OCL 1.4/5 vs. 2.0 Expressions
Formal semantics and expressiveness

María Victoria Cengarle[1,*], **Alexander Knapp**[2]

[1] Fraunhofer Institut, Experimental Software Engineering
[2] Ludwig-Maximilians-Universität München, Germany; E-mail: knapp@informatik.uni-muenchen.de

Abstract. A type inference system and a big-step operational semantics for expressions of the "Object Constraint Language" (OCL), the declarative and navigational constraint language for the "Unified Modeling Language" (UML), are provided; the account is mainly based on OCL 1.4/5, but also includes the main features of OCL 2.0. The formal systems are parameterised in terms of UML static structures and UML object models, which are treated abstractly. It is proved that the operational semantics satisfies a subject reduction property with respect to the type inference system. Proceeding from the operational semantics and providing a denotational semantics, pure OCL 2.0 expressions are shown to exactly represent the primitive recursive functions, whereas pure OCL 1.4/5 expressions are Turing complete.

Keywords: OCL – UML – Formal semantics

1 Introduction

The "Object Constraint Language" (OCL [43]) provides a specification language for the definition of constraints and well-formedness requirements, like invariants and pre- and post-conditions, for models of the "Unified Modeling Language" (UML [7]). The OCL has been extensively employed in the specification of the UML metamodel itself throughout UML 1.1 and, moreover, has gained considerable interest in its intended application domain, precise and rigorous software modelling [2, 16]. The OCL, in particular, comprises a navigational expression language which can be used to compute object values in a UML model. This navigational expression language forms the basic building blocks from which the requirements on system states and executions may be constructed. The practical use of the OCL thus depends not to the least degree on a clear understanding of the basic OCL expressions. Moreover, the expressiveness of OCL expressions necessarily delimits the application areas which may be specifiable in a natural way. A clear semantic foundation of the OCL in general represents a sine qua non if we want to put OCL on equal footing with other well-established model specification languages like Z or VDM (see e.g. [25]), which are not directly geared to UML.

The original OCL 1.1 specification [29] showed several weaknesses and vaguenesses, in particular with respect to its expression type system and undefined results in expressions, which were partly remedied and clarified in OCL 1.3 [30]. The OCL 1.4 definition [31], moreover, apparently increased the expressiveness of OCL by adding the possibility of defining auxiliary object attributes and operations via the `let` construct and the `def:` clause, which facilitate the expression of well-formedness rules at modelling time; the OCL 1.3 meaning of `let` expressions as local variable definitions was thereby replaced. The specification remained essentially unchanged in OCL 1.5 [32]. The OCL 2.0 proposal [33], on the one hand, provides a precise metamodel for OCL, partially defines context conditions for parsing a concrete OCL syntax, introduces tuple types, nested collections, and an `isUndef` function, and strives to clarify the type system and the semantics of OCL. On the other hand, however, the OCL 2.0 proposal re-replaces the OCL 1.4/5 `let` construct with its OCL 1.3 meaning, viz. local variable definition, and removes the `def:` clause, suggesting to incorporate auxiliary definitions into the underlying UML model itself by using a stereotype for these attributes and operations. Most noteworthy, the OCL 2.0 proposal for the first time tries to supersede the pragmatic, plain

* Current address: Technische Universität München, Dept. of Computer Science, Boltzmannstraße 3, 95749 Garching, E-mail: cengarle@in.tum.de

English text description of the meaning of OCL terms by including a formal definition of the OCL semantics, though the definition employed, being based on the denotational semantics of Richters and Gogolla [39], still refers to OCL 1.1/3.

Several formal semantics for OCL have already been presented, in particular by Bickford and Guaspari [6], Hamie, Howse, and Kent [20], and Richters and Gogolla [36] for OCL 1.1, by Clark [11], Richters and Gogolla [39], Schmitt [40], Beckert, Keller, and Schmitt [5], and the authors [10] for OCL 1.3, and by the authors [9] for OCL 1.4. With respect to OCL expressions, however, these semantics show deficiencies in handling the OCL types `OclAny` and `OclType` [20, 36], empty collections [11, 36], undefined values [5, 11, 20], non-determinism [6, 20, 36], overridden properties [6, 11, 20, 36], and the `let` construct [6, 11, 20, 36, 39]; the authors' semantics [9, 10] do not cover the new OCL 2.0 features like `undef`, tuples, and nested collections. Also, several implementations up to OCL 1.3 have been provided, most noteworthy the Bremen USE tool [38], the Dresden OCL tool [23], the Karlsruhe KeY-tool [1], and the pUML "Meta-Modelling Tool" (MMT [12]). The OCL constructs covered by these tools vary to some extent and the implementations differ e.g. in their handling of collections, `oclAsType`, undefined values, and the `let` construct. Most importantly, none of these semantics resp. tools considers the features proposed by the OCL 2.0 draft.

In the following, we provide an improved and more comprehensive formal semantics of OCL expressions as well as a comparison of the expressive power of the OCL proposals 1.4/5 and 2.0. The formal semantics is based on [9, 10] and primarily addresses the OCL 2.0 proposal while simultaneously retaining the OCL 1.4 auxiliary definitions for increased modelling expressivity. The semantics is parameterised in UML static structures and object models, which are only assumed to satisfy certain properties, and is thus directly adadptable to different UML interpretations. We concentrate on OCL as a navigational expression language for computing object values in UML models. Relying on well-known terminology, tools, and results from the programming language literature we can classify OCL within the family of programming languages.

The abstract syntax of OCL terms is summarised in Sect. 2. In Sect. 3 we introduce a type inference and annotation system for OCL terms and in Sect. 4 we define a big-step operational semantics that evaluates annotated OCL terms. The operational semantics satisfies a subject reduction property with respect to the type inference system, i.e., evaluation returns values of the expected type. Proceeding from the operational semantics, in Sect. 5 we provide a denotational semantics for OCL expressions and prove that the operational semantics and the denotational semantics coincide on well-typed OCL terms. This denotational semantics is used in Sect. 6 to show that pure OCL expressions without auxiliary definitions (as in OCL 2.0) represent exactly the primitive recursive functions and that auxiliary definitions (as in OCL 1.4/5) increase the expressivity making OCL Turing complete. We conclude with some remarks on using OCL as a logic and future research.

We assume a working knowledge of the OCL syntax and informal semantics as well as of UML and its metamodel.

2 Syntax

The abstract syntax of the OCL sub-language that we consider is given in Table 1. Although the OCL 2.0 proposal [33] provides the OCL with a UML-based metamodel and thus an abstract syntax, we stick to a more conventional presentation as a BNF-grammar, mimicking the OCL 1.5 grammar [32].

An OCL specification in *Spec* consists of optional pseudofeature definitions in *Def*, i.e. attributes and operations, and an invariant constraint in *Constr*. As in OCL 1.5 and deviating from OCL 2.0, pseudofeature definitions provide auxiliary functions for OCL expressions for enhanced modelling expressiveness; like invariants, pseudofeatures are defined in the context of a UML classifier. Without loss of generality, we only admit a single definition or constraint per context. In addition to the OCL expressions in *Expr* already present in OCL 1.4 and following the OCL 2.0 proposal, we introduce an explicit `undef` symbol, representing failing computations, a corresponding function symbol `isUndef` for testing whether a given expression results in `undef`, and tuple expressions and types that can be regarded as records. However, we retain a type `Type`, the type of all types (abbreviating `OclType`, which has been removed in OCL 2.0), and define a new type `Void`, the empty type, which afford a consistent typing system. Moreover, we abbreviate `OclAny` to `Any`. The boolean connectives `and` and `or` are singled out, as they show "parallel" semantical behaviour different from the other OCL properties [32, p. 6–11]. The OCL built-ins `allInstances` and `asType` (abbreviating `oclAsType`) do not apply to arbitrary expressions but only to types, leading to a simpler type system as in OCL 2.0.

The OCL sub-language in Table 1 leaves out some syntactic sugar like names for invariants, the `let` for pseudofeature definitions, the functions that can be defined in terms of `iterate` [11], pre- and in-fix notation, and comments. More importantly, we do not treat template types, navigation to association classes and through qualified associations, and package pathnames as primitives. This decision is justified as follows. Firstly, although the UML 1.4/5 specification only considers models with fully instantiated templates [31], OCL constraints can also be written for uninstantiated templates as the template type as well as its formal parameters are model elements. Secondly, for association classes, as described in the OCL 1.5 specification [32, pp. 6–15f.], the name of an association

Table 1. OCL abstract syntax

Term	::= *Spec* | *Constr* | *Expr*
Spec	::= {*Def*} *Constr*
Def	::= `context` *Type* `def:` (*AttrDef* | *OpDef*)
Constr	::= `context` *Type* `inv:` *Expr*
AttrDef	::= *Name* `:` *Type* `=` *Expr*
OpDef	::= *Name* `(` [*Var* `:` *Type* {`,` *Var* `:` *Type*}] `)` `:` *Type* `=` *Expr*
Expr	::= *Literal* | `self` | *Var* | *Type* | `undef` |
	(`Set` | `Bag` | `Sequence`) `{` [*Expr* {`,` *Expr*}] `}` |
	`Tuple` `{` *Name* `=` *Expr* {`,` *Name* `=` *Expr*} `}` |
	`let` *Var* [`:` *Type*] `=` *Expr* `in` *Expr* |
	`if` *Expr* `then` *Expr* `else` *Expr* `endif` |
	Expr `and` *Expr* | *Expr* `or` *Expr* |
	Expr `.` `asType` `(` *Type* `)` |
	Type `.` `allInstances` `(` `)` |
	Expr `.` `isUndef` `(` `)` |
	Expr `->` `iterate` `(` *Var* [`:` *Type*] `;` *Var* [`:` *Type*] `=` *Expr* `|` *Expr* `)` |
	Expr `.` *Name* |
	Expr `.` *Name* `(` [*Expr* {`,` *Expr*}] `)` |
	Expr `->` *Name* `(` [*Expr* {`,` *Expr*}] `)`
Literal	::= *IntegerLiteral* | *RealLiteral* | *BooleanLiteral* | *StringLiteral*
Type	::= *Name* | `Void` | `Integer` | `Real` | `Boolean` | `String` | `Any` | `Type` |
	(`Set` | `Bag` | `Sequence` | `Collection`) `(` *Type* `)` |
	`Tuple` `(` *Name* `:` *Type* {`,` *Name* `:` *Type*} `)`
Var	::= *Name*

class with an initial lower-case letter can either be used like an additional structural feature for navigation or, if the navigation direction is not clear, this name has to be extended by the role name of the opposite association end through which the association class is to be reached; this is considered as a simple naming convention. Thirdly, qualified association ends [32, pp. 6–16f.] are taken to be abbreviations for association classes showing the qualifiers as structural features [19]. Finally, pathnames involving packages are taken to be names conforming to an additional naming convention.

We omit the types `OclExpression` (which has also been discarded in OCL 2.0) and `OclState` and pre- and post-conditions thus concentrating on OCL as a constraint navigational expression language.

3 Type system

The type of an OCL term, and thus its well-formedness, depends on information from an underlying UML static structure. In particular, we need to take into account classifiers like classes, structural and query behavioural features like attributes resp. operations, the generalisation (or inheritance) relationship, opposite association ends, association classes, etc., and the built-in OCL types and properties. We abstractly axiomatise this information as static bases. These are parametric in the classifiers and the generalisation relationship and provide an extension mechanism by pseudofeatures. This axiomatisation also captures the declaration retrieval of (overloaded and overridden) features and properties that is only vaguely described in the UML specification by full-descriptors [31, p. 2–75]. We present a type inference system for OCL terms over such a static basis. This system annotates the terms for later evaluation of overloaded or overridden features and properties and of pseudofeatures. We prove that the type system entails unique annotations and types.

The type inference system and the axiomatisation of static bases generalise Clark's definition [11]. A static basis corresponds to Clark's static models, but does not enforce a contra/covariant overriding scheme of behavioural features and allows for the redefinition of structural features. Moreover, in contrast to Clark's approach [11, Thm. 6] and the typing system by Richters and Gogolla [39], the type inference system entails unique types. Richters and Gogolla base the OCL typing system on more explicit algebraic signatures for capturing the static properties of a UML model.

3.1 Static bases

A *static basis* Ω defines types, a type hierarchy, functions for declaration retrieval, and an extension mechanism for declarations. The types and the type hierarchy are the ones defined in the underlying UML static structure. The declaration retrieval is necessary for typing an OCL expression that uses names of the UML static structure. The purpose of the extension mechanism is to extend the underlying UML static structure with the declarations

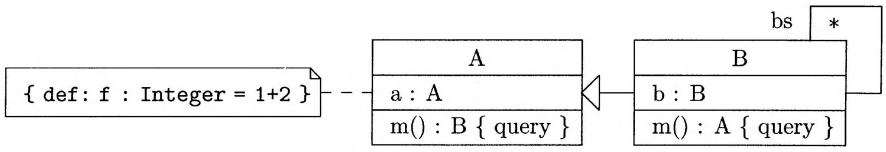

Fig. 1. Sample class diagram

local to OCL specifications, and in this way to allow a uniform typing mechanism.

Let us illustrate the main characteristics of static bases by means of a simple example. Consider the UML static structure diagram of Fig. 1. The diagram induces the static basis Ω with classifier types $C_\Omega = \{\mathtt{A}, \mathtt{B}\}$ and the inheritance relation $\mathtt{B} \leq_{C_\Omega} \mathtt{A}$. The declared features are retrieved by the full-descriptor function fd_Ω. Given a feature name, a class, and possibly a list of argument types, fd_Ω searches for the feature declaration closest to the class in the inheritance hierarchy. In particular, $fd_\Omega(\mathtt{a}, \mathtt{B}) = \mathtt{A.a} : \mathtt{A}$ although `a` is not a feature declared in `B`, since `a` is inherited by `B` from `A` and has type `A`. Moreover,

$fd_\Omega(\mathtt{b}, \mathtt{B}) = \mathtt{B.b} : \mathtt{B}$
$fd_\Omega(\mathtt{bs}, \mathtt{B}) = \mathtt{B.bs} : \mathtt{Set(B)}$
$fd_\Omega(\mathtt{m}, \mathtt{A}, ()) = \mathtt{A.m} : () \to \mathtt{B}$
$fd_\Omega(\mathtt{m}, \mathtt{B}, ()) = \mathtt{B.m} : () \to \mathtt{A}$

and $fd_\Omega(\mathtt{b}, \mathtt{A})$ is undefined. Opposite association ends showing the annotation {ordered} are captured by defining the full-descriptor to yield a feature declaration with resulting collection type `Sequence(...)`, instead of `Set(...)` as for `bs`.

Taking the constraint in the note into account, the static basis Ω is extended by the declaration $\delta = \mathtt{A.f} : \mathtt{Integer}$ and thus, in particular, the declarations of the extended static basis Ω, δ contain

$fd_{\Omega,\delta}(\mathtt{f}, \mathtt{B}) = \mathtt{A.f} : \mathtt{Integer}$.

In the following, we formally define the tools used in this example, namely types, type hierarchy, declaration retrieval, and extensions of static bases.

Types. The *(compile-time) types* of OCL include the *classifier types* of the underlying UML model and the OCL built-in types. The OCL built-in types consist of: the set B of *basic* types like `Integer`, the collection types like `Set(...)`, tuple types, and the distinguished types `Any`, `Void`, and `Type`. Formally, let Ω be a static basis and C_Ω be the classifier types of Ω. Then, the set T_Ω of OCL (compile-time) types is defined as follows:

$T_\Omega ::= U_\Omega \mid \overline{S}\,(\,T_\Omega\,)$
$U_\Omega ::= A_\Omega \mid \mathtt{Type} \mid \mathtt{Tuple}\,(\,\mathit{Name} : T_\Omega\,\{\,,\,\mathit{Name} : T_\Omega\}\,)$
$A_\Omega ::= \mathtt{Void} \mid B \mid C_\Omega \mid \mathtt{Any}$
$\overline{S} ::= S \mid \mathtt{Collection}$
$S ::= \mathtt{Set} \mid \mathtt{Bag} \mid \mathtt{Sequence}$
$B ::= \mathtt{Integer} \mid \mathtt{Real} \mid \mathtt{Boolean} \mid \mathtt{String}$

We require that the names in the finite set parameter C_Ω are different from the other type names in the grammar above. 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*.

The type `Void` is not required by the OCL specification; it is subtype of all types and denotes the *empty type*, that will be used for typing both empty collections and the undefined value. `Any` is the common supertype of all basic and classifier types. The *tuple types* are defined by the type constructor `Tuple`; all names in a type `Tuple(...)` have to be distinct, their order is immaterial. The type `Type` is the type of all types (as used in impredicative polymorphism [28]).

Each *Literal* l (see Table 1) has a type, written as $\mathit{type}(l)$, such that $\mathit{type}(n) = \mathtt{Integer}$ if n is an *IntegerLiteral*, etc.

Type hierarchy. The type hierarchy of OCL includes the generalisation (or inheritance) relationship of the underlying UML model and puts also the OCL built-in types in relation. Let Ω be a static basis, C_Ω be its classifiers, and $\leq_{C_\Omega}$ denote the *generalisation hierarchy* on the classifier types C_Ω. The *subtype relation* $\leq_\Omega$ of a static basis Ω is formally defined as the least partial order that satisfies the following axioms:

1. for all $\tau \in T_\Omega$, $\mathtt{Void} \leq_\Omega \tau$
2. for all $\alpha \in A_\Omega$, $\alpha \leq_\Omega \mathtt{Any}$
3. $\mathtt{Integer} \leq_\Omega \mathtt{Real}$
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 $\tau_1, \ldots, \tau_n \in T_\Omega$, $\tau'_1, \ldots, \tau'_n \in T_\Omega$ and $a_1, \ldots, a_n \in \mathit{Name}$, if $\tau_i \leq_\Omega \tau'_i$, $1 \leq i \leq n$, then $\mathtt{Tuple}(a_1 : \tau_1, \ldots, a_n : \tau_n) \leq_\Omega \mathtt{Tuple}(a_1 : \tau'_1, \ldots, a_n : \tau'_n)$
6. for all $\tau_1, \ldots, \tau_n, \tau_{n+1} \in T_\Omega$ and $a_1, \ldots, a_n, a_{n+1} \in \mathit{Name}$, $\mathtt{Tuple}(a_1 : \tau_1, \ldots, a_n : \tau_n, a_{n+1} : \tau_{n+1}) \leq_\Omega \mathtt{Tuple}(a_1 : \tau_1, \ldots, a_n : \tau_n)$
7. for all $\sigma \in S$ and $\tau \in T_\Omega$, $\sigma(\tau) \leq_\Omega \mathtt{Collection}(\tau)$
8. for all $\overline{\sigma} \in \overline{S}$ and $\tau_1, \tau_2 \in T_\Omega$, if $\tau_1 \leq_\Omega \tau_2$, then $\overline{\sigma}(\tau_1) \leq_\Omega \overline{\sigma}(\tau_2)$

In words, $\leq_\Omega$ includes $\leq_{C_\Omega}$, has `Void` as minimum, `Any` as maximum of "simple" types (i.e., types excluding collections, tuples and the type `Type` of types), and the intuitive relationship among basic types and among collections.

