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

Burkhart Wolff

July 22, 2011

Contents

theory
OCL-core
imports
Main
begin

1 OCL Core Definitions

1.1 Foundational Notations

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:

syntax
  lift $:: {}'\alpha \Rightarrow {}'\alpha$ *option* $(\lfloor(\text{-})\rfloor)$
translations
  $\lfloor a \rfloor$ == *CONST Some a*

syntax
  bottom $:: {}'\alpha$ *option* $(\bot)$
translations
  $\bot$ == *CONST None*

fun *drop* $:: {}'\alpha$ *option* $\Rightarrow {}'\alpha$ $(\lceil(\text{-})\rceil)$
where *drop* (*Some v*) = *v*

1.2 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*

All OCL expressions *denote* functions that map the underlying

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

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 $\mathit{WFF} :: ('\mathfrak{A}::\mathit{object})\mathit{st} \Rightarrow \mathit{bool}$
where $\mathit{WFF}\ \tau = ((\forall\ x \in \mathit{dom}(\mathit{fst}\ \tau).\ x = \mathit{oid\text{-}of}(\mathit{the}(\mathit{fst}\ \tau\ x))) \wedge$
$(\forall\ x \in \mathit{dom}(\mathit{snd}\ \tau).\ x = \mathit{oid\text{-}of}(\mathit{the}(\mathit{snd}\ \tau\ x))))$

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.

1.3 Basic Constants

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

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

1.4 Boolean Type and Logic

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

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

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

lemma *bool-split*: $X\ \tau = \mathit{invalid}\ \tau \vee X\ \tau = \mathit{null}\ \tau \vee$
$X\ \tau = \mathit{true}\ \tau \vee X\ \tau = \mathit{false}\ \tau$
⟨*proof*⟩

thm *bool-split*

lemma [*simp*]: $\mathit{false}\ (a,\ b) = \lfloor\lfloor \mathit{False} \rfloor\rfloor$

$\langle proof \rangle$

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

2 Logical (Strong) Equality and Definedness

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

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

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 $(X \triangleq Z) = true$
  $\langle proof \rangle$

definition *valid* :: $('\mathfrak{A},'a)val \Rightarrow ('\mathfrak{A})Boolean$ (υ - [*100*]*100*)
where $\upsilon\ X \equiv \lambda\ \tau\ .\ case\ X\ \tau\ of$

```
                 $\bot$          $\Rightarrow false\ \tau$
               | $\lfloor \bot \rfloor$      $\Rightarrow true\ \tau$
               | $\lfloor\lfloor x \rfloor\rfloor$    $\Rightarrow true\ \tau$
```

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

lemma *valid1*[*simp*]: $\upsilon\ invalid = false$
  $\langle proof \rangle$

lemma *valid2*[*simp*]: $\upsilon\ null = true$
  $\langle proof \rangle$

lemma *valid3*[*simp*]: $\upsilon\ \upsilon\ X = true$
  $\langle proof \rangle$

definition *defined* :: $('\mathfrak{A},'a)val \Rightarrow ('\mathfrak{A})Boolean$ (δ - [*100*]*100*)
where $\delta\ X \equiv \lambda\ \tau\ .\ case\ X\ \tau\ of$

```
                 $\bot$          $\Rightarrow false\ \tau$
               | $\lfloor \bot \rfloor$      $\Rightarrow false\ \tau$
               | $\lfloor\lfloor x \rfloor\rfloor$    $\Rightarrow true\ \tau$
```

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

lemma *defined1*[*simp*]: δ *invalid* = *false*
⟨*proof*⟩

lemma *defined2*[*simp*]: δ *null* = *false*
⟨*proof*⟩

lemma *defined3*[*simp*]: $\delta\ \delta\ X$ = *true*
⟨*proof*⟩

lemma *valid4*[*simp*]: $\upsilon\ (X \triangleq Y)$ = *true*
⟨*proof*⟩

lemma *defined4*[*simp*]: $\delta\ (X \triangleq Y)$ = *true*
⟨*proof*⟩

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

lemma *valid5*[*simp*]: $\upsilon\ \delta\ X$ = *true*
⟨*proof*⟩

3 Logical Connectives and their Universal Properties

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

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

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

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

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

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

lemma *not4*[*simp*]: *not false* = *true*

⟨*proof*⟩

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

definition *ocl-and* :: $[('\mathfrak{A})Boolean,\ ('\mathfrak{A})Boolean] \Rightarrow ('\mathfrak{A})Boolean$ (**infixl** *and 30*)

```
where   X and Y ≡ (λ τ . case X τ of
                  ⊥ ⇒ (case Y τ of
                            ⊥ ⇒ ⊥
                          | ⌊⊥⌋ ⇒ ⊥
                          | ⌊⌊True⌋⌋ ⇒ ⊥
                          | ⌊⌊False⌋⌋ ⇒ ⌊⌊False⌋⌋)
                | ⌊ ⊥ ⌋ ⇒ (case Y τ of
                            ⊥ ⇒ ⊥
                          | ⌊⊥⌋ ⇒ ⌊⊥⌋
                          | ⌊⌊True⌋⌋ ⇒ ⌊⊥⌋
                          | ⌊⌊False⌋⌋ ⇒ ⌊⌊False⌋⌋)
                | ⌊⌊True⌋⌋ ⇒ (case Y τ of
                            ⊥ ⇒ ⊥
                          | ⌊⊥⌋ ⇒ ⌊⊥⌋
                          | ⌊⌊y⌋⌋ ⇒ ⌊⌊y⌋⌋)
                | ⌊⌊False⌋⌋ ⇒ ⌊⌊ False ⌋⌋)
```

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*]: $(X\ \mathit{and}\ \mathit{true}) = X$
  ⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

4 Logical Equality and Referential 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* $(x{::}('\mathfrak{A},'a{::}\mathit{object})\mathit{val})\ (y{::}('\mathfrak{A},'a{::}\mathit{object})\mathit{val})$
$\equiv \lambda\ \tau.\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau \wedge (\delta\ y)\ \tau = \mathit{true}\ \tau$
    $\mathit{then}\ \lfloor\lfloor\ (\mathit{oid\text{-}of}\ \lceil\lceil x\ \tau\rceil\rceil) = (\mathit{oid\text{-}of}\ \lceil\lceil y\ \tau\rceil\rceil)\ \rfloor\rfloor$
    $\mathit{else}\ \mathit{invalid}\ \tau$

lemma *gen-ref-eq-object-strict1*[*simp*] :
$(\mathit{gen\text{-}ref\text{-}eq}\ (x{::}('\mathfrak{A},'a{::}\mathit{object})\mathit{val})\ \mathit{invalid}) = \mathit{invalid}$

$\langle proof \rangle$

lemma *gen-ref-eq-object-strict2* [*simp*] :
$(\textit{gen-ref-eq invalid } (x{::}('\mathfrak{A},'a{::}object)val)) = \textit{invalid}$
$\langle proof \rangle$

lemma *gen-ref-eq-object-strict3* [*simp*] :
$(\textit{gen-ref-eq } (x{::}('\mathfrak{A},'a{::}object)val) \textit{ null}) = \textit{invalid}$
$\langle proof \rangle$

lemma *gen-ref-eq-object-strict4* [*simp*] :
$(\textit{gen-ref-eq null } (x{::}('\mathfrak{A},'a{::}object)val)) = \textit{invalid}$
$\langle proof \rangle$

lemma *cp-gen-ref-eq-object*:
$(\textit{gen-ref-eq } x \ (y{::}('\mathfrak{A},'a{::}object)val)) \ \tau =$
$(\textit{gen-ref-eq } (\lambda\text{-}. \ x \ \tau) \ (\lambda\text{-}. \ y \ \tau)) \ \tau$
$\langle proof \rangle$

5 Local Validity

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

6 Global vs. Local Judgements

lemma *transform1*: $P = \textit{true} \Longrightarrow \tau \models P$
$\langle proof \rangle$

lemma *transform2*: $(P = Q) \Longrightarrow ((\tau \models P) = (\tau \models Q))$
$\langle proof \rangle$

