Essential OCL - S Study a Consistent Semantics for UML/OCL inHOL.

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 $oid\text{-}of :: {}'a \Rightarrow oid$

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

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

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 $WFF :: ('\mathfrak{A}::object)st \Rightarrow bool$
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.

1.3 Basic Constants

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

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

1.4 Boolean Type and Logic

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

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 $bool\text{-}split$: $X\ \tau = invalid\ \tau \vee X\ \tau = null\ \tau \vee$
$X\ \tau = true\ \tau \vee X\ \tau = false\ \tau$
apply(*simp add*: *invalid-def null-def true-def false-def*)
apply(*case-tac* $X\ \tau$,*simp*)
apply(*case-tac* a,*simp*)
apply(*case-tac* aa,*simp*)
apply *auto*
done

thm *bool-split*

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*)

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*: $(X \triangleq Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau) \triangleq (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
by(*simp add*: *StrongEq-def*)

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*: $(X \triangleq Y) = true$
and *B*: $(Y \triangleq Z) = true$
shows $(X \triangleq Z) = true$
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*

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

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

lemma *valid1*[*simp*]: $\upsilon\ invalid = false$
by(*rule ext*,*simp add*: *valid-def null-def invalid-def true-def false-def*)

lemma *valid2*[*simp*]: $\upsilon\ null = true$
by(*rule ext*,*simp add*: *valid-def null-def invalid-def true-def false-def*)

lemma *valid3*[*simp*]: $\upsilon\ \upsilon\ X = true$
  apply(*rule ext*,*simp add*: *valid-def null-def invalid-def true-def false-def*)
  apply(*case-tac X x*, *simp-all add*: *true-def false-def*)
  apply(*case-tac a*, *simp-all add*: *true-def false-def*)
  done

definition *defined* :: $('\mathfrak{A},'a)val \Rightarrow ('\mathfrak{A})Boolean\ (\delta$ - [*100*]*100*)
where $\delta\ X \equiv\ \lambda\ \tau\ .\ case\ X\ \tau\ of$
$$\begin{array}{ll} \ \ \perp & \Rightarrow false\ \tau \\ |\ \lfloor\ \perp\ \rfloor & \Rightarrow false\ \tau \\ |\ \lfloor\lfloor\ x\ \rfloor\rfloor & \Rightarrow true\ \tau \end{array}$$

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

lemma *defined1*[*simp*]: $\delta\ invalid = false$
  by(*rule ext*,*simp add*: *defined-def null-def invalid-def true-def false-def*)

lemma *defined2*[*simp*]: $\delta\ null = false$
  by(*rule ext*,*simp add*: *defined-def null-def invalid-def true-def false-def*)

lemma *defined3*[*simp*]: $\delta\ \delta\ X = true$
  apply(*rule ext*,*simp add*: *defined-def null-def invalid-def true-def false-def*)
  apply(*case-tac X x*, *simp-all add*: *true-def false-def*)
  apply(*case-tac a*, *simp-all add*: *true-def false-def*)
  done

lemma *valid4*[*simp*]: $\upsilon\ (X \triangleq Y) = true$
  by(*rule ext*,
    simp add: *valid-def null-def invalid-def StrongEq-def true-def false-def*)

lemma *defined4*[*simp*]: $\delta\ (X \triangleq Y) = true$
  by(*rule ext*,
    simp add: *defined-def null-def invalid-def StrongEq-def true-def false-def*)

lemma *defined5*[*simp*]: $\delta\ \upsilon\ X = true$
  apply(*rule ext*,*simp add*: *valid-def defined-def null-def invalid-def true-def false-def*)
  apply(*case-tac X x*, *simp-all add*: *true-def false-def*)
  apply(*case-tac a*, *simp-all add*: *true-def false-def*)
  done

lemma *valid5*[*simp*]: $\upsilon\ \delta\ X = true$
  apply(*rule ext*,*simp add*: *valid-def defined-def null-def invalid-def true-def false-def*)
  apply(*case-tac X x*, *simp-all add*: *true-def false-def*)
  apply(*case-tac a*, *simp-all add*: *true-def false-def*)
  done

