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

Burkhart Wolff

July 27, 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* ($\bot$)

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

2.2 State, State Transitions, Well-formed 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 $('\mathfrak{A})st = '\mathfrak{A}\ state \times '\mathfrak{A}\ state$

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-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*

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

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

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

instance *option* :: (*plus*) *plus* **by** *intro-classes*
instance *fun* :: (*type*, *plus*) *plus* **by** *intro-classes*

class *bot* =
  fixes *bot* :: $'a$

assumes *nonEmpty* : $\exists\ x.\ x \neq bot$

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

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-option-def*: $(bot{::}'a\ option) \equiv (None{::}'a\ option)$
instance proof show $\exists\, x{::}'a\ option.\ x \neq bot$
by(*rule-tac* $x{=}Some\ x$ **in** *exI*, *simp add*:*bot-option-def*)
qed
end

instantiation *option* :: (*bot*)*null*
begin
definition *null-option-def*: $(null{::}'a{::}bot\ option) \equiv\ \lfloor\ bot\ \rfloor$
instance proof show $(null{::}'a{::}bot\ option) \neq bot$
by(*simp add*:*null-option-def bot-option-def*)
qed
end

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

instance proof show $\exists (x{::}'a \Rightarrow 'b).\ x \neq bot$
apply(*rule-tac* $x{=}\lambda$ -. $(SOME\ y.\ y \neq bot)$ **in** *exI*, *auto*)
apply(*drule-tac* $x{=}x$ **in** *fun-cong*,*auto simp*:*bot-fun-def*)
apply(*erule contrapos-pp*, *simp*)
apply(*rule some-eq-ex*[*THEN iffD2*])
apply(*simp add*: *nonEmpty*)
done
qed
end

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

instance proof
show $(null{::}'a \Rightarrow 'b{::}null) \neq bot$
apply(*auto simp*: *null-fun-def bot-fun-def*)
apply(*drule-tac x=x* **in** *fun-cong*)
apply(*erule contrapos-pp*, *simp add*: *null-is-valid*)
done
qed
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 $null \equiv \lambda x.\ null$.

2.6 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 $WFF\ \tau = ((\forall\ x \in dom(fst\ \tau).\ x = oid\text{-}of(the(fst\ \tau\ x))) \wedge$
$(\forall\ x \in dom(snd\ \tau).\ x = oid\text{-}of(the(snd\ \tau\ x))))$

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.

3 The OCL Base Type Boolean.

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

4.1 Basic Constants

lemma *bot-Boolean-def* : $(bot::('\mathfrak{A})Boolean) = (\lambda\ \tau.\ \bot)$
by(*simp add*: *bot-fun-def bot-option-def*)

lemma *null-Boolean-def* : $(null::('\mathfrak{A})Boolean) = (\lambda\ \tau.\ \lfloor\bot\rfloor)$
by(*simp add*: *null-fun-def null-option-def bot-option-def*)

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

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

lemma *bool-split*: $X\ \tau = invalid\ \tau \vee X\ \tau = null\ \tau \vee$
$X\ \tau = true\ \tau \quad \vee X\ \tau = false\ \tau$
apply(*simp add*: *invalid-def null-def true-def false-def*)
apply(*case-tac* $X\ \tau$,*simp-all add*: *null-fun-def null-option-def bot-option-def*)
apply(*case-tac a*,*simp*)
apply(*case-tac aa*,*simp*)
apply *auto*
done

lemma [*simp*]: *false* $(a,\ b) = \lfloor\lfloor False \rfloor\rfloor$
by(*simp add:false-def*)

lemma [*simp*]: *true* $(a,\ b) = \lfloor\lfloor True \rfloor\rfloor$
by(*simp add:true-def*)

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

lemma *valid1*[*simp*]: υ *invalid* = *false*
by(*rule ext,simp add: valid-def bot-fun-def bot-option-def invalid-def true-def false-def*)

lemma *valid2*[*simp*]: υ *null* = *true*
by(*rule ext,simp add: valid-def bot-fun-def bot-option-def null-is-valid null-fun-def invalid-def true-def false-def*)

lemma *valid3*[*simp*]: υ *true* = *true*
by(*rule ext,simp add: valid-def bot-fun-def bot-option-def null-is-valid null-fun-def invalid-def true-def false-def*)

lemma *valid4*[*simp*]: υ *false* = *true*
by(*rule ext,simp add: valid-def bot-fun-def bot-option-def null-is-valid null-fun-def invalid-def true-def false-def*)

lemma *cp-valid*: $(\upsilon\ X)\ \tau = (\upsilon\ (\lambda\ \text{-}.\ X\ \tau))\ \tau$
by(*simp add: valid-def*)

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

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

lemma *defined1*[*simp*]: δ *invalid* = *false*
by(*rule ext,simp add: defined-def bot-fun-def bot-option-def null-def invalid-def true-def false-def*)

lemma *defined2*[*simp*]: δ *null* = *false*
by(*rule ext,simp add: defined-def bot-fun-def bot-option-def null-def null-option-def null-fun-def invalid-def true-def false-def*)

lemma *defined3*[*simp*]: δ *true* = *true*
by(*rule ext*,*simp add*: *defined-def bot-fun-def bot-option-def null-is-valid null-option-def null-fun-def invalid-def true-def false-def*)

lemma *defined4*[*simp*]: δ *false* = *true*
by(*rule ext*,*simp add*: *defined-def bot-fun-def bot-option-def null-is-valid null-option-def null-fun-def invalid-def true-def false-def*)

lemma *defined5*[*simp*]: $\delta\ \delta\ X$ = *true*
by(*rule ext*,*auto simp*: *defined-def true-def false-def bot-fun-def bot-option-def null-option-def null-fun-def*)

lemma *defined6*[*simp*]: $\delta\ \upsilon\ X$ = *true*
by(*rule ext*,
auto simp: *valid-def defined-def true-def false-def bot-fun-def bot-option-def null-option-def null-fun-def*)

lemma *defined7*[*simp*]: $\delta\ \delta\ X$ = *true*
by(*rule ext*,
auto simp: *valid-def defined-def true-def false-def bot-fun-def bot-option-def null-option-def null-fun-def*)

lemma *valid6*[*simp*]: $\upsilon\ \delta\ X$ = *true*
by(*rule ext*,
auto simp: *valid-def defined-def true-def false-def bot-fun-def bot-option-def null-option-def null-fun-def*)

lemma *cp-defined*:$(\delta\ X)\tau = (\delta\ (\lambda\ \text{-}.\ X\ \tau))\ \tau$
by(*simp add*: *defined-def*)

4.3 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*

by(*rule ext*, *simp add*: *null-def invalid-def true-def false-def StrongEq-def*)

lemma *StrongEq-sym* [*simp*]: $(X \triangleq Y) = (Y \triangleq X)$
by(*rule ext*, *simp add*: *eq-sym-conv invalid-def true-def false-def StrongEq-def*)

lemma *StrongEq-trans-strong* [*simp*]:
  assumes *A*: $(X \triangleq Y) = \mathit{true}$
  and *B*: $(Y \triangleq Z) = \mathit{true}$
  shows $(X \triangleq Z) = \mathit{true}$
  apply(*insert A B*) **apply**(*rule ext*)
  apply(*simp add*: *null-def invalid-def true-def false-def StrongEq-def*)
  apply(*drule-tac x=x* **in** *fun-cong*)+
  by *auto*

... 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*: $(X \triangleq Y)\tau = \mathit{true}\ \tau$
  shows $(P\ X \triangleq P\ Y)\tau = \mathit{true}\ \tau$
  apply(*insert cp eq*)
  apply(*simp add*: *null-def invalid-def true-def false-def StrongEq-def*)
  apply(*subst cp*[*of X*])
  apply(*subst cp*[*of Y*])
  by *simp*

4.4 Fundamental Predicates III: (Generic) Referential Strict Equality

Construction by overloading: for each base type, there is an equality.