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

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 $H : P = \textit{true} \Longrightarrow Q = \textit{true}$
shows $\tau \models P \Longrightarrow \tau \models Q$
$\langle proof \rangle$

7 Local Validity and Meta-logic

lemma *foundation1* [*simp*]: $\tau \models \textit{true}$
$\langle proof \rangle$

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

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

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

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

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

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

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

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

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

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

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

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*:
$\tau \models \delta\ x \Longrightarrow\ \ \tau \models \delta\ y \Longrightarrow (\tau \models (x\ \mathit{or}\ y)) = (\ (\tau \models x) \lor (\tau \models y))$
⟨*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 *strictEqGen-vs-strongEq*:
$\mathit{WFF}\ \tau \Longrightarrow \tau \models (\delta\ x) \Longrightarrow \tau \models (\delta\ y) \Longrightarrow$
$(\tau \models (\mathit{gen\text{-}ref\text{-}eq}\ (x{::}('b{::}\mathit{object},'a{::}\mathit{object})\mathit{val})\ y)) = (\tau \models (x \triangleq y))$
⟨*proof*⟩

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

8 Local Judgements and Strong Equality

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

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

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

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

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

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*: $!!\ \tau.\ \mathit{cp}\ P \Longrightarrow \tau \models (x \triangleq y) \Longrightarrow \tau \models (P\ x \triangleq P\ y)$
⟨*proof*⟩

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

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

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

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

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

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

9 Laws to Establish Definedness (Delta-Closure)

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

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

lemma *ocl-and-defargs*:
$\tau \models (P\ and\ Q) \Longrightarrow (\tau \models \delta\ P) \land (\tau \models \delta\ Q)$
⟨*proof*⟩

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

end
theory *OCL-lib*
imports *OCL-core*

begin

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* : $(x::('\mathfrak{A},int)val) \doteq y \equiv$
$\lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$then\ (x \triangleq y)\tau$
$else\ invalid\ \tau$

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

lemma *StrictRefEq-int-strict1*[*simp*] : $((x::('\mathfrak{A},int)val) \doteq invalid) = invalid$
$\langle proof \rangle$

lemma *StrictRefEq-int-strict2*[*simp*] : $(invalid \doteq (x::('\mathfrak{A},int)val)) = invalid$
$\langle proof \rangle$

lemma *StrictRefEq-int-strict3*[*simp*] : $((x::('\mathfrak{A},int)val) \doteq null) = invalid$
$\langle proof \rangle$

lemma *StrictRefEq-int-strict4*[*simp*] : $(null \doteq (x::('\mathfrak{A},int)val)) = invalid$
$\langle proof \rangle$

lemma *strictEqBool-vs-strongEq*:
$\tau \models(\delta\ x) \Longrightarrow \tau \models(\delta\ y) \Longrightarrow (\tau \models ((x::('\mathfrak{A},bool)val) \doteq y)) = (\tau \models (x \triangleq y))$
$\langle proof \rangle$

lemma *strictEqInt-vs-strongEq*:
$\tau \models(\delta\ x) \Longrightarrow \tau \models(\delta\ y) \Longrightarrow (\tau \models ((x::('\mathfrak{A},int)val) \doteq y)) = (\tau \models (x \triangleq y))$
$\langle proof \rangle$

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

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

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

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

lemma *StrictRefEq-int-strict′* :
  assumes A: $\delta\ ((x::('\mathfrak{A},int)val) \doteq y) = true$
  shows $\delta\ x = true \wedge \delta\ y = true$
  ⟨*proof*⟩

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

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

lemma *StrictRefEq-bool-strict3*[*simp*] : $((x::('\mathfrak{A},bool)val) \doteq null) = invalid$
⟨*proof*⟩

lemma *StrictRefEq-bool-strict4*[*simp*] : $(null \doteq (x::('\mathfrak{A},bool)val)) = invalid$
⟨*proof*⟩