3 Logical Connectives and their Universal Properties

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

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

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

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

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

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

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*:$(X\ and\ Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau)\ and\ (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
by(*simp add*: *ocl-and-def*)

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

lemma *cp-ocl-implies*:$(X\ implies\ Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau)\ implies\ (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
apply(*simp add*: *ocl-implies-def*)
apply(*subst cp-ocl-or*[*of not* $(\lambda\text{-}.\ X\ \tau)$ $(\lambda\text{-}.\ Y\ \tau)$])
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*)
lemma *ocl-and2*[*simp*]: (*invalid and false*) = *false*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-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*)
lemma *ocl-and4*[*simp*]: (*invalid and invalid*) = *invalid*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-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*)
lemma *ocl-and6*[*simp*]: (*null and false*) = *false*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-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*)
lemma *ocl-and8*[*simp*]: (*null and invalid*) = *invalid*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-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*)
lemma *ocl-and16*[*simp*]: (*true and invalid*) = *invalid*
by(*rule ext*,*simp add*: *ocl-and-def null-def invalid-def true-def false-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 Logical Equality and Referential Equality

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

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

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

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

lemma *gen-ref-eq-object-strict1*[*simp*] :
$(gen\text{-}ref\text{-}eq\ (x::('\mathfrak{A},'a::object)val)\ invalid) = invalid$
by(*rule ext*, *simp add*: *gen-ref-eq-def true-def false-def*)

lemma *gen-ref-eq-object-strict2*[*simp*] :

$(gen\text{-}ref\text{-}eq\ invalid\ (x::('\mathfrak{A},'a::object)val)) = invalid$
by(*rule ext*, *simp add*: *gen-ref-eq-def true-def false-def*)

lemma *gen-ref-eq-object-strict3*[*simp*] :
$(gen\text{-}ref\text{-}eq\ (x::('\mathfrak{A},'a::object)val)\ null) = invalid$
by(*rule ext*, *simp add*: *gen-ref-eq-def true-def false-def*)

lemma *gen-ref-eq-object-strict4*[*simp*] :
$(gen\text{-}ref\text{-}eq\ null\ (x::('\mathfrak{A},'a::object)val)) = invalid$
by(*rule ext*, *simp add*: *gen-ref-eq-def true-def false-def*)

lemma *cp-gen-ref-eq-object*:
$(gen\text{-}ref\text{-}eq\ x\ (y::('\mathfrak{A},'a::object)val))\ \tau =$
$(gen\text{-}ref\text{-}eq\ (\lambda\text{-}.\ x\ \tau)\ (\lambda\text{-}.\ y\ \tau))\ \tau$
by(*auto simp*: *gen-ref-eq-def StrongEq-def invalid-def cp-defined*[*symmetric*])

5 Local Validity

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

6 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*
    split: *option.split option.split-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

7 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*)

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

lemma *bool-split-local*[*simp*]:
$(\tau \models (x \triangleq invalid)) \lor (\tau \models (x \triangleq null)) \lor (\tau \models (x \triangleq true)) \lor (\tau \models (x \triangleq false))$
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*:
$(\tau \models \delta\ x) = ((\neg(\tau \models (x \triangleq invalid))) \land (\neg\ (\tau \models (x \triangleq null))))$
apply(*simp add*:*defined-def true-def false-def invalid-def null-def*
StrongEq-def OclValid-def)
apply(*case-tac x* τ,*simp*,*simp add*:*false-def*)
apply(*case-tac a*,*simp only*:)
apply(*simp-all add*:*false-def true-def*)
done

lemma *foundation5*:
$\tau \models (P\ and\ Q) \Longrightarrow (\tau \models P) \land (\tau \models Q)$
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*
split: *option.split option.split-asm*)

lemma *foundation7*[*simp*]:
$(\tau \models not\ (\delta\ x)) = (\neg\ (\tau \models \delta\ x))$
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*:
$(\tau \models \delta\ x) \vee (\tau \models (x \triangleq invalid)) \vee (\tau \models (x \triangleq null))$
proof −
  have *1* : $(\tau \models \delta\ x) \vee (\neg(\tau \models \delta\ x))$ **by** *auto*
  have *2* : $(\neg(\tau \models \delta\ x)) = ((\tau \models (x \triangleq invalid)) \vee (\tau \models (x \triangleq null)))$
    by(*simp only*: *def-split-local*, *simp*)
  show *?thesis* **by**(*insert 1*, *simp add*:*2*)
qed

