Essential OCL - A Study for a Consistent Semantics of UML/OCL 2.2 in HOL.

Burkhart Wolff

October 9, 2012

Contents

1 OCL Core Definitions

theory
OCL-core
imports
Main
begin

2 Foundational Notations

2.1 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* ($\perp$)

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

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

2.2 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*

2.3 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 un-comparable with the raw-types (consisting of nested option type constructions), such that the data-invariant mussed be expressed in terms of the interface. In a second step, our base-types will be shown to be instances of this interface.

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

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

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

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

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

2.4 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 $bot\text{-}option\text{-}def$: $(bot::'a\ option) \equiv (None::'a\ option)$
  instance $\langle proof \rangle$
end

instantiation $option :: (bot)null$
begin
  definition $null\text{-}option\text{-}def$: $(null::'a::bot\ option) \equiv\ \lfloor\ bot\ \rfloor$
  instance $\langle proof \rangle$
end

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

  instance $\langle proof \rangle$
end

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

 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).

2.5 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* $\Rightarrow$ $'\alpha$

All OCL expressions *denote* functions that map the underlying

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

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* $\equiv \lambda\ \tau.\ bot$

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* $\equiv \lambda x.\ null$.

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

3.1 Basic Constants

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

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

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

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

lemma *bool-split*: $X\ \tau = invalid\ \tau \lor X\ \tau = null\ \tau \lor$

$X\ \tau = true\ \tau \quad \vee X\ \tau = false\ \tau$
⟨*proof*⟩

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

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

The definitions above for the constants *true* and *false* are 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 $Sem :: 'a \Rightarrow 'a\ (I[\![\text{-}]\!])$
where $I[\![x]\!] \equiv x$

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

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

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

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

lemma *textbook-valid*: $I[\![\upsilon(X)]\!]\ \tau = (if\ I[\![X]\!]\ \tau = I[\![bot]\!]\ \tau$

$then\ I[\![false]\!]\ \tau$
$else\ I[\![true]\!]\ \tau)$

$\langle proof \rangle$

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*]: $(X \triangleq X) = true$
$\langle proof \rangle$

lemma *StrongEq-sym* [*simp*]: $(X \triangleq Y) = (Y \triangleq X)$
$\langle proof \rangle$

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

... 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 poststate 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 $(P\ X \triangleq P\ Y)\tau = true\ \tau$
$\langle proof \rangle$

3.4 Fundamental Predicates III

And, last but not least,

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

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.

3.5 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 $not\ X \equiv\ \lambda\ \tau\ .\ case\ X\ \tau\ of$

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

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*] ⇒ ($'\mathfrak{A}$)*Boolean* (**infixl** *and 30*)
where X *and* Y ≡ ($\lambda\, \tau$. *case* $X\ \tau$ *of*

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

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

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

lemma *textbook-not*:

$$
I[\![not(X)]\!]\ \tau = \ (case\ I[\![X]\!]\ \tau\ of\ \ \perp \ \Rightarrow \perp \\
\qquad\qquad \mid\ \lfloor \perp \rfloor \Rightarrow \lfloor \perp \rfloor \\
\qquad\qquad \mid\ \lfloor\lfloor\ x\ \rfloor\rfloor \Rightarrow \lfloor\lfloor\ \neg\ x\ \rfloor\rfloor)
$$

⟨*proof*⟩

lemma *textbook-and*:

$$
I[\![X\ and\ Y]\!]\ \tau = (case\ I[\![X]\!]\ \tau\ of
$$

$$
\begin{array}{ll}
\perp & \Rightarrow (case\ I[\![Y]\!]\ \tau\ of \\
 & \quad \perp \Rightarrow \perp \\
 & \quad \mid \lfloor\perp\rfloor \Rightarrow \perp \\
 & \quad \mid \lfloor\lfloor True\rfloor\rfloor \Rightarrow \perp \\
 & \quad \mid \lfloor\lfloor False\rfloor\rfloor \Rightarrow \lfloor\lfloor False\rfloor\rfloor) \\
\mid \lfloor \perp \rfloor & \Rightarrow (case\ I[\![Y]\!]\ \tau\ of \\
 & \quad \perp \Rightarrow \perp \\
 & \quad \mid \lfloor\perp\rfloor \Rightarrow \lfloor\perp\rfloor \\
 & \quad \mid \lfloor\lfloor True\rfloor\rfloor \Rightarrow \lfloor\perp\rfloor \\
 & \quad \mid \lfloor\lfloor False\rfloor\rfloor \Rightarrow \lfloor\lfloor False\rfloor\rfloor)
\end{array}
$$

