

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

Burkhart Wolff

June 8, 2012

Contents

1	OCL Core Definitions	2
2	Foundational Notations	2
2.1	Notations for the option type	2
2.2	State, State Transitions, Well-formed States	2
2.3	Prerequisite: An Abstract Interface for OCL Types	3
2.4	Accommodation of Basic Types to the Abstract Interface	4
2.5	The Semantic Space of OCL Types: Valuations.	5
2.6	Further requirements on States	5
3	The OCL Base Type Boolean.	6
4	Boolean Type and Logic	6
4.1	Basic Constants	6
4.2	Fundamental Predicates I: Validity and Definedness	7
4.3	Fundamental Predicates II: Logical (Strong) Equality	8
4.4	Fundamental Predicates III: (Generic) Referential Strict Equality	9
4.5	Logical Connectives and their Universal Properties	10
4.6	A Standard Logical Calculus for OCL	15
5	Global vs. Local Judgements	15
5.0.1	Local Validity and Meta-logic	15
6	Local Judgements and Strong Equality	18
7	Laws to Establish Definedness (Delta-Closure)	19
8	Miscellaneous: OCL's if then else endif	20
9	Simple, Basic Types like Void, Boolean and Integer	20

10 Strict equalities.	21
10.1 Example: The Set-Collection Type on the Abstract Interface	25
10.2 Some computational laws:	31
11 Example Data-Universe	33
12 Instantiation of the generic strict equality	34
13 Selector Definition	34
14 Standard State Infrastructure	36
15 Invariant	36
16 The contract of a recursive query :	36
17 The contract of a method.	37

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 ( $\lfloor(-)\rfloor$ )
notation None ( $\perp$ )

```

```

fun   drop :: 'α option ⇒ 'α ( $\lceil(-)\rceil$ )
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 \mathcal{A} , and state transitions pairs of states...

type-synonym $(\mathcal{A})state = oid \rightarrow \mathcal{A}$

type-synonym $(\mathcal{A})st = \mathcal{A} \text{ state} \times \mathcal{A} \text{ state}$

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 $\mathcal{A} :: \text{object}$

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 $Set\{Set\{2\}, null\}$, 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 mused 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 $'a \text{ option option}$ to a null - element, which may have an arbitrary 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)  $\equiv$  [ bot ]
  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 distinguished *null* and *invalid* element).

type-synonym ($\mathfrak{A}, 'a$) val = $\mathfrak{A} \text{ st} \Rightarrow 'a$

All OCL expressions *denote* functions that map the underlying

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

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

The definition :

```

definition null      :: "('\ $\alpha$ >, ' $\alpha$ >::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 $\text{null} \equiv \lambda x. \text{null}$.

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

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 $('A)Boolean = ('A, bool option option) val$

4.1 Basic Constants

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

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

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

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

lemma *bool-split*: $X \tau = invalid \tau \vee X \tau = null \tau \vee$
 $X \tau = true \tau \quad \vee \quad X \tau = false \tau$
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) = $\llbracket \text{False} \rrbracket$
by(simp add:false-def)

lemma [simp]: true (a, b) = $\llbracket \text{True} \rrbracket$
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 :: ('A,'a::null)val \Rightarrow ('A)Boolean (v - [100]100)
where v X \equiv $\lambda \tau$. if X τ = bot τ then false τ else true τ

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

lemma valid2[simp]: v 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]: v 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]: v 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: (v X) τ = (v (λ -. X τ)) τ
by(simp add: valid-def)

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

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]: $\delta \text{ true} = \text{true}$
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]: $\delta \text{ false} = \text{true}$
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]: $\delta \delta X = \text{true}$
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]: $\delta v X = \text{true}$
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]: $\delta \delta X = \text{true}$
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]: $v \delta X = \text{true}$
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*: $(\delta X)\tau = (\delta (\lambda \cdot. X \tau)) \tau$
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*:: $[\mathfrak{A} \text{ st} \Rightarrow 'a, \mathfrak{A} \text{ st} \Rightarrow 'a] \Rightarrow ('a) \text{ Boolean}$ (**infixl** \triangleq 30)
where $X \triangleq Y \equiv \lambda \tau. \llbracket X \tau = Y \tau \rrbracket$

Equality reasoning in OCL is not humpty dumpty. While strong equality is clearly an equivalence:

lemma *StrongEq-refl* [simp]: $(X \triangleq X) = \text{true}$

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

... 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 refers 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: $\bigwedge X. P(X)\tau = P(\lambda _. X \tau)\tau$

and eq: $(X \triangleq Y)\tau = true \tau$

shows $(P X \triangleq P Y)\tau = true \tau$

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

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

definition gen-ref-eq :: $(\mathfrak{A}, 'a::\{object, null\})val \Rightarrow (\mathfrak{A}, 'a)val \Rightarrow (\mathfrak{A})Boolean$

where gen-ref-eq x y

$\equiv \lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (\delta y) \tau = true \tau$

$\text{then } \llbracket (oid-of (x \tau)) = (oid-of (y \tau)) \rrbracket$

$\text{else } invalid \tau$