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

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

lemma *foundation11*:
$\tau \models \delta\ x \Longrightarrow\ \tau \models \delta\ y \Longrightarrow (\tau \models (x\ or\ y)) = (\ (\tau \models x) \vee (\tau \models y))$
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
  split:*bool.split-asm bool.split*)

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

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

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

8 Local Judgements and Strong Equality

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

lemma *StrongEq-L-subst2*:
$!!\ \tau.\ \ cp\ P \Longrightarrow \tau \models (x \triangleq y) \Longrightarrow \tau \models (P\ x) \Longrightarrow \tau \models (P\ y)$
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]]

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

lemma *ocl-and-defargs*:
$\tau \models (P\ and\ Q) \Longrightarrow (\tau \models \delta\ P) \wedge (\tau \models \delta\ Q)$
by(*auto dest*: *foundation5 foundation6*)

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* : $(x{::}('\mathfrak{A},bool)val) \doteq y \equiv$

$$\lambda\ \tau.\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau \wedge (\delta\ y)\ \tau = \mathit{true}\ \tau$$
$$\mathit{then}\ (x \triangleq y)\tau$$
$$\mathit{else}\ \mathit{invalid}\ \tau$$

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

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

lemma *StrictRefEq-int-strict3* [*simp*] : $((x::('\mathfrak{A},int)val) \doteq null) = invalid$
by(*rule ext*, *simp add*: *StrictRefEq-int true-def false-def*)

lemma *StrictRefEq-int-strict4* [*simp*] : $(null \doteq (x::('\mathfrak{A},int)val)) = invalid$
by(*rule ext*, *simp add*: *StrictRefEq-int true-def false-def*)

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

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

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

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

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

lemma *StrictRefEq-int-strict* :
assumes A: $\delta\ (x::('\mathfrak{A},int)val) = true$
and B: $\delta\ y = true$
shows $\delta\ (x \doteq y) = true$
apply(*insert A B*)
apply(*rule ext*, *simp add*: *StrongEq-def StrictRefEq-int true-def defined-def*)

done

lemma *StrictRefEq-int-strict'* :
  assumes A: δ $((x::('\mathfrak{A},int)val) \doteq y) = true$
  shows $\delta\ x = true \wedge \delta\ y = true$
  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 defined-def)
  apply(*case-tac y xa*, *auto*)
  apply(*simp-all add*: *true-def invalid-def*)
  apply(*case-tac aa*, *auto simp*:*true-def false-def invalid-def*
    split: *option.split option.split-asm*)
  done

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

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

lemma *StrictRefEq-bool-strict3*[*simp*] : $((x::('\mathfrak{A},bool)val) \doteq null) = invalid$
by(*rule ext*, *simp add*: *StrictRefEq-bool true-def false-def*)

lemma *StrictRefEq-bool-strict4*[*simp*] : $(null \doteq (x::('\mathfrak{A},bool)val)) = invalid$
by(*rule ext*, *simp add*: *StrictRefEq-bool true-def false-def*)

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

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

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*
apply(*insert A B*)
apply(*rule ext*, *simp add*: *StrongEq-def StrictRefEq-int true-def defined-def*)
done

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

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

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

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

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

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

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

definition *ocl-seven* ::($'\mathfrak{A}$)*Integer* (**7**)
where **7** = (λ - . $\lfloor\lfloor$*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 [*simp*]:δ **0** = *true*
by(*simp add*:*ocl-zero-def defined-def true-def*)

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

instance *option* :: (*plus*) *plus*
by *intro-classes*

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

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

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

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 $\lfloor\ \bot\ \rfloor$ on *'a option option* to a NULL - element, which may have an abritrary semantic structure, and an undefinedness element $\bot$ to an abstract undefinedness element *UU* (also written $\bot$ 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* ($\perp$)
end