$\quad\quad\quad\quad | \lfloor\lfloor True\rfloor\rfloor \Rightarrow (case\ I[\![Y]\!]\ \tau\ of$
$\quad\quad\quad\quad\quad\quad\quad\quad \bot \Rightarrow \bot$
$\quad\quad\quad\quad\quad\quad\quad | \lfloor\bot\rfloor \Rightarrow \lfloor\bot\rfloor$
$\quad\quad\quad\quad\quad\quad\quad | \lfloor\lfloor y\rfloor\rfloor \Rightarrow \lfloor\lfloor y\rfloor\rfloor)$
$\quad\quad\quad\quad | \lfloor\lfloor False\rfloor\rfloor \Rightarrow \lfloor\lfloor False \rfloor\rfloor)$

⟨*proof*⟩

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

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

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

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

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

3.6 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 $\tau \models P \equiv ((P\ \tau) = true\ \tau)$

4 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*⟩

4.0.1 Local Validity and Meta-logic

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

$\langle proof \rangle$

lemma *foundation2*[*simp*]: $\neg(\tau \models \mathit{false})$
$\langle proof \rangle$

lemma *foundation3*[*simp*]: $\neg(\tau \models \mathit{invalid})$
$\langle proof \rangle$

lemma *foundation4*[*simp*]: $\neg(\tau \models \mathit{null})$
$\langle proof \rangle$

lemma *bool-split-local*[*simp*]:
$(\tau \models (x \triangleq \mathit{invalid})) \vee (\tau \models (x \triangleq \mathit{null})) \vee (\tau \models (x \triangleq \mathit{true})) \vee (\tau \models (x \triangleq \mathit{false}))$
$\langle proof \rangle$

lemma *def-split-local*:
$(\tau \models \delta\ x) = ((\neg(\tau \models (x \triangleq \mathit{invalid}))) \wedge (\neg\ (\tau \models (x \triangleq \mathit{null}))))$
$\langle proof \rangle$

lemma *foundation5*:
$\tau \models (P\ \mathit{and}\ Q) \Longrightarrow (\tau \models P) \wedge (\tau \models Q)$
$\langle proof \rangle$

lemma *foundation6*:
$\tau \models P \Longrightarrow \tau \models \delta\ P$
$\langle proof \rangle$

lemma *foundation7*[*simp*]:
$(\tau \models \mathit{not}\ (\delta\ x)) = (\neg\ (\tau \models \delta\ x))$
$\langle proof \rangle$

lemma *foundation7'*[*simp*]:
$(\tau \models \mathit{not}\ (\upsilon\ x)) = (\neg\ (\tau \models \upsilon\ x))$
$\langle proof \rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *defined-not-I* : $\tau \models \delta\ (x) \Longrightarrow \tau \models \delta\ (\mathit{not}\ x)$

$\langle proof \rangle$

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

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

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

5 Local Judgements and Strong Equality

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

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

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

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*: $\bigwedge\ \tau.\ cp\ P \Longrightarrow \tau \models (x \triangleq y) \Longrightarrow \tau \models (P\ x \triangleq P\ y)$
$\langle proof \rangle$

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

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

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

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]]

6 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.

