Extended Version

Featherweight OCL

A Study for a Consistent Semantics of UML/OCL 2.3 in HOL

Achim D. Brucker Burkhart Wolff

February 22, 2013

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 in a uniform manner.

Further readings: This theory extends the paper "Featherweight OCL: A study for the consistent semantics of OCL 2.3 in HOL" [10] that is published as part of the proceedings of the OCL workshop 2012.

Contents

I. Introduction 9

1. Motivation 11

2. Background 13

2.1. Formal Foundation . . . 13
2.1.1. Isabelle . . . 13
2.1.2. Higher-order logic . . . 14
2.1.3. Higher-order Logic and Isabelle . . . 16
2.1.4. Specification Constructs in Isabelle/HOL . . . 17
2.2. Featherweight OCL: Design Goals . . . 17
2.3. The Theory Organization . . . 18
2.3.1. Denotational Semantics . . . 18
2.3.2. Logical Layer . . . 20
2.3.3. Algebraic Layer . . . 21
2.4. A Machine-checked Annex A . . . 24

II. A Formal Semantics of OCL 2.3 in Isabelle/HOL 27

2.5. Formal and Technical Background . . . 29
2.5.1. Validity and Evaluations . . . 29
2.5.2. Strict Operations . . . 29
2.5.3. Boolean Operators . . . 30
2.5.4. Object-oriented Data Structures . . . 31
2.5.5. The Accessors . . . 31
2.6. A Proposal for an OCL 2.1 Semantics . . . 32
2.6.1. Revised Operations on Primitive Types . . . 32
2.6.2. Null in Class Types . . . 34
2.6.3. Revised Accessors . . . 35
2.6.4. Other Operations on States . . . 36
2.7. Attribute Values . . . 37
2.7.1. Single-Valued Attributes . . . 37
2.7.2. Collection-Valued Attributes . . . 37
2.7.3. The Precise Meaning of Multiplicity Constraints . . . 38

3. Part I: Core Definitions and Library 39
3.0.4. Notations for the option type 39
3.0.5. Minimal Notions of State and State Transitions 39
3.0.6. Prerequisite: An Abstract Interface for OCL Types 40
3.0.7. Accomodation of Basic Types to the Abstract Interface 40
3.1. The Semantic Space of OCL Types: Valuations. 41
3.2. Boolean Type and Logic 42
3.2.1. Basic Constants 42
3.2.2. Fundamental Predicates I: Validity and Definedness 43
3.2.3. Fundamental Predicates II: Logical (Strong) Equality 45
3.2.4. Fundamental Predicates III 46
3.2.5. Logical Connectives and their Universal Properties 46
3.3. A Standard Logical Calculus for OCL 50
3.3.1. Global vs. Local Judgements 50
3.3.2. Local Validity and Meta-logic 51
3.3.3. Local Judgements and Strong Equality 53
3.3.4. Laws to Establish Definedness (Delta-Closure) 54
3.4. Miscellaneous: OCL's if then else endif 55
3.5. Basic Types like Void, Boolean and Integer 56
3.5.1. Strict equalities on Basic Types. 56
3.5.2. Logic and algebraic layer on Basic Types. 56
3.5.3. Test Statements on Basic Types. 59
3.5.4. More algebraic and logical layer on basic types 60
3.6. Example for Complex Types: The Set-Collection Type 61
3.6.1. The construction of the Set-Collection Type 61
3.6.2. Constants on Sets 62
3.6.3. Strict Equality on Sets 63
3.6.4. Algebraic Properties on Strict Equality on Sets 64
3.6.5. Library Operations on Sets 64
3.6.6. Logic and Algebraic Layer on Set Operations 66
3.6.7. Test Statements 73

4. Part II: State Operations and Objects 75
4.0.8. Recall: The generic structure of States 75
4.0.9. Referential Object Equality in States 75
4.0.10. Further requirements on States 76
4.1. Miscellaneous: Initial States (for Testing and Code Generation) 77
4.1.1. Generic Operations on States 77

III. Conclusion 81

5. Conclusion 83
5.1. Lessons Learned 83

5.2. Conclusion and Future Work 83

Part I.

Introduction

1. Motivation

At its origins [14, 17], 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 [15, 16] added a second exception element, which 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 [16].

In the paper "Extending OCL with Null-References" [4] 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 short paper is based on a formalization of [4], called "Featherweight OCL," in Isabelle/HOL [13]. 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.

2. Background

2.1. Formal Foundation

2.1.1. Isabelle

Isabelle [13] is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and natural deduction inference rules. Among other logics, Isabelle supports first-order logic, Zermelo-Fraenkel set theory and the instance for Church's higher-order logic HOL, which we choose as basis for HOL-TestGen and which is introduced in the subsequent section.

Isabelle's inference rules are based on the built-in meta-level implication $\_\Longrightarrow\_$ allowing to form constructs like $A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow A_{n+1}$, which are viewed as a *rule* of the form "from assumptions A_1 to A_n, infer conclusion A_{n+1}" and which is written in Isabelle as

$$[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1} \qquad \text{or, in mathematical notation,} \qquad \frac{A_1 \qquad \cdots \qquad A_n}{A_{n+1}} \; . \tag{2.1}$$

The built-in meta-level quantification $\bigwedge x.\ x$ captures the usual side-constraints "x must not occur free in the assumptions" for quantifier rules; meta-quantified variables can be considered as "fresh" free variables. Meta-level quantification leads to a generalization of Horn-clauses of the form:

$$\bigwedge x_1, \ldots, x_m.\ [\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1} \, . \tag{2.2}$$

Isabelle supports forward- and backward reasoning on rules. For backward-reasoning, a *proof-state* can be initialized and further transformed into others. For example, a proof of ϕ, using the Isar [**?**] language, will look as follows in Isabelle:

```
lemma label:   φ
  apply(case_tac)
  apply(simp_all)
done
```
(2.3)

This proof script instructs Isabelle to prove ϕ by case distinction followed by a simplification of the resulting proof state. Such a proof state is an implicitly conjoint sequence of generalized Horn-clauses (called *subgoals*) $\phi_1, \ldots, \phi_n$ and a *goal* ϕ. Proof states were usually denoted by:

$$\begin{array}{rl} \text{label}: & \phi \\ 1. & \phi_1 \\ & \vdots \\ n. & \phi_n \end{array} \tag{2.4}$$

Subgoals and goals may be extracted from the proof state into theorems of the form $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$ at any time; this mechanism helps to generate test theorems. Further, Isabelle supports meta-variables (written $?x, ?y, \ldots$), which can be seen as "holes in a term" that can still be substituted. Meta-variables are instantiated by Isabelle's built-in higher-order unification.

2.1.2. Higher-order logic

Higher-order logic (HOL) [? ?] is a classical logic based on a simple type system. It provides the usual logical connectives like $\_ \wedge \_$, $\_ \to \_$, $\neg \_$ as well as the object-logical quantifiers $\forall x.\ P\ x$ and $\exists x.\ P\ x$; in contrast to first-order logic, quantifiers my range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centered around extensional equality $\_ = \_ :: \alpha \Rightarrow \alpha \Rightarrow \text{bool}$. HOL is more expressive than first-order logic, since, e. g., induction schemes can be expressed inside the logic. Being based on some polymorphically typed λ-calculus, HOL can be viewed as a combination of a programming language like SML or Haskell and a specification language providing powerful logical quantifiers ranging over elementary and function types.

Isabelle/HOL is a logical embedding of HOL into Isabelle. The (original) simple-type system underlying HOL has been extended by Hindley/Milner style polymorphism with type-classes similar to Haskell. While Isabelle/HOL is usually seen as proof assistant, we use it as symbolic computation environment. Implementations on top of Isabelle/HOL can re-use existing powerful deduction mechanisms such as higher-order resolution, tableaux-based reasoners, rewriting procedures, Presburger Arithmetic, and via various integration mechanisms, also external provers such as Vampire and the SMT-solver Z3.

Isabelle/HOL offers support for a particular methodology to extend given theories in a logically safe way: A theory-extension is *conservative* if the extended theory is consistent provided that the original theory was consistent. Conservative extensions can be *constant definitions*, *type definitions*, *datatype definitions*, *primitive recursive definitions* and *wellfounded recursive definitions*.

For example, typed sets were built in the Isabelle libraries conservatively on top of the kernel of HOL as functions to bool; consequently, the constant definitions for membership is as follows:[1]

$$
\begin{array}{llll}
\textsf{types} & \alpha\,\text{set} & = \alpha \Rightarrow \text{bool} & \\
\textsf{definition} & \text{Collect} & ::(\alpha \Rightarrow \text{bool}) \Rightarrow \alpha\,\text{set} & \text{— set comprehension} \\
\textsf{where} & \text{Collect } S & \equiv S & \\
\textsf{definition} & \text{member} & ::\alpha \Rightarrow \alpha \Rightarrow \text{bool} & \text{— membership test} \\
\textsf{where} & \text{member } s\,S & \equiv Ss &
\end{array}
\tag{2.5}
$$

Isabelle's powerful syntax engine is instructed to accept the notation $\{x \mid P\}$ for Collect $\lambda\, x.\ P$ and the notation $s \in S$ for member $s\ S$. As can be inferred from the example, constant definitions are axioms that introduce a fresh constant symbol by some closed, non-recursive expressions; this type of axiom is logically safe since it works like an abbreviation. The syntactic side-conditions of this axiom are mechanically checked, of course. It is straight-forward to express the usual operations on sets like $\_ \cup \_, \_ \cap \_ :: \alpha\ \text{set} \Rightarrow \alpha\ \text{set} \Rightarrow \alpha\ \text{set}$ as conservative extensions, too, while the rules of typed set-theory were derived by proofs from these definitions.

Similarly, a logical compiler is invoked for the following statements introducing the types option and list:

$$
\begin{array}{lll}
\textsf{datatype} & \text{option} & = \text{None} \mid \text{Some}\,\alpha \\
\textsf{datatype} & \alpha\,\text{list} & = \text{Nil} \mid \text{Cons}\ a\ l
\end{array}
\tag{2.6}
$$

Here, $[]$ or $a\#l$ are an alternative syntax for Nil or Cons $a\ l$; moreover, $[a, b, c]$ is defined as alternative syntax for $a\#b\#c\#[]$. These (recursive) statements were internally represented in by internal type- and constant definitions. Besides the *constructors* None, Some, $[]$ and Cons, there is the match-operation case x of None$\Rightarrow F$ $\mid$ Some $a\Rightarrow G\ a$ respectively case x of $[]\Rightarrow F$ $\mid$ Cons $a\,r\Rightarrow G\,a\,r$. From the internal definitions (not shown here) a number and properties were automatically derived. We show only the case for lists:

$$
\begin{array}{ll}
(\text{case } []\ \text{ of } [] \Rightarrow F \mid (a\#r) \Rightarrow G\ a\ r) = F & \\
(\text{case } b\#t \text{ of } [] \Rightarrow F \mid (a\#r) \Rightarrow G\ a\ r) = G\ b\ t & \\
[] \neq a\#t & \text{– distinctness} \\
[\![a = [] \to P; \exists\ x\ t.\ a = x\#t \to P]\!] \Longrightarrow P & \text{– exhaust} \\
[\![P[]; \forall\ at.\ Pt \to P(a\#t)]\!] \Longrightarrow Px & \text{– induct}
\end{array}
\tag{2.7}
$$

Finally, there is a compiler for primitive and well-founded recursive function definitions. For example, we may define the sort-operation of our running test example by:

$$
\begin{array}{lll}
\textsf{fun} & \text{ins} & ::[\alpha :: \text{linorder}, \alpha\ \text{list}] \Rightarrow \alpha\ \text{list} \\
\textsf{where} & \text{ins } x\ [\,] & = [x] \\
 & \text{ins } x\ (y\#ys) & = \text{if } x < y \text{ then } x\#y\#ys \text{ else } y\#(\text{ins } x\ ys)
\end{array}
\tag{2.8}
$$

$$
\begin{array}{lll}
\textsf{fun} & \text{sort} & ::(\alpha :: \text{linorder})\ \text{list} \Rightarrow \alpha\ \text{list} \\
\textsf{where} & \text{sort } [\,] & = [\,] \\
 & \text{sort}(x\#xs) & = \text{ins } x\ (\text{sort } xs)
\end{array}
\tag{2.9}
$$

[1] To increase readability, we use a slightly simplified presentation.

The internal (non-recursive) constant definition for these operations is quite involved; however, the logical compiler will finally derive all the equations in the statements above from this definition and make them available for automated simplification.

Thus, Isabelle/HOL also provides a large collection of theories like sets, lists, multisets, orderings, and various arithmetic theories which only contain rules derived from conservative definitions. In particular, Isabelle manages a set of *executable types and operators*, i. e., types and operators for which a compilation to SML, OCaml or Haskell is possible. Setups for arithmetic types such as int have been done; moreover any datatype and any recursive function were included in this executable set (providing that they only consist of executable operators). Similarly, Isabelle manages a large set of (higher-order) rewrite rules into which recursive function definitions were included. Provided that this rule-set represents a terminating and confluent rewrite-system, the Isabelle simplifier provides also a highly potent decision procedure for many fragments of theories underlying the constraints to be processed when constructing test-theorems.

2.1.3. Higher-order Logic and Isabelle

Higher-order Logic (HOL) [**? ?**] 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 $\alpha :: \mathit{order}$ 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.

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 $\tau_\perp := \perp \mid \llcorner\_\lrcorner : \alpha$ 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 $(\mathrm{dom}\ \_)$ and range $(\mathrm{ran}\ \_)$. The library is built entirely by logically safe, conservative definitions and derived rules. This methodology is also applied to HOL-OCL [6] and Featherweight OCL.

2.1.4. Specification Constructs in Isabelle/HOL

2.2. Featherweight OCL: Design Goals

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, 6], Featherweight OCL is based on the following principles:

1. It is an embedding into a powerful semantic meta-language and environment, namely Isabelle/HOL [13].
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, and there is a `null`-element in the type `Set(A)`.
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 [2]. 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 [7] the universe construction also gives semantics to type casts, dynamic type tests, as well as functions such as `oclAllInstances()`, or `isNewInState()`.
6. Featherweight OCL types may be arbitrarily nested: `Set{Set{1,2}} = Set{Set{2,1}}` is legal and true.
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 [9]), which is nasty but can be hidden from the user inside tools.

2.3. 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.

2.3.1. 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 [4]) 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 $\equiv \lfloor \perp \rfloor$. Now, any OCL type can be represented by an HOL type of the form:

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

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

$$I[\![\texttt{invalid} :: V(\alpha)]\!]\tau \equiv \mathrm{bot} \qquad I[\![\texttt{null} :: V(\alpha)]\!]\tau \equiv \mathrm{null}$$
$$I[\![\texttt{true} :: \texttt{Boolean}]\!]\tau = \lfloor\lfloor \mathrm{true} \rfloor\rfloor \qquad I[\![\texttt{false}]\!]\tau = \lfloor\lfloor \mathrm{false} \rfloor\rfloor$$

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

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

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 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:

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

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

2.3.2. Logical Layer

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

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

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:

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

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:

$$\tau \models \mathtt{true} \quad \neg(\tau \models \mathtt{false}) \quad \neg(\tau \models \mathtt{invalid}) \quad \neg(\tau \models \mathtt{null})$$
$$\tau \models \mathtt{not}\ P \Longrightarrow \tau\neg \models P$$
$$\tau \models P\ \mathtt{and}\ Q \Longrightarrow \tau \models P \qquad \tau \models P\ \mathtt{and}\ Q \Longrightarrow \tau \models Q$$
$$\tau \models P \Longrightarrow (\mathtt{if}\ P\ \mathtt{then}\ B_1\ \mathtt{else}\ B_2\ \mathtt{endif})\tau = B_1\tau$$
$$\tau \models \mathtt{not}\ P \Longrightarrow (\mathtt{if}\ P\ \mathtt{then}\ B_1\ \mathtt{else}\ B_2\ \mathtt{endif})\tau = B_2\tau$$
$$\tau \models P \Longrightarrow \tau \models \delta P \qquad \tau \models (\delta X) \Longrightarrow \tau \models \upsilon X$$

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:

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

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!** [**?**] or Z3 [11]. Delta-closure rules for all logical connectives have the following format, e. g.:

$$\tau \models \delta x \Longrightarrow (\tau \models \texttt{not}\, x) = (\neg(\tau \models x))$$
$$\tau \models \delta x \Longrightarrow \tau \models \delta y \Longrightarrow (\tau \models x\, \texttt{and}\, y) = (\tau \models x \wedge \tau \models y)$$
$$\begin{aligned}&\tau \models \delta x \Longrightarrow \tau \models \delta y\\ &\quad\Longrightarrow (\tau \models (x\, \texttt{implies}\, y)) = ((\tau \models x) \longrightarrow (\tau \models y))\end{aligned}$$

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 $\tau \models x\texttt{>}0\ \texttt{or}3\texttt{*}y\texttt{>}x\texttt{*}x$ 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.

2.3.3. 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 re-formulated 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, 6], to Featherweight OCL as presented here. The standard expresses at many places that most operations are strict, i. e., enjoy the properties (exemplary for `_ + _`):

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

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:

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

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.[2] This semantic alternative (this is *not* Featherweight OCL at present) would yield:

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

Algebraic rules are also the key for execution and compilation of Featherweight OCL

[2] 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`.

expressions. We derived, e. g.:

$$\delta\ \texttt{Set\{\}} = \texttt{true}$$

$$\delta\ (X\texttt{->including(}x\texttt{)}) = \delta X \texttt{ and } \delta x$$

$$\texttt{Set\{\}->includes(}x\texttt{)} = (\texttt{if } \upsilon\ x \texttt{ then false else invalid endif})$$

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

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

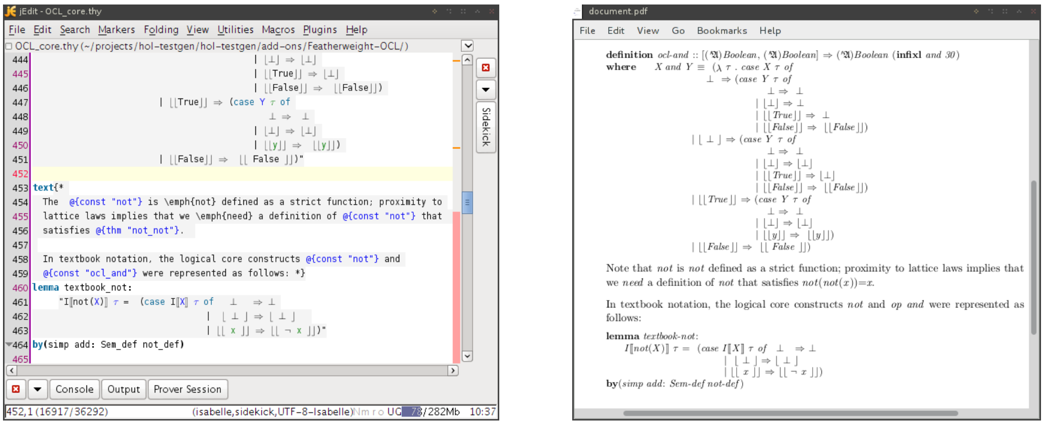
which have been machine-checked and which present a high-level and in our opinion fairly readable information for OCL tool manufactures and users.

