Extended Version

Featherweight OCL

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

Achim D. Brucker   Burkhart Wolff

November 21, 2012

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

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

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 α list or α set). The type constructor for the function space $\Rightarrow$ is written infix: $\alpha\Rightarrow\beta$; multiple applications like $\tau_1\Rightarrow(\ldots\Rightarrow(\tau_n\Rightarrow\tau_{n+1})\ldots)$ have the alternative syntax $[\tau_1,\ldots,\tau_n]\Rightarrow\tau_{n+1}$. HOL is centered around the extensional logical equality $\_ = \_$ with type $[\alpha,\alpha]\Rightarrow\mathrm{bool}$, 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 $\_\land\_$, $\_\lor\_$, $\_ \Rightarrow \_$ 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 $\tau_\bot := \bot \mid \lfloor\_\rfloor : \alpha$ that assigns to each type τ a type $\tau_\bot$ *disjointly extended* by the exceptional element $\bot$. The function $\ulcorner\_\urcorner : \alpha_\bot \Rightarrow \alpha$ is the inverse of $\lfloor\_\rfloor$ (unspecified for $\bot$). Partial functions $\alpha\rightharpoonup\beta$ are defined as functions $\alpha\Rightarrow\beta_\bot$ supporting the usual concepts of domain (dom _) and range (ran _). The library is built entirely by logically safe, conservative definitions and derived rules. This methodology is also applied to HOL-OCL [6] and Featherweight OCL.

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.

Part II.

A Formal Semantics of OCL 2.3 in Isabelle/HOL

3. Part I: Core Definitions and Library

theory
  OCL-core
imports
  Main
begin

3.1. Foundational Notations

3.1.1. 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* ($\bot$)

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

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

3.1.2. 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.

type-synonym *oid* = *ind*

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

type-synonym $('\mathfrak{A})$*state* = *oid* $\rightharpoonup$ $'\mathfrak{A}$

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

3.1.3. 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*{**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 uncomparable 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* : $null \neq bot$

3.1.4. 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 $\langle proof \rangle$
end

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

 instance $\langle proof \rangle$
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).

3.2. 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*: $I[\![null::('\mathfrak{A},'\alpha::null)\ val]\!]\ \tau = (null::'\alpha::null)$
⟨*proof*⟩

3.3. 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 $('\mathfrak{A})Boolean = ('\mathfrak{A},bool\ option\ option)\ val$

3.3.1. Basic Constants

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

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

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*: $X\ \tau = invalid\ \tau \lor X\ \tau = null\ \tau \lor$
$X\ \tau = true\ \tau \quad \lor X\ \tau = false\ \tau$
⟨*proof*⟩

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

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

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*)

3.3.2. 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 $\delta\ X \equiv\ \lambda\ \tau\ .\ if\ X\ \tau = bot\ \tau\ \lor\ X\ \tau = null\ \tau\ then\ false\ \tau\ else\ true\ \tau$

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

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

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

lemma *defined3*[*simp*]: $\delta\ true = true$

⟨*proof*⟩

lemma *defined4*[*simp*]: δ *false* = *true*
⟨*proof*⟩

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

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

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

lemma *valid6*[*simp*]: $\upsilon\ \delta\ X$ = *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*: $I[\![\delta(X)]\!]\ \tau = (\textit{if}\ I[\![X]\!]\ \tau = I[\![\textit{bot}]\!]\ \tau\ \vee I[\![X]\!]\ \tau = I[\![\textit{null}]\!]\ \tau$
$\textit{then}\ I[\![\textit{false}]\!]\ \tau$
$\textit{else}\ I[\![\textit{true}]\!]\ \tau)$
⟨*proof*⟩

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

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

Name	Theorem
defined-def-textbook	$I[\![\delta\ ?X]\!]\ ?\tau = (\textit{if}\ I[\![?X]\!]\ ?\tau = I[\![\textit{OCL-core.bot-class.bot}]\!]\ ?\tau \vee I[\![?X]\!]\ ?\tau =$
valid-def-textbook	$I[\![\upsilon\ ?X]\!]\ ?\tau = (\textit{if}\ I[\![?X]\!]\ ?\tau = I[\![\textit{OCL-core.bot-class.bot}]\!]\ ?\tau\ \textit{then}\ I[\![\textit{false}]\!]\ ?$

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

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

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

3.3.3. 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*::$['\mathfrak{A}\ st \Rightarrow '\alpha,'\mathfrak{A}\ st \Rightarrow '\alpha] \Rightarrow ('\mathfrak{A})Boolean$ (**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*]: $(X \triangleq X) = true$
⟨*proof*⟩

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

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

3.3.4. Fundamental Predicates III

And, last but not least,

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

lemma *valid5*[*simp*]: $\upsilon\ (X \triangleq Y) = true$
⟨*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.

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

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

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*)

$$
\begin{array}{llllll}
\textbf{where} & X \textit{ and } Y \equiv (\lambda\,\tau\,.\ \textit{case } X\ \tau \textit{ of} \\
& \quad \bot \Rightarrow (\textit{case } Y\ \tau \textit{ of} \\
& \qquad\qquad \bot \Rightarrow \bot \\
& \qquad\qquad \mid \lfloor\bot\rfloor \Rightarrow \bot \\
& \qquad\qquad \mid \lfloor\lfloor \textit{True}\rfloor\rfloor \Rightarrow \bot \\
& \qquad\qquad \mid \lfloor\lfloor \textit{False}\rfloor\rfloor \Rightarrow \lfloor\lfloor \textit{False}\rfloor\rfloor) \\
& \quad \mid \lfloor\,\bot\,\rfloor \Rightarrow (\textit{case } Y\ \tau \textit{ of} \\
& \qquad\qquad \bot \Rightarrow \bot \\
& \qquad\qquad \mid \lfloor\bot\rfloor \Rightarrow \lfloor\bot\rfloor \\
& \qquad\qquad \mid \lfloor\lfloor \textit{True}\rfloor\rfloor \Rightarrow \lfloor\bot\rfloor \\
& \qquad\qquad \mid \lfloor\lfloor \textit{False}\rfloor\rfloor \Rightarrow \lfloor\lfloor \textit{False}\rfloor\rfloor) \\
& \quad \mid \lfloor\lfloor \textit{True}\rfloor\rfloor \Rightarrow (\textit{case } Y\ \tau \textit{ of} \\
& \qquad\qquad \bot \Rightarrow \bot \\
& \qquad\qquad \mid \lfloor\bot\rfloor \Rightarrow \lfloor\bot\rfloor \\
& \qquad\qquad \mid \lfloor\lfloor y\rfloor\rfloor \Rightarrow \lfloor\lfloor y\rfloor\rfloor) \\
& \quad \mid \lfloor\lfloor \textit{False}\rfloor\rfloor \Rightarrow \lfloor\lfloor\ \textit{False}\ \rfloor\rfloor)
\end{array}
$$

