x_i \mapsto v_i)_{1\le i\le n} \vdash \tilde{e}' \downarrow \overline{v}'}{\omega;\gamma \vdash \tilde{e}.o_\tau(\tilde{e}_1, \ldots, \tilde{e}_n) \downarrow \overline{v}'} \quad \text{if } \mathit{impl}_\omega(o_\tau, v, (v_i)_{1\le i\le n}) = \zeta.o((x_i)_{1\le i\le n}) \equiv \tilde{e}' : \tau$$

$$(\text{Prop}^{\downarrow})\quad \frac{\omega;\gamma \vdash \tilde{e} \downarrow v \qquad (\omega;\gamma \vdash \tilde{e}_i \downarrow v_i)_{1\le i\le n}}{\omega;\gamma \vdash \tilde{e}\texttt{->}o_\tau(\tilde{e}_1, \ldots, \tilde{e}_n) \downarrow \overline{v}'} \quad \text{if } \mathit{impl}_\omega(o_\tau, v, (v_i)_{1\le i\le n}) = \overline{v}'$$

in V_ω in order to deliver a result for a term. In particular, this makes for a strict conditional; on the other hand, the (And$^{\downarrow}$) and (Or$^{\downarrow}$) rules yield parallel `Boolean` properties `and` and `or`; see [12, Sect. 6.4.10] (not treated in [2,14,4]). The only rules that introduce the undefined result $\bot$ are the (Cast$^{\downarrow}$) rule in Table 6 (cf. [12, p. 6-56]) and, possibly, (Feat$_1^{\downarrow}$), (Feat$_3^{\downarrow}$), and (Prop$^{\downarrow}$) in Table 7. The (Iter$^{\downarrow}$) allows for considerable non-determinism if applied to a collection value that is not a sequence (in contrast to [14,16]; not present in [2]).

3.3 Subject Reduction

We define a relation between dynamic and static bases ensuring that, on the one hand, the compile-time and run-time types and type hierarchies are compatible and, on the other hand, that implementations respect declarations. A dynamic basis ω *conforms* to a static basis Ω if $T_\Omega = T_\omega$ and $\le_\Omega = \le_\omega$ and

1. for every $a \in \mathit{Name}$ such that $\mathit{fd}_\Omega(a, \tau) = \tau'.a : \tau''$, $\mathit{impl}_\omega(a_{\tau'}, v)$ is defined for all $v \in V_\omega$ with $v :_\omega \tau$. If $\mathit{impl}_\omega(a_{\tau'}, v)$ is $\tau'.a \equiv v'$ with $v' \in V_\omega$ then $v' :_\omega \tau''$; if $\mathit{impl}_\omega(a_{\tau'}, v)$ is $\tau'.a \equiv \tilde{e}$ then $\Omega; \texttt{self} : \tau' \vdash \tilde{e} : \tau'''$ with $\tau''' \le_\Omega \tau''$.
2. for every $o \in \mathit{Name}$ such that $\mathit{fd}_\Omega(o, \tau, (\tau_i)_{1\le i\le n}) = \tau'.o : (\tau_i')_{1\le i\le n} \to \tau_0'$, $\mathit{impl}_\omega(o_{\tau_0'}, v, (v_i)_{1\le i\le n})$ is defined for all $v, v_1, \ldots, v_n \in V_\omega$ with $v :_\omega \tau$ and $v_i :_\omega \tau_i$ for all $1 \le i \le n$. If $\mathit{impl}_\omega(o_{\tau_0'}, v, (v_i)_{1\le i\le n})$ is $\tau''.o((x_i)_{1\le i\le n}) \equiv v' : \tau_0'$ with $v' \in V_\omega$ then $v' :_\Omega \tau_0'$; if $\mathit{impl}_\omega(o_{\tau_0'}, v, (v_i)_{1\le i\le n})$ is $\tau''.o((x_i)_{1\le i\le n}) \equiv \tilde{e} : \tau_0'$ then $\tau'' \le_\Omega \tau'$ and $\Omega; \texttt{self} : \tau', (x_i : \tau_i')_{1\le i\le n} \vdash \tilde{e} : \tau_0''$ with $\tau_0'' \le_\Omega \tau_0'$.

Even when typing and annotating an OCL term over a static basis and evaluating the annotated term over a dynamic basis that conforms to the static

basis, the operational semantics turns out to be not type sound in the strict sense, i.e., converging well-typed terms do not always yield a result of the expected type. For example,

```
Set{1, 1.2}->iterate(i : OclAny;
                     a : Sequence(OclAny) = Sequence{} |
                     a->including(i))->first.oclAsType(Integer)
```

may evaluate (after annotation) to `1`, if `Set{1, 1.2}` is chosen to be represented by `Sequence{1, 1.2}`; or it may evaluate to $\bot$, if `Set{1, 1.2}` is represented by `Sequence{1.2, 1}`.