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

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

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

lemma *gen-ref-eq-object-strict1* [*simp*] :
(*gen-ref-eq x invalid*) = *invalid*
by(*rule ext*, *simp add*: *gen-ref-eq-def true-def false-def*)

lemma *gen-ref-eq-object-strict2* [*simp*] :
(*gen-ref-eq invalid x*) = *invalid*
by(*rule ext*, *simp add*: *gen-ref-eq-def true-def false-def*)

lemma *gen-ref-eq-object-strict3* [*simp*] :
(*gen-ref-eq x null*) = *invalid*
by(*rule ext*, *simp add*: *gen-ref-eq-def true-def false-def*)

lemma *gen-ref-eq-object-strict4* [*simp*] :
(*gen-ref-eq null x*) = *invalid*
by(*rule ext*, *simp add*: *gen-ref-eq-def true-def false-def*)

lemma *cp-gen-ref-eq-object*:
(*gen-ref-eq x y* τ) = (*gen-ref-eq* (λ-. *x* τ) (λ-. *y* τ)) τ
by(*auto simp*: *gen-ref-eq-def StrongEq-def invalid-def cp-defined*[*symmetric*])

And, last but not least,

lemma *defined8*[*simp*]: δ ($X \triangleq Y$) = *true*
by(*rule ext*,
auto simp: *valid-def defined-def true-def false-def StrongEq-def*
bot-fun-def bot-option-def null-option-def null-fun-def)

lemma *valid5*[*simp*]: υ ($X \triangleq Y$) = *true*
by(*rule ext*,
auto simp: *valid-def true-def false-def StrongEq-def*
bot-fun-def bot-option-def null-option-def null-fun-def)

lemma *cp-StrongEq*: ($X \triangleq Y$) τ = ((λ -. X τ) $\triangleq$ (λ -. Y τ)) τ
by(*simp add*: *StrongEq-def*)

4.5 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 $\quad not\ X \equiv\ \lambda\ \tau\ .\ case\ X\ \tau\ of$

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

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

lemma *cp-not*: $(not\ X)\tau = (not\ (\lambda\ \text{-}.\ X\ \tau))\ \tau$
by(*simp add*: *not-def*)

lemma *not1* [*simp*]: *not invalid* = *invalid*
by(*rule ext*,*simp add*: *not-def null-def invalid-def true-def false-def bot-option-def*)

lemma *not2* [*simp*]: *not null* = *null*
by(*rule ext*,*simp add*: *not-def null-def invalid-def true-def false-def*
bot-option-def null-fun-def null-option-def)

lemma *not3* [*simp*]: *not true* = *false*
by(*rule ext*,*simp add*: *not-def null-def invalid-def true-def false-def*)

lemma *not4* [*simp*]: *not false* = *true*
by(*rule ext*,*simp add*: *not-def null-def invalid-def true-def false-def*)

lemma *not-not*[*simp*]: *not* (*not* X) = X
apply(*rule ext*,*simp add*: *not-def null-def invalid-def true-def false-def*)
apply(*case-tac X x*, *simp-all*)
apply(*case-tac a*, *simp-all*)
done

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

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

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

$| \lfloor \perp \rfloor \Rightarrow (case\ Y\ \tau\ of$
$\quad\quad \perp \Rightarrow\ \perp$
$\quad\quad | \lfloor\perp\rfloor \Rightarrow \lfloor\perp\rfloor$
$\quad\quad | \lfloor\lfloor True\rfloor\rfloor \Rightarrow \lfloor\perp\rfloor$
$\quad\quad | \lfloor\lfloor False\rfloor\rfloor \Rightarrow\ \lfloor\lfloor False\rfloor\rfloor)$
$| \lfloor\lfloor True\rfloor\rfloor \Rightarrow (case\ Y\ \tau\ of$
$\quad\quad \perp \Rightarrow\ \perp$
$\quad\quad | \lfloor\perp\rfloor \Rightarrow \lfloor\perp\rfloor$
$\quad\quad | \lfloor\lfloor y\rfloor\rfloor \Rightarrow\ \lfloor\lfloor y\rfloor\rfloor)$
$| \lfloor\lfloor False\rfloor\rfloor \Rightarrow\ \lfloor\lfloor\ False\ \rfloor\rfloor)$

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*) τ = ((λ -. *X* τ) *and* (λ -. *Y* τ)) τ
by(*simp add*: *ocl-and-def*)

lemma *cp-ocl-or*:((*X*::('$\mathfrak{A}$)*Boolean*) *or Y*) τ = ((λ -. *X* τ) *or* (λ -. *Y* τ)) τ
apply(*simp add*: *ocl-or-def*)
apply(*subst cp-not*[*of not* (λ-. *X* τ) *and not* (λ-. *Y* τ)])
apply(*subst cp-ocl-and*[*of not* (λ-. *X* τ) *not* (λ-. *Y* τ)])
by(*simp add*: *cp-not*[*symmetric*] *cp-ocl-and*[*symmetric*])

lemma *cp-ocl-implies*:(*X implies Y*) τ = ((λ -. *X* τ) *implies* (λ -. *Y* τ)) τ
apply(*simp add*: *ocl-implies-def*)
apply(*subst cp-ocl-or*[*of not* (λ-. *X* τ) (λ-. *Y* τ)])
by(*simp add*: *cp-not*[*symmetric*] *cp-ocl-or*[*symmetric*])

lemma *ocl-and1*[*simp*]: (*invalid and true*) = *invalid*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-def bot-option-def*)
lemma *ocl-and2*[*simp*]: (*invalid and false*) = *false*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-def bot-option-def*)
lemma *ocl-and3*[*simp*]: (*invalid and null*) = *invalid*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-def bot-option-def*

null-fun-def null-option-def)
lemma *ocl-and4*[*simp*]: (*invalid and invalid*) = *invalid*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-def bot-option-def*)

lemma *ocl-and5*[*simp*]: (*null and true*) = *null*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-def bot-option-def*
null-fun-def null-option-def)

lemma *ocl-and6*[*simp*]: (*null and false*) = *false*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-def bot-option-def null-fun-def null-option-def*)
lemma *ocl-and7*[*simp*]: (*null and null*) = *null*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-def bot-option-def null-fun-def null-option-def*)
lemma *ocl-and8*[*simp*]: (*null and invalid*) = *invalid*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-def bot-option-def null-fun-def null-option-def*)

lemma *ocl-and9*[*simp*]: (*false and true*) = *false*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-def*)
lemma *ocl-and10*[*simp*]: (*false and false*) = *false*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-def*)
lemma *ocl-and11*[*simp*]: (*false and null*) = *false*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-def*)
lemma *ocl-and12*[*simp*]: (*false and invalid*) = *false*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-def*)