On the one hand and according to OCL 1.5 [32, p. 6–7], collection types are basic types; on the other and following OCL 1.5 [32, pp. 6–29f.] and OCL 2.0 [33, p. 6–1], these types are *not* basic types (cf. also [4]). We choose the second definition (in contrast to [36]) and define $\sigma(\tau) \not\leq_\Omega \texttt{Any}$, in order to avoid the Russell paradox that could arise from $\texttt{Set(Any)} \leq_\Omega \texttt{Any}$ (see [6]), in this way allowing a naïve set interpretation of types. As a consequence, none of the properties of `Any`, like inequality or `oclIsKindOf`, is immediately available for collections. Note also that $\texttt{Type} \not\leq_\Omega \texttt{Any}$, in contrast to [12], accounting for a clearer separation between sorts and kinds [28].

The subtyping rules for `Tuple` are modelled on named products [28]: Subtypes may constrain the types of the components and may show additional components. Note that tuple types showing the same components but in different order are equal.

As a notational convention, if the least upper bound of types $\tau_1, \ldots, \tau_n$ with respect to $\leq_\Omega$ exists, we denote it by $\bigsqcup_\Omega\{\tau_1, \ldots, \tau_n\}$. For example, $\bigsqcup_\Omega\{\texttt{Integer}, \texttt{Real}\} = \texttt{Real}$ and $\bigsqcup_\Omega\{\texttt{Integer}, \texttt{Boolean}\} = \texttt{Any}$, but `Integer` and `Set(Integer)` have no least upper bound. In the presence of multiple inheritance, least upper bounds may also fail to exist for sets of classifier types. Note that $\bigsqcup_\Omega \emptyset = \texttt{Void}$.

Declaration retrieval. The typing of OCL expressions relies on information of features in the underlying UML model as well as on the predefined OCL properties. A mechanism for fetching feature declarations is therefore necessary. A *declaration* describes the signature of an existing feature, i.e. information on the location of a feature in the type hierarchy, possibly the type of its parameters, and its return type. Given a static basis Ω, the declarations D_Ω in Ω read as follows:

$$\begin{aligned} D_\Omega ::= {} & T_\Omega \,.\, \mathit{Name} : T_\Omega \mid \\ & T_\Omega \,.\, \mathit{Name} : T_\Omega^* \to T_\Omega \,. \end{aligned}$$

The retrieval of (overloaded or overridden) properties, features, pseudofeatures, opposite association ends, and association classes in a static basis Ω is defined by two suitably axiomatised maps. These maps represent the search for a type showing the desired feature in the generalisation hierarchy above and including a given type.

For attribute-like features, 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, an opposite association end, or an association class 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 by all subtypes of τ.

In the example of Fig. 1 and considering the diagram without the constraint note,

$\mathit{fd}_\Omega(\texttt{a}, \texttt{A}) = \texttt{A.a} : \texttt{A}$,
$\mathit{fd}_\Omega(\texttt{a}, \texttt{B}) = \texttt{A.a} : \texttt{A}$,
$\mathit{fd}_\Omega(\texttt{b}, \texttt{B}) = \texttt{B.b} : \texttt{B}$,
$\mathit{fd}_\Omega(\texttt{bs}, \texttt{B}) = \texttt{B.bs} : \texttt{Set(B)}$, and
$\mathit{fd}_\Omega(\texttt{b}, \texttt{A})$ is undefined.

For behavioural features, 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 also $\mathit{fd}_\Omega(o, \tau', (\tau_i')_{1 \leq i \leq n})$ is defined for all $\tau' \leq_\Omega \tau$ and $\tau_i' \leq \tau_i$ for all $1 \leq i \leq n$, i.e., again, o is inherited by all subtypes of τ.

In the example of Fig. 1 and considering the diagram without the constraint note,

$\mathit{fd}_\Omega(\texttt{m}, \texttt{A}, ()) = \texttt{A.m} : () \to \texttt{B}$ and
$\mathit{fd}_\Omega(\texttt{m}, \texttt{B}, ()) = \texttt{B.m} : () \to \texttt{A}$.

Static bases also provide the declaration retrieval of the predefined OCL properties [32, Sect. 6.8]. Table 2 shows a non-exhaustive list of axioms for OCL properties, where $\tau, \tau' \in T_\Omega$ and $a \in \mathit{Name}$. Note that projections of tuples are treated like properties.

Extensions. A constraint may contain definitions of auxiliary features by means of the `def:` clause [32, Sect. 6.8]. These additional features extend the context within which the constraint is to be typed. We thus require that static bases be *extendable* by new declarations. We only put mild requirements on the chosen extension mechanism.

A static basis Ω can be extended by a declaration of a structural pseudofeature $\tau.a : \tau'$ with $\tau, \tau' \in T_\Omega$ if $\mathit{fd}_\Omega(a, \tau)$ is undefined (i.e., if the attribute is indeed new). A static basis Ω can also be extended by a declaration of a behavioural pseudofeature $\tau.o : (\tau_i)_{1 \leq i \leq n} \to \tau_0$ with $\tau, \tau_0, \tau_1, \ldots, \tau_n \in T_\Omega$ if $\mathit{fd}_\Omega(o, \tau, (\tau_i)_{1 \leq i \leq n})$ is undefined (i.e., if the operation is in fact new). The result Ω' of such an extension of Ω must again be a static basis.

If the extension consists in a declaration $\delta = \tau.a : \tau'$, we require that $\mathit{fd}_{\Omega'}(a, \tau) = \delta$ and that $\mathit{fd}_{\Omega'}$ is the same as before for a previously existing a', that is, $\mathit{fd}_{\Omega'}(a', \tau') = \mathit{fd}_\Omega(a', \tau')$ if $a' \neq a$. Analogously, for the extension by a declaration $\delta = \tau.o : (\tau_i)_{1 \leq i \leq n} \to \tau_0$ we require that $\mathit{fd}_{\Omega'}(o, \tau, (\tau_i)_{1 \leq i \leq n}) = \delta$ and that $\mathit{fd}_{\Omega'}(o', \tau', (\tau_i')_{1 \leq i \leq n})$

Table 2. Typing of sample built-in OCL properties

$fd_\Omega(a, \texttt{Tuple}(a : \tau)) = \texttt{Tuple}(a : \tau).a : \tau$
$fd_\Omega(\texttt{=}, \tau, \tau') = \tau.\texttt{=} : \tau' \to \texttt{Boolean}$
$fd_\Omega(\texttt{oclIsKindOf}, \alpha, \texttt{Type}) = \alpha.\texttt{oclIsKindOf} : \texttt{Type} \to \texttt{Boolean}$
$fd_\Omega(\texttt{first}, \texttt{Sequence}(\tau)) = \texttt{Sequence}(\tau).\texttt{first} : \to \tau$
$fd_\Omega(\texttt{including}, \sigma(\tau), \tau') = \sigma(\tau).\texttt{including} : \tau' \to \sigma(\bigsqcup_\Omega\{\tau, \tau'\})$
$fd_\Omega(\texttt{union}, \sigma(\tau), \sigma(\tau')) = \sigma(\tau).\texttt{union} : \sigma(\tau') \to \sigma(\bigsqcup_\Omega\{\tau, \tau'\})$

is the same as $fd_\Omega(o', \tau', (\tau'_i)_{1\le i\le n})$ if $o' \neq o$. Finally, we require that $T_{\Omega'} = T_\Omega$ and $\le_{\Omega'} = \le_\Omega$.

The constraint note in the diagram of Fig. 1 extends the static basis Ω with the declaration $\delta = \texttt{A.f} : \texttt{Integer}$ yielding the static basis Ω' with

$$fd_{\Omega'}(\texttt{f}, \texttt{A}) = \texttt{A.f} : \texttt{Integer}$$
$$fd_{\Omega'}(\texttt{f}, \texttt{B}) = \texttt{A.f} : \texttt{Integer}$$

by the inheritance requirement for static bases, and which leaves Ω unchanged with respect to its types and its type hierarchy.

We assume that some scheme of extending static bases is fixed that observes the above requirements. We let Ω, δ denote the extension of a static basis Ω by the declaration δ according to the chosen scheme.

3.2 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 pseudofeature declarations and overloaded or overridden properties for later evaluation: The declaring type of a structural feature has to be determined statically for accessing overridden attributes [32, Sect. 6.5.9]; we also record the expected return type of a behavioural feature or property call, in order to cope with the ad hoc polymorphism of UML, as illustrated by the overloading of operation `m()` in Fig. 1. The adoption of a contra/covariant overriding scheme for behavioural features would render the recording of return types dispensable [11].

We first define annotated OCL terms by means of a grammar and then introduce the rules that allow to infer the type and the annotation of a given term using the type and annotation of the terms that form part of the term of interest. Finally, we draw a detailed comparison of our approach with the typing systems in the OCL literature.

The grammar for *annotated* OCL terms transforms the grammar in Table 1 by consistently replacing *Term*, *Spec*, *Def*, *Constr*, *AttrDef*, *OpDef*, and *Expr* by *A-Term*, *A-Spec*, *A-Constr*, *A-AttrDef*, *A-OpDef*, and *A-Expr*, respectively. Furthermore, the original clauses

AttrDef ::= *Name* : *Type* = *Expr*
OpDef ::= *Name* ([*Var* : *Type* { , *Var* : *Type* }]) : *Type* = *Expr*
Expr ::= ⋯ |
`let` *Var* [: *Type*] = *Expr* `in` *Expr* |
Expr `->` `iterate` (*Var* [: *Type*] ; *Var* [: *Type*] = *Expr* | *Expr*) |
Expr . *Name* |
Expr . *Name* ([*Expr* { , *Expr* }]) |
Expr `->` *Name* ([*Expr* { , *Expr* }])

are replaced by

A-AttrDef ::= *Name Type* : *Type* = *A-Expr*
A-OpDef ::= *Name Type* ([*Var* : *Type* { , *Var* : *Type* }]) : *Type* = *A-Expr*
A-Expr ::= ⋯ |
`let` *Var* = *A-Expr* `in` *A-Expr* |
A-Expr `->` `iterate` (*Var* ; *Var* = *A-Expr* | *A-Expr*) |
A-Expr . *Name Type* |
A-Expr . *Name Type* ([*A-Expr* { , *A-Expr* }]) |
A-Expr `->` *Name Type* ([*A-Expr* { , *A-Expr* }])

The annotations by a *Type* for the definition of attributes and of operations as well as for using a structural feature record the defining class. The *Type* annotations for (overloaded or overridden) behavioural feature retrieval keep track of the expected return type in order to deal with ad hoc polymorphism. Annotations are written as subscripts. The optional type assertions for `let` and `iterate` are omitted, as suitable types can be inferred.

A typing judgement of the type inference system puts in relation a static basis, a type environment, the (unannotated) term of interest, an annotated term, and a type. These judgements are to be interpreted as follows: On the basis of the underlying UML model and in the context of the given type environment, the (unannotated) term can be annotated as expressed by the annotated term and has the given type. The type environment is used to trace the types of subexpressions in the context of the unannotated term. The annotation is used to deal with inheritance

and overloading, in order to type and later to evaluate correctly the term of interest. The annotated term simultaneously represents the normal form of the unannotated term and thus simplifies the rules of evaluation.

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 $1 \leq i \leq n$; 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 $i < j \leq n$. The empty type environment is denoted by $\emptyset$, concatenation of type environments Γ and Γ' by Γ, Γ'.

The type inference system derives *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 θ is a type in T_Ω or a declaration in D_Ω. When writing such a judgement, we assume that `self`, `undef`, `isUndef`, `asType`, `allInstances`, `and`, and `or` are reserved names and that *Var* and $T_\Omega \subseteq \mathit{Type}$ are disjoint. The empty type environment may be omitted.

Judgements are derived using the rules in Tables 3–4. A rule may only be applied if its premises and its conclusion as well as its side conditions (in particular ap-

Table 3. Type inference system I

$$\text{(Spec}^{\cdot}\text{)}\quad \frac{(\Omega, (\delta_j)_{1\leq j\leq n}; \Gamma, \texttt{self} : \zeta_i \vdash d_i \triangleright \tilde{d_i} : \delta_i)_{1\leq i\leq n} \qquad \Omega, (\delta_j)_{1\leq j\leq n}; \Gamma, \texttt{self} : \zeta \vdash e \triangleright \tilde{e} : \texttt{Boolean}}{\begin{array}{l}\Omega; \Gamma \vdash (\texttt{context}\ \zeta_i\ \texttt{def:}\ d_i)_{1\leq i\leq n}\ \texttt{context}\ \zeta\ \texttt{inv:}\ e \triangleright \\ \quad (\texttt{context}\ \zeta_i\ \texttt{def:}\ \tilde{d_i})_{1\leq i\leq n}\ \texttt{context}\ \zeta\ \texttt{inv:}\ \tilde{e} : \texttt{Boolean}\end{array}}$$

$$\text{(Def}^{\cdot}_1\text{)}\quad \frac{\Omega; \Gamma \vdash e \triangleright \tilde{e} : \tau'}{\begin{array}{l}\Omega; \Gamma \vdash a\ \texttt{:}\ \tau\ \texttt{=}\ e \triangleright a_\zeta\ \texttt{:}\ \tau\ \texttt{=}\ \tilde{e} : (\zeta.a : \tau) \\ \quad \text{where } \zeta = \Gamma(\texttt{self}) \text{ and if } \tau' \leq_\Omega \tau\end{array}}$$

$$\text{(Def}^{\cdot}_2\text{)}\quad \frac{\Omega; \Gamma, (x_i : \tau_i)_{1\leq i\leq n} \vdash e \triangleright \tilde{e} : \tau'}{\begin{array}{l}\Omega; \Gamma \vdash o(x_1\ \texttt{:}\ \tau_1\texttt{,}\ \ldots\texttt{,}\ x_n\ \texttt{:}\ \tau_n)\ \texttt{:}\ \tau\ \texttt{=}\ e \triangleright \\ \quad o_\zeta(x_1\ \texttt{:}\ \tau_1\texttt{,}\ \ldots\texttt{,}\ x_n\ \texttt{:}\ \tau_n)\ \texttt{:}\ \tau\ \texttt{=}\ \tilde{e} : (\zeta.o : (\tau_i)_{1\leq i\leq n} \to \tau) \\ \quad \text{where } \zeta = \Gamma(\texttt{self}) \text{ and if } \tau' \leq_\Omega \tau\end{array}}$$

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

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

$$\text{(Undef}^{\cdot}\text{)}\ \Omega; \Gamma \vdash \texttt{undef} \triangleright \texttt{undef} : \texttt{Void}$$

$$\text{(Coll}^{\cdot}\text{)}\quad \frac{(\Omega; \Gamma \vdash e_i \triangleright \tilde{e}_i : \tau_i)_{1\leq i\leq n}}{\begin{array}{l}\Omega; \Gamma \vdash \sigma\texttt{\{}e_1\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{\}} \triangleright \\ \quad \sigma\texttt{\{}\tilde{e}_1\texttt{,}\ \ldots\texttt{,}\ \tilde{e}_n\texttt{\}} : \sigma(\tau)\end{array}} \quad \text{where } \tau = \bigsqcup_\Omega \{\tau_i \mid 1 \leq i \leq n\}$$

$$\text{(Tup}^{\cdot}\text{)}\quad \frac{(\Omega; \Gamma \vdash e_i \triangleright \tilde{e}_i : \tau_i)_{1\leq i\leq n}}{\begin{array}{l}\Omega; \Gamma \vdash \texttt{Tuple\{}a_1\ \texttt{=}\ e_1\texttt{,}\ \ldots\texttt{,}\ a_n\ \texttt{=}\ e_n\texttt{\}} \triangleright \\ \quad \texttt{Tuple\{}a_1\ \texttt{=}\ \tilde{e}_1\texttt{,}\ \ldots\texttt{,}\ a_n\ \texttt{=}\ \tilde{e}_n\texttt{\}} : \texttt{Tuple(}a_1\ \texttt{:}\ \tau_1\texttt{,}\ \ldots\texttt{,}\ a_n\ \texttt{:}\ \tau_n\texttt{)}\end{array}}$$

$$\text{(Let}^{\cdot}\text{)}\quad \frac{\Omega; \Gamma \vdash e \triangleright \tilde{e} : \tau \qquad \Omega; \Gamma, x : \tau_x \vdash e' \triangleright \tilde{e}' : \tau'}{\Omega; \Gamma \vdash \texttt{let}\ x\ [\texttt{:}\ \tau_x]\ \texttt{=}\ e\ \texttt{in}\ e' \triangleright \texttt{let}\ x\ \texttt{=}\ \tilde{e}\ \texttt{in}\ \tilde{e}' : \tau'} \quad \begin{array}{l}]\text{where } \tau_x = \tau[\\ \text{if } \tau \leq_\Omega \tau_x\end{array}$$

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

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

$$\text{(Iter}^{\cdot}\text{)}\quad \frac{\Omega; \Gamma \vdash e \triangleright \tilde{e} : \overline{\sigma}(\tau) \qquad \Omega; \Gamma \vdash e' \triangleright \tilde{e}' : \tau' \qquad \Omega; \Gamma, x : \tau_x, x' : \tau_{x'} \vdash e'' \triangleright \tilde{e}'' : \tau''}{\begin{array}{l}\Omega; \Gamma \vdash e\texttt{->iterate(}x\ [\texttt{:}\ \tau_x]\texttt{;}\ x'\ [\texttt{:}\ \tau_{x'}]\ \texttt{=}\ e'\ \texttt{|}\ e''\texttt{)} \triangleright \\ \quad \tilde{e}\texttt{->iterate(}x\texttt{;}\ x'\ \texttt{=}\ \tilde{e}'\ \texttt{|}\ \tilde{e}''\texttt{)} : \tau_{x'}\end{array}}$$

$$]\text{where } \tau_x = \tau,\ \tau_{x'} = \tau'[\text{ if } \tau \leq_\Omega \tau_x \text{ and } \tau', \tau'' \leq_\Omega \tau_{x'}$$

Table 4. Type inference system II

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

$$(\text{Inst}^{:}) \quad \Omega;\Gamma \vdash \tau\texttt{.allInstances()} \triangleright \tau\texttt{.allInstances()} : \texttt{Set(}\tau\texttt{)}$$

$$(\text{IsUndef}^{:}) \quad \frac{\Omega;\Gamma \vdash e \triangleright \tilde{e} : \tau}{\Omega;\Gamma \vdash e\texttt{.isUndef()} \triangleright \tilde{e}\texttt{.isUndef()} : \texttt{Boolean}}$$

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

$$(\text{Feat}^{:}_2) \quad \frac{\Omega;\Gamma \vdash e \triangleright \tilde{e} : \upsilon \qquad (\Omega;\Gamma \vdash e_i \triangleright \tilde{e}_i : \tau_i)_{1\leq i\leq n}}{\Omega;\Gamma \vdash e\texttt{.}o\texttt{(}e_1\texttt{, ..., }e_n\texttt{)} \triangleright \tilde{e}\texttt{.}o_{\tau'_0}\texttt{(}\tilde{e}_1\texttt{, ..., }\tilde{e}_n\texttt{)} : \tau'_0} \quad \text{if } fd_\Omega(o,\upsilon,(\tau_i)_{1\leq i\leq n}) = \tau'.o : (\tau'_i)_{1\leq i\leq n} \to \tau'_0$$