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*:

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

⟨*proof*⟩

lemma *textbook-and*:

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

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

⟨*proof*⟩

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

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

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

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

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

lemma *ocl-and1*[*simp*]: (*invalid and true*) = *invalid*
⟨*proof*⟩
lemma *ocl-and2*[*simp*]: (*invalid and false*) = *false*
⟨*proof*⟩
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*⟩

3.4. 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 $\tau \models P \equiv ((P\ \tau) = true\ \tau)$

3.4.1. 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*⟩

3.4.2. 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*]:
$(\tau \models not\ (\upsilon\ x)) = (\neg\ (\tau \models \upsilon\ x))$
⟨*proof*⟩

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*:
$(\tau \models \delta\ x) \vee (\tau \models (x \triangleq invalid)) \vee (\tau \models (x \triangleq null))$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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*⟩

3.4.3. 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]]

3.4.4. 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.

3.5. 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*:$((if\ C\ then\ B_1\ else\ B_2\ endif)\ \tau =$

$(\mathit{if}\ (\lambda\ \text{-}.\ C\ \tau)\ \mathit{then}\ (\lambda\ \text{-}.\ B_1\ \tau)\ \mathit{else}\ (\lambda\ \text{-}.\ B_2\ \tau)\ \mathit{endif})\ \tau)$

⟨*proof*⟩

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

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

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

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

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

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

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

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

end

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

3.6. 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 $('\mathfrak{A})\mathit{Integer} = ('\mathfrak{A}, \mathit{int\ option\ option})\ \mathit{val}$

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

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.

3.6.1. 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
$notequal \quad :: ('\mathfrak{A})Boolean \Rightarrow ('\mathfrak{A})Boolean \Rightarrow ('\mathfrak{A})Boolean$ (**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*] :

$$(x::('\mathfrak{A})Boolean) \doteq y \equiv \lambda\ \tau.\ if\ (\upsilon\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau\ then\ (x \triangleq y)\tau\ else\ invalid\ \tau$$

3.6.2. 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*:
$\tau \models(\upsilon\ x) \Longrightarrow \tau \models(\upsilon\ y) \Longrightarrow (\tau \models (((x::('\mathfrak{A})Integer) \doteq y) \triangleq (x \triangleq y)))$
$\langle proof \rangle$

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

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

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

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

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

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*:
$((X{::}({}'\mathfrak{A})\mathit{Integer}) \doteq Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau) \doteq (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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".

3.6.3. Test Statements on Basic Types.

Elementary computations on Booleans

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

Elementary computations on Integer

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

lemma $\delta(\mathit{null}::('\mathfrak{A})\mathit{Integer}) = \mathit{false}$ $\langle \mathit{proof} \rangle$
lemma $\upsilon(\mathit{null}::('\mathfrak{A})\mathit{Integer}) = \mathit{true}$ $\langle \mathit{proof} \rangle$

3.6.4. More algebraic and logical layer on basic types

lemma [*simp*,*code-unfold*]:$\upsilon\ \mathbf{0} = \mathit{true}$
$\langle \mathit{proof} \rangle$

lemma [*simp*,*code-unfold*]:$\delta\ \mathbf{1} = \mathit{true}$
$\langle \mathit{proof} \rangle$

lemma [*simp*,*code-unfold*]:$\upsilon\ \mathbf{1} = \mathit{true}$
$\langle \mathit{proof} \rangle$

lemma [*simp*,*code-unfold*]:$\delta\ \mathbf{2} = \mathit{true}$

$\langle proof \rangle$

lemma $[simp,code\text{-}unfold]$:$\upsilon$ $\mathbf{2} = true$
$\langle proof \rangle$

lemma $[simp,code\text{-}unfold]$: υ $\mathbf{6} = true$
$\langle proof \rangle$

lemma $[simp,code\text{-}unfold]$: υ $\mathbf{8} = true$
$\langle proof \rangle$

lemma $[simp,code\text{-}unfold]$: υ $\mathbf{9} = true$
$\langle proof \rangle$

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

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

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

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 $x \oplus y \equiv \lambda\, \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$\quad then\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil + \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\quad else\ invalid\ \tau$

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

definition *ocl-le-int* $::('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Boolean$ (**infix** $\preceq$ *40*)

where $x \preceq y \equiv \lambda\ \tau.\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau \wedge (\delta\ y)\ \tau = \mathit{true}\ \tau$
$\quad\quad \mathit{then}\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil \leq \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\quad\quad \mathit{else}\ \mathit{invalid}\ \tau$

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

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

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

3.7.1. 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 $\mathit{Set}(\mathit{Set}(\mathit{Sequences}(\mathit{Pairs}(X,Y))))$, and

The former principe rules out the option to define $'\alpha$ *Set* just by $('\mathfrak{A}, ('\alpha\ \mathit{option}\ \mathit{option})\ \mathit{set})\ \mathit{val}$. This would allow sets to contain junk elements such as $\{\bot\}$ 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* $= \{X::('\alpha::\mathit{null})\ \mathit{set}\ \mathit{option}\ \mathit{option}.$
$\quad\quad X = \mathit{bot} \vee X = \mathit{null} \vee (\forall x\in\lceil\lceil X\rceil\rceil.\ x \neq \mathit{bot})\}$
$\quad \langle \mathit{proof} \rangle$

instantiation *Set-0* :: $(\mathit{null})\mathit{bot}$
begin

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

instance $\langle \mathit{proof} \rangle$
end

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

definition *null-Set-0-def*: (*null*::('*a*::*null*) *Set-0*) ≡ *Abs-Set-0* ⌊ *None* ⌋

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-0*$)$ *val*

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

3.7.2. Constants on Sets

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

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.