2.4. A Machine-checked Annex A

Isabelle, as a framework for building formal tools [?], 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 2.1 illustrates this approach: 2.1a shows the jEdit-based development environment of Isabelle with an excerpt of one of the core theories of Featherweight OCL. 2.1b



(a) The Isabelle jEdit environment. (b) The generated formal document.

Figure 2.1.: Generating documents with guaranteed syntactical and semantical consistency.

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.

Part II.

A Formal Semantics of OCL 2.3 in Isabelle/HOL

2.5. Formal and Technical Background

2.5.1. Validity and Evaluations

The topmost goal of the formal semantics is to define the *validity statement*:

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

where σ is the pre-state and σ' the post-state of the underlying system and P is a Boolean expression (a *formula*). The assertion language of P is composed of 1) operators on built-in data structures such as Boolean 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. Informally, a formula P is valid if and only if its evaluation in the context (σ, σ') yields true. As all types in HOL-OCL are extended by the special element $\bot$ denoting undefinedness, we define formally:

$$(\sigma, \sigma') \vDash P \equiv \big(P(\sigma, \sigma') = \llcorner \mathrm{true} \lrcorner\big)\,.$$

Since all operators of the assertion language depend on the context (σ, σ') and result in values that can be $\bot$, all expressions can be viewed as *evaluations* from (σ, σ') to a type $\tau_\bot$. All types of expressions are of a form captured by

$$V(\alpha) := \mathrm{state} \times \mathrm{state} \Rightarrow \alpha_\bot\,,$$

where state stands for the system state and state $\times$ state describes the pair of pre-state and post-state and $\_ := \_$ denotes the type abbreviation.

The OCL semantics [**?** , Annex A] uses different interpretation functions for invariants and pre-conditions; we achieve their semantic effect by a syntactic transformation $\_{}_{\mathrm{pre}}$ which replaces all accessor functions $\_\,.\,\mathrm{a}$ by their counterparts $\_\,.\,\mathrm{a}$ `@pre`. For example, $(\mathit{self}\,.\,\mathrm{a} > 5)_{\mathrm{pre}}$ is just $(\mathit{self}\,.\,\mathrm{a}\ \texttt{@pre} > 5)$.

2.5.2. Strict Operations

An operation is called strict if it returns $\bot$ if one of its arguments is $\bot$. Most OCL operations are strict, e. g., the Boolean negation is formally presented as:

$$I[\![\texttt{not}\ X]\!]\tau \equiv \begin{cases} \llcorner \neg \ulcorner I[\![X]\!]\tau \urcorner \lrcorner & \text{if } I[\![X]\!]\tau \neq \bot, \\ \bot & \text{otherwise}\,. \end{cases}$$

where $\tau = (\sigma, \sigma')$ and $I[\![\_]\!]$ is a notation marking the HOL-OCL constructs to be defined. This notation is motivated by the definitions in the OCL standard [**?**]. In our case, $I[\![\_]\!]$ is just the identity, i. e., $I[\![X]\!] \equiv X$. These constructs, i. e., `not` _ are HOL functions (in this case of HOL type $V(\mathrm{bool}) \Rightarrow V(\mathrm{bool})$) that can be viewed as *transformers on evaluations*.

The binary case of the integer addition is analogous:

$$I[\![X + Y]\!]\,\tau \equiv \begin{cases} \llcorner \ulcorner I[\![X]\!]\,\tau \urcorner + \ulcorner I[\![Y]\!]\,\tau \urcorner \lrcorner & \text{if } I[\![X]\!]\,\tau \neq \bot \text{ and } I[\![Y]\!]\,\tau \neq \bot, \\ \bot & \text{otherwise}\,. \end{cases}$$

Here, the operator $\_ + \_$ on the right refers to the integer HOL operation with type $[\text{int}, \text{int}] \Rightarrow \text{int}$. The type of the corresponding strict HOL-OCL operator $\_+\_$ is $[V(\text{int}), V(\text{int})] \Rightarrow V(\text{int})$. A slight variation of this definition scheme is used for the operators on collection types such as HOL-OCL sets or sequences:

$$I[\![X\texttt{->union}(Y)]\!]\tau \equiv \begin{cases} S_{\llcorner}\ulcorner I[\![X]\!]\tau\urcorner \cup \ulcorner I[\![Y]\!]\tau\urcorner{}_{\lrcorner} & \text{if } I[\![X]\!]\tau \neq \bot \text{ and } I[\![Y]\!]\tau \neq \bot, \\ \bot & \text{otherwise.} \end{cases}$$

Here, S ("smash") is a function that maps a lifted set $\llcorner X \lrcorner$ to $\bot$ if and only if $\bot \in X$ and to the identity otherwise. Smashedness of collection types is the natural extension of the strictness principle for data structures.

Intuitively, the type expression $V(\tau)$ is a representation of the type that corresponds to the HOL-OCL type τ. We introduce the following type abbreviations:

$$\texttt{Boolean} := V(\text{bool}), \qquad \alpha\,\texttt{Set} := V(\alpha\,\text{set}),$$
$$\texttt{Integer} := V(\text{int}), \text{ and} \qquad \alpha\,\texttt{Sequence} := V(\alpha\,\text{list}).$$

The mapping of an expression `E` of HOL-OCL type `T` to a HOL expression E of HOL type T is injective and preserves well-typedness.

2.5.3. Boolean Operators

There is a small number of explicitly stated exceptions from the general rule that HOL-OCL operators are strict: the strong equality, the definedness operator and the logical connectives. As a prerequisite, we define the logical constants for truth, absurdity and undefinedness. We write these definitions as follows:

$$I[\![\texttt{true}]\!]\tau \equiv \llcorner\text{true}\lrcorner, \qquad I[\![\texttt{false}]\!]\tau \equiv \llcorner\text{false}\lrcorner, \text{ and} \qquad I[\![\texttt{invalid}]\!]\tau \equiv \bot.$$

HOL-OCL has a *strict equality* $\_ \doteq \_$. On the primitive types, it is defined similarly to the integer addition; the case for objects is discussed later. For logical purposes, we introduce also a *strong equality* $\_ \triangleq \_$ which is defined as follows:

$$I[\![X \triangleq Y]\!]\,\tau \equiv (I[\![X]\!]\,\tau = I[\![Y]\!]\,\tau),$$

where the $\_ = \_$ operator on the right denotes the logical equality of HOL. The undefinedness test is defined by X `.oclIsInvalid()` $\equiv (X \triangleq \texttt{invalid})$. The strong equality can be used to state reduction rules like: $\tau \vDash (\texttt{invalid} \doteq X) \triangleq \texttt{invalid}$. The OCL standard requires a Strong Kleene Logic. In particular:

$$I[\![X \texttt{ and } Y]\!]\tau \equiv \begin{cases} \llcorner\ulcorner x\urcorner \wedge \ulcorner y\urcorner\lrcorner & \text{if } x \neq \bot \text{ and } y \neq \bot, \\ \llcorner\text{false}\lrcorner & \text{if } x = \llcorner\text{false}\lrcorner \text{ or } y = \llcorner\text{false}\lrcorner, \\ \bot & \text{otherwise.} \end{cases}$$

where $x = I[\![X]\!]\tau$ and $y = I[\![Y]\!]\tau$. The other Boolean connectives were just shortcuts: $X \texttt{ or } Y \equiv \texttt{not}\,(\texttt{not}\ X \texttt{ and } \texttt{not}\ Y)$ and $X \texttt{ implies } Y \equiv \texttt{not}\ X \texttt{ or } Y$.

2.5.4. Object-oriented Data Structures

Now we turn to several families of operations that the user implicitly defines when stating a class model as logical context of a specification. This is the part of the language where object-oriented features such as type casts, accessor functions, and tests for dynamic types come into play. Syntactically, a class model provides a collection of classes C, an inheritance relation $\_ < \_$ on classes and a collection of attributes A associated to classes. Semantically, a class model means a collection of accessor functions (denoted $\_.\mathrm{a} :: A \to B$ and $\_.\mathrm{a}$ `@pre` $:: A \to B$ for $\mathrm{a} \in A$ and $A, B \in C$), type casts that can change the static type of an object of a class (denoted $\_{}_{[\mathrm{C}]}$ of type $A \to C$) and dynamic type tests (denoted $\mathrm{isType}_{\mathrm{C}}\ \_$). A precise formal definition can be found in [9].

Class models: A simplified semantics.

In this section, we will have to clarify the notions of *object identifiers*, *object representations*, *class types* and *state*. We will give a formal model for this, that will satisfy all properties discussed in the subsequent section except one (see [7] for the complete model).

First, object identifiers are captured by an abstract type oid comprising countably many elements and a special element `nullid`. Second, 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). Third, the class type C will be the type of such an object representation: $C := (\mathrm{oid} \times \mathrm{C_t} \times A_1 \times \cdots \times A_k)$ where a unique tag-type $\mathrm{C_t}$ (ensuring type-safety) is created for each class type, where the types $A_1, \ldots, A_k$ 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. Fourth, for a class model M with the classes $C_1, \ldots, C_n$, we define states as partial functions from oid's to object representations satisfying a *state invariant* inv_σ:

$$\mathrm{state} := \{f :: \mathrm{oid} \rightharpoonup (C_1 + \ldots + C_n) \mid \mathrm{inv}_\sigma(f)\}$$

where $\mathrm{inv}_\sigma(f)$ states two conditions: 1) there is no object representation for `nullid`: `nullid` $\notin (\mathrm{dom}\ f)$. 2) there is a"one-to-one" correspondence between object representations and oid's: $\forall oid \in \mathrm{dom}\ f.\ oid = \mathrm{OidOf}\ \ulcorner f(oid) \urcorner$. The latter condition is also mentioned in [**?** , Annex A] and goes back to Mark Richters [17].

2.5.5. The Accessors

On states built over object universes, we can now define accessors, casts, and type tests of an object model. We consider the case of an attribute a of class C which has the simple class type D (not a primitive type, not a collection):

$$I[\![\mathit{self}\,.\,\mathrm{a}]\!](\sigma,\sigma') \equiv \begin{cases} \bot & \text{if } O = \bot \vee \mathrm{OidOf}\ulcorner O\urcorner \notin \mathrm{dom}\ \sigma' \\ \mathrm{get_D}\ u & \text{if } \sigma'(\mathrm{get_C}\ulcorner\sigma'(\mathrm{OidOf}\ulcorner O\urcorner)\urcorner.\,\mathrm{a}^{(0)}) = \llcorner u \lrcorner, \\ \bot & \text{otherwise.} \end{cases}$$

$$I[\![\mathit{self}\,.\,\mathrm{a@pre}]\!](\sigma,\sigma') \equiv \begin{cases} \bot & \text{if } O = \bot \vee \mathrm{OidOf}\ulcorner O\urcorner \notin \mathrm{dom}\ \sigma \\ \mathrm{get_D}\ u & \text{if } \sigma(\mathrm{get_C}\ulcorner\sigma(\mathrm{OidOf}\ulcorner O\urcorner)\urcorner.\,\mathrm{a}) = \llcorner u \lrcorner, \\ \bot & \text{otherwise.} \end{cases}$$

where $O = I[\![\mathit{self}]\!](\sigma,\sigma')$. Here, $\mathrm{get_D}$ is the projection function from the object universe to $D_\bot$, and $x.\,\mathrm{a}$ is the projection of the attribute from the class type (the Cartesian product). For simple class types, we have to evaluate expression *self*, get an object representation (or undefined), project the attribute, de-reference it in the pre or post state and project the class object from the object universe ($\mathrm{get_D}$ may yield $\bot$ if the element in the universe does not correspond to a D object representation.) In the case for a primitive type attribute, the de-referentiation step is left out, and in the case of a collection over class types, the elements of the collection have to be point-wise de-referenced and smashed.

In our model accessors always yield (type-safe) object representations; not oid's. Thus, a dangling reference, i. e., one that is *not* in dom σ, results in `invalid` (this is a subtle difference to [? , Annex A] where the undefinedness is detected one de-referentiation step later). The strict equality $\_ \doteq \_$ must be defined via OidOf when applied to objects. It satisfies $(\texttt{invalid} \doteq X) \triangleq \texttt{invalid}$.

The definitions of casts and type tests can be found in [7], together with other details of the construction above and its automation in HOL-OCL.

2.6. A Proposal for an OCL 2.1 Semantics

In this section, we describe our OCL 2.1 semantics proposal as an increment to the OCL 2.0 semantics (underlying HOL-OCL and essentially formalizing [? , Annex A]). In later versions of the standard [15] the formal semantics appendix reappears although being incompatible with the normative parts of the standard. Not all rules shown here are formally proven; technically, these are informal proofs "with a glance" on the formal proofs shown in the previous section.

2.6.1. Revised Operations on Primitive Types

In UML, and since [15] in OCL, all primitive types comprise the `null`-element, modeling the possibility to be non-existent. From a functional language perspective, this corresponds to the view that each basic value is a type like `int option` as in SML. Technically, this results in lifting any primitive type twice:

$$\texttt{Integer} := V(\mathrm{int}_{\perp}) \text{ , etc.}$$

and basic operations have to take the null elements into account. The distinguishable undefined and null-elements were defined as follows:

$$I[\![\texttt{invalid}]\!]\tau \equiv \perp \quad \text{and} \quad I[\![\texttt{null}_{\texttt{Integer}}]\!]\tau \equiv \llcorner\perp\lrcorner\,.$$

An interpretation (consistent with [15]) is that $\texttt{null}_{\texttt{Integer}} + 3 = \texttt{invalid}$, and due to commutativity, we postulate $3 + \texttt{null}_{\texttt{Integer}} = \texttt{invalid}$, too. The necessary modification of the semantic interpretation looks as follows:

$$I[\![X + Y]\!]\,\tau \equiv \begin{cases} \llcorner\llcorner\ulcorner\ulcorner x\urcorner\urcorner + \ulcorner\ulcorner y\urcorner\urcorner\lrcorner\lrcorner & \text{if } x \neq \perp,\ y \neq \perp,\ \ulcorner x\urcorner \neq \perp \text{ and } \ulcorner y\urcorner \neq \perp \\ \perp & \text{otherwise}\,. \end{cases}$$

where $x = I[\![X]\!]\,\tau$ and $y = I[\![Y]\!]\,\tau$. The resulting principle here is that operations on the primitive types Boolean, Integer, Real, and String treat null as invalid (except $\_ \doteq \_$, `_.oclIsInvalid()`, `_.oclIsUndefined()`, casts between the different representations of `null`, and type-tests).

This principle is motivated by our intuition that invalid represents known errors, and null-arguments of operations for Boolean, Integer, Real, and String belong to this class. Thus, we must also modify the logical operators such that $\texttt{null}_{\texttt{Boolean}}$ `and false` $\triangleq$ `false` and $\texttt{null}_{\texttt{Boolean}}$ `and true` $\triangleq \perp$.

With respect to definedness reasoning, there is a price to pay. For most basic operations we have the rule:

$$\texttt{not}\ (X + Y)\ \texttt{.oclIsInvalid()} \triangleq (\texttt{not}\ X\ \texttt{.oclIsUndefined()}) \\ \texttt{and}\ (\texttt{not}\ Y\ \texttt{.oclIsUndefined()})$$

where the test x `.oclIsUndefined()` covers two cases: x `.oclIsInvalid()` and $x \doteq$ `null` (i. e., x is `invalid` or `null`). As a consequence, for the inverse case $(X+Y)$ `.oclIsInvalid()`[3] there are four possible cases for the failure instead of two in the semantics described in [**?**]: each expression can be an erroneous `null`, or report an error. However, since all built-in OCL operations yield non-null elements (e. g., we have the rule `not` $(X + Y \doteq \texttt{null}_{\texttt{Integer}})$), a pre-computation can drastically reduce the number of cases occurring in expressions except for the base case of variables (e. g., parameters of operations and *self* in invariants). For these cases, it is desirable that implicit pre-conditions were generated as default, ruling out the `null` case. A convenient place for this are the multiplicities, which can be set to `1` (i. e., `1..1`) and will be interpreted as being non-null (see discussion in section 2.7 for more details).

Besides, the case for collection types is analogous: in addition to the invalid collection, there is a $\texttt{null}_{\mathrm{Set(T)}}$ collection as well as collections that contain null values (such as $\mathrm{Set}\{\texttt{null}_{\mathrm{T}}\}$) but never `invalid`.

[3] The same holds for $(X + Y)$ `.oclIsUndefined()`.

2.6.2. Null in Class Types

It is a viable option to rule out undefinedness in object-graphs *as such*. The essential source for such undefinedness are oid's which do not occur in the state, i. e., which represent "dangling references." Ruling out undefinedness as result of object accessors would correspond to a world where an accessor is always set explicitly to `null` or to a defined object; in a programming language without explicit deletion and where constructors always initialize their arguments (e. g., Spec# [**?**]), this may suffice. Semantically, this can be modeled by strengthening the state invariant inv_σ by adding clauses that state that in each object representation all oid's are either `nullid` or element of the domain of the state.

We deliberately decided against this option for the following reasons:

1. *methodologically* we do not like to constrain the semantics of OCL without clear reason; in particular, "dangling references" exist in C and C++ programs and it might be necessary to write contracts for them, and
2. *semantically*, the condition "no dangling references" can only be formulated with the complete knowledge of all classes and their layout in form of object representations. This restricts the OCL semantics to a closed world model.[4]

We can model `null`-elements as object-representations with `nullid` as their oid:

1 (Representation of `null`-Elements) *Let C_i be a class type with the attributes $A_1, \ldots, A_n$. Then we define its null object representation by:*

$$I[\![\mathtt{null}_{Ci}]\!]\tau \equiv \lfloor(\mathtt{nullid}, \mathrm{arb}_t, a_1, \ldots, a_n)\rfloor$$

where the a_i are $\perp$ for primitive types and collection types, and `nullid` *for simple class types.* arb_t *is an arbitrary underspecified constant of the tag-type.*

Due to the outermost lifting, the null object representation is a defined value, and due to its special reference `nullid` and the state invariant, it is a typed value not "living" in the state. The $\mathtt{null}_\mathrm{T}$-elements are not equal, but isomorphic: Each type, has its own unique $\mathtt{null}_\mathrm{T}$-element; they can be mapped, i. e., casted, isomorphic to each other. In HOL-OCL, we can overload constants by parametrized polymorphism which allows us to drop the index in this environment.