7 Miscellaneous: OCL's if then else endif

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

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

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

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

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

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

end

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

8 Simple, Basic Types like Void, Boolean and Integer

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

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

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

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

9 Strict equalities.

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

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

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

defs *StrictRefEq-int*[*code-unfold*] :

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

defs *StrictRefEq-bool*[*code-unfold*] :

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

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

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

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

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

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

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

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

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

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

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

lemma *strictEqBool-valid-args-valid*:

$(\tau \models \upsilon((x::(\prime\mathfrak{A})Boolean) \doteq y)) = ((\tau \models(\upsilon\ x)) \wedge (\tau \models(\upsilon\ y)))$
⟨*proof*⟩

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

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

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

lemma *StrictRefEq-int-strict″* : $\upsilon\ ((x::(\prime\mathfrak{A})Integer) \doteq y) = (\upsilon(x)\ and\ \upsilon(y))$
⟨*proof*⟩

lemma *StrictRefEq-bool-strict″* : $\upsilon\ ((x::(\prime\mathfrak{A})Boolean) \doteq y) = (\upsilon(x)\ and\ \upsilon(y))$
⟨*proof*⟩

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

lemma *cp-StrictRefEq-int*:
$((X::(\prime\mathfrak{A})Integer) \doteq Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau) \doteq (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
⟨*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* $::(\prime\mathfrak{A})Integer$ ($\mathbf{0}$)
where $\mathbf{0} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 0::int \rfloor\rfloor)$

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

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

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

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

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

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

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

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

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

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

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

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

Elementary computations on Booleans

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

value $\tau_0 \models ((invalid \triangleq false) \triangleq false)$
value $\tau_0 \models ((invalid \doteq false) \triangleq invalid)$

Elementary computations on Integer

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

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

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

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

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

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

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

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

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*]: $(null \doteq \mathbf{1}) = false$
⟨*proof*⟩

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

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

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

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

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

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

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

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

9.1 Example: The Set-Collection Type on the Abstract Interface

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

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

1. we want the types to be *fully abstract*, i.e. the type should not contain junk-elements that are not representable by OCL expressions.

2. We want a possibility to nest collection types (so, we want the potential to talking about $Set(Set(Sequences(Pairs(X, Y))))$), and

The former principe rules out the option to define $'\alpha$ *Set* just by $('\mathfrak{A}, ('\alpha$ *option option*) *set*) *val*. This would allow sets to contain junk elements such as $\{\bot\}$ 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 $'\alpha\ \mathit{Set\text{-}0} = \{X::('\alpha::\mathit{null})\ \mathit{set\ option\ option}.$
$X = \mathit{bot} \lor X = \mathit{null} \lor (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq \mathit{bot})\}$
⟨*proof*⟩

instantiation *Set-0* :: (*null*)*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 ⟨*proof*⟩
end

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

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

instance ⟨*proof*⟩
end

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

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

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

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

... which means that we can have a type $('\mathfrak{A},('\mathfrak{A},('\mathfrak{A})\ \mathit{Integer})\ \mathit{Set})\ \mathit{Set}$ 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.

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

lemma $\mathit{mtSet\text{-}defined}[\mathit{simp},\mathit{code\text{-}unfold}]$:$\delta(\mathit{Set}\{\}) = \mathit{true}$
$\langle \mathit{proof} \rangle$

lemma $\mathit{mtSet\text{-}valid}[\mathit{simp},\mathit{code\text{-}unfold}]$:$\upsilon(\mathit{Set}\{\}) = \mathit{true}$
$\langle \mathit{proof} \rangle$

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

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

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

The key for an operational definition if OclForall given below.

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

definition $\mathit{OclSize}\ \ \ \ :: ('\mathfrak{A},'\alpha::\mathit{null})\mathit{Set} \Rightarrow\ '\mathfrak{A}\ \mathit{Integer}$

where $\mathit{OclSize}\ x = (\lambda\ \tau.\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau \wedge \mathit{finite}(\lceil\lceil \mathit{Rep\text{-}Set\text{-}0}\ (x\ \tau)\rceil\rceil)$
$\mathit{then}\ \lfloor\lfloor\ \mathit{int}(\mathit{card}\ \lceil\lceil \mathit{Rep\text{-}Set\text{-}0}\ (x\ \tau)\rceil\rceil)\ \rfloor\rfloor$
$\mathit{else}\ \bot\ )$

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

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

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

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

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

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

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

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

syntax
$\text{-}\mathit{OclForall} :: [('\mathfrak{A},'\alpha::\mathit{null})\ \mathit{Set},\mathit{id},('\mathfrak{A})\mathit{Boolean}] \Rightarrow '\mathfrak{A}\ \mathit{Boolean}\quad ((\text{-}){-}{>}\mathit{forall}'(\text{-}|\text{-}'))$

translations
$X\text{−>}forall(x \mid P)$ == *CONST OclForall X* (%x. P)

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

consts

OclUnion :: [($'\mathfrak{A}$,$'\alpha$::*null*) *Set*,($'\mathfrak{A}$,$'\alpha$) *Set*] ⇒ ($'\mathfrak{A}$,$'\alpha$) *Set*
OclIntersection:: [($'\mathfrak{A}$,$'\alpha$::*null*) *Set*,($'\mathfrak{A}$,$'\alpha$) *Set*] ⇒ ($'\mathfrak{A}$,$'\alpha$) *Set*
OclIncludesAll :: [($'\mathfrak{A}$,$'\alpha$::*null*) *Set*,($'\mathfrak{A}$,$'\alpha$) *Set*] ⇒ $'\mathfrak{A}$ *Boolean*
OclExcludesAll :: [($'\mathfrak{A}$,$'\alpha$::*null*) *Set*,($'\mathfrak{A}$,$'\alpha$) *Set*] ⇒ $'\mathfrak{A}$ *Boolean*
OclComplement :: ($'\mathfrak{A}$,$'\alpha$::*null*) *Set* ⇒ ($'\mathfrak{A}$,$'\alpha$) *Set*
OclSum :: ($'\mathfrak{A}$,$'\alpha$::*null*) *Set* ⇒ $'\mathfrak{A}$ *Integer*
OclCount :: [($'\mathfrak{A}$,$'\alpha$::*null*) *Set*,($'\mathfrak{A}$,$'\alpha$) *Set*] ⇒ $'\mathfrak{A}$ *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 (-−>*notEmpty*′(′) [*66*])
and
OclIncluding (-−>*including*′(-′))
and
OclExcluding (-−>*excluding*′(-′))
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*⟩

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

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

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

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

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

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

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

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

⟨*proof*⟩

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

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

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

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

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

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

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

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

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

9.2 Some computational laws:

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

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

lemma *including-charn1*:
assumes *def-X*:$\tau \models (\delta\ X)$
assumes *val-x*:$\tau \models (\upsilon\ x)$
shows $\tau \models (X{-}{>}including(x){-}{>}includes(x))$
⟨*proof*⟩

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 $\tau \models (X{-}{>}including(x){-}{>}includes(y)) \triangleq (X{-}{>}includes(y))$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

find-theorems *name*:*corev* -

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

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

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

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

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

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

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

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

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

syntax
-OclIterate :: $[('\mathfrak{A},'\alpha::null)\ Set,\ idt,\ idt,\ '\alpha,\ '\beta] => ('\mathfrak{A},'\gamma)val$
$(\text{-} -{>}iterate'(\text{-};\text{-}{=}\text{-} \mid \text{-}')\ [71,100,70]50)$
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 *short-cut*[*simp*]: $x \models \delta\ S{-}{>}size()$
⟨proof⟩

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

lemma [*simp*]: $\upsilon\ \mathbf{6} = true$ ⟨proof⟩
lemma [*simp*]: $\upsilon\ \mathbf{8} = true$ ⟨proof⟩
lemma [*simp*]: $\upsilon\ \mathbf{9} = true$ ⟨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 $\tau \models ((Set\{\mathbf{2},\mathbf{1}\}){-}{>}forall(z \mid \mathbf{0} \prec z))$
value $\neg\ (\tau \models ((Set\{\mathbf{2},\mathbf{1}\}){-}{>}exists(z \mid z \prec \mathbf{0}\ )))$

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

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

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

end

10 OCL State Operations

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

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

10.2 Referential Object Equality in States

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

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

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

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

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

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

lemma *cp-gen-ref-eq-object*:
$(\textit{gen-ref-eq}\ x\ y\ \tau) = (\textit{gen-ref-eq}\ (\lambda\text{-}.\ x\ \tau)\ (\lambda\text{-}.\ y\ \tau))\ \tau$
⟨*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*:
$\tau \models (\textit{gen-ref-eq}\ x\ (y{::}('\mathfrak{A},'a{::}\{\textit{null},\textit{object}\})\textit{val})) \Longrightarrow (\tau \models (\delta\ x)) \land (\tau \models (\delta\ y))$
⟨*proof*⟩

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

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

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

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

11 Miscillaneous: Initial States (for Testing and Code Generation)

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

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

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

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

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

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

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-x*: $\tau \models \delta(x \text{ @}pre\ H)$
  and *H-is-typerepr*: *inj H*
  shows $\tau \models (x \triangleq (x \text{ @}pre\ H))$
⟨*proof*⟩

end

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

end

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

end

12 OCL Data Universes: Generic Definition and an Example

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

12.1 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 [?]. 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.

12.2 Outlining the Example

12.3 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 $node = \; mk_{node} \;\; oid$
$\qquad\qquad\qquad int\ option$
$\qquad\qquad\qquad oid\ option$

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

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 $\mathfrak{A} = in_{node}\ node \mid in_{object}\ object$

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 $\quad Node \equiv (the\text{-}inv\ in_{node})$

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

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

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*: $oid\text{-}of\ x = (case\ x\ of\ mk_{node}\ oid\ \text{-}\ \text{-} \Rightarrow oid)$
  instance $\langle proof \rangle$
end

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

instantiation $\mathfrak{A}$ :: *object*
begin
  definition *oid-of-*$\mathfrak{A}$*-def*: $oid\text{-}of\ x = (case\ x\ of$
$in_{node}\ node \Rightarrow oid\text{-}of\ node$
$\mid in_{object}\ obj \Rightarrow oid\text{-}of\ obj)$
  instance $\langle proof \rangle$
end

instantiation *option* :: (*object*)*object*
begin
  definition *oid-of-option-def*: $oid\text{-}of\ x = oid\text{-}of\ (the\ x)$
  instance $\langle proof \rangle$
end

13 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* $x{::}Node\ y{::}Node\ \tau$,
    simplified $StrictRefEq_{node}$[*symmetric*]]
  cp-intro(*9*) [*of* $P{::}Node \Rightarrow NodeQ{::}Node \Rightarrow Node$,
    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

$[of\ x{::}Node,$
$simplified\ StrictRefEq_{node}[symmetric]]$
$gen\text{-}ref\text{-}eq\text{-}object\text{-}strict2$
$[of\ x{::}Node,$
$simplified\ StrictRefEq_{node}[symmetric]]$
$gen\text{-}ref\text{-}eq\text{-}object\text{-}strict3$
$[of\ x{::}Node,$
$simplified\ StrictRefEq_{node}[symmetric]]$
$gen\text{-}ref\text{-}eq\text{-}object\text{-}strict3$
$[of\ x{::}Node,$
$simplified\ StrictRefEq_{node}[symmetric]]$
$gen\text{-}ref\text{-}eq\text{-}object\text{-}strict4$
$[of\ x{::}Node,$
$simplified\ StrictRefEq_{node}[symmetric]]$

13.1 AllInstances

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

lemma $(Object\ .oclAllInstances@pre()) =$
$(\lambda\tau.\ \ Abs\text{-}Set\text{-}0\ \lfloor\lfloor(Some \circ Some \circ (the\text{-}inv\ in_{object}))`(ran(fst\ \tau))\ \rfloor\rfloor)$
$\langle proof \rangle$

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.

14 Selector Definition

Should be generated entirely from a class-diagram.

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

| ⌊ - ⌋⇒ *invalid* τ)

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

fun *dot-next-at-pre*:: *Node* ⇒ *Node* ((*1*(-).*next@pre*) *50*)
  where $(X).next@pre = (\lambda\ \tau.\ case\ X\ \tau\ of$
  $\bot \Rightarrow invalid\ \tau$
  $|\ \lfloor\ \bot\ \rfloor \Rightarrow invalid\ \tau$
  $|\ \lfloor\lfloor\ mk_{node}\ oid\ i\ \bot\ \rfloor\rfloor \Rightarrow null\ \tau$(∗ *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.* ∗)
  $(case\ (fst\ \tau)\ next\ of$
  $\bot \Rightarrow invalid\ \tau$
  $|\ \lfloor in_{node}\ (mk_{node}\ a\ b\ c)\rfloor \Rightarrow \lfloor\lfloor mk_{node}\ a\ b\ c\ \rfloor\rfloor$
  $|\ \lfloor\ \text{-}\ \rfloor \Rightarrow invalid\ \tau))$

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

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 λ *X* -. *X* λ - τ. τ, *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*⟩

14.1 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* τ *of*
$\bot$ $\Rightarrow$ *invalid* τ
| $\lfloor\bot\rfloor$ $\Rightarrow$ *invalid* τ (* *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* τ *of*
$\bot$ $\Rightarrow$ *invalid* τ
| $\lfloor\bot\rfloor$ $\Rightarrow$ *invalid* τ (* *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* τ *of*
$\bot$ $\Rightarrow$ *invalid* τ
| $\lfloor\bot\rfloor$ $\Rightarrow$ *invalid* τ

$| \lfloor\lfloor mk_{object}\ oid\ \bot\ \rfloor\rfloor \Rightarrow\ invalid\ \tau$ (* *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*:
$(X{::}Node)\ .oclAsType(Node) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\bot\ \ \Rightarrow invalid\ \tau$
$| \lfloor\bot\rfloor \Rightarrow invalid\ \tau$ (* *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*⟩

15 Tests for Actual Types

consts $oclistypeof_{object}$ :: $'\alpha \Rightarrow$ *Boolean* ((-).*oclIsTypeOf*'(*Object*'))
consts $oclistypeof_{node}$ :: $'\alpha \Rightarrow$ *Boolean* ((-).*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*
$\perp \Rightarrow$ *invalid* τ
$\mid \lfloor\perp\rfloor \Rightarrow$ *invalid* τ
$\mid \lfloor\lfloor - \rfloor\rfloor \Rightarrow$ *true* $\tau)$

16 Standard State Infrastructure

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

17 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* $\Rightarrow$ *Boolean*
where $A : (\tau \models (\delta\ self)) \longrightarrow$
$(\tau \models inv\text{-}Node(self)) =$
$((\tau \models (self\ .next \doteq null)) \vee$
$(\ \tau \models (self\ .next <> null) \wedge (\tau \models (self\ .next\ .i \prec self\ .i))\ \wedge$
$(\tau \models (inv\text{-}Node(self\ .next)))))$

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

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

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

axiomatization *dot-contents-def* **where**

$$\begin{array}{l}
(\tau \models ((self).contents() \triangleq result)) = \\
\ (if\ (\delta\ self)\ \tau = true\ \tau \\
\quad then\ ((\tau \models true) \wedge \\
\qquad\quad (\tau \models (result \triangleq if\ (self\ .next \doteq null) \\
\qquad\qquad\qquad\qquad then\ (Set\{self\ .i\}) \\
\qquad\qquad\qquad\qquad else\ (self\ .next\ .contents() {-}{>} including(self\ .i)) \\
\qquad\qquad\qquad\qquad endif))) \\
\quad else\ \tau \models result \triangleq invalid)
\end{array}$$

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

axiomatization where *dot-contents-AT-pre-def*:

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

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

19 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* $\Rightarrow$ *Integer* $\Rightarrow$ *Void* $((1(\text{-}).insert'(\text{-}')) \ 50)$

axiomatization where *dot-insert-def*:

$$\begin{array}{l}
(\tau \models (self).insert(x) \triangleq result) = \\
\ (if\ (\delta\ self)\ \tau = true\ \tau \wedge (\upsilon\ x)\ \tau = true\ \tau \\
\quad then\ \tau \models true\ \wedge \\
\qquad\quad \tau \models (self).contents() \triangleq (self).contents@pre() {-}{>} including(x)
\end{array}$$

$else\ \tau \models (self).insert(x) \triangleq invalid)$

lemma $H : (\tau \models (self).insert(x) \triangleq result)$
nitpick
thm *dot-insert-def*
$\langle proof \rangle$

end