3.7.3. 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*⟩

3.7.4. 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* :
$((x::('\mathfrak{A},'\alpha::null)Set) \doteq x) = (if\ (\upsilon\ x)\ then\ true\ else\ invalid\ endif)$
⟨*proof*⟩

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.

3.7.5. Library Operations on Sets

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

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

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

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

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 $OclForall\ S\ P = (\lambda\ \tau.\ if\ (\delta\ S)\ \tau = true\ \tau$
$then\ if\ (\forall\, x\in\lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil.\ P\ (\lambda\ \text{-}.\ x)\ \tau = true\ \tau)$
$then\ true\ \tau$
$else\ if\ (\forall\, x\in\lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil.\ P(\lambda\ \text{-}.\ x)\ \tau = true\ \tau \vee$
$P(\lambda\ \text{-}.\ x)\ \tau = false\ \tau)$
$then\ false\ \tau$
$else\ \perp$
$else\ \perp)$

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

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

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 (-−>*size*'(') [66])
and
OclCount (-−>*count*'(-') [66,65]65)
and
OclIncludes (-−>*includes*'(-') [66,65]65)
and
OclExcludes (-−>*excludes*'(-') [66,65]65)
and
OclSum (-−>*sum*'(') [66])
and
OclIncludesAll (-−>*includesAll*'(-') [66,65]65)
and
OclExcludesAll (-−>*excludesAll*'(-') [66,65]65)
and
OclIsEmpty (-−>*isEmpty*'(') [66])
and

$OclNotEmpty$ $(\text{-}{-}{>}notEmpty'(')$ $[66])$
and
$OclIncluding$ $(\text{-}{-}{>}including'(\text{-}'))$
and
$OclExcluding$ $(\text{-}{-}{>}excluding'(\text{-}'))$
and
$OclComplement$ $(\text{-}{-}{>}complement'('))$
and
$OclUnion$ $(\text{-}{-}{>}union'(\text{-}')$ $[66,65]65)$
and
$OclIntersection(\text{-}{-}{>}intersection'(\text{-}')$ $[71,70]70)$

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*⟩

3.7.6. 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*]:$(X{-}{>}includes(invalid)) = invalid$
$\langle proof \rangle$

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

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

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

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

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

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
$(X{-}{>}including(x{::}('\mathfrak{A},'a{::}null)val){-}{>}includes(y)) =$
$(if\ \delta\ X\ then\ if\ x \doteq y\ then\ true\ else\ X{-}{>}includes(y)\ endif\ else\ invalid\ endif)$
⟨*proof*⟩

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 $def\text{-}X{:}\tau \models (\delta\ X)$
and $val\text{-}x{:}\tau \models (\upsilon\ x)$
and $val\text{-}y{:}\tau \models (\upsilon\ y)$
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 $def\text{-}X{:}\tau \models (\delta\ X)$
and $val\text{-}x{:}\tau \models (\upsilon\ x)$
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
  $\text{-}OclFinset :: args => ('\mathfrak{A},'a{::}null)\ Set \quad (Set\{(\text{-})\})$
translations
  $Set\{x,\ xs\} == CONST\ OclIncluding\ (Set\{xs\})\ x$
  $Set\{x\} == CONST\ OclIncluding\ (Set\{\})\ x$

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

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

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

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

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

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

⟨*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*] :
$((S{-}{>}including(x)){-}{>}exists(z \mid P(z))) = (if\ (\delta\ S)\ and\ (\upsilon\ x)$
$then\ P(x)\ or\ S{-}{>}exists(z \mid P(z))$
$else\ invalid$
$endif)$
⟨*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 $OclIterate_{Set}\ S\ A\ F = (\lambda\ \tau.\ if\ (\delta\ S)\ \tau = true\ \tau \wedge (\upsilon\ A)\ \tau = true\ \tau \wedge finite\lceil\lceil Rep\text{-}Set\text{-}0$
$(S\ \tau)\rceil\rceil$
$then\ (Finite\text{-}Set.fold\ (F)\ (A)\ ((\lambda a\ \tau.\ a)\ `\ \lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil))\tau$
$else\ \perp)$

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

3.7.7. Test Statements

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

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

Elementary computations on Sets.

value $\neg\ (\tau_0 \models \upsilon(invalid::('\mathfrak{A},'\alpha::null)\ Set))$
value $\tau_0 \models \upsilon(null::('\mathfrak{A},'\alpha::null)\ Set)$
value $\neg\ (\tau_0 \models \delta(null::('\mathfrak{A},'\alpha::null)\ Set))$
value $\tau_0 \models \upsilon(Set\{\})$
value $\tau_0 \models \upsilon(Set\{Set\{\mathbf{2}\},null\})$
value $\tau_0 \models \delta(Set\{Set\{\mathbf{2}\},null\})$
value $\tau_0 \models (Set\{\mathbf{2},\mathbf{1}\}{-}{>}includes(\mathbf{1}))$
value $\neg\ (\tau_0 \models (Set\{\mathbf{2}\}{-}{>}includes(\mathbf{1})))$
value $\neg\ (\tau_0 \models (Set\{\mathbf{2},\mathbf{1}\}{-}{>}includes(null)))$
value $\tau_0 \models (Set\{\mathbf{2},null\}{-}{>}includes(null))$

value $\tau \models ((\mathit{Set}\{\mathbf{2},\mathbf{1}\}){-}{>}\mathit{forall}(z \mid \mathbf{0} \prec z))$
value $\neg\, (\tau \models ((\mathit{Set}\{\mathbf{2},\mathbf{1}\}){-}{>}\mathit{exists}(z \mid z \prec \mathbf{0}\ )))$