```

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  $\tau$ ) = (gen-ref-eq ( $\lambda\cdot$ . x  $\tau$ ) ( $\lambda\cdot$ . y  $\tau$ ))  $\tau$ 
by(auto simp: gen-ref-eq-def StrongEq-def invalid-def cp-defined[symmetric])

And, last but not least,

lemma defined8[simp]:  $\delta$  ( $X \triangleq Y$ ) = 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]:  $v$  ( $X \triangleq Y$ ) = 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: ( $X \triangleq Y$ )  $\tau$  = (( $\lambda\cdot$ . X  $\tau$ )  $\triangleq$  ( $\lambda\cdot$ . Y  $\tau$ ))  $\tau$ 
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 than 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 $\text{not } X \equiv \lambda \tau . \text{case } X \text{ } \tau \text{ of}$

$$\begin{aligned} \perp &\Rightarrow \perp \\ | \lfloor \perp \rfloor &\Rightarrow \lfloor \perp \rfloor \\ | \lfloor \lfloor x \rfloor \rfloor &\Rightarrow \lfloor \lfloor \neg x \rfloor \rfloor \end{aligned}$$

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

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

by(*simp add: not-def*)

lemma *not1[simp]*: $\text{not invalid} = \text{invalid}$

by(*rule ext,simp add: not-def null-def invalid-def true-def false-def bot-option-def*)

lemma *not2[simp]*: $\text{not null} = \text{null}$

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]*: $\text{not true} = \text{false}$

by(*rule ext,simp add: not-def null-def invalid-def true-def false-def*)

lemma *not4[simp]*: $\text{not false} = \text{true}$

by(*rule ext,simp add: not-def null-def invalid-def true-def false-def*)

lemma *not-not[simp]*: $\text{not } (\text{not } X) = X$

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

$a <> b == \text{CONST not}(a \doteq b)$

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

where $X \text{ and } Y \equiv (\lambda \tau . \text{case } X \text{ } \tau \text{ of}$

$$\begin{aligned} \perp &\Rightarrow (\text{case } Y \text{ } \tau \text{ of} \\ &\quad \perp \Rightarrow \perp \\ &\quad | \lfloor \perp \rfloor \Rightarrow \perp \\ &\quad | \lfloor \lfloor \text{True} \rfloor \rfloor \Rightarrow \perp \\ &\quad | \lfloor \lfloor \text{False} \rfloor \rfloor \Rightarrow \lfloor \lfloor \text{False} \rfloor \rfloor) \end{aligned}$$

$$\begin{aligned}
& | \lfloor \perp \rfloor \Rightarrow (\text{case } Y \ \tau \text{ of} \\
& \quad \perp \Rightarrow \perp \\
& \quad | \lfloor \perp \rfloor \Rightarrow \lfloor \perp \rfloor \\
& \quad | \lfloor \text{True} \rfloor \Rightarrow \lfloor \perp \rfloor \\
& \quad | \lfloor \text{False} \rfloor \Rightarrow \lfloor \text{False} \rfloor) \\
& | \lfloor \text{True} \rfloor \Rightarrow (\text{case } Y \ \tau \text{ of} \\
& \quad \perp \Rightarrow \perp \\
& \quad | \lfloor \perp \rfloor \Rightarrow \lfloor \perp \rfloor \\
& \quad | \lfloor y \rfloor \Rightarrow \lfloor y \rfloor) \\
& | \lfloor \text{False} \rfloor \Rightarrow \lfloor \text{False} \rfloor
\end{aligned}$$

definition *ocl-or* :: $((\mathfrak{A})\text{Boolean}, (\mathfrak{A})\text{Boolean}) \Rightarrow (\mathfrak{A})\text{Boolean}$
(**infixl** or 25)

where $X \text{ or } Y \equiv \text{not}(\text{not } X \text{ and } \text{not } Y)$

definition *ocl-implies* :: $((\mathfrak{A})\text{Boolean}, (\mathfrak{A})\text{Boolean}) \Rightarrow (\mathfrak{A})\text{Boolean}$
(**infixl** implies 25)

where $X \text{ implies } Y \equiv \text{not } X \text{ or } Y$

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

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

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

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

lemma *ocl-and5*[*simp*]: $(\text{null and true}) = \text{null}$
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* :: $[(\mathfrak{A})st, (\mathfrak{A})Boolean] \Rightarrow bool \ ((1(-)/ \models (-)) \ 50)$
where $\tau \models P \equiv ((P \ \tau) = true \ \tau)$