However, if the operational semantics reduces an OCL term of inferred type τ to some value then this value is indeed of type τ, i.e., the operational semantics in Sect. 3.2 satisfies the subject reduction property with respect to the type inference system in Sect. 2.2. In order to state and prove this result, we say that a variable environment γ over ω *conforms* to a type environment Γ over Ω if $\mathrm{dom}(\gamma) \supseteq \mathrm{dom}(\Gamma)$ and $\gamma(x) :_\omega \Gamma(x)$ for all $x \in \mathrm{dom}(\gamma)$.

Proposition 2. *Let Ω be a static basis and ω a dynamic basis conforming to Ω; let Γ be a type environment over Ω and γ a variable environment over ω conforming to Γ; let t be a Term and $\tilde{t}$ an A-Term; let $\tau \in T_\Omega$ and $v \in V_\omega$. If $\Omega; \Gamma \vdash t \triangleright \tilde{t} : \tau$ and $\omega; \gamma \vdash \tilde{t} \downarrow v$, then $v :_\omega \tau$.*

Proof. By induction on the proof tree for $\Omega; \Gamma \vdash t \triangleright \tilde{t} : \tau$.

4 Constraint Semantics

The operational semantics, as detailed in the previous section, suggests that an (annotated) OCL constraint $\tilde{c} = \texttt{context}\ \zeta\ (\texttt{inv:}\ \tilde{p})_{1 \le i \le n}$ over a static basis Ω is *satisfied* by a dynamic basis ω conforming to Ω if, and only if $\omega; \vdash \tilde{c} \downarrow \texttt{true}$; and thus, that $\tilde{c}$ is not satisfied by ω if either $\omega; \vdash \tilde{c} \downarrow \texttt{false}$, or $\omega; \vdash \tilde{c} \downarrow \bot$, or when the operational evaluation of $\tilde{c}$ over ω does not terminate.

The possibility of non-termination can be tracked down to the introduction of recursive pseudo-features in OCL 1.4: When the operational evaluation of a constraint $\tilde{c}$ does not involve applications of rules $(\mathrm{Feat}_2^\downarrow)$ or $(\mathrm{Feat}_4^\downarrow)$, the evaluation will always terminate and yield a result. In fact, it may even be shown [3], that *expressions* of either OCL 1.3 or OCL 1.4 over empty UML static structures, that is, where no additional features other than built-in OCL properties are available, represent exactly all primitive recursive functions. Thus, when evaluated operationally, the declaration of pseudo-features increases the expressive power of OCL 1.4 over OCL 1.3 considerably.

Non-termination is illustrated by the acyclicity constraint on the generalisation relationship stated in the introduction: Assuming two classes `A` and `B`, such that `A` is the parent of `B` and, vice versa, `B` is the parent of `A`, the operational evaluation of `allParents` on `A` or `B` will loop. However, declarations may be interpreted differently, when taking the acyclicity constraint to be read as

```
context GeneralizableElement inv:
      self.parent = self.generalization.parent
  and self.allParents = self.parent->union(self.parent.allParents)
  and not self.allParents->includes(self)
```

over an extended static and dynamic basis, where `GeneralizableElement` shows the features `parent` and `allParents`. This reading would only require *constraints* on the implementation of `parent` and `allParents`. For the cyclic generalisation relation above `parent` of `A` must yield `B`, `parent` of `B` must yield `A`; the implementations of `allParents` for `A` and `B` are only required to result in a fix-point, e.g., both could yield `Set{A, B}` or both could yield $\bot$.

More generally, we call a dynamic basis ω a *result dynamic basis* if all implementation retrieval functions $impl_\omega$ yield only implementations showing a result in R_ω. Given an annotated declaration of a structural pseudo-feature `let` a_ζ `:` τ `=` $\tilde{e}$ we say that a result dynamic basis ω has a *fix-point* for a_ζ if

$$\omega; \texttt{self} \mapsto v \vdash \tilde{e} \downarrow impl_\omega(a_\zeta, v)$$

for all $v \in \omega(\zeta)$; and likewise for an annotated declaration of a query behavioural pseudo-feature `let` o_ζ`(`x_1 `:` τ_1`, ...,` x_n `:` τ_n`)` `:` τ `=` $\tilde{e}$.