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

$$(\text{Sing}^{:}_1) \quad \frac{\Omega;\Gamma \vdash e \triangleright \tilde{e} : \upsilon \qquad \Omega;\Gamma \vdash e' \triangleright \tilde{e}' : \tau' \qquad \Omega;\Gamma, x : \upsilon_x, x' : \tau_{x'} \vdash e'' \triangleright \tilde{e}'' : \tau''}{\Omega;\Gamma \vdash e\texttt{->iterate(}x\ \texttt{[:}\ \upsilon_x\texttt{]; }x'\ \texttt{[:}\ \tau_{x'}\texttt{] = }e'\ \texttt{|}\ e''\texttt{)} \triangleright \texttt{Set\{}\tilde{e}\texttt{\}->iterate(}x\texttt{; }x'\texttt{ = }\tilde{e}'\ \texttt{|}\ \tilde{e}''\texttt{)} : \tau_{x'}}$$
$$]\text{where } \upsilon_x = \upsilon,\ \tau_{x'} = \tau'[\text{ if } \upsilon \leq_\Omega \upsilon_x \text{ and } \tau', \tau'' \leq_\Omega \tau_{x'}$$

$$(\text{Sing}^{:}_2) \quad \frac{\Omega;\Gamma \vdash e \triangleright \tilde{e} : \upsilon \qquad (\Omega;\Gamma \vdash e_i \triangleright \tilde{e}_i : \tau_i)_{1\leq i\leq n}}{\Omega;\Gamma \vdash e\texttt{->}o\texttt{(}e_1\texttt{, ..., }e_n\texttt{)} \triangleright \texttt{Set\{}\tilde{e}\texttt{\}->}o_{\tau'_0}\texttt{(}\tilde{e}_1\texttt{, ..., }\tilde{e}_n\texttt{)} : \tau'_0} \quad \text{if } fd_\Omega(o,\texttt{Set(}\upsilon\texttt{)},(\tau_i)_{1\leq i\leq n}) = \tau'.o : (\tau'_i)_{1\leq i\leq n} \to \tau'_0$$

$$(\text{Short}^{:}_1) \quad \frac{\Omega;\Gamma \vdash e \triangleright \tilde{e} : \overline{\sigma}(\tau)}{\Omega;\Gamma \vdash e.a \triangleright \tilde{e}\texttt{->iterate(i; a = }\sigma\texttt{\{\} |}\ \texttt{a->including}_{\sigma(\tau'')}\texttt{(i.}a_\tau\texttt{))} : \sigma(\tau'')}$$
if $fd_\Omega(a,\tau) = \tau'.a : \tau''$ and
where $\sigma = \texttt{Bag}$ if $\overline{\sigma} \neq \texttt{Sequence}$, and $\sigma = \texttt{Sequence}$ otherwise

$$(\text{Short}^{:}_2) \quad \frac{\Omega;\Gamma \vdash e \triangleright \tilde{e} : \overline{\sigma}(\tau) \qquad (\Omega;\Gamma \vdash e_i \triangleright \tilde{e}_i : \tau_i)_{1\leq i\leq n}}{\Omega;\Gamma \vdash e\texttt{.}o\texttt{(}e_1\texttt{, ..., }e_n\texttt{)} \triangleright \tilde{e}\texttt{->iterate(i; a = }\sigma\texttt{\{\} |}\ \texttt{a->including}_{\sigma(\tau'_0)}\texttt{(i.}o_{\tau'_0}\texttt{(}\tilde{e}_1\texttt{, ..., }\tilde{e}_n\texttt{)))} : \sigma(\tau'_0)}$$
if $fd_\Omega(o,\tau,(\tau_i)_{1\leq i\leq n}) = \tau'.o : (\tau'_i)_{1\leq i\leq n} \to \tau'_0$ and
where $\sigma = \texttt{Bag}$ if $\overline{\sigma} \neq \texttt{Sequence}$, and $\sigma = \texttt{Sequence}$ otherwise

plications of the least upper bound operator $\bigsqcup$) are well defined. The metavariables that are used in the rules and which may be variously decorated range as follows: $l \in \mathit{Literal}$; $\upsilon \in U_\Omega$, $\zeta \in C_\Omega$, $\sigma \in S$, $\overline{\sigma} \in \overline{S}$, $\tau \in T_\Omega$, $\delta \in D_\Omega$; $x \in \mathit{Var}$; $a, o \in \mathit{Name}$; $e \in \mathit{Expr}$, $\tilde{e} \in \mathit{A\text{-}Expr}$, $d \in \mathit{AttrDef} \cup \mathit{OpDef}$, $\tilde{d} \in \mathit{A\text{-}AttrDef} \cup \mathit{A\text{-}OpDef}$. For the optional type assertions for `let` and `iterate` (see rules (Let$^{:}$), (Iter$^{:}$), and (Sing$^{:}$)) we use the following convention: If they are not present, the part of the side-conditions bracketed with $]\ldots[$ applies.

The rules follow the OCL specification ([32, Chapt. 6] and also [33, Chapt. 2]) as closely as possible. The rules

(Spec$^\cdot$) and (Def$^\cdot_1$–Def$^\cdot_2$) in Table 3 treat definitions of pseudofeatures as being simultaneous; dependent definitions may be easily introduced (cf. [27]).

The rule (Undef$^\cdot$) assigns type `Void` to `undef`; alternatively, every type could show an undefined value and some naming convention, like `Integer::undef`, or type assertion, like `undef : Integer`, could be used to avoid `Void`. Similarly, the rule (Coll$^\cdot$) provides a unique type for the empty concrete collections (in contrast to [11]) using σ`(Void)`. As required by OCL 2.0 [33] and as [12], we include nested collections, in contrast to OCL 1.5 [32, Sect. 6.5.13] where flattening is applied to all collections.

Generally, the least upper bound (cf. [38]) is not directly justified by the specification. In particular, Schürr [42] suggests to employ union types instead; [11, 38] require homogenous collections; the typing rules of [23] depend on the expression order.

The type of a conditional expression, as given by the rule (Cond$^\cdot$), complies with OCL 2.0 [33], which differs from OCL 1.4 [31, pp. 6–35]. There, independently of the type of e_2, the (evaluation) type of e_1 is assumed to be the type of the whole expression; [36] requires comparable types, [11] a single type.

The casting rule (Cast$^\cdot$) in Table 4 allows both down- and up-cast, see OCL 1.5 [32, pp. 6–10, 6–17f., 6–30] ([36] requires that the new type is smaller than the original type; casting not present in [11, 20]). Note, however, that this rule does not allow for arbitrary expressions resulting in a type as the argument for `asType`, since this would imply term-dependent types as, for example, in

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

which is ruled out by the grammar in Table 1. Similarly, (Inst$^\cdot$) does not allow `allInstances` to be called on arbitrary expressions, but only type literals.

The annotation in (Feat$^\cdot_1$) accounts for the retrieval of an overloaded or overridden structural feature, opposite association end, association class, see OCL 1.5 [32, pp. 6–17f.] (not present in [11, 36]), and also of tuple components. The annotations in (Feat$^\cdot_2$) and (Prop$^\cdot$) are necessary, since we do not require any return type restriction for query behavioural features and properties (in contrast to [11]).

The rules (Sing$^\cdot_1$–Sing$^\cdot_2$) are the so-called "singleton" rules, see OCL 1.5 [32, pp. 6–14], allowing to apply collection properties to non-collection expressions (not present in [11, 36]); note that such an expression must be of a type in U_Ω in contrast to (Iter$^\cdot$) and (Prop$^\cdot$). In particular, these rules account for `self.a->notEmpty()` to be well-formed, if `a` is an opposite association end with multiplicity $0\,..\,1$ [32, pp. 6–14]. The rules (Short$^\cdot_1$–Short$^\cdot_2$) define the shorthand notation for features on members of collections, see OCL 1.5 [32, pp. 6–25] (not present in [6, 11, 20, 36]); notice that the expression must be of collection type in contrast to (Feat$^\cdot_1$) and (Feat$^\cdot_2$). There is no subsumption rule since such a rule would interfere with the overriding of properties and features (cf. e.g. [15]).

Let us illustrate type inference by the following examples.

- The term `Set{}` has type `Set(Void)` by (Coll$^\cdot$).
- The term `Set{"1", 1}` has type `Set(Any)` by (Coll$^\cdot$), but the term `Set{Boolean, 1}` cannot be typed since `Type` and `Integer` have no common supertype.
- The term `Set{Set{},Set{1}}` has type `Set(Set(Integer))` by (Coll$^\cdot$), but the term `Set{1,Set{1}}` cannot be typed because there is no least upper bound for both `Integer` and `Set(Integer)`.
- Given that `Set(Void)` $\leq_\Omega$ `Set(Integer)`, the term `Set{}->union(Set{1})` has type `Set(Integer)` by (Feat$^\cdot_2$).
- The term `1->including(1)` has type `Boolean` by (Sing$^\cdot_2$); the term `Set{1, 2, 3}.+(1)` has type `Bag(Integer)` by (Short$^\cdot_2$).

3.3 Unique typing

The above presented type inference system, in contrast to [11, 39], entails unique annotations as well as unique types and declarations. Unique typing, on the one hand, makes a more efficient parsing possible, and on the other increases the readability of an OCL specification.

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

Proof. By structural induction on the term t using the fact that, in particular, the antecedents of (Iter$^\cdot$) and (Sing$^\cdot_1$), (Feat$^\cdot_1$) and (Short$^\cdot_1$), (Feat$^\cdot_2$) and (Short$^\cdot_2$), (Prop$^\cdot$) and (Short$^\cdot_2$) are mutually exclusive. □

If $\Omega;\Gamma\vdash t\triangleright\tilde{t}:\theta$, then t and $\tilde{t}$ are said to be *well typed*. We also write $\Omega;\Gamma\vdash\tilde{t}:\theta$; the corresponding t can be obtained by erasing the annotations and adding suitable type assertions for `let` and `iterate`.

4 Operational semantics

The evaluation 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; such dynamic bases are independent of static bases. Dynamic bases uniformly capture the possible non-determinism and non-termination of pseudofeatures. We define a big-step operational semantics for evaluating OCL terms over a dynamic basis. This operational semantics operates on

normalised and annotated terms that carry information for retrieving overridden structural features as well as information on the expected return type of a behavioural feature call. These data in general will be obtained by type inference. Accordingly, we also define 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: If a term over a static basis can be typed, then the result of evaluating it in a conforming dynamic basis has the expected type.

The big-step operational semantics subsumes the operational semantics as defined by Clark [11]. In particular, the non-determinism of OCL expressions is made explicit (cf. [11, Thm. 1]). We also take into account other special features of OCL like the non-sequential evaluation order of `and` and `or` and auxiliary definitions.

4.1 Dynamic bases

A *dynamic basis* ω defines values and results, a typing relation, implementation retrieval functions, and an extension mechanism for implementations. Dynamic bases and their implementation retrieval functions as well as the extension mechanism are in one-to-one correspondence to the paronymous entities in static bases. The particular notion of run-time types and a special typing relation underline the independence of dynamic bases from static bases.

Let us illustrate dynamic bases and their associated properties by means of an exemplary definition of a dynamic basis ω based on the static structure diagram in Fig. 1. Let the collection of instances (or objects) O_ω in ω be the set $\{o, o'\}$. Every instance is associated with its actual types from the run-time classifiers $C_\omega = \{\mathtt{A}, \mathtt{B}\}$; we let $o :_\omega \mathtt{A}$, $o' :_\omega \mathtt{B}$, and $o' :_\omega \mathtt{A}$; this instance relation between instances and (run-time) classifiers records all types of an instance. Let attribute `a` of o be bound to o itself, attribute `b` of o' be bound to o' itself, the opposite association end `bs` be bound to the empty set, method `m` called on o not terminate, and method `m` called on o' yield o.

In the dynamic basis ω, the valuation of attributes and the behaviour of operations is reflected by the implementation retrieval function $impl_\omega$. For attributes and opposite association ends, the implementation retrieval function takes as parameters the feature name, the type in which the feature has been declared, and the instance for which the feature is to be evaluated. Thus

$$impl_\omega(\mathtt{a}_\mathtt{A}, o) = o$$
$$impl_\omega(\mathtt{b}_\mathtt{B}, o') = o'$$
$$impl_\omega(\mathtt{bs}_\mathtt{B}, o') = \mathtt{Set\{\}}$$

and $impl_\omega(\mathtt{x}_\mathtt{A}, o)$ is undefined. The extension of attribute and opposite association end names by their declaring types allows us to retrieve overridden features. For operations, the implementation retrieval function takes as parameters the operation name, the expected return type, the instance for which the operation call is to be evaluated, and the arguments of the operation call. Thus

$$impl_\omega(\mathtt{m}_\mathtt{B}, o, ()) = \bot$$
$$impl_\omega(\mathtt{m}_\mathtt{A}, o', ()) = o$$

The extension of operation names by the expected return type caters for UML's ad hoc polymorphism which does not require method `m` of `B` to be declared with a return type less than or equal to the return type of `m` in `A`.

Finally, taking the constraint note into account, the dynamic basis is extended by the implementation $\iota = \mathtt{A.f} \equiv \mathtt{1+2}$, and thus, in particular, the implementations of the extended dynamic basis ω, ι contain

$$impl_{\omega,\iota}(\mathtt{f}_\mathtt{A}, o') = \mathtt{A.f} \equiv \mathtt{1+2}$$

where the annotation in $\mathtt{f}_\mathtt{A}$ reflects the defining type of `f`, as `f` is a pseudoattribute. Note the use of the symbol "≡" to avoid confusion with the equality sign: The implementation of the pseudofeature $\mathtt{f}_\mathtt{A}$ called on o' is ι, namely $\mathtt{A.f} \equiv \mathtt{1+2}$.

In the following we formally define the tools used in this example, namely values, types, typing relation, implementation retrieval, and extensions of dynamic bases.

Values and results. The outcome of an OCL term may be a literal basic value like `1` or `true`; an instance (or object) from the collection of instances over which the term is evaluated; a collection or tuple value like `Set{ ... }`; a (run-time) type like `Integer`; or the result `undef`. However, the evaluation of an OCL term may not terminate at all.

The *(run-time) types* T_ω of a dynamic basis ω are defined as T_Ω for a static basis Ω in Sect. 3.1, but replacing C_Ω by a finite set parameter C_ω representing the classifier types in a model. The finite set O_ω represents the *instances* in a model, disjoint from *Literal* (see Table 1) and from T_ω. With these parameters, the *values* V_ω and *results* $\overline{V}_\omega$ of a dynamic basis ω are defined as follows:

$$\begin{aligned}
\overline{V}_\omega &::= V_\omega \mid \mathtt{undef} \\
V_\omega &::= N_\omega \mid (\mathtt{Set} \mid \mathtt{Bag} \mid \mathtt{Sequence})\,\mathtt{\{}\,[V_\omega\,\mathtt{\{,}\,V_\omega\}]\,\mathtt{\}} \mid \\
&\qquad \mathtt{Tuple\{}\,\mathit{Name}\,\mathtt{=}\,V_\omega\,\mathtt{\{,}\,\mathit{Name}\,\mathtt{=}\,V_\omega\}\,\mathtt{\}} \\
N_\omega &::= \mathit{Literal} \mid T_\omega \mid O_\omega
\end{aligned}$$

All names in a value `Tuple{...}` have to be distinct.

Values of the form σ`{...}` with $\sigma \in S = \{$`Set`, `Bag`, `Sequence`$\}$ are *collection values*, the values of the form `Tuple{...}` are *tuple values*. The values in N_ω are *basic values*. We assume suitably axiomatised arithmetical, boolean, etc. functions and relations on values such that, e.g., $\mathtt{1} + \mathtt{1} = \mathtt{2}$, $\mathtt{1.0} = \mathtt{1}$, $\mathtt{false} \wedge \mathtt{true} = \mathtt{false}$, `Set{1, 2}` = `Set{2, 1, 1}`, $\mathtt{1} \leq \mathtt{2}$, `Tuple{a = 1, b = 2}` = `Tuple{b = 2, a = 1}`, etc.

Collection values are constructed by a map $make_\omega : S \times V_\omega^* \to V_\omega$ such that $make_\omega(\sigma, v_1 \cdots v_n) = \sigma$`{`$v_1$`, ...,`

v_n`}`; if $\sigma = $ `Set`, repetitions in $v_1 \cdots v_n$ are discarded, such that only the leftmost occurrence of a value remains. If $n = 0$, we write $make_\omega(\sigma, \emptyset)$. More generally, for a set $M = \{v_1, \ldots, v_n\}$, we let $make_\omega(\sigma, M)$ denote $make_\omega(\sigma, v_1 \cdots v_n)$. By abuse of notation, we put $make_\omega($`undef`$) =$ `undef`.

A collection value $v = \sigma$`{`v_1`, ..., `v_n`}` has a *sequence value representation* v', written as $v \rightsquigarrow v'$, if either $\sigma =$ `Sequence` and $v = v'$ or $make_\omega($`Sequence`$, v_{\pi(1)} \cdots v_{\pi(n)}) = v'$ for some permutation π of $1, \ldots, n$ and where $v_1, \ldots, v_n$ are all different if $\sigma =$ `Set`. In general, a collection value has several different sequence value representations; e.g. `Set{1, 2}` $\rightsquigarrow$ `Sequence{1, 2}`, `Set{1, 2}` $\rightsquigarrow$ `Sequence{2, 1}`, and, furthermore, `Set{1, 2, 1}` $\rightsquigarrow$ `Sequence{1, 2}`, by `Set{1, 2, 1}` $=$ `Set{1, 2}`; but, in particular, `Set{1, 2}` $\not\rightsquigarrow$ `Sequence{1, 1, 2}`.

For the representation of (bounded) non-determinism and non-termination, as for instance witnessed by turning a set value into a sequence value by `asSequence` [32, pp. 6–40], we introduce the *powerdomain of results* $\wp(\overline{V}_\omega)_\perp$, defined as the set of all non-empty subsets X of $\overline{V}_\omega \cup \{\perp\}$ such that if X is infinite then X contains $\perp$ [35]; the special element $\perp$ indicates non-termination.

Typing relation. The typing relation in a dynamic basis ω is based on an *instance relation* between instances and classifiers, denoted by $:_{O_\omega} \subseteq O_\omega \times C_\omega$. This relation puts an object into relation with all the classifiers the object is an instance of, which may be several in the presence of inheritance; thus the instance relation is left-total and not necessarily functional.

Given a instance relation $:_{O_\omega} \subseteq C_\omega \times C_\omega$, the *typing relation* $:_\omega \subseteq \overline{V}_\omega \times T_\omega$ between results and types of ω is defined as the least relation satisfying the following axioms:

1. for all $v \in$ *Literal*, $v :_\omega type(v)$,
   where $type(v)$ is the type of the literal as defined in Sect. 3.1
2. for all $v \in T_\omega$, $v :_\omega$ `Type`
3. for all $v \in O_\omega$, if $v :_{O_\omega} \tau$ then $v :_\omega \tau$
4. for all $\tau \in T_\omega$, `undef` $:_\omega \tau$
5. for all $v \in V_\omega$, if $v :_\omega$ `Integer` then $v :_\omega$ `Real`
6. for all $v \in V_\omega$, if $v :_\omega \alpha \in A_\omega$ then $v :_\omega$ `Any`
7. for all $v_1, \ldots, v_n \in V_\omega$ and $a_1, \ldots, a_n \in$ *Name*,
   if $v_i :_\omega \tau_i \in T_\Omega$, $1 \le i \le n$,
   then `Tuple{`a_1 `=` v_1`, ..., `a_n `=` v_n`}` $:_\omega$ `Tuple(`$a_{i_1} : \tau_{i_1}$`, ..., `$a_{i_k} : \tau_{i_k}$`)`
   for $1 \le i_1 < \cdots < i_k \le n$, $k \ge 1$
8. σ`{}` $:_\omega \sigma$`(Void)`
9. for all $v_1, \ldots, v_n \in V_\omega$,
   if $v_i :_\omega \tau \in T_\omega$, $1 \le i \le n$, then σ`{`v_1`, ..., `v_n`}` $:_\omega \sigma$`(`τ`)`
10. for all $v \in V_\omega$, if $v :_\omega \sigma$`(`τ`)` then $v :_\omega$ `Collection(`τ`)`

Note that the relation $:_\omega$ is again left-total. If $\overline{v} :_\omega \tau$, we say that "$\overline{v}$ is in (or belongs to) τ" and also that "$\overline{v}$ inhabits τ". In particular, the only inhabitant of type `Void` is `undef`. The type hierarchy on the run-time types of ω can be derived from the typing relation: We define the *subtype relation* $\le_\omega \subseteq T_\omega \times T_\omega$ as follows: $\tau \le_\omega \tau'$ if, and only if, $\overline{v} :_\omega \tau$ implies $\overline{v} :_\omega \tau'$ for every $\overline{v} \in \overline{V}$. Thus in contrast to the type inference system, we introduce explicit type subsumption in dynamic bases, i.e., if $v :_\omega \tau$ and $\tau \le_\omega \tau'$ then also $v :_\omega \tau'$.

A *finite type* is a type $\tau \in T_\omega$ such that the set $\{v \in V_\omega \mid v :_\omega \tau\}$ is finite; explicitly, the only finite types are `Void`, `Boolean`, `Type`, a type in C_ω, and a type of the form σ`(`τ`)` with τ finite. For a finite type τ, we denote the finite set $\{v \in V_\omega \mid v :_\omega \tau\}$ by $\omega(\tau)$; for all other types τ, $\omega(\tau)$ is `undef`.

Implementation retrieval. The retrieval of implementations of (overloaded or overridden) properties, features, pseudofeatures, opposite association ends, and association classes in a dynamic basis ω is defined by two partial maps yielding *implementations*. Implementations consist of an *implementation signature* and an *implementation body*, separated by $\equiv$. The implementation signature locates the feature in the type hierarchy and, in the case of behavioural features, it also holds information on the formal parameter names and the declared return type of the feature. The implementation body is either an annotated expression, originating from an auxiliary definition, or a function from a list of values to the powerdomain of results, for predefined features. Formally,

$$\begin{aligned} I_\omega ::= {} & T_\omega \,.\, Name \equiv A\text{-}Expr \mid \\ & T_\omega \,.\, Name \equiv F_\omega \mid \\ & T_\omega \,.\, Name(\ Var^*\ ) : T_\omega \equiv A\text{-}Expr \mid \\ & T_\omega \,.\, Name(\ Var^*\ ) : T_\omega \equiv F_\omega \end{aligned}$$

where F_ω denotes the set of non-deterministic *implementation functions* $V_\omega^+ \to \wp(\overline{V}_\omega)_\perp$. An implementation function takes a value for the implicit `self` parameter and the remaining formal parameters and yields a set in the powerdomain of results, which can possibly contain `undef` or $\perp$.

For attribute-like features, given a name a, a type τ (the annotation), and a value v with $v :_\omega \tau$, the partial function

$$impl_\omega : Name \times T_\omega \times V_\omega \rightharpoonup I_\omega$$

yields, when defined, an implementation $\tau.a \equiv \psi$ representing the implementation of a structural (pseudo-) feature, an opposite association end, an association class, or a tuple component with name a, which has to be defined in τ as required by the annotation.

In particular, if $impl_\omega(a, \tau, v) = \tau.a \equiv \psi$ with $\psi \in F_\omega$, then $\psi : V_\omega \to \wp(\overline{V}_\omega)_\perp$, i.e., ψ has to be unary. If $impl_\omega(a, \tau, v)$ is defined, then $impl_\omega(a, \tau, v')$ 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 (run-time) types as v.

For operation-like features, given a name o, a type τ (the annotation), a value v with $v :_\omega \tau'$, and a sequence of values $(v_i)_{1 \le i \le n}$, the partial function

$$impl_\omega : Name \times T_\omega \times V_\omega \times V_\omega^* \rightharpoonup I_\omega$$

yields, when defined, an implementation $\tau'.o((x_i)_{1 \le i \le n}) : \tau \equiv \psi$ representing the implementation of a query be-

havioural (pseudo-)feature or a property with name o, defined with return type τ as required by the annotation.

In particular, if $impl_\omega(o, \tau, v, (v_i)_{1\le i\le n}) = \tau'.o((x_i)_{1\le i\le n}) : \tau \equiv \psi$ with $\psi \in F_\omega$, then $\psi : V_\omega^n \to \wp(\overline{V}_\omega)_\perp$, i.e., ψ has to show the desired arity. If $impl_\omega(o, \tau, v, (v_i)_{1\le i\le n})$ is defined, then $impl_\omega(o, \tau, v', (v_i)_{1\le i\le n})$ is defined for all v' such that $v :_\omega \tau'$ implies $v' :_\omega \tau'$.

In order to enhance readability, the type argument of $impl_\omega$ is written as a subscript of the name argument as e.g. $impl_\omega(a_\tau, v)$ and $impl_\omega(o_\tau, v, (v_i)_{1\le i\le n})$. By abuse of notation, if $impl_\omega(a_\tau, v) = \tau.a \equiv \psi$, with $\psi \in F_\omega$, we write $impl_\omega(a_\tau, v)$ for $\psi(v)$. Similarly, if $impl_\omega(o_\tau, v, (v_i)_{1\le i\le n}) = \tau'.o((x_i)_{1\le i\le n}) : \tau \equiv \psi$ with $\psi \in F_\omega$, we write $impl_\omega(o_\tau, v, (v_i)_{1\le i\le n})$ for $\psi(v, (v_i)_{1\le i\le n})$. If the resulting function is deterministic, we identify the singleton result set and its element.

For the example introduced above, see Fig. 1, we have

$$impl_\omega(\mathtt{m_B}, o, ()) = \mathtt{B.m}() \equiv m$$

with $m : V_\omega \to \wp(\overline{V}_\omega)_\perp$ and $m(o) = \{\perp\}$ which we conveniently abbreviated into $impl_\omega(\mathtt{m_B}, o, ()) = \perp$. Similarly, we have

$$impl_\omega(\mathtt{a_A}, o) = \mathtt{A.a} \equiv a$$

with $a : V_\omega \to \wp(\overline{V}_\omega)_\perp$ and $a(o) = \{o\}$.

Table 5 contains some sample axioms for the retrieval of the implementation of built-in OCL properties.

Extensions. We require that a dynamic basis ω be *extendable*. If $\tau \in T_\omega$ and $impl_\omega(a_\tau, v)$ is undefined for all $v :_\omega \tau$, then ω can be extended by putting $\tau.a \equiv \psi$ with ψ either an *A-Expr* of an F_ω. If $impl_\omega(o_{\tau_0}, v, (v_i)_{1\le i\le n})$ is undefined for all $v :_\omega \tau$ and all $v_1, \ldots, v_n \in V_\omega$, then ω can be extended by putting $\tau.o((x_i)_{1\le i\le n}) : \tau_0 \equiv \psi$. An extension ω' of ω must again be a dynamic basis, i.e., $impl_{\omega'}$ must respect inheritance. For the extension by an implementation $\iota = \tau.a \equiv \psi$ we require that $impl_{\omega'}(a_\tau, v) = \iota$ for all $v :_\omega \tau$ and that $impl_{\omega'}(a'_{\tau'}, v)$ is the same as $impl_\omega(a'_{\tau'}, v)$ if $a' \neq a$. Similarly, for the extension by an implementation $\iota = \tau.o((x_i)_{1\le i\le n}) : \tau_0 \equiv \psi$ we require that $impl_{\omega'}(o_{\tau_0}, v, (v_i)_{1\le i\le n}) = \iota$ for all $v :_\omega \tau$ and $v_1, \ldots, v_n \in V_\omega$ and that $impl_{\omega'}(o'_{\tau'}, v', (v_i)_{1\le i\le n})$ is the same as $impl_\omega(o'_{\tau'}, v', (v_i)_{1\le i\le n})$ if $o' \neq o$. Finally, we require that $T_{\omega'} = T_\omega$ and $:_{\omega'} = :_\omega$.

Table 5. Semantics of sample built-in OCL properties

$impl_\omega(a_{\mathtt{Tuple}(a\,:\,\tau)}, \mathtt{Tuple\{}a\,\mathtt{=}\,v\mathtt{\}}) = \mathtt{Tuple}(a\,:\,\tau).a \equiv v$
$impl_\omega(\mathtt{=_{Boolean}}, v, v') = (v = v')$
$impl_\omega(\mathtt{oclIsKindOf_{Boolean}}, v, \tau) = \{v :_\omega \tau\}$
$impl_\omega(\mathtt{first}_\tau, \mathtt{Sequence\{}v_1\mathtt{,}\ \ldots\mathtt{,}\ v_n\mathtt{\}}) = \{v_1\}$
$impl_\omega(\mathtt{including}_{\sigma(\tau)}, v, v') = \{make_\omega(\sigma, v\ v')\}$
$impl_\omega(\mathtt{union}_{\sigma(\tau)}, v, v') = \{make_\omega(\sigma, v\ v')\}$

In the example above, the constraint note in the diagram of Fig. 1 extends the dynamic basis ω with the implementation $\iota = \mathtt{A.f} \equiv \mathtt{1.+(2)}$ yielding the dynamic basis ω' with

$$impl_{\omega'}(\mathtt{f_A}, o') = \mathtt{A.f} \equiv \mathtt{1.+(2)}$$

which leaves ω unchanged with respect to its types and its typing relation.

The above are only weak requirements on 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 the chosen scheme.

4.2 Evaluation

The operational semantics evaluates annotated OCL terms in the context of a dynamic basis and some variable assignments. The variable assignments record the values of the terms on which the term of interest depends. The dynamic basis delivers the information on the actual state of the object system, as sketched above.

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 $x_i \in Var \cup \{\mathtt{self}\}$ and $v_i \in V_\omega$ for all $1 \le i \le n$. We denote the set $\{x_1, \ldots, x_n\}$ by $\mathrm{dom}(\gamma)$ and the value v_i by $\gamma(x_i)$ if $x_i \neq x_j$ for all $i < j \le n$. The empty variable environment is denoted by $\emptyset$, concatenation of variable environments γ and γ' by γ, γ'.

The operational semantics derives *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 annotated term, and $\rho \in \overline{V}_\omega \cup I_\omega$. If such a judgement can be derived, then the term $\tilde{t}$ is said to evaluate to ρ in the dynamic basis ω and with the variables (on which the evaluation of $\tilde{t}$ may depend) assigned as given by γ. The empty variable environment may be omitted.

Judgements are derived by the rules in Tables 6–7. As in the previous section we require that a rule be applied only if all its constituents are well defined. The metavariables, that may be variously decorated, range as follows: $l \in Literal$; $\zeta \in C_\omega$, $\sigma \in S$, $\tau \in T_\omega$; $x \in Var$; $a, o \in Name$; $v \in V_\omega, \overline{v} \in \overline{V}_\omega, \iota \in I_\omega$; $\tilde{e} \in$ *A-Expr*, $\tilde{d} \in$ *A-AttrDef* $\cup$ *A-OpDef*. We additionally adopt the following general *failing convention* that applies to all rules with the exceptions of the rules ($\mathrm{And}_1^\downarrow$–$\mathrm{And}_3^\downarrow$) and ($\mathrm{Or}_1^\downarrow$–$\mathrm{Or}_3^\downarrow$): if `undef` occurs as a result in a judgement of a premise of some rule where a value $v \in V_\omega$ is required, the term in the conclusion evaluates to `undef`.

The operational rules are presented in close correspondence to the typing rules in Tables 3–4. All rules, except the rules ($\mathrm{Cond}_1^\downarrow$–$\mathrm{Cond}_2^\downarrow$), ($\mathrm{And}_2^\downarrow$–$\mathrm{And}_3^\downarrow$), and ($\mathrm{Or}_2^\downarrow$–$\mathrm{Or}_3^\downarrow$) in Table 6, require of all subterms to be fully evaluated and to result in a value in V_ω in order to deliver a result for the term of interest. In particular, in contrast to [31, Sect. 6.4.10], we treat conditionals as being non-failing

Table 6. Operational semantics I

$$(\mathrm{Spec}^{\downarrow})\ \frac{(\omega,(\iota_j)_{1\le j\le n};\gamma\vdash \tilde{d}_i\downarrow\iota_i)_{1\le i\le n}\quad (\omega,(\iota_j)_{1\le j\le n};\gamma,\texttt{self}\mapsto v\vdash\tilde{e}\downarrow v_v)_{v\in\omega(\zeta)}}{\omega;\gamma\vdash(\texttt{context}\ \zeta_i\ \texttt{def:}\ \tilde{d}_i)_{1\le i\le n}\ \texttt{context}\ \zeta\ \texttt{inv:}\ \tilde{e}\downarrow\bigwedge_v v_v}$$

$$(\mathrm{Def}_1^{\downarrow})\ \omega;\gamma\vdash a_\zeta\ \texttt{:}\ \tau\ \texttt{=}\ \tilde{e}\downarrow\zeta.a\equiv\tilde{e}$$

$$(\mathrm{Def}_2^{\downarrow})\ \omega;\gamma\vdash o_\zeta(x_1\ \texttt{:}\ \tau_1\texttt{,}\ \ldots\texttt{,}\ x_n\ \texttt{:}\ \tau_n)\ \texttt{:}\ \tau\ \texttt{=}\ \tilde{e}\downarrow\zeta.o((x_i)_{1\le i\le n}):\tau\equiv\tilde{e}$$

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

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

$$(\mathrm{Undef}^{\downarrow})\ \omega;\gamma\vdash\texttt{undef}\downarrow\texttt{undef}$$

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

$$(\mathrm{Tup}^{\downarrow})\ \frac{(\omega;\gamma\vdash\tilde{e}_i\downarrow v_i)_{1\le i\le n}}{\omega;\gamma\vdash\texttt{Tuple\{}a_1\ \texttt{=}\ \tilde{e}_1\texttt{,}\ \ldots\texttt{,}\ a_n\ \texttt{=}\ \tilde{e}_n\texttt{\}}\downarrow\texttt{Tuple\{}a_1\ \texttt{=}\ v_1\texttt{,}\ \ldots\texttt{,}\ a_n\ \texttt{=}\ v_n\texttt{\}}}$$

$$(\mathrm{Let}^{\downarrow})\ \frac{\omega;\gamma\vdash\tilde{e}\downarrow v\qquad\omega;\gamma,x\mapsto v\vdash\tilde{e}'\downarrow v'}{\omega;\gamma\vdash\texttt{let}\ x\ \texttt{=}\ \tilde{e}\ \texttt{in}\ \tilde{e}'\downarrow v'}$$

$$(\mathrm{Cond}_1^{\downarrow})\ \frac{\omega;\gamma\vdash\tilde{e}\downarrow\texttt{true}\qquad\omega;\gamma\vdash\tilde{e}_1\downarrow\overline{v}_1}{\omega;\gamma\vdash\texttt{if}\ \tilde{e}\ \texttt{then}\ \tilde{e}_1\ \texttt{else}\ \tilde{e}_2\ \texttt{endif}\downarrow\overline{v}_1}$$

$$(\mathrm{Cond}_2^{\downarrow})\ \frac{\omega;\gamma\vdash\tilde{e}\downarrow\texttt{false}\qquad\omega;\gamma\vdash\tilde{e}_2\downarrow\overline{v}_2}{\omega;\gamma\vdash\texttt{if}\ \tilde{e}\ \texttt{then}\ \tilde{e}_1\ \texttt{else}\ \tilde{e}_2\ \texttt{endif}\downarrow\overline{v}_2}$$

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

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

$$(\mathrm{And}_3^{\downarrow})\ \frac{(\omega;\gamma\vdash\tilde{e}_i\downarrow\overline{v}_i)_{1\le i\le 2}}{\omega;\gamma\vdash\tilde{e}_1\ \texttt{and}\ \tilde{e}_2\downarrow\texttt{undef}}\ \text{if } \overline{v}_1\neq\texttt{false},\ \overline{v}_2=\texttt{undef} \text{ or } \overline{v}_1=\texttt{undef},\ \overline{v}_2\neq\texttt{false} \qquad (\mathrm{Or}_3^{\downarrow})\ \frac{(\omega;\gamma\vdash\tilde{e}_i\downarrow\overline{v}_i)_{1\le i\le 2}}{\omega;\gamma\vdash\tilde{e}_1\ \texttt{or}\ \tilde{e}_2\downarrow\texttt{undef}}\ \text{if } \overline{v}_1\neq\texttt{true},\ \overline{v}_2=\texttt{undef} \text{ or } \overline{v}_1=\texttt{undef},\ \overline{v}_2\neq\texttt{true}$$

$$(\mathrm{Iter}^{\downarrow})\ \frac{\omega;\gamma\vdash\tilde{e}\downarrow v\qquad\omega;\gamma\vdash\tilde{e}'\downarrow v'_0\qquad(\omega;\gamma,x\mapsto v_i,x'\mapsto v'_{i-1}\vdash\tilde{e}''\downarrow v'_i)_{1\le i\le n}}{\omega;\gamma\vdash\tilde{e}\texttt{->iterate(}x\texttt{;}\ x'\ \texttt{=}\ \tilde{e}'\ \texttt{|}\ \tilde{e}''\texttt{)}\downarrow v'_n}\quad \text{if } v\rightsquigarrow\texttt{Sequence\{}v_1\texttt{,}\ \ldots\texttt{,}\ v_n\texttt{\}}$$

and non-strict in the evaluation of the `then` and `else` clauses (for strict rules for conditionals see [9]). Moreover, the (And$^{\downarrow}$) and (Or$^{\downarrow}$) rules yield parallel `Boolean` connectives `and` and `or` [31, pp. 6–11] (not treated in [6, 11, 36]), such that even terms containing a non-terminating subterm may evaluate to a result (for a more detailed discussion on the three-valued logic of OCL see [21]; however, non-termination is not discussed). The parallel behaviour of `and` may be summarised by the following table:

and	true	false	undef	$\bot$
true	true	false	undef	$\bot$
false	false	false	false	false
undef	undef	false	undef	$\bot$
$\bot$	$\bot$	false	$\bot$	$\bot$