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

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

lemmas *cp-rules* =
    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*: δ (x::($'\mathfrak{A}$,*int*)*val*) = *true*
  and *B*: δ y = *true*
  shows δ ($x \doteq y$) = *true*
  ⟨*proof*⟩

definition *ocl-zero* ::($'\mathfrak{A}$)*Integer* (**0**)
where **0** = (λ - . ⌊⌊*0::int*⌋⌋)

definition *ocl-one* ::($'\mathfrak{A}$)*Integer* (**1**)
where **1** = (λ - . ⌊⌊*1::int*⌋⌋)

definition *ocl-two* ::($'\mathfrak{A}$)*Integer* (**2**)
where **2** = (λ - . ⌊⌊*2::int*⌋⌋)

definition *ocl-three* ::($'\mathfrak{A}$)*Integer* (**3**)
where **3** = (λ - . ⌊⌊*3::int*⌋⌋)

definition *ocl-four* ::($'\mathfrak{A}$)*Integer* (**4**)
where **4** = (λ - . ⌊⌊*4::int*⌋⌋)

definition *ocl-five* ::($'\mathfrak{A}$)*Integer* (**5**)
where **5** = (λ - . ⌊⌊*5::int*⌋⌋)

definition *ocl-six* ::($'\mathfrak{A}$)*Integer* (**6**)
where **6** = (λ - . ⌊⌊*6::int*⌋⌋)

definition *ocl-seven* ::($'\mathfrak{A}$)*Integer* (**7**)
where **7** = (λ - . ⌊⌊*7::int*⌋⌋)

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

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

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

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

lemma [*simp*]:δ **0** = *true*
⟨*proof*⟩

lemma [*simp*]:υ **0** = *true*
⟨*proof*⟩

instance *option* :: (*plus*) *plus*
⟨*proof*⟩

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

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

lemma *zero-non-null*[*simp*]: $\mathbf{0} \neq null$
⟨*proof*⟩

10 Collection Types

10.1 Prerequisite: An Abstract Interface for OCL Types

In order to have the possibility to nest collection types, it is necessary to introduce a uniform interface for types having the "invalid" (= bottom) element. In a second step, our base-types will be shown to be instances of this class.

This uniform interface consists in 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 ⊥ to an abstract undefinedness element *UU* (also written ⊥ 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 *bottom* and *null* for the class of all types comprising a bottom and a distinct null element.

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

begin
  notation (*xsymbols*) *UU* (⊥)
end

class *null* = *bottom* +
  fixes *NULL* :: *'a*
  assumes *null-is-valid* : *NULL* ≠ *UU*

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.

instantiation *option* :: (*type*)*bottom*
begin

  definition *UU-option-def*: (*UU*::*'a option*) ≡ (*None*::*'a option*)

  instance ⟨*proof*⟩
end

instantiation *option* :: (*bottom*)*null*
begin
  definition *NULL-option-def*: (*NULL*::*'a*::*bottom option*) ≡ ⌊ *UU* ⌋

  instance ⟨*proof*⟩
end

instantiation *fun* :: (*type*,*bottom*) *bottom*
begin
  definition *UU-fun-def*: *UU* ≡ (λ *x*. *UU*)

  instance ⟨*proof*⟩
end

instantiation *fun* :: (*type*,*null*) *null*
begin
 definition *NULL-fun-def*: (*NULL*::*'a* ⇒ *'b*::*null*) ≡ (λ *x*. *NULL*)

 instance ⟨*proof*⟩
end

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

lemma [*simp*]: *null* = (*NULL*::(*'a*)*Integer*)
⟨*proof*⟩

lemma [*simp*]: $null = (NULL::('a)Boolean)$
⟨*proof*⟩

lemma [*simp*]: $\mathbf{0} \neq NULL$
⟨*proof*⟩

Now, on this basis we generalize the concept of a valuation: we do no longer care that the $\bot$ and $NULL$ were actually constructed by the type constructor option; rather, we require that the type is just from the null-class:

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

However, this has also the consequence that core concepts like definedned or validness have to be redefined on this type class:

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

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

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

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

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

lemma *defined3*[*simp*]: $\delta'\ \delta'\ X = true$
⟨*proof*⟩

lemma *valid4*[*simp*]: $\upsilon'\ (X \triangleq Y) = true$
⟨*proof*⟩

lemma *defined4*[*simp*]: $\delta'\ (X \triangleq Y) = true$
⟨*proof*⟩

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

lemma *valid5*[*simp*]: $\upsilon'\ \delta'\ X = true$
⟨*proof*⟩

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

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

lemmas *cp-intro*[*simp*,*intro*!] =
    cp-defined'[*THEN allI*[*THEN allI*[*THEN cpI1*], *of defined'*]]
    cp-valid'[*THEN allI*[*THEN allI*[*THEN cpI1*], *of valid'*]]
    cp-intro

In fact, it can be proven for the base types that both versions of undefined and invalid are actually the same:

lemma *defined-is-defined'*: $\delta \ X = \delta' \ X$
$\langle proof \rangle$

lemma *valid-is-valid'*: $\upsilon' \ X = \upsilon' \ X$
$\langle proof \rangle$

10.2 Example: The Set-Collection Type

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

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

The former principe rules out the option to define *'α Set* just by ('𝔄, (*'α option option*) *set*) *val*. This would allow sets to contain junk elements such as {⊥} 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 *'α Set-0*. it is shown that this type "fits" indeed into the abstract type interface discussed in the previous section.

typedef *'α Set-0* = {*X*::(*'α*::*null*) *set option option*.
                    $X = UU \lor X = NULL \lor (\forall x \in \lceil\lceil X \rceil\rceil. \ x \neq UU)$}

    $\langle proof \rangle$

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

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

instance $\langle proof \rangle$
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 $\langle proof \rangle$
end

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

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

... 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 $Set\{\} \equiv (\lambda\ \tau.\ \ Abs\text{-}Set\text{-}0\ \lfloor\lfloor\{\}::'\alpha\ set\rfloor\rfloor\ )$

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

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

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

consts
$OclSize\ \ :: ('\mathfrak{A},'\alpha::null)\ Set \Rightarrow\ '\mathfrak{A}\ Integer$
$OclCount\ \ :: [('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow\ '\mathfrak{A}\ Integer$

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

$OclExcluding\ \ :: [('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ val'] \Rightarrow ('\mathfrak{A},'\alpha)\ Set$
$OclSum\ \ :: ('\mathfrak{A},'\alpha::null)\ Set \Rightarrow\ '\mathfrak{A}\ Integer$
$OclIncludesAll :: [('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow\ '\mathfrak{A}\ Boolean$
$OclExcludesAll :: [('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow\ '\mathfrak{A}\ Boolean$

$OclIsEmpty$ :: $('\mathfrak{A},'\alpha{::}null)$ $Set \Rightarrow '\mathfrak{A}$ $Boolean$
$OclNotEmpty$ :: $('\mathfrak{A},'\alpha{::}null)$ $Set \Rightarrow '\mathfrak{A}$ $Boolean$
$OclComplement$ :: $('\mathfrak{A},'\alpha{::}null)$ $Set \Rightarrow ('\mathfrak{A},'\alpha)$ Set
$OclUnion$ :: $[('\mathfrak{A},'\alpha{::}null)$ $Set,('\mathfrak{A},'\alpha)$ $Set] \Rightarrow ('\mathfrak{A},'\alpha)$ Set
$OclIntersection$:: $[('\mathfrak{A},'\alpha{::}null)$ $Set,('\mathfrak{A},'\alpha)$ $Set] \Rightarrow ('\mathfrak{A},'\alpha)$ Set

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 *including-strict1*[*simp*]:$(\bot{-}{>}including(x)) = \bot$
⟨*proof*⟩

lemma *including-strict2*[*simp*]:$(X{-}{>}including(\bot)) = \bot$
⟨*proof*⟩

lemma *including-strict3*[*simp*]:$(NULL{-}{>}including(x)) = \bot$
⟨*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*$\{\mathbf{2},\mathbf{1}\}$ = (*Set*$\{\}$−>*including*($\mathbf{1}$)−>*including*($\mathbf{2}$))
⟨*proof*⟩

end

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

end

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

end