Featherweight OCL

A study for the consistent semantics of OCL 2.3 in HOL

Achim D. Brucker
SAP AG, SAP Research
Vincenz-Priessnitz-Str. 1
76131 Karlsruhe, Germany
achim.brucker@sap.com

Burkhart Wolff
Laboratoire de Recherche en Informatique (LRI)
Bât 650, Université Paris-Sud
91405 Orsay Cedex, France
wolff@lri.fr

ABSTRACT

At its origins, OCL was conceived as a strict semantics for undefinedness, with the exception of the logical connectives of type `Boolean` that constitute a three-valued propositional logic. Recent versions of the OCL standard added a second exception element, which, similar to the null references in programming languages, is given a non-strict semantics.

In this paper, we report on our results in formalizing the core of OCL in higher-order logic (HOL). This formalization revealed several inconsistencies and contradictions in the current version of the OCL standard. These inconsistencies and contradictions are reflected in the challenge to define and implement OCL tools (e. g., interpreters, code-generators, or theorem provers) in a uniform manner.

Categories and Subject Descriptors

D3.1.1 [**Software**]: Programming Languages—*Formal Definitions and Theory*

General Terms

Languages, Standardization, Theory

Keywords

OCL, HOL-OCL, formal semantics

1. INTRODUCTION

At its origins [16, 19], OCL was conceived as a strict semantics for undefinedness, with the exception of the logical connectives of type `Boolean` that constitute a three-valued propositional logic. Recent versions of the OCL standard [17, 18] added a second exception element, which, similar to the null references in programming languages, is given a non-strict semantics. Unfortunately, this extension results in several inconsistencies and contradictions. These problems are reflected in difficulties to define interpreters, code-generators, specification animators or theorem provers for OCL in a uniform manner and resulting incompatibilities of various tools. For the OCL community, this results in the challenge to define a new formal semantics definition OCL that could replace the "Annex A" of the OCL standard [18].

In the paper "Extending OCL with Null-References" [10] we explored—based on mathematical arguments and paper and pencil proofs—a consistent formal semantics that comprises two exception elements: `invalid` ("bottom" in semantics terminology) and `null` (for "non-existing element").

This paper is based on a formalization of [10], called "Featherweight OCL," in Isabelle/HOL [15]. This formalization is in its present form merely a semantical study and a proof of technology than a real tool. It focuses on the formalization of the key semantical constructions, i. e., the type `Boolean` and the logic, the type `Integer` and a standard strict operator library, and the collection type `Set(A)` with quantifiers, iterators and key operators.

The rest of this paper summarizes our experiences and findings in formalizing a core of OCL 2.3 in Isabelle/HOL. Thus, this paper serves as an extended abstract of the detailed documents that are available at `http://www.brucker.ch/projects/hol-ocl/Featherweight-OCL/`.

2. DESIGN DECISIONS

Featherweight OCL is a formalization of the core of OCL aiming at formally investigation the relationship between the different notions of "undefinedness," i. e., `invalid` and `null`. As such, it does not attempt to define the complete OCL library. Instead, it concentrates on the core concepts of OCL as well as the types `Boolean`, `Integer`, and typed sets (`Set(T)`). Following the tradition of HOL-OCL [5, 7], Featherweight OCL is based on the following principles:

1. It is an embedding into a powerful semantic meta-language and environment, namely Isabelle/HOL [15].
2. It is a *shallow embedding* in HOL; types in OCL were injectively mapped to types in Featherweight OCL. Ill-typed OCL specifications cannot be represented in Featherweight OCL and a type in Featherweight OCL contains exactly the values that are possible in OCL. Thus, sets may contain `null` (`Set{null}` is a defined set) but not `invalid` (`Set{invalid}` is just `invalid`).
3. Any Featherweight OCL type contains at least `invalid` and `null` (the type `Void` contains only these instances). The logic is consequently four-valued.
4. It is a strongly typed language in the Hindley-Milner tradition. We assume that a pre-process eliminates all implicit conversions due to subtyping by introducing explicit casts (e. g., `oclAsType()`). The details of such

a pre-processing are described in [4]. Casts are semantic functions, typically injections, that may convert data between the different Featherweight OCL types.