Table 7. Operational semantics II

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

$$(\text{Inst}^{\downarrow})\ \omega;\gamma\vdash\tau\texttt{.allInstances()}\downarrow \mathit{make}_\omega(\texttt{Set},\omega(\tau))$$

$$(\text{IsUndef}^{\downarrow})\ \frac{\omega;\gamma\vdash\tilde{e}\downarrow\overline{v}}{\omega;\gamma\vdash\tilde{e}\texttt{.isUndef()}\downarrow v}\quad \begin{array}{l}\text{where } v=\texttt{true} \text{ if } \overline{v}=\texttt{undef}, \text{ and}\\ v=\texttt{false} \text{ if } \overline{v}\neq\texttt{undef}\end{array}$$

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

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

$$(\text{Feat}_3^{\downarrow})\ \frac{\omega;\gamma\vdash\tilde{e}\downarrow v\quad (\omega;\gamma\vdash\tilde{e}_i\downarrow v_i)_{1\le i\le n}}{\omega;\gamma\vdash\tilde{e}.o_{\tau_0}\texttt{(}\tilde{e}_1\texttt{, ... , }\tilde{e}_n\texttt{)}\downarrow\overline{v}'}\ \text{if } \overline{v}'\in \mathit{impl}_\omega(o_{\tau_0},v,(v_i)_{1\le i\le n})$$

$$(\text{Feat}_4^{\downarrow})\ \frac{\omega;\gamma\vdash\tilde{e}\downarrow v\quad (\omega;\gamma\vdash\tilde{e}_i\downarrow v_i)_{1\le i\le n}\quad \omega;\texttt{self}\mapsto v,(x_i\mapsto v_i)_{1\le i\le n}\vdash\tilde{e}'\downarrow\overline{v}'}{\omega;\gamma\vdash\tilde{e}.o_{\tau_0}\texttt{(}\tilde{e}_1\texttt{, ... , }\tilde{e}_n\texttt{)}\downarrow\overline{v}'}\ \begin{array}{l}\text{if } \mathit{impl}_\omega(o_{\tau_0},v,(v_i)_{1\le i\le n})=\\ \tau.o((x_i)_{1\le i\le n}):\tau_0\equiv\tilde{e}'\end{array}$$

$$(\text{Prop}_1^{\downarrow})\ \frac{\omega;\gamma\vdash\tilde{e}\downarrow v\quad (\omega;\gamma\vdash\tilde{e}_i\downarrow v_i)_{1\le i\le n}}{\omega;\gamma\vdash\tilde{e}\texttt{->}o_{\tau_0}\texttt{(}\tilde{e}_1\texttt{, ... , }\tilde{e}_n\texttt{)}\downarrow\overline{v}'}\ \text{if } \overline{v}'\in \mathit{impl}_\omega(o_{\tau_0},v,(v_i)_{1\le i\le n})$$

$$(\text{Prop}_2^{\downarrow})\ \frac{\omega;\gamma\vdash\tilde{e}\downarrow v\quad (\omega;\gamma\vdash\tilde{e}_i\downarrow v_i)_{1\le i\le n}\quad \omega;\texttt{self}\mapsto v,(x_i\mapsto v_i)_{1\le i\le n}\vdash\tilde{e}'\downarrow\overline{v}'}{\omega;\gamma\vdash\tilde{e}\texttt{->}o_{\tau_0}\texttt{(}\tilde{e}_1\texttt{, ... , }\tilde{e}_n\texttt{)}\downarrow\overline{v}'}\ \begin{array}{l}\text{if } \mathit{impl}_\omega(o_{\tau_0},v,(v_i)_{1\le i\le n})=\\ \tau.o((x_i)_{1\le i\le n}):\tau_0\equiv\tilde{e}'\end{array}$$

The only rules that possibly introduce the undefined result `undef` are (Undef$^{\downarrow}$) in Table 6, the (Cast$^{\downarrow}$) rule (cf. [31, pp. 6–56]), the (Inst$^{\downarrow}$) rule (cf. [31, pp. 6–19]) and the (Feat$_1^{\downarrow}$), (Feat$_3^{\downarrow}$), and (Prop$_1^{\downarrow}$–Prop$_2^{\downarrow}$) rules in Table 7. Undefined results can only be caught by the (IsUndef$^{\downarrow}$) rule, but not by feature or property calls. The (Iter$^{\downarrow}$) rule allows for considerable, but bounded non-determinism if applied to a collection value that is not a sequence (in contrast to [36, 39]; not present in [6]). Finally, note that (Feat$_1^{\downarrow}$), (Feat$_3^{\downarrow}$), or (Prop$_1^{\downarrow}$) are only applicable if the implementation body retrieved by impl_ω yields a result such that $\bot$, i.e. non-termination, cannot be the outcome of an evaluation; similarly, the recursive nature of (Feat$_2^{\downarrow}$) and (Feat$_4^{\downarrow}$) may prevent the evaluation from terminating.

Let us list some illustrative examples of term evaluation.

- The evaluation of

  ```
  context A def: g(n : Integer) : Integer =
                                        n*g(n+1)
  context A inv: g(1) > 0
  ```

  does not terminate.
- The term `if true then undef else 1` evaluates to `undef` by (Cond$_1^{\downarrow}$),
  the term `if undef then true else false` evaluates to `undef` by the failing convention.
- `true or if 1/0 then true else false` evaluates to `true` by (Or$_2^{\downarrow}$).
  Similarly, $\tilde{e}$ `and false` evaluates to `false` by (And$_2^{\downarrow}$) for every well-typed annotated expression $\tilde{e}$, even if the evaluation of $\tilde{e}$ does not terminate.
- `Set{1, 2}->iterate(i; a = 0 | i)` non-deterministically evaluates to either `2` or `1` by (Iter$^{\downarrow}$), depending on the chosen sequence value representation for `Set{1, 2}`, namely `Sequence{1, 2}` resp. `Sequence{2, 1}`.
  `Sequence{Set{1},Set{2}}->iterate(i;a = Set{} | a->union(i))` deterministically evaluates to `Set{1, 2}`.
- `undef.asType(Boolean)` evaluates to `undef` by the failing convention.
- `Set(Boolean).allInstances()`$\downarrow$
  `Set{Set{}, Set{true}, Set{false}, Set{true, false}}`
  by definition of ω(`Set(Boolean)`) and by (Inst$^{\downarrow}$).

– `Integer.allInstances()`$\downarrow$`undef` by definition of $make_\omega$ and (Inst$^\downarrow$).
– `2.=(2.0)`$\downarrow$`true`

However, the evaluation of `self.a->notEmpty()`, where `a` is an opposite association end with multiplicity $0..1$ will never result in `false`: The powerdomain of results does not contain the empty set, and thus the evaluation of `a` must either contain some value or `undef`, or $\bot$.

4.3 Subject reduction

We define a conformance relation between dynamic and static bases that ensures, on the one hand, that the compile-time and the run-time types as well as the type hierarchies are compatible and, on the other hand, that implementations respect declarations.

Let Ω be static basis and ω a dynamic basis. We say that the *type hierarchy* of ω *conforms to* Ω if $T_\omega = T_\Omega$ and $\leq_\omega = \leq_\Omega$. If the type hierarchy of ω conforms to Ω, an *implementation* ι of ω *conforms to* a *declaration* δ of Ω, written $\iota :_\omega \delta$, if one of the following axioms is satisfied:

1. for $f \in F_\omega$, $(\tau.a \equiv f) :_\omega (\tau.a : \tau')$ if for all $v :_\omega \tau$ and for all $r \in f(v)$ either $r :_\omega \tau'$ or $r = \bot$
2. for $\tilde{e} \in$ *A-Expr*, $(\tau.a \equiv \tilde{e}) :_\omega (\tau.a : \tau')$ if $\Omega; \mathtt{self} : \tau \vdash \tilde{e} : \tau''$ with $\tau'' \leq_\omega \tau'$
3. for $f \in F_\omega$, $(\tau.o((x_i)_{1\leq i\leq n}) : \tau_0 \equiv f) :_\omega (\tau'.o : (\tau'_i)_{1\leq i\leq n} \to \tau'_0)$ if $\tau \leq_\omega \tau'$ and for all $v :_\omega \tau'$, $v_i :_\omega \tau'_i$, $1 \leq i \leq n$, and for all $r \in f(v, (v_i)_{1\leq i\leq n})$ either $r :_\omega \tau'_0$ or $r = \bot$
4. for $\tilde{e} \in$ *A-Expr*, $(\tau.o((x_i)_{1\leq i\leq n}) \equiv \tilde{e} : \tau'_0) :_\omega (\tau'.o : (\tau'_i)_{1\leq i\leq n} \to \tau'_0)$ if $\tau \leq_\omega \tau'$ and $\Omega; \mathtt{self} : \tau', (x_i : \tau'_i)_{1\leq i\leq n} \vdash \tilde{e} : \tau''_0$ with $\tau''_0 \leq_\omega \tau'_0$

A *dynamic basis* ω *conforms to* a *static basis* Ω if

1. the type hierarchy of ω conforms to Ω
2. for every $a \in$ *Name* such that $fd_\Omega(a, \tau) = \tau'.a : \tau''$, $impl_\omega(a_{\tau'}, v)$ is defined for all $v \in V_\omega$ with $v :_\omega \tau$ and $impl_\omega(a_{\tau'}, v) :_\omega fd_\Omega(a, \tau)$
3. for every $o \in$ *Name* such that $fd_\Omega(o, \tau, (\tau_i)_{1\leq i\leq n}) = \tau'.o : (\tau'_i)_{1\leq i\leq n} \to \tau'_0$, $impl_\omega(o_{\tau'_0}, v, (v_i)_{1\leq i\leq n})$ 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 $1 \leq i \leq n$ and $impl_\omega(o_{\tau'_0}, v, (v_i)_{1\leq i\leq n}) :_\omega fd_\Omega(o, \tau, (\tau_i)_{1\leq i\leq n})$

Note that if ω conforms to Ω, $\iota :_\omega \delta$, and both ω, ι and Ω, δ are defined, then ω, ι conforms to Ω, δ.