lemma *ocl-and13*[*simp*]: (*true and true*) = *true*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-def*)
lemma *ocl-and14*[*simp*]: (*true and false*) = *false*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-def*)
lemma *ocl-and15*[*simp*]: (*true and null*) = *null*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-def bot-option-def null-fun-def null-option-def*)
lemma *ocl-and16*[*simp*]: (*true and invalid*) = *invalid*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-def bot-option-def null-fun-def null-option-def*)

lemma *ocl-and-idem*[*simp*]: (*X and X*) = *X*
apply(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-def*)
apply(*case-tac X x*, *simp-all*)
apply(*case-tac a*, *simp-all*)
apply(*case-tac aa*, *simp-all*)
done

lemma *ocl-and-commute*: (*X and Y*) = (*Y and X*)
by(*rule ext*,*auto simp*:*true-def false-def ocl-and-def invalid-def*
split: *option.split option.split-asm*
bool.split bool.split-asm)

lemma *ocl-and-false1*[*simp*]: (*false and X*) = *false*
apply(*rule ext*, *simp add*: *ocl-and-def*)
apply(*auto simp*:*true-def false-def invalid-def*
split: *option.split option.split-asm*)
done

lemma *ocl-and-false2*[*simp*]: (*X and false*) = *false*
  by(*simp add*: *ocl-and-commute*)

lemma *ocl-and-true1*[*simp*]: (*true and X*) = *X*
  apply(*rule ext*, *simp add*: *ocl-and-def*)
  apply(*auto simp*:*true-def false-def invalid-def*
    split: *option.split option.split-asm*)
  done

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

lemma *ocl-and-assoc*: (*X and* (*Y and Z*)) = (*X and Y and Z*)
  apply(*rule ext*, *simp add*: *ocl-and-def*)
  apply(*auto simp*:*true-def false-def null-def invalid-def*
    split: *option.split option.split-asm*
      bool.split bool.split-asm)
done

lemma *ocl-or-idem*[*simp*]: (*X or X*) = *X*
  by(*simp add*: *ocl-or-def*)

lemma *ocl-or-commute*: (*X or Y*) = (*Y or X*)
  by(*simp add*: *ocl-or-def ocl-and-commute*)

lemma *ocl-or-false1*[*simp*]: (*false or Y*) = *Y*
  by(*simp add*: *ocl-or-def*)

lemma *ocl-or-false2*[*simp*]: (*Y or false*) = *Y*
  by(*simp add*: *ocl-or-def*)

lemma *ocl-or-true1*[*simp*]: (*true or Y*) = *true*
  by(*simp add*: *ocl-or-def*)

lemma *ocl-or-true2*: (*Y or true*) = *true*
  by(*simp add*: *ocl-or-def*)

lemma *ocl-or-assoc*: (*X or* (*Y or Z*)) = (*X or Y or Z*)
  by(*simp add*: *ocl-or-def ocl-and-assoc*)

lemma *deMorgan1*: *not*(*X and Y*) = ((*not X*) *or* (*not Y*))
  by(*simp add*: *ocl-or-def*)

lemma *deMorgan2*: *not*(*X or Y*) = ((*not X*) *and* (*not Y*))
  by(*simp add*: *ocl-or-def*)

4.6 A Standard Logical Calculus for OCL

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

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

5 Global vs. Local Judgements

lemma *transform1*: $P = true \Longrightarrow \tau \models P$
by(*simp add*: *OclValid-def*)

lemma *transform2*: $(P = Q) \Longrightarrow ((\tau \models P) = (\tau \models Q))$
by(*auto simp*: *OclValid-def*)

lemma *transform2-rev*: $\forall\ \tau.\ (\tau \models \delta\ P) \wedge (\tau \models \delta\ Q) \wedge (\tau \models P) = (\tau \models Q) \Longrightarrow$
$P = Q$
apply(*rule ext*,*auto simp*: *OclValid-def true-def defined-def*)
apply(*erule-tac x=a* **in** *allE*)
apply(*erule-tac x=b* **in** *allE*)
apply(*auto simp*: *false-def true-def defined-def bot-Boolean-def null-Boolean-def*
split: *option.split-asm HOL.split-if-asm*)
done

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 $\tau \models P \Longrightarrow \tau \models Q$
apply(*simp add*: *OclValid-def*)
apply(*rule H*[*THEN fun-cong*])
apply(*rule ext*)
oops

5.0.1 Local Validity and Meta-logic

lemma *foundation1*[*simp*]: $\tau \models true$
by(*auto simp*: *OclValid-def*)

lemma *foundation2*[*simp*]: $\neg(\tau \models false)$
by(*auto simp*: *OclValid-def true-def false-def*)

lemma *foundation3*[*simp*]: $\neg(\tau \models invalid)$
by(*auto simp*: *OclValid-def true-def false-def invalid-def bot-option-def*)

lemma *foundation4*[*simp*]: $\neg(\tau \models null)$
by(*auto simp*: *OclValid-def true-def false-def null-def null-fun-def null-option-def bot-option-def*)

lemma *bool-split-local*[*simp*]:
$(\tau \models (x \triangleq \mathit{invalid})) \lor (\tau \models (x \triangleq \mathit{null})) \lor (\tau \models (x \triangleq \mathit{true})) \lor (\tau \models (x \triangleq \mathit{false}))$
apply(*insert bool-split*[*of x τ*], *auto*)
apply(*simp-all add*: *OclValid-def StrongEq-def true-def null-def invalid-def*)
done

lemma *def-split-local*:
$(\tau \models \delta\ x) = ((\neg(\tau \models (x \triangleq \mathit{invalid}))) \land (\neg\ (\tau \models (x \triangleq \mathit{null}))))$
by(*simp add*:*defined-def true-def false-def invalid-def null-def*
StrongEq-def OclValid-def bot-fun-def null-fun-def)

lemma *foundation5*:
$\tau \models (P\ \mathit{and}\ Q) \Longrightarrow (\tau \models P) \land (\tau \models Q)$
by(*simp add*: *ocl-and-def OclValid-def true-def false-def defined-def*
split: *option.split option.split-asm bool.split bool.split-asm*)

lemma *foundation6*:
$\tau \models P \Longrightarrow \tau \models \delta\ P$
by(*simp add*: *not-def OclValid-def true-def false-def defined-def*
null-option-def null-fun-def bot-option-def bot-fun-def
split: *option.split option.split-asm*)

lemma *foundation7*[*simp*]:
$(\tau \models \mathit{not}\ (\delta\ x)) = (\neg\ (\tau \models \delta\ x))$
by(*simp add*: *not-def OclValid-def true-def false-def defined-def*
split: *option.split option.split-asm*)