The referential strict equality allows us to write *self* $\doteq$ `null` in OCL. Recall that $\_ \doteq \_$ is based on the projection OidOf from object-representations.

[4]In our presentation, the definition of `state` in **??** assumes a closed world. This limitation can be easily overcome by leaving "polymorphic holes" in our object representation universe, i. e., by extending the type sum in the state definition to $C_1 + \cdots + C_n + \alpha$. The details of the management of universe extensions are involved, but implemented in HOL-OCL (see [7] for details). However, these constructions exclude knowing the set of sub-oid's in advance.

2.6.3. Revised Accessors

The modification of the accessor functions is now straight-forward:

$$I[\![obj.\,\mathrm{a}]\!](\sigma,\sigma') \equiv \begin{cases} \bot & \text{if } I[\![obj]\!](\sigma,\sigma') = \bot \vee \mathrm{OidOf}\ulcorner I[\![obj]\!](\sigma,\sigma')\urcorner \notin \operatorname{dom}\sigma' \\ \mathtt{null}_{\mathrm{D}} & \text{if } \mathrm{get}_{\mathrm{C}}\ulcorner\sigma'(\mathrm{OidOf}\ulcorner I[\![obj]\!](\sigma,\sigma')\urcorner)\urcorner.\,\mathrm{a}^{(0)} = \mathtt{nullid} \\ \mathrm{get}_{\mathrm{D}}\, u & \text{if } \sigma'(\mathrm{get}_{\mathrm{C}}\ulcorner\sigma'(\mathrm{OidOf}\ulcorner I[\![obj]\!](\sigma,\sigma')\urcorner)\urcorner.\,\mathrm{a}^{(0)}) = \llcorner u \lrcorner, \\ \bot & \text{otherwise.} \end{cases}$$

The definitions for type-cast and dynamic type test—which are not explicitly shown in this paper, see [7] for details—can be generalized accordingly. In the sequel, we will discuss the resulting properties of these modified accessors.

All functions of the induced signature are strict. This means that this holds for accessors, casts and tests, too:

$$\texttt{invalid}.\,\mathrm{a} \triangleq \texttt{invalid} \qquad \texttt{invalid}_{[\mathrm{C}]} \triangleq \texttt{invalid} \qquad \mathrm{isType}_{\mathrm{C}}\,\texttt{invalid} \triangleq \texttt{invalid}$$

Casts on `null` are always valid, since they have an individual dynamic type and can be casted to any other null-element due to their isomorphism.

$$\texttt{null}_{\mathrm{A}}.\,\mathrm{a} \triangleq \texttt{invalid} \qquad \texttt{null}_{\mathrm{A}[\mathrm{B}]} \triangleq \texttt{null}_{\mathrm{B}} \qquad \mathrm{isType}_{\mathrm{A}}\,\texttt{null}_{\mathrm{A}} \triangleq \texttt{true}$$

for all attributes a and classes A, B, C where $C < B < A$. These rules are further exceptions from the standard's general rule that `null` may never be passed as first ("*self*") argument.

2.6.4. Other Operations on States

Defining `_.allInstances()` is straight-forward; the only difference is the property T `.allInstances()->excludes(null)` which is a consequence of the fact that `null`'s are values and do not "live" in the state. In our semantics which admits states with "dangling references," it is possible to define a counterpart to `_.oclIsNew()` called `_.oclIsDeleted()` which asks if an object id (represented by an object representation) is contained in the pre-state, but not the post-state.

OCL does not guarantee that an operation only modifies the path-expressions mentioned in the postcondition, i. e., it allows arbitrary relations from pre-states to post-states. This framing problem is well-known (one of the suggested solutions is [?]). We define

```
(S:Set(OclAny))->modifiedOnly():Boolean
```

where `S` is a set of object representations, encoding a set of oid's. The semantics of this operator is defined such that for any object whose oid is *not* represented in `S` and that is defined in pre and post state, the corresponding object representation will not change in the state transition:

$$I[\![X\texttt{->modifiedOnly()}]\!](\sigma,\sigma') \equiv \begin{cases} \bot & \text{if } X' = \bot \\ \llcorner \forall i{\in}M.\ \sigma\ i = \sigma'\ i \lrcorner & \text{otherwise}. \end{cases}$$

where $X' = I[\![X]\!](\sigma,\sigma')$ and $M = (\mathrm{dom}\ \sigma \cap \mathrm{dom}\ \sigma') - \{\mathrm{OidOf}\, x |\ x \in \ulcorner X' \urcorner\}$. Thus, if we require in a postcondition `Set{}->modifiedOnly()` and exclude via `_.oclIsNew()` and `_.oclIsDeleted()` the existence of new or deleted objects, the operation is a query in the sense of the OCL standard, i. e., the `isQuery` property is true. So, whenever we have $\tau \models X$`->modifiedOnly()` and $\tau \models X$`->excludes`$(s.a)$, we can infer that $\tau \models s.a = s.a$ `@pre` (if they are valid).

2.7. Attribute Values

Depending on the specified multiplicity, the evaluation of an attribute can yield a value or a collection of values. A multiplicity defines a lower bound as well as a possibly infinite upper bound on the cardinality of the attribute's values.

2.7.1. Single-Valued Attributes

If the upper bound specified by the attribute's multiplicity is one, then an evaluation of the attribute yields a single value. Thus, the evaluation result is not a collection. If the lower bound specified by the multiplicity is zero, the evaluation is not required to yield a non-null value. In this case an evaluation of the attribute can return `null` to indicate an absence of value.

To facilitate accessing attributes with multiplicity `0..1`, the OCL standard states that single values can be used as sets by calling collection operations on them. This implicit conversion of a value to a `Set` is not defined by the standard. We argue that the resulting set cannot be constructed the same way as when evaluating a `Set` literal. Otherwise, `null` would be mapped to the singleton set containing `null`, but the standard demands that the resulting set is empty in this case. The conversion should instead be defined as follows:

```
context OclAny::asSet():T
  post: if self ≐ null then result ≐ Set{}
        else result ≐ Set{self} endif
```

2.7.2. Collection-Valued Attributes

If the upper bound specified by the attribute's multiplicity is larger than one, then an evaluation of the attribute yields a collection of values. This raises the question whether `null` can belong to this collection. The OCL standard states that `null` can be owned by collections. However, if an attribute can evaluate to a collection containing `null`, it is not clear how multiplicity constraints should be interpreted for this attribute. The question arises whether the `null` element should be counted or not when determining the cardinality of the collection. Recall that `null` denotes the absence of value in the case of a cardinality upper bound of one, so we would assume that `null` is not counted. On the other hand, the operation `size` defined for collections in OCL does count `null`.

We propose to resolve this dilemma by regarding multiplicities as optional. This point of view complies with the UML standard, that does not require lower and upper bounds to be defined for multiplicities.[5] In case a multiplicity is specified for an attribute, i. e., a lower and an upper bound are provided, we require any collection the attribute evaluates to to not contain `null`. This allows for a straightforward interpretation of the multiplicity

[5] We are however aware that a well-formedness rule of the UML standard does define a default bound of one in case a lower or upper bound is not specified.

constraint. If bounds are not provided for an attribute, we consider the attribute values to not be restricted in any way. Because in particular the cardinality of the attribute's values is not bounded, the result of an evaluation of the attribute is of collection type. As the range of values that the attribute can assume is not restricted, the attribute can evaluate to a collection containing `null`. The attribute can also evaluate to `invalid`. Allowing multiplicities to be optional in this way gives the modeler the freedom to define attributes that can assume the full ranges of values provided by their types. However, we do not permit the omission of multiplicities for association ends, since the values of association ends are not only restricted by multiplicities, but also by other constraints enforcing the semantics of associations. Hence, the values of association ends cannot be completely unrestricted.

2.7.3. The Precise Meaning of Multiplicity Constraints

We are now ready to define the meaning of multiplicity constraints by giving equivalent invariants written in OCL. Let `a` be an attribute of a class `C` with a multiplicity specifying a lower bound m and an upper bound n. Then we can define the multiplicity constraint on the values of attribute `a` to be equivalent to the following invariants written in OCL:

```
context C inv lowerBound: a->size() >= m
          inv upperBound: a->size() <= n
          inv notNull: not a->includes(null)
```

If the upper bound n is infinite, the second invariant is omitted. For the definition of these invariants we are making use of the conversion of single values to sets described in subsection 2.7.1. If $n \leq 1$, the attribute `a` evaluates to a single value, which is then converted to a `Set` on which the `size` operation is called.

If a value of the attribute `a` includes a reference to a non-existent object, the attribute call evaluates to `invalid`. As a result, the entire expressions evaluate to `invalid`, and the invariants are not satisfied. Thus, references to non-existent objects are ruled out by these invariants. We believe that this result is appropriate, since we argue that the presence of such references in a system state is usually not intended and likely to be the result of an error. If the modeler wishes to allow references to non-existent objects, she can make use of the possibility described above to omit the multiplicity.

3. Part I: Core Definitions and Library

theory
  OCL-core
imports
  Main
begin

3.0.4. Notations for the option type

First of all, we will use a more compact notation for the library option type which occur all over in our definitions and which will make the presentation more "textbook"-like:

notation *Some* ($\lfloor(\text{-})\rfloor$)
notation *None* ($\perp$)

The following function (correspon ding to *the* in the Isabelle/HOL library) is defined as the inverse of the injection *Some*.

fun *drop* :: $'\alpha$ *option* $\Rightarrow$ $'\alpha$ ($\lceil(\text{-})\rceil$)
where *drop-lift*[*simp*]: $\lceil\lfloor v\rfloor\rceil = v$

3.0.5. Minimal Notions of State and State Transitions

Next we will introduce the foundational concept of an object id (oid), which is just some infinite set.

In order to assure executability of as much as possible formulas, we fixed the type of object id's to just natural numbers.

type-synonym *oid* = *nat*

We refrained from the alternative:

type_synonym oid = ind

which is slightly more abstract but non-executable.

States are just a partial map from oid's to elements of an object universe $'\mathfrak{A}$, and state transitions pairs of states...

record ($'\mathfrak{A}$)*state* =
  heap :: *oid* $\rightharpoonup$ $'\mathfrak{A}$
  assocs :: *oid* $\rightharpoonup$ (*oid* $\times$ *oid*) *list*

type-synonym ($'\mathfrak{A}$)*st* = $'\mathfrak{A}$ *state* $\times$ $'\mathfrak{A}$ *state*

3.0.6. Prerequisite: An Abstract Interface for OCL Types

In order to have the possibility to nest collection types, such that we can give semantics to expressions like $Set\{Set\{\mathbf{2}\},null\}$, it is necessary to introduce a uniform interface for types having the *invalid* (= bottom) element. The reason is that we impose a data-invariant on raw-collection `types_code` which assures that the *invalid* element is not allowed inside the collection; all raw-collections of this form were identified with the *invalid* element itself. The construction requires that the new collection type is un-comparable with the raw-types (consisting of nested option type constructions), such that the data-invariant mussed be expressed in terms of the interface. In a second step, our base-types will be shown to be instances of this interface.

This uniform interface consists in a type class requiring the existence of a bot and a null element. The construction proceeds by abstracting the null (which is defined by $\lfloor \perp \rfloor$ on $'a\ option\ option$ to a null - element, which may have an abritrary semantic structure, and an undefinedness element $\perp$ to an abstract undefinedness element *bot* (also written $\perp$ whenever no confusion arises). As a consequence, it is necessary to redefine the notions of invalid, defined, valuation etc. on top of this interface.

This interface consists in two abstract type classes *bot* and *null* for the class of all types comprising a bot and a distinct null element.

instance *option* :: (*plus*) *plus* ⟨*proof*⟩
instance *fun* :: (*type*, *plus*) *plus* ⟨*proof*⟩

class *bot* =
  fixes $bot :: 'a$
  assumes *nonEmpty* : $\exists\ x.\ x \neq bot$

class *null* = *bot* +
  fixes $null :: 'a$
  assumes *null-is-valid* : $null \neq bot$

3.0.7. Accomodation of Basic Types to the Abstract Interface

In the following it is shown that the option-option type type is in fact in the *null* class and that function spaces over these classes again "live" in these classes. This motivates the default construction of the semantic domain for the basic types (Boolean, Integer, Reals, ...).

instantiation *option* :: (*type*)*bot*
begin
  definition *bot-option-def*: $(bot::'a\ option) \equiv (None::'a\ option)$
  instance ⟨*proof*⟩
end

instantiation *option* :: (*bot*)*null*

begin
  definition *null-option-def*: $(null::'a::bot\ option) \equiv \lfloor\ bot\ \rfloor$
  instance ⟨*proof*⟩
end

instantiation *fun* :: (*type*,*bot*) *bot*
begin
  definition *bot-fun-def*: $bot \equiv (\lambda\ x.\ bot)$

  instance ⟨*proof*⟩
end

instantiation *fun* :: (*type*,*null*) *null*
begin
 definition *null-fun-def*: $(null::'a \Rightarrow 'b::null) \equiv (\lambda\ x.\ null)$

 instance ⟨*proof*⟩
end

A trivial consequence of this adaption of the interface is that abstract and concrete versions of null are the same on base types (as could be expected).

3.1. The Semantic Space of OCL Types: Valuations.

Valuations are now functions from a state pair (built upon data universe $'\mathfrak{A}$) to an arbitrary null-type (i.e. containing at least a destinguished *null* and *invalid* element.

type-synonym $('\mathfrak{A},'\alpha)\ val = '\mathfrak{A}\ st \Rightarrow '\alpha::null$

The definitions for the constants and operations based on valuations will be geared towards a format that Isabelle can check to be a "conservative" (i.e. logically safe) axiomatic definition. By introducing an explicit interpretation function (which happens to be defined just as the identity since we are using a shallow embedding of OCL into HOL), all these definions can be rewritten into the conventional semantic "textbook" format as follows:

definition $Sem :: 'a \Rightarrow 'a\ (I[\![-]\!])$
where $I[\![x]\!] \equiv x$

As a consequence of semantic domain definition, any OCL type will have the two semantic constants *invalid* (for exceptional, aborted computation) and *null*; the latter, however is either defined

definition $invalid :: ('\mathfrak{A},'\alpha::bot)\ val$
where $invalid \equiv \lambda\ \tau.\ bot$

This conservative Isabelle definition of the polymorphic constant *invalid* is equivalent with the textbook definition:

lemma *invalid-def-textbook*: $I[\![invalid]\!]\tau = bot$
$\langle proof \rangle$

Note that the definition :

```
definition null    :: "('\<AA>,'\<alpha>::null) val"
where     "null    \<equiv> \<lambda> \<tau>. null"
```

is not necessary since we defined the entire function space over null types again as null-types; the crucial definition is $null \equiv \lambda x.\ null$. Thus, the polymporhic constant *null* is simply the result of a general type class construction. Nevertheless, we can derive the semantic textbook definition for the OCL null constant based on the abstract null:

lemma *null-def-textbook*: $I[\![null::('\mathfrak{A},'\alpha::null)\ val]\!]\ \tau = (null::'\alpha::null)$
$\langle proof \rangle$

3.2. Boolean Type and Logic

The semantic domain of the (basic) boolean type is now defined as standard: the space of valuation to *bool option option*:

type-synonym $('\mathfrak{A})Boolean = ('\mathfrak{A},bool\ option\ option)\ val$

3.2.1. Basic Constants

lemma *bot-Boolean-def* : $(bot::('\mathfrak{A})Boolean) = (\lambda\ \tau.\ \bot)$
$\langle proof \rangle$

lemma *null-Boolean-def* : $(null::('\mathfrak{A})Boolean) = (\lambda\ \tau.\ \lfloor\bot\rfloor)$
$\langle proof \rangle$

definition $true :: ('\mathfrak{A})Boolean$
where $true \equiv \lambda\ \tau.\ \lfloor\lfloor True\rfloor\rfloor$

definition $false :: ('\mathfrak{A})Boolean$
where $false \equiv\ \lambda\ \tau.\ \lfloor\lfloor False\rfloor\rfloor$

lemma *bool-split*: $X\ \tau = invalid\ \tau \vee X\ \tau = null\ \tau \vee$
$X\ \tau = true\ \tau\ \ \vee X\ \tau = false\ \tau$
$\langle proof \rangle$

lemma [*simp*]: $false\ (a,\ b) = \lfloor\lfloor False\rfloor\rfloor$
$\langle proof \rangle$

lemma [*simp*]: $true\ (a,\ b) = \lfloor\lfloor True\rfloor\rfloor$
$\langle proof \rangle$

lemma *true-def-textbook*: $I[\![true]\!]\ \tau = \lfloor\lfloor True\rfloor\rfloor$
⟨*proof*⟩

lemma *false-def-textbook*: $I[\![false]\!]\ \tau = \lfloor\lfloor False\rfloor\rfloor$
⟨*proof*⟩

Summary:

Name	Theorem
invalid-def-textbook	$I[\![invalid]\!]\ ?\tau = OCL\text{-}core.bot\text{-}class.bot$
null-def-textbook	$I[\![null]\!]\ ?\tau = null$
true-def-textbook	$I[\![true]\!]\ ?\tau = \lfloor\lfloor True\rfloor\rfloor$
false-def-textbook	$I[\![false]\!]\ ?\tau = \lfloor\lfloor False\rfloor\rfloor$

Table 3.1.: Basic semantic constant definitions of the logic (except *null*)