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 may well result in `undef`. For example,

```
Set{1, 1.2}->iterate(i : Any;
                     a : Sequence(Any) = Sequence{} |
      a->including(i))->first().asType(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 `undef`, 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. 4.2 satisfies the subject reduction property with respect to the type inference system in Sect. 3.2. 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. *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 $\theta \in T_\Omega \cup D_\Omega$ and $\rho \in V_\omega \cup I_\omega$. If $\Omega; \Gamma \vdash t \triangleright \tilde{t} : \theta$ and $\omega; \gamma \vdash \tilde{t} \downarrow \rho$, then $\rho :_\omega \theta$.*

Proof. By induction on the height of the proof tree of $\Omega; \Gamma \vdash t \triangleright \tilde{t} : \theta$; see Appendix A. □

Obviously, the proposition remains valid when replacing $\rho \in V_\omega \cup I_\omega$ by $\rho \in \overline{V}_\omega \cup I_\omega$ since `undef` $:_\omega \tau$ for all $\tau \in T_\omega$.

4.4 Termination

A well-typed OCL constraint, i.e., a well-typed OCL term not showing any auxiliary definitions, always yields some result when evaluated over a termination dynamic basis, viz. a dynamic basis not showing recursive definitions or non-termination.

More formally, we call a dynamic basis ω a *termination dynamic basis* if all implementation retrieval functions $impl_\omega$ yield only implementation bodies in F_ω and all these implementation bodies do not result in a set containing $\bot$. In particular, the sets in the range of these bodies are all finite.

Proposition. *Let Ω be a static basis and ω a termination dynamic basis conforming to Ω; let Γ be a type environment and γ a variable environment over ω conforming to Γ; let t be a term in Constr $\cup$ Expr and $\tilde{t}$ an annotated term in A-Constr $\cup$ A-Expr; let $\tau \in T_\Omega$. If $\Omega; \Gamma \vdash t \triangleright \tilde{t} : \tau$ then there is a result $\overline{v} \in \overline{V}_\omega$ such that $\omega; \gamma \vdash \tilde{t} \downarrow \overline{v}$.*

Proof. By induction on the type derivation and using the failing convention for the operational rules. □

5 Denotational semantics

The big-step operational semantics for OCL terms described in the previous section gives rise to a denotational semantics, by endowing the semantic domains with suitable orderings and assigning an element of the

powerdomain of results to each OCL term. All operational rules applying to the same OCL term are gathered into a single clause defining the denotation of this term; each clause has to take into account the potential non-determinism of the operational evaluation process, undefined results, and non-termination. Moreover, dynamic basis extensions only show implementation bodies that are functions, that is, the expressions of all auxiliary pseudofeature definitions are replaced by its functional denotations.

The denotational semantics goes beyond the definition of the set-based denotational semantics by Richters and Gogolla [39], as non-determinism and non-termination is included. In particular, the denotation of a term is a set possibly including $\bot$ for non-termination rather than a single value.

5.1 Semantic domains

The denotational semantics is defined over the restricted class of *denotational dynamic bases*, i.e., all those dynamic bases ω showing only implementation bodies in F_ω; we denote by J_ω the set of implementations in I_ω with implementations bodies in F_ω.

We view $(\overline{V}_\omega)_\bot$ as a flat domain $((\overline{V}_\omega)_\bot, \sqsubseteq)$ with $\bot$ as the least element. We endow the powerdomain of results $\wp(\overline{V}_\omega)_\bot$ with the Egli-Milner ordering [35]

$$X \sqsubseteq Y \text{ if, and only if } (\forall x \in X \,.\, \exists y \in Y \,.\, x \sqsubseteq y) \wedge (\forall y \in Y \,.\, \exists x \in X \,.\, x \sqsubseteq y)\,;$$

the domain F_ω is equipped with the point-wise ordering, again written $\sqsubseteq$. Finally, the implementations with an implementation body in F_ω, i.e. J_ω, are ordered by $(\pi \equiv \psi) \sqsubseteq (\pi' \equiv \psi')$ if, and only if $\pi = \pi'$ and $\psi \sqsubseteq \psi'$. The least fixed-point operator on J_ω is denoted by $\mathbf{Y}$.

We use, besides the semantic functions in Sect. 4.1, some additional maps defined over a denotational dynamic basis ω, viz., liftings that extend the domain of functions from values and results to the powerdomain of results, a strict conditional, parallel boolean connectives, and an iterate functional.

The families of *liftings*

$$\bigcup_\omega f^{(n)}, \overline{\bigcup}_\omega g^{(n)} : (\wp(\overline{V}_\omega)_\bot)^n \to \wp(\overline{V}_\omega)_\bot$$

defined for all n-ary functions $f : (V_\omega)^n \to \wp(\overline{V}_\omega)_\bot$ and $g : (\overline{V}_\omega)^n \to \wp(\overline{V}_\omega)_\bot$ by

$$(\bigcup_\omega v_1 \cdots v_n \,.\, f) X_1 \cdots X_n = (\bigcup_{v_1 \in X_1 \cap V_\omega, \ldots, v_n \in X_n \cap V_\omega} f(v_1, \ldots, v_n)) \cup ((X_1 \cup \cdots \cup X_n) \cap \{\texttt{undef}, \bot\})$$

$$(\overline{\bigcup}_\omega v_1 \cdots v_n \,.\, g)\, X_1 \cdots X_n = (\bigcup_{\overline{v}_1 \in X_1 \cap \overline{V}_\omega, \ldots, \overline{v}_n \in X_n \cap \overline{V}_\omega} g(\overline{v}_1, \ldots, \overline{v}_n)) \cup (X_1 \cup \cdots \cup X_n) \cap \{\bot\})$$

respectively, lift functions on the semantic domains $(V_\omega)^n$ and $(\overline{V}_\omega)^n$ to functions on $(\wp(\overline{V}_\omega)_\bot)^n$, propagating undefined results and non-termination.

The *strict conditional* is defined by

$$cond_\omega : V_\omega \times (\overline{V}_\omega)_\bot \times (\overline{V}_\omega)_\bot \to (\overline{V}_\omega)_\bot$$

$$cond_\omega\, v\, r\, r' = \begin{cases} r, & \text{if } v = \texttt{true} \\ r', & \text{if } v = \texttt{false} \\ \bot, & \text{otherwise} \end{cases}$$

The *parallel boolean connectives* are defined by

$$\wedge_\omega, \vee_\omega : (\overline{V}_\omega)_\bot \times (\overline{V}_\omega)_\bot \to (\overline{V}_\omega)_\bot$$

$$r_1 \wedge_\omega r_2 = \begin{cases} r_1 \wedge r_2, \text{ if } r_1, r_2 \in \{\texttt{true}, \texttt{false}\} \\ \texttt{false}, \text{ if } r_1 = \texttt{false} \text{ or } r_2 = \texttt{false} \\ \texttt{undef}, \text{ if } r_1 = \texttt{undef} \text{ and } r_2 \notin \{\texttt{false}, \bot\} \text{ or} \\ \qquad\qquad r_1 \notin \{\texttt{false}, \bot\} \text{ and } r_2 = \texttt{undef} \\ \bot, \qquad \text{otherwise} \end{cases}$$

$$r_1 \vee_\omega r_2 = \begin{cases} r_1 \vee r_2, & \text{if } r_1, r_2 \in \{\texttt{true}, \texttt{false}\} \\ \texttt{true}, & \text{if } r_1 = \texttt{true} \text{ or } r_2 = \texttt{true} \\ \texttt{undef}, & \text{if } r_1 = \texttt{undef} \text{ and } r_2 \notin \{\texttt{true}, \bot\} \text{ or} \\ & \quad r_1 \notin \{\texttt{true}, \bot\} \text{ and } r_2 = \texttt{undef} \\ \bot, & \text{otherwise} \end{cases}$$

The family of *iterate functionals*

$$iterate_\omega - f\, \gamma : Var \times V_\omega^* \times Var \times V_\omega \to \wp(\overline{V}_\omega)_\bot$$

defined for all functions f from variable environments over ω to $\wp(\overline{V}_\omega)_\bot$ and for all variable environments γ over ω by

$$iterate_\omega\, x\, \emptyset\, x'\, v\, f\, \gamma = \{v\}$$
$$iterate_\omega\, x\, v_1 \cdots v_n\, x'\, v\, f\, \gamma = (\bigcup_\omega v' \,.\, iterate_\omega\, x\, v_2 \cdots v_n\, x'\, v'\, f\, \gamma)\, f(\gamma, x \mapsto v_1, x' \mapsto v)$$

iterates a function on variable environments over a sequence of values and accumulates previous results.

Finally, we interpret a variable environment γ over a dynamic basis ω as a partial function from variables to results, defining $\overline{\gamma}(x)$ as $\gamma(x)$ if $\gamma(x)$ is defined and $\overline{\gamma}(x) = \texttt{undef}$ otherwise. Likewise, the partial functions $impl_\omega(a_\tau, v)$ are extended to $\overline{impl}_\omega(a_\tau, v) = impl_\omega(a_\tau, v)$ if $impl_\omega(a_\tau, v)$ is defined and $\overline{impl}_\omega(a_\tau, v) = \{\texttt{undef}\}$ otherwise, and analogously for $impl_\omega(o_\tau, v, (v_i)_{1 \le i \le n})$.

5.2 Denotational equations

The denotational semantics is given by a family of functions

$$[\![-]\!]\, \omega\, \gamma : A\text{-}Term \to (\wp(\overline{V}_\omega)_\bot \cup J_\omega)$$

depending on a denotational dynamic basis ω and a variable environment γ over ω.

Table 8. Denotational semantics

$$(\mathrm{Spec}^{\in})\quad [\![(\texttt{context}\ \zeta_i\ \texttt{def:}\ \tilde{d}_i)_{1\le i\le n}\ \texttt{context}\ \zeta\ \texttt{inv:}\ \tilde{e}]\!]\,\omega\,\gamma = \bigcup_{v\in\omega(\zeta)}([\![\tilde{e}]\!]\,(\omega,(\iota_i)_{1\le i\le n})\,(\gamma,\texttt{self}\mapsto v)) \quad \text{where } (\iota_i)_{1\le i\le n} = \mathbf{Y}\lambda(\iota_i)_{1\le i\le n}\,.\,([\![\tilde{d}_i]\!]\,(\omega,(\iota_j)_{1\le j\le n})\,\gamma)_{1\le i\le n}$$

$$(\mathrm{Def}_1^{\in})\quad [\![a_\zeta\ \texttt{:}\ \tau\ \texttt{=}\ \tilde{e}]\!]\,\omega\,\gamma = \zeta.a \equiv \lambda v\,.\,[\![\tilde{e}]\!]\,\omega\,(\gamma,\texttt{self}\mapsto v)$$

$$(\mathrm{Def}_2^{\in})\quad [\![o_\zeta\texttt{(}x_1\ \texttt{:}\ \tau_1\texttt{,}\ \ldots\texttt{,}\ x_n\ \texttt{:}\ \tau_n\texttt{)}\ \texttt{:}\ \tau\ \texttt{=}\ \tilde{e}]\!]\,\omega\,\gamma = \zeta.o((x_i)_{1\le i\le n}) : \tau \equiv \lambda v\,.\,\lambda(v_i)_{1\le i\le n}\,.\,[\![\tilde{e}]\!]\,\omega\,(\gamma,\texttt{self}\mapsto v,(x_i\mapsto v_i))$$

$$(\mathrm{Const}^{\in})\quad [\![l]\!]\,\omega\,\gamma = \{l\}$$

$$(\mathrm{Self}^{\in})\quad [\![\texttt{self}]\!]\,\omega\,\gamma = \{\overline{\gamma}(\texttt{self})\}$$

$$(\mathrm{Var}^{\in})\quad [\![x]\!]\,\omega\,\gamma = \{\overline{\gamma}(x)\}$$

$$(\mathrm{Type}^{\in})\quad [\![\tau]\!]\,\omega\,\gamma = \{\tau\}$$

$$(\mathrm{Undef}^{\in})\quad [\![\texttt{undef}]\!]\,\omega\,\gamma = \{\texttt{undef}\}$$

$$(\mathrm{Coll}^{\in})\quad [\![\sigma\texttt{\{}\tilde{e}_1\texttt{,}\ \ldots\texttt{,}\ \tilde{e}_n\texttt{\}}]\!]\,\omega\,\gamma = (\textstyle\bigcup_\omega\, v_1\cdots v_n\,.\,\{make_\omega(\sigma,v_1\cdots v_n)\})\,([\![\tilde{e}_1]\!]\,\omega\,\gamma)\cdots([\![\tilde{e}_n]\!]\,\omega\,\gamma)$$

$$(\mathrm{Tup}^{\in})\quad [\![\texttt{Tuple\{}a_1\ \texttt{=}\ \tilde{e}_1\texttt{,}\ \ldots\texttt{,}\ a_n\ \texttt{=}\ \tilde{e}_n\texttt{\}}]\!]\,\omega\,\gamma = (\textstyle\bigcup_\omega\, v_1\cdots v_n\,.\,\{\texttt{Tuple\{}a_1\ \texttt{=}\ v_1\texttt{,}\ \ldots\texttt{,}\ a_n\ \texttt{=}\ v_n\texttt{\}}\})\ ([\![\tilde{e}_1]\!]\,\omega\,\gamma)\cdots([\![\tilde{e}_n]\!]\,\omega\,\gamma)$$

$$(\mathrm{Let}^{\in})\quad [\![\texttt{let}\ x\ \texttt{=}\ \tilde{e}\ \texttt{in}\ \tilde{e}')]\!]\,\omega\,\gamma = (\textstyle\bigcup_\omega\, v\,.\,[\![\tilde{e}']\!]\,\omega\,(\gamma,x\mapsto v))\,([\![\tilde{e}]\!]\,\omega\,\gamma)$$

$$(\mathrm{Cond}^{\in})\quad [\![\texttt{if}\ \tilde{e}\ \texttt{then}\ \tilde{e}_1\ \texttt{else}\ \tilde{e}_2\ \texttt{endif})]\!]\,\omega\,\gamma = (\textstyle\bigcup_\omega\, v\,.\,\{cond_\omega\ v\,r\,r' \mid r\in([\![\tilde{e}_1]\!]\,\omega\,\gamma), r'\in([\![\tilde{e}_2]\!]\,\omega\,\gamma)\})\,([\![\tilde{e}]\!]\,\omega\,\gamma)$$

$$(\mathrm{And}^{\in})\quad [\![\tilde{e}_1\ \texttt{and}\ \tilde{e}_2]\!]\,\omega\,\gamma = \{r_1\wedge_\omega r_2 \mid r_1\in([\![\tilde{e}]\!]\,\omega\,\gamma), r_2\in([\![\tilde{e}_2]\!]\,\omega\,\gamma)\}$$

$$(\mathrm{Or}^{\in})\quad [\![\tilde{e}_1\ \texttt{or}\ \tilde{e}_2]\!]\,\omega\,\gamma = \{r_1\vee_\omega r_2 \mid r_1\in([\![\tilde{e}]\!]\,\omega\,\gamma), r_2\in([\![\tilde{e}_2]\!]\,\omega\,\gamma)\}$$

$$(\mathrm{Iter}^{\in})\quad [\![\tilde{e}\texttt{->iterate(}x\texttt{;}\ x'\ \texttt{=}\ \tilde{e}'\ \texttt{|}\ \tilde{e}''\texttt{)}]\!]\,\omega\,\gamma = (\textstyle\bigcup_\omega\, v\,v_0'\,\cdot\bigcup_{v\rightsquigarrow\texttt{Sequence\{}v_1\texttt{,}\ \ldots\texttt{,}\ v_n\texttt{\}}} iterate_\omega\,x\,v_1\cdots v_n\,x'\,v_0'\,([\![\tilde{e}'']\!]\,\omega)\,\gamma)\ ([\![\tilde{e}]\!]\,\omega\,\gamma)\,([\![\tilde{e}']\!]\,\omega\,\gamma)$$

$$(\mathrm{Cast}^{\in})\quad [\![\tilde{e}\texttt{.asType(}\tau\texttt{)}]\!]\,\omega\,\gamma = (\textstyle\bigcup_\omega\, v\,.\,\{\overline{v}' \mid (v :_\omega \tau\wedge\overline{v}'=v)\vee(v\not:_\omega\tau\wedge\overline{v}'=\texttt{undef})\})\,([\![\tilde{e}]\!]\,\omega\,\gamma)$$

$$(\mathrm{Inst}^{\in})\quad [\![\tau\texttt{.allInstances()}]\!]\,\omega\,\gamma = \{make_\omega(\texttt{Set},\omega(\tau))\}$$

$$(\mathrm{IsUndef}^{\in})\quad [\![\tilde{e}\texttt{.isUndef()}]\!]\,\omega\,\gamma = (\overline{\textstyle\bigcup}_\omega\,\overline{v}\,.\,\{v \mid (\overline{v}=\texttt{undef}\wedge v=\texttt{true})\vee(\overline{v}\neq\texttt{undef}\wedge v=\texttt{false})\})\,[\![\tilde{e}]\!]\,\omega\,\gamma$$

$$(\mathrm{Feat}_1^{\in})\quad [\![\tilde{e}.a_\tau]\!]\,\omega\,\gamma = (\textstyle\bigcup_\omega\, v\,.\,\overline{impl}_\omega(a_\tau,v))\,([\![\tilde{e}]\!]\,\omega\,\gamma)$$

$$(\mathrm{Feat}_2^{\in})\quad [\![\tilde{e}.o_\tau\texttt{(}\tilde{e}_1\texttt{,}\ \ldots\texttt{,}\ \tilde{e}_n\texttt{)}]\!]\,\omega\,\gamma = (\textstyle\bigcup_\omega\, v\,v_1\cdots v_n\,.\,\overline{impl}_\omega(o_\tau,v,(v_i)_{1\le i\le n}))\ ([\![\tilde{e}]\!]\,\omega\,\gamma)\,([\![\tilde{e}_1]\!]\,\omega\,\gamma)\cdots([\![\tilde{e}_n]\!]\,\omega\,\gamma)$$

$$(\mathrm{Prop}^{\in})\quad [\![\tilde{e}\texttt{->}o_\tau\texttt{(}\tilde{e}_1\texttt{,}\ \ldots\texttt{,}\ \tilde{e}_n\texttt{)}]\!]\,\omega\,\gamma = (\textstyle\bigcup_\omega\, v\,v_1\cdots v_n\,.\,\overline{impl}_\omega(o_\tau,v,(v_i)_{1\le i\le n}))\ ([\![\tilde{e}]\!]\,\omega\,\gamma)\,([\![\tilde{e}_1]\!]\,\omega\,\gamma)\cdots([\![\tilde{e}_n]\!]\,\omega\,\gamma)$$

The definition of $[\![-]\!]\,\omega\,\gamma$ is stated in Table 8. The metavariables range as follows: $l \in \mathit{Literal}$; $\zeta \in C_\omega$, $\sigma \in S$, $\tau \in T_\omega$; $x \in \mathit{Var}$; $a, o \in \mathit{Name}$; $v \in V_\omega$, $\overline{v} \in \overline{V}_\omega$, $r \in (\overline{V}_\omega)_\perp$, $\iota \in J_\omega$; $\tilde{e} \in \mathit{A\text{-}Expr}$, $\tilde{d} \in \mathit{A\text{-}AttrDef} \cup \mathit{OpDef}$.

Given an ascending chain $(\iota_{i,0})_{1\le i\le n} \sqsubseteq (\iota_{i,1})_{1\le i\le n} \sqsubseteq (\iota_{i,2})_{1\le i\le n} \sqsubseteq \ldots$ of implementation tuples, we have that $\bigsqcup_n [\![\tilde{e}]\!]\,(\omega, (\iota_{i,n})_{1\le i\le n})\,\gamma = [\![\tilde{e}]\!]\,(\omega, \bigsqcup_n (\iota_{i,n})_{1\le i\le n})\,\gamma$. Therefore, the denotational equations are well defined. In particular, the functional $\lambda(\iota_i)_{1\le i\le n}\,.\,([\![\tilde{d}_i]\!]\,(\omega, (\iota_j)_{1\le j\le n})\,\gamma)_{1\le i\le n}$ used in ($\mathrm{Spec}^{\in}$) is continuous.

5.3 Adequacy

By its very construction, the denotational and the operational semantics coincide on well-typed OCL terms over a denotational dynamic basis. Note that every termination dynamic basis is also a denotational dynamic basis, but the converse does not hold. For instance, evaluation of the first term evaluation example in Sect. 4.2 leads to a denotational dynamic basis, but `g` does not terminate, and thus this dynamic basis does not form a termination dynamic basis.

In order to state and prove that the denotational semantics is adequate with respect to the operational semantics we define the *extensional equality* on implementations over a dynamic basis ω by:

1. $(\zeta.a \equiv \tilde{e}) = (\zeta.a \equiv f)$ with $\tilde{e} \in \mathit{A\text{-}Term}$ and $f \in F_\omega$ in case $\omega; \gamma, \mathtt{self} \mapsto v \vdash \tilde{e} \downarrow \overline{v}$ if, and only if $\overline{v} \in f(v)$
2. $(\zeta.o((x_i)_{1\le i\le n}) : \tau \equiv \tilde{e}) = (\zeta.o((x_i)_{1\le i\le n}) \equiv f)$ with $\tilde{e} \in \mathit{A\text{-}Term}$ and $f \in F_\omega$ in case $\omega; \gamma, \mathtt{self} \mapsto v, (x_i \mapsto v_i)_{1\le i\le n} \vdash \tilde{e} \downarrow \overline{v}$ if, and only if $\overline{v} \in f(v, (v_i)_{1\le i\le n})$

Proposition. *Let Ω be a static basis and ω a denotational 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 $\theta \in T_\Omega \cup D_\Omega$ and $\rho \in \overline{V}_\omega \cup I_\omega$. If $\Omega; \Gamma \vdash t \triangleright \tilde{t} : \theta$, then $\omega; \gamma \vdash \tilde{t} \downarrow \delta$ if, and only if either $\rho \in [\![\tilde{t}]\!]\,\omega\,\gamma$ and $\rho \in \overline{V}_\omega$ or $\rho = [\![\tilde{t}]\!]\,\omega\,\gamma$ and $\rho \in I_\omega$.*

Proof. By structural induction on $\tilde{t}$, see Appendix B. □

6 Expressiveness

Mandel and Cengarle [26] have argued that all primitive recursive functions can be encoded as OCL expressions. We prove the reverse direction of this observation: that the evaluation of a well-typed OCL constraint or expression, i.e. a well-typed OCL term not showing pseudofeature definitions, over a static basis and a denotational dynamic basis conforming to the static basis, and only showing primitive recursive implementation bodies, is primitive recursive. In particular, the evaluation of a well-typed OCL expression over the static basis corresponding to an empty UML static structure is primitive recursive.

Since every primitive recursive function can be represented by an OCL expression, a function evaluating all OCL expressions cannot be primitive recursive [22]. However, for every OCL term the denotational semantics in Sect. 5 defines a function that evaluates the term and is moreover primitive recursive. For example, the OCL expression