5. All objects are represented in an object universe in the HOL-OCL tradition [6]. Our universe construction also gives semantics to type casts, dynamic type tests, as well as functions such as `allInstances()`.
6. Featherweight OCL types may be arbitrarily nested, e. g., `Set{Set{1,2}} = Set{Set{2,1}}` is legal and true. Since there is a `null`-element in the type `Set(A)`, the set `Set{null, Set{3}}` is a legal expression of type `Set(Set(Integer))`.
7. For demonstration purposes, the set-type in Featherweight OCL may be infinite, allowing infinite quantification and a constant that contains the set of all Integers. Arithmetic laws like commutativity may therefore expressed in OCL itself. The iterator is only defined on finite sets.
8. It supports equational reasoning and congruence reasoning, but this requires a differentiation of the different equalities like strict equality, strong equality, meta-equality (HOL). Strict equality and strong equality require a subcalculus, "cp" (a detailed discussion of the different equalities as well the subcalculus "cp"—for three-valued OCL 2.0—is given in [8]).

3. FORMAL FOUNDATION

Higher-order Logic (HOL) [1, 12] 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. Pragmatically, HOL can be viewed as "Haskell with Quantifiers."

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 $\alpha :: order$ or $\alpha :: \text{bot}$) or *type constructors*. Type constructors may have arguments (as in α list or α 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 $[\alpha, \alpha] \Rightarrow \text{bool}$, 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 $[\text{bool}, \text{bool}] \Rightarrow \text{bool}$, $\neg\_$ has type $\text{bool} \Rightarrow \text{bool}$. The quantifiers $\forall \_. \_$ and $\exists \_. \_$ have type $[\alpha \Rightarrow \text{bool}] \Rightarrow \text{bool}$. 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.

Isabelle is a theorem is 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 $\tau_\bot := \bot \mid \lfloor\_\rfloor : \alpha$ that assigns to each type τ a type $\tau_\bot$ *disjointly extended* by the exceptional element $\bot$. The function $\lceil\_\rceil : \alpha_\bot \Rightarrow \alpha$ is the inverse of $\lfloor\_\rfloor$ (unspecified for $\bot$). Partial functions $\alpha \rightharpoonup \beta$ are defined as functions $\alpha \Rightarrow \beta_\bot$ supporting the usual concepts of domain (dom $\_$) and range (ran $\_$). The library is built entirely by logically safe, conservative definitions and derived rules. This methodology is also applied to HOL-OCL [7] and Featherweight OCL.

4. THE THEORY ORGANIZATION

The semantic theory is organized in a quite conventional manner in three layers. The first layer, called the *denotational semantics* comprises a set of definitions of the operators of the language. Presented as *definitional axioms* inside Isabelle/HOL, this part assures the logically consistency of the overall construction. The second layer, called *logical layer*, is derived from the former and centered around the notion of validity of an OCL formula P for a state-transition from pre-state σ to post-state σ', validity statements were written $(\sigma, \sigma') \models P$. The third layer, called *algebraic layer*, also derived from the former layers, tries to establish a number of algebraic laws of the form $P = P'$; such laws are amenable to equational reasoning and also help for automated reasoning and code-generation.

For space reasons, we will restrict ourselves in this paper to a few operators and make a traversal through all three layers in order to give a high-level description of our formalization. Especially, the details of the semantic construction for sets and the handling of objects and object universes were excluded from a presentation here.

4.1 Denotational Semantics

OCL is composed of 1) operators on built-in data structures such as Boolean, Integer or Set(A), 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*) that depends on the pair (σ, σ') of pre-and post-state. The precise form of states is irrelevant for this paper (compare [10]) 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 $(\alpha_\bot)_\bot$ is an instance of this type-class with $\text{bot} \equiv \bot$ and $\text{null} \equiv \lfloor\bot\rfloor$. Now, any OCL type can be represented by an HOL type of the form:

$$V(\alpha) := \text{state} \times \text{state} \Rightarrow \alpha :: \text{null}\ .$$

On this basis, we define $V((\text{bool}_\bot)_\bot)$ as the HOL type for the OCL type `Boolean` by and define:

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

$$I[\![\texttt{true} :: \texttt{Boolean}]\!]\tau = \lfloor\lfloor\text{true}\rfloor\rfloor \qquad I[\![\texttt{false}]\!]\tau = \lfloor\lfloor\text{false}\rfloor\rfloor$$

$$I[\![X\texttt{.oclIsUndefined()}]\!]\tau = (\text{if } I[\![X]\!]\tau \in \{\text{bot}, \text{null}\} \text{ then } I[\![\texttt{true}]\!]\tau \text{ else } I[\![\texttt{false}]\!]\tau)$$

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

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 checks 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.