The *constraint semantics* interprets a constraint over result dynamic bases showing fix-points for all declarations occurring in the given constraint: Let $\tilde{c} =$ `context` ζ `inv:` $(\tilde{d}_i)_{1 \le i \le n}$ `in` $\tilde{e}$ be a constraint annotated and typed over a static basis Ω such that $\Omega'; \texttt{self} : \zeta \vdash \tilde{d}_i : \delta_i$ for all $1 \le i \le n$ for the static basis Ω' extending Ω. Then $\tilde{c}$ *holds* in a dynamic basis ω conforming to Ω with respect to the constraint semantics if, and only if $\omega'; \texttt{self} \mapsto v \vdash \tilde{e} \downarrow \texttt{true}$ for all result dynamic bases ω' conforming to Ω', which extend ω and show fix-points for all declarations $\tilde{d}_i$ with $1 \le i \le n$, and all $v \in \omega'(\zeta)$.

This constraint semantics employs all fix-point dynamic bases; it may be desirable to restrict attention only to *least* fix-points.

5 Conclusions

We have presented a type inference system and a big-step operational semantics for the OCL 1.4 including the possibility of declaring additional pseudo-features; the operational semantics satisfies a subject reduction with respect to the type inference system. The corrections and additions to previous formal approaches to OCL 1.1/3 are pervasive. We have also discussed an alternative, non-operational interpretation of the declaration of pseudo-features as model constraints.

On the one hand, the semantics may form a new, more comprehensive basis for the treatment of OCL pre- and post-conditions, cf. Richters and Gogolla [16]; global pseudo-feature declarations using the `def:` stereotype may be easily incorporated. On the other hand, we have abstractly axiomatised UML static structures and UML object models, stating only some sufficient conditions such that OCL terms can be typed uniquely and evaluated type-safely. In particular, we have not treated the more complex UML template types, which have been in-

vestigated by Clark [4] though making additional assumptions on the inheritance relationship and contra-variance. However, this axiomatisation may contribute to the necessary clarification of the overall UML type system.

Acknowledgements. We thank Hubert Baumeister for pointing out the constraint interpretation of `let`-declarations and careful proof-reading.

References

1. T. Baar and R. Hähnle. An Integrated Metamodel for OCL Types. In R. France, editor, *Proc. OOPSLA'2000 Wsh. Refactoring the UML: In Search of the Core*, Minneapolis, 2000.
2. M. Bickford and D. Guaspari. Lightweight Analysis of UML. Draft NAS1-20335/10, Odyssey Research Assoc., 1998. http://cgi.omg.org/cgi-bin/doc?ad/98-10-01.
3. M. V. Cengarle and A. Knapp. On the Expressive Power of Pure OCL. Technical Report 0101, Ludwig–Maximilians–Universität München, 2001.
4. T. Clark. Type Checking UML Static Diagrams. In R. B. France and B. Rumpe, editors, *Proc. 2nd Int. Conf. UML*, volume 1723 of *Lect. Notes Comp. Sci.*, pages 503–517. Springer, Berlin, 1999.
5. S. Drossopoulou and S. Eisenbach. Describing the Semantics of Java and Proving Typing Soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes Comp. Sci.*, pages 41–82. Springer, Berlin, 1999.
6. A. Evans, S. Kent, and B. Selic, editors. *Proc. 3nd Int. Conf. UML*, volume 1939 of *Lect. Notes Comp. Sci.* Springer, Berlin, 2000.
7. A. Hami, J. Howse, and S. Kent. Interpreting the Object Constraint Language. In *Proc. Asia Pacific Conf. Software Engineering*. IEEE Press, 1998.
8. H. Hußmann, B. Demuth, and F. Finger. Modular Architecture for a Toolset Supporting OCL. In Evans et al. [6], pages 278–293.
9. http://www.cs.york.ac.uk/puml/mmf/mmt.zip.
10. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Mass., 1997.
11. J. C. Mitchell. *Foundations for Programming Languages*. Foundations of Computing. MIT Press, Cambridge, Mass.–London, England, 1996.
12. Object Management Group. Unified Modeling Language Specification, Version 1.4. Draft, OMG, 2001. http://cgi.omg.org/cgi-bin/doc?ad/01-02-14.
13. http://www.cs.york.ac.uk/puml/puml-list-archive.
14. M. Richters and M. Gogolla. On Formalizing the UML Object Constraint Language OCL. In T. W. Ling, S. Ram, and M. L. Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling*, volume 1507 of *Lect. Notes Comp. Sci.*, pages 449–464. Springer, Berlin, 1998.
15. M. Richters and M. Gogolla. Validating UML Models and OCL Constraints. In Evans et al. [6], pages 265–277.
16. M. Richters and M. Gogolla. OCL — Syntax, Semantics and Tools. In T. Clark and J. Warmer, editors, *Advances in Object Modelling with the OCL*, Lect. Notes Comp. Sci., pages 38–63. Springer, Berlin, 2001.
17. A. Schürr. New Type Checking Rules for OCL (Collection) Expressions. In T. Clark and J. Warmer, editors, *Proc. UML'2000 Wsh. UML 2.0 — The Future of OCL*, York, 2000.