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*:
$(\tau \models \delta\ x) \lor (\tau \models (x \triangleq \mathit{invalid})) \lor (\tau \models (x \triangleq \mathit{null}))$
proof −
  have *1* : $(\tau \models \delta\ x) \lor (\neg(\tau \models \delta\ x))$ **by** *auto*
  have *2* : $(\neg(\tau \models \delta\ x)) = ((\tau \models (x \triangleq \mathit{invalid})) \lor (\tau \models (x \triangleq \mathit{null})))$
    by(*simp only*: *def-split-local*, *simp*)
  show *?thesis* **by**(*insert 1*, *simp add*:*2*)
qed

lemma *foundation9*:
$\tau \models \delta\ x \Longrightarrow (\tau \models \mathit{not}\ x) = (\neg\ (\tau \models x))$
apply(*simp add*: *def-split-local*)
by(*auto simp*: *not-def null-fun-def null-option-def bot-option-def*
OclValid-def invalid-def true-def null-def StrongEq-def)

lemma *foundation10*:
$\tau \models \delta\ x \Longrightarrow \tau \models \delta\ y \Longrightarrow (\tau \models (x\ \mathit{and}\ y)) = (\ (\tau \models x) \wedge (\tau \models y))$
apply(*simp add*: *def-split-local*)
by(*auto simp*: *ocl-and-def OclValid-def invalid-def*
true-def null-def StrongEq-def null-fun-def null-option-def bot-option-def
split:*bool.split-asm*)

lemma *foundation11*:
$\tau \models \delta\ x \Longrightarrow\ \ \tau \models \delta\ y \Longrightarrow (\tau \models (x\ \mathit{or}\ y)) = (\ (\tau \models x) \vee (\tau \models y))$
apply(*simp add*: *def-split-local*)
by(*auto simp*: *not-def ocl-or-def ocl-and-def OclValid-def invalid-def*
true-def null-def StrongEq-def null-fun-def null-option-def bot-option-def
split:*bool.split-asm bool.split*)

lemma *foundation12*:
$\tau \models \delta\ x \Longrightarrow\ \ \tau \models \delta\ y \Longrightarrow (\tau \models (x\ \mathit{implies}\ y)) = (\ (\tau \models x) \longrightarrow (\tau \models y))$
apply(*simp add*: *def-split-local*)
by(*auto simp*: *not-def ocl-or-def ocl-and-def ocl-implies-def bot-option-def*
OclValid-def invalid-def true-def null-def StrongEq-def null-fun-def
null-option-def
split:*bool.split-asm bool.split*)

lemma *foundation13*:$(\tau \models A \triangleq \mathit{true}) \quad = (\tau \models A)$
by(*auto simp*: *not-def OclValid-def invalid-def true-def null-def StrongEq-def*
split:*bool.split-asm bool.split*)

lemma *foundation14*:$(\tau \models A \triangleq \mathit{false}) \quad = (\tau \models \mathit{not}\ A)$
by(*auto simp*: *not-def OclValid-def invalid-def false-def true-def null-def StrongEq-def*

split:*bool.split-asm bool.split option.split*)

lemma *foundation15*:$(\tau \models A \triangleq \mathit{invalid}) = (\tau \models \mathit{not}(\upsilon\ A))$
by(*auto simp*: *not-def OclValid-def valid-def invalid-def false-def true-def null-def*

StrongEq-def bot-option-def null-fun-def null-option-def bot-option-def
bot-fun-def
split:*bool.split-asm bool.split option.split*)

lemma *foundation16*: $\tau \models (\delta\ X) = (X\ \tau \neq \mathit{bot} \wedge X\ \tau \neq \mathit{null})$
by(*auto simp*: *OclValid-def defined-def false-def true-def bot-fun-def null-fun-def*
split:*split-if-asm*)

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

lemma *foundation18*: $\tau \models (\upsilon\ X) = (X\ \tau \neq bot)$
by(*auto simp*: *OclValid-def valid-def false-def true-def bot-fun-def*
split:*split-if-asm*)

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

lemma *foundation20* : $\tau \models (\delta\ X) \Longrightarrow \tau \models \upsilon\ X$
by(*simp add*: *foundation18 foundation16*)

theorem *strictEqGen-vs-strongEq*:
$WFF\ \tau \Longrightarrow \tau \models(\delta\ x) \Longrightarrow \tau \models(\delta\ y) \Longrightarrow$
$(\tau \models (gen\text{-}ref\text{-}eq\ x\ y)) = (\tau \models (x \triangleq y))$
apply(*auto simp*: *gen-ref-eq-def OclValid-def WFF-def StrongEq-def true-def*)
sorry

WFF and ref_eq must be defined strong enough defined that this can be proven!

6 Local Judgements and Strong Equality

lemma *StrongEq-L-refl*: $\tau \models (x \triangleq x)$
by(*simp add*: *OclValid-def StrongEq-def*)

lemma *StrongEq-L-sym*: $\tau \models (x \triangleq y) \Longrightarrow \tau \models (y \triangleq x)$
by(*simp add*: *OclValid-def StrongEq-def*)

lemma *StrongEq-L-trans*: $\tau \models (x \triangleq y) \Longrightarrow \tau \models (y \triangleq z) \Longrightarrow \tau \models (x \triangleq z)$
by(*simp add*: *OclValid-def StrongEq-def true-def*)

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*: $\bigwedge \tau.\ cp\ P \Longrightarrow \tau \models (x \triangleq y) \Longrightarrow \tau \models (P\ x \triangleq P\ y)$
by(*auto simp*: *OclValid-def StrongEq-def true-def cp-def*)

lemma *StrongEq-L-subst2*:

$\bigwedge \tau.\ \ cp\ P \Longrightarrow \tau \models (x \triangleq y) \Longrightarrow \tau \models (P\ x) \Longrightarrow \tau \models (P\ y)$
by(*auto simp*: *OclValid-def StrongEq-def true-def cp-def*)

lemma *cpI1*:
$(\forall\ \ X\ \tau.\ f\ X\ \tau = f(\lambda\text{-}.\ X\ \tau)\ \tau) \Longrightarrow cp\ P \Longrightarrow cp(\lambda X.\ f\ \ (P\ X))$
apply(*auto simp*: *true-def cp-def*)
apply(*rule exI*, (*rule allI*)+)
by(*erule-tac x*=*P X* **in** *allE*, *auto*)

lemma *cpI2*:
$(\forall\ \ X\ Y\ \tau.\ f\ X\ Y\ \tau = f(\lambda\text{-}.\ X\ \tau)(\lambda\text{-}.\ Y\ \tau)\ \tau) \Longrightarrow$
$cp\ P \Longrightarrow cp\ Q \Longrightarrow cp(\lambda X.\ f\ \ (P\ X)\ (Q\ X))$
apply(*auto simp*: *true-def cp-def*)
apply(*rule exI*, (*rule allI*)+)
by(*erule-tac x*=*P X* **in** *allE*, *auto*)

lemma *cp-const* : $cp(\lambda\text{-}.\ c)$
by (*simp add*: *cp-def*, *fast*)

lemma *cp-id* : $cp(\lambda X.\ X)$
by (*simp add*: *cp-def*, *fast*)

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]]
cp-gen-ref-eq-object[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]],
of gen-ref-eq]]

7 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*:
$\tau \models (not\ P) \Longrightarrow \tau \models \delta\ P$
by(*auto simp*: *not-def OclValid-def true-def invalid-def defined-def false-def*
bot-fun-def bot-option-def null-fun-def null-option-def
split: *bool.split-asm HOL.split-if-asm option.split option.split-asm*)

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

8 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 *if-ocl-invalid* : $(if\ invalid\ then\ B_1\ else\ B_2\ endif) = invalid$
by(*rule ext*, *auto simp*: *if-ocl-def*)

lemma *if-ocl-null* : $(if\ null\ then\ B_1\ else\ B_2\ endif) = invalid$
by(*rule ext*, *auto simp*: *if-ocl-def*)

lemma *if-ocl-true* : $(if\ true\ then\ B_1\ else\ B_2\ endif) = B_1$
by(*rule ext*, *auto simp*: *if-ocl-def*)

lemma *if-ocl-false* : $(if\ false\ then\ B_1\ else\ B_2\ endif) = B_2$
by(*rule ext*, *auto simp*: *if-ocl-def*)

end

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

9 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.