On this basis, one can define the core logical operators `not` and `and` as follows:

$$
\begin{array}{lll}
I[\![\texttt{not}\ X]\!]\tau & = (\text{case}\ I[\![X]\!]\tau\ \text{of} & \\
 & \quad \bot & \Rightarrow \bot \\
 & \quad \lfloor\lfloor\bot\rfloor\rfloor & \Rightarrow \lfloor\bot\rfloor \\
 & \quad \lfloor\lfloor\lfloor x\rfloor\rfloor\rfloor & \Rightarrow \lfloor\lfloor\neg x\rfloor\rfloor) \\
I[\![X\ \texttt{and}\ Y]\!]\tau & = (\text{case}\ I[\![X]\!]\tau\ \text{of} & \\
 & \quad \bot & \Rightarrow (\text{case}\ I[\![Y]\!]\tau\ \text{of} \\
 & & \quad \bot \Rightarrow \bot \\
 & & \quad \lfloor\lfloor\bot\rfloor\rfloor \Rightarrow \bot \\
 & & \quad \lfloor\lfloor\lfloor\text{true}\rfloor\rfloor\rfloor \Rightarrow \bot \\
 & & \quad \lfloor\lfloor\lfloor\text{false}\rfloor\rfloor\rfloor \Rightarrow \lfloor\lfloor\text{false}\rfloor\rfloor) \\
 & \quad \lfloor\lfloor\bot\rfloor\rfloor & \Rightarrow (\text{case}\ I[\![Y]\!]\tau\ \text{of} \\
 & & \quad \bot \Rightarrow \bot \\
 & & \quad \lfloor\lfloor\bot\rfloor\rfloor \Rightarrow \lfloor\lfloor\bot\rfloor\rfloor \\
 & & \quad \lfloor\lfloor\lfloor\text{true}\rfloor\rfloor\rfloor \Rightarrow \lfloor\lfloor\bot\rfloor\rfloor \\
 & & \quad \lfloor\lfloor\lfloor\text{false}\rfloor\rfloor\rfloor \Rightarrow \lfloor\lfloor\text{false}\rfloor\rfloor) \\
 & \quad \lfloor\lfloor\lfloor\text{true}\rfloor\rfloor\rfloor & \Rightarrow (\text{case}\ I[\![Y]\!]\tau\ \text{of} \\
 & & \quad \bot \Rightarrow \bot \\
 & & \quad \lfloor\lfloor\bot\rfloor\rfloor \Rightarrow \lfloor\lfloor\bot\rfloor\rfloor \\
 & & \quad \lfloor\lfloor\lfloor y\rfloor\rfloor\rfloor \Rightarrow \lfloor\lfloor\lfloor y\rfloor\rfloor\rfloor) \\
 & \quad \lfloor\lfloor\lfloor\text{false}\rfloor\rfloor\rfloor & \Rightarrow \lfloor\lfloor\text{false}\rfloor\rfloor)
\end{array}
$$

These non-strict operations were used to define the other logical connectives in the usual classical way: $X\ \texttt{or}\ Y \equiv (\texttt{not}\ X)\ \texttt{and}\ (\texttt{not}\ Y)$ or $X\ \texttt{implies}\ Y \equiv (\texttt{not}\ X)\ \texttt{or}\ Y$.

The default semantics for an OCL library operator is strict semantics; this means that the result of an operation f is invalid if one of its arguments is invalid. For a semantics comprising null, we suggest to stay conform to the standard and define the addition for integers as follows:

$$
\begin{array}{ll}
I[\![x\texttt{+}y]\!]\tau = & \text{if}\ I[\![\delta\ x]\!]\tau = \lfloor\lfloor\text{true}\rfloor\rfloor \wedge I[\![\delta\ y]\!]\tau = \lfloor\lfloor\text{true}\rfloor\rfloor \\
 & \text{then}\lfloor\lfloor\lceil\lceil I[\![x]\!]\tau\rceil\rceil + \lceil\lceil I[\![y]\!]\tau\rceil\rceil\rfloor\rfloor \\
 & \text{else}\ \bot
\end{array}
$$

where the operator "+" on the left-hand side of the equation denotes the OCL addition of type $[V((\text{int}_\bot)_\bot), V((\text{int}_\bot)_\bot)] \Rightarrow V((\text{int}_\bot)_\bot)$ while the "+" on the right-hand side of the equation of type $[\text{int}, \text{int}] \Rightarrow \text{int}$ denotes the integer-addition from the HOL library.