3.2.2. Fundamental Predicates I: Validity and Definedness

However, this has also the consequence that core concepts like definedness, validness and even cp have to be redefined on this type class:

definition *valid* :: $('\mathfrak{A},'a::null)val \Rightarrow ('\mathfrak{A})Boolean\ (\upsilon\ \text{-}\ [100]100)$
where $\upsilon\ X \equiv\ \lambda\ \tau\ .\ if\ X\ \tau = bot\ \tau\ then\ false\ \tau\ else\ true\ \tau$

lemma *valid1*[*simp*]: $\upsilon\ invalid = false$
⟨*proof*⟩

lemma *valid2*[*simp*]: $\upsilon\ null = true$
⟨*proof*⟩

lemma *valid3*[*simp*]: $\upsilon\ true = true$
⟨*proof*⟩

lemma *valid4*[*simp*]: $\upsilon\ false = true$
⟨*proof*⟩

lemma *cp-valid*: $(\upsilon\ X)\ \tau = (\upsilon\ (\lambda\ \text{-}.\ X\ \tau))\ \tau$
⟨*proof*⟩

definition *defined* :: $('\mathfrak{A},'a::null)val \Rightarrow ('\mathfrak{A})Boolean\ (\delta\ \text{-}\ [100]100)$

where $\delta\ X \equiv\ \lambda\ \tau\ .\ \mathit{if}\ X\ \tau = \mathit{bot}\ \tau\ \ \vee X\ \tau = \mathit{null}\ \tau\ \mathit{then}\ \mathit{false}\ \tau\ \mathit{else}\ \mathit{true}\ \tau$

The generalized definitions of invalid and definedness have the same properties as the old ones :

lemma *defined1*[*simp*]: $\delta\ \mathit{invalid} = \mathit{false}$
⟨*proof*⟩

lemma *defined2*[*simp*]: $\delta\ \mathit{null} = \mathit{false}$
⟨*proof*⟩

lemma *defined3*[*simp*]: $\delta\ \mathit{true} = \mathit{true}$
⟨*proof*⟩

lemma *defined4*[*simp*]: $\delta\ \mathit{false} = \mathit{true}$
⟨*proof*⟩

lemma *defined5*[*simp*]: $\delta\ \delta\ X = \mathit{true}$
⟨*proof*⟩

lemma *defined6*[*simp*]: $\delta\ \upsilon\ X = \mathit{true}$
⟨*proof*⟩

lemma *defined7*[*simp*]: $\delta\ \delta\ X = \mathit{true}$
⟨*proof*⟩

lemma *valid6*[*simp*]: $\upsilon\ \delta\ X = \mathit{true}$
⟨*proof*⟩

lemma *cp-defined*:$(\delta\ X)\tau = (\delta\ (\lambda\ \text{-}.\ X\ \tau))\ \tau$
⟨*proof*⟩

The definitions above for the constants *defined* and *valid* can be rewritten into the conventional semantic "textbook" format as follows:

lemma *defined-def-textbook*: $I[\![\delta(X)]\!]\ \tau = (\mathit{if}\ I[\![X]\!]\ \tau = I[\![\mathit{bot}]\!]\ \tau\ \ \vee I[\![X]\!]\ \tau = I[\![\mathit{null}]\!]\ \tau$
$\mathit{then}\ I[\![\mathit{false}]\!]\ \tau$
$\mathit{else}\ I[\![\mathit{true}]\!]\ \tau)$
⟨*proof*⟩

lemma *valid-def-textbook*: $I[\![\upsilon(X)]\!]\ \tau = (\mathit{if}\ I[\![X]\!]\ \tau = I[\![\mathit{bot}]\!]\ \tau$
$\mathit{then}\ I[\![\mathit{false}]\!]\ \tau$
$\mathit{else}\ I[\![\mathit{true}]\!]\ \tau)$
⟨*proof*⟩

Summary: These definitions lead quite directly to the algebraic laws on these predicates:

Name	Theorem
defined-def-textbook	$I[\![\delta\ X]\!]\ \tau = (\mathit{if}\ I[\![X]\!]\ \tau = I[\![\mathit{OCL\text{-}core.bot\text{-}class.bot}]\!]\ \tau \vee I[\![X]\!]\ \tau = I[\![\mathit{null}]\!]\ \tau\ \mathit{then}\ I[\![\mathit{fals}$
valid-def-textbook	$I[\![\upsilon\ X]\!]\ \tau = (\mathit{if}\ I[\![X]\!]\ \tau = I[\![\mathit{OCL\text{-}core.bot\text{-}class.bot}]\!]\ \tau\ \mathit{then}\ I[\![\mathit{false}]\!]\ \tau\ \mathit{else}\ I[\![\mathit{true}]\!]\ \tau)$

Table 3.2.: Basic predicate definitions of the logic.)

Name	Theorem
defined1	$\delta\ \mathit{invalid} = \mathit{false}$
defined2	$\delta\ \mathit{null} = \mathit{false}$
defined3	$\delta\ \mathit{true} = \mathit{true}$
defined4	$\delta\ \mathit{false} = \mathit{true}$
defined5	$\delta\ \delta\ ?X\ = \mathit{true}$
defined6	$\delta\ \upsilon\ ?X\ = \mathit{true}$
defined7	$\delta\ \delta\ ?X\ = \mathit{true}$

Table 3.3.: Laws of the basic predicates of the logic.)

3.2.3. Fundamental Predicates II: Logical (Strong) Equality

Note that we define strong equality extremely generic, even for types that contain an *null* or $\perp$ element:

definition *StrongEq*::$['\mathfrak{A}\ st \Rightarrow '\alpha,'\mathfrak{A}\ st \Rightarrow '\alpha] \Rightarrow ('\mathfrak{A})\mathit{Boolean}$ (**infixl** $\triangleq$ *30*)
where $X \triangleq Y \equiv \lambda\,\tau.\ \lfloor\lfloor X\ \tau = Y\ \tau\ \rfloor\rfloor$

Equality reasoning in OCL is not humpty dumpty. While strong equality is clearly an equivalence:

lemma *StrongEq-refl* [*simp*]: $(X \triangleq X) = \mathit{true}$
⟨*proof*⟩

lemma *StrongEq-sym*: $(X \triangleq Y) = (Y \triangleq X)$
⟨*proof*⟩

lemma *StrongEq-trans-strong* [*simp*]:
assumes A: $(X \triangleq Y) = \mathit{true}$
and B: $(Y \triangleq Z) = \mathit{true}$
shows $(X \triangleq Z) = \mathit{true}$
⟨*proof*⟩

... it is only in a limited sense a congruence, at least from the point of view of this semantic theory. The point is that it is only a congruence on OCL- expressions, not arbitrary

HOL expressions (with which we can mix Essential OCL expressions. A semantic — not syntactic — characterization of OCL-expressions is that they are *context-passing* or *context-invariant*, i.e. the context of an entire OCL expression, i.e. the pre-and post-state it referes to, is passed constantly and unmodified to the sub-expressions, i.e. all sub-expressions inside an OCL expression refer to the same context. Expressed formally, this boils down to:

lemma *StrongEq-subst* :
  assumes *cp*: $\bigwedge X.\ P(X)\tau = P(\lambda\ \text{-}.\ X\ \tau)\tau$
  and *eq*: $(X \triangleq Y)\tau = \mathit{true}\ \tau$
  shows $(P\ X \triangleq P\ Y)\tau = \mathit{true}\ \tau$
  ⟨*proof*⟩

3.2.4. Fundamental Predicates III

And, last but not least,

lemma *defined8*[*simp*]: $\delta\ (X \triangleq Y) = \mathit{true}$
  ⟨*proof*⟩

lemma *valid5*[*simp*]: $\upsilon\ (X \triangleq Y) = \mathit{true}$
  ⟨*proof*⟩

lemma *cp-StrongEq*: $(X \triangleq Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau) \triangleq (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
⟨*proof*⟩

The semantics of strict equality of OCL is constructed by overloading: for each base type, there is an equality.

3.2.5. Logical Connectives and their Universal Properties

It is a design goal to give OCL a semantics that is as closely as possible to a "logical system" in a known sense; a specification logic where the logical connectives can not be understood other that having the truth-table aside when reading fails its purpose in our view.

Practically, this means that we want to give a definition to the core operations to be as close as possible to the lattice laws; this makes also powerful symbolic normalizations of OCL specifications possible as a pre-requisite for automated theorem provers. For example, it is still possible to compute without any definedness- and validity reasoning the DNF of an OCL specification; be it for test-case generations or for a smooth transition to a two-valued representation of the specification amenable to fast standard SMT-solvers, for example.

Thus, our representation of the OCL is merely a 4-valued Kleene-Logics with *invalid* as least, *null* as middle and *true* resp. *false* as unrelated top-elements.

definition *not* :: $('\mathfrak{A})\mathit{Boolean} \Rightarrow ('\mathfrak{A})\mathit{Boolean}$
where $\mathit{not}\ X \equiv\ \lambda\ \tau\ .\ \mathit{case}\ X\ \tau\ \mathit{of}$

$$\begin{array}{ll} \quad\perp & \Rightarrow \perp \\ \mid \lfloor \perp \rfloor & \Rightarrow \lfloor \perp \rfloor \\ \mid \lfloor\lfloor x \rfloor\rfloor & \Rightarrow \lfloor\lfloor \neg x \rfloor\rfloor \end{array}$$

lemma *cp-not*: $(not\ X)\tau = (not\ (\lambda\ \text{-}.\ X\ \tau))\ \tau$
⟨*proof*⟩

lemma *not1*[*simp*]: *not invalid* = *invalid*
⟨*proof*⟩

lemma *not2*[*simp*]: *not null* = *null*
⟨*proof*⟩

lemma *not3*[*simp*]: *not true* = *false*
⟨*proof*⟩

lemma *not4*[*simp*]: *not false* = *true*
⟨*proof*⟩

lemma *not-not*[*simp*]: *not* (*not X*) = *X*
⟨*proof*⟩

definition *ocl-and* :: $[('\mathfrak{A})Boolean,\ ('\mathfrak{A})Boolean] \Rightarrow ('\mathfrak{A})Boolean$ (**infixl** *and 30*)
where $X\ and\ Y \equiv\ (\lambda\ \tau\ .\ case\ X\ \tau\ of$

$$\begin{array}{l} \quad\perp\ \Rightarrow (case\ Y\ \tau\ of \\ \qquad\qquad \perp \Rightarrow\ \perp \\ \qquad\qquad \mid \lfloor\perp\rfloor \Rightarrow \perp \\ \qquad\qquad \mid \lfloor\lfloor True\rfloor\rfloor \Rightarrow\ \perp \\ \qquad\qquad \mid \lfloor\lfloor False\rfloor\rfloor \Rightarrow\ \lfloor\lfloor False\rfloor\rfloor) \\ \mid \lfloor\ \perp\ \rfloor \Rightarrow (case\ Y\ \tau\ of \\ \qquad\qquad \perp \Rightarrow\ \perp \\ \qquad\qquad \mid \lfloor\perp\rfloor \Rightarrow \lfloor\perp\rfloor \\ \qquad\qquad \mid \lfloor\lfloor True\rfloor\rfloor \Rightarrow \lfloor\perp\rfloor \\ \qquad\qquad \mid \lfloor\lfloor False\rfloor\rfloor \Rightarrow\ \lfloor\lfloor False\rfloor\rfloor) \\ \mid \lfloor\lfloor True\rfloor\rfloor \Rightarrow (case\ Y\ \tau\ of \\ \qquad\qquad \perp \Rightarrow\ \perp \\ \qquad\qquad \mid \lfloor\perp\rfloor \Rightarrow \lfloor\perp\rfloor \\ \qquad\qquad \mid \lfloor\lfloor y\rfloor\rfloor \Rightarrow\ \lfloor\lfloor y\rfloor\rfloor) \\ \mid \lfloor\lfloor False\rfloor\rfloor \Rightarrow\ \lfloor\lfloor\ False\ \rfloor\rfloor) \end{array}$$

Note that *not* is *not* defined as a strict function; proximity to lattice laws implies that we *need* a definition of *not* that satisfies $not(not(x))=x$.

In textbook notation, the logical core constructs *not* and *op and* were represented as follows:

lemma *textbook-not*:

$$
\begin{array}{llll}
I[\![not(X)]\!]\ \tau = & (case\ I[\![X]\!]\ \tau\ of & \bot & \Rightarrow \bot \\
& & \mid\ \lfloor \bot \rfloor & \Rightarrow \lfloor \bot \rfloor \\
& & \mid\ \lfloor\lfloor x \rfloor\rfloor & \Rightarrow \lfloor\lfloor \neg\ x \rfloor\rfloor)
\end{array}
$$

$\langle proof \rangle$

lemma *textbook-and*:

$$
\begin{array}{l}
I[\![X\ and\ Y]\!]\ \tau = (case\ I[\![X]\!]\ \tau\ of \\
\quad \bot \ \Rightarrow (case\ I[\![Y]\!]\ \tau\ of \\
\qquad \bot \Rightarrow \bot \\
\qquad \mid \lfloor\bot\rfloor \Rightarrow \bot \\
\qquad \mid \lfloor\lfloor True\rfloor\rfloor \Rightarrow \bot \\
\qquad \mid \lfloor\lfloor False\rfloor\rfloor \Rightarrow \lfloor\lfloor False\rfloor\rfloor) \\
\quad \mid \lfloor \bot \rfloor \Rightarrow (case\ I[\![Y]\!]\ \tau\ of \\
\qquad \bot \Rightarrow \bot \\
\qquad \mid \lfloor\bot\rfloor \Rightarrow \lfloor\bot\rfloor \\
\qquad \mid \lfloor\lfloor True\rfloor\rfloor \Rightarrow \lfloor\bot\rfloor \\
\qquad \mid \lfloor\lfloor False\rfloor\rfloor \Rightarrow \lfloor\lfloor False\rfloor\rfloor) \\
\quad \mid \lfloor\lfloor True\rfloor\rfloor \Rightarrow (case\ I[\![Y]\!]\ \tau\ of \\
\qquad \bot \Rightarrow \bot \\
\qquad \mid \lfloor\bot\rfloor \Rightarrow \lfloor\bot\rfloor \\
\qquad \mid \lfloor\lfloor y\rfloor\rfloor \Rightarrow \lfloor\lfloor y\rfloor\rfloor) \\
\quad \mid \lfloor\lfloor False\rfloor\rfloor \Rightarrow \lfloor\lfloor\ False\ \rfloor\rfloor)
\end{array}
$$

$\langle proof \rangle$

definition *ocl-or* :: $[('\mathfrak{A})Boolean, ('\mathfrak{A})Boolean] \Rightarrow ('\mathfrak{A})Boolean$
(**infixl** *or 25*)
where $X\ or\ Y \equiv not(not\ X\ and\ not\ Y)$

definition *ocl-implies* :: $[('\mathfrak{A})Boolean, ('\mathfrak{A})Boolean] \Rightarrow ('\mathfrak{A})Boolean$
(**infixl** *implies 25*)
where $X\ implies\ Y \equiv not\ X\ or\ Y$

lemma *cp-ocl-and*:$(X\ and\ Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau)\ and\ (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
$\langle proof \rangle$

lemma *cp-ocl-or*:$((X::('\mathfrak{A})Boolean)\ or\ Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau)\ or\ (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
$\langle proof \rangle$

lemma *cp-ocl-implies*:$(X\ implies\ Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau)\ implies\ (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
$\langle proof \rangle$

lemma *ocl-and1*[*simp*]: $(invalid\ and\ true) = invalid$
$\langle proof \rangle$
lemma *ocl-and2*[*simp*]: $(invalid\ and\ false) = false$
$\langle proof \rangle$
lemma *ocl-and3*[*simp*]: $(invalid\ and\ null) = invalid$

⟨*proof*⟩
lemma *ocl-and4*[*simp*]: (*invalid and invalid*) = *invalid*
⟨*proof*⟩

lemma *ocl-and5*[*simp*]: (*null and true*) = *null*
⟨*proof*⟩
lemma *ocl-and6*[*simp*]: (*null and false*) = *false*
⟨*proof*⟩
lemma *ocl-and7*[*simp*]: (*null and null*) = *null*
⟨*proof*⟩
lemma *ocl-and8*[*simp*]: (*null and invalid*) = *invalid*
⟨*proof*⟩

lemma *ocl-and9*[*simp*]: (*false and true*) = *false*
⟨*proof*⟩
lemma *ocl-and10*[*simp*]: (*false and false*) = *false*
⟨*proof*⟩
lemma *ocl-and11*[*simp*]: (*false and null*) = *false*
⟨*proof*⟩
lemma *ocl-and12*[*simp*]: (*false and invalid*) = *false*
⟨*proof*⟩

lemma *ocl-and13*[*simp*]: (*true and true*) = *true*
⟨*proof*⟩
lemma *ocl-and14*[*simp*]: (*true and false*) = *false*
⟨*proof*⟩
lemma *ocl-and15*[*simp*]: (*true and null*) = *null*
⟨*proof*⟩
lemma *ocl-and16*[*simp*]: (*true and invalid*) = *invalid*
⟨*proof*⟩

lemma *ocl-and-idem*[*simp*]: (X *and* X) = X
⟨*proof*⟩

lemma *ocl-and-commute*: (X *and* Y) = (Y *and* X)
⟨*proof*⟩

lemma *ocl-and-false1*[*simp*]: (*false and* X) = *false*
⟨*proof*⟩

lemma *ocl-and-false2*[*simp*]: (X *and false*) = *false*
⟨*proof*⟩

lemma *ocl-and-true1*[*simp*]: (*true and* X) = X
⟨*proof*⟩

lemma *ocl-and-true2*[*simp*]: (X *and true*) = X

⟨*proof*⟩

lemma *ocl-and-assoc*: $(X \ and \ (Y \ and \ Z)) = (X \ and \ Y \ and \ Z)$
⟨*proof*⟩

lemma *ocl-or-idem*[*simp*]: $(X \ or \ X) = X$
⟨*proof*⟩

lemma *ocl-or-commute*: $(X \ or \ Y) = (Y \ or \ X)$
⟨*proof*⟩

lemma *ocl-or-false1*[*simp*]: $(false \ or \ Y) = Y$
⟨*proof*⟩

lemma *ocl-or-false2*[*simp*]: $(Y \ or \ false) = Y$
⟨*proof*⟩

lemma *ocl-or-true1*[*simp*]: $(true \ or \ Y) = true$
⟨*proof*⟩

lemma *ocl-or-true2*: $(Y \ or \ true) = true$
⟨*proof*⟩

lemma *ocl-or-assoc*: $(X \ or \ (Y \ or \ Z)) = (X \ or \ Y \ or \ Z)$
⟨*proof*⟩

lemma *deMorgan1*: $not(X \ and \ Y) = ((not \ X) \ or \ (not \ Y))$
⟨*proof*⟩

lemma *deMorgan2*: $not(X \ or \ Y) = ((not \ X) \ and \ (not \ Y))$
⟨*proof*⟩

3.3. A Standard Logical Calculus for OCL

Besides the need for algebraic laws for OCL in order to normalize

definition *OclValid* :: $[('\mathfrak{A})st, ('\mathfrak{A})Boolean] \Rightarrow bool \ ((1(\text{-})/ \models (\text{-})) \ 50)$
where $\tau \models P \equiv ((P \ \tau) = true \ \tau)$