5 Global vs. Local Judgements

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

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

lemma *transform2-rev*: $\forall \tau. (\tau \models \delta \ P) \wedge (\tau \models \delta \ Q) \wedge (\tau \models P) = (\tau \models Q) \implies P = Q$
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 $H : P = true \implies Q = true$
shows $\tau \models P \implies \tau \models Q$
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]:
 $(\tau \models (x \triangleq \text{invalid})) \vee (\tau \models (x \triangleq \text{null})) \vee (\tau \models (x \triangleq \text{true})) \vee (\tau \models (x \triangleq \text{false}))$
apply(*insert bool-split*[of $x \ \tau$], *auto*)
apply(*simp-all add: OclValid-def StrongEq-def true-def null-def invalid-def*)
done

lemma *def-split-local*:
 $(\tau \models \delta \ x) = ((\neg(\tau \models (x \triangleq \text{invalid}))) \wedge (\neg(\tau \models (x \triangleq \text{null}))))$
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*:
 $\tau \models (P \text{ and } Q) \implies (\tau \models P) \wedge (\tau \models Q)$
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*:
 $\tau \models P \implies \tau \models \delta \ P$
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]:
 $(\tau \models \text{not } (\delta \ x)) = (\neg(\tau \models \delta \ x))$
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*:
 $(\tau \models \delta \ x) \vee (\tau \models (x \triangleq \text{invalid})) \vee (\tau \models (x \triangleq \text{null}))$
proof –
 have *1* : $(\tau \models \delta \ x) \vee (\neg(\tau \models \delta \ x))$ **by** *auto*
 have *2* : $(\neg(\tau \models \delta \ x)) = ((\tau \models (x \triangleq \text{invalid})) \vee (\tau \models (x \triangleq \text{null})))$
 by(*simp only: def-split-local, simp*)
 show *?thesis* **by**(*insert 1, simp add:2*)
qed

lemma *foundation9*:
 $\tau \models \delta \ x \implies (\tau \models \text{not } x) = (\neg(\tau \models x))$
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*:

$\tau \models \delta x \implies \tau \models \delta y \implies (\tau \models (x \text{ and } y)) = ((\tau \models x) \wedge (\tau \models y))$

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*:

$\tau \models \delta x \implies \tau \models \delta y \implies (\tau \models (x \text{ or } y)) = ((\tau \models x) \vee (\tau \models y))$

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*:

$\tau \models \delta x \implies \tau \models \delta y \implies (\tau \models (x \text{ implies } y)) = ((\tau \models x) \longrightarrow (\tau \models y))$

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*: $(\tau \models A \triangleq \text{true}) = (\tau \models A)$

by(*auto simp: not-def OclValid-def invalid-def true-def null-def StrongEq-def
split:bool.split-asm bool.split*)

lemma *foundation14*: $(\tau \models A \triangleq \text{false}) = (\tau \models \text{not } A)$

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*: $(\tau \models A \triangleq \text{invalid}) = (\tau \models \text{not}(v A))$

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*: $\tau \models (\delta X) = (X \tau \neq \text{bot} \wedge X \tau \neq \text{null})$

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*: $\tau \models (v\ X) = (X\ \tau \neq \text{bot})$
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* : $\tau \models (\delta\ X) \implies \tau \models v\ X$
by(*simp add: foundation18 foundation16*)

theorem *strictEqGen-vs-strongEq*:
 $WFF\ \tau \implies \tau \models (\delta\ x) \implies \tau \models (\delta\ y) \implies$
 $(\tau \models (\text{gen-ref-eq}\ x\ y)) = (\tau \models (x \triangleq y))$
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*: $\tau \models (x \triangleq y) \implies \tau \models (y \triangleq x)$
by(*simp add: OclValid-def StrongEq-def*)

lemma *StrongEq-L-trans*: $\tau \models (x \triangleq y) \implies \tau \models (y \triangleq z) \implies \tau \models (x \triangleq z)$
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* :: $((\mathfrak{A}, \alpha)\ \text{val} \Rightarrow (\mathfrak{A}, \beta)\ \text{val}) \Rightarrow \text{bool}$
where $\text{cp}\ P \equiv (\exists f. \forall X\ \tau. P\ X\ \tau = f\ (X\ \tau)\ \tau)$

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

lemma *StrongEq-L-subst2*:

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

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

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

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

lemma *cp-id* : $\text{cp}(\lambda X. X)$
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 $?\tau \models ?P \implies ?\tau \models \delta \ ?P$ — the following facts:

lemma *ocl-not-defargs*:
 $\tau \models (\text{not } P) \implies \tau \models \delta \ P$
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 equal-

ity.