4.2 Logical Layer

The topmost goal of the logic for OCL is to define the *validity statement*:

$$(\sigma, \sigma') \models P\,,$$

where σ is the pre-state and σ' the post-state of the underlying system and P is a formula. Informally, a formula P is valid if and only if its evaluation in (σ, σ') (i. e., τ for short) yields true. Formally this means:

$$\tau \models P \equiv \big(I[\![P]\!]\tau = \lfloor\lfloor\text{true}\rfloor\rfloor\big)\,.$$

On this basis, classical, two-valued inference rules can be established for reasoning over the logical connective, the different notions of equality, definedness and validity. Generally speaking, rules over logical validity can relate bits and pieces in various OCL terms and allow—via strong logical equality discussed below—the replacement of semantically equivalent sub-expressions. The core inference rules are:

$$\tau \models \texttt{true} \quad \neg(\tau \models \texttt{false}) \quad \neg(\tau \models \texttt{invalid}) \quad \neg(\tau \models \texttt{null})$$

$$\tau \models \texttt{not}\ P \Longrightarrow \tau\neg \models P$$

$$\tau \models P\ \texttt{and}\ Q \Longrightarrow \tau \models P \qquad \tau \models P\ \texttt{and}\ Q \Longrightarrow \tau \models Q$$

$$\tau \models P \Longrightarrow (\texttt{if}\ P\ \texttt{then}\ B_1\ \texttt{else}\ B_2\ \texttt{endif})\tau = B_1\tau$$

$$\tau \models \texttt{not}\ P \Longrightarrow (\texttt{if}\ P\ \texttt{then}\ B_1\ \texttt{else}\ B_2\ \texttt{endif})\tau = B_2\tau$$

$$\tau \models P \Longrightarrow \tau \models \delta P \qquad \tau \models (\delta X) \Longrightarrow \tau \models \upsilon X$$

By the latter two properties it can be inferred that any valid property P (so for example: a valid invariant) is actually defined, which allows to infer for terms composed by strict operations that their arguments and finally the variables occurring in it are valid or defined.

We propose to distinguish the *strong logical equality* (written $\_ \triangleq \_$), which follows the general principle that "equals can be replaced by equals," from the *strict referential equality* (written $\_ \doteq \_$), which is an object-oriented concept that attempts to approximate and to implement the former. Strict referential equality, which is the default in the OCL language and is written simply `_ = _` in the standard, is an overloaded concept and has to be defined for each OCL type individually; for objects resulting from class definitions, it is implemented by simply comparing the references to the objects. In contrast, strong logical equality is a polymorphic concept which is defined once and for all by:

$$I[\![X \triangleq Y]\!]\tau \equiv \lfloor\lfloor I[\![X]\!]\tau = I[\![Y]\!]\tau\rfloor\rfloor$$

It enjoys nearly the laws of a congruence:

$$\tau \models (x \triangleq x)$$

$$\tau \models (x \triangleq y) \Longrightarrow \tau \models (y \triangleq x)$$

$$\tau \models (x \triangleq y) \Longrightarrow \tau \models (y \triangleq z) \Longrightarrow \tau \models (x \triangleq z)$$

$$\text{cp}\,P \Longrightarrow \tau \models (x \triangleq y) \Longrightarrow \tau \models (P\,x) \Longrightarrow \tau \models (P\,y)$$

where the predicate cp stands for *context-passing*, a property that is characterized by $P(X)$ equals $\lambda\tau.\ P(\lambda\_.\ X\tau)\tau$. It means that the state tuple $\tau = (\sigma, \sigma')$ is passed unchanged from surrounding expressions to sub-expressions. it is true for all pure OCL expressions (but not arbitrary mixtures of OCL and HOL) in Featherweight OCL. The necessary side-calculus for establishing cp can be fully automated.

The logical layer of the Featherweight OCL rules gives also a means to convert an OCL formula living in its for-valued world into a representation that is classically two-valued and can be processed by standard SMT solvers such as CVC3 [2] or Z3 [13]. Delta-closure rules for all logical connectives have the following format, e. g.:

$$\tau \models \delta x \Longrightarrow (\tau \models\ \texttt{not}\ x) = (\neg(\tau \models x))$$

$$\tau \models \delta x \Longrightarrow \tau \models \delta y \Longrightarrow (\tau \models x\ \texttt{and}\ y) = (\tau \models x \wedge \tau \models y)$$

$$\tau \models \delta x \Longrightarrow \tau \models \delta y \Longrightarrow (\tau \models (x\ \texttt{implies}\ y)) = ((\tau \models x) \longrightarrow (\tau \models y))$$