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

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*: $(UU{::}'a\ option) \equiv (None{::}'a\ option)$

  instance proof
      show $\exists x{::}'a\ option.\ x \neq UU$
      by(*rule-tac x=Some x* **in** *exI*, *simp add*:*UU-option-def*)
    qed
end

instantiation *option* :: (*bottom*)*null*
begin
  definition *NULL-option-def*: $(NULL{::}'a{::}bottom\ option) \equiv \lfloor\ UU\ \rfloor$

  instance proof **show** $(NULL{::}'a{::}bottom\ option) \neq UU$
      by(*simp add*:*NULL-option-def UU-option-def*)
    qed
end

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

  instance proof **show** $\exists (x{::}'a \Rightarrow 'b).\ x \neq UU$
      apply(*rule-tac x*=λ -. (*SOME y*. $y \neq UU$) **in** *exI*, *auto*)
      apply(*drule-tac x=x* **in** *fun-cong*,*auto simp*:*UU-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 UU-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).

lemma [*simp*]: $null = (NULL{::}('a)Integer)$
by(*rule ext*,*simp add*: *UU-option-def NULL-option-def null-def NULL-fun-def*)

lemma [*simp*]: $null = (NULL{::}('a)Boolean)$
by(*rule ext*,*simp add*: *UU-option-def NULL-option-def null-def NULL-fun-def*)

lemma [*simp*]: $\mathbf{0} \neq NULL$
by(*simp add*: *zero-non-null*[*simplified*])

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

definition *defined'* :: $('\mathfrak{A},'a{::}null)val' \Rightarrow ('\mathfrak{A})Boolean$ (δ' - [*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*]: $\delta'\ invalid = false$
by(*rule ext*,*simp add*: *defined'-def UU-fun-def UU-option-def*
null-def invalid-def true-def false-def)

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

by(*rule ext*,*simp add*: *defined′-def bot-fun-def UU-option-def*
null-def NULL-option-def NULL-fun-def invalid-def true-def false-def)

lemma *defined3*[*simp*]: $\delta'\ \delta'\ X = true$
by(*rule ext*,*auto simp*: *defined′-def true-def false-def*
UU-fun-def UU-option-def NULL-option-def NULL-fun-def)

lemma *valid4*[*simp*]: $\upsilon'\ (X \triangleq Y) = true$
by(*rule ext*,
auto simp: *valid′-def true-def false-def StrongEq-def*
UU-fun-def UU-option-def NULL-option-def NULL-fun-def)

lemma *defined4*[*simp*]: $\delta'\ (X \triangleq Y) = true$
by(*rule ext*,
auto simp: *valid′-def defined′-def true-def false-def StrongEq-def*
UU-fun-def UU-option-def NULL-option-def NULL-fun-def)

lemma *defined5*[*simp*]: $\delta'\ \upsilon'\ X = true$
by(*rule ext*,
auto simp: *valid′-def defined′-def true-def false-def StrongEq-def*
UU-fun-def UU-option-def NULL-option-def NULL-fun-def)

lemma *valid5*[*simp*]: $\upsilon'\ \delta'\ X = true$
by(*rule ext*,
auto simp: *valid′-def defined′-def true-def false-def StrongEq-def*
UU-fun-def UU-option-def NULL-option-def NULL-fun-def)

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

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

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′*: $\delta\ X = \delta'\ X$
by(*rule ext*,
auto simp: *defined′-def defined-def true-def false-def false-def true-def*
UU-fun-def UU-option-def NULL-option-def NULL-fun-def)

lemma *valid-is-valid'*: $\upsilon'\ X = \upsilon'\ X$
by(*rule ext*,
auto simp: *valid'-def valid-def true-def false-def false-def true-def UU-fun-def UU-option-def NULL-option-def NULL-fun-def*)

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

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

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

by (*rule-tac* $x{=}UU$ **in** *exI*, *simp*)

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 proof show $\exists\, x::'a\ Set\text{-}0.\ x \neq \bot$
apply(*rule-tac* $x{=}Abs\text{-}Set\text{-}0\ \lfloor None\rfloor$ **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 UU-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 UU-option-def*)
done
qed
end

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

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

... 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*]:$(\bot{-}{>}including(x)) = \bot$
by(*simp add*: *UU-fun-def OclIncluding-def defined'-def valid'-def false-def true-def*)

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

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

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*

end

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

end

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

end