ON THE SEMANTICS OF OBJECT-ORIENTED DATA STRUCTURES AND PATH EXPRESSIONS

BRUCKER A D / LONGUET D / TUONG F / WOLFF B

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

09/2013

Rapport de Recherche N° 1565

On the Semantics of Object-oriented Data Structures and Path Expressions
Extended Version

Achim D. Brucker[1], Delphine Longuet[2], Frédéric Tuong[3], and Burkhart Wolff[2]

[1] SAP AG, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
`achim.brucker@sap.com`
[2] Univ. Paris-Sud, Laboratoire LRI, UMR8623, 91405 Orsay, France
CNRS, 91405 Orsay, France
`{delphine.longuet, burkhart.wolff}@lri.fr`
[3] Univ. Paris-Sud, IRT SystemX, 8 av. de la Vauve, 91120 Palaiseau, France
`frederic.tuong@{u-psud, irt-systemx}.fr`

Abstract UML/OCL is perceived as the de-facto standard for specifying object-oriented models in general and data models in particular. Since recently, all data types of UML/OCL comprise two different exception elements: `invalid` ("bottom" in semantics terminology) and `null` (for "non-existing element"). This has far-reaching consequences on both the logical and algebraic properties of OCL expressions as well as the path expressions over object-oriented data structures, i. e., class models.
In this paper, we present a formal semantics for object-oriented data models in which all data types and, thus, all class attributes and path expressions, support `invalid` and `null`. Based on this formal semantics, we present a set of OCL test cases that can be used for evaluating the support of `null` and `invalid` in OCL tools.
Keywords: Object-oriented Data Structures, Path Expressions, Featherweight OCL, Null, Invalid, Formal Semantics

1 Introduction

UML/OCL is perceived as the de-facto standard for modeling object-oriented systems in general and object-oriented data structures in particular. Since 2006 [10], all data types of UML/OCL comprise two different exception elements: `invalid` ("bottom" in semantics terminology) and `null` (for "non-existing element"). This has far-reaching consequences on both the logical and algebraic properties of OCL expressions as well as the path expressions of class models.

In [5], we presented a formal semantics for a subset of OCL 2.3.1 [11], called Featherweight OCL, and we discussed the consequences of `invalid` and `null` on the logic layer and the algebraic layer. In this paper, we discuss the consequences on the data modeling layer: we present a formal semantics for object-oriented data structures as well as for path expressions that are necessary to express class invariants and contracts consisting of preconditions and postconditions.

Consider, for example, a simple design model capturing a management hierarchy in a company (see Fig. 1). While, theoretically, both the attribute `salary` and the association end `boss` can be `invalid`, valid but not represent a "regular

```
context Person
 inv min_salary: salary > 0
 inv boss_earns_more:
     boss <> null implies boss.salary >= salary
```

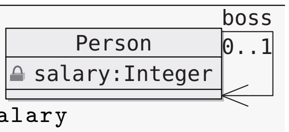

Fig. 1. A simple design model capturing a management hierarchy in a company

value" (i. e., `null`), or valid and represent a regular value (i. e., an integer value representing a salary, respectively, a valid object of type `Person`), this is not true in reality: from the multiplicity requirement `0..1` we can directly infer that the association end `boss` is valid. Still, it is not immediately clear if the `null` is a valid representation of an association end with multiplicity `0..1`. In fact, this is one of the questions we answer in this paper. From the invariant `min_salary`, we would expect that the attribute `salary` is always valid as well as never `null`.

The main contribution of this paper is a formal, machine-checked semantics for object-oriented data models that can be enriched with class invariants as well as preconditions and postconditions expressed in Featherweight OCL [5]. The underlying formalization of object-oriented datatypes[4] extends the work of [3] with support for `null`: all data types and, thus, all class attributes and path expressions, support both exception elements (see Sec. 3). Moreover, based on this formal semantics, we present a set of OCL test cases that can be used for evaluating the support of `null` and `invalid` in OCL tools (see Sec. 4).

2 Background

2.1 Higher-order Logic and Isabelle

Higher-order Logic (HOL) [1, 8] is a classical logic with equality enriched by total polymorphic higher-order functions. It is more expressive than first-order logic, e. g., induction schemes can be expressed inside the logic. HOL is based on the typed λ-calculus, i. e., the *terms* of HOL are λ-expressions. Types of terms may be built from *type variables* (like α, β, ..., optionally annotated by Haskell-like *type classes* as in $\alpha :: \mathit{order}$ or $\alpha :: \mathrm{bot}$) or *type constructors*. Type constructors may have arguments (as in $\alpha\ \mathrm{list}$ or $\alpha\ \mathrm{set}$). The type constructor for the function space $\Rightarrow$ is written infix: $\alpha \Rightarrow \beta$; multiple applications like $\tau_1 \Rightarrow (\ldots \Rightarrow (\tau_n \Rightarrow \tau_{n+1})\ldots)$ have the alternative syntax $[\tau_1, \ldots, \tau_n] \Rightarrow \tau_{n+1}$. HOL is centered around the extensional logical equality $\_ = \_$ with type $[\alpha, \alpha] \Rightarrow \mathrm{bool}$, where bool is the fundamental logical type. We use infix notation: instead of $(\_ = \_)\ E_1\ E_2$ we write $E_1 = E_2$. The logical connectives $\_ \wedge \_$, $\_ \vee \_$, $\_ \Rightarrow \_$ of HOL have type $[\mathrm{bool}, \mathrm{bool}] \Rightarrow \mathrm{bool}$, $\neg\_$ has type $\mathrm{bool} \Rightarrow \mathrm{bool}$. The quantifiers $\forall\,\_.\_$ and $\exists\,\_.\_$ have type $[\alpha \Rightarrow \mathrm{bool}] \Rightarrow \mathrm{bool}$. The quantifiers may range over types of higher order, i. e., functions or sets. The definition of the element-hood $\_ \in \_$, the set comprehension $\{\_.\_\}$, as well as $\_ \cup \_$ and $\_ \cap \_$ are standard.