Together with the general case-distinction

$$\tau \models \delta x \vee \tau \models x \triangleq \texttt{invalid} \vee \tau \models x \triangleq \texttt{null}\,,$$

which is possible for any OCL type, a case distinction on the variables in a formula can be performed; due to strictness rules, formulae containing somewhere a variable x that is known to be `invalid` or `null` reduce usually quickly to contradictions. For example, we can infer from an invariant $\tau \models x \doteq y\texttt{-}3$ that we have actually $\tau \models x \doteq y\texttt{-}3 \wedge \tau \models \delta x \wedge \tau \models \delta y$. We call the latter formula the δ-closure of the former. Now, we can convert a formula like $\tau \models x\texttt{>}0$ or $3\texttt{*}y\texttt{>}x\texttt{*}x$ into the equivalent formula $\tau \models x > 0 \vee \tau \models 3\texttt{*}y\texttt{>}x\texttt{*}x$ and thus

internalize the OCL-logic into a classical (and more tool-conform) logic. This works—for the price of a potential, but due to the usually "rich" δ-closures of invariants rare—exponential blow-up of the formula for all OCL formulas.

4.3 Algebraic Layer

Based on the logical layer, we build a system with simpler rules which are amenable to automated reasoning. We restrict ourselves to pure equations on OCL expressions, where the used equality is the meta-(HOL-)equality.

Our denotational definitions on `not` and `and` can be reformulated in the following ground equations:

$$\upsilon\ \texttt{invalid} = \texttt{false} \qquad \upsilon\ \texttt{null} = \texttt{true}$$
$$\upsilon\ \texttt{true} = \texttt{true} \qquad \upsilon\ \texttt{false} = \texttt{true}$$
$$\delta\ \texttt{invalid} = \texttt{false} \qquad \delta\ \texttt{null} = \texttt{false}$$
$$\delta\ \texttt{true} = \texttt{true} \qquad \delta\ \texttt{false} = \texttt{true}$$
$$\texttt{not invalid} = \texttt{invalid} \qquad \texttt{not null} = \texttt{null}$$
$$\texttt{not true} = \texttt{false} \qquad \texttt{not false} = \texttt{true}$$
$$(\texttt{null and true}) = \texttt{null} \qquad (\texttt{null and false}) = \texttt{false}$$
$$(\texttt{null and null}) = \texttt{null} \qquad (\texttt{null and invalid}) = \texttt{invalid}$$
$$(\texttt{false and true}) = \texttt{false} \qquad (\texttt{false and false}) = \texttt{false}$$
$$(\texttt{false and null}) = \texttt{false} \qquad (\texttt{false and invalid}) = \texttt{false}$$
$$(\texttt{true and true}) = \texttt{true} \qquad (\texttt{true and false}) = \texttt{false}$$
$$(\texttt{true and null}) = \texttt{null} \qquad (\texttt{true and invalid}) = \texttt{invalid}$$
$$(\texttt{invalid and true}) = \texttt{invalid}$$
$$(\texttt{invalid and false}) = \texttt{false}$$
$$(\texttt{invalid and null}) = \texttt{invalid}$$
$$(\texttt{invalid and invalid}) = \texttt{invalid}$$

On this core, the structure of a conventional lattice arises:

$$X\ \texttt{and}\ X = X \qquad X\ \texttt{and}\ Y = Y\ \texttt{and}\ X$$
$$\texttt{false and}\ X = \texttt{false} \qquad X\ \texttt{and false} = \texttt{false}$$
$$\texttt{true and}\ X = X \qquad X\ \texttt{and true} = X$$
$$X\ \texttt{and}\ (Y\ \texttt{and}\ Z) = X\ \texttt{and}\ Y\ \texttt{and}\ Z$$

as well as the dual equalities for `or` and the De Morgan rules. This wealth of algebraic properties makes the understanding of the logic easier as well as automated analysis possible: it allows for, for example, computing a DNF of invariant systems (by clever term-rewriting techniques) which are a prerequisite for δ-closures.

The above equations explain the behavior for the most-important non-strict operations. The clarification of the exceptional behaviors is of key-importance for a semantic definition the standard and the major deviation point from HOL-OCL [5, 7], to Featherweight OCL as presented here. The standard expresses at many places that most operations are strict, i. e., enjoy the properties (exemplary for `_ + _`):

$$\texttt{invalid} + x = \texttt{invalid} \qquad \texttt{x + invalid} = \texttt{invalid}$$
$$x + \texttt{null} = \texttt{invalid} \qquad \texttt{null} + x = \texttt{invalid}$$
$$\texttt{null.asType}(X) = \texttt{invalid}$$

besides "classical" exceptional behavior:

$$\texttt{1 / 0} = \texttt{invalid} \qquad \texttt{1 / null} = \texttt{invalid}$$
$$\texttt{null->isEmpty()} = \texttt{true}$$