8 Miscellaneous: OCL's if then else endif

definition *if-ocl* :: $[(\mathcal{A})\text{Boolean}, (\mathcal{A}, \alpha::\text{null}) \text{ val}, (\mathcal{A}, \alpha) \text{ val}] \Rightarrow (\mathcal{A}, \alpha) \text{ val}$
 $(\text{if } (-) \text{ then } (-) \text{ else } (-) \text{ endif } [10,10,10]50)$

where $(\text{if } C \text{ then } B_1 \text{ else } B_2 \text{ endif}) = (\lambda \tau. \text{if } (\delta \ C) \ \tau = \text{true} \ \tau$
 $\quad \text{then } (\text{if } (C \ \tau) = \text{true} \ \tau$
 $\quad \quad \text{then } B_1 \ \tau$
 $\quad \quad \text{else } B_2 \ \tau)$
 $\quad \text{else invalid } \tau)$

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

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

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

lemma *if-ocl-false* : $(\text{if false then } B_1 \text{ else } B_2 \text{ endif}) = B_2$
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 $(\mathcal{A})\text{Integer} = (\mathcal{A}, \text{int option option}) \text{ val}$

type-synonym $(\mathcal{A})\text{Void} = (\mathcal{A}, \text{unit option}) \text{ val}$

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

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

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

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

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

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

lemma *strictEqBool-defargs*:
 $\tau \models ((x::(\mathfrak{A})Boolean) \doteq y) \implies (\tau \models (v\ x)) \wedge (\tau \models (v\ y))$
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*:
 $\tau \models ((x::(\mathfrak{A})Integer) \doteq y) \implies (\tau \models (v\ x)) \wedge (\tau \models (v\ y))$
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*:
 $(\tau \models v((x::(\mathfrak{A})Boolean) \doteq y)) = ((\tau \models (v\ x)) \wedge (\tau \models (v\ y)))$
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:*

$(\tau \models v((x::('A)Integer) \doteq y)) = ((\tau \models (v\ x)) \wedge (\tau \models (v\ y)))$

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:*

$\tau \models (gen-ref-eq\ x\ (y::('A,'a::\{null,object\})val)) \implies (\tau \models (\delta\ x)) \wedge (\tau \models (\delta\ y))$

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: v (x::('A)Integer) = true*

and *B: v y = true*

shows *v (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: v (((x::('A)Integer)) \doteq y) = true*

shows *v x = true \wedge v 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::('A)Boolean) \doteq invalid) = invalid*

by(*rule ext, simp add: StrictRefEq-bool true-def false-def*)

lemma *StrictRefEq-bool-strict2[simp] : (invalid \doteq (x::('A)Boolean)) = invalid*

by(*rule ext, simp add: StrictRefEq-bool true-def false-def*)

lemma *cp-StrictRefEq-bool:*

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

lemma *cp-StrictRefEq-int*:
 $((X::('A)Integer) \doteq Y) \tau = ((\lambda -. X \tau) \doteq (\lambda -. Y \tau)) \tau$
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*: $v (x::('A)Integer) = true$
and *B*: $v y = true$
shows $v (x \doteq y) = true$
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* :: $('A)Integer$ (**0**)
where $\mathbf{0} = (\lambda -. \llbracket 0::int \rrbracket)$

definition *ocl-one* :: $('A)Integer$ (**1**)
where $\mathbf{1} = (\lambda -. \llbracket 1::int \rrbracket)$

definition *ocl-two* :: $('A)Integer$ (**2**)
where $\mathbf{2} = (\lambda -. \llbracket 2::int \rrbracket)$

definition *ocl-three* :: $('A)Integer$ (**3**)
where $\mathbf{3} = (\lambda -. \llbracket 3::int \rrbracket)$

definition *ocl-four* :: $('A)Integer$ (**4**)
where $\mathbf{4} = (\lambda -. \llbracket 4::int \rrbracket)$

definition *ocl-five* :: $('A)Integer$ (**5**)
where $\mathbf{5} = (\lambda -. \llbracket 5::int \rrbracket)$

definition *ocl-six* :: $('A)Integer$ (**6**)
where $\mathbf{6} = (\lambda -. \llbracket 6::int \rrbracket)$

definition *ocl-seven* :: $('A)Integer$ (**7**)
where $\mathbf{7} = (\lambda -. \llbracket 7::int \rrbracket)$

definition *ocl-eight* :: (' \mathcal{A}) Integer (8)

where $\mathbf{8} = (\lambda - . \lfloor \lfloor 8 :: \text{int} \rfloor \rfloor)$

definition *ocl-nine* :: (' \mathcal{A}) Integer (9)

where $\mathbf{9} = (\lambda - . \lfloor \lfloor 9 :: \text{int} \rfloor \rfloor)$

definition *ten-nine* :: (' \mathcal{A}) Integer (10)

where $\mathbf{10} = (\lambda - . \lfloor \lfloor 10 :: \text{int} \rfloor \rfloor)$