[4] The formalization is available at: `https://projects.brucker.ch/hol-testgen/svn/HOL-TestGen/trunk/hol-testgen/add-ons/Featherweight-OCL/`.

Isabelle is a generic interactive theorem proving system; Isabelle/HOL is an instance of the former with HOL. The Isabelle/HOL library contains formal definitions and theorems for a wide range of mathematical concepts used in computer science, including typed set theory, well-founded recursion theory, number theory and theories for data-structures like Cartesian products $\alpha \times \beta$ and disjoint type sums $\alpha + \beta$. The library also includes the type constructor $\tau_\perp := \perp \mid \llcorner\_\lrcorner : \alpha$ that assigns to each type τ a type $\tau_\perp$ *disjointly extended* by the exceptional element $\perp$. The function $\ulcorner\_\urcorner : \alpha_\perp \Rightarrow \alpha$ is the inverse of $\llcorner\_\lrcorner$ (unspecified for $\perp$). Partial functions $\alpha \rightharpoonup \beta$ are defined as functions $\alpha \Rightarrow \beta_\perp$ supporting the usual concepts of domain (dom _) and range (ran _). The library is built entirely by logically safe, conservative definitions and derived rules. This is also true for HOL-OCL [4] and Featherweight OCL [5].

2.2 Formalizing the Core of OCL in HOL: Featherweight OCL

OCL is composed of 1) operators on built-in data structures such as Boolean, Integer or Set(_), 2) operators of the user-defined data model such as accessors, type casts and tests, and 3) user-defined, side-effect-free methods. Conceptually, an OCL expression in general and Boolean expressions in particular (i. e., *formulae*) depends on a pair (σ, σ') of pre- and post-states. The precise form of states is irrelevant for this paper (compare [6]) and will be left abstract in this presentation. We construct in Isabelle a type class null that contains two distinguishable elements bot and null. Any type of the form $(\alpha_\perp)_\perp$ is an instance of this type class with bot $\equiv \perp$ and null $\equiv \llcorner\perp\lrcorner$. Now, any OCL type can be represented by an HOL type of the form: $V(\alpha) := \mathrm{state} \times \mathrm{state} \Rightarrow \alpha :: \mathrm{null}$. We define $V((\mathrm{bool}_\perp)_\perp)$ as the HOL type for the OCL type `Boolean`:

$$I[\![\texttt{invalid} :: V(\alpha)]\!]\tau = \mathrm{bot} \qquad I[\![\texttt{null} :: V(\alpha)]\!]\tau = \mathrm{null}$$

$$I[\![\texttt{true} :: \texttt{Boolean}]\!]\tau = \llcorner\llcorner\mathrm{true}\lrcorner\lrcorner \qquad I[\![\texttt{false}]\!]\tau = \llcorner\llcorner\mathrm{false}\lrcorner\lrcorner$$

$$I[\![X\texttt{.oclIsUndefined()}]\!]\tau = (\mathrm{if}\; I[\![X]\!]\tau \in \left\{\begin{matrix}\mathrm{bot},\\ \mathrm{null}\end{matrix}\right\} \mathrm{then}\; I[\![\texttt{true}]\!]\tau \;\mathrm{else}\; I[\![\texttt{false}]\!]\tau)$$

$$I[\![X\texttt{.oclIsInvalid()}]\!]\tau = (\mathrm{if}\; I[\![X]\!]\tau = \mathrm{bot}\;\mathrm{then}\; I[\![\texttt{true}]\!]\tau \;\mathrm{else}\; I[\![\texttt{false}]\!]\tau)$$

where $I[\![E]\!]$ is the semantic interpretation function commonly used in mathematical textbooks and τ stands for pairs of pre- and post state (σ, σ'). Due to the used style of semantic representation (a shallow embedding) I is in fact superfluous and defined semantically as the identity; in Isabelle theories, it is usually left out in definitions to pave the way for Isabelle to check that the underlying equations are axiomatic definitions and therefore logically safe. For reasons of conciseness, we will write $\delta\ X$ for `not` X`.oclIsUndefined()` and $\upsilon\ X$ for `not` X`.oclIsInvalid()` throughout this paper.

3 Semantics of States and Class Models

In the following, we will refine the notion of state used in the previous section to much more detail. In contrast to wide-spread opinions, UML class diagrams represent in a compact and visual manner quite complex, object-oriented data-types with a surprisingly rich theory. It is part of our endeavor here to make this theory explicit and to point out corner cases. A UML class diagram—underlying a given OCL formula—produces a number of implicit operations which become accessible via appropriate OCL syntax:

1. Classes and class names (written as C_1, ..., C_n), which become types of data in OCL . Class names declare two projector functions to the set of all objects in a state: C_i`.allInstances()` and C_i`.allInstances@pre()`,
2. an inheritance relation _ < _ on classes and a collection of attributes A associated to classes,
3. two families of accessors; for each attribute a in a class definition (denoted $X.a :: C_i \to A$ and $X.a$ `@pre` $:: C_i \to A$ for $A \in \{V(\ldots_\perp), C_1, \ldots, C_n\}$),
4. type casts that can change the static type of an object of a class (denoted X`.oclAsType(`C_i`)` of type $C_j \to C_i$)
5. two dynamic type tests (denoted X`.oclIsTypeOf(`C_i`)` and X`.oclIsKindOf(`C_i`)`),
6. and last but not least, for each class name C_i there is an instance of the overloaded referential equality (written $\_ \doteq \_$).

We will assume a strong static type discipline in the sense of Hindley-Milner types; Featherweight OCL has no "syntactic subtyping." This does not mean that subtyping can not be expressed *semantically* in Featherweight OCL; by giving a formal semantics to type-casts, subtyping becomes an issue of the front-end that can make implicit type-coersions explicit by introducing explicit type-casts.

3.1 Object Universes.

It is natural to construct system states by a set of partial functions f that map object identifiers oid to some representations of objects:

$$\text{typedef} \qquad \alpha \ \text{state} := \{\sigma :: \text{oid} \rightharpoonup \alpha \mid \text{inv}_\sigma(\sigma)\}$$

where inv_σ is a to be discussed invariant on states. The key point is that we need a common type α for the set of all possible *object representations*. Object representations model "a piece of typed memory," i. e., a kind of record comprising administration information and the information for all attributes of an object; here, the primitive types as well as collections over them are stored directly in the object representations, class types and collections over them are represented by oid's (respectively lifted collections over them). In a shallow embedding which must represent UML types injectively by HOL types, there are two fundamentally different ways to construct such a set of object representations, which we call an *object universe* $\mathfrak{A}$:

1. an object universe can be constructed for a given class model, leading to *closed world semantics*, and
2. an object universe can be constructed for a given class model *and all its extensions by new classes added into the leaves of the class hierarchy*, leading to an *open world semantics*.

For the sake of simplicity, we chose the first option for Featherweight OCL, while HOL-OCL [3] used an involved construction allowing the latter.

A naïve attempt to construct $\mathfrak{A}$ would look like this: the class type C_i induced by a class will be the type of such an object representation: $C_i := (\text{oid} \times A_{i_1} \times \cdots \times A_{i_k})$ where the types $A_{i_1}, \ldots, A_{i_k}$ are the attribute types (including inherited attributes) with class types substituted by oid. The function OidOf projects the first component, the oid, out of an object representation. Then the object universe will be constructed by the type definition: $\mathfrak{A} := C_1 + \cdots + C_n$.

It is possible to define constructors, accessors, and the referential equality on this object universe. However, the treatment of type casts and type tests cannot be faithful with common object-oriented semantics, be it in UML or Java: casting up along the class hierarchy can only be implemented by loosing information, such that casting up and casting down will *not* give the required identity:

$$X\texttt{.oclIsTypeOf(}C_k\texttt{)}\ \texttt{implies}\ X\texttt{.oclAsType(}C_i\texttt{).oclAsType(}C_k\texttt{)} \doteq X$$
$$\text{whenever } C_k < C_i \text{ and } X \text{ is valid.}$$

To overcome this limitation, we introduce an auxiliary type $C_{i\text{ext}}$ for *class type extension*; together, they were inductively defined for a given class diagram:

Let C_i be a class with a possibly empty set of subclasses $\{C_{j_1}, \ldots, C_{j_m}\}$.

- Then the *class type extension* $C_{i\text{ext}}$ associated to C_i is $A_{i_1} \times \cdots \times A_{i_n} \times (C_{j_1\text{ext}} + \cdots + C_{j_m\text{ext}})_\perp$ where A_{i_k} ranges over the local attribute types of C_i and $C_{j_l\text{ext}}$ ranges over all class type extensions of the subclass C_j of C_i.
- Then the *class type* for C_i is $oid \times A_{i_1} \times \cdots \times A_{i_n} \times (C_{j_1\text{ext}} + \cdots + C_{j_m\text{ext}})_\perp$ where A_{i_k} ranges over the inherited *and* local attribute types of C_i and $C_{j_l\text{ext}}$ ranges over all class type extensions of the subclass C_j of C_i.

This construction can *not* be done in HOL itself since it involves quantifications and iterations over the "set of types"; rather it is a meta-level construction. Technically, this means that we need a compiler to be done in SML on the syntactic "meta-model"-level of a class model.

It remains to clarify the role of the state invariant $\text{inv}_\sigma(\sigma)$ mentioned above that states the condition that there is a "one-to-one" correspondence between object representations and oid's: $\forall oid \in \text{dom}\ \sigma.\ oid = \text{OidOf}\ \ulcorner\sigma(oid)\urcorner$. This condition is also mentioned in [11, Annex A] and goes back to Richters [12]; however, we state this condition as an invariant on states rather than a global axiom. It can, therefore, not be taken for granted that an oid makes sense both in pre- and post-states of OCL expressions.

Running Example. Although our class model (recall Fig. 1) appears to be trivial, we have already two classes in the class model: `OclAny` and `Person`. `Person < OclAny`, and thus a family of tests and casts. The construction of the universe comprises the following datatype definitions:

$$\begin{array}{ll} \text{datatype} & \text{oclany} = \text{mk}_{\text{OclAny}}\ \ \text{oid}\ \ (\text{int}_\perp \times \text{oid}_\perp)_\perp \\ \text{datatype} & \text{person} = \text{mk}_{\text{Person}}\ \ \text{oid}\ \ (\text{int}_\perp)\ \ (\text{oid}_\perp) \end{array}$$

$$\text{datatype} \qquad \mathfrak{A} = \text{in}_{\text{Person}}\ \text{person} \mid \text{in}_{\text{OclAny}}\ \text{oclany}$$

Here, $(\text{int}_\perp \times \text{oid}_\perp)_\perp$ is (the only) optional extension that represents `Person` objects casted to `OclAny`. In UML terminology, these are objects with dynamic type Person and static type OclAny.

3.2 The Accessors

Our choice to use a shallow embedding of OCL in HOL and, thus having an injective mapping from OCL types to HOL types, results in type-safety of Featherweight OCL. Arguments and results of accessors are based on type-safe object representations and *not* oid's. This implies the following scheme for an accessor:

1. The *evaluation and extraction* phase. If the argument evaluation results in an object representation, the oid is extracted, if not, exceptional cases like `invalid` are reported.
2. The *dereferentiation* phase. The oid is interpreted in the pre- or post-state, the resulting object is casted to the expected format. The exceptional case of nonexistence in this state must be treated.
3. The *selection* phase. The corresponding attribute is extracted from the object representation.
4. The *re-construction* phase. The resulting value has to be embedded in the adequate HOL type. If an attribute has the type of an object (not value), it is represented by an optional (set of) oid, which must be converted via dereferentiation in one of the states in order to produce an object representation again. The exceptional case of nonexistence in this state must be treated.

The first phase directly translates into the following formalization:

$$\begin{array}{llll} \text{definition} & & & \\ \text{eval\_extract}\ X\ f = (\lambda\,\tau.\ \text{case}\ X\ \tau\ \text{of}\ \perp & \Rightarrow \texttt{invalid}\ \tau & & \text{exception} \\ \quad\mid \lfloor\perp\rfloor & \Rightarrow \texttt{invalid}\ \tau & & \text{deref. null} \\ \quad\mid \lfloor\lfloor obj \rfloor\rfloor & \Rightarrow f\ (\text{oid\_of}\ \ obj)\ \tau) & & \end{array}$$

For each class C, we introduce the dereferentiation phase of this form:

$$\begin{array}{l} \text{definition}\ \text{deref\_oid}_C\ \mathit{fst\_snd}\ f\ \mathit{oid} = (\lambda\,\tau.\ \text{case}\ (\text{heap}\ (\mathit{fst\_snd}\ \tau))\ \mathit{oid}\ \text{of} \\ \qquad\qquad \lfloor \text{in}_C\ obj \rfloor \Rightarrow f\ obj\ \tau \\ \qquad\qquad \mid \_ \qquad\quad \Rightarrow \texttt{invalid}\ \tau) \end{array}$$

The operation yields undefined if the oid is uninterpretable in the state or referencing an object representation not conforming to the expected type.

We turn to the selection phase: for each class C in the class model with at least one attribute, and each attribute a in this class, we introduce the selection phase of this form:

$$\begin{array}{l} \text{definition}\ \text{select}_a\ f = (\lambda\ \text{mk}_C\ \mathit{oid}\ \cdots \perp \cdots\ C_{X\text{ext}} \Rightarrow \texttt{null} \\ \qquad\qquad \mid \text{mk}_C\ \mathit{oid} \cdots \lfloor a \rfloor \cdots\ C_{X\text{ext}} \Rightarrow f\ (\lambda\ x\ \_ .\ \lfloor\lfloor x \rfloor\rfloor)\ a) \end{array}$$

This works for definitions of basic values as well as for object references in which the a is of type oid. To increase readability, we introduce the functions:

$$\begin{array}{lrr} \textbf{definition} & \text{in\_pre\_state} = \text{fst} & \text{first component} \\ \textbf{definition} & \text{in\_post\_state} = \text{snd} & \text{second component} \\ \textbf{definition} & \text{reconst\_basetype} = \text{id} & \text{identity function} \end{array}$$

Let `_.getBase` be an accessor of class C yielding a value of base-type A_{base}. Then its definition is of the form:

$$\begin{array}{ll} \textbf{definition} & \texttt{\_.getBase} :: C \Rightarrow A_{base} \\ \textbf{where} & X\texttt{.getBase} = \text{eval\_extract } X \; (\text{deref\_oid}_C \text{ in\_post\_state} \\ & \qquad\qquad (\text{select}_{\text{getBase}} \text{ reconst\_basetype})) \end{array}$$

Let `_.getObject` be an accessor of class C yielding a value of object-type A_{object}. Then its definition is of the form:

$$\begin{array}{ll} \textbf{definition} & \texttt{\_.getObject} :: C \Rightarrow A_{object} \\ \textbf{where} & X\texttt{.getObject} = \text{eval\_extract } X \; (\text{deref\_oid}_C \text{ in\_post\_state} \\ & \qquad\qquad (\text{select}_{\text{getObject}} \; (\text{deref\_oid}_C \text{ in\_post\_state}))) \end{array}$$

The variant for an accessor yielding a collection is omitted here; its construction follows by the application of the principles of the former two. The respective variants $\_.\,a$ `@pre` were produced when in_post_state is replaced by in_pre_state.

Running Example. The dereference-operation instantiated for the class `Person` is clear and will not be repeated here. We focus on the select functions:

$$\begin{array}{lll} \textbf{definition} & & \\ \text{select}_{\text{salary}} \; f & = (\lambda \; \text{mk}_{\text{Person}} \; \_ \; \bot \; \_ & \Rightarrow \texttt{null} \\ & \;\;\; \mid \text{mk}_{\text{Person}} \; \_ \; \llcorner s \lrcorner \; \_ & \Rightarrow f \; (\lambda \, x \; \_ . \; \llcorner\llcorner x \lrcorner\lrcorner) \; s) \\ \text{select}_{\text{boss}} \; f & = (\lambda \; \text{mk}_{\text{Person}} \; \_ \; \_ \; \bot & \Rightarrow \texttt{null} \\ & \;\;\; \mid \text{mk}_{\text{Person}} \; \_ \; \_ \; \llcorner b \lrcorner & \Rightarrow f \; (\lambda \, x \; \_ . \; \llcorner\llcorner x \lrcorner\lrcorner) \; b) \end{array}$$

Which gives the top-level definitions:

$$\begin{array}{ll} \textbf{definition} & \texttt{\_.salary} :: \text{Person} \Rightarrow \text{Integer} \\ \textbf{where} & X\texttt{.salary} = \text{eval\_extract } X \; (\text{deref\_oid}_{\text{Person}} \text{ in\_post\_state} \\ & \qquad\qquad (\text{select}_{\text{salary}} \text{ reconst\_basetype})) \end{array}$$

$$\begin{array}{ll} \textbf{definition} & \texttt{\_.boss} :: \text{Person} \Rightarrow \text{Person} \\ \textbf{where} & X\texttt{.boss} = \text{eval\_extract } X \; (\text{deref\_oid}_{\text{Person}} \text{ in\_post\_state} \\ & \qquad\qquad (\text{select}_{\text{boss}} \; (\text{deref\_oid}_{\text{Person}} \text{ in\_post\_state}))) \end{array}$$

3.3 Tests for Types and Casts

As a consequence of our decision to consider subtyping an issue to be solved by a static type-checker, the semantic treatment of casts and dynamic types lie in the heart of the concept of object-orientedness of Featherweight OCL. We reduce subtyping to castability, and type-tests allow for specifying exactly the semantics of operation calls. Although OCL has no constructors inside the language, objects can be constructed in HOL and can be specified via OCL operation contracts. The problem needs therefore to be solved that objects have an implicit dynamic ("actual") type, which is invariant under cast; casts change only the static (statically inferable, "apparent") type of an object.

First, we focus on dynamic type tests, which are written in OCL by X`.oclIsTypeOf(`C`)`, where C is a class identifier and X an OCL expression. To implement a similar syntax in Featherweight OCL, we declare for each class C_i of the class model a constant X`.oclIsTypeOf(`C_i`)` of a too large type $\alpha \Rightarrow$ `Boolean`. These constants will now be defined by concrete instances for pairs of classes C_i, C_j (where C_j is either a subtype or a super-type of C_i). Classes not in subtype relation have no casts, so nothing is to define.

$$
\begin{array}{lll}
\text{defs (overloaded) } (X :: C_i)\texttt{.oclIsTypeOf(}C_j\texttt{)} & \equiv (\lambda\,\tau.\ \text{case}\, X\ \tau\ \text{of} & \\
\quad \perp & \Rightarrow \texttt{invalid}\,\tau & \\
\quad |\ \llcorner\perp\lrcorner & \Rightarrow \texttt{true}\,\tau & \\
\quad |\ \llcorner\llcorner \text{mk}_{C_i}\ \mathit{oid}\ \cdots a \cdots\ \perp\lrcorner\lrcorner & \Rightarrow \texttt{true}\,\tau & \text{if } C_i = C_j \\
\quad |\ \llcorner\llcorner \text{mk}_{C_i}\ \mathit{oid}\ \cdots a \cdots\ \llcorner X\lrcorner\lrcorner\lrcorner & \Rightarrow \texttt{true}\,\tau & \text{if } (*) \\
\quad |\ \llcorner\llcorner\_\lrcorner\lrcorner & \Rightarrow \texttt{false}\,\tau &)\ \text{otherwise}
\end{array}
$$

where (*) stands for "$C_i > C_j$ and the pattern X matches exactly the extension $C_{j\text{ext}}$ of the subclass C_j of C_i". Given the fact that we assume closed-world semantics, a simple way to define the overloaded `oclIsKindOf` operation is by the disjunction:

$$
\begin{aligned}
(X :: C_i)\texttt{.oclIsKindOf(}C_j\texttt{)} \equiv\ & X\texttt{.oclIsTypeOf(}C_j\texttt{)}\ \texttt{or} \\
& X\texttt{.oclIsKindOf(}C_{j_1}\texttt{)}\ \texttt{or}\ \cdots\ \texttt{or}\ X\texttt{.oclIsKindOf(}C_{j_n}\texttt{)}
\end{aligned}
$$

where $C_{j_1}, \cdots, C_{j_n}$ all the sub-classes of C_j occurring in the given class model. We refrain from the attempt of a generic definition of the type cast function family; here, we refer to an analogy of the pattern-matching equations shown above and the concrete example below. Both definitions make tests and casts strict and neutral or idempotent on `null`:

$$
\begin{aligned}
(\texttt{invalid} :: C_i)\texttt{.oclIsTypeOf(}C_j\texttt{)} &= \texttt{invalid} \\
(\texttt{null} :: C_i)\texttt{.oclAsType(}C_j\texttt{)} &= \texttt{null} \\
(\texttt{invalid} :: C_i)\texttt{.oclAsType(}C_j\texttt{)} &= \texttt{invalid} \\
(\texttt{null} :: C_i)\texttt{.oclIsTypeOf(}C_j\texttt{)} &= \texttt{true}
\end{aligned}
$$

This is a slight deviation from the standard: `null` as argument should in general yield `invalid`. However, `null` is usually considered as one unique constant appearing in all types; we have technically one polymorphic constant `null`. To mimic the desired effect, the last equation is required. Another issue is that casts yield `null` for a `null`- argument (with the right static type). Since casts can appear everywhere, this is to avoid non intuitive effects. Consider the case that X

and Y have a distinct class type C_i and C_j. Then the OCL term

$$X \doteq \texttt{null}\ \texttt{and}\ Y \doteq \texttt{null}\ \texttt{and}\ X \doteq Y$$

is either false or undefined, since $X \doteq Y$ is either translated to $X\texttt{.oclAsType(}C_j\texttt{)} \doteq Y$ or $X \doteq Y\texttt{.oclAsType(}C_i\texttt{)}$ and thus to `invalid` if we apply the rule `null.oclAsType(_)` $=$ `invalid` as assumes the standard.

Running Example. In the following, we instantiate the generic definitions for our example. We discussed the overloaded constant declarations for dynamic type tests in the previous section. A concrete instance of the definition is:

$$\begin{array}{lll}
\text{defs (overloaded) } (X :: \texttt{OclAny})\ \texttt{.oclIsTypeOf(Person)} & \equiv (\lambda\,\tau.\ \text{case}\ X\ \tau\ \text{of} & \\
\quad \bot & \Rightarrow \texttt{invalid}\ \tau & \\
\quad \mid \lfloor\bot\rfloor & \Rightarrow \texttt{true}\ \tau & \\
\quad \mid \lfloor\lfloor \mathrm{mk}_{\mathrm{OclAny}}\ \_\ \bot\rfloor\rfloor & \Rightarrow \texttt{false}\ \tau & \\
\quad \mid \lfloor\lfloor \mathrm{mk}_{\mathrm{OclAny}}\ \_\ \lfloor\_\rfloor\rfloor\rfloor & \Rightarrow \texttt{true}\ \tau &)
\end{array}$$

Analogously, the casts were declared as overloaded family of constants:

$$\text{consts}\ \ \_\texttt{.oclAsType(OclAny)} :: \alpha \Rightarrow \texttt{OclAny}$$
$$\text{consts}\ \ \_\texttt{.oclAsType(Person)} :: \alpha \Rightarrow \texttt{Person}$$

whose instances were provided, for example, by:

$$\begin{array}{ll}
\text{defs (overloaded) } (X :: \texttt{OclAny})\ \texttt{.oclAsType(Person)} & \equiv (\lambda\,\tau.\ \text{case}\ X\ \tau\ \text{of} \\
\quad \bot & \Rightarrow \texttt{invalid}\ \tau \\
\quad \mid \lfloor\bot\rfloor & \Rightarrow \texttt{null}\ \tau \\
\quad \mid \lfloor\lfloor \mathrm{mk}_{\mathrm{OclAny}}\ \_\ \bot\rfloor\rfloor & \Rightarrow \texttt{invalid}\ \tau \\
\quad \mid \lfloor\lfloor \mathrm{mk}_{\mathrm{OclAny}}\ \mathit{oid}\ \lfloor(a,b)\rfloor\rfloor\rfloor & \Rightarrow \lfloor\lfloor \mathrm{mk}_{\mathrm{Person}}\ \mathit{oid}\ a\ b\rfloor\rfloor)
\end{array}$$

Besides the lemmas on strictness and null-preservation, we prove formally:

$$\begin{array}{l}
\tau \models (X :: \texttt{OclAny})\texttt{.oclIsTypeOf(OclAny)} \Longrightarrow \tau \models \delta\ X \\
\qquad \Longrightarrow \tau \not\models \upsilon\ (X\texttt{.oclAsType(Person)})
\end{array}$$

$$((X :: \texttt{Person})\texttt{.oclAsType(OclAny).oclAsType(Person)} = X$$

These lemmas show the key-properties of the object-universe construction wrt. to casting and type tests.

3.4 Access to the global state

With a little syntactico/semantico trick it is possible to define the global accessor on the state in a univeral, generic (class model independent) way:

$$\begin{array}{l}
\text{definition} \\
\_\texttt{.allInstances()} \qquad :: (\mathfrak{A} \Rightarrow \alpha_{\bot}) \Rightarrow (\mathfrak{A} :: \text{object}, (\alpha_{\bot})_{\bot})\ \text{set} \\
\text{where} \\
(H\texttt{.allInstances()})\ \tau = \text{Abs\_Set\_0}\ \lfloor\lfloor \text{Some} \\
\qquad\qquad\qquad\qquad\qquad `\,(\quad (H\ `\ \text{ran}(\text{heap}(\text{snd}\ \tau)))\ -\ \{\bot\}\quad)\rfloor\rfloor
\end{array}$$

In our running example, this boils down to the definition of the two type characterization functions:

definition

$$\begin{array}{lllll}
\texttt{Person} = (\lambda\ \mathrm{in}_{\mathrm{Person}} & (\mathrm{mk}_{\mathrm{Person}}\ oid\ a\ b) & \Rightarrow \llcorner \mathrm{mk}_{\mathrm{Person}}\ \ oid\ a\ b \lrcorner & \\
\qquad\qquad\ \mid \mathrm{in}_{\mathrm{OclAny}} & (\mathrm{mk}_{\mathrm{OclAny}}\ \ oid\ \llcorner(a,b)\lrcorner) & \Rightarrow \llcorner \mathrm{mk}_{\mathrm{Person}}\ \ oid\ a\ b \lrcorner & \\
\qquad\qquad\ \mid \mathrm{in}_{\mathrm{OclAny}} & (\mathrm{mk}_{\mathrm{OclAny}}\ \ \_\ \bot) & \Rightarrow \bot &) \\
\texttt{OclAny} = (\lambda\ \mathrm{in}_{\mathrm{Person}} & (\mathrm{mk}_{\mathrm{Person}}\ oid\ a\ b) & \Rightarrow \llcorner \mathrm{mk}_{\mathrm{OclAny}}\ \ oid\ \llcorner(a,b)\lrcorner\lrcorner & \\
\qquad\qquad\ \mid \mathrm{in}_{\mathrm{OclAny}} & X & \Rightarrow \llcorner X \lrcorner &)
\end{array}$$

It is easy to prove on the basis of these definitions, that our global accessors have "isKindOf"-semantics:

$$\forall \tau.\quad \tau \models \texttt{OclAny.allInstances()->forAll(}X\,\texttt{|}\,X\texttt{.oclIsKindOf(OclAny))}$$

instead of "isTypeOf"-semantics since we can also prove:

$$\begin{array}{ll} \exists \tau_1. & \tau_1 \not\models \\ \exists \tau_2. & \tau_2 \models \end{array} \texttt{OclAny.allInstances()->forAll(}X\,\texttt{|}\,X\texttt{.oclIsTypeOf(OclAny))}$$

We found out that the current Annex A actually defines the latter, while the mandatory part apparently favors the former.

4 Corner Cases of Path Expression Semantics

4.1 Objects and Accessors

In this section, we illustrate the definitions of the previous section on a concrete example. Figure 2 shows two states, i.e., two object diagrams, of the system described in Figure 1. We consider the state on the left as a pre-state and call it σ, while the state on the right is used as a post-state and is called σ'.

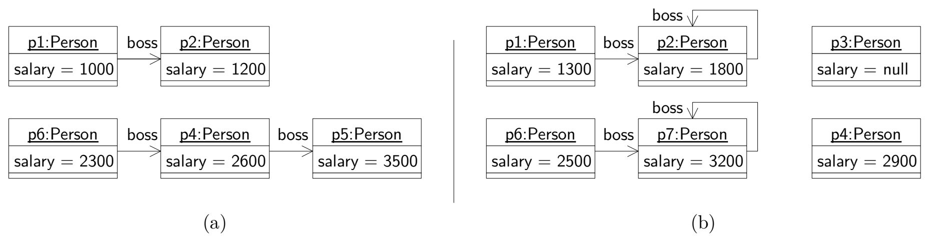

Fig. 2. Two system states for the model of Fig. 1: (a) pre-state σ; (b) post-state σ'.

An OCL formula φ on this system is interpreted with respect to the pair (σ, σ') according to the semantics given in the previous section; we write as usual $(\sigma, \sigma') \models \varphi$ if φ holds in the context of (σ, σ').

For instance, we have $(\sigma,\sigma') \models$ `p1.salary` $\doteq$ 1300, since the attribute salary of object $p1$ has the value 1300 in the post-state. We also have $(\sigma,\sigma') \models$ `p1.salary@pre` $\doteq$ 1000 since $p1$ also existed in the pre-state and its salary was 1000. In the same way, we have $(\sigma,\sigma') \models$ `p6.boss` $\doteq$ `p7` since $p7$ is the boss of $p6$ in the post-state, while $(\sigma,\sigma') \models$ `p6.boss@pre` $\doteq$ `p4` since $p6$ existed in the pre-state and its boss was $p4$ there.

We have a particular case with $p3$, which has no salary in the post-state. Therefore we have $(\sigma,\sigma') \models$ `p3.salary` $\doteq$ `null`.[5] It also has no boss so $(\sigma,\sigma') \models$ `p3.boss` $\doteq$ `null`.Trying to de-referenciate a null association end yields an *invalid* value, so $(\sigma,\sigma') \not\models \upsilon$ `p3.boss.salary`. In a similar way, since $p3$ didn't exist in the pre-state, its de-referenciation in this state necessarily fails, yielding an *invalid* value: $(\sigma,\sigma') \not\models \upsilon$ `p3.salary@pre`, and $(\sigma,\sigma') \not\models \upsilon$ `p3.boss@pre`.

More complex expressions lead to other cases that are well-defined although not always intuitive. When an expression refers to only one state, the semantics remains easily comprehensible. For instance, the following formulas are evaluated in the post-state only:

$$\begin{array}{ll} \forall\sigma. & (\sigma,\sigma') \models \texttt{p1.boss.salary} \doteq 1800 \\ \forall\sigma. & (\sigma,\sigma') \models \texttt{p1.boss.boss} \doteq \texttt{p2} \\ \forall\sigma. & (\sigma,\sigma') \models \texttt{p7.boss.salary} \doteq 3200 \end{array}$$

while those are evaluated in the pre-state only:

$$\begin{array}{lll} \forall\sigma'. & (\sigma,\sigma') \models & \texttt{p1.boss@pre.salary@pre} \doteq 1200 \\ \forall\sigma'. & (\sigma,\sigma') \models & \texttt{p6.boss@pre.boss@pre} \doteq \texttt{p5} \\ \forall\sigma'. & (\sigma,\sigma') \models & \texttt{p1.boss@pre.boss@pre} \doteq \texttt{null} \\ \forall\sigma'. & (\sigma,\sigma') \not\models \upsilon & \texttt{p2.boss@pre.salary@pre} \end{array}$$

A path expression involving both the pre and post-state is for instance `p6.boss@pre.salary`. The boss of `p6` in the pre-state is `p4` and the salary of `p4` in the post-state is 2900, so we have $(\sigma,\sigma') \models$ `p6.boss@pre.salary` $\doteq$ 2900. As another example, consider the path expression `p2.boss.salary@pre`: in the post-state, `p2` is its own boss, and its salary in the pre-state is 1200, so $(\sigma,\sigma') \models$ `p2.boss.salary@pre` $\doteq$ 1200. Since `p2` has no boss in the pre-state, we also have that $(\sigma,\sigma') \models$ `p2.boss.boss@pre` $\doteq$ `null` and $(\sigma,\sigma') \not\models \upsilon$ `p2.boss@pre.boss`.

We have a particular case with `p5` that does not exist anymore in the post-state, leading to *invalid* when trying to access to the actual value of its salary attribute: $(\sigma,\sigma') \not\models \upsilon$ `p4.boss@pre.salary`.

4.2 Types and Casts

As we already pointed out before, even if only the class `Person` appears in our class model, there are in fact two classes, `Person` and `OclAny`, since `OclAny` is the superclass of all classes. Figure 3(b) shows a state of this model. We consider only one state here, a pre-state being irrelevant for evaluating types.

[5] Note that we omit the `min_salary`, which ensures that salary is not `null`.

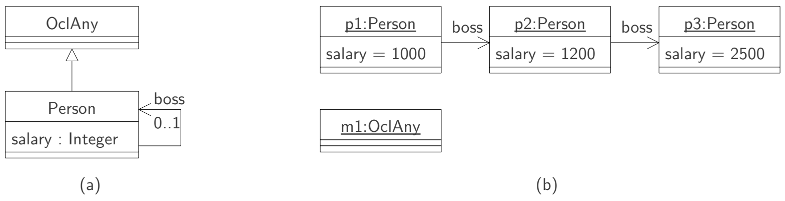

Fig. 3. The whole class model for the management hierarchy and a state σ for it.

As demonstrated in Section 3.3, casting an instance of `Person` up to `OclAny`, then down to `Person` again returns the original object: $(\sigma, \sigma) \models$ `p1.oclAsType(OclAny).oclAsType(Person)` $\doteq$ `p1`. However, casting an instance of `OclAny` down to `Person` is not possible if this instance is not a cast up of an instance of `Person`: $(\sigma, \sigma) \not\models \upsilon$ `m1.oclAsType(Person)`.