Moreover, there is also the proposal to use `null` as a kind of "don't know" value for all strict operations, not only in the semantics of the logical connectives. Expressed in algebraic equations, this semantic alternative (this is *not* Featherweight OCL at present) would boil down to:

$$\texttt{invalid} + x = \texttt{invalid} \qquad x + \texttt{invalid} = \texttt{invalid}$$
$$x + \texttt{null} = \texttt{null} \qquad \texttt{null} + x = \texttt{null}$$
$$1/0 = \texttt{invalid} \qquad 1/\texttt{null} = \texttt{null}$$
$$\texttt{null->isEmpty()} = \texttt{null} \qquad \texttt{null.asType}(X) = \texttt{null}$$

While this is logically perfectly possible, while it can be argued that this semantics is "intuitive," and although we do not expect a too heavy cost in deduction when computing δ-closures, we object that there are other, also "intuitive" interpretations that are even more wide-spread: In classical spreadsheet programs, for example, the semantics tend to interpret `null` (representing empty cells in a sheet) as the neutral element of the type, so `0` or the empty string, for example.[1] This semantic alternative (this is *not* Featherweight OCL at present) would yield:

$$\texttt{invalid} + x = \texttt{invalid} \qquad x + \texttt{invalid} = \texttt{invalid}$$
$$x + \texttt{null} = x \qquad \texttt{null} + x = x$$
$$1/0 = \texttt{invalid} \qquad 1/\texttt{null} = \texttt{invalid}$$
$$\texttt{null->isEmpty()} = \texttt{true} \qquad \texttt{null.asType}(X) = \texttt{invalid}$$

Algebraic rules are also the key for execution and compilation of Featherweight OCL expressions. We derived, e. g.:

$$\delta\ \texttt{Set\{\}} = \texttt{true}$$
$$\delta\ (X\texttt{->including}(x)) = \delta X\ \texttt{and}\ \delta x$$
$$\texttt{Set\{\}->includes}(x) = (\texttt{if}\ \upsilon\ x\ \texttt{then false else invalid endif})$$
$$(X\texttt{->including}(x)\texttt{->includes}(y)) = (\texttt{if}\ \delta\ X\ \texttt{then if}\ x \doteq y\ \texttt{then true else}\ X\texttt{->includes}(y)\ \texttt{endif else invalid endif})$$

As `Set{1,2}` is only syntactic sugar for

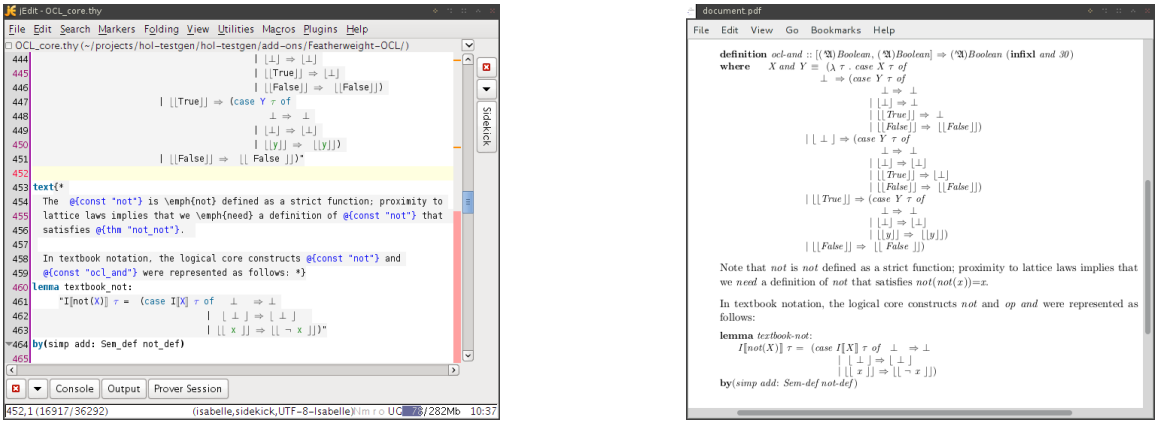
```
Set{}->including(1)->including(2)
```

an expression like `Set{1,2}->includes(null)` becomes automatically decidable in Featherweight OCL by a combination of rewriting and code-generation and execution. The generated documentation from the theory files can thus be enriched by numerous "test-statements" like:

value "$\tau \models (\texttt{Set\{Set\{2, null\}}\} \doteq \texttt{Set\{Set\{null, 2\}}\})$"

which have been machine-checked and which present a high-level and in our opinion fairly readable information for OCL tool manufactures and users.