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

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

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

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 $oid = ind$

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

type-synonym $('\mathfrak{A})state = oid \rightharpoonup {'\mathfrak{A}}$

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 $gen\text{-}ref\text{-}eq\ x\ y$

$$\begin{aligned}&\equiv \lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau\\ &\qquad then\ if\ x\ \tau = null \vee y\ \tau = null\\ &\qquad\qquad then\ \lfloor\lfloor x\ \tau = null \wedge y\ \tau = null\rfloor\rfloor\\ &\qquad\qquad else\ \lfloor\lfloor(oid\text{-}of\ (x\ \tau)) = (oid\text{-}of\ (y\ \tau))\ \rfloor\rfloor\\ &\qquad else\ invalid\ \tau\end{aligned}$$

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 $WFF\ \tau = ((\forall\ x \in ran(fst\ \tau).\ \lceil fst\ \tau\ (oid\text{-}of\ x)\rceil = x) \wedge$
        $(\forall\ x \in ran(snd\ \tau).\ \lceil snd\ \tau\ (oid\text{-}of\ x)\rceil = x))$

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*:

$WFF\ \tau \Longrightarrow \tau \models (\delta\ x) \Longrightarrow \tau \models (\delta\ y) \Longrightarrow$
$\quad (x\ \tau \in ran\ (fst\ \tau) \wedge y\ \tau \in ran\ (fst\ \tau)) \wedge$
$\quad (x\ \tau \in ran\ (snd\ \tau) \wedge y\ \tau \in ran\ (snd\ \tau)) \Longrightarrow$ (* *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. Miscellaneous: Initial States (for Testing and Code Generation)

definition $\tau_0 :: ('\mathfrak{A})st$
where $\quad \tau_0 \equiv (Map.empty, Map.empty)$

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

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

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 $\quad cp\ P$
shows $((\sigma, \sigma'(oid \mapsto Object)) \models (P(Type\ .oclAllInstances()))) =$
$\qquad ((\sigma, \sigma') \models (P((Type\ .oclAllInstances())\text{->}including(\lambda\ \text{-}.\ Some(Some((the\text{-}inv\ Type)\ Object))))))$
⟨*proof*⟩

theorem *state-update-vs-allInstancesATpre*:

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

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

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

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

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

5. Part III: OCL Contracts and an Example

theory
  OCL-linked-list
imports
  ../OCL-main
begin

5.0.2. Introduction

For certain concepts like Classes and Class-types, only a generic definition for its resulting semantics can be given. Generic means, there is a function outside HOL that "compiles" a concrete, closed-world class diagram into a "theory" of this data model, consisting of a bunch of definitions for classes, accessors, method, casts, and tests for actual types, as well as proofs for the fundamental properties of these operations in this concrete data model.

Such generic function or "compiler" can be implemented in Isabelle on the ML level. This has been done, for a semantics following the open-world assumption, for UML 2.0 in [7]. In this paper, we follow another approach for UML 2.4: we define the concepts of the compilation informally, an present a concrete example which is verified in Isabelle/HOL.

5.0.3. Outlining the Example

5.0.4. Example Data-Universe and its Infrastructure

Should be generated entirely from a class-diagram.

Our data universe consists in the concrete class diagram just of node's, and implicitly of the class object. Each class implies the existence of a class type defined for the corresponding object representations as follows:

datatype *node* = mk_{node} *oid*
    int option
    oid option

datatype *object*= mk_{object} *oid*
    (*int option* $\times$ *oid option*) *option*

Now, we construct a concrete "universe of object types" by injection into a sum type containing the class types. This type of objects will be used as instance for all resp. type-variables ...

datatype $\mathfrak{A} = in_{node}$ *node* | in_{object} *object*

Recall that 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 *Node* :: $\mathfrak{A} \Rightarrow$ *node*
where *Node* $\equiv$ (*the-inv* in_{node})

definition *Object* :: $\mathfrak{A} \Rightarrow$ *object*
where *Object* $\equiv$ (*the-inv* in_{object})

Having fixed the object universe, we can introduce type synonyms that exactly correspond to OCL types. Again, we exploit that our representation of OCL is a "shallow embedding" with a one-to-one correspondance of OCL-types to types of the meta-language HOL.

type-synonym *Boolean* = ($\mathfrak{A}$)*Boolean*
type-synonym *Integer* = ($\mathfrak{A}$)*Integer*
type-synonym *Void* = ($\mathfrak{A}$)*Void*
type-synonym *Object* = ($\mathfrak{A}$,*object option option*) *val*
type-synonym *Node* = ($\mathfrak{A}$, *node option option*)*val*
type-synonym *Set-Integer* = ($\mathfrak{A}$, *int option option*)*Set*
type-synonym *Set-Node* = ($\mathfrak{A}$, *node option option*)*Set*

Just a little check:

typ *Boolean*

In order to reuse key-elements of the library like referential equality, we have to show that the object universe belongs to the type class "object", i.e. each class type has to provide a function *oid-of* yielding the object id (oid) of the object.

instantiation *node* :: *object*
begin
  definition *oid-of-node-def*: *oid-of* x = (*case* x *of* mk_{node} *oid* - - $\Rightarrow$ *oid*)
  instance ⟨*proof*⟩
end

instantiation *object* :: *object*
begin
  definition *oid-of-object-def*: *oid-of* x = (*case* x *of* mk_{object} *oid* - $\Rightarrow$ *oid*)
  instance ⟨*proof*⟩
end

instantiation $\mathfrak{A}$ :: *object*
begin
  definition *oid-of-*$\mathfrak{A}$*-def*: *oid-of* x = (*case* x *of*
    in_{node} *node* $\Rightarrow$ *oid-of node*
    | in_{object} *obj* $\Rightarrow$ *oid-of obj*)
  instance ⟨*proof*⟩

end

instantiation *option* :: (*object*)*object*
begin
  definition *oid-of-option-def*: *oid-of* x = *oid-of* (*the* x)
  instance ⟨*proof*⟩
end