10 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.

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

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

lemma *StrictRefEq-int-strict1*[*simp*] : $((x{::}('\mathfrak{A})\mathit{Integer}) \doteq \mathit{invalid}) = \mathit{invalid}$
by(*rule ext*, *simp add*: *StrictRefEq-int true-def false-def*)

lemma *StrictRefEq-int-strict2*[*simp*] : $(\mathit{invalid} \doteq (x{::}('\mathfrak{A})\mathit{Integer})) = \mathit{invalid}$
by(*rule ext*, *simp add*: *StrictRefEq-int true-def false-def*)

lemma *strictEqBool-vs-strongEq*:
$\tau \models (\upsilon\ x) \Longrightarrow \tau \models (\upsilon\ y) \Longrightarrow (\tau \models ((x{::}('\mathfrak{A})\mathit{Boolean}) \doteq y)) = (\tau \models (x \triangleq y))$
by(*simp add*: *StrictRefEq-bool OclValid-def*)

lemma *strictEqInt-vs-strongEq*:
$\tau \models (\upsilon\ x) \Longrightarrow \tau \models (\upsilon\ y) \Longrightarrow (\tau \models ((x{::}('\mathfrak{A})\mathit{Integer}) \doteq y)) = (\tau \models (x \triangleq y))$
by(*simp add*: *StrictRefEq-int OclValid-def*)

lemma *strictEqBool-defargs*:
$\tau \models ((x{::}('\mathfrak{A})\mathit{Boolean}) \doteq y) \Longrightarrow (\tau \models (\upsilon\ x)) \wedge (\tau \models (\upsilon\ y))$
by(*simp add*: *StrictRefEq-bool OclValid-def true-def invalid-def*
bot-option-def
split: *bool.split-asm HOL.split-if-asm*)

lemma *strictEqInt-defargs*:
$\tau \models ((x{::}('\mathfrak{A})\mathit{Integer}) \doteq y) \Longrightarrow (\tau \models (\upsilon\ x)) \wedge (\tau \models (\upsilon\ y))$
by(*simp add*: *StrictRefEq-int OclValid-def true-def invalid-def valid-def bot-option-def*
split: *bool.split-asm HOL.split-if-asm*)

lemma *strictEqBool-valid-args-valid*:
$(\tau \models \upsilon((x{::}('\mathfrak{A})\mathit{Boolean}) \doteq y)) = ((\tau \models (\upsilon\ x)) \wedge (\tau \models (\upsilon\ y)))$
by(*auto simp*: *StrictRefEq-bool OclValid-def true-def valid-def false-def StrongEq-def*

defined-def invalid-def valid-def bot-option-def bot-fun-def
split: *bool.split-asm HOL.split-if-asm option.split*)

lemma *strictEqInt-valid-args-valid*:
$(\tau \models \upsilon((x::('\mathfrak{A})Integer) \doteq y)) = ((\tau \models(\upsilon\ x)) \wedge (\tau \models(\upsilon\ y)))$
by(*auto simp*: *StrictRefEq-int OclValid-def true-def valid-def false-def StrongEq-def*

defined-def invalid-def bot-fun-def bot-option-def
split: *bool.split-asm HOL.split-if-asm option.split*)

lemma *gen-ref-eq-defargs*:
$\tau \models (\textit{gen-ref-eq}\ x\ (y::('\mathfrak{A},'a::\{null,object\})val)) \Longrightarrow (\tau \models(\delta\ x)) \wedge (\tau \models(\delta\ y))$
by(*simp add*: *gen-ref-eq-def OclValid-def true-def invalid-def*
defined-def invalid-def bot-fun-def bot-option-def
split: *bool.split-asm HOL.split-if-asm*)

lemma *StrictRefEq-int-strict* :
assumes A: $\upsilon\ (x::('\mathfrak{A})Integer) = true$
and B: $\upsilon\ y = true$
shows $\upsilon\ (x \doteq y) = true$
apply(*insert A B*)
apply(*rule ext*, *simp add*: *StrongEq-def StrictRefEq-int true-def valid-def defined-def*
bot-fun-def bot-option-def)
done

lemma *StrictRefEq-int-strict'* :
assumes A: $\upsilon\ (((x::('\mathfrak{A})Integer)) \doteq y) = true$
shows $\upsilon\ x = true \wedge \upsilon\ y = true$
apply(*insert A*, *rule conjI*)
apply(*rule ext*, *drule-tac x=xa* **in** *fun-cong*)
prefer *2*
apply(*rule ext*, *drule-tac x=xa* **in** *fun-cong*)
apply(*simp-all add*: *StrongEq-def StrictRefEq-int*
false-def true-def valid-def defined-def)
apply(*case-tac y xa*, *auto*)
apply(*simp-all add*: *true-def invalid-def bot-fun-def*)
done

lemma *StrictRefEq-bool-strict1*[*simp*] : $((x::('\mathfrak{A})Boolean) \doteq invalid) = invalid$
by(*rule ext*, *simp add*: *StrictRefEq-bool true-def false-def*)

lemma *StrictRefEq-bool-strict2*[*simp*] : $(invalid \doteq (x::('\mathfrak{A})Boolean)) = invalid$
by(*rule ext*, *simp add*: *StrictRefEq-bool true-def false-def*)

lemma *cp-StrictRefEq-bool*:

$((X::('\mathfrak{A})Boolean) \doteq Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau) \doteq (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
by(*auto simp*: *StrictRefEq-bool StrongEq-def defined-def valid-def cp-defined*[*symmetric*])

lemma *cp-StrictRefEq-int*:
$((X::('\mathfrak{A})Integer) \doteq Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau) \doteq (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
by(*auto simp*: *StrictRefEq-int StrongEq-def valid-def cp-defined*[*symmetric*])

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*]]

lemma *StrictRefEq-strict* :
assumes *A*: $\upsilon\ (x::('\mathfrak{A})Integer) = true$
and *B*: $\upsilon\ y = true$
shows $\upsilon\ (x \doteq y) = true$
apply(*insert A B*)
apply(*rule ext*, *simp add*: *StrongEq-def StrictRefEq-int true-def valid-def bot-fun-def bot-option-def*)
done

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

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

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

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

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

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

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

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

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

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

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

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

lemma δ *null* = *false* **by** *simp*
lemma υ *null* = *true* **by** *simp*

lemma [*simp*]:δ **0** = *true*
by(*simp add*:*ocl-zero-def defined-def true-def*
bot-fun-def bot-option-def null-fun-def null-option-def)

lemma [*simp*]:υ **0** = *true*
by(*simp add*:*ocl-zero-def valid-def true-def*
bot-fun-def bot-option-def null-fun-def null-option-def)

lemma [*simp*]:δ **1** = *true*
by(*simp add*:*ocl-one-def defined-def true-def*
bot-fun-def bot-option-def null-fun-def null-option-def)

lemma [*simp*]:υ **1** = *true*
by(*simp add*:*ocl-one-def valid-def true-def*
bot-fun-def bot-option-def null-fun-def null-option-def)

lemma [*simp*]:δ **2** = *true*
by(*simp add*:*ocl-two-def defined-def true-def*
bot-fun-def bot-option-def null-fun-def null-option-def)

lemma [*simp*]:υ **2** = *true*
by(*simp add*:*ocl-two-def valid-def true-def*
bot-fun-def bot-option-def null-fun-def null-option-def)

lemma *one-non-null*[*simp*]: **0** $\neq$ *null*
apply(*auto simp*:*ocl-zero-def null-def*)
apply(*drule-tac x*=*x* **in** *fun-cong*,
simp add:*null-fun-def null-option-def bot-option-def*)
done

lemma *zero-non-null*[*simp*]: **1** $\neq$ *null*
apply(*auto simp*:*ocl-one-def null-def*)
apply(*drule-tac x*=*x* **in** *fun-cong*,

simp add:null-fun-def null-option-def bot-option-def)
done

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).