[1] In spreadsheet programs the interpretation of null varies from operation to operation; e. g., the `average` function treats `null` as non-existing value and not as `0`.



(a) The Isabelle jEdit environment.

(b) The generated formal document.

Figure 1: Generating documents with guaranteed syntactical and semantical consistency.

5. A MACHINE-CHECKED ANNEX A

Isabelle, as a framework for building formal tools [21], provides the means for generating *formal documents*. With formal documents we refer to documents that are machine-generated and ensure certain formal guarantees. In particular, all formal content (e. g., definitions, formulae, types) are checked for consistency during the document generation.

For writing documents, Isabelle supports the embedding of informal texts using a LaTeX-based markup language within the theory files. To ensure the consistency, Isabelle supports to use, within these informal texts, *antiquotations* that refer to the formal parts and that are checked while generating the actual document as PDF. For example, in an informal text, the antiquotation @{thm "not_not"} will instruct Isabelle to lock-up the (formally proven) theorem of name ocl_not_not and to replace the antiquotation with the actual theorem, i. e., `not (not x) = x`.

Figure 1 illustrates this approach: Figure 1a shows the jEdit-based development environment of Isabelle with an excerpt of one of the core theories of Featherweight OCL. Figure 1b shows the generated PDF document where all antiquotations are replaced. Moreover, the document generation tools allows for defining syntactic sugar as well as skipping technical details of the formalization.

Thus, applying the Featherweight OCL approach to writing an updated Annex A that provides a formal semantics of the most fundamental concepts of OCL would ensure 1. that all formal context is syntactically correct and well-typed, and 2. all formal definitions and the derived logical rules are semantically consistent.

6. LESSONS LEARNED

While our paper and pencil arguments, given in [10], turned out to be essentially correct, there had also been a lesson to be learned: If the logic is not defined as a Kleene-Logic, having a structure similar to a complete partial order (CPO), reasoning becomes complicated: several important algebraic laws break down which makes reasoning in OCL inherent messy and a semantically clean compilation of OCL formulae to a two-valued presentation, that is amenable to animators like KodKod [20] or SMT-solvers like Z3 [13] completely impractical. Concretely, if the expression `not(null)` is defined `invalid` (as is the case in the present standard [18]), than standard involution does not hold, i. e., `not(not(A)) = A` does not hold universally. Similarly, if `null and null` is `invalid`, then not even idempotence `X and X = X` holds. We strongly argue in favor of a lattice-like organization, where `null` represents "more information" than `invalid` and the logical operators are monotone with respect to this semantical "information ordering." However, we do not see any clear advantage for the "more information" interpretation for strict operations (see discussion 4.3) and prefer therefore to remain conform to the requirements of the standard.

Featherweight OCL makes these two deviations from the standard, builds all logical operators on Kleene-`not` and Kleene-`and`, and shows that the entire construction of our paper "Extending OCL with Null-References" [10] is then correct, and the DNF-normalization as well as δ-closure laws (necessary for a transition into a two-valued presentation of OCL specifications ready for interpretation in SMT solvers (see [11] for details) are valid in Featherweight OCL.

7. CONCLUSION AND FUTURE WORK

Featherweight OCL concentrates on formalizing the semantics of a core subset of OCL in general and in particular on formalizing the consequences of a four-valued logic (i. e., OCL versions that support, besides the truth values `true` and `false` also the two exception values `invalid` and `null`).

In the following, we outline the necessary steps for turning Featherweight OCL into a fully fledged tool for OCL, e. g., similar to HOL-OCL as well as for supporting test case generation similar to HOL-TestGen [9]. There are essentially five extensions necessary:

- extension of the library to support all OCL data types, e. g., `Sequence(T)`, `OrderedSet(T)`. This formalization of the OCL standard library can be used for checking the consistency of the formal semantics (known as "Annex A") with the informal and semi-formal requirements in the normative part of the OCL standard.
- development of a compiler that compiles a textual or CASE tool representation (e. g., using XMI or the textual syntax of the USE tool [19]) of class models. Such compiler could also generate the necessary casts when converting standard OCL to Featherweight OCL as well

as providing "normalizations" such as converting multiplicities of class attributes to into OCL class invariants.
- a setup for translating Featherweight OCL into a two-valued representation as described in [11]. As, in real-world scenarios, large parts of UML/OCL specifications are defined (e. g., from the default multiplicity `1` of an attributes `x`, we can directly infer that for all valid states `x` is neither `invalid` nor `null`), such a translation enables an efficient test case generation approach.
- a setup in Featherweight OCL of the Nitpick animator [3]. It remains to be shown that the standard, Kodkod [20] based animator in Isabelle can give a similar quality of animation as the OCLexec Tool [14]
- a code-generator setup for Featherweight OCL for Isabelle's code generator. For example, the Isabelle code generator supports the generation of F#, which would allow to use OCL specifications for testing arbitrary .net-based applications.