5.1. Instantiation of the generic strict equality. We instantiate the referential equality on $Node$ and $Object$

defs(**overloaded**) $StrictRefEq_{node}$: $(x{::}Node) \doteq y \equiv gen\text{-}ref\text{-}eq\ x\ y$
defs(**overloaded**) $StrictRefEq_{object}$: $(x{::}Object) \doteq y \equiv gen\text{-}ref\text{-}eq\ x\ y$

lemmas *strict-eq-node* =
  cp-gen-ref-eq-object[*of x*::*Node y*::*Node* τ,
    simplified $StrictRefEq_{node}$[*symmetric*]]
  cp-intro(9) [*of P*::*Node* ⇒*NodeQ*::*Node* ⇒*Node*,
    simplified $StrictRefEq_{node}$[*symmetric*]]
  gen-ref-eq-def [*of x*::*Node y*::*Node*,
    simplified $StrictRefEq_{node}$[*symmetric*]]
  gen-ref-eq-defargs [*of - x*::*Node y*::*Node*,
    simplified $StrictRefEq_{node}$[*symmetric*]]
  gen-ref-eq-object-strict1
    [*of x*::*Node*,
    simplified $StrictRefEq_{node}$[*symmetric*]]
  gen-ref-eq-object-strict2
    [*of x*::*Node*,
    simplified $StrictRefEq_{node}$[*symmetric*]]
  gen-ref-eq-object-strict3
    [*of x*::*Node*,
    simplified $StrictRefEq_{node}$[*symmetric*]]
  gen-ref-eq-object-strict3
    [*of x*::*Node*,
    simplified $StrictRefEq_{node}$[*symmetric*]]
  gen-ref-eq-object-strict4
    [*of x*::*Node*,
    simplified $StrictRefEq_{node}$[*symmetric*]]

thm *strict-eq-node*

5.1.1. AllInstances

lemma (*Node* .*oclAllInstances*()) =
  $(\lambda\tau.\ Abs\text{-}Set\text{-}0\ \lfloor\lfloor(Some \circ Some \circ (the\text{-}inv\ in_{node}))`(ran(snd\ \tau))\ \rfloor\rfloor)$
⟨*proof*⟩

lemma (*Object* .*oclAllInstances@pre*()) =

$(\lambda\tau.\ \textit{Abs-Set-0}\ \lfloor\lfloor(\textit{Some} \circ \textit{Some} \circ (\textit{the-inv in}_{object}))`(\textit{ran}(\textit{fst}\ \tau))\ \rfloor\rfloor)$
$\langle proof \rangle$

For each Class C, we will have an casting operation `.oclAsType(`C`)`, a test on the actual type `.oclIsTypeOf(`C`)` as well as its relaxed form `.oclIsKindOf(`C`)` (corresponding exactly to Java's `instanceof`-operator.

Thus, since we have two class-types in our concrete class hierarchy, we have two operations to declare and and to provide two overloading definitions for the two static types.

5.2. Selector Definition

Should be generated entirely from a class-diagram.

typ *Node* ⇒ *Node*
fun *dot-next*:: *Node* ⇒ *Node* ((*1*(-).*next*) *50*)
  where $(X).next = (\lambda\ \tau.\ case\ X\ \tau\ of$
      $\perp \Rightarrow invalid\ \tau$ (∗ *undefined pointer* ∗)
    $|\ \lfloor\ \perp\ \rfloor \Rightarrow invalid\ \tau$ (∗ *dereferencing null pointer* ∗)
    $|\ \lfloor\lfloor\ mk_{node}\ oid\ i\ \perp\ \rfloor\rfloor \Rightarrow null\ \tau$(∗ *object contains null pointer* ∗)
    $|\ \lfloor\lfloor\ mk_{node}\ oid\ i\ \lfloor next \rfloor\ \rfloor\rfloor \Rightarrow$ (∗ *We assume here that oid is indeed 'the' oid of the Node,*
        ie. we assume that τ *is well−formed.* ∗)
      $case\ (snd\ \tau)\ next\ of$
        $\perp \Rightarrow invalid\ \tau$
      $|\ \lfloor in_{node}\ (mk_{node}\ a\ b\ c)\rfloor \Rightarrow \lfloor\lfloor mk_{node}\ a\ b\ c\ \rfloor\rfloor$
      $|\ \lfloor\ \text{-}\ \rfloor \Rightarrow invalid\ \tau)$

fun *dot-i*:: *Node* ⇒ *Integer* ((*1*(-).*i*) *50*)
  where $(X).i = (\lambda\ \tau.\ case\ X\ \tau\ of$
      $\perp \Rightarrow invalid\ \tau$
    $|\ \lfloor\ \perp\ \rfloor \Rightarrow invalid\ \tau$
    $|\ \lfloor\lfloor\ mk_{node}\ oid\ \perp\ \text{-}\ \rfloor\rfloor \Rightarrow\ null\ \tau$
    $|\ \lfloor\lfloor\ mk_{node}\ oid\ \lfloor i \rfloor\ \text{-}\ \rfloor\rfloor \Rightarrow\ \lfloor\lfloor\ i\ \rfloor\rfloor)$

fun *dot-next-at-pre*:: *Node* ⇒ *Node* ((*1*(-).*next*@*pre*) *50*)
  where $(X).next@pre = (\lambda\ \tau.\ case\ X\ \tau\ of$
      $\perp \Rightarrow invalid\ \tau$
    $|\ \lfloor\ \perp\ \rfloor \Rightarrow invalid\ \tau$
    $|\ \lfloor\lfloor\ mk_{node}\ oid\ i\ \perp\ \rfloor\rfloor \Rightarrow null\ \tau$(∗ *object contains null pointer. REALLY ?*
        And if this pointer was defined in the pre−state ?∗)
    $|\ \lfloor\lfloor\ mk_{node}\ oid\ i\ \lfloor next \rfloor\ \rfloor\rfloor \Rightarrow$ (∗ *We assume here that oid is indeed 'the' oid of the Node,*
        ie. we assume that τ *is well−formed.* ∗)
      $(case\ (fst\ \tau)\ next\ of$
        $\perp \Rightarrow invalid\ \tau$
      $|\ \lfloor in_{node}\ (mk_{node}\ a\ b\ c)\rfloor \Rightarrow \lfloor\lfloor mk_{node}\ a\ b\ c\ \rfloor\rfloor$
      $|\ \lfloor\ \text{-}\ \rfloor \Rightarrow invalid\ \tau))$