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

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

10.1 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 $\{\perp\}$ which we need to identify with undefinedness itself. Abandoning fully abstractness of rules would later on produce all sorts of problems when quantifying over the elements of a type. However, if we build an own type, then it must conform to our abstract interface in order to have nested types: arguments of type-constructors must conform to our abstract interface, and the result type too.

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

typedef $'\alpha$ *Set-0* = $\{X::('\alpha::null)\ set\ option\ option.$
$X = bot \vee X = null \vee (\forall x\in\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
by (*rule-tac x=bot* **in** *exI*, *simp*)

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

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

instance proof show $\exists\, x::'a\ Set\text{-}0.\ x \neq bot$
apply(*rule-tac x=Abs-Set-0* $\lfloor None \rfloor$ **in** *exI*)
apply(*simp add:bot-Set-0-def*)
apply(*subst Abs-Set-0-inject*)
apply(*simp-all add: Set-0-def bot-Set-0-def null-option-def bot-option-def*)
done
qed
end

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

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

instance proof show $(null::('a::null)\ Set\text{-}0) \neq bot$
apply(*simp add:null-Set-0-def bot-Set-0-def*)
apply(*subst Abs-Set-0-inject*)
apply(*simp-all add: Set-0-def bot-Set-0-def null-option-def bot-option-def*)
done
qed
end

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

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

lemma *Set-inv-lemma*: $\tau \models (\delta\ X) \Longrightarrow (X\ \tau = Abs\text{-}Set\text{-}0\ \lfloor bot \rfloor) \lor (\forall\, x{\in}\lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil.\ x \neq bot)$
apply(*insert OCL-lib.Set-0.Rep-Set-0 [of X* τ], *simp add:Set-0-def*)
apply(*auto simp: OclValid-def defined-def false-def true-def cp-def bot-fun-def bot-Set-0-def null-Set-0-def null-fun-def split:split-if-asm*)
apply(*erule contrapos-pp [of Rep-Set-0* $(X\ \tau) = bot$])
apply(*subst Abs-Set-0-inject[symmetric], simp add:Rep-Set-0*)
apply(*simp add: Set-0-def*)
apply(*simp add: Rep-Set-0-inverse bot-Set-0-def bot-option-def*)
apply(*erule contrapos-pp [of Rep-Set-0* $(X\ \tau) = null$])
apply(*subst Abs-Set-0-inject[symmetric], simp add:Rep-Set-0*)
apply(*simp add: Set-0-def*)
apply(*simp add: Rep-Set-0-inverse null-option-def*)
done

... 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.

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

lemma $mtSet\text{-}defined[simp]$:$\delta(Set\{\}) = true$
apply(*rule ext*, *auto simp*: *mtSet-def defined-def null-Set-0-def*
bot-Set-0-def bot-fun-def null-fun-def)
apply(*simp-all add*: *Abs-Set-0-inject Set-0-def bot-option-def null-Set-0-def null-option-def*)
done

lemma $mtSet\text{-}valid[simp]$:$\upsilon(Set\{\}) = true$
apply(*rule ext*,*auto simp*: *mtSet-def valid-def null-Set-0-def*
bot-Set-0-def bot-fun-def null-fun-def)
apply(*simp-all add*: *Abs-Set-0-inject Set-0-def bot-option-def null-Set-0-def null-option-def*)
done

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

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

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

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

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

definition $OclExcludes$ :: $[('\mathfrak{A},'\alpha$::$null)\ Set,('\mathfrak{A},'\alpha)\ val] \Rightarrow\ '\mathfrak{A}\ Boolean$

where $\quad$ *OclExcludes x y* = (*not*(*OclIncludes x y*))

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

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

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

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

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

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

consts

OclUnion :: [('$\mathfrak{A}$,'α::*null*) *Set*,('$\mathfrak{A}$,'α) *Set*] $\Rightarrow$ ('$\mathfrak{A}$,'α) *Set*
OclIntersection:: [('$\mathfrak{A}$,'α::*null*) *Set*,('$\mathfrak{A}$,'α) *Set*] $\Rightarrow$ ('$\mathfrak{A}$,'α) *Set*
OclIncludesAll :: [('$\mathfrak{A}$,'α::*null*) *Set*,('$\mathfrak{A}$,'α) *Set*] $\Rightarrow$ '$\mathfrak{A}$ *Boolean*
OclExcludesAll :: [('$\mathfrak{A}$,'α::*null*) *Set*,('$\mathfrak{A}$,'α) *Set*] $\Rightarrow$ '$\mathfrak{A}$ *Boolean*
OclComplement :: ('$\mathfrak{A}$,'α::*null*) *Set* $\Rightarrow$ ('$\mathfrak{A}$,'α) *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 (-->notEmpty'(') [66])
and
OclIncluding (-->including'(-'))
and
OclExcluding (-->excluding'(-'))
and
OclComplement (-->complement'('))
and
OclUnion (-->union'(-') [66,65]65)
and
OclIntersection(-->intersection'(-') [71,70]70)

lemma *cp-OclIncluding*:
$(X{-}{>}including(x))\ \tau = ((\lambda\ \text{-}.\ X\ \tau){-}{>}including(\lambda\ \text{-}.\ x\ \tau))\ \tau$
by(*auto simp*: *OclIncluding-def StrongEq-def invalid-def*
cp-defined[*symmetric*] *cp-valid*[*symmetric*])

lemma *cp-OclExcluding*:
$(X{-}{>}excluding(x))\ \tau = ((\lambda\ \text{-}.\ X\ \tau){-}{>}excluding(\lambda\ \text{-}.\ x\ \tau))\ \tau$
by(*auto simp*: *OclExcluding-def StrongEq-def invalid-def*
cp-defined[*symmetric*] *cp-valid*[*symmetric*])

lemma *cp-OclIncludes*:
$(X{-}{>}includes(x))\ \tau = (OclIncludes\ (\lambda\ \text{-}.\ X\ \tau)\ (\lambda\ \text{-}.\ x\ \tau)\ \tau)$
by(*auto simp*: *OclIncludes-def StrongEq-def invalid-def*
cp-defined[*symmetric*] *cp-valid*[*symmetric*])

lemma *including-strict1*[*simp*]:$(\bot\text{->}including(x)) = \bot$
by(*simp add*: *bot-fun-def OclIncluding-def defined-def valid-def false-def true-def*)