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

lemma $\delta \text{ null} = \text{false}$ **by** *simp*

lemma $v \text{ null} = \text{true}$ **by** *simp*

lemma [*simp*]: $\delta \mathbf{0} = \text{true}$

by(*simp* *add:ocl-zero-def* *defined-def* *true-def*
bot-fun-def *bot-option-def* *null-fun-def* *null-option-def*)

lemma [*simp*]: $v \mathbf{0} = \text{true}$

by(*simp* *add:ocl-zero-def* *valid-def* *true-def*
bot-fun-def *bot-option-def* *null-fun-def* *null-option-def*)

lemma [*simp*]: $\delta \mathbf{1} = \text{true}$

by(*simp* *add:ocl-one-def* *defined-def* *true-def*
bot-fun-def *bot-option-def* *null-fun-def* *null-option-def*)

lemma [*simp*]: $v \mathbf{1} = \text{true}$

by(*simp* *add:ocl-one-def* *valid-def* *true-def*
bot-fun-def *bot-option-def* *null-fun-def* *null-option-def*)

lemma [*simp*]: $\delta \mathbf{2} = \text{true}$

by(*simp* *add:ocl-two-def* *defined-def* *true-def*
bot-fun-def *bot-option-def* *null-fun-def* *null-option-def*)

lemma [*simp*]: $v \mathbf{2} = \text{true}$

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*]: $\mathbf{0} \neq \text{null}$

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*]: $\mathbf{1} \neq \text{null}$

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* :: (' \mathcal{A})Integer \Rightarrow (' \mathcal{A})Integer \Rightarrow (' \mathcal{A})Boolean (**infix** \prec 40)
where $x \prec y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
 then $\llbracket \llbracket x \tau \rrbracket < \llbracket y \tau \rrbracket \rrbracket$
 else invalid τ

definition *ocl-le-int* :: (' \mathcal{A})Integer \Rightarrow (' \mathcal{A})Integer \Rightarrow (' \mathcal{A})Boolean (**infix** \preceq 40)
where $x \preceq y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
 then $\llbracket \llbracket x \tau \rrbracket \leq \llbracket y \tau \rrbracket \rrbracket$
 else invalid τ

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 $\text{Set}(\text{Set}(\text{Sequences}(\text{Pairs}(X, Y))))$), and

The former principle rules out the option to define ' α Set just by (' \mathcal{A} , (' α 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 ' 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 = { $X :: (' \alpha :: \text{null}) \text{ set option option}.$
 $X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})$ }
by (rule-tac $x=\text{bot}$ **in** *exI*, *simp*)

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

```

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

instance proof show  $\exists x::'a$  Set-0.  $x \neq bot$ 
  apply(rule-tac  $x=Abs-Set-0 \ [None]$  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-0)  $\equiv$  Abs-Set-0 [ None ]

  instance proof show (null::('a::null) Set-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 (' $\mathcal{A}$ , ' $\alpha$ ) Set = (' $\mathcal{A}$ , ' $\alpha$  Set-0) val