fun *dot-i-at-pre*:: *Node* ⇒ *Integer* $((1(\text{-}).i@pre)\ 50)$
where $(X).i@pre = (\lambda\ \tau.\ case\ X\ \tau\ of$
$\quad\quad \bot \Rightarrow invalid\ \tau$
$\quad | \lfloor\ \bot\ \rfloor \Rightarrow invalid\ \tau$
$\quad | \lfloor\lfloor\ mk_{node}\ oid\ \text{-}\ \text{-}\ \rfloor\rfloor \Rightarrow$
$\quad\quad\quad if\ oid \in dom\ (fst\ \tau)$
$\quad\quad\quad then\ (case\ (fst\ \tau)\ oid\ of$
$\quad\quad\quad\quad\quad \bot \Rightarrow invalid\ \tau$
$\quad\quad\quad\quad | \lfloor in_{node}\ (mk_{node}\ oid\ \bot\ next)\ \rfloor \Rightarrow null\ \tau$
$\quad\quad\quad\quad | \lfloor in_{node}\ (mk_{node}\ oid\ \lfloor i \rfloor next)\ \rfloor \Rightarrow \lfloor\lfloor\ i\ \rfloor\rfloor$
$\quad\quad\quad\quad | \lfloor\ \text{-}\ \rfloor \Rightarrow invalid\ \tau)$
$\quad\quad\quad else\ invalid\ \tau)$

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

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

lemma *cp-dot-next-at-pre*: $((X).next@pre)\ \tau = ((\lambda\text{-}.\ X\ \tau).next@pre)\ \tau$ ⟨*proof*⟩

lemma *cp-dot-i-pre*: $((X).i@pre)\ \tau = ((\lambda\text{-}.\ X\ \tau).i@pre)\ \tau$ ⟨*proof*⟩

lemmas *cp-dot-nextI* [*simp*, *intro*!]=
cp-dot-next[*THEN allI*[*THEN allI*], *of* $\lambda\ X\ \text{-}.\ X\ \lambda\ \text{-}\ \tau.\ \tau$, *THEN cpI1*]

lemmas *cp-dot-nextI-at-pre* [*simp*, *intro*!]=
cp-dot-next-at-pre[*THEN allI*[*THEN allI*],
of $\lambda\ X\ \text{-}.\ X\ \lambda\ \text{-}\ \tau.\ \tau$, *THEN cpI1*]

lemma *dot-next-nullstrict* [*simp*]: $(null).next = invalid$
⟨*proof*⟩

lemma *dot-next-at-pre-nullstrict* [*simp*] : $(null).next@pre = invalid$
⟨*proof*⟩

lemma *dot-next-strict*[*simp*] : $(invalid).next = invalid$
⟨*proof*⟩

lemma *dot-next-strict'*[*simp*] : $(null).next = invalid$
⟨*proof*⟩

lemma *dot-nextATpre-strict*[*simp*] : $(invalid).next@pre = invalid$
⟨*proof*⟩

lemma *dot-nextATpre-strict'*[*simp*] : $(null).next@pre = invalid$
⟨*proof*⟩

5.2.1. Casts

consts $oclastype_{object}$ :: $'\alpha \Rightarrow Object$ ((-) .oclAsType'(Object'))
consts $oclastype_{node}$ :: $'\alpha \Rightarrow Node$ ((-) .oclAsType'(Node'))

defs (**overloaded**) $oclastype_{object}$-*Object*:
$$(X{::}Object)\ .oclAsType(Object) \equiv$$
$$(\lambda\tau.\ case\ X\ \tau\ of$$
$$\perp \ \Rightarrow invalid\ \tau$$
$$|\ \lfloor\perp\rfloor \Rightarrow invalid\ \tau \quad (*\ to\ avoid{:}\ null\ .oclAsType(Object) = null\ ?\ *)$$
$$|\ \lfloor\lfloor mk_{object}\ oid\ a\ \rfloor\rfloor \Rightarrow\ \lfloor\lfloor mk_{object}\ oid\ a\ \rfloor\rfloor)$$

defs (**overloaded**) $oclastype_{object}$-*Node*:
$$(X{::}Node)\ .oclAsType(Object) \equiv$$
$$(\lambda\tau.\ case\ X\ \tau\ of$$
$$\perp \ \Rightarrow invalid\ \tau$$
$$|\ \lfloor\perp\rfloor \Rightarrow invalid\ \tau \quad (*\ OTHER\ POSSIBILITY : null\ ???\ Really\ excluded\ by\ standard\ *)$$
$$|\ \lfloor\lfloor mk_{node}\ oid\ a\ b\ \rfloor\rfloor \Rightarrow\ \lfloor\lfloor\ (mk_{object}\ oid\ \lfloor(a,b)\rfloor)\ \rfloor\rfloor)$$

defs (**overloaded**) $oclastype_{node}$-*Object*:
$$(X{::}Object)\ .oclAsType(Node) \equiv$$
$$(\lambda\tau.\ case\ X\ \tau\ of$$
$$\perp \ \Rightarrow invalid\ \tau$$
$$|\ \lfloor\perp\rfloor \Rightarrow invalid\ \tau$$
$$|\ \lfloor\lfloor mk_{object}\ oid\ \perp\ \rfloor\rfloor \Rightarrow\ invalid\ \tau \quad (*\ down{-}cast\ exception\ *)$$
$$|\ \lfloor\lfloor mk_{object}\ oid\ \lfloor(a,b)\rfloor\ \rfloor\rfloor \Rightarrow\ \lfloor\lfloor mk_{node}\ oid\ a\ b\ \rfloor\rfloor)$$

defs (**overloaded**) $oclastype_{node}$-*Node*:
$$(X{::}Node)\ .oclAsType(Node) \equiv$$
$$(\lambda\tau.\ case\ X\ \tau\ of$$
$$\perp \ \Rightarrow invalid\ \tau$$
$$|\ \lfloor\perp\rfloor \Rightarrow invalid\ \tau \quad (*\ to\ avoid{:}\ null\ .oclAsType(Object) = null\ ?\ *)$$
$$|\ \lfloor\lfloor mk_{node}\ oid\ a\ b\ \rfloor\rfloor \Rightarrow\ \lfloor\lfloor mk_{node}\ oid\ a\ b\rfloor\rfloor)$$