3.3.1. Global vs. Local Judgements

lemma *transform1*: $P = true \Longrightarrow \tau \models P$
⟨*proof*⟩

lemma *transform1-rev*: $\forall \ \tau. \ \tau \models P \Longrightarrow P = true$
⟨*proof*⟩

lemma *transform2*: $(P = Q) \Longrightarrow ((\tau \models P) = (\tau \models Q))$
⟨*proof*⟩

lemma *transform2-rev*: $\forall\ \tau.\ (\tau \models \delta\ P) \wedge (\tau \models \delta\ Q) \wedge (\tau \models P) = (\tau \models Q) \Longrightarrow P = Q$
⟨*proof*⟩

However, certain properties (like transitivity) can not be *transformed* from the global level to the local one, they have to be re-proven on the local level.

lemma *transform3*:
assumes $H : P = true \Longrightarrow Q = true$
shows $\tau \models P \Longrightarrow \tau \models Q$
⟨*proof*⟩

3.3.2. Local Validity and Meta-logic

lemma *foundation1*[*simp*]: $\tau \models true$
⟨*proof*⟩

lemma *foundation2*[*simp*]: $\neg(\tau \models false)$
⟨*proof*⟩

lemma *foundation3*[*simp*]: $\neg(\tau \models invalid)$
⟨*proof*⟩

lemma *foundation4*[*simp*]: $\neg(\tau \models null)$
⟨*proof*⟩

lemma *bool-split-local*[*simp*]:
$(\tau \models (x \triangleq invalid)) \vee (\tau \models (x \triangleq null)) \vee (\tau \models (x \triangleq true)) \vee (\tau \models (x \triangleq false))$
⟨*proof*⟩

lemma *def-split-local*:
$(\tau \models \delta\ x) = ((\neg(\tau \models (x \triangleq invalid))) \wedge (\neg\ (\tau \models (x \triangleq null))))$
⟨*proof*⟩

lemma *foundation5*:
$\tau \models (P\ and\ Q) \Longrightarrow (\tau \models P) \wedge (\tau \models Q)$
⟨*proof*⟩

lemma *foundation6*:
$\tau \models P \Longrightarrow \tau \models \delta\ P$
⟨*proof*⟩

lemma *foundation7*[*simp*]:
$(\tau \models not\ (\delta\ x)) = (\neg\ (\tau \models \delta\ x))$
⟨*proof*⟩

lemma *foundation7'*[*simp*]:

$(\tau \models not\ (\upsilon\ x)) = (\neg\ (\tau \models \upsilon\ x))$
$\langle proof \rangle$

Key theorem for the Delta-closure: either an expression is defined, or it can be replaced (substituted via `StrongEq_L_subst2`; see below) by invalid or null. Strictness-reduction rules will usually reduce these substituted terms drastically.

lemma *foundation8*:
$(\tau \models \delta\ x) \vee (\tau \models (x \triangleq invalid)) \vee (\tau \models (x \triangleq null))$
$\langle proof \rangle$

lemma *foundation9*:
$\tau \models \delta\ x \Longrightarrow (\tau \models not\ x) = (\neg\ (\tau \models x))$
$\langle proof \rangle$

lemma *foundation10*:
$\tau \models \delta\ x \Longrightarrow \tau \models \delta\ y \Longrightarrow (\tau \models (x\ and\ y)) = (\ (\tau \models x) \wedge (\tau \models y))$
$\langle proof \rangle$

lemma *foundation11*:
$\tau \models \delta\ x \Longrightarrow\ \ \tau \models \delta\ y \Longrightarrow (\tau \models (x\ or\ y)) = (\ (\tau \models x) \vee (\tau \models y))$
$\langle proof \rangle$

lemma *foundation12*:
$\tau \models \delta\ x \Longrightarrow\ \ \tau \models \delta\ y \Longrightarrow (\tau \models (x\ implies\ y)) = (\ (\tau \models x) \longrightarrow (\tau \models y))$
$\langle proof \rangle$

lemma *foundation13*:$(\tau \models A \triangleq true) \quad = (\tau \models A)$
$\langle proof \rangle$

lemma *foundation14*:$(\tau \models A \triangleq false) \quad = (\tau \models not\ A)$
$\langle proof \rangle$

lemma *foundation15*:$(\tau \models A \triangleq invalid) = (\tau \models not(\upsilon\ A))$
$\langle proof \rangle$

lemma *foundation16*: $\tau \models (\delta\ X) = (X\ \tau \neq bot \wedge X\ \tau \neq null)$
$\langle proof \rangle$

lemmas *foundation17* = *foundation16*[*THEN iffD1*,*standard*]

lemma *foundation18*: $\tau \models (\upsilon\ X) = (X\ \tau \neq invalid\ \tau)$
$\langle proof \rangle$

lemma *foundation18'*: $\tau \models (\upsilon\ X) = (X\ \tau \neq bot)$
⟨*proof*⟩

lemmas *foundation19* = *foundation18*[*THEN iffD1*,*standard*]

lemma *foundation20* : $\tau \models (\delta\ X) \Longrightarrow \tau \models \upsilon\ X$
⟨*proof*⟩

lemma *foundation21*: $(not\ A \triangleq not\ B) = (A \triangleq B)$
⟨*proof*⟩

lemma *foundation22*: $(\tau \models (X \triangleq Y)) = (X\ \tau = Y\ \tau)$
⟨*proof*⟩

lemma *foundation23*: $(\tau \models P) = (\tau \models (\lambda\ \text{-}\ .\ P\ \tau))$
⟨*proof*⟩

lemmas *cp-validity=foundation23*

lemma *defined-not-I* : $\tau \models \delta\ (x) \Longrightarrow \tau \models \delta\ (not\ x)$
⟨*proof*⟩

lemma *valid-not-I* : $\tau \models \upsilon\ (x) \Longrightarrow \tau \models \upsilon\ (not\ x)$
⟨*proof*⟩

lemma *defined-and-I* : $\tau \models \delta\ (x) \Longrightarrow\ \tau \models \delta\ (y) \Longrightarrow \tau \models \delta\ (x\ and\ y)$
⟨*proof*⟩

lemma *valid-and-I* : $\tau \models \upsilon\ (x) \Longrightarrow\ \tau \models \upsilon\ (y) \Longrightarrow \tau \models \upsilon\ (x\ and\ y)$
⟨*proof*⟩

3.3.3. Local Judgements and Strong Equality

lemma *StrongEq-L-refl*: $\tau \models (x \triangleq x)$
⟨*proof*⟩

lemma *StrongEq-L-sym*: $\tau \models (x \triangleq y) \Longrightarrow \tau \models (y \triangleq x)$
⟨*proof*⟩

lemma *StrongEq-L-trans*: $\tau \models (x \triangleq y) \Longrightarrow \tau \models (y \triangleq z) \Longrightarrow \tau \models (x \triangleq z)$
⟨*proof*⟩

In order to establish substitutivity (which does not hold in general HOL-formulas we introduce the following predicate that allows for a calculus of the necessary side-conditions.

definition *cp* :: $(('\mathfrak{A},'\alpha)\ val \Rightarrow ('\mathfrak{A},'\beta)\ val) \Rightarrow bool$
where $cp\ P \equiv (\exists\ f.\ \forall\ X\ \tau.\ P\ X\ \tau = f\ (X\ \tau)\ \tau)$

The rule of substitutivity in HOL-OCL holds only for context-passing expressions - i.e. those, that pass the context τ without changing it. Fortunately, all operators of the OCL language satisfy this property (but not all HOL operators).

lemma *StrongEq-L-subst1*: $\bigwedge \tau.\ cp\ P \Longrightarrow \tau \models (x \triangleq y) \Longrightarrow \tau \models (P\ x \triangleq P\ y)$
⟨*proof*⟩

lemma *StrongEq-L-subst2*:
$\bigwedge \tau.\ \ cp\ P \Longrightarrow \tau \models (x \triangleq y) \Longrightarrow \tau \models (P\ x) \Longrightarrow \tau \models (P\ y)$
⟨*proof*⟩

lemma *cpI1*:
$(\forall\ X\ \tau.\ f\ X\ \tau = f(\lambda\text{-}.\ X\ \tau)\ \tau) \Longrightarrow cp\ P \Longrightarrow cp(\lambda X.\ f\ (P\ X))$
⟨*proof*⟩

lemma *cpI2*:
$(\forall\ X\ Y\ \tau.\ f\ X\ Y\ \tau = f(\lambda\text{-}.\ X\ \tau)(\lambda\text{-}.\ Y\ \tau)\ \tau) \Longrightarrow$
$cp\ P \Longrightarrow cp\ Q \Longrightarrow cp(\lambda X.\ f\ (P\ X)\ (Q\ X))$
⟨*proof*⟩

lemma *cp-const* : $cp(\lambda\text{-}.\ c)$
⟨*proof*⟩

lemma *cp-id* : $cp(\lambda X.\ X)$
⟨*proof*⟩

lemmas *cp-intro*[*simp*,*intro*!] =
cp-const
cp-id
cp-defined[*THEN allI*[*THEN allI*[*THEN cpI1*], *of defined*]]
cp-valid[*THEN allI*[*THEN allI*[*THEN cpI1*], *of valid*]]
cp-not[*THEN allI*[*THEN allI*[*THEN cpI1*], *of not*]]
cp-ocl-and[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of op and*]]
cp-ocl-or[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of op or*]]
cp-ocl-implies[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of op implies*]]
cp-StrongEq[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]],
of StrongEq]]

3.3.4. Laws to Establish Definedness (Delta-Closure)

For the logical connectives, we have — beyond $?\tau \models ?P \Longrightarrow ?\tau \models \delta\ ?P$ — the following facts:

lemma *ocl-not-defargs*:
$\tau \models (not\ P) \Longrightarrow \tau \models \delta\ P$
⟨*proof*⟩

So far, we have only one strict Boolean predicate (-family): The strict equality.

3.4. Miscellaneous: OCL's if then else endif

definition *if-ocl* :: $[('\mathfrak{A})Boolean\ ,\ ('\mathfrak{A},'\alpha::null)\ val,\ ('\mathfrak{A},'\alpha)\ val] \Rightarrow ('\mathfrak{A},'\alpha)\ val$
$(if\ (\text{-})\ then\ (\text{-})\ else\ (\text{-})\ endif\ [10,10,10]50)$
where $(if\ C\ then\ B_1\ else\ B_2\ endif) = (\lambda\ \tau.\ if\ (\delta\ C)\ \tau = true\ \tau$
$then\ (if\ (C\ \tau) = true\ \tau$
$then\ B_1\ \tau$
$else\ B_2\ \tau)$
$else\ invalid\ \tau)$

lemma *cp-if-ocl*:$((if\ C\ then\ B_1\ else\ B_2\ endif)\ \tau =$
$(if\ (\lambda\ \text{-}.\ C\ \tau)\ then\ (\lambda\ \text{-}.\ B_1\ \tau)\ else\ (\lambda\ \text{-}.\ B_2\ \tau)\ endif)\ \tau)$
⟨*proof*⟩

lemma *if-ocl-invalid* [*simp*]: $(if\ invalid\ then\ B_1\ else\ B_2\ endif) = invalid$
⟨*proof*⟩

lemma *if-ocl-null* [*simp*]: $(if\ null\ then\ B_1\ else\ B_2\ endif) = invalid$
⟨*proof*⟩

lemma *if-ocl-true* [*simp*]: $(if\ true\ then\ B_1\ else\ B_2\ endif) = B_1$
⟨*proof*⟩

lemma *if-ocl-true'* [*simp*]: $\tau \models P \Longrightarrow (if\ P\ then\ B_1\ else\ B_2\ endif)\tau = B_1\ \tau$
⟨*proof*⟩

lemma *if-ocl-false* [*simp*]: $(if\ false\ then\ B_1\ else\ B_2\ endif) = B_2$
⟨*proof*⟩

lemma *if-ocl-false'* [*simp*]: $\tau \models not\ P \Longrightarrow (if\ P\ then\ B_1\ else\ B_2\ endif)\tau = B_2\ \tau$
⟨*proof*⟩

lemma *if-ocl-idem1*[*simp*]:$(if\ \delta\ X\ then\ A\ else\ A\ endif) = A$
⟨*proof*⟩

lemma *if-ocl-idem2*[*simp*]:$(if\ \upsilon\ X\ then\ A\ else\ A\ endif) = A$
⟨*proof*⟩

end

theory *OCL-lib*
imports *OCL-core*
begin

3.5. Basic Types like Void, Boolean and Integer

Since Integer is again a basic type, we define its semantic domain as the valuations over *int option option*

type-synonym $('\mathfrak{A})Integer = ('\mathfrak{A},int\ option\ option)\ val$

type-synonym $('\mathfrak{A})\ Void = ('\mathfrak{A},unit\ option)\ val$

Note that this *minimal* OCL type contains only two elements: undefined and null. For technical reasons, he does not contain to the null-class yet.

3.5.1. Strict equalities on Basic Types.

Note that the strict equality on basic types (actually on all types) must be exceptionally defined on null — otherwise the entire concept of null in the language does not make much sense. This is an important exception from the general rule that null arguments — especially if passed as "self"-argument — lead to invalid results.

consts *StrictRefEq* :: $[('\mathfrak{A},'a)val,('\mathfrak{A},'a)val] \Rightarrow ('\mathfrak{A})Boolean$ (**infixl** $\doteq$ *30*)

syntax
  notequal :: $('\mathfrak{A})Boolean \Rightarrow ('\mathfrak{A})Boolean \Rightarrow ('\mathfrak{A})Boolean$ (**infix** <> *40*)
translations
  $a <> b == CONST\ not(\ a \doteq b)$

defs *StrictRefEq-int*[*code-unfold*] :
$$(x{::}('\mathfrak{A})Integer) \doteq y \equiv \lambda\,\tau.\ if\ (\upsilon\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau\ then\ (x \triangleq y)\ \tau\ else\ invalid\ \tau$$

defs *StrictRefEq-bool*[*code-unfold*] :
$$(x{::}('\mathfrak{A})Boolean) \doteq y \equiv \lambda\,\tau.\ if\ (\upsilon\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau\ then\ (x \triangleq y)\tau\ else\ invalid\ \tau$$

3.5.2. Logic and algebraic layer on Basic Types.

lemma *RefEq-int-refl*[*simp*,*code-unfold*] :
$((x{::}('\mathfrak{A})Integer) \doteq x) = (if\ (\upsilon\ x)\ then\ true\ else\ invalid\ endif)$
⟨*proof*⟩

lemma *RefEq-bool-refl*[*simp*,*code-unfold*] :
$((x{::}('\mathfrak{A})Boolean) \doteq x) = (if\ (\upsilon\ x)\ then\ true\ else\ invalid\ endif)$
⟨*proof*⟩

lemma *StrictRefEq-int-strict1*[*simp*] : $((x{::}('\mathfrak{A})Integer) \doteq invalid) = invalid$
⟨*proof*⟩

lemma *StrictRefEq-int-strict2* [*simp*] : $(invalid \doteq (x::({}'\mathfrak{A})Integer)) = invalid$
⟨*proof*⟩

lemma *StrictRefEq-bool-strict1* [*simp*] : $((x::({}'\mathfrak{A})Boolean) \doteq invalid) = invalid$
⟨*proof*⟩

lemma *StrictRefEq-bool-strict2* [*simp*] : $(invalid \doteq (x::({}'\mathfrak{A})Boolean)) = invalid$
⟨*proof*⟩

lemma *strictEqBool-vs-strongEq*:
$\tau \models (\upsilon\ x) \Longrightarrow \tau \models (\upsilon\ y) \Longrightarrow (\tau \models (((x::({}'\mathfrak{A})Boolean) \doteq y) \triangleq (x \triangleq y)))$
⟨*proof*⟩

lemma *strictEqInt-vs-strongEq*:
$\tau \models (\upsilon\ x) \Longrightarrow \tau \models (\upsilon\ y) \Longrightarrow (\tau \models (((x::({}'\mathfrak{A})Integer) \doteq y) \triangleq (x \triangleq y)))$
⟨*proof*⟩

lemma *strictEqBool-defargs*:
$\tau \models ((x::({}'\mathfrak{A})Boolean) \doteq y) \Longrightarrow (\tau \models (\upsilon\ x)) \wedge (\tau \models (\upsilon\ y))$
⟨*proof*⟩

lemma *strictEqInt-defargs*:
$\tau \models ((x::({}'\mathfrak{A})Integer) \doteq y) \Longrightarrow (\tau \models (\upsilon\ x)) \wedge (\tau \models (\upsilon\ y))$
⟨*proof*⟩

lemma *strictEqBool-valid-args-valid*:
$(\tau \models \delta((x::({}'\mathfrak{A})Boolean) \doteq y)) = ((\tau \models (\upsilon\ x)) \wedge (\tau \models (\upsilon\ y)))$
⟨*proof*⟩

lemma *strictEqInt-valid-args-valid*:
$(\tau \models \delta((x::({}'\mathfrak{A})Integer) \doteq y)) = ((\tau \models (\upsilon\ x)) \wedge (\tau \models (\upsilon\ y)))$
⟨*proof*⟩

lemma *StrictRefEq-int-strict* :
assumes A: $\upsilon\ (x::({}'\mathfrak{A})Integer) = true$
and B: $\upsilon\ y = true$
shows $\upsilon\ (x \doteq y) = true$
⟨*proof*⟩

lemma *StrictRefEq-int-strict'* :
assumes A: $\upsilon\ (((x::({}'\mathfrak{A})Integer)) \doteq y) = true$

shows $\upsilon\ x = true \wedge \upsilon\ y = true$
$\langle proof \rangle$

lemma *StrictRefEq-int-strict''* : $\delta\ ((x::({}'\mathfrak{A})Integer) \doteq y) = (\upsilon(x)\ and\ \upsilon(y))$
$\langle proof \rangle$

lemma *StrictRefEq-bool-strict''* : $\delta\ ((x::({}'\mathfrak{A})Boolean) \doteq y) = (\upsilon(x)\ and\ \upsilon(y))$
$\langle proof \rangle$

lemma *cp-StrictRefEq-bool*:
$((X::({}'\mathfrak{A})Boolean) \doteq Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau) \doteq (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
$\langle proof \rangle$

lemma *cp-StrictRefEq-int*:
$((X::({}'\mathfrak{A})Integer) \doteq Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau) \doteq (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
$\langle proof \rangle$

lemmas *cp-intro*[*simp*,*intro*!] =
cp-intro
cp-StrictRefEq-bool[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of StrictRefEq*]]
cp-StrictRefEq-int[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of StrictRefEq*]]

definition *ocl-zero* $::({}'\mathfrak{A})Integer$ (**0**)
where $\mathbf{0} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 0::int\rfloor\rfloor)$

definition *ocl-one* $::({}'\mathfrak{A})Integer$ (**1**)
where $\mathbf{1}\ = (\lambda\ \text{-}\ .\ \lfloor\lfloor 1::int\rfloor\rfloor)$

definition *ocl-two* $::({}'\mathfrak{A})Integer$ (**2**)
where $\mathbf{2} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 2::int\rfloor\rfloor)$

definition *ocl-three* $::({}'\mathfrak{A})Integer$ (**3**)
where $\mathbf{3} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 3::int\rfloor\rfloor)$

definition *ocl-four* $::({}'\mathfrak{A})Integer$ (**4**)
where $\mathbf{4} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 4::int\rfloor\rfloor)$

definition *ocl-five* $::({}'\mathfrak{A})Integer$ (**5**)
where $\mathbf{5} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 5::int\rfloor\rfloor)$

definition *ocl-six* $::({}'\mathfrak{A})Integer$ (**6**)
where $\mathbf{6} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 6::int\rfloor\rfloor)$

definition *ocl-seven* $::({}'\mathfrak{A})Integer$ (**7**)
where $\mathbf{7} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 7::int\rfloor\rfloor)$

definition *ocl-eight* ::($'\mathfrak{A}$)*Integer* (**8**)
where $\mathbf{8} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 8{::}int \rfloor\rfloor)$

definition *ocl-nine* ::($'\mathfrak{A}$)*Integer* (**9**)
where $\mathbf{9} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 9{::}int \rfloor\rfloor)$

definition *ten-nine* ::($'\mathfrak{A}$)*Integer* (**10**)
where $\mathbf{10} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 10{::}int \rfloor\rfloor)$

Here is a way to cast in standard operators via the type class system of Isabelle.

Here follows a list of code-examples, that explain the meanings of the above definitions by compilation to code and execution to "True".

3.5.3. Test Statements on Basic Types.

Elementary computations on Booleans

value $\tau_0 \models \upsilon(true)$
value $\tau_0 \models \delta(false)$
value $\neg(\tau_0 \models \delta(null))$
value $\neg(\tau_0 \models \delta(invalid))$
value $\tau_0 \models \upsilon((null{::}('\mathfrak{A})Boolean))$
value $\neg(\tau_0 \models \upsilon(invalid))$
value $\tau_0 \models (true\ and\ true)$
value $\tau_0 \models (true\ and\ true \triangleq true)$
value $\tau_0 \models ((null\ or\ null) \triangleq null)$
value $\tau_0 \models ((null\ or\ null) \doteq null)$
value $\tau_0 \models ((true \triangleq false) \triangleq false)$
value $\tau_0 \models ((invalid \triangleq false) \triangleq false)$
value $\tau_0 \models ((invalid \doteq false) \triangleq invalid)$

Elementary computations on Integer

value $\tau_0 \models \upsilon(\mathbf{4})$
value $\tau_0 \models \delta(\mathbf{4})$
value $\tau_0 \models \upsilon((null{::}('\mathfrak{A})Integer))$
value $\tau_0 \models (invalid \triangleq invalid\ )$
value $\tau_0 \models (null \triangleq null\ )$
value $\tau_0 \models (\mathbf{4} \triangleq \mathbf{4})$
value $\neg(\tau_0 \models (\mathbf{9} \triangleq \mathbf{10}\ ))$
value $\neg(\tau_0 \models (invalid \triangleq \mathbf{10}\ ))$
value $\neg(\tau_0 \models (null \triangleq \mathbf{10}\ ))$
value $\neg(\tau_0 \models (invalid \doteq (invalid{::}('\mathfrak{A})Integer)))$
value $\tau_0 \models (null \doteq (null{::}('\mathfrak{A})Integer)\ )$
value $\tau_0 \models (null \doteq (null{::}('\mathfrak{A})Integer)\ )$
value $\tau_0 \models (\mathbf{4} \doteq \mathbf{4})$
value $\neg(\tau_0 \models (\mathbf{4} \doteq \mathbf{10}\ ))$

lemma $\delta(null::('\mathfrak{A})Integer) = false$ ⟨*proof*⟩
lemma $\upsilon(null::('\mathfrak{A})Integer) = true$ ⟨*proof*⟩

3.5.4. More algebraic and logical layer on basic types

lemma [*simp*,*code-unfold*]:υ $\mathbf{0} = true$
⟨*proof*⟩

lemma [*simp*,*code-unfold*]:δ $\mathbf{1} = true$
⟨*proof*⟩

lemma [*simp*,*code-unfold*]:υ $\mathbf{1} = true$
⟨*proof*⟩

lemma [*simp*,*code-unfold*]:δ $\mathbf{2} = true$
⟨*proof*⟩

lemma [*simp*,*code-unfold*]:υ $\mathbf{2} = true$
⟨*proof*⟩

lemma [*simp*,*code-unfold*]: υ $\mathbf{6} = true$
⟨*proof*⟩

lemma [*simp*,*code-unfold*]: υ $\mathbf{8} = true$
⟨*proof*⟩

lemma [*simp*,*code-unfold*]: υ $\mathbf{9} = true$
⟨*proof*⟩

lemma *zero-non-null* [*simp*]: $(\mathbf{0} \doteq null) = false$
⟨*proof*⟩
lemma *null-non-zero* [*simp*]: $(null \doteq \mathbf{0}) = false$
⟨*proof*⟩

lemma *one-non-null* [*simp*]: $(\mathbf{1} \doteq null) = false$
⟨*proof*⟩
lemma *null-non-one* [*simp*]: $(null \doteq \mathbf{1}) = false$
⟨*proof*⟩

lemma *two-non-null* [*simp*]: $(\mathbf{2} \doteq null) = false$
⟨*proof*⟩
lemma *null-non-two* [*simp*]: $(null \doteq \mathbf{2}) = false$
⟨*proof*⟩

Here is a common case of a built-in operation on built-in types. Note that the arguments must be both defined (non-null, non-bot).

Note that we can not follow the lexis of standard OCL for Isabelle- technical reasons; these operators are heavily overloaded in the library that a further overloading would lead to heavy technical buzz in this document...

definition *ocl-add-int* $::('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Integer$ (**infix** $\oplus$ *40*)
where $x \oplus y \equiv \lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \land (\delta\ y)\ \tau = true\ \tau$
$\quad then\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil + \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\quad else\ invalid\ \tau$

definition *ocl-less-int* $::('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Boolean$ (**infix** $\prec$ *40*)
where $x \prec y \equiv \lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \land (\delta\ y)\ \tau = true\ \tau$
$\quad then\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil < \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\quad else\ invalid\ \tau$

definition *ocl-le-int* $::('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Boolean$ (**infix** $\preceq$ *40*)
where $x \preceq y \equiv \lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \land (\delta\ y)\ \tau = true\ \tau$
$\quad then\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil \leq \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\quad else\ invalid\ \tau$

Here follows a list of code-examples, that explain the meanings of the above definitions by compilation to code and execution to "True".

value $\tau_0 \models (\mathbf{9} \preceq \mathbf{10}\ )$
value $\tau_0 \models ((\ \mathbf{4} \oplus \mathbf{4}\ ) \preceq \mathbf{10}\ )$
value $\neg(\tau_0 \models ((\mathbf{4} \oplus(\ \mathbf{4} \oplus \mathbf{4}\ ))\prec \mathbf{10}\ ))$

3.6. Example for Complex Types: The Set-Collection Type

no-notation *None* ($\perp$)
notation *bot* ($\perp$)

3.6.1. The construction of the Set-Collection Type

For the semantic construction of the collection types, we have two goals:

1. we want the types to be *fully abstract*, i.e. the type should not contain junk-elements that are not representable by OCL expressions.
2. We want a possibility to nest collection types (so, we want the potential to talking about $Set(Set(Sequences(Pairs(X,Y))))$, and

The former principe rules out the option to define $'\alpha\ Set$ just by $('\mathfrak{A}, ('\alpha\ option\ option)\ set)\ val$. This would allow sets to contain junk elements such as $\{\perp\}$ which we need to identify with undefinedness itself. Abandoning fully abstractness of rules would later on produce all sorts of problems when quantifying over the elements of a type. However, if we build an own type, then it must conform to our abstract interface in order to have nested types: arguments of type-constructors must conform to our abstract interface, and the result type too.

The core of an own type construction is done via a type definition which provides the raw-type $'\alpha$ *Set-0*. it is shown that this type "fits" indeed into the abstract type interface discussed in the previous section.

typedef $'\alpha$ *Set-0* $=\{X::('\alpha::null)$ *set option option*.
$X = bot \vee X = null \vee (\forall x \in \lceil\lceil X \rceil\rceil.\ x \neq bot)\}$
⟨*proof*⟩

instantiation *Set-0* :: $(null)bot$
begin

definition *bot-Set-0-def*: $(bot::('a::null)\ Set\text{-}0) \equiv Abs\text{-}Set\text{-}0\ None$

instance ⟨*proof*⟩
end

instantiation *Set-0* :: $(null)null$
begin

definition *null-Set-0-def*: $(null::('a::null)\ Set\text{-}0) \equiv Abs\text{-}Set\text{-}0\ \lfloor\ None\ \rfloor$

instance ⟨*proof*⟩
end

... and lifting this type to the format of a valuation gives us:

type-synonym $('\mathfrak{A},'\alpha)\ Set = ('\mathfrak{A},\ '\alpha\ Set\text{-}0)\ val$

lemma *Set-inv-lemma*: $\tau \models (\delta\ X) \Longrightarrow (X\ \tau = Abs\text{-}Set\text{-}0\ \lfloor bot \rfloor)$
$\vee (\forall x \in \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil.\ x \neq bot)$
⟨*proof*⟩

lemma *invalid-set-not-defined* [*simp*,*code-unfold*]:$\delta(invalid::('\mathfrak{A},'\alpha::null)\ Set) = false$ ⟨*proof*⟩
lemma *null-set-not-defined* [*simp*,*code-unfold*]:$\delta(null::('\mathfrak{A},'\alpha::null)\ Set) = false$
⟨*proof*⟩
lemma *invalid-set-valid* [*simp*,*code-unfold*]:$\upsilon(invalid::('\mathfrak{A},'\alpha::null)\ Set) = false$
⟨*proof*⟩
lemma *null-set-valid* [*simp*,*code-unfold*]:$\upsilon(null::('\mathfrak{A},'\alpha::null)\ Set) = true$
⟨*proof*⟩

... which means that we can have a type $('\mathfrak{A},('\mathfrak{A},('\mathfrak{A})\ Integer)\ Set)\ Set$ corresponding exactly to Set(Set(Integer)) in OCL notation. Note that the parameter $\mathfrak{A}$ still refers to the object universe; making the OCL semantics entirely parametric in the object universe makes it possible to study (and prove) its properties independently from a concrete class diagram.

3.6.2. Constants on Sets

definition $mtSet::('\mathfrak{A},'\alpha::null)\ Set\ \ (Set\{\})$
where $Set\{\} \equiv (\lambda\ \tau.\ \ Abs\text{-}Set\text{-}0\ \lfloor\lfloor\{\}::'\alpha\ set\rfloor\rfloor\ )$

lemma *mtSet-defined*[*simp*,*code-unfold*]:$\delta(Set\{\}) = true$
⟨*proof*⟩

lemma *mtSet-valid*[*simp*,*code-unfold*]:$\upsilon(Set\{\}) = true$
⟨*proof*⟩

Note that the collection types in OCL allow for null to be included; however, there is the null-collection into which inclusion yields invalid.

3.6.3. Strict Equality on Sets

This section of foundational operations on sets is closed with a paragraph on equality. Strong Equality is inherited from the OCL core, but we have to consider the case of the strict equality. We decide to overload strict equality in the same way we do for other value's in OCL:

defs *StrictRefEq-set* :
$$(x::('\mathfrak{A},'\alpha::null)Set) \doteq y \equiv \lambda\ \tau.\ if\ (\upsilon\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau\ then\ (x \triangleq y)\tau\ else\ invalid\ \tau$$

lemma *RefEq-set-refl*[*simp*,*code-unfold*]:
$((x::('\mathfrak{A},'\alpha::null)Set) \doteq x) = (if\ (\upsilon\ x)\ then\ true\ else\ invalid\ endif)$
⟨*proof*⟩

lemma *StrictRefEq-set-strict1*: $((x::('\mathfrak{A},'\alpha::null)Set) \doteq invalid) = invalid$
⟨*proof*⟩

lemma *StrictRefEq-set-strict2*: $(invalid \doteq (y::('\mathfrak{A},'\alpha::null)Set)) = invalid$
⟨*proof*⟩

lemma *StrictRefEq-set-strictEq-valid-args-valid*:
$(\tau \models \delta\ ((x::('\mathfrak{A},'\alpha::null)Set) \doteq y)) = ((\tau \models (\upsilon\ x)) \wedge (\tau \models \upsilon\ y))$
⟨*proof*⟩

lemma *cp-StrictRefEq-set*:$((X::('\mathfrak{A},'\alpha::null)Set) \doteq Y)\ \tau = ((\lambda\text{-}.\ X\ \tau) \doteq (\lambda\text{-}.\ Y\ \tau))\ \tau$
⟨*proof*⟩

lemma *strictRefEq-set-vs-strongEq*:
$\tau \models \upsilon\ x \Longrightarrow \tau \models \upsilon\ y \Longrightarrow (\tau \models (((x::('\mathfrak{A},'\alpha::null)Set) \doteq y) \triangleq (x \triangleq y)))$
⟨*proof*⟩

3.6.4. Algebraic Properties on Strict Equality on Sets

One might object here that for the case of objects, this is an empty definition. The answer is no, we will restrain later on states and objects such that any object has its id stored inside the object (so the ref, under which an object can be referenced in the store will represented in the object itself). For such well-formed stores that satisfy this invariant (the WFF - invariant), the referential equality and the strong equality — and therefore the strict equality on sets in the sense above) coincides.

To become operational, we derive:

lemma *StrictRefEq-set-refl* :
$((x::('\mathfrak{A},'\alpha::null)Set) \doteq x) = (if\ (\upsilon\ x)\ then\ true\ else\ invalid\ endif)$
$\langle proof \rangle$

The key for an operational definition if OclForall given below.

The case of the size definition is somewhat special, we admit explicitly in Essential OCL the possibility of infinite sets. For the size definition, this requires an extra condition that assures that the cardinality of the set is actually a defined integer.

3.6.5. Library Operations on Sets

definition *OclSize* $:: ('\mathfrak{A},'\alpha::null)Set \Rightarrow '\mathfrak{A}\ Integer$
where $OclSize\ x = (\lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge finite(\lceil\lceil Rep\text{-}Set\text{-}0\ (x\ \tau)\rceil\rceil)$
$then\ \lfloor\lfloor\ int(card\ \lceil\lceil Rep\text{-}Set\text{-}0\ (x\ \tau)\rceil\rceil)\ \rfloor\rfloor$
$else\ \bot\ )$

definition *OclIncluding* $:: [('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ val] \Rightarrow ('\mathfrak{A},'\alpha)\ Set$
where $OclIncluding\ x\ y = (\lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau$
$then\ Abs\text{-}Set\text{-}0\ \lfloor\lfloor\ \lceil\lceil Rep\text{-}Set\text{-}0\ (x\ \tau)\rceil\rceil\ \cup \{y\ \tau\}\ \rfloor\rfloor$
$else\ \bot\ )$

definition *OclIncludes* $:: [('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ val] \Rightarrow '\mathfrak{A}\ Boolean$
where $OclIncludes\ x\ y = (\lambda\ \tau.\ \ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau$
$then\ \lfloor\lfloor(y\ \tau) \in \lceil\lceil Rep\text{-}Set\text{-}0\ (x\ \tau)\rceil\rceil\ \rfloor\rfloor$
$else\ \bot\ \ )$

definition *OclExcluding* $:: [('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ val] \Rightarrow ('\mathfrak{A},'\alpha)\ Set$
where $OclExcluding\ x\ y = (\lambda\ \tau.\ \ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau$
$then\ Abs\text{-}Set\text{-}0\ \lfloor\lfloor\ \lceil\lceil Rep\text{-}Set\text{-}0\ (x\ \tau)\rceil\rceil - \{y\ \tau\}\ \rfloor\rfloor$
$else\ \bot\ )$

definition *OclExcludes* $:: [('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ val] \Rightarrow '\mathfrak{A}\ Boolean$
where $OclExcludes\ x\ y = (not(OclIncludes\ x\ y))$

The following definition follows the requirement of the standard to treat null as neutral element of sets. It is a well-documented exception from the general strictness rule and the rule that the distinguished argument self should be non-null.

