A. Overview of the OCL Semantics

A.1. Introduction

This annex formally defines the semantics of OCL. It will proceed by describing the OCL semantics by a translation into a core language — called FeatherweightOCL— which has in itself a formally described semantics presented in Isabelle/HOL [25] [1]. The semantic definitions are in large parts executable, in some parts only provable, namely the essence of Set-constructions. The first goal of its construction is *consistency*, i. e. it should be possible to apply logical rules and/or evaluation rules for OCL in an arbitrary manner always yielding the same result. Moreover, except in pathological cases, this result should be unambiguously defined, i. e. represent a value.

In order to motivate the need for logical consistency and also the magnitude of the problem, we focus on one particular feature of the language as example: `Tuples`. Recall that tuples (in other languages known as *records*) are n-ary cartesian products with named components, where the component names are used also as projection functions: the special case `Pair{x:First, y:Second}` stands for the usual binary pairing operator `Pair{true,null}` and the two projection functions `x.First()` and `x.Second()`. For a developer of a compiler or proof-tool (based on, say, a connection to an SMT solver designed to animate OCL contracts) it would be natural to add the rules `Pair{X,Y}.First() = X` and `Pair{X,Y}.Second() = Y` to give pairings the usual semantics. At some place, the OCL Standard requires the existance of a constant symbol `invalid` and requires all operators to be strict. To implement this, the developer might be tempted to add a generator for corresponding strictness axioms, producing among hundreds of other rules `Pair{invalid,Y}=invalid`,`Pair{X,invalid}=invalid`, `invalid.First()=invalid`, `invalid.Seco` etc. Unfortunately, this "natural" axiomatization of pairing and projection together with strictness is already inconsistent. One can derive:

```
  Pair{true,invalid}.First() = invalid.First() = invalid
```

and:

```
  Pair{true,invalid}.First() = true
```

which then results in the absurd logical consequence that `invalid = true`. Obviously, we need to be more careful on the side-conditions of our rules[2]. And obviously, only a mechanized check of these definitions, following a rigourous methodology, can establish strong guarantees for logical consistency of the OCL language.

This leads us to our second goal of this annex: it should not only be usable by logicians, but also by developers of compilers and proof-tools. For this end, we *derived* from the Isabelle definitions also *logical rules* allowing

[1] An updated, machine-checked version and formally complete version of this document is maintained by the Isabelle Archive of Formal Proofs (AFP), see http://afp.sourceforge.net/entries/Featherweight_OCL.shtml

[2] The solution to this little riddle can be found in Section B.2.7.

formal interactive and automated proofs on UML/OCL specifications, as well as *execution rules* and *test-cases* revealing corner-cases resulting from this semantics which give vital information for the implementor.

OCL is an annotation language for UML models, in particular class models allowing for specifying data and operations on them. As such, it is a *typed* object-oriented language. This means that it is — like Java or C++ — based on the concept of a *static type*, that is the type that the type-checker infers from a UML class model and its OCL annotation, as well as a *dynamic type*, that is the type at which an object is dynamically created [3]. Types are not only a means for efficient compilation and a support of separation of concerns in programming, there are of fundamental importance for our goal of logical consistency: it is impossible to have sets that contain themselves, i.e. to state Russels Paradox in OCL typed set-theory. Moreover, object-oriented typing means that types there can be in sub-typing relation; technically speaking, this means that they can be *casted* via `oclIsTypeOf(T)` one to the other, and under particular conditions to be described in detail later, these casts are semantically *lossless*, i. e.

$$(X.oclAsType(C_j).oclAsType(C_i) = X) \tag{A.1}$$

(where C_j and C_i are class types.) Furthermore, object-orientedness means that operations and object-types can be grouped to *classes* on which an inheritance relation can be established; the latter induces a sub-type relation between the corresponding types.

Here is a feature-list of FeatherweightOCL:

- it specifies key built-in types such as `Boolean`, `Void`, `Integer`, `Real` and `String` as well as generic types such as `Pair(T,T')`, `Sequence(T)` and `Set(T)`.
- it defines the semantics of the operations of these types in *denotational form* — see explanation below —, and thus in an unambiguous (and in Isabelle/HOL executable or animatable) way.
- it develops the *theory* of these definitions, i.e. the collection of lemmas and theorems that can be proven from these definitions.
- all types in FeatherweightOCL contain the elements `null` and `invalid`; since this extends to `Boolean` type, this results in a four-valued logic. Consequently, FeatherweightOCL contains the derivation of the *logic* of OCL.
- collection types may contain `null` (so `Set{null}` is a defined set) but not `invalid` (`Set{invalid}` is just `invalid`).
- Wrt. to the static types, FeatherweightOCL is a strongly typed language in the Hindley-Milner tradition. We assume that a pre-process for full OCL eliminates all implicit conversions due to subtyping by introducing explicit casts (e. g., `oclAