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

Burkhart Wolff

July 27, 2012

Contents

1 OCL Core Definitions

theory
OCL-core
imports
Main
begin

2 Foundational Notations

2.1 Notations for the option type

First of all, we will use a more compact notation for the library option type which occur all over in our definitions and which will make the presentation more "textbook"-like:

notation *Some* ($\lfloor(\text{-})\rfloor$)
notation *None* ($\perp$)

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

2.2 State, State Transitions, Well-formed States

Next we will introduce the foundational concept of an object id (oid), which is just some infinite set.

type-synonym *oid* = *ind*

States are just a partial map from oid's to elements of an object universe $'\mathfrak{A}$, and state transitions pairs of states...

type-synonym $('\mathfrak{A})state = oid \rightharpoonup '\mathfrak{A}$

type-synonym $('\mathfrak{A})st = '\mathfrak{A}\ state \times '\mathfrak{A}\ state$

In certain contexts, we will require that the elements of the object universe have a particular structure; more precisely, we will require that there is a function that reconstructs the oid of an object in the state (we will settle the question how to define this function later).

class *object* =
  fixes $oid\text{-}of :: 'a \Rightarrow oid$

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

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

2.3 Prerequisite: An Abstract Interface for OCL Types

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

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

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

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

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

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

class $null = bot +$
fixes $null :: 'a$
assumes *null-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*⟩
end

instantiation *option* :: $(bot)null$
begin
definition *null-option-def*: $(null{::}'a{::}bot\ option) \equiv\ \lfloor\ bot\ \rfloor$
instance ⟨*proof*⟩
end

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

instance ⟨*proof*⟩
end

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

instance ⟨*proof*⟩
end

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

2.5 The Semantic Space of OCL Types: Valuations.

Valuations are now functions from a state pair (built upon data universe $'\mathfrak{A}$) to an arbitrary null-type (i.e. containing at least a destinguished *null* and *invalid* element.

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

All OCL expressions *denote* functions that map the underlying

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

As a consequence of semantic domain definition, any OCL type will have the two semantic constants *invalid* (for exceptional, aborted computation) and *null*; the latter, however is either defined

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

The definition :

```
definition null    :: "('\<AA>,'\<alpha>::null) val"
where     "null    \<equiv> \<lambda> \<tau>. null"
```

is not necessary since we defined the entire function space over null types again as null-types; the crucial definition is $null \equiv \lambda x.\ null$.

2.6 Further requirements on States

A key-concept for linking strict referential equality to logical equality: in well-formed states (i.e. those states where the self (oid-of) field contains the pointer to which the object is associated to in the state), referential equality coincides with logical equality.

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

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

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

3 The OCL Base Type Boolean.

4 Boolean Type and Logic

The semantic domain of the (basic) boolean type is now defined as standard: the space of valuation to *bool option option*:

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

4.1 Basic Constants

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

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

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

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

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

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

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

4.2 Fundamental Predicates I: Validity and Definedness

However, this has also the consequence that core concepts like definedness, validness and even cp have to be redefined on this type class:

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

lemma *valid1*[*simp*]: $\upsilon\ invalid = false$
⟨*proof*⟩

lemma *valid2*[*simp*]: $\upsilon\ null = true$
⟨*proof*⟩

lemma *valid3*[*simp*]: $\upsilon\ true = true$
⟨*proof*⟩

lemma *valid4*[*simp*]: υ *false* = *true*
⟨*proof*⟩

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

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

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

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

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

lemma *defined3*[*simp*]: δ *true* = *true*
⟨*proof*⟩

lemma *defined4*[*simp*]: δ *false* = *true*
⟨*proof*⟩

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

lemma *defined6*[*simp*]: $\delta\ \upsilon\ X$ = *true*
⟨*proof*⟩

lemma *defined7*[*simp*]: $\delta\ \delta\ X$ = *true*
⟨*proof*⟩

lemma *valid6*[*simp*]: $\upsilon\ \delta\ X$ = *true*
⟨*proof*⟩

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

4.3 Fundamental Predicates II: Logical (Strong) Equality

Note that we define strong equality extremely generic, even for types that contain an *null* or $\perp$ element:

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

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

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

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

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

... it is only in a limited sense a congruence, at least from the point of view of this semantic theory. The point is that it is only a congruence on OCL- expressions, not arbitrary HOL expressions (with which we can mix Essential OCL expressions. A semantic — not syntactic — characterization of OCL-expressions is that they are *context-passing* or *context-invariant*, i.e. the context of an entire OCL expression, i.e. the pre-and poststate it referes to, is passed constantly and unmodified to the sub-expressions, i.e. all sub-expressions inside an OCL expression refer to the same context. Expressed formally, this boils down to:

lemma *StrongEq-subst* :
assumes *cp*: $\bigwedge X.\ P(X)\tau = P(\lambda\ \text{-}.\ X\ \tau)\tau$
and *eq*: $(X \triangleq Y)\tau = true\ \tau$
shows $(P\ X \triangleq P\ Y)\tau = true\ \tau$
$\langle proof \rangle$

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*

$$\equiv \lambda\ \tau.\ \textit{if}\ (\delta\ x)\ \tau = \textit{true}\ \tau \wedge (\delta\ y)\ \tau = \textit{true}\ \tau$$
$$\textit{then}\ \lfloor\lfloor(\textit{oid-of}\ (x\ \tau)) = (\textit{oid-of}\ (y\ \tau))\ \rfloor\rfloor$$
$$\textit{else invalid}\ \tau$$

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

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

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

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

lemma *cp-gen-ref-eq-object*:
(*gen-ref-eq x y* τ) = (*gen-ref-eq* (λ-. $x\ \tau$) (λ-. $y\ \tau$)) τ
⟨*proof*⟩

And, last but not least,

lemma *defined8* [*simp*]: $\delta\ (X \triangleq Y) = \textit{true}$
⟨*proof*⟩

lemma *valid5* [*simp*]: $\upsilon\ (X \triangleq Y) = \textit{true}$
⟨*proof*⟩

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

4.5 Logical Connectives and their Universal Properties

It is a design goal to give OCL a semantics that is as closely as possible to a "logical system" in a known sense; a specification logic where the logical connectives can not be understood other that having the truth-table aside when reading fails its purpose in our view.

Practically, this means that we want to give a definition to the core operations to be as close as possible to the lattice laws; this makes also powerful symbolic normalizations of OCL specifications possible as a pre-requisite for automated theorem provers. For example, it is still possible to compute without any definedness- and validity reasoning the DNF of an OCL specification; be it for test-case generations or for a smooth transition to a two-valued representation of the specification amenable to fast standard SMT-solvers, for example.

Thus, our representation of the OCL is merely a 4-valued Kleene-Logics with *invalid* as least, *null* as middle and *true* resp. *false* as unrelated top-elements.

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

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

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

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

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

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

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

lemma *not4*[*simp*]: *not false* = *true*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

lemma *ocl-and1*[*simp*]: (*invalid and true*) = *invalid*
⟨*proof*⟩
lemma *ocl-and2*[*simp*]: (*invalid and false*) = *false*
⟨*proof*⟩
lemma *ocl-and3*[*simp*]: (*invalid and null*) = *invalid*
⟨*proof*⟩
lemma *ocl-and4*[*simp*]: (*invalid and invalid*) = *invalid*
⟨*proof*⟩

lemma *ocl-and5*[*simp*]: (*null and true*) = *null*
⟨*proof*⟩
lemma *ocl-and6*[*simp*]: (*null and false*) = *false*
⟨*proof*⟩
lemma *ocl-and7*[*simp*]: (*null and null*) = *null*
⟨*proof*⟩
lemma *ocl-and8*[*simp*]: (*null and invalid*) = *invalid*
⟨*proof*⟩

lemma *ocl-and9*[*simp*]: (*false and true*) = *false*
⟨*proof*⟩
lemma *ocl-and10*[*simp*]: (*false and false*) = *false*
⟨*proof*⟩
lemma *ocl-and11*[*simp*]: (*false and null*) = *false*

⟨*proof*⟩
lemma *ocl-and12*[*simp*]: (*false and invalid*) = *false*
⟨*proof*⟩

lemma *ocl-and13*[*simp*]: (*true and true*) = *true*
⟨*proof*⟩
lemma *ocl-and14*[*simp*]: (*true and false*) = *false*
⟨*proof*⟩
lemma *ocl-and15*[*simp*]: (*true and null*) = *null*
⟨*proof*⟩
lemma *ocl-and16*[*simp*]: (*true and invalid*) = *invalid*
⟨*proof*⟩

lemma *ocl-and-idem*[*simp*]: (X *and* X) = X
⟨*proof*⟩

lemma *ocl-and-commute*: (X *and* Y) = (Y *and* X)
⟨*proof*⟩

lemma *ocl-and-false1*[*simp*]: (*false and* X) = *false*
⟨*proof*⟩

lemma *ocl-and-false2*[*simp*]: (X *and false*) = *false*
⟨*proof*⟩

lemma *ocl-and-true1*[*simp*]: (*true and* X) = X
⟨*proof*⟩

lemma *ocl-and-true2*[*simp*]: (X *and true*) = X
⟨*proof*⟩

lemma *ocl-and-assoc*: (X *and* (Y *and* Z)) = (X *and* Y *and* Z)
⟨*proof*⟩

lemma *ocl-or-idem*[*simp*]: (X *or* X) = X
⟨*proof*⟩

lemma *ocl-or-commute*: (X *or* Y) = (Y *or* X)
⟨*proof*⟩

lemma *ocl-or-false1*[*simp*]: (*false or* Y) = Y
⟨*proof*⟩

lemma *ocl-or-false2*[*simp*]: (Y *or false*) = Y
⟨*proof*⟩

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

$\langle proof \rangle$

lemma *ocl-or-true2*: $(Y\ or\ true) = true$
$\langle proof \rangle$

lemma *ocl-or-assoc*: $(X\ or\ (Y\ or\ Z)) = (X\ or\ Y\ or\ Z)$
$\langle proof \rangle$

lemma *deMorgan1*: $not(X\ and\ Y) = ((not\ X)\ or\ (not\ Y))$
$\langle proof \rangle$

lemma *deMorgan2*: $not(X\ or\ Y) = ((not\ X)\ and\ (not\ Y))$
$\langle proof \rangle$

4.6 A Standard Logical Calculus for OCL

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

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

5 Global vs. Local Judgements

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

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

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

However, certain properties (like transitivity) can not be *transformed* from the global level to the local one, they have to be re-proven on the local level.

lemma *transform3*:
assumes $H : P = true \Longrightarrow Q = true$
shows $\tau \models P \Longrightarrow \tau \models Q$
$\langle proof \rangle$

5.0.1 Local Validity and Meta-logic

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *foundation12*:
$\tau \models \delta\ x \Longrightarrow\ \ \tau \models \delta\ y \Longrightarrow (\tau \models (x\ implies\ y)) = (\ (\tau \models x) \longrightarrow (\tau \models y))$
$\langle proof \rangle$

lemma *foundation13*:$(\tau \models A \triangleq true) \quad = (\tau \models A)$
$\langle proof \rangle$

lemma *foundation14*:$(\tau \models A \triangleq false) \quad = (\tau \models not\ A)$
$\langle proof \rangle$

lemma *foundation15*:$(\tau \models A \triangleq invalid) = (\tau \models not(\upsilon\ A))$
$\langle proof \rangle$

lemma *foundation16*: $\tau \models (\delta\ X) = (X\ \tau \neq bot \wedge X\ \tau \neq null)$
$\langle proof \rangle$

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

lemma *foundation18*: $\tau \models (\upsilon\ X) = (X\ \tau \neq bot)$
$\langle proof \rangle$

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

lemma *foundation20* : $\tau \models (\delta\ X) \Longrightarrow \tau \models \upsilon\ X$
$\langle proof \rangle$

theorem *strictEqGen-vs-strongEq*:
$WFF\ \tau \Longrightarrow \tau \models(\delta\ x) \Longrightarrow \tau \models(\delta\ y) \Longrightarrow$
$(\tau \models (gen\text{-}ref\text{-}eq\ x\ y)) = (\tau \models (x \triangleq y))$
$\langle proof \rangle$

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*: $\tau \models (x \triangleq x)$
$\langle proof \rangle$

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

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

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

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

The rule of substitutivity in HOL-OCL holds only for context-passing expressions - i.e. those, that pass the context τ without changing it. Fortunately, all operators of the OCL language satisfy this property (but not all HOL operators).

lemma *StrongEq-L-subst1*: $\bigwedge\ \tau.\ cp\ P \Longrightarrow \tau \models (x \triangleq y) \Longrightarrow \tau \models (P\ x \triangleq P\ y)$
⟨*proof*⟩

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

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

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

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

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

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

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

7 Laws to Establish Definedness (Delta-Closure)

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

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

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

8 Miscellaneous: OCL's if then else endif

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

lemma *if-ocl-invalid* : $(if\ invalid\ then\ B_1\ else\ B_2\ endif) = invalid$
⟨*proof*⟩

lemma *if-ocl-null* : $(if\ null\ then\ B_1\ else\ B_2\ endif) = invalid$
⟨*proof*⟩

lemma *if-ocl-true* : $(if\ true\ then\ B_1\ else\ B_2\ endif) = B_1$
⟨*proof*⟩

lemma *if-ocl-false* : $(if\ false\ then\ B_1\ else\ B_2\ endif) = B_2$
⟨*proof*⟩

end

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

9 Simple, Basic Types like Void, Boolean and Integer

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

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

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

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

10 Strict equalities.

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

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

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

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

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

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

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

lemma *strictEqBool-defargs*:

$\tau \models ((x::(\prime\mathfrak{A})Boolean) \doteq y) \Longrightarrow (\tau \models (\upsilon\ x)) \wedge (\tau \models (\upsilon\ y))$
$\langle proof \rangle$

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

lemma *strictEqBool-valid-args-valid*:
$(\tau \models \upsilon((x::(\prime\mathfrak{A})Boolean) \doteq y)) = ((\tau \models (\upsilon\ x)) \wedge (\tau \models (\upsilon\ y)))$
$\langle proof \rangle$

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

lemma *gen-ref-eq-defargs*:
$\tau \models (gen\text{-}ref\text{-}eq\ x\ (y::(\prime\mathfrak{A},\prime a::\{null,object\})val)) \Longrightarrow (\tau \models (\delta\ x)) \wedge (\tau \models (\delta\ y))$
$\langle proof \rangle$

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

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

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

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

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

lemma *cp-StrictRefEq-int*:
$((X::(\prime\mathfrak{A})Integer) \doteq Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau) \doteq (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
$\langle proof \rangle$

lemmas *cp-intro*[*simp*,*intro*!] =
cp-intro
cp-StrictRefEq-bool[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of StrictRefEq*]]
cp-StrictRefEq-int[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of StrictRefEq*]]

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

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

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

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

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

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

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

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

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

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

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

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

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

Isabelle.

lemma δ *null* = *false* ⟨*proof*⟩
lemma υ *null* = *true* ⟨*proof*⟩

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

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

lemma [*simp*]:δ **1** = *true*
⟨*proof*⟩

lemma [*simp*]:υ **1** = *true*
⟨*proof*⟩

lemma [*simp*]:δ **2** = *true*
⟨*proof*⟩

lemma [*simp*]:υ **2** = *true*
⟨*proof*⟩

lemma *one-non-null*[*simp*]: **0** $\neq$ *null*
⟨*proof*⟩

lemma *zero-non-null*[*simp*]: **1** $\neq$ *null*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

typedef $'\alpha$ *Set-0* $= \{X::('\alpha::null)$ *set option option*.
$X = bot \lor X = null \lor (\forall x \in \lceil\lceil X \rceil\rceil.\ x \neq bot)\}$
$\langle proof \rangle$

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

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

instance $\langle proof \rangle$
end

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

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

instance $\langle proof \rangle$
end

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

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

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

... which means that we can have a type $('\mathfrak{A},('\mathfrak{A},('\mathfrak{A})$ *Integer*) *Set*) *Set* corresponding exactly to Set(Set(Integer)) in OCL notation. Note that the parameter $\mathfrak{A}$ still refers to the object universe; making the OCL semantics entirely parametric in the object universe makes it possible to study (and prove) its properties independently from a concrete class diagram.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

consts

$OclUnion$ :: $[('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow ('\mathfrak{A},'\alpha)\ Set$
$OclIntersection$ :: $[('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow ('\mathfrak{A},'\alpha)\ Set$
$OclIncludesAll$ :: $[('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow '\mathfrak{A}\ Boolean$
$OclExcludesAll$ :: $[('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow '\mathfrak{A}\ Boolean$
$OclComplement$ :: $('\mathfrak{A},'\alpha::null)\ Set \Rightarrow ('\mathfrak{A},'\alpha)\ Set$
$OclSum$ :: $('\mathfrak{A},'\alpha::null)\ Set \Rightarrow '\mathfrak{A}\ Integer$
$OclCount$ :: $[('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow '\mathfrak{A}\ Integer$

notation
$OclSize$ $(\text{-}\text{-}{>}size'(')\ [66])$

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

lemma *cp-OclIncluding*:
$(X\text{->}including(x))\ \tau = ((\lambda\ \text{-}.\ X\ \tau)\text{->}including(\lambda\ \text{-}.\ x\ \tau))\ \tau$
⟨*proof*⟩

lemma *cp-OclExcluding*:
$(X\text{->}excluding(x))\ \tau = ((\lambda\ \text{-}.\ X\ \tau)\text{->}excluding(\lambda\ \text{-}.\ x\ \tau))\ \tau$
⟨*proof*⟩

lemma *cp-OclIncludes*:
$(X\text{->}includes(x))\ \tau = (OclIncludes\ (\lambda\ \text{-}.\ X\ \tau)\ (\lambda\ \text{-}.\ x\ \tau)\ \tau)$
⟨*proof*⟩

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

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

lemma *including-strict3*[*simp*]:$(null{-}{>}including(x)) = \bot$
⟨*proof*⟩

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

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

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

lemma *excluding-strict3*[*simp*]:$(null{-}{>}excluding(x)) = \bot$
⟨*proof*⟩

10.2 Some computational laws:

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

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

lemma *including-charn2*:
assumes *def-X*:$\tau \models (\delta\ X)$
and *val-x*:$\tau \models (\upsilon\ x)$
and *val-y*:$\tau \models (\upsilon\ y)$
and *neq* :$\tau \models not(x \triangleq y)$
shows $\tau \models (X{-}{>}including(x){-}{>}includes(y)) \triangleq (X{-}{>}includes(y))$
⟨*proof*⟩

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

translations
$Set\{x, xs\}$ == CONST OclIncluding $(Set\{xs\})$ x
$Set\{x\}$ == CONST OclIncluding $(Set\{\})$ x

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

lemma *semantic-test*: $\tau \models (Set\{\mathbf{2},null\}{-}{>}includes(null))$
⟨*proof*⟩

Here is an example of a nested collection. Note that we have to use the abstract null (since we did not (yet) define a concrete constant *null* for the non-existing Sets) :

lemma *semantic-test*: $\tau \models (Set\{Set\{\mathbf{2}\},null\}{-}{>}includes(null))$
⟨*proof*⟩

lemma *hurx* : $\tau \models Set\{Set\{\mathbf{2}\},null\} \triangleq Set\{null,Set\{\mathbf{2}\}\}$
⟨*proof*⟩

lemma *semantic-test*: $\tau \models (Set\{null,\mathbf{2}\}{-}{>}includes(null))$
⟨*proof*⟩

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

lemma $OclIterate_{Set}$-*strict1* [*simp*]: $(OclIterate_{Set}\ \bot\ A\ F) = \bot$
⟨*proof*⟩

lemma $OclIterate_{Set}$-*null1* [*simp*]: $(OclIterate_{Set}\ null\ A\ F) = \bot$
⟨*proof*⟩

lemma $OclIterate_{Set}$-*strict2* [*simp*]: $(OclIterate_{Set}\ S\ \bot\ F) = \bot$
⟨*proof*⟩

An open question is this ...

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

In the definition above, this does not hold in general. And I believe, this is how it should be ...

lemma *OclIterate*$_{Set}$*-infinite*:
assumes *non-finite*: $\tau \models not(\delta(S{-}{>}size()))$
shows $(OclIterate_{Set}\ S\ A\ F) = \bot$
⟨*proof*⟩

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

In particular, this does hold for A = null.

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

end

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

end

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

end

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

11 Example Data-Universe

Should be generated entirely from a class-diagram.

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

datatype *node* = *Node oid*
int
oid

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

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

instantiation *option* :: (*object*)*object*
begin
definition *oid-of-option-def*: *oid-of x* = *oid-of* (*the x*)
instance ⟨*proof*⟩
end

12 Instantiation of the generic strict equality

Should be generated entirely from a class-diagram.

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

lemmas *strict-eq-node* =
cp-gen-ref-eq-object[*of x*::*Node y*::*Node* τ,
simplified StrictRefEq-node[*symmetric*]]
cp-intro(*9*) [*of P*::*Node* ⇒*NodeQ*::*Node* ⇒*Node*,
simplified StrictRefEq-node[*symmetric*]]
gen-ref-eq-def [*of x*::*Node y*::*Node*,
simplified StrictRefEq-node[*symmetric*]]
gen-ref-eq-defargs [*of - x*::*Node y*::*Node*,
simplified StrictRefEq-node[*symmetric*]]
gen-ref-eq-object-strict1
[*of x*::*Node*,
simplified StrictRefEq-node[*symmetric*]]
gen-ref-eq-object-strict2
[*of x*::*Node*,
simplified StrictRefEq-node[*symmetric*]]
gen-ref-eq-object-strict3
[*of x*::*Node*,
simplified StrictRefEq-node[*symmetric*]]
gen-ref-eq-object-strict3
[*of x*::*Node*,

simplified StrictRefEq-node[*symmetric*]]
gen-ref-eq-object-strict4
[*of x*::*Node*,
simplified StrictRefEq-node[*symmetric*]]

13 Selector Definition

Should be generated entirely from a class-diagram.