```

```

lemma Set-inv-lemma:  $\tau \models (\delta \ X) \implies (X \ \tau = Abs-Set-0 \ [bot]) \vee (\forall x \in [[Rep-Set-0 \ (X \ \tau)]] . x \neq bot)$ 
apply(insert OCL-lib.Set-0.Rep-Set-0 [of X  $\tau$ ], 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 (' \mathcal{A} ,('' \mathcal{A} ,('' \mathcal{A}) Integer) Set) Set corresponding exactly to Set(Set(Integer)) in OCL notation. Note that the parameter \mathcal{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 $mtSet :: ('A, 'α :: null) Set (Set\{\})$
where $Set\{\} \equiv (\lambda \tau. Abs-Set-0 \llbracket \{\} :: 'α\ set \rrbracket)$

lemma $mtSet-defined[simp]: \delta(Set\{\}) = true$
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 $mtSet-valid[simp]: v(Set\{\}) = true$
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 $OclSize :: ('A, 'α :: null) Set \Rightarrow 'A Integer$
where $OclSize\ x = (\lambda \tau. \text{if } (\delta\ x)\ \tau = true\ \tau \wedge finite(\llbracket Rep-Set-0\ (x\ \tau) \rrbracket))$
 $\text{then } \llbracket int(card\ \llbracket Rep-Set-0\ (x\ \tau) \rrbracket) \rrbracket$
 $\text{else } \perp)$

definition $OclIncluding :: (('A, 'α :: null) Set, ('A, 'α) val) \Rightarrow ('A, 'α) Set$
where $OclIncluding\ x\ y = (\lambda \tau. \text{if } (\delta\ x)\ \tau = true\ \tau \wedge (v\ y)\ \tau = true\ \tau$
 $\text{then } Abs-Set-0\ \llbracket \llbracket Rep-Set-0\ (x\ \tau) \rrbracket \cup \{y\ \tau\} \rrbracket$
 $\text{else } \perp)$

definition $OclIncludes :: (('A, 'α :: null) Set, ('A, 'α) val) \Rightarrow 'A Boolean$
where $OclIncludes\ x\ y = (\lambda \tau. \text{if } (\delta\ x)\ \tau = true\ \tau \wedge (v\ y)\ \tau = true\ \tau$
 $\text{then } \llbracket (y\ \tau) \in \llbracket Rep-Set-0\ (x\ \tau) \rrbracket \rrbracket$
 $\text{else } \perp)$

definition $OclExcluding :: (('A, 'α :: null) Set, ('A, 'α) val) \Rightarrow ('A, 'α) Set$
where $OclExcluding\ x\ y = (\lambda \tau. \text{if } (\delta\ x)\ \tau = true\ \tau \wedge (v\ y)\ \tau = true\ \tau$
 $\text{then } Abs-Set-0\ \llbracket \llbracket Rep-Set-0\ (x\ \tau) \rrbracket - \{y\ \tau\} \rrbracket$
 $\text{else } \perp)$

definition $OclExcludes :: (('A, 'α :: null) Set, ('A, 'α) val) \Rightarrow 'A Boolean$

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

definition $OclIsEmpty :: ('A, 'α :: null) Set \Rightarrow 'A Boolean$

where $OclIsEmpty\ x = ((OclSize\ x) \doteq 0)$

definition $OclNotEmpty :: ('A, 'α :: null) Set \Rightarrow 'A Boolean$

where $OclNotEmpty\ x = not(OclIsEmpty\ x)$

definition $OclForall :: [('A, 'α :: null) Set, ('A, 'α) val \Rightarrow ('A) Boolean] \Rightarrow 'A Boolean$

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

definition $OclExists :: [('A, 'α :: null) Set, ('A, 'α) val \Rightarrow ('A) Boolean] \Rightarrow 'A Boolean$

where $OclExists\ S\ P = not(OclForall\ S\ (\lambda\ X. not\ (P\ X)))$

syntax

$-OclForall :: [('A, 'α :: null) Set, id, ('A) Boolean] \Rightarrow 'A Boolean \quad ((-)->forall'(-|-'))$

translations

$X->forall(x\ |\ P) == CONST\ OclForall\ X\ (\%x. P)$

syntax

$-OclExist :: [('A, 'α :: null) Set, id, ('A) Boolean] \Rightarrow 'A Boolean \quad ((-)->exists'(-|-'))$

translations

$X->exists(x\ |\ P) == CONST\ OclExists\ X\ (\%x. P)$

consts

$OclUnion :: [('A, 'α :: null) Set, ('A, 'α) Set] \Rightarrow ('A, 'α) Set$
 $OclIntersection :: [('A, 'α :: null) Set, ('A, 'α) Set] \Rightarrow ('A, 'α) Set$
 $OclIncludesAll :: [('A, 'α :: null) Set, ('A, 'α) Set] \Rightarrow 'A Boolean$
 $OclExcludesAll :: [('A, 'α :: null) Set, ('A, 'α) Set] \Rightarrow 'A Boolean$
 $OclComplement :: ('A, 'α :: null) Set \Rightarrow ('A, 'α) Set$

$OclSum \quad :: (\mathcal{A}, \alpha :: null) \text{ Set} \Rightarrow \mathcal{A} \text{ Integer}$
 $OclCount \quad :: [(\mathcal{A}, \alpha :: null) \text{ Set}, (\mathcal{A}, \alpha) \text{ Set}] \Rightarrow \mathcal{A} \text{ Integer}$

notation

$OclSize \quad (\rightarrow size'(') [66])$
and
 $OclCount \quad (\rightarrow count'(-) [66,65] 65)$
and
 $OclIncludes \quad (\rightarrow includes'(-) [66,65] 65)$
and
 $OclExcludes \quad (\rightarrow excludes'(-) [66,65] 65)$
and
 $OclSum \quad (\rightarrow sum'(') [66])$
and
 $OclIncludesAll \quad (\rightarrow includesAll'(-) [66,65] 65)$
and
 $OclExcludesAll \quad (\rightarrow excludesAll'(-) [66,65] 65)$
and
 $OclIsEmpty \quad (\rightarrow isEmpty'(') [66])$
and
 $OclNotEmpty \quad (\rightarrow notEmpty'(') [66])$
and
 $OclIncluding \quad (\rightarrow including'(-))$
and
 $OclExcluding \quad (\rightarrow excluding'(-))$
and
 $OclComplement \quad (\rightarrow complement'('))$
and
 $OclUnion \quad (\rightarrow union'(-) \quad [66,65] 65)$
and
 $OclIntersection \quad (\rightarrow intersection'(-) \quad [71,70] 70)$

lemma *cp-OclIncluding*:

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

lemma *cp-OclExcluding*:

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

lemma *cp-OclIncludes*:

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

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

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

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

lemma *including-valid-args-valid*:
 $(\tau \models \delta(X \rightarrow \text{including}(x))) = ((\tau \models (\delta X)) \wedge (\tau \models (v x)))$
proof –
have $A : \perp \in \text{Set-0}$ **by**(simp add: Set-0-def bot-option-def)
have $B : \lfloor \perp \rfloor \in \text{Set-0}$ **by**(simp add: Set-0-def null-option-def bot-option-def)
have $C : (\tau \models (\delta X)) \implies (\tau \models (v x)) \implies \lfloor \text{insert } (x \ \tau) \text{ } \llbracket \text{Rep-Set-0 } (X \ \tau) \rrbracket \rfloor \rfloor \rfloor$
 $\in \text{Set-0}$
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 $D : (\tau \models \delta(X \rightarrow \text{including}(x))) \implies ((\tau \models (\delta X)) \wedge (\tau \models (v x)))$
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 $E : (\tau \models (\delta X)) \implies (\tau \models (v x)) \implies (\tau \models \delta(X \rightarrow \text{including}(x)))$
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]:($\perp \rightarrow \text{excluding}(x)$) = \perp
by(simp add: bot-fun-def OclExcluding-def defined-def valid-def false-def true-def)

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

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

10.2 Some computational laws:

lemma *including-cha0*[simp]:
assumes *val-x*: $\tau \models (v\ x)$
shows $\tau \models \text{not}(\text{Set}\{\} \rightarrow \text{includes}(x))$
using *val-x*
apply(*auto* *simp*: *OclValid-def* *OclIncludes-def* *not-def* *false-def* *true-def*)
apply(*auto* *simp*: *mtSet-def* *OClib-lib.Set-0.Abs-Set-0-inverse* *Set-0-def*)
done

lemma *including-cha1*:
assumes *def-X*: $\tau \models (\delta\ X)$
assumes *val-x*: $\tau \models (v\ x)$
shows $\tau \models (X \rightarrow \text{including}(x) \rightarrow \text{includes}(x))$
proof –
have *A* : $\perp \in \text{Set-0}$ **by**(*simp* *add*: *Set-0-def* *bot-option-def*)
have *B* : $\lfloor \perp \rfloor \in \text{Set-0}$ **by**(*simp* *add*: *Set-0-def* *null-option-def* *bot-option-def*)
have *C* : $\lfloor \lfloor \text{insert}(x\ \tau) \rfloor \rfloor \text{Rep-Set-0}(X\ \tau) \rfloor \rfloor \in \text{Set-0}$
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-cha2*:
assumes *def-X*: $\tau \models (\delta\ X)$
and *val-x*: $\tau \models (v\ x)$
and *val-y*: $\tau \models (v\ y)$
and *neq* : $\tau \models \text{not}(x \triangleq y)$
shows $\tau \models (X \rightarrow \text{including}(x) \rightarrow \text{includes}(y)) \triangleq (X \rightarrow \text{includes}(y))$
proof –

```

have A :  $\perp \in \text{Set-0}$  by(simp add: Set-0-def bot-option-def)
have B :  $[\perp] \in \text{Set-0}$  by(simp add: Set-0-def null-option-def bot-option-def)
have C :  $[[\text{insert } (x \ \tau) \ [\text{Rep-Set-0 } (X \ \tau)]]] \in \text{Set-0}$ 
      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 neg)
      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:  $\tau \models (\text{Set}\{\mathbf{2}, \text{null}\} \rightarrow \text{includes}(\text{null}))$ 
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:  $\tau \models (\text{Set}\{\text{Set}\{\mathbf{2}\}, \text{null}\} \rightarrow \text{includes}(\text{null}))$ 
oops

```

```

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

```

```

lemma semantic-test:  $\tau \models (\text{Set}\{\text{null}, \mathbf{2}\} \rightarrow \text{includes}(\text{null}))$ 
by(simp-all add: including-cha1 including-valid-args-valid)

```



```

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 - -  $\Rightarrow$  oid)
  instance ..
end

instantiation option :: (object)object
begin

```

```

definition oid-of-option-def: oid-of  $x = \text{oid-of } (the\ x)$ 
instance ..
end

```

12 Instantiation of the generic strict equality

Should be generated entirely from a class-diagram.

```

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

```

```

lemmas strict-eq-node =
  cp-gen-ref-eq-object [of  $x::Node\ y::Node\ \tau$ ,
    simplified StrictRefEq-node[symmetric]]
  cp-intro(9) [of  $P::Node \Rightarrow NodeQ::Node \Rightarrow Node$ ,
    simplified StrictRefEq-node[symmetric] ]
  gen-ref-eq-def [of  $x::Node\ y::Node$ ,
    simplified StrictRefEq-node[symmetric]]
  gen-ref-eq-defargs [of  $- x::Node\ y::Node$ ,
    simplified StrictRefEq-node[symmetric]]
  gen-ref-eq-object-strict1
    [of  $x::Node$ ,
    simplified StrictRefEq-node[symmetric]]
  gen-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::  $Node \Rightarrow Node$  (( $1(-).next$ ) 50)
  where  $(X).next = (\lambda\ \tau. \text{case } X\ \tau\ \text{of}$ 
     $\text{None} \Rightarrow \text{None}$ 
     $| \text{None} \Rightarrow \text{None}$ 
     $| \text{Node } oid\ i\ next \Rightarrow \text{if } next \in \text{dom } (snd\ \tau)$ 
       $\text{then } (snd\ \tau)\ next$ 
       $\text{else } \text{None})$ 

```

```

fun dot-i::  $Node \Rightarrow Integer$  (( $1(-).i$ ) 50)
  where  $(X).i = (\lambda\ \tau. \text{case } X\ \tau\ \text{of}$ 

```

```

      None  $\Rightarrow$  None
    | | None ]  $\Rightarrow$  None
    | [| Node oid i next ] ]  $\Rightarrow$ 
      if oid  $\in$  dom (snd  $\tau$ )
      then (case (snd  $\tau$ ) oid of
        None  $\Rightarrow$  None
        | | Node oid i next ]  $\Rightarrow$  [| i ])
      else None)

fun dot-next-at-pre:: Node  $\Rightarrow$  Node ((1(-).next@pre) 50)
where (X).next@pre = ( $\lambda$   $\tau$ . case X  $\tau$  of
  None  $\Rightarrow$  None
  | | None ]  $\Rightarrow$  None
  | [| Node oid i next ] ]  $\Rightarrow$  if next  $\in$  dom (fst  $\tau$ )
    then | (fst  $\tau$ ) next ]
    else None)

fun dot-i-at-pre:: Node  $\Rightarrow$  Integer ((1(-).i@pre) 50)
where (X).i@pre = ( $\lambda$   $\tau$ . case X  $\tau$  of
  None  $\Rightarrow$  None
  | | None ]  $\Rightarrow$  None
  | [| Node oid i next ] ]  $\Rightarrow$ 
    if oid  $\in$  dom (fst  $\tau$ )
    then (case (fst  $\tau$ ) oid of
      None  $\Rightarrow$  None
      | | Node oid i next ]  $\Rightarrow$  [| i ])
    else None)

lemma cp-dot-next:
  ((X).next)  $\tau$  = (( $\lambda$ -. X  $\tau$ ).next)  $\tau$  by(simp)

lemma cp-dot-i:
  ((X).i)  $\tau$  = (( $\lambda$ -. X  $\tau$ ).i)  $\tau$  by(simp)

lemma cp-dot-next-at-pre:
  ((X).next@pre)  $\tau$  = (( $\lambda$ -. X  $\tau$ ).next@pre)  $\tau$  by(simp)

lemma cp-dot-i-pre:
  ((X).i@pre)  $\tau$  = (( $\lambda$ -. X  $\tau$ ).i@pre)  $\tau$  by(simp)

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

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

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

axiomatization *inv-Node-at-pre* :: *Node* \Rightarrow *Boolean*

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

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

coinductive *inv* :: *Node* \Rightarrow (*node*)*st* \Rightarrow *bool* **where**

$(\tau \models (\delta \text{ self})) \implies ((\tau \models (\text{self}.\text{next} \doteq \text{null})) \vee$
 $(\tau \models (\text{self}.\text{next} <> \text{null}) \wedge (\tau \models (\text{self}.\text{next}.i \prec \text{self}.i)) \wedge$
 $(\text{inv}(\text{self}.\text{next})\tau)))$
 $\implies (\text{inv self } \tau)$

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

```

```

    then  $\tau \models \text{true} \wedge$ 
       $\tau \models (\text{self}).\text{contents}() \triangleq (\text{self}).\text{contents@pre()} \rightarrow \text{including}(x)$ 
    else  $\tau \models (\text{self}).\text{insert}(x) \triangleq \text{invalid}$ 

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

end

```