lemma $oclastype_{object}$-*Object-strict*[*simp*] : (*invalid*::*Object*) .*oclAsType*(*Object*) = *invalid*
⟨*proof*⟩

lemma $oclastype_{object}$-*Object-nullstrict*[*simp*] : (*null*::*Object*) .*oclAsType*(*Object*) = *invalid*
⟨*proof*⟩

lemma $oclastype_{node}$-*Object-strict*[*simp*] : (*invalid*::*Node*) .*oclAsType*(*Object*) = *invalid*
⟨*proof*⟩

lemma $oclastype_{node}$-*Object-nullstrict*[*simp*] : (*null*::*Node*) .*oclAsType*(*Object*) = *invalid*
⟨*proof*⟩

5.3. Tests for Actual Types

consts $oclistypeof_{object}$ :: $'\alpha \Rightarrow Boolean$ $((\text{-}).oclIsTypeOf'(Object'))$
consts $oclistypeof_{node}$ :: $'\alpha \Rightarrow Boolean$ $((\text{-}).oclIsTypeOf'(Node'))$

defs (**overloaded**) $oclistypeof_{object}$-*Object*:
$$(X::Object)\ .oclIsTypeOf(Object) \equiv$$
$$(\lambda\tau.\ case\ X\ \tau\ of$$
$$\bot \ \Rightarrow invalid\ \tau$$
$$|\ \lfloor\bot\rfloor \Rightarrow invalid\ \tau$$
$$|\ \lfloor\lfloor mk_{object}\ oid\ \bot\ \rfloor\rfloor \Rightarrow true\ \tau$$
$$|\ \lfloor\lfloor mk_{object}\ oid\ \lfloor\text{-}\rfloor\ \rfloor\rfloor \Rightarrow false\ \tau)$$

defs (**overloaded**) $oclistypeof_{object}$-*Node*:
$$(X::Node)\ .oclIsTypeOf(Object) \equiv$$
$$(\lambda\tau.\ case\ X\ \tau\ of$$
$$\bot \ \Rightarrow invalid\ \tau$$
$$|\ \lfloor\bot\rfloor \Rightarrow invalid\ \tau$$
$$|\ \lfloor\lfloor\ \text{-}\ \rfloor\rfloor \Rightarrow false\ \tau)$$

defs (**overloaded**) $oclistypeof_{node}$-*Object*:
$$(X::Object)\ .oclIsTypeOf(Node) \equiv$$
$$(\lambda\tau.\ case\ X\ \tau\ of$$
$$\bot \ \Rightarrow invalid\ \tau$$
$$|\ \lfloor\bot\rfloor \Rightarrow invalid\ \tau$$
$$|\ \lfloor\lfloor mk_{object}\ oid\ \bot\ \rfloor\rfloor \Rightarrow false\ \tau$$
$$|\ \lfloor\lfloor mk_{object}\ oid\ \lfloor\text{-}\rfloor\ \rfloor\rfloor \Rightarrow true\ \tau)$$

defs (**overloaded**) $oclistypeof_{node}$-*Node*:
$$(X::Node)\ .oclIsTypeOf(Node) \equiv$$
$$(\lambda\tau.\ case\ X\ \tau\ of$$
$$\bot \ \Rightarrow invalid\ \tau$$
$$|\ \lfloor\bot\rfloor \Rightarrow invalid\ \tau$$
$$|\ \lfloor\lfloor\ \text{-}\ \rfloor\rfloor \Rightarrow true\ \tau)$$

lemma $oclistypeof_{object}$-*Object-strict1*[*simp*]:
$(invalid::Object)\ .oclIsTypeOf(Object) = invalid$
⟨*proof*⟩
lemma $oclistypeof_{object}$-*Object-strict2*[*simp*]:
$(null::Object)\ .oclIsTypeOf(Object) = invalid$
⟨*proof*⟩
lemma $oclistypeof_{object}$-*Node-strict1*[*simp*]:
$(invalid::Node)\ .oclIsTypeOf(Object) = invalid$
⟨*proof*⟩
lemma $oclistypeof_{object}$-*Node-strict2*[*simp*]:
$(null::Node)\ .oclIsTypeOf(Object) = invalid$

⟨*proof*⟩
lemma *oclistypeof$_{node}$-Object-strict1*[*simp*]:
  $(invalid::Object)$.*oclIsTypeOf*$(Node) = invalid$
⟨*proof*⟩
lemma *oclistypeof$_{node}$-Object-strict2*[*simp*]:
  $(null::Object)$.*oclIsTypeOf*$(Node) = invalid$
⟨*proof*⟩
lemma *oclistypeof$_{node}$-Node-strict1*[*simp*]:
  $(invalid::Node)$.*oclIsTypeOf*$(Node) = invalid$
⟨*proof*⟩
lemma *oclistypeof$_{node}$-Node-strict2*[*simp*]:
  $(null::Node)$.*oclIsTypeOf*$(Node) = invalid$
⟨*proof*⟩

lemma *actualType-larger-staticType*:
assumes *isdef*: $\tau \models (\delta\ X)$
shows $\tau \models (X::Node)$.*oclIsTypeOf*$(Object) \triangleq false$
⟨*proof*⟩

lemma *down-cast*:
assumes *isObject*: $\tau \models (X::Object)$.*oclIsTypeOf*$(Object)$
shows $\tau \models (X$.*oclAsType*$(Node)) \triangleq invalid$
⟨*proof*⟩

lemma *up-down-cast* :
assumes *isdef*: $\tau \models (\delta\ X)$
shows $\tau \models ((X::Node)$.*oclAsType*$(Object)$.*oclAsType*$(Node) \triangleq X)$
⟨*proof*⟩

5.4. Standard State Infrastructure

These definitions should be generated — again — from the class diagram.