The first two extensions are sufficient to provide a formal proof environment for OCL 2.3 similar to HOL-OCL while the remaining extensions are geared towards increasing the degree of proof automation and usability as well as providing a tool-supported test methodology for UML/OCL.

Our work shows that developing a machine-checked formal semantics of recent OCL standards still reveals significant inconsistencies—even though this type of research is not new. In fact, we started our work already with the 1.x series of OCL. The reasons for this ongoing consistency problems of OCL standard are manifold. For example, the consequences of adding an additional exception value to OCL 2.2 are widespread across the whole language and many of them are also quite subtle. Here, a machine-checked formal semantics is of great value, as one is forced to formalize all details and subtleties.

Moreover, the standardization process of the OMG, in which standards (e. g., the UML infrastructure and the OCL standard) that need to be aligned closely are developed quite independently, are prone to ad-hoc changes that attempt to align these standards. And, even worse, updating a standard document by voting on the acceptance (or rejection) of isolated text changes does not help either. Here, a tool for the editor of the standard that helps to check the consistency of the whole standard after each and every modifications can be of great value as well.

Acknowledgments

We would like to thank Edward Willink for suggesting the alternative interpretation of `null` as "don't know."

References

[1] P. B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Kluwer Academic Publishers, 2002.

[2] C. Barrett and C. Tinelli. Cvc3. In W. Damm and H. Hermanns, editors, *CAV*, LNCS 4590, pages 298–302. Springer, 2007. doi: 10.1007/978-3-540-73368-3_34.

[3] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. C. Paulson, editors, *ITP*, LNCS 6172 , pages 131–146. Springer, 2010. doi: 10.1007/978-3-642-14052-5_11.

[4] A. D. Brucker. *An Interactive Proof Environment for Object-oriented Specifications*. PhD thesis, ETH Zurich, Mar. 2007. ETH Dissertation No. 17097.

[5] A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.

[6] A. D. Brucker and B. Wolff. An extensible encoding of object-oriented data models in HOL. *Journal of Automated Reasoning*, 41:219–249, 2008. doi: 10.1007/s10817-008-9108-3.

[7] 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)*, LNCS 4961 , pages 97–100. Springer, 2008. doi: 10.1007/978-3-540-78743-3_8.

[8] A. D. Brucker and B. Wolff. Semantics, calculi, and analysis for object-oriented specifications. *Acta Informatica*, 46(4): 255–284, July 2009. doi: 10.1007/s00236-009-0093-8.

[9] A. D. Brucker and B. Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 2012. doi: 10.1007/s00165-012-0222-y.

[10] A. D. Brucker, M. P. Krieger, and B. Wolff. Extending OCL with null-references. In S. Gosh, editor, *Models in Software Engineering*, LNCS 6002 , pages 261–275. Springer, 2009. doi: 10.1007/978-3-642-12261-3_25.

[11] A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff. A specification-based test case generation method for UML/OCL. In J. Dingel and A. Solberg, editors, *MoDELS Workshops*, LNCS 6627, pages 334–348. Springer, 2010. doi: 10.1007/978-3-642-21210-9_33.

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

[13] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, LNCS 4963, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24.

[14] M. P. Krieger, A. Knapp, and B. Wolff. Generative programming and component engineering. In E. Visser and J. Järvi, editors, *GPCE*, pages 53–62. ACM, Oct. 2010.

[15] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, LNCS 2283. Springer, 2002. doi: 10.1007/3-540-45949-9.

[16] Object Management Group. Object constraint language specification (version 1.1), Sept. 1997. Available as OMG document ad/97-08-08.

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

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

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

[20] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS*, LNCS 4424, pages 632–647. Springer, 2007. doi: 10.1007/978-3-540-71209-1_49.

[21] M. Wenzel and B. Wolff. Building formal method tools in the Isabelle/Isar framework. In K. Schneider and J. Brandt, editors, *TPHOLS 2007*, LNCS 4732, pages 352–367. Springer, 2007. doi: 10.1007/978-3-540-74591-4_26.