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

Achim D. Brucker, Delphine Longuet, Frédéric Tuong, and Burkhart Wolff

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 [12], 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 [13], 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 value" (i.e., null), or valid and represent a regular value (i.e., an integer value...
Fig. 1. A simple design model capturing a management hierarchy in a company 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]. This paper is a short version that only introduces the formalization using a small running example. An extended version of this paper is available as technical report [7]. The underlying formalization of object-oriented datatypes 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, 9] 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 α :: order or α :: bot) or type constructors. Type constructors may have arguments (as in α list or α set). The type constructor for the function space ⇒ is written infix: α ⇒ β; multiple applications like τ₁ ⇒ (... ⇒ (τₙ ⇒ τₙ₊₁)... ) have the alternative syntax [τ₁, ..., τₙ] ⇒ τₙ₊₁. HOL is centered around the extensional logical equality _ =_ with type [α, α] ⇒ bool, where bool is the fundamental logical type. We use infix notation: instead of (_ =_ ) E₁ E₂ we write E₁ = E₂. The logical connectives _ ∧_, _ ∨_, _ ⇒_ have type [bool, bool] ⇒ bool, _¬_ has type bool ⇒ bool. The quantifiers _∀_ and _∃_ have type [α ⇒ bool] ⇒ bool. The quantifiers may range over types of higher order, i.e., functions or sets. The definition of the element-hood _ ∈_ , the set comprehension {_ _}, as well as _ _∪_ _ and _ _∩_ _ are standard.

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 \( \tau \bot := \bot | \_\bot \) that assigns to each type \( \tau \) a type \( \tau \bot \) disjointly extended by the exceptional element \( \bot \). The function \( \_\bot : \alpha \Rightarrow \alpha \) is the inverse of \( \_\bot \) (unspecified for \( \bot \)). Partial functions \( \alpha \Rightarrow \beta \) are defined as functions \( \alpha \Rightarrow \beta \bot \) supporting the usual concepts of domain (\( \text{dom} \)) and range (\( \text{ran} \)). 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 \( \text{Set}(_) \), 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 \( (\sigma,\sigma') \) 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 \( \text{null} \) that contains two distinguishable elements \( \text{bot} \) and \( \text{null} \). Any type of the form \( \alpha \bot \bot \) is an instance of this type class with \( \text{bot} \equiv \bot \) and \( \text{null} \equiv \_\bot \). Now, any OCL type can be represented by an HOL type of the form:

\[
V(\alpha) := \text{state} \times \text{state} \Rightarrow \alpha :: \text{null}
\]

We define

\[
I[\text{invalid} :: V(\alpha)] \tau = \text{bot} \quad I[\text{null} :: V(\alpha)] \tau = \text{null} \\
I[\text{true} :: \text{Boolean}] \tau = \_\text{true}_\tau \\
I[\text{false}] \tau = \_\text{false}_\tau \\
I[X.\text{oclIsUndefined()}] \tau = (\text{if } I[X] \tau \in \{\text{bot, null} \} \text{ then } I[\text{true}] \tau \text{ else } I[\text{false}] \tau)
\]

where \( I[E] \) is the semantic interpretation function commonly used in mathematical textbooks and \( \tau \) stands for pairs of pre- and post state \( (\sigma,\sigma') \). 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 \( \text{not } X.\text{oclIsUndefined()} \) and \( \nu X \) for \( \text{not } X.\text{oclIsInvalid()} \) 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
type 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: $C_i \text{allInstances}()$ and $C_i \text{allInstances@pre}()$,
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
$\_\_a :: C_i \rightarrow A$ and $\_\_a@pre :: C_i \rightarrow A$ for $A \in \{ V(\ldots), C_1, ..., C_n \}$),
4. type casts that can change the static type of an object of a class (denoted
$X.oclAsType(C_i)$ of type $C_j \rightarrow C_i$)
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 $\_\_ \equiv \_\_$).

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:

\[
\text{typedef } \alpha \text{ state } := \{ \sigma :: \text{oid} \rightarrow \alpha \mid \text{inv}_\sigma(\sigma) \}
\]

where inv$_\sigma$ is a to be discussed invariant on states. The key point is that we need
a common type $\alpha$ for the set of all possible object representations. Object repre-
sentations 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 fundam-
mentally 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.