5.5. Invariant

These recursive predicates can be defined conservatively by greatest fix-point constructions - automatically. See HOL-OCL Book for details. For the purpose of this example, we state them as axioms here.

axiomatization *inv-Node* :: *Node* ⇒ *Boolean*
where $A : (\tau \models (\delta\ self)) \longrightarrow$
  $(\tau \models inv\text{-}Node(self)) =$
   $((\tau \models (self\ .next \doteq null)) \lor$
   $(\ \tau \models (self\ .next <> null) \land (\tau \models (self\ .next\ .i \prec self\ .i))\ \land$
   $(\tau \models (inv\text{-}Node(self\ .next)))))$

axiomatization $\mathit{inv\text{-}Node\text{-}at\text{-}pre} :: \mathit{Node} \Rightarrow \mathit{Boolean}$
where $B : (\tau \models (\delta\ \mathit{self})) \longrightarrow$
$(\tau \models \mathit{inv\text{-}Node\text{-}at\text{-}pre}(\mathit{self})) =$
$((\tau \models (\mathit{self}\ .\mathit{next@pre} \doteq \mathit{null})) \vee$
$(\ \tau \models (\mathit{self}\ .\mathit{next@pre} <> \mathit{null}) \wedge (\tau \models (\mathit{self}\ .\mathit{next@pre}\ .\mathit{i@pre} \prec \mathit{self}\ .\mathit{i@pre}))$
$\wedge$
$(\tau \models (\mathit{inv\text{-}Node\text{-}at\text{-}pre}(\mathit{self}\ .\mathit{next@pre})))))$

A very first attempt to characterize the axiomatization by an inductive definition - this can not be the last word since too weak (should be equality!)

coinductive $\mathit{inv} :: \mathit{Node} \Rightarrow (\mathfrak{A})\mathit{st} \Rightarrow \mathit{bool}$ **where**
$(\tau \models (\delta\ \mathit{self})) \Longrightarrow ((\tau \models (\mathit{self}\ .\mathit{next} \doteq \mathit{null})) \vee$
$(\tau \models (\mathit{self}\ .\mathit{next} <> \mathit{null}) \wedge (\tau \models (\mathit{self}\ .\mathit{next}\ .\mathit{i} \prec \mathit{self}\ .\mathit{i}))\ \wedge$
$(\ (\mathit{inv}(\mathit{self}\ .\mathit{next}))\tau\ )))$
$\Longrightarrow (\ \mathit{inv}\ \mathit{self}\ \tau)$

5.6. The contract of a recursive query :

The original specification of a recursive query :

```
context Node::contents():Set(Integer)
post:  result = if self.next = null
                then Set{i}
                else self.next.contents()->including(i)
                endif
```

consts $\mathit{dot\text{-}contents} :: \mathit{Node} \Rightarrow \mathit{Set\text{-}Integer}\ \ ((1(\text{-}).\mathit{contents}'(')) \ 50)$

axiomatization $\mathit{dot\text{-}contents\text{-}def}$ **where**
$(\tau \models ((\mathit{self}).\mathit{contents}() \triangleq \mathit{result})) =$
$(\mathit{if}\ \ (\delta\ \mathit{self})\ \tau = \mathit{true}\ \tau$
$\mathit{then}\ ((\tau \models \mathit{true}) \wedge$
$(\tau \models (\mathit{result} \triangleq \mathit{if}\ \ (\mathit{self}\ .\mathit{next} \doteq \mathit{null})$
$\mathit{then}\ (\mathit{Set}\{\mathit{self}\ .\mathit{i}\})$
$\mathit{else}\ (\mathit{self}\ .\mathit{next}\ .\mathit{contents}()\text{−}{>}\mathit{including}(\mathit{self}\ .\mathit{i}))$
$\mathit{endif})))$
$\mathit{else}\ \tau \models \mathit{result} \triangleq \mathit{invalid})$

consts $\mathit{dot\text{-}contents\text{-}AT\text{-}pre} :: \mathit{Node} \Rightarrow \mathit{Set\text{-}Integer}\ \ ((1(\text{-}).\mathit{contents@pre}'(')) \ 50)$

axiomatization where $\mathit{dot\text{-}contents\text{-}AT\text{-}pre\text{-}def}$:
$(\tau \models (\mathit{self}).\mathit{contents@pre}() \triangleq \mathit{result}) =$
$(\mathit{if}\ \ (\delta\ \mathit{self})\ \tau = \mathit{true}\ \tau$
$\mathit{then}\ \tau \models \mathit{true} \wedge \qquad\qquad (*\ \mathit{pre}\ *)$

$$
\begin{array}{l}
\quad\quad \tau \models (result \triangleq if\ \ (self).next@pre \doteq null\ \ (*\ post\ *) \\
\quad\quad\quad\quad\quad\quad\quad then\ Set\{(self).i@pre\} \\
\quad\quad\quad\quad\quad\quad\quad else\ \ (self).next@pre\ .contents@pre()\text{−}{>}including(self\ \ .i@pre) \\
\quad\quad\quad\quad\quad\quad\quad endif) \\
\ \ else\ \tau \models result \triangleq invalid)
\end{array}
$$

Note that these @pre variants on methods are only available on queries, i.e. operations without side-effect.

5.7. The contract of a method.

The specification in high-level OCL input syntax reads as follows:

```
context Node::insert(x:Integer)
post: contents():Set(Integer)
contents() = contents@pre()->including(x)
```

consts *dot-insert* :: *Node* ⇒ *Integer* ⇒ *Void* ((*1*(-).*insert*′(-′)) *50*)

axiomatization where *dot-insert-def*:

$$
\begin{array}{l}
(\tau \models ((self).insert(x) \triangleq result)) = \\
\ (if\ \ (\delta\ self)\ \tau = true\ \tau \land (\upsilon\ x)\ \tau = true\ \tau \\
\ \ then\ \tau \models true \land \\
\quad\quad \tau \models ((self).contents() \triangleq (self).contents@pre()\text{−}{>}including(x)) \\
\ \ else\ \tau \models ((self).insert(x) \triangleq invalid))
\end{array}
$$

end

Part III.

Conclusion

6. Conclusion

6.1. 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.

6.2. 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.