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

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

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

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

lemma *cp-dot-next*:
((*X*).*next*) τ = ((λ-. *X* τ).*next*) τ ⟨*proof*⟩

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

lemma *cp-dot-next-at-pre*:
$((X).next@pre)\ \tau = ((\lambda\text{-}.\ X\ \tau).next@pre)\ \tau$ ⟨*proof*⟩

lemma *cp-dot-i-pre*:
$((X).i@pre)\ \tau = ((\lambda\text{-}.\ X\ \tau).i@pre)\ \tau$ ⟨*proof*⟩

lemmas *cp-dot-nextI* [*simp*, *intro*!]=
cp-dot-next[*THEN allI*[*THEN allI*], *of* λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

lemmas *cp-dot-nextI-at-pre* [*simp*, *intro*!]=
cp-dot-next-at-pre[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

lemma *dot-next-nullstrict* [*simp*]: $(null).next = invalid$
⟨*proof*⟩

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

lemma *dot-next-strict*[*simp*] : $(invalid).next = invalid$
⟨*proof*⟩

lemma *dot-nextATpre-strict*[*simp*] : $(invalid).next@pre = invalid$
⟨*proof*⟩

14 Standard State Infrastructure

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

15 Invariant

These recursive predicates can be defined conservatively by greatest fix-point constructions - automatically. See HOL-OCL Book for details. For the purpose of this example, we state them as axioms here.

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

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

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

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

axiomatization $\mathit{dot\text{-}contents\text{-}def}$ **where**
$(\tau \models ((\mathit{self}).\mathit{contents}() \triangleq \mathit{result})) =$
$(\mathit{if}\ \ (\delta\ \mathit{self})\ \tau = \mathit{true}\ \tau$
$\mathit{then}\ ((\tau \models \mathit{true}) \wedge$
$(\tau \models (\mathit{result} \triangleq \mathit{if}\ \ (\mathit{self}\ .\mathit{next} \doteq \mathit{null})$
$\mathit{then}\ (\mathit{Set}\{\mathit{self}\ .\mathit{i}\})$
$\mathit{else}\ (\mathit{self}\ .\mathit{next}\ .\mathit{contents}() {-}{>} \mathit{including}(\mathit{self}\ .\mathit{i}))$
$\mathit{endif})))$
$\mathit{else}\ \tau \models \mathit{result} \triangleq \mathit{invalid})$

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

axiomatization where $\mathit{dot\text{-}contents\text{-}AT\text{-}pre\text{-}def}$:
$(\tau \models (\mathit{self}).\mathit{contents@pre}() \triangleq \mathit{result}) =$
$(\mathit{if}\ \ (\delta\ \mathit{self})\ \tau = \mathit{true}\ \tau$
$\mathit{then}\ \tau \models \mathit{true} \wedge \qquad\qquad (*\ \mathit{pre}\ *)$
$\tau \models (\mathit{result} \triangleq \mathit{if}\ \ (\mathit{self}).\mathit{next@pre} \doteq \mathit{null}\ \ (*\ \mathit{post}\ *)$
$\mathit{then}\ \mathit{Set}\{(\mathit{self}).\mathit{i@pre}\}$

$else\ (self).next@pre\ .contents@pre()->including(self\ .i@pre)$
$endif)$
$else\ \tau \models result \triangleq invalid)$

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 $dot\text{-}insert :: Node \Rightarrow Integer \Rightarrow Void\ \ ((1(\text{-}).insert'(\text{-}'))\ 50)$

axiomatization where $dot\text{-}insert\text{-}def$:
$(\tau \models (self).insert(x) \triangleq result) =$
$(if\ (\delta\ self)\ \tau = true\ \tau \wedge (\upsilon\ x)\ \tau = true\ \tau$
$then\ \tau \models true\ \wedge$
$\tau \models (self).contents() \triangleq (self).contents@pre()->including(x)$
$else\ \tau \models (self).insert(x) \triangleq invalid)$

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

end