**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:

\[
\begin{align*}
\text{datatype} & \quad \text{oclany} = \text{mk\text{OclAny}} \ (\text{int}_\bot \times \text{oid}_\bot) \bot \\
\text{datatype} & \quad \text{person} = \text{mk\text{Person}} \ (\text{int}_\bot) \ (\text{oid}_\bot) \\
\text{datatype} & \quad \text{A} = \text{in\text{Person}} \ \text{person} | \text{in\text{OclAny}} \ \text{oclany}
\end{align*}
\]

Here, \((\text{int}_\bot \times \text{oid}_\bot)_\bot\) 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.

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

\[
\begin{align*}
\text{definition} & \quad \text{select\_salary} \ f = (\lambda \text{mk\text{Person}} \ \_ \ \_ \_ \Rightarrow \text{null} \\
& \quad \quad | \text{mk\text{Person}} \ \_ \ _ \ s \_ \Rightarrow f (\lambda x \ _ \ . \ x) s) \\
\text{select\_boss} \ f & = (\lambda \text{mk\text{Person}} \ \_ \ _ \ \_ \_ \Rightarrow \text{null} \\
& \quad | \text{mk\text{Person}} \ _ \ _ \ b \_ \Rightarrow f (\lambda x \ _ \ . \ x) b)
\end{align*}
\]
Which gives the top-level definitions:

```plaintext
definition .salary :: Person ⇒ Integer
where   X.salary = eval_extract X (deref_oidPerson in_post_state
           (select_salary reconst _basetype))

definition .boss :: Person ⇒ Person
where   X.boss = eval_extract X (deref_oidPerson in_post_state
           (select_boss (deref_oidPerson in_post_state)))
```

### 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.

**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:

```plaintext
defs (overloaded) (X :: OclAny).oclIsTypeOf(Person) ≡ (λτ. case Xτ of
             | ⊥               ⇒ invalid τ
             | ⊥                ⇒ true τ
             | ⊥ mkOclAny ⊥ ⊥ ⇒ false τ
             | ⊥ mkOclAny ⊥ ⊥ ⇒ true τ
)
```

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

```plaintext
consts .oclAsType(OclAny) :: α ⇒ OclAny
consts .oclAsType(Person) :: α ⇒ Person
```

whose instances were provided, for example, by:

```plaintext
defs (overloaded) (X :: OclAny).oclAsType(Person) ≡ (λτ. case Xτ of
             | ⊥                 ⇒ invalid τ
             | ⊥                ⇒ null τ
             | ⊥ mkOclAny ⊥ ⊥   ⇒ invalid τ
             | ⊥ mkOclAny oid (a,b) ⊥ ⇒ mkPerson oid a b ⊥)
```

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

\[ \tau \mid\mid (X :: OclAny).oclIsTypeOf(OclAny) \implies \tau \mid\mid \delta X \]
\[ (X : \text{Person}).\text{oclAsType(oclAny)}.\text{oclAsType(Person)} = X \]

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

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 \( \sigma \), while the state on the right is used as a post-state and is called \( \sigma' \).

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

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

For instance, we have \((\sigma, \sigma') \models p1.\text{salary} \triangleq 1300\), since the attribute
salary of object \( p1 \) has the value 1300 in the post-state. We also have \((\sigma, \sigma') \models p1.\text{salary}@\text{pre} \triangleq 1000\) since \( p1 \) also existed in the pre-state and its salary was
1000. In the same way, we have \((\sigma, \sigma') \models p6.\text{boss} \triangleq p7 \) since \( p7 \) is the boss of
\( p6 \) in the post-state, while \((\sigma, \sigma') \models p6.\text{boss}@\text{pre} \triangleq p4 \) 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 \((\sigma, \sigma') \models p3.\text{salary} \triangleq \text{null} \). It also has no boss so \((\sigma, \sigma') \models p3.\text{boss} \triangleq \text{null} \). Trying to de-referenciate a null association end yields an invalid
value, so \((\sigma, \sigma') \not\models v p3.\text{boss}.\text{salary} \). 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: \((\sigma, \sigma') \not\models v p3.\text{boss}@\text{pre} \), and \((\sigma, \sigma') \not\models v p3.\text{boss}@\text{pre} \).

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

\[ \text{Note that we omit the min\_salary, which ensures that salary is not null.} \]
remains easily comprehensible. For instance, the following formulas are evaluated in the post-state only:

\[ \forall \sigma. (\sigma, \sigma') \models p_1.boss.salary = 1800 \]
\[ \forall \sigma. (\sigma, \sigma') \models p_1.boss.boss = p_2 \]
\[ \forall \sigma. (\sigma, \sigma') \models p_7.boss.salary = 3200 \]

while those are evaluated in the pre-state only:

\[ \forall \sigma'. (\sigma, \sigma') \models p_1.boss.pre.salary = 1200 \]
\[ \forall \sigma'. (\sigma, \sigma') \models p_6.boss.pre.boss = p_5 \]
\[ \forall \sigma'. (\sigma, \sigma') \models p_1.boss.pre.boss = \text{null} \]
\[ \forall \sigma'. (\sigma, \sigma') \not\models p_2.boss.pre.salary \]

A path expression involving both the pre and post-state is for instance \( p_6.boss@pre.salary \). The boss of \( p_6 \) in the pre-state is \( p_4 \) and the salary of \( p_4 \) in the post-state is 2900, so we have \( (\sigma, \sigma') \models p_6.boss@pre.salary = 2900 \).

As another example, consider the path expression \( p_2.boss.salary@pre \): in the post-state, \( p_2 \) is its own boss, and its salary in the pre-state is 1200, so \( (\sigma, \sigma') \models p_2.boss.salary@pre = 1200 \). Since \( p_2 \) has no boss in the pre-state, we also have that \( (\sigma, \sigma') \models p_2.boss.boss@pre = \text{null} \) and \( (\sigma, \sigma') \not\models p_2.boss@pre.boss \).

We have a particular case with \( p_5 \) that does not exist anymore in the post-state, leading to *invalid* when trying to access to the actual value of its salary attribute: \( (\sigma, \sigma') \not\models p_4.boss@pre.salary \).

### 4.2 Types and Casts

As we already pointed out before, even if only the class \texttt{Person} appears in our class model, there are in fact two classes, \texttt{Person} and \texttt{OclAny}, since \texttt{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.

![Class Model](image)

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

As demonstrated in Section 3.3, casting an instance of \texttt{Person} up to \texttt{OclAny}, then down to \texttt{Person} again returns the original object: \( (\sigma, \sigma') \models p_1.oclAsType(OclAny).oclAsType(Person) = p_1 \). However, casting an instance of \texttt{OclAny} down to \texttt{Person} is not possible if this instance is not a cast up of an instance of \texttt{Person}: \( (\sigma, \sigma') \not\models \nu m_1.oclAsType(Person) \).
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:

\[(\sigma, \sigma') \models m1.oclIsTypeOf(OclAny) \equiv true\]
\[(\sigma, \sigma') \models m1.oclIsTypeOf(Person) \equiv false\]
\[(\sigma, \sigma') \models p1.oclIsTypeOf(Person) \equiv true\]
\[(\sigma, \sigma') \models p1.oclIsTypeOf(OclAny) \equiv false\]
\[(\sigma, \sigma') \models m1.oclIsKindOf(OclAny) \equiv true\]
\[(\sigma, \sigma') \models m1.oclIsKindOf(Person) \equiv false\]
\[(\sigma, \sigma') \models p1.oclIsKindOf(OclAny) \equiv true\]
\[(\sigma, \sigma') \models p1.oclIsKindOf(Person) \equiv true\]

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

\[(\sigma, \sigma') \models p1.oclAsType(OclAny).oclIsTypeOf(OclAny) \equiv false\]

5 Related Work and Conclusion

5.1 Related Work

Albeit, there are object-oriented specification languages that support null elements, namely JML [10] 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 [8]. 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 an 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