definition *OclIsEmpty* $:: ('\mathfrak{A},'\alpha::null)\ Set \Rightarrow '\mathfrak{A}\ Boolean$
where $OclIsEmpty\ x = \ ((x \doteq null)\ or\ ((OclSize\ x) \doteq \mathbf{0}))$

definition *OclNotEmpty* $:: ('\mathfrak{A},'\alpha::null)\ Set \Rightarrow '\mathfrak{A}\ Boolean$
where $OclNotEmpty\ x = \ not(OclIsEmpty\ x)$

definition *OclForall* $:: [('\mathfrak{A},'\alpha::null)Set,('\mathfrak{A},'\alpha)val \Rightarrow ('\mathfrak{A})Boolean] \Rightarrow '\mathfrak{A}\ Boolean$
where $OclForall\ S\ P = (\lambda\ \tau.\ if\ (\delta\ S)\ \tau = true\ \tau$
$then\ if\ (\forall x \in \lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil.\ P\ (\lambda\ \text{-}.\ x)\ \tau = true\ \tau)$
$then\ true\ \tau$
$else\ if\ (\forall x \in \lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil.\ P(\lambda\ \text{-}.\ x)\ \tau = true\ \tau \vee$
$P(\lambda\ \text{-}.\ x)\ \tau = false\ \tau)$
$then\ false\ \tau$
$else\ \bot$
$else\ \bot)$

definition *OclExists* $:: [('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)val \Rightarrow ('\mathfrak{A})Boolean] \Rightarrow '\mathfrak{A}\ Boolean$
where $OclExists\ S\ P = not(OclForall\ S\ (\lambda\ X.\ not\ (P\ X)))$

syntax
$\text{-}OclForall :: [('\mathfrak{A},'\alpha::null)\ Set,id,('\mathfrak{A})Boolean] \Rightarrow '\mathfrak{A}\ Boolean \quad ((\text{-})\text{−}{>}forall'(\text{-}|\text{-}'))$
translations
$X\text{−}{>}forall(x \mid P) == CONST\ OclForall\ X\ (\%x.\ P)$

syntax
$\text{-}OclExist :: [('\mathfrak{A},'\alpha::null)\ Set,id,('\mathfrak{A})Boolean] \Rightarrow '\mathfrak{A}\ Boolean \quad ((\text{-})\text{−}{>}exists'(\text{-}|\text{-}'))$
translations
$X\text{−}{>}exists(x \mid P) == CONST\ OclExists\ X\ (\%x.\ P)$

consts

$OclUnion :: [('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow ('\mathfrak{A},'\alpha)\ Set$
$OclIntersection :: [('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow ('\mathfrak{A},'\alpha)\ Set$
$OclIncludesAll :: [('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow '\mathfrak{A}\ Boolean$
$OclExcludesAll :: [('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow '\mathfrak{A}\ Boolean$
$OclComplement :: ('\mathfrak{A},'\alpha::null)\ Set \Rightarrow ('\mathfrak{A},'\alpha)\ Set$
$OclSum :: ('\mathfrak{A},'\alpha::null)\ Set \Rightarrow '\mathfrak{A}\ Integer$
$OclCount :: [('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow '\mathfrak{A}\ Integer$

notation
  OclSize $(\text{-}{-}{>}size'(')\ [66])$
and
  OclCount $(\text{-}{-}{>}count'(\text{-}')\ [66,65]65)$
and
  OclIncludes $(\text{-}{-}{>}includes'(\text{-}')\ [66,65]65)$
and
  OclExcludes $(\text{-}{-}{>}excludes'(\text{-}')\ [66,65]65)$
and
  OclSum $(\text{-}{-}{>}sum'(')\ [66])$
and
  OclIncludesAll $(\text{-}{-}{>}includesAll'(\text{-}')\ [66,65]65)$
and
  OclExcludesAll $(\text{-}{-}{>}excludesAll'(\text{-}')\ [66,65]65)$
and
  OclIsEmpty $(\text{-}{-}{>}isEmpty'(')\ [66])$
and
  OclNotEmpty $(\text{-}{-}{>}notEmpty'(')\ [66])$
and
  OclIncluding $(\text{-}{-}{>}including'(\text{-}'))$
and
  OclExcluding $(\text{-}{-}{>}excluding'(\text{-}'))$
and
  OclComplement $(\text{-}{-}{>}complement'('))$
and
  OclUnion $(\text{-}{-}{>}union'(\text{-}')\quad [66,65]65)$
and
  OclIntersection$(\text{-}{-}{>}intersection'(\text{-}')\quad [71,70]70)$

lemma *cp-OclIncluding*:
$(X{-}{>}including(x))\ \tau = ((\lambda\ \text{-}.\ X\ \tau){-}{>}including(\lambda\ \text{-}.\ x\ \tau))\ \tau$
⟨*proof*⟩

lemma *cp-OclExcluding*:
$(X{-}{>}excluding(x))\ \tau = ((\lambda\ \text{-}.\ X\ \tau){-}{>}excluding(\lambda\ \text{-}.\ x\ \tau))\ \tau$
⟨*proof*⟩

lemma *cp-OclIncludes*:
$(X{-}{>}includes(x))\ \tau = (OclIncludes\ (\lambda\ \text{-}.\ X\ \tau)\ (\lambda\ \text{-}.\ x\ \tau)\ \tau)$
⟨*proof*⟩

3.6.6. Logic and Algebraic Layer on Set Operations

lemma *including-strict1*[*simp*,*code-unfold*]:$(invalid{-}{>}including(x)) = invalid$
⟨*proof*⟩

lemma *including-strict2*[*simp*,*code-unfold*]:$(X{-}{>}including(invalid)) = invalid$
⟨*proof*⟩

lemma *including-strict3*[*simp*,*code-unfold*]:$(null{-}{>}including(x)) = invalid$
⟨*proof*⟩

lemma *excluding-strict1*[*simp*,*code-unfold*]:$(invalid{-}{>}excluding(x)) = invalid$
⟨*proof*⟩

lemma *excluding-strict2*[*simp*,*code-unfold*]:$(X{-}{>}excluding(invalid)) = invalid$
⟨*proof*⟩

lemma *excluding-strict3*[*simp*,*code-unfold*]:$(null{-}{>}excluding(x)) = invalid$
⟨*proof*⟩

lemma *includes-strict1*[*simp*,*code-unfold*]:$(invalid{-}{>}includes(x)) = invalid$
⟨*proof*⟩

lemma *includes-strict2*[*simp*,*code-unfold*]:$(X{-}{>}includes(invalid)) = invalid$
⟨*proof*⟩

lemma *includes-strict3*[*simp*,*code-unfold*]:$(null{-}{>}includes(x)) = invalid$
⟨*proof*⟩

lemma *including-defined-args-valid*:
$(\tau \models \delta(X{-}{>}including(x))) = ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
⟨*proof*⟩

lemma *including-valid-args-valid*:
$(\tau \models \upsilon(X{-}{>}including(x))) = ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
⟨*proof*⟩

lemma *including-defined-args-valid'*[*simp*,*code-unfold*]:
$\delta(X{-}{>}including(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
⟨*proof*⟩

lemma *including-valid-args-valid''*[*simp*,*code-unfold*]:
$\upsilon(X{-}{>}including(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
⟨*proof*⟩

lemma *excluding-defined-args-valid*:
$(\tau \models \delta(X{-}{>}excluding(x))) = ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$

$\langle proof \rangle$

lemma *excluding-valid-args-valid*:
$(\tau \models \upsilon(X{-}{>}excluding(x))) = ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
$\langle proof \rangle$

lemma *excluding-valid-args-valid'*[*simp*,*code-unfold*]:
$\delta(X{-}{>}excluding(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
$\langle proof \rangle$

lemma *excluding-valid-args-valid''*[*simp*,*code-unfold*]:
$\upsilon(X{-}{>}excluding(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
$\langle proof \rangle$

lemma *includes-defined-args-valid*:
$(\tau \models \delta(X{-}{>}includes(x))) = ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
$\langle proof \rangle$

lemma *includes-valid-args-valid*:
$(\tau \models \upsilon(X{-}{>}includes(x))) = ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
$\langle proof \rangle$

lemma *includes-valid-args-valid'*[*simp*,*code-unfold*]:
$\delta(X{-}{>}includes(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
$\langle proof \rangle$

lemma *includes-valid-args-valid''*[*simp*,*code-unfold*]:
$\upsilon(X{-}{>}includes(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
$\langle proof \rangle$

Some computational laws:

lemma *including-charn0*[*simp*]:
assumes *val-x*:$\tau \models (\upsilon\ x)$
shows $\tau \models not(Set\{\}{-}{>}includes(x))$
$\langle proof \rangle$

lemma *including-charn0'*[*simp*,*code-unfold*]:
$Set\{\}{-}{>}includes(x) = (if\ \upsilon\ x\ then\ false\ else\ invalid\ endif)$
$\langle proof \rangle$

lemma *including-charn1*:

assumes *def-X*:$\tau \models (\delta\ X)$
assumes *val-x*:$\tau \models (\upsilon\ x)$
shows $\tau \models (X{-}{>}including(x){-}{>}includes(x))$
$\langle proof \rangle$

lemma *including-charn2*:
assumes *def-X*:$\tau \models (\delta\ X)$
and *val-x*:$\tau \models (\upsilon\ x)$
and *val-y*:$\tau \models (\upsilon\ y)$
and *neq* :$\tau \models not(x \triangleq y)$
shows $\tau \models (X{-}{>}including(x){-}{>}includes(y)) \triangleq (X{-}{>}includes(y))$
$\langle proof \rangle$

One would like a generic theorem of the form:

```
lemma includes_execute[code_unfold]:
"(X->including(x)->includes(y)) = (if \<delta> X then if x \<doteq> y
                                                   then true
                                                   else X->includes(y)
                                                   endif
                                              else invalid endif)"
```

Unfortunately, this does not hold in general, since referential equality is an overloaded concept and has to be defined for each type individually. Consequently, it is only valid for concrete type instances for Boolean, Integer, and Sets thereof...

The computational law `includes_execute` becomes generic since it uses strict equality which in itself is generic. It is possible to prove the following generic theorem and instantiate it if a number of properties that link the polymorphic logical, Strong Equality with the concrete instance of strict quality.

lemma *includes-execute-generic*:
assumes *strict1*: $(x \doteq invalid) = invalid$
and *strict2*: $(invalid \doteq y) = invalid$
and *strictEq-valid-args-valid*: $\bigwedge (x::('\mathfrak{A},'a::null)val)\ y\ \tau.$
$(\tau \models \delta\ (x \doteq y)) = ((\tau \models (\upsilon\ x)) \wedge (\tau \models \upsilon\ y))$
and *cp-StrictRefEq*: $\bigwedge (X::('\mathfrak{A},'a::null)val)\ Y\ \tau.\ (X \doteq Y)\ \tau = ((\lambda\text{-}.\ X\ \tau) \doteq (\lambda\text{-}.\ Y\ \tau))\ \tau$
and *strictEq-vs-strongEq*: $\bigwedge (x::('\mathfrak{A},'a::null)val)\ y\ \tau.$
$\tau \models \upsilon\ x \Longrightarrow \tau \models \upsilon\ y \Longrightarrow (\tau \models ((x \doteq y) \triangleq (x \triangleq y)))$
shows
$(X{-}{>}including(x::('\mathfrak{A},'a::null)val){-}{>}includes(y)) =$
$(if\ \delta\ X\ then\ if\ x \doteq y\ then\ true\ else\ X{-}{>}includes(y)\ endif\ else\ invalid\ endif)$
$\langle proof \rangle$

schematic-lemma *includes-execute-int*[*code-unfold*]: *?X*

⟨*proof*⟩

schematic-lemma *includes-execute-bool*[*code-unfold*]: *?X*
⟨*proof*⟩

schematic-lemma *includes-execute-set*[*code-unfold*]: *?X*
⟨*proof*⟩

lemma *excluding-charn0*[*simp*]:
assumes *val-x*:$\tau \models (\upsilon\ x)$
shows $\tau \models ((Set\{\}{-}{>}excluding(x))\ \triangleq\ Set\{\})$
⟨*proof*⟩

lemma *excluding-charn0-exec*[*code-unfold*]:
$(Set\{\}{-}{>}excluding(x)) = (if\ (\upsilon\ x)\ then\ Set\{\}\ else\ invalid\ endif)$
⟨*proof*⟩

lemma *excluding-charn1*:
assumes *def-X*:$\tau \models (\delta\ X)$
and *val-x*:$\tau \models (\upsilon\ x)$
and *val-y*:$\tau \models (\upsilon\ y)$
and *neq* :$\tau \models not(x \triangleq y)$
shows $\tau \models ((X{-}{>}including(x)){-}{>}excluding(y)) \triangleq ((X{-}{>}excluding(y)){-}{>}including(x))$
⟨*proof*⟩

lemma *excluding-charn2*:
assumes *def-X*:$\tau \models (\delta\ X)$
and *val-x*:$\tau \models (\upsilon\ x)$
shows $\tau \models (((X{-}{>}including(x)){-}{>}excluding(x)) \triangleq (X{-}{>}excluding(x)))$
⟨*proof*⟩

lemma *excluding-charn-exec*[*code-unfold*]:
$(X{-}{>}including(x){-}{>}excluding(y)) = (if\ \delta\ X\ then\ if\ x \doteq y$
$then\ X{-}{>}excluding(y)$
$else\ X{-}{>}excluding(y){-}{>}including(x)$
$endif$
$else\ invalid\ endif)$
⟨*proof*⟩

syntax
-OclFinset :: *args* => $('\mathfrak{A},'a{::}null)$ *Set* (*Set*{(-)})
translations
Set{*x*, *xs*} == *CONST OclIncluding* (*Set*{*xs*}) *x*

$Set\{x\}$ $\quad == CONST\ OclIncluding\ (Set\{\})\ x$

lemma *syntax-test*: $Set\{\mathbf{2},\mathbf{1}\} = (Set\{\}{-}{>}including(\mathbf{1}){-}{>}including(\mathbf{2}))$
⟨*proof*⟩

lemma *set-test1*: $\tau \models (Set\{\mathbf{2},null\}{-}{>}includes(null))$
⟨*proof*⟩

lemma *set-test2*: $\neg(\tau \models (Set\{\mathbf{2},\mathbf{1}\}{-}{>}includes(null)))$
⟨*proof*⟩

Here is an example of a nested collection. Note that we have to use the abstract null (since we did not (yet) define a concrete constant *null* for the non-existing Sets) :

lemma *semantic-test2*:
assumes $H{:}(Set\{\mathbf{2}\} \doteq null) = (false{::}('\mathfrak{A})Boolean)$
shows $(\tau{::}('\mathfrak{A})st) \models (Set\{Set\{\mathbf{2}\},null\}{-}{>}includes(null))$
⟨*proof*⟩

lemma *semantic-test3*: $\tau \models (Set\{null,\mathbf{2}\}{-}{>}includes(null))$
⟨*proof*⟩

lemma *StrictRefEq-set-exec*[*simp*,*code-unfold*] :
$((x{::}('\mathfrak{A},'\alpha{::}null)Set) \doteq y) =$
$\quad (if\ \delta\ x\ then\ (if\ \delta\ y$
$\qquad\qquad then\ ((x{-}{>}forall(z|\ y{-}{>}includes(z))\ and\ (y{-}{>}forall(z|\ x{-}{>}includes(z)))))$
$\qquad\qquad else\ if\ \upsilon\ y$
$\qquad\qquad\quad then\ false\ (*\ x'{-}{>}includes = null\ *)$
$\qquad\qquad\quad else\ invalid$
$\qquad\qquad\quad endif$
$\qquad\qquad endif)$
$\qquad else\ if\ \upsilon\ x\ (*\ null = ???\ *)$
$\qquad\quad then\ if\ \upsilon\ y\ then\ not(\delta\ y)\ else\ invalid\ endif$
$\qquad\quad else\ invalid$
$\qquad\quad endif$
$\qquad endif)$
⟨*proof*⟩

lemma *forall-set-null-exec*[*simp*,*code-unfold*] :
$(null{-}{>}forall(z|\ P(z))) = invalid$

⟨*proof*⟩

lemma *forall-set-mt-exec*[*simp*,*code-unfold*] :
$((Set\{\})\text{−>}forall(z|\ P(z))) = true$
⟨*proof*⟩

lemma *exists-set-null-exec*[*simp*,*code-unfold*] :
$(null\text{−>}exists(z \mid P(z))) = invalid$
⟨*proof*⟩

lemma *exists-set-mt-exec*[*simp*,*code-unfold*] :
$((Set\{\})\text{−>}exists(z \mid P(z))) = false$
⟨*proof*⟩

lemma *forall-set-including-exec*[*simp*,*code-unfold*] :
$((S\text{−>}including(x))\text{−>}forall(z \mid P(z))) = (if\ (\delta\ S)\ and\ (\upsilon\ x)$
$then\ P(x)\ and\ S\text{−>}forall(z \mid P(z))$
$else\ invalid$
$endif)$
⟨*proof*⟩

lemma *not-if*[*simp*]:
$not(if\ P\ then\ C\ else\ E\ endif) = (if\ P\ then\ not\ C\ else\ not\ E\ endif)$
⟨*proof*⟩

lemma *exists-set-including-exec*[*simp*,*code-unfold*] :
$((S\text{−>}including(x))\text{−>}exists(z \mid P(z))) = (if\ (\delta\ S)\ and\ (\upsilon\ x)$
$then\ P(x)\ or\ S\text{−>}exists(z \mid P(z))$
$else\ invalid$
$endif)$
⟨*proof*⟩

lemma *set-test4* : $\tau \models (Set\{\mathbf{2},null,\mathbf{2}\} \doteq Set\{null,\mathbf{2}\})$
⟨*proof*⟩

definition $OclIterate_{Set}$ :: $[('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\beta::null)val,$
$('\mathfrak{A},'\alpha)val \Rightarrow ('\mathfrak{A},'\beta)val \Rightarrow ('\mathfrak{A},'\beta)val] \Rightarrow ('\mathfrak{A},'\beta)val$
where $OclIterate_{Set}\ S\ A\ F = (\lambda\ \tau.\ if\ (\delta\ S)\ \tau = true\ \tau \wedge (\upsilon\ A)\ \tau = true\ \tau \wedge finite\lceil\lceil Rep\text{-}Set\text{-}0$
$(S\ \tau)\rceil\rceil$
$then\ (Finite\text{-}Set.fold\ (F)\ (A)\ ((\lambda a\ \tau.\ a)\ `\ \lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil))\tau$
$else\ \bot)$

syntax
-OclIterate :: $[('\mathfrak{A},'\alpha::null)\ Set,\ idt,\ idt,\ '\alpha,\ '\beta] => ('\mathfrak{A},'\gamma)val$
(- −>*iterate*′(-;-=- | -′) [*71*,*100*,*70*]*50*)

translations
$X{-}{>}iterate(a;\ x = A \mid P) == CONST\ OclIterate_{Set}\ X\ A\ (\%a.\ (\%\ x.\ P))$

lemma $OclIterate_{Set}$*-strict1*[*simp*]:$invalid{-}{>}iterate(a;\ x = A \mid P\ a\ x) = invalid$
⟨*proof*⟩

lemma $OclIterate_{Set}$*-null1*[*simp*]:$null{-}{>}iterate(a;\ x = A \mid P\ a\ x) = invalid$
⟨*proof*⟩

lemma $OclIterate_{Set}$*-strict2*[*simp*]:$S{-}{>}iterate(a;\ x = invalid \mid P\ a\ x) = invalid$
⟨*proof*⟩

An open question is this ...

lemma $OclIterate_{Set}$*-null2*[*simp*]:$S{-}{>}iterate(a;\ x = null \mid P\ a\ x) = invalid$
⟨*proof*⟩

In the definition above, this does not hold in general. And I believe, this is how it should be ...