We also saw in Section 3.3 that the `oclIsTypeOf` operator checks the static type of an object while `oclIsKindOf` checks its dynamic type. This leads to the following properties:

$$\begin{aligned}
(\sigma, \sigma) &\models \texttt{m1.oclIsTypeOf(OclAny)} \doteq \texttt{true}\\
(\sigma, \sigma) &\models \texttt{m1.oclIsTypeOf(Person)} \doteq \texttt{false}\\
(\sigma, \sigma) &\models \texttt{p1.oclIsTypeOf(Person)} \doteq \texttt{true}\\
(\sigma, \sigma) &\models \texttt{p1.oclIsTypeOf(OclAny)} \doteq \texttt{false}\\
(\sigma, \sigma) &\models \texttt{m1.oclIsKindOf(OclAny)} \doteq \texttt{true}\\
(\sigma, \sigma) &\models \texttt{m1.oclIsKindOf(Person)} \doteq \texttt{false}\\
(\sigma, \sigma) &\models \texttt{p1.oclIsKindOf(OclAny)} \doteq \texttt{true}\\
(\sigma, \sigma) &\models \texttt{p1.oclIsKindOf(Person)} \doteq \texttt{true}
\end{aligned}$$

As expected, casting an instance of `Person` up to `OclAny` does not return an object of static type `OclAny`:

$$(\sigma, \sigma) \models \texttt{p1.oclAsType(OclAny).oclIsTypeOf(OclAny)} \doteq \texttt{false}$$

In Section 3.4, we showed that we defined the `allInstances` operator according to dynamic types. This leads to the following expected property for class `Person`: $(\sigma, \sigma) \models$ `Person.allInstances()` $\doteq$ `Set{p1,p2,p3}`. For class `OclAny`, `allInstances` returns all the instances of `OclAny` and of its child classes, while casting the latter up to `OclAny`, so that the result is a set of instances of `OclAny`:

$$\begin{aligned}
(\sigma, \sigma) \models \texttt{OclAny.allInstances()} \doteq \texttt{Set\{ m1,}&\\
&\texttt{p1.oclAsType(OclAny),}\\
&\texttt{p2.oclAsType(OclAny),}\\
&\texttt{p3.oclAsType(OclAny)\}}.
\end{aligned}$$

5 Related Work and Conclusion

5.1 Related Work

Albeit, there are object-oriented specification languages that support null elements, namely JML [9] or Spec# [2]. Notably, both languages limit null elements to class types and provide a type system supporting non-null types. In the case of JML, the non-null types are even chosen as the default types [7]. Supporting non-null types simplifies the analysis of specifications drastically, as many cases resulting in potential invalid states (e. g., de-referencing a null) are already ruled out by the type system.

5.2 Conclusion and Future Work

We presented a formal semantics for object-oriented data structures that provides the basis for a formalization of OCL and that supports both exception elements: `null` and `invalid`.

The overall goal of Featherweight OCL is to study the details of the various semantical variants of a object-oriented formal specification language: Featherweight OCL contributes to closing the formal gaps as well as the removing inconsistencies in the standard. Ultimately, we aim at providing a machine-checked formal semantics that can be included in the OCL standard, i. e., replacing the current Annex A.

Acknowledgments. The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant no. 257930 (`http://www.aniketos.eu/`). This research work has also been carried out under the leadership of the Technological Research Institute SystemX, and therefore granted with public funds within the scope of the French Program "Investissements d'Avenir" (`http://www.irt-systemx.fr/`).

References

[1] P. B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Kluwer Academic Publishers, 2nd edition, 2002. ISBN 1-402-00763-9.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, volume 3362 of *LNCS*, pages 49–69. Springer, May 25 2005. ISBN 978-3-540-24287-1. doi: 10.1007/b105030.

[3] A. D. Brucker and B. Wolff. An extensible encoding of object-oriented data models in HOL. *Journal of Automated Reasoning*, 41:219–249, 2008. ISSN 0168-7433. doi: 10.1007/s10817-008-9108-3. URL `http://www.brucker.ch/bibliography/abstract/brucker.ea-extensible-2008-b`.

[4] A. D. Brucker and B. Wolff. HOL-OCL – A Formal Proof Environment for UML/OCL. In J. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering (FASE08)*, number 4961 in LNCS, pages 97–100. Springer, 2008. doi: 10.1007/978-3-540-78743-3_8. URL `http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-ocl-2008`.

[5] A. D. Brucker and B. Wolff. Featherweight OCL: A study for the consistent semantics of OCL 2.3 in HOL. In *Workshop on OCL and Textual Modelling (OCL 2012)*, pages 19–24, 2012. ISBN 978-1-4503-1799-3. doi: 10.1145/2428516.2428520. URL `http://www.brucker.ch/bibliography/abstract/brucker.ea-featherweight-2012`.

[6] A. D. Brucker, M. P. Krieger, and B. Wolff. Extending OCL with null-references. In S. Gosh, editor, *Models in Software Engineering*, number 6002 in LNCS, pages 261–275. Springer, 2009. doi: 10.1007/978-3-642-12261-3_25. URL `http://www.brucker.ch/bibliography/abstract/brucker.ea-ocl-null-2009`. Selected best papers from all satellite events of the MoDELS 2009 conference.

[7] P. Chalin and F. Rioux. Non-null references by default in the Java modeling language. In *SAVCBS '05: Proceedings of the 2005 conference on Specification and verification of component-based systems*, page 9. ACM Press, 2005. ISBN 1-59593-371-9. doi: 10.1145/1123058.1123068.

[8] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[9] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML reference manual (revision 1.2), Feb. 2007. Available from `http://www.jmlspecs.org`.

[10] Object Management Group. UML 2.0 OCL specification, Apr. 2006. Available as OMG document formal/06-05-01.

[11] Object Management Group. UML 2.3.1 OCL specification, Feb. 2012. Available as OMG document formal/2012-01-01.

[12] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.