lemma *including-strict2*[*simp*]:$(X\text{->}including(\bot)) = \bot$
by(*simp add*: *OclIncluding-def bot-fun-def defined-def valid-def false-def true-def*)

lemma *including-strict3*[*simp*]:$(null\text{->}including(x)) = \bot$
by(*simp add*: *OclIncluding-def bot-fun-def defined-def valid-def false-def true-def*)

lemma *including-valid-args-valid*:
$(\tau \models \delta(X\text{->}including(x))) = ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
proof −
 have $A : \bot \in$ *Set-0* **by**(*simp add*: *Set-0-def bot-option-def*)
 have $B : \lfloor\bot\rfloor \in$ *Set-0* **by**(*simp add*: *Set-0-def null-option-def bot-option-def*)
 have $C : (\tau \models(\delta\ X)) \Longrightarrow (\tau \models(\upsilon\ x)) \Longrightarrow \lfloor\lfloor insert\ (x\ \tau)\ \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil\rfloor\rfloor$
$\in$ *Set-0*
  apply(*frule Set-inv-lemma*)
  apply(*simp add*: *Set-0-def bot-option-def null-Set-0-def null-fun-def*
   foundation18 foundation16)
  done
 have D: $(\tau \models \delta(X\text{->}including(x))) \Longrightarrow ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
  by(*auto simp*: *OclIncluding-def OclValid-def true-def valid-def false-def*
StrongEq-def
   defined-def invalid-def bot-fun-def null-fun-def
  split: *bool.split-asm HOL.split-if-asm option.split*)
 have E: $(\tau \models(\delta\ X)) \Longrightarrow (\tau \models(\upsilon\ x)) \Longrightarrow (\tau \models \delta(X\text{->}including(x)))$
  apply(*frule C*, *simp*)
  apply(*auto simp*: *OclIncluding-def OclValid-def true-def valid-def false-def*
StrongEq-def
   defined-def invalid-def valid-def bot-fun-def null-fun-def
  split: *bool.split-asm HOL.split-if-asm option.split*)
  apply(*simp-all add*: *null-Set-0-def bot-Set-0-def bot-option-def*)
  apply(*simp-all add*: *Abs-Set-0-inject A B bot-option-def*[*symmetric*],
   simp-all add: *bot-option-def*)
  done
show *?thesis* **by**(*auto dest*:*D intro*:*E*)
qed

lemma *excluding-strict1*[*simp*]:$(\bot\text{->}excluding(x)) = \bot$
by(*simp add*: *bot-fun-def OclExcluding-def defined-def valid-def false-def true-def*)

lemma *excluding-strict2*[*simp*]:$(X\text{->}excluding(\bot)) = \bot$
by(*simp add*: *OclExcluding-def bot-fun-def defined-def valid-def false-def true-def*)

lemma *excluding-strict3*[*simp*]:$(null{-}{>}excluding(x)) = \bot$
by(*simp add*: *OclExcluding-def bot-fun-def defined-def valid-def false-def true-def*)

10.2 Some computational laws:

lemma *including-charn0*[*simp*]:
assumes *val-x*:$\tau \models (\upsilon\ x)$
shows $\tau \models not(Set\{\}{-}{>}includes(x))$
using *val-x*
apply(*auto simp*: *OclValid-def OclIncludes-def not-def false-def true-def*)
apply(*auto simp*: *mtSet-def OCL-lib.Set-0.Abs-Set-0-inverse Set-0-def*)
done

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 −
have $A : \bot \in$ *Set-0* **by**(*simp add*: *Set-0-def bot-option-def*)
have $B : \lfloor\bot\rfloor \in$ *Set-0* **by**(*simp add*: *Set-0-def null-option-def bot-option-def*)
have $C : \lfloor\lfloor insert\ (x\ \tau)\ \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil\rfloor\rfloor \in$ *Set-0*
apply(*insert def-X*[*THEN foundation17*] *val-x*[*THEN foundation19*] *Set-inv-lemma*[*OF def-X*])
apply(*simp add*: *Set-0-def bot-option-def null-Set-0-def null-fun-def*)
done
show *?thesis*
apply(*insert def-X*[*THEN foundation17*] *val-x*[*THEN foundation19*])
apply(*auto simp*: *OclValid-def bot-fun-def OclIncluding-def OclIncludes-def false-def true-def*
defined-def valid-def bot-Set-0-def null-fun-def null-Set-0-def bot-option-def)
apply(*simp-all add*: *Abs-Set-0-inject A B C bot-option-def*[*symmetric*],
simp-all add: *bot-option-def Abs-Set-0-inverse C*)
done
qed

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 −

have A : $\bot \in$ *Set-0* **by**(*simp add*: *Set-0-def bot-option-def*)
have B : $\lfloor\bot\rfloor \in$ *Set-0* **by**(*simp add*: *Set-0-def null-option-def bot-option-def*)
have C : $\lfloor\lfloor$*insert* (x τ) $\lceil\lceil$*Rep-Set-0* (X τ)$\rceil\rceil\rfloor\rfloor \in$ *Set-0*
apply(*insert def-X*[*THEN foundation17*] *val-x*[*THEN foundation19*]
Set-inv-lemma[*OF def-X*])
apply(*simp add*: *Set-0-def bot-option-def null-Set-0-def null-fun-def*)
done
have D : $y\ \tau \neq x\ \tau$
apply(*insert neq*)
by(*auto simp*: *OclValid-def bot-fun-def OclIncluding-def OclIncludes-def*
false-def true-def defined-def valid-def bot-Set-0-def
null-fun-def null-Set-0-def StrongEq-def not-def)
show *?thesis*
apply(*insert def-X*[*THEN foundation17*] *val-x*[*THEN foundation19*])
apply(*auto simp*: *OclValid-def bot-fun-def OclIncluding-def OclIncludes-def false-def*
true-def
defined-def valid-def bot-Set-0-def null-fun-def null-Set-0-def
StrongEq-def)
apply(*simp-all add*: *Abs-Set-0-inject Abs-Set-0-inverse A B C D*)
apply(*simp-all add*: *Abs-Set-0-inject A B C bot-option-def*[*symmetric*],
simp-all add: *bot-option-def Abs-Set-0-inverse C*)
done
qed

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**))
by *simp*

lemma *semantic-test*: $\tau \models$ (*Set*{**2**,*null*}−>*includes*(*null*))
oops

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*))
oops

lemma *hurx* : $\tau \models$ *Set*{*Set*{**2**},*null*} $\triangleq$ *Set*{*null*,*Set*{**2**}}
oops

lemma *semantic-test*: $\tau \models$ (*Set*{*null*,**2**}−>*includes*(*null*))
by(*simp-all add*: *including-charn1 including-valid-args-valid*)

find-theorems *fold*
term *comp-fun-commute*
term *undefined*
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 $OclIterate_{Set}\ S\ A\ F = (\lambda\ \tau.\ if\ (\delta\ S)\ \tau = true\ \tau \wedge (\upsilon\ A)\ \tau = true\ \tau \wedge$
$finite\lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil$
$then\ (fold\ F\ A\ ((\lambda a\ \tau.\ a)\ `\ \lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil))\tau$
$else\ \bot)$

lemma $OclIterate_{Set}$*-strict1*[*simp*]:$(OclIterate_{Set}\ \bot\ A\ F) = \bot$
by(*simp add*: *bot-fun-def* $OclIterate_{Set}$*-def defined-def valid-def false-def true-def*)

lemma $OclIterate_{Set}$*-null1*[*simp*]:$(OclIterate_{Set}\ null\ A\ F) = \bot$
by(*simp add*: *bot-fun-def* $OclIterate_{Set}$*-def defined-def valid-def false-def true-def*)

lemma $OclIterate_{Set}$*-strict2*[*simp*]:$(OclIterate_{Set}\ S\ \bot\ F) = \bot$
by(*simp add*: *bot-fun-def* $OclIterate_{Set}$*-def defined-def valid-def false-def true-def*)