lemma $OclIterate_{Set}$*-infinite*:
assumes *non-finite*: $\tau \models not(\delta(S{-}{>}size()))$
shows $(OclIterate_{Set}\ S\ A\ F)\ \tau = invalid\ \tau$
⟨*proof*⟩

lemma $OclIterate_{Set}$*-empty*[*simp*]: $((Set\{\}){-}{>}iterate(a;\ x = A \mid P\ a\ x)) = A$
⟨*proof*⟩

In particular, this does hold for A = null.

lemma $OclIterate_{Set}$*-including*:
assumes *S-finite*: $\tau \models \delta(S{-}{>}size())$

shows $((S{-}{>}including(a)){-}{>}iterate(a;\ x = A \mid F\ a\ x))\ \tau =$
$(\ ((S{-}{>}excluding(a)){-}{>}iterate(a;\ x = F\ a\ A \mid F\ a\ x)))\ \tau$
⟨*proof*⟩

lemma [*simp*]: $\delta\ (Set\{\}\ {-}{>}size()) = true$
⟨*proof*⟩

lemma [*simp*]: $\delta\ ((X\ {-}{>}including(x))\ {-}{>}size()) = (\delta(X)\ and\ \upsilon(x))$
⟨*proof*⟩

3.6.7. Test Statements

lemma *short-cut'*[*simp*]: $(\mathbf{8} \doteq \mathbf{6}) = false$
⟨*proof*⟩

lemma *GogollasChallenge-on-sets*:
$(Set\{\ \mathbf{6},\mathbf{8}\ \}\ {-}{>}iterate(i;r1{=}Set\{\mathbf{9}\}|$
$r1{-}{>}iterate(j;r2{=}r1|$
$r2{-}{>}including(\mathbf{0}){-}{>}including(i){-}{>}including(j))) =\ Set\{\mathbf{0},\ \ \mathbf{6},\ \mathbf{9}\})$
$\langle proof\rangle$

Elementary computations on Sets.

value $\neg\ (\tau_0 \models \upsilon(invalid{::}({'}\mathfrak{A},{'}\alpha{::}null)\ Set))$
value $\quad \tau_0 \models \upsilon(null{::}({'}\mathfrak{A},{'}\alpha{::}null)\ Set)$
value $\neg\ (\tau_0 \models \delta(null{::}({'}\mathfrak{A},{'}\alpha{::}null)\ Set))$
value $\quad \tau_0 \models \upsilon(Set\{\})$
value $\quad \tau_0 \models \upsilon(Set\{Set\{\mathbf{2}\},null\})$
value $\quad \tau_0 \models \delta(Set\{Set\{\mathbf{2}\},null\})$
value $\quad \tau_0 \models (Set\{\mathbf{2},\mathbf{1}\}{-}{>}includes(\mathbf{1}))$
value $\neg\ (\tau_0 \models (Set\{\mathbf{2}\}{-}{>}includes(\mathbf{1})))$
value $\neg\ (\tau_0 \models (Set\{\mathbf{2},\mathbf{1}\}{-}{>}includes(null)))$
value $\quad \tau_0 \models (Set\{\mathbf{2},null\}{-}{>}includes(null))$
value $\quad \tau \models ((Set\{\mathbf{2},\mathbf{1}\}){-}{>}forall(z \mid \mathbf{0} \prec z))$
value $\neg\ (\tau \models ((Set\{\mathbf{2},\mathbf{1}\}){-}{>}exists(z \mid z \prec \mathbf{0}\ )))$

value $\neg\ (\tau \models ((Set\{\mathbf{2},null\}){-}{>}forall(z \mid \mathbf{0} \prec z)))$
value $\quad \tau \models ((Set\{\mathbf{2},null\}){-}{>}exists(z \mid \mathbf{0} \prec z))$

value $\quad \tau \models (Set\{\mathbf{2},null,\mathbf{2}\} \doteq Set\{null,\mathbf{2}\})$
value $\quad \tau \models (Set\{\mathbf{1},null,\mathbf{2}\} <> Set\{null,\mathbf{2}\})$

value $\quad \tau \models (Set\{Set\{\mathbf{2},null\}\} \doteq Set\{Set\{null,\mathbf{2}\}\})$
value $\quad \tau \models (Set\{Set\{\mathbf{2},null\}\} <> Set\{Set\{null,\mathbf{2}\},null\})$

end

4. Part II: State Operations and Objects

theory *OCL-state*
imports *OCL-lib*
begin

4.0.8. Recall: The generic structure of States

Next we will introduce the foundational concept of an object id (oid), which is just some infinite set.

type-synonym $oid = nat$

States are pair of a partial map from oid's to elements of an object universe $'\mathfrak{A}$ — the heap — and a map to relations of objects. The relations were encoded as lists of pairs in order to leave the possibility to have Bags, OrderedSets or Sequences as association ends.

Recall:

```
record ('\<AA>)state =
        heap   :: "oid ⇀ '\<AA> "
        assocs :: "oid  ⇀ (oid  × oid)  list "
```

type-synonym $('\mathfrak{A})st = {'\mathfrak{A}}\ state \times {'\mathfrak{A}}\ state$

Now we refine our state-interface. In certain contexts, we will require that the elements of the object universe have a particular structure; more precisely, we will require that there is a function that reconstructs the oid of an object in the state (we will settle the question how to define this function later).

class *object* = **fixes** $oid\text{-}of :: {'a} \Rightarrow oid$

Thus, if needed, we can constrain the object universe to objects by adding the following type class constraint:

typ $'\mathfrak{A} :: object$

4.0.9. Referential Object Equality in States

Generic referential equality - to be used for instantiations with concrete object types ...

definition $gen\text{-}ref\text{-}eq :: ('\mathfrak{A},'a{::}\{object,null\})val \Rightarrow ('\mathfrak{A},'a)val \Rightarrow ('\mathfrak{A})Boolean$
where $gen\text{-}ref\text{-}eq\ x\ y$
$\equiv \lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$then\ if\ x\ \tau = null \vee y\ \tau = null$

$then\ \lfloor\lfloor x\ \tau = null \wedge y\ \tau = null\rfloor\rfloor$
$else\ \lfloor\lfloor(oid\text{-}of\ (x\ \tau)) = (oid\text{-}of\ (y\ \tau))\ \rfloor\rfloor$
$else\ invalid\ \tau$

lemma *gen-ref-eq-object-strict1*[*simp*] :
$(gen\text{-}ref\text{-}eq\ x\ invalid) = invalid$
⟨*proof*⟩

lemma *gen-ref-eq-object-strict2*[*simp*] :
$(gen\text{-}ref\text{-}eq\ invalid\ x) = invalid$
⟨*proof*⟩

lemma *gen-ref-eq-object-strict3*[*simp*] :
$(gen\text{-}ref\text{-}eq\ x\ null) = invalid$
⟨*proof*⟩

lemma *gen-ref-eq-object-strict4*[*simp*] :
$(gen\text{-}ref\text{-}eq\ null\ x) = invalid$
⟨*proof*⟩

lemma *cp-gen-ref-eq-object*:
$(gen\text{-}ref\text{-}eq\ x\ y\ \tau) = (gen\text{-}ref\text{-}eq\ (\lambda\text{-}.\ x\ \tau)\ (\lambda\text{-}.\ y\ \tau))\ \tau$
⟨*proof*⟩

lemmas *cp-intro*[*simp*,*intro*!] =
OCL-core.cp-intro
cp-gen-ref-eq-object[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]],
of gen-ref-eq]]

Finally, we derive the usual laws on definedness for (generic) object equality:

lemma *gen-ref-eq-defargs*:
$\tau \models (gen\text{-}ref\text{-}eq\ x\ (y{::}('\mathfrak{A},'a{::}\{null,object\})val)) \Longrightarrow (\tau \models (\delta\ x)) \wedge (\tau \models (\delta\ y))$
⟨*proof*⟩

4.0.10. Further requirements on States

A key-concept for linking strict referential equality to logical equality: in well-formed states (i.e. those states where the self (oid-of) field contains the pointer to which the object is associated to in the state), referential equality coincides with logical equality.

definition $WFF :: ('\mathfrak{A}{::}object)st \Rightarrow bool$
where $WFF\ \tau = ((\forall\ x \in ran(heap(fst\ \tau)).\ \lceil heap(fst\ \tau)\ (oid\text{-}of\ x)\rceil = x) \wedge$
$(\forall\ x \in ran(heap(snd\ \tau)).\ \lceil heap(snd\ \tau)\ (oid\text{-}of\ x)\rceil = x))$

This is a generic definition of referential equality: Equality on objects in a state is reduced to equality on the references to these objects. As in HOL-OCL, we will store the reference of an object inside the object in a (ghost) field. By establishing certain invariants ("consistent state"), it can be assured that there is a "one-to-one-correspondance" of objects to their references — and therefore the definition below behaves as we expect.

Generic Referential Equality enjoys the usual properties: (quasi) reflexivity, symmetry, transitivity, substitutivity for defined values. For type-technical reasons, for each concrete object type, the equality $\doteq$ is defined by generic referential equality.

theorem *strictEqGen-vs-strongEq*:
$WFF\ \tau \Longrightarrow \tau \models(\delta\ x) \Longrightarrow \tau \models(\delta\ y) \Longrightarrow$
$(x\ \tau \in ran\ (heap(fst\ \tau)) \wedge y\ \tau \in ran\ (heap(fst\ \tau))) \wedge$
$(x\ \tau \in ran\ (heap(snd\ \tau)) \wedge y\ \tau \in ran\ (heap(snd\ \tau))) \Longrightarrow$ (* *x and y must be object representations that exist in either the pre or post state* *)
$\quad (\tau \models (gen\text{-}ref\text{-}eq\ x\ y)) = (\tau \models (x \triangleq y))$
⟨*proof*⟩

So, if two object descriptions live in the same state (both pre or post), the referential equality on objects implies in a WFF state the logical equality. Uffz.

4.1. Miscillaneous: Initial States (for Testing and Code Generation)

definition τ_0 :: $('\mathfrak{A})st$
where $\quad \tau_0 \equiv ((\!|heap{=}Map.empty,\ assocs{=}\ Map.empty|\!),$
$\quad (\!|heap{=}Map.empty,\ assocs{=}\ Map.empty|\!))$

4.1.1. Generic Operations on States

In order to denote OCL-types occuring in OCL expressions syntactically — as, for example, as "argument" of allInstances — we use the inverses of the injection functions into the object universes; we show that this is sufficient "characterization".

definition *allinstances* :: $('\mathfrak{A} \Rightarrow '\alpha) \Rightarrow ('\mathfrak{A}{::}object,'\alpha\ option\ option)\ Set$
$\quad$ (- .oclAllInstances'(')))
where $((H).oclAllInstances())\ \tau =$
$\quad Abs\text{-}Set\text{-}0\ \lfloor\lfloor(Some\ o\ Some\ o\ H)\ `\ (ran(heap(snd\ \tau)) \cap \{x.\ \exists\ y.\ y{=}H\ x\})\ \rfloor\rfloor$

definition *allinstancesATpre* :: $('\mathfrak{A} \Rightarrow '\alpha) \Rightarrow ('\mathfrak{A}{::}object,'\alpha\ option\ option)\ Set$
$\quad$ (- .oclAllInstances@pre'(')))
where $((H).oclAllInstances@pre())\ \tau =$
$\quad Abs\text{-}Set\text{-}0\ \lfloor\lfloor(Some\ o\ Some\ o\ H)\ `\ (ran(heap(fst\ \tau)) \cap \{x.\ \exists\ y.\ y{=}H\ x\})\ \rfloor\rfloor$

lemma $\tau_0 \models H\ .oclAllInstances() \triangleq Set\{\}$
⟨*proof*⟩

lemma $\tau_0 \models H\ .oclAllInstances@pre() \triangleq Set\{\}$
⟨*proof*⟩

theorem *state-update-vs-allInstances*:

assumes $oid \notin dom\ \sigma'$
and $cp\ P$
shows $((\sigma, (\!|heap{=}\sigma'(oid \mapsto Object),\ assocs{=}A|\!)) \models (P(Type\ .oclAllInstances()))) =$
$((\sigma, (\!|heap{=}\sigma',\ assocs{=}A|\!)) \models (P((Type\ .oclAllInstances()){-}{>}including(\lambda\ \text{-}.\ Some(Some((the\text{-}inv\ Type)\ Object))))))$
$\langle proof \rangle$

theorem *state-update-vs-allInstancesATpre*:
assumes $oid \notin dom\ \sigma$
and $cp\ P$
shows $(((\!|heap{=}\sigma(oid \mapsto Object),\ assocs{=}A|\!),\ \sigma') \models (P(Type\ .oclAllInstances@pre()))) =$
$(((\!|heap{=}\sigma,\ assocs{=}A|\!),\ \sigma') \models (P((Type\ .oclAllInstances@pre()){-}{>}including(\lambda\ \text{-}.\ Some(Some((the\text{-}inv\ Type)\ Object))))))$
$\langle proof \rangle$

definition $oclisnew{::}\ ('\mathfrak{A},\ '\alpha{::}\{null,object\})val \Rightarrow ('\mathfrak{A})Boolean \quad ((\text{-}).oclIsNew'('))$
where $X\ .oclIsNew() \equiv (\lambda\tau\ .\ if\ (\delta\ X)\ \tau = true\ \tau$
$\quad then\ \lfloor\lfloor oid\text{-}of\ (X\ \tau) \notin dom(heap(fst\ \tau)) \wedge$
$\quad\quad oid\text{-}of\ (X\ \tau) \in dom(heap(snd\ \tau))\rfloor\rfloor$
$\quad else\ invalid\ \tau)$

The following predicate — which is not part of the OCL standard descriptions — provides a simple, but powerful means to describe framing conditions. For any formal approach, be it animation of OCL contracts, test-case generation or die-hard theorem proving, the specification of the part of a system transistion that DOES NOT CHANGE is of premordial importance. The following operator establishes the equality between old and new objects in the state (provided that they exist in both states), with the exception of those objects

definition $oclismodified\ {::}('\mathfrak{A}{::}object,'\alpha{::}\{null,object\})Set \Rightarrow\ '\mathfrak{A}\ Boolean$
$\quad (\text{-}{-}{>}oclIsModifiedOnly'('))$
where $X{-}{>}oclIsModifiedOnly() \equiv (\lambda(\sigma,\sigma').\ \ let\ \ X' = (oid\text{-}of\ `\ \lceil\lceil Rep\text{-}Set\text{-}0(X(\sigma,\sigma'))\rceil\rceil);$
$\quad\quad S = ((dom\ (heap\ \sigma) \cap dom\ (heap\ \sigma')) - X')$
$\quad in\ if\ (\delta\ X)\ (\sigma,\sigma') = true\ (\sigma,\sigma')$
$\quad\quad then\ \lfloor\lfloor\forall\ x \in S.\ (heap\ \sigma)\ x = (heap\ \sigma')\ x\rfloor\rfloor$
$\quad\quad else\ invalid\ (\sigma,\sigma'))$

definition $atSelf\ {::}\ ('\mathfrak{A}{::}object,'\alpha{::}\{null,object\})val \Rightarrow$
$\quad ('\mathfrak{A} \Rightarrow\ '\alpha) \Rightarrow$
$\quad ('\mathfrak{A}{::}object,'\alpha{::}\{null,object\})val\ ((\text{-})@pre(\text{-}))$
where $x\ @pre\ H = (\lambda\tau\ .\ if\ (\delta\ x)\ \tau = true\ \tau$
$\quad then\ if\ oid\text{-}of\ (x\ \tau) \in dom(heap(fst\ \tau)) \wedge oid\text{-}of\ (x\ \tau) \in dom(heap\ (snd\ \tau))$
$\quad\quad then\ \ H\ \lceil(heap(fst\ \tau))(oid\text{-}of\ (x\ \tau))\rceil$
$\quad\quad else\ invalid\ \tau$
$\quad else\ invalid\ \tau)$

theorem *framing*:
  assumes *modifiesclause*:$\tau \models (X{-}{>}excluding(x)){-}{>}oclIsModifiedOnly()$
  and *represented-x*: $\tau \models \delta(x\ @pre\ H)$
  and *H-is-typerepr*: *inj H*
  shows $\tau \models (x \ \triangleq \ (x\ @pre\ H))$
$\langle proof \rangle$

end

theory *OCL-tools*
imports *OCL-core*
begin

end

theory *OCL-main*
imports *OCL-lib OCL-state OCL-tools*
begin

end

Part III.

Conclusion

5. Conclusion

5.1. Lessons Learned

While our paper and pencil arguments, given in [4], 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 [18] or SMT-solvers like Z3 [11] completely impractical. Concretely, if the expression `not(null)` is defined `invalid` (as is the case in the present standard [16]), 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."

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" [4] is then correct, and the DNF-normaliation 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 [3] for details) are valid in Featherweight OCL.

5.2. 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 [8]. 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 [17]) 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 [3]. 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 [1]. It remains to be shown that the standard, Kodkod [18] based animator in Isabelle can give a similar quality of animation as the OCLexec Tool [12]
- 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.

Bibliography

[1] 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*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer-Verlag, 2010.

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

[3] 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*, number 6627 in Lecture Notes in Computer Science, pages 334–348. Springer-Verlag, 2010. Selected best papers from all satellite events of the MoDELS 2010 conference. Workshop on OCL and Textual Modelling.

[4] 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 Lecture Notes in Computer Science, pages 261–275. Springer-Verlag, 2009. Selected best papers from all satellite events of the MoDELS 2009 conference.

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

[6] 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 Lecture Notes in Computer Science, pages 97–100. Springer-Verlag, 2008.

[7] A. D. Brucker and B. Wolff. An extensible encoding of object-oriented data models in HOL. *Journal of Automated Reasoning*, 41:219–249, 2008.

[8] A. D. Brucker and B. Wolff. HOL-TestGen: An interactive test-case generation framework. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering (FASE09)*, number 5503 in Lecture Notes in Computer Science, pages 417–420. Springer-Verlag, 2009.

[9] A. D. Brucker and B. Wolff. Semantics, calculi, and analysis for object-oriented specifications. *Acta Informatica*, 46(4):255–284, July 2009.

[10] 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)*, 2012.

[11] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Heidelberg, 2008. Springer-Verlag.

[12] M. P. Krieger, A. Knapp, and B. Wolff. Generative programming and component engineering. In E. Visser and J. Järvi, editors, *International Conference on Generative Programming and Component Engineering (GPCE 2010)*, pages 53–62. ACM, Oct. 2010.

[13] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2002.

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

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

[16] UML 2.3.1 OCL specification, Feb. 2012. Available as OMG document formal/2012-01-01.

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

[18] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647, Heidelberg, 2007. Springer-Verlag.