```
Sequence{x, y}->iterate(i; a = 0 | a.+(i))
```

encoding addition of `x` and `y` denotes the following function parameterised over variable environments γ:

$$
\begin{aligned}
&[\![\texttt{Sequence\{x, y\}->iterate(i; a = 0 | a.+}_{\texttt{Integer}}\texttt{(i))}]\!]\,\omega\,\gamma = \\
&(\textstyle\bigcup_\omega\, v\, v_0' \,.\, \bigcup_{v \rightsquigarrow \texttt{Sequence}\{v_1,\,\ldots,\,v_n\}} \mathit{iterate}_\omega\ \mathtt{i}\, v_1 \cdots v_n\, \mathtt{a}\, v_0' \\
&([\![\texttt{a.+}_{\texttt{Integer}}\texttt{(i)}]\!]\,\omega)\,\gamma)([\![\texttt{Sequence\{x, y\}}]\!]\,\omega\,\gamma)\,([\![\texttt{0}]\!]\,\omega\,\gamma) = \\
&(\textstyle\bigcup_\omega\, v\, v_0' \,.\, \bigcup_{v \rightsquigarrow \texttt{Sequence}\{v_1,\,\ldots,\,v_n\}} \mathit{iterate}_\omega\ \mathtt{i}\, v_1 \cdots v_n\, \mathtt{a}\, v_0' \\
&(\lambda\gamma' \,.\, (\textstyle\bigcup_\omega\, v\, v_1 \,.\, \overline{\mathit{impl}}_\omega(\texttt{+}_{\texttt{Integer}}, v, v_1))\,([\![\texttt{a}]\!]\,\omega\,\gamma')\,([\![\texttt{i}]\!]\,\omega\,\gamma'))\,\gamma) \\
&((\textstyle\bigcup_\omega\, v_1\, v_2 \,.\, \{\mathit{make}_\omega(\texttt{Sequence}, v_1\, v_2)\})\,([\![\texttt{x}]\!]\,\omega\,\gamma)\,([\![\texttt{y}]\!]\,\omega\,\gamma)) \\
&([\![\texttt{0}]\!]\,\omega\,\gamma) = \\
&(\textstyle\bigcup_\omega\, v\, v_0' \,.\, \bigcup_{v \rightsquigarrow \texttt{Sequence}\{v_1,\,\ldots,\,v_n\}} \mathit{iterate}_\omega\ \mathtt{i}\, v_1 \cdots v_n\, \mathtt{a}\, v_0' \\
&(\lambda\gamma' \,.\, (\textstyle\bigcup_\omega\, v\, v_1 \,.\, \overline{\mathit{impl}}_\omega(\texttt{+}_{\texttt{Integer}}, v, v_1))\,\{\overline{\gamma}'(\mathtt{a})\}\,\{\overline{\gamma}'(\mathtt{i})\})\,\gamma) \\
&((\textstyle\bigcup_\omega\, v_1\, v_2 \,.\, \{\mathit{make}_\omega(\texttt{Sequence}, v_1\, v_2)\})\,\{\overline{\gamma}(\mathtt{x})\}\,\{\overline{\gamma}(\mathtt{y})\}\,\})\,\{\texttt{0}\}\,.
\end{aligned}
$$

The variable environment represents parameter passing.

In order to state and prove this result generally, we call a denotational dynamic basis ω *primitive recursive* if all the functions $\mathit{impl}_\omega(a, \tau, v)$ and $\mathit{impl}_\omega(o, \tau, v, (v_i)_{1\le i\le n})$, defined in ω, and the relation $:_\omega$ are primitive recursive.

Proposition. *Let Ω be a static basis and ω a primitive recursive denotational dynamic basis conforming to Ω; let Γ be a type environment over Ω; let t be a term in Constr ∪ Expr and $\tilde{t}$ be an annotated term in A-Constr ∪ A-Expr; let $\tau \in T_\Omega$. If $\Omega; \Gamma \vdash t \triangleright \tilde{t} : \tau$, then there is a primitive recursive function $\mathit{eval}(\tilde{t})\,\omega\,\gamma$ from variable environments over ω conforming to Γ to $\wp(\overline{V}_\omega)_\perp$ such that $\mathit{eval}(\tilde{t})\,\omega\,\gamma = [\![\tilde{t}]\!]\,\omega\,\gamma$.*

Proof. Using suitable encodings for variable environments over ω, $(\overline{V}_\omega)_\perp$, and finite sequences and sets of $(\overline{V}_\omega)_\perp$, all auxiliary functions used in the definition of $[\![\tilde{t}]\!]\,\omega\,\gamma$ are primitive recursive. Thus, by induction over the term structure of $\tilde{t}$, all functions in $[\![\tilde{t}]\!]\,\omega$ are primitive recursive. Hence, we may define $\mathit{eval}(\tilde{t})\,\omega\,\gamma = [\![\tilde{t}]\!]\,\omega\,\gamma$. □

Any denotational dynamic basis conforming to the static basis for the empty UML static structure can obviously be chosen to be primitive recursive. Thus, every OCL expression that is well typed over the static basis for the empty UML static structure, i.e., a pure OCL expression, denotes a primitive recursive function. However, the `def:` clause allows the definition of arbitrary recursive functions. Therefore, this syntactic enhancement considerably enlarges the expressiveness of OCL.

7 Conclusions

We have presented a type inference system, a big-step operational semantics, and a denotational semantics for OCL 1.4/2.0 including the possibility of defining additional pseudofeatures. The operational semantics satisfies a subject reduction property with respect to the type inference system, the denotational semantics coincides with the operational semantics on well-typed terms. The corrections and additions to previous formal approaches to OCL 1.1/3/4 are pervasive. Moreover, we have studied the expressiveness of OCL by showing that all pure OCL expressions are primitive recursive.

The semantics of OCL terms relies on the semantics of the underlying UML model. We have abstractly axiomatised UML static structures and UML object models, stating only some sufficient conditions, such that OCL terms can be uniquely typed and type-safely evaluated. However, it seems desirable to define a single, agreed-upon semantics of UML static structures and object models in order to further the prospect of tool-support. Such a semantics may restrict overriding of operations to a co-/contra-variance scheme and may also show a tighter support for UML templates (cf. the treatment by Clark [11]), association classes and qualified association ends (see the equivalence rules by Gogolla and Richters [19]).

We expect our semantics to be properly integrable with the existing semantics for OCL pre- and post-conditions by Clark [11] or Gogolla and Richters [37, 39], which amounts, roughly speaking, to interpreting OCL post-conditions over two dynamic bases. Again, a clear understanding of the interplay between OCL pre-/post-condition specifications and UML inheritance is indispensable (see [21, 34]). Furthermore, the type inference system may be useful for alternative, graphical representations of OCL constraints, as investigated by Kent and Howse [24] or Bottoni et al. [8].

The operational interpretation of OCL constraints is complemented by treating OCL as a logic. On the one hand, Beckert, Keller, and Schmitt suggest to translate OCL, however, omitting pseudofeature definitions, `iterate`, and `undef`, into first-order predicate logic [5], thus providing an immediate to use base for theorem provers. Schmitt [41] also investigates the expressiveness of the `iterate` construct in a first-order logic and finite models showing some connections with a transitive closure operator. Moreover, Baar, Beckert, and Schmitt propose to interpret the `@pre` modality in a dynamic logic [3]. On the other hand, pseudofeature declarations may be viewed as additional implementation constraints by a fixed-point interpretation [9]. The study of the precise relation between the operational and the logical view of OCL remains to be explored.

References

1. Ahrendt W, Baar T, Beckert B, Giese M, Hähnle R, Menzel W, Mostowski W, Schmitt PH (2002) The KeY System: Integrating Object-Oriented Design and Formal Methods. In: Ojeda-Aciego M, Pérez de Guzmán I, Brewka G, Moniz Pereira L (eds) Europ. Wsh. Logics in Artifical Intelligence, Lect. Notes Artif. Intell., vol. 1919. Springer, Berlin, pp 21–36
2. Baar T (2000) Experiences with the UML/OCL-Approach to Precise Software Modeling. In: Proc. Net.ObjectDays, Erfurt.
   `http://i12www.ira.uka.de/~key/doc/2000/baar00.pdf.gz`
3. Baar T, Beckert B, Schmitt PH (2001) An Extension of Dynamic Logic for Modelling OCL's `@pre` Operator. In: Bjørner D, Broy M, Zamulin AV (eds) Proc. 4th Andrei Ershov Int. Conf. Perspectives of System Informatics, Lect. Notes Comp. Sci., vol. 2244. Springer, Berlin, pp 47–54
4. Baar T, Hähnle R (2000) An Integrated Metamodel for OCL Types. In: France R (ed) Proc. OOPSLA'2000 Wsh. Refactoring the UML: In Search of the Core, Minneapolis
5. Beckert B, Keller U, Schmitt PH (2003) Translating the Object Constraint Language into First-order Predicate Logic. (Submitted for publication)
6. Bickford M, Guaspari D (1998) Lightweight Analysis of UML. Draft NAS1-20335/10, Odyssey Research Assoc.
   `http://cgi.omg.org/cgi-bin/doc?ad/98-10-01`
7. Booch G, Rumbaugh J, Jacobson I (1998) The Unified Modeling Language User Guide. Addison–Wesley, Reading, Mass.
8. Bottoni P, Koch M, Parisi-Presicce F, Taentzer G (2000) Consistency Checking and Visualization of OCL Constraints. In: Evans et al. [17], pp 294–308
9. Cengarle MV, Knapp A (2001) A Formal Semantics for OCL 1.4. In: Gogolla M, Kobryn C (eds) Proc. 4th Int. Conf. UML, Lect. Notes Comp. Sci., vol. 2185. Springer, Berlin, pp 118–133
10. Cengarle MV, Knapp A (2001) On the Expressive Power of Pure OCL. Technical Report 0101, Ludwig–Maximilians–Universität München
11. Clark T (1999) Type Checking UML Static Diagrams. In: France and Rumpe [18], pp 503–517
12. Clark T, Evans A, Kent S (2000) Meta-Modelling Tool. `http://www.cs.york.ac.uk/puml/mmf/mmt.zip`
13. Clark T, Warmer J (2000) (eds) Proc. UML'2000 Wsh. UML 2.0 – The Future of OCL, York
14. Clark T, Warmer J (2002) (eds) Advances in Object Modelling with the OCL, Lect. Notes Comp. Sci., vol. 2263. Springer, Berlin
15. Drossopoulou S, Eisenbach S (1999) Describing the Semantics of Java and Proving Typing Soundness. In: Alves-Foss J (ed) Formal Syntax and Semantics of Java, Lect. Notes Comp. Sci., vol. 1523. Springer, Berlin, pp 41–82
16. D'Souza DF, Wills AC (1998) Object, Components, Frameworks with UML: The Catalysis Approach. Addison-Wesley, Reading, Mass.
17. Evans A, Kent S, Selic B (eds) (2000) Proc. 3rd Int. Conf. UML, Lect. Notes Comp. Sci., vol. 1939. Springer, Berlin
18. France R, Rumpe B (1999) (eds) Proc. 2nd Int. Conf. UML, Lect. Notes Comp. Sci., vol. 1723. Springer, Berlin
19. Gogolla M, Richters M (1998) Equivalence Rules for UML Class Diagrams. In: Bézivin J, Muller P-A (eds) Proc. 1st Int. Wsh. UML. Mulhouse, pp 87–96
20. Hami A, Howse J, Kent S (1998) Interpreting the Object Constraint Language. In: Proc. Asia Pacific Conf. Software Engineering. IEEE Press
21. Hennicker R, Hußmann H, Bidoit M (2002) On the Precise Meaning of OCL Constraints. In: Clark and Warmer [14], pp 70–85
22. Hermes H (1978) Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit, Heidelberger Taschenbücher, vol. 87. Springer, Berlin Heidelberg New York, 3rd edition
23. Hußmann H, Demuth B, Finger F (2000) Modular Architecture for a Toolset Supporting OCL. In: Evans et al. [17], pp 278–293

24. Kent S, Howse J (1999) Mixing Visual and Textual Constraint Languages. In: France and Rumpe [18], pp 384–398
25. Lano K (1995) Formal Object-Oriented Development. Formal Approaches to Computing and Information Technology. Springer, London
26. Mandel L, Cengarle MV (1999) On the Expressive Power of OCL. In: Wing JM, Woodcock J, Davies J (eds) Proc. World Congress Formal Methods, Vol. 1, Lect. Notes Comp. Sci., vol. 1708. Springer, Berlin, pp 854–874
27. Milner R, Tofte M, Harper R, MacQueen D (1997) The Definition of Standard ML (Revised). MIT Press, Cambridge, Mass.
28. Mitchell JC (1996) Foundations for Programming Languages. Foundations of Computing. MIT Press, Cambridge, Mass.
29. Object Management Group (1997) Unified Modeling Language Specification, Version 1.1. Technical report, OMG. `http://www.omg.org/cgi-bin/doc?ad/97-08-04`
30. Object Management Group (1999) Unified Modeling Language Specification, Version 1.3. Technical report, OMG. `http://www.omg.org/cgi-bin/doc?ad/99-06-08`
31. Object Management Group (2001) Unified Modeling Language Specification, Version 1.4. Specification, OMG. `http://www.omg.org/cgi-bin/doc?formal/01-09-67`
32. Object Management Group (2003) Unified Modeling Language Specification, Version 1.5. Specification, OMG. `http://www.omg.org/cgi-bin/doc?formal/03-03-01`
33. Response to OMG RfP ad/00-09-03 "UML 2.0 OCL" (2003) 2nd revised submission, OMG. `http://www.omg.org/cgi-bin/doc?ad/03-01-07`
34. Reus B, Wirsing M, Hennicker R (2001) A Hoare Calculus for Verifying Java Realizations of OCL-Constrained Design Models. In: Hußmann H (ed) Proc. 4th Int. Conf. Fundamental Approaches to Software Engineering, Lect. Notes Comp. Sci., vol. 2029. Springer, Berlin, pp 300–317
35. Reynolds JC (1998) Theories of Programming Languages. Cambridge University Press, Cambridge
36. Richters M, Gogolla M (1998) On Formalizing the UML Object Constraint Language OCL. In: Wang Ling T, Ram S, Li Lee M (eds) Proc. 17th Int. Conf. Conceptual Modeling, Lect. Notes Comp. Sci., vol. 1507. Springer, Berlin, pp 449–464
37. Richters M, Gogolla M (2000) A Semantics for OCL Pre- and Postconditions. In: Clark and Warmer [13]
38. Richters M, Gogolla M (2000) Validating UML Models and OCL Constraints. In: Evans et al. [17], pp 265–277
39. Richters M, Gogolla M (2002) OCL – Syntax, Semantics and Tools. In: Clark and Warmer [14], pp 38–63
40. Schmitt PH (2001) A Model Theoretic Semantics of OCL. In: Beckert B, France R, Hähnle R, Jacobs B (eds) Proc. Wsh. Precise Modelling and Deduction for Object-Oriented Software Development, Technical Report DII 07/01. Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Siena, pp 43–57
41. Schmitt PH (2001) Iterate Logic. In: Schroeder-Heister P, Stärk R, Kahle R (eds) Proc. Int. Sem. Proof Theory in Computer Science, Lect. Notes Comp. Sci., vol. 2183. Springer, Berlin, pp 191–201
42. Schürr A (2000) New Type Checking Rules for OCL (Collection) Expressions. In: Clark and Warmer [13]
43. Warmer J, Kleppe A (1999) The Object Constraint Language. Addison-Wesley, Reading, Mass.

Appendices

A Proof of subject reduction

Proposition. *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 $\theta \in T_\Omega \cup D_\Omega$ and $\rho \in V_\omega \cup I_\omega$. If $\Omega;\Gamma \vdash t \triangleright \tilde{t} : \theta$ and $\omega;\gamma \vdash \tilde{t} \downarrow \rho$, then $\rho :_\omega \theta$.*

Proof. By induction on the height of the proof tree of $\Omega;\Gamma \vdash t \triangleright \tilde{t} : \theta$.
If $n = 0$, nothing has to be proved. Thus, let $n > 0$ and let the claim be proved for all proof tree heights $n' < n$. We proceed by case analysis on the last proof step for $\Omega;\Gamma \vdash t \triangleright \tilde{t} : \theta$; in the following, we treat only some exemplary cases:

(Spec$^\cdot$): Then $\tilde{t} = (\texttt{context}\ \zeta_i\ \texttt{def:}\ \tilde{d_i})_{1\le i\le n}\ \texttt{context}\ \zeta\ \texttt{inv:}\ \tilde{e}_i$ and $\theta = \texttt{Boolean}$ with $(\Omega, (\delta_j)_{1\le j\le n}; \Gamma, \texttt{self} : \zeta_i \vdash \tilde{e}_i : \delta_i)_{1\le i\le n}$ and $\Omega, (\delta_j)_{1\le j\le n}; \Gamma, \texttt{self} : \zeta \vdash \tilde{e} : \texttt{Boolean}$. The only applicable operational rule for such a $\tilde{t}$ is (Spec$^\downarrow$) and, thus, if we have $\omega, (\iota_i)_{1\le i\le n}; \gamma \vdash \tilde{t} \downarrow v$ then necessarily $\omega, (\iota_j)_{1\le i\le n} \vdash \tilde{d_i} \downarrow \iota_i$ for $1 \le i \le n$ and $\omega, (\iota_j)_{1\le j\le n}; \gamma, \texttt{self} \mapsto v' \vdash \tilde{e} \downarrow v_{v'}$ for all $v' \in \omega(\zeta)$ with $v = \bigwedge_{v'} v_{v'}$. But then $\omega, (\iota_i)_{1\le i\le n}$ conforms to $\Omega, (\delta_i)_{1\le i\le n}$ and $\gamma, \texttt{self} \mapsto v'$ conforms to $\Gamma, \texttt{self} : \zeta$ for all $v' \in \omega(\zeta)$. Since $v_{v'} \in V_\omega$ for all v' and there is no value v'' with $v'' :_\omega \texttt{Void}$ and `Void` is the only sub-type of `Boolean`, we have $v_{v'} :_\omega \texttt{Boolean}$ by the induction hypothesis and, a fortiori, $v :_\omega \texttt{Boolean} = \theta$.

(Def$_1^\cdot$): Then $\tilde{t} = x_\zeta\ \texttt{:}\ \tau\ \texttt{=}\ \tilde{e}$ and $\theta = \zeta.x : \tau$ with $\Omega;\Gamma \vdash \tilde{e} : \tau'$, $\Gamma(\texttt{self}) = \zeta$, and $\tau' \le_\Omega \tau$. The only applicable operational rule for such a $\tilde{t}$ is (Def$_1^\downarrow$) and, thus, if we have $\omega;\gamma \vdash \tilde{t} \downarrow \iota$, then $\iota = \zeta.x \equiv \tilde{e}$. Since $\tau' \le_\Omega \tau$, we have $\iota :_\omega \zeta.x : \tau = \theta$.

(Self$^\cdot$): Then $\tilde{t} = \texttt{self}$ and $\theta = \Gamma(\texttt{self})$. The only applicable operational rule for such a $\tilde{t}$ is (Self$^\downarrow$) and, thus, if we have $\omega;\gamma \vdash \texttt{self} \downarrow v$, then $v = \gamma(\texttt{self})$; but, then, the claim is clear, for γ conforms to Γ and thus we have $\gamma(\texttt{self}) :_\omega \Gamma(\texttt{self}) = \theta$.

(Coll$^\cdot$): Then $\tilde{t} = \sigma\texttt{\{}\tilde{e}_1\texttt{, ..., }\tilde{e}_n\texttt{\}}$ and $\theta = \sigma\texttt{(}\tau\texttt{)}$ with $\tau = \bigsqcup_\Omega\{\tau_i \mid 1 \le i \le n\}$, and $\Omega;\Gamma \vdash \tilde{e}_i : \tau_i$ for $1 \le i \le n$. The only applicable operational rule for such a $\tilde{t}$ is (Coll$^\downarrow$) and, thus, if we have $\omega;\gamma \vdash \tilde{t} \downarrow v$ then necessarily $\omega;\gamma \vdash \tilde{e}_i \downarrow v_i$ for $1 \le i \le n$ with $v = \mathit{make}_\omega(\sigma, \emptyset) = \sigma\texttt{\{\}}$ if $n = 0$, and $v = \mathit{make}_\omega(\sigma, v_1 \cdots v_n)$, otherwise. If $n = 0$ then $v :_\omega \sigma\texttt{(Void)}$ and thus $v :_\omega \sigma\texttt{(}\tau\texttt{)}$. However, if $n > 0$, by the induction hypothesis, $v_i :_\omega \tau_i$ and thus $v_i :_\omega \tau'$ for all $1 \le i \le n$. Hence we have $v :_\omega \sigma\texttt{(}\tau\texttt{)} = \theta$.

(Let$^\cdot$): Then $\tilde{t} = \texttt{let}\ x\ \texttt{=}\ \tilde{e}\ \texttt{in}\ \tilde{e}'$ and $\theta = \tau'$ with $\Omega;\Gamma \vdash \tilde{e} : \tau$ and $\Omega;\Gamma, x : \tau_x \vdash \tilde{e}' : \tau'$ with $\tau \le_\Omega \tau_x$. The only applicable operational rule for such a $\tilde{t}$ is (Let$^\downarrow$) and, thus, if we have $\omega;\gamma \vdash \tilde{t} \downarrow v$ then necessarily $\omega;\gamma \vdash \tilde{e} \downarrow v'$ and $\omega;\gamma, x \mapsto v' \vdash \tilde{e}' \downarrow v$. By the induction hypothesis, $v' :_\omega \tau$ and thus $v' :_\omega \tau_x$, and, since hence $\gamma, x \mapsto v'$ conforms to $\Gamma, x : \tau_x$, also $v :_\omega \tau' = \theta$.

(Cast$^\cdot$): Then $\tilde{t} = e\texttt{.asType(}\tau\texttt{)}$ and $\theta = \tau$ with $\Omega;\Gamma \vdash \tilde{e} : \tau'$ and $\tau' \le_\Omega \tau$ or $\tau \le_\Omega \tau'$. The only applicable rule for such a $\tilde{t}$ is (Cast$^\downarrow$) and, thus, if we have $\omega;\gamma \vdash \tilde{t} \downarrow v$ then necessarily $\omega;\gamma \vdash \tilde{e} \downarrow v$ with $v :_\omega \tau = \theta$.

(Feat$_1^{\downarrow}$): Then $\tilde{t} = \tilde{e}.a_\tau$ and $\theta = \tau'$ with $\Gamma \vdash_\Omega \tilde{e} : \upsilon$ and $fd_\Omega(a, \upsilon) = \tau.a : \tau'$. There are two applicable rules for such a $\tilde{t}$: (Feat$_1^{\downarrow}$) and (Feat$_2^{\downarrow}$) with $\omega; \gamma \vdash \tilde{e}.a_\tau \downarrow v$.
If (Feat$_1^{\downarrow}$) has been applied, then we have $\omega; \gamma \vdash \tilde{e} \downarrow v'$ with $impl_\omega(a_\tau, v') = \tau.a \equiv v$. By the induction hypothesis $v' :_\omega \tau$ and since $(\tau.a \equiv v) :_\omega (\tau.a : \tau')$, we also have $v :_\omega \tau' = \theta$.
If (Feat$_2^{\downarrow}$) has been applied, then we necessarily have $\omega; \gamma \vdash \tilde{e} \downarrow v'$ and $\omega; \mathtt{self} \mapsto v' \vdash \tilde{e}' \downarrow v$ and $impl_\omega(a_\tau, v') = \tau.a \equiv \tilde{e}'$. By the induction hypothesis $v' :_\omega \tau$ and since $(\tau.a \equiv \tilde{e}') :_\omega (\tau.a : \tau')$, we also have $\Omega; \mathtt{self} : \tau \vdash \tilde{e}' : \tau''$ with $\tau'' \leq_\omega \tau'$. Since $\mathtt{self} \mapsto v'$ conforms to $\mathtt{self} : \tau$, again applying the induction hypothesis, $v :_\omega \tau'' \leq_\omega \tau' = \theta$.

(Short$_2^{\downarrow}$): Then $\tilde{t} = \tilde{e}\mathtt{->iterate(i; a = } \tilde{e}' \mathtt{ \mid } \tilde{e}''\mathtt{)}$ and $\theta = \sigma(\tau_0')$ with $\tilde{e}' = \sigma\mathtt{\{\}}$, $\tilde{e}'' = \mathtt{a->including}_{\sigma(\tau_0')}\mathtt{(i.}o_{\tau_0'}\mathtt{(}\tilde{e}_1\mathtt{, \ldots, } \tilde{e}_n\mathtt{))}$ and $\Omega; \Gamma \vdash \tilde{e} : \overline{\sigma}(\tau)$, $\Omega; \Gamma \vdash e_i \triangleright \tilde{e}_i : \tau_i$ for $1 \leq i \leq n$, and $fd_\Omega(o, \tau, (\tau_i)_{1\leq i\leq n}) = \tau'.o((\tau_i')_{1\leq i\leq n}) \to \tau_0'$. The only applicable rule for such a $\tilde{t}$ is (Iter$^{\downarrow}$) and, thus, if we have $\omega; \gamma \vdash \tilde{t} \downarrow v$ then necessarily $\omega; \gamma \vdash \tilde{e} \downarrow v'$ and $\omega; \gamma \vdash \tilde{e}' \downarrow v_0''$ and $\omega; \gamma, \mathtt{i} \mapsto v_i', \mathtt{a} \mapsto v_{i-1}'' \vdash \tilde{e}'' \downarrow v_i''$ for $1 \leq i \leq n$ where $v' \rightsquigarrow \mathtt{Sequence\{}v_1', \ldots, v_n'\mathtt{\}}$ with $v = v_n''$. By the induction hypothesis $v' :_\omega \overline{\sigma}(\tau)$. Moreover, $\sigma\mathtt{\{\}} :_\omega \sigma(\mathtt{Void}) \leq_\Omega \sigma(\tau_0')$ and $\Omega; \Gamma, \mathtt{i} : \tau, \mathtt{a} : \sigma(\tau_0') \vdash e'' \triangleright \tilde{e}'' : \sigma(\tau_0')$ for $e'' = \mathtt{a->including(i.}o(e_1, \ldots, e_n)\mathtt{)}$ by the assumptions in Table 2. Thus by the induction hypothesis, inductively, $v_i'' :_\omega \sigma(\tau_0')$ for $1 \leq i \leq n$, since hence $\gamma, \mathtt{i} \mapsto v_i', \mathtt{a} \mapsto v_{i-1}''$ conforms to $\Gamma, \mathtt{i} : \tau, \mathtt{a} : \sigma(\tau_0')$. In particular, $v :_\omega \sigma(\tau_0') = \theta$. □

B Proof of adequacy

Proposition. *Let Ω be a static basis and ω a denotational 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 $\theta \in T_\Omega \cup D_\Omega$ and $\rho \in \overline{V}_\omega \cup I_\omega$. If $\Omega; \Gamma \vdash t \triangleright \tilde{t} : \theta$, then $\omega; \gamma \vdash \tilde{t} \downarrow \delta$ if, and only if either $\rho \in [\![\tilde{t}]\!]\,\omega\,\gamma$ and $\rho \in \overline{V}_\omega$ or $\rho = [\![\tilde{t}]\!]\,\omega\,\gamma$ and $\rho \in I_\omega$.*

Proof. By structural induction on $\tilde{t}$. We show only some exemplary cases:

Let $\tilde{t} = (\mathtt{context}\ \zeta_i\ \mathtt{def:}\ \tilde{e}_i)_{1\leq i\leq n}\ \ \mathtt{context}\ \zeta\ \mathtt{inv:}\ \tilde{e}$. The only applicable operational rule for such a $\tilde{t}$ is (Spec$^{\downarrow}$), the only applicable denotational clause (Spec$^{\in}$). By (Spec$^{\downarrow}$), we have $\omega; \gamma \vdash \tilde{t} \downarrow \overline{v}'$ with $\overline{v}' \in \overline{V}_\omega$ if, and only if $\omega, (\iota_j)_{1\leq j\leq n}; \gamma \vdash \tilde{d}_i \downarrow \iota_i$ for all $1 \leq i \leq n$ and $\omega, (\iota_j)_{1\leq j\leq n}; \gamma, \mathtt{self} \mapsto v \vdash \tilde{e} \downarrow \overline{v}_v$ with $\overline{v}_v \in \overline{V}_\omega$ for all $v \in \omega(\zeta)$. But, $\omega, (\iota_j)_{1\leq j\leq n}; \gamma \vdash \tilde{d}_i \downarrow \iota_i$ for all $1 \leq i \leq n$ if, and only if we have $(\iota_i')_{1\leq i\leq n} = \mathbf{Y}\lambda(\iota_i')_{1\leq i\leq n}.([\![\tilde{d}_i]\!]\,(\omega, (\iota_j')_{1\leq j\leq n})\,\gamma)_{1\leq i\leq n}$ and $\iota_i = \iota_i'$ for all $1 \leq i \leq n$ by induction on the structure of $\tilde{d}$ and Kleene's fixed-point theorem.
If $\omega; \gamma \vdash \tilde{t} \downarrow v'$ with $v' \in V_\omega$ then, since $\Omega; \Gamma \vdash t \triangleright \tilde{t} : \tau$, the variable environment $\gamma, \mathtt{self} \mapsto v$ conforms to $\Gamma, \mathtt{self} : \zeta$ for all $v \in \omega(\zeta)$. Thus we have, by the induction hypothesis, that $\omega, (\iota_j')_{1\leq j\leq n}; \gamma, \mathtt{self} \mapsto v \vdash \tilde{e} \downarrow v_v$ if, and only if $v_v \in [\![\tilde{e}]\!]\,(\omega, (\iota_j')_{1\leq j\leq n})\,(\gamma, \mathtt{self} \mapsto v)$, and hence if, and only if $v' \in [\![\tilde{t}]\!]\,\omega\,\gamma$ by the definition of (Spec$^{\in}$). Conversely, $\omega; \gamma \vdash \tilde{t} \downarrow \mathtt{undef}$ if, and only if $\omega, (\iota_j')_{1\leq j\leq n}; \gamma, \mathtt{self} \mapsto v \vdash \tilde{e} \downarrow \mathtt{undef}$ for some $v \in \omega(\zeta)$, if, and only if $\mathtt{undef} \in [\![\tilde{e}]\!]\,(\omega, (\iota_j')_{1\leq j\leq n})\,(\gamma, \mathtt{self} \mapsto v)$, if, and only if $\mathtt{undef} \in [\![\tilde{t}]\!]\,\omega\,\gamma$ by the definition of $\bigcup_\omega$.

Let $\tilde{t} = \mathtt{if}\ \tilde{e}\ \mathtt{then}\ \tilde{e}_1\ \mathtt{else}\ \tilde{e}_2\ \mathtt{endif}$. The only applicable operational rules for such a $\tilde{t}$ are (Cond$_1^{\downarrow}$) and (Cond$_2^{\downarrow}$), the only applicable denotational clause (Cond$^{\in}$). By (Cond$_1^{\downarrow}$) and (Cond$_2^{\downarrow}$), $\omega; \gamma \vdash \tilde{t} \downarrow v$ with $v \in V_\omega$, if, and only if either $\omega; \gamma \vdash \tilde{e} \downarrow \mathtt{true}$ and $\omega; \gamma \vdash \tilde{e}_1 \downarrow v_1$ with $v_1 \in V_\omega$ or $\omega; \gamma \vdash \tilde{e} \downarrow \mathtt{false}$ and $\omega; \gamma \vdash \tilde{e}_2 \downarrow v_2$ with $v_2 \in V_\omega$, if, and only if either $\mathtt{true} \in [\![\tilde{e}]\!]\,\omega\,\gamma$ and $v_1 \in [\![\tilde{e}_1]\!]\,\omega\,\gamma$ or $\mathtt{false} \in [\![\tilde{e}]\!]\,\omega\,\gamma$ and $v_2 \in [\![\tilde{e}_2]\!]\,\omega\,\gamma$ by the induction hypothesis, and hence if, and only if $v \in [\![\tilde{t}]\!]\,\omega\,\gamma$ by the definition of (Cond$^{\in}$). Conversely, $\omega; \gamma \vdash \tilde{t} \downarrow \mathtt{undef}$ if, and only if $\omega; \gamma \vdash \tilde{e} \downarrow \mathtt{undef}$ or $\omega; \gamma \vdash \tilde{e} \downarrow \mathtt{true}$ and $\omega; \gamma \vdash \tilde{e}_1 \downarrow \mathtt{undef}$ or $\omega; \gamma \vdash \tilde{e} \downarrow \mathtt{false}$ and $\omega; \gamma \vdash \tilde{e}_2 \downarrow \mathtt{undef}$, if, and only if $\mathtt{undef} \in [\![\tilde{e}]\!]\,\omega\,\gamma$ or $\mathtt{true} \in [\![\tilde{e}]\!]\,\omega\,\gamma$ and $\mathtt{undef} \in [\![\tilde{e}_1]\!]\,\omega\,\gamma$ or $\mathtt{false} \in [\![\tilde{e}]\!]\,\omega\,\gamma$ and $\mathtt{undef} \in [\![\tilde{e}_2]\!]\,\omega\,\gamma$, if, and only if $\mathtt{undef} \in [\![\tilde{t}]\!]\,\omega\,\gamma$ by the definition of $\bigcup_\omega$.

Let $\tilde{t} = \tilde{e}_1\ \mathtt{and}\ \tilde{e}_2$. The only applicable operational rules for such a $\tilde{t}$ are (And$_1^{\downarrow}$–And$_3^{\downarrow}$), the only applicable denotational clause (And$^{\in}$). By (And$_1^{\downarrow}$–And$_3^{\downarrow}$), $\omega; \gamma \vdash \tilde{t} \downarrow v$ with $v \in V_\omega$ if, and only if $\omega; \gamma \vdash \tilde{e}_i \downarrow v_i$ with $v_1, v_2 \in V_\omega$ or $\omega; \gamma \vdash \tilde{e}_1 \downarrow \mathtt{false}$ or $\omega; \gamma \vdash \tilde{e}_2 \downarrow \mathtt{false}$, if, and only if $v_i \in [\![\tilde{e}_i]\!]\,\omega\,\gamma$ for $1 \leq i \leq 2$ or $\mathtt{false} \in [\![\tilde{e}_1]\!]\,\omega\,\gamma$ or $\mathtt{false} \in [\![\tilde{e}_2]\!]\,\omega\,\gamma$ by the induction hypothesis, if and only if $v \in [\![\tilde{t}]\!]\,\omega\,\gamma$ by the definition of $\wedge_\omega$. Conversely, $\omega; \gamma \vdash \tilde{t} \downarrow \mathtt{undef}$ if, and only if $\omega; \gamma \vdash \tilde{e}_i \downarrow \overline{v}_i$ with $\overline{v}_1, \overline{v}_2 \in \overline{V}_\omega$ and $\overline{v}_1 \neq \mathtt{false}$ and $\overline{v}_2 = \mathtt{undef}$ or $\overline{v}_1 = \mathtt{undef}$ and $\overline{v}_2 \neq \mathtt{false}$, if, and only if $\mathtt{undef} \in [\![\tilde{t}]\!]\,\omega\,\gamma$ by the definition of $\wedge_\omega$.

Let $\tilde{t} = \tilde{e}\mathtt{->iterate(}x\mathtt{;}\ x' = \tilde{e}' \mathtt{ \mid } \tilde{e}''\mathtt{)}$. The only applicable operational rule for such a $\tilde{t}$ is (Iter$^{\downarrow}$), the only applicable denotational clause (Iter$^{\in}$). By (Iter$^{\downarrow}$), $\omega; \gamma \vdash \tilde{t} \downarrow v$ with $v \in V_\omega$ if, and only if $\omega; \gamma \vdash \tilde{e} \downarrow v'$ with $v' \in V_\omega$ and $\omega; \gamma \vdash \tilde{e}' \downarrow v_0''$ and $\omega; \gamma, x \mapsto v_i', x' \mapsto v_{i-1}'' \vdash \tilde{e}'' \downarrow v_i''$ with $v_i'' \in V_\omega$ for $0 \leq i \leq n$ where $v' \rightsquigarrow \mathtt{Sequence\{}v_1', \ldots, v_n'\mathtt{\}}$. By the induction hypothesis, $\omega; \gamma \vdash \tilde{e} \downarrow v'$ if, and only if $v' \in [\![\tilde{e}]\!]\,\omega\,\gamma$ and $\omega; \gamma \vdash \tilde{e}' \downarrow v_0''$ if, and only if $v_0'' \in [\![\tilde{e}']\!]\,\omega\,\gamma$. Moreover, since $\Omega; \Gamma \vdash t \triangleright \tilde{t} : \tau_{x'}$, inductively, all variable environments $\gamma, x \mapsto v_i', x' \mapsto v_{i-1}''$ conform to $\Gamma, x : \tau_x, x' : \tau_{x'}$ for some $\tau_x, \tau_{x'} \in T_\Omega$, and thus we have that $\omega; \gamma, x \mapsto v_i', x' \mapsto v_{i-1}'' \vdash \tilde{e}'' \downarrow v_i''$ if, and only if $v_i'' \in [\![\tilde{e}'']\!]\,\omega\,(\gamma, x \mapsto v_i', x' \mapsto v_{i-1}'')$. Thus, $\omega; \gamma \vdash \tilde{t} \downarrow v$ if, and only if $v \in [\![\tilde{t}]\!]\,\omega\,\gamma$ by the definition of (Iter$^{\in}$). Conversely, $\omega; \gamma \vdash \tilde{t} \downarrow \mathtt{undef}$ if, and only if $\omega; \gamma \vdash \tilde{e} \downarrow \mathtt{undef}$ or $\omega; \gamma \vdash \tilde{e}' \downarrow \mathtt{undef}$ or $\omega; \gamma \vdash \tilde{e} \downarrow v'$ and $\omega; \gamma \vdash \tilde{e}' \downarrow v_0''$ with $v', v_0'' \in V_\omega$ and $\omega; \gamma, x \mapsto v_i', x' \mapsto v_{i-1}'' \vdash \tilde{e}'' \downarrow v_i''$ for $1 \leq i \leq k$ and $\omega; \gamma, x \mapsto v_k', x' \mapsto v_{k-1}'' \vdash \tilde{e}'' \downarrow \mathtt{undef}$ and $1 \leq k \leq n$ where $v' \rightsquigarrow \mathtt{Sequence\{}v_1', \ldots, v_n'\mathtt{\}}$. But hence $\omega; \gamma \vdash \tilde{t} \downarrow \mathtt{undef}$ if, and only if $\mathtt{undef} \in [\![\tilde{t}]\!]\,\omega\,\gamma$ by the induction hypothesis and the definition of $iterate_\omega$ and $\bigcup_\omega$. □

Index of symbols

$\Omega; \Gamma \vdash t \triangleright \tilde{t} : \theta$ (type judgement)
$\omega; \gamma \vdash \tilde{t} \downarrow \rho$ (evaluation judgement)

Ω (static basis), Sect. 3.1
ω (dynamic basis), Sect. 4.1
Ω, δ (static basis extension), Sect. 3.1
ω, ι (dynamic basis extension), Sect. 4.1

A_Ω (simple types), Sect. 3.1
B (basic types), Sect. 3.1
C_Ω (classifiers), Sect. 3.1
S (concrete collections), Sect. 3.1
$\overline{S}$ (collections), Sect. 3.1
T_Ω (compile-time types), Sect. 3.1
U_Ω (non-collection types), Sect. 3.1
$\leq_\Omega$ (subtype relation), Sect. 3.1
$\leq_{C_\Omega}$ (generalisation relation), Sect. 3.1
$\bigsqcup_\Omega$ (supremum of types), Sect. 3.1
type (type of a literal), Sect. 3.1

$\bot$ (non-termination), Sect. 4.1
C_ω (classifiers), Sect. 4.1
N_ω (basic values), Sect. 4.1
O_ω (instances), Sect. 4.1
T_ω (run-time types), Sect. 4.1
V_ω (values), Sect. 4.1
$\overline{V}_\omega$ (results), Sect. 4.1
$\wp(\overline{V}_\omega)_\bot$ (powerdomain of results), Sect. 4.1
$:_\omega$ (subtype relation), Sect. 4.1
$:_\omega$ (typing relation), Sect. 4.1
$:_{O_\omega}$ (instance relation), Sect. 4.1
$\rightsquigarrow$ (sequence value representation), Sect. 4.1
$\wedge_\omega$ (parallel and), Sect. 5.1
$\vee_\omega$ (parallel or), Sect. 5.1
$cond_\omega$ (conditional), Sect. 5.1
$iterate_\omega$ (iterate functional), Sect. 5.1
$make_\omega$ (collection constructor), Sect. 4.1
$\bigcup_\omega$ (lifting), Sect. 5.1
$\overline{\bigcup}_\omega$ (lifting), Sect. 5.1

D_Ω (declarations), Sect. 3.1
fd_Ω (declaration retrieval), Sect. 3.1

F_ω (non-deterministic functions), Sect. 4.1
I_ω (implementations), Sect. 4.1
J_ω (denotational implementations), Sect. 5.1
$\cdots \equiv \psi$ (implementation), Sect. 4.1
$impl_\omega$ (implementation retrieval), Sect. 4.1
$\overline{impl}_\omega$ (lifted implementation retrieval), Sect. 4.1

$\emptyset$ (empty type environment), Sect. 3.2
Γ (type environment), Sect. 3.2
$x : \tau$ (variable typing), Sect. 3.2
Γ, Γ' (concatenation of type environments), Sect. 3.2
$\Gamma(x)$ (variable typing retrieval), Sect. 3.2
$\mathrm{dom}(\Gamma)$ (domain of a type environment), Sect. 3.2

$\emptyset$ (empty variable environment), Sect. 4.2
γ (variable environment), Sect. 4.2
$x \mapsto v$ (variable assignment), Sect. 4.2
γ, γ' (concatenation of variable environments), Sect. 4.2
$\gamma(x)$ (variable assignment retrieval), Sect. 4.2
$\overline{\gamma}(x)$ (lifted variable assignment retrieval), Sect. 4.2
$\mathrm{dom}(\gamma)$ (domain of a variable environment), Sect. 4.2

α ($\in A_\Omega$), Sect. 3.1
δ ($\in D_\Omega$), Sect. 3.1
ι ($\in I_\omega$), Sect. 4.2
ψ ($\in A\text{-}Expr \cup F_\omega$), Sect. 4.1
ρ ($\in \overline{V}_\omega \cup I_\omega$), Sect. 4.2
σ ($\in S$), Sect. 3.1
$\overline{\sigma}$ ($\in \overline{S}$), Sect. 3.1
τ ($\in T_\Omega$), Sect. 3.1
θ ($\in T_\Omega \cup D_\Omega$), Sect. 3.2
υ ($\in U_\Omega$), Sect. 3.2
ζ ($\in C_\Omega$), Sect. 3.1
a ($\in Name$), Sect. 3.1
d ($\in AttrDef \cup OpDef$), Sect. 3.2
$\tilde{d}$ ($\in A\text{-}AttrDef \cup A\text{-}OpDef$), Sect. 3.2
e ($\in Expr$), Sect. 3.2
l ($\in Literal$), Sect. 3.1
r ($\in (\overline{V}_\omega)_\bot$), Sect. 4.3
v ($\in V_\omega$), Sects. 3.1, 3.2
$\overline{v}$ ($\in \overline{V}_\omega$), Sect. 3.2
x ($\in Var$), Sect. 3.2

$]\ldots[$ (optional clause), Sect. 3.2
$A\text{-}\ldots$ (annotated terms), Sect. 3.2