An open question is this ...

lemma $OclIterate_{Set}$*-null2*[*simp*]:$(OclIterate_{Set}\ S\ null\ F) = \bot$
oops

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 $(OclIterate_{Set}\ S\ A\ F) = \bot$
oops

lemma $OclIterate_{Set}$*-empty*:
assumes *non-finite*: $\tau \models \delta(S{-}{>}size())$
shows $(OclIterate_{Set}\ (Set\{\})\ A\ F) = A$
oops

In particular, this does hold for A = null.

lemma $OclIterate_{Set}$*-including*:
assumes *S-finite*: $\tau \models \delta(S{-}{>}size())$
and *F-strict1*:$\bigwedge x.\ \tau \models (F\ x\ \bot \triangleq \bot)$
and *F-strict2*:$\bigwedge x.\ \tau \models (F\ \bot\ x \triangleq \bot)$
and *F-commute*:$\bigwedge x\ y.\ F\ y \circ F\ x = F\ x \circ F\ y$
and *F-cp*: $\bigwedge x\ y\ \tau.\ F\ x\ y\ \tau = F\ (\lambda\ \text{-}.\ x\ \tau)\ (\lambda\ \text{-}.\ y\ \tau)\ \tau$
shows $(OclIterate_{Set}\ (S{-}{>}including(a))\ A\ F) = F\ a\ (OclIterate_{Set}\ (S{-}{>}excluding(a))$
$A\ F)$

oops

end

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

end

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

end

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

11 Example Data-Universe

Should be generated entirely from a class-diagram.

Our data universe consists in the concrete class diagram just of node's.

datatype *node* = *Node oid*
int
oid

type-synonym *Boolean* = (*node*)*Boolean*
type-synonym *Integer* = (*node*)*Integer*
type-synonym *Void* = (*node*)*Void*
type-synonym *Node* = (*node*,*node option option*)*val*
type-synonym *Set-Integer* = (*node*,*int option option*)*Set*

instantiation *node* :: *object*
begin
  definition *oid-of-def*: *oid-of x* = (*case x of Node oid - -* ⇒ *oid*)
  instance ..
end

instantiation *option* :: (*object*)*object*
begin
  definition *oid-of-option-def*: *oid-of x* = *oid-of* (*the x*)
  instance ..

end

12 Instantiation of the generic strict equality

Should be generated entirely from a class-diagram.

defs *StrictRefEq-node* : $(x{::}Node) \doteq y \equiv$ *gen-ref-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*]]

13 Selector Definition

Should be generated entirely from a class-diagram.

fun *dot-next*:: *Node* ⇒ *Node* ((*1*(-).*next*) *50*)
where (*X*).*next* = (λ τ. *case X* τ *of*
None ⇒ *None*
| ⌊ *None* ⌋ ⇒ *None*
| ⌊⌊ *Node oid i next* ⌋⌋ ⇒ *if next* ∈ *dom* (*snd* τ)
then ⌊ (*snd* τ) *next* ⌋
else None)

fun *dot-i*:: *Node* ⇒ *Integer* ((*1*(-).*i*) *50*)
where (*X*).*i* = (λ τ. *case X* τ *of*
None ⇒ *None*
| ⌊ *None* ⌋ ⇒ *None*

| ⌊⌊ *Node oid i next* ⌋⌋ ⇒
if oid ∈ *dom* (*snd* τ)
then (*case* (*snd* τ) *oid of*
None ⇒ *None*
| ⌊ *Node oid i next* ⌋ ⇒ ⌊⌊ *i* ⌋⌋)
else None)

fun *dot-next-at-pre*:: *Node* ⇒ *Node* ((*1*(-).*next@pre*) *50*)
where (*X*).*next@pre* = (λ τ. *case X* τ *of*
None ⇒ *None*
| ⌊ *None* ⌋ ⇒ *None*
| ⌊⌊*Node oid i next*⌋⌋ ⇒ *if next* ∈ *dom* (*fst* τ)
then ⌊ (*fst* τ) *next* ⌋
else None)

fun *dot-i-at-pre*:: *Node* ⇒ *Integer* ((*1*(-).*i@pre*) *50*)
where (*X*).*i@pre* = (λ τ. *case X* τ *of*
None ⇒ *None*
| ⌊ *None* ⌋ ⇒ *None*
| ⌊⌊ *Node oid i next* ⌋⌋ ⇒
if oid ∈ *dom* (*fst* τ)
then (*case* (*fst* τ) *oid of*
None ⇒ *None*
| ⌊ *Node oid i next* ⌋ ⇒ ⌊⌊ *i* ⌋⌋)
else None)

lemma *cp-dot-next*:
((*X*).*next*) τ = ((λ-. *X* τ).*next*) τ **by**(*simp*)

lemma *cp-dot-i*:
((*X*).*i*) τ = ((λ-. *X* τ).*i*) τ **by**(*simp*)

lemma *cp-dot-next-at-pre*:
((*X*).*next@pre*) τ = ((λ-. *X* τ).*next@pre*) τ **by**(*simp*)

lemma *cp-dot-i-pre*:
((*X*).*i@pre*) τ = ((λ-. *X* τ).*i@pre*) τ **by**(*simp*)

lemmas *cp-dot-nextI* [*simp*, *intro*!]=
cp-dot-next[*THEN allI*[*THEN allI*], *of* λ *X* -. *X* λ - τ. τ, *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*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*)

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

by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*)

lemma *dot-next-strict*[*simp*] : (*invalid*).*next* = *invalid*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*)

lemma *dot-nextATpre-strict*[*simp*] : (*invalid*).*next*@*pre* = *invalid*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*)

14 Standard State Infrastructure

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

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

axiomatization *inv-Node-at-pre* :: *Node* ⇒ *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* ⇒ (*node*)*st* ⇒ *bool* **where**
$(\tau \models (\delta\ self)) \Longrightarrow ((\tau \models (self\ .next \doteq null)) \vee$
$(\tau \models (self\ .next <> null) \wedge (\tau \models (self\ .next\ .i \prec self\ .i)) \wedge$
$((inv(self\ .next))\tau)))$
$\Longrightarrow (inv\ self\ \tau)$

16 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*(-).*contents'*(')) *50*)

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

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

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

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

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

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

17 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*(-).*insert'*(-')) *50*)

axiomatization where *dot-insert-def*:

$(\tau \models (self).insert(x) \triangleq result) =$
$\quad (if\ (\delta\ self)\ \tau = true\ \tau \wedge (\upsilon\ x)\ \tau = true\ \tau$

$\quad$ *then* $\tau \models true \wedge$

$\qquad \tau \models (self).contents() \triangleq (self).contents@pre()\text{−>}including(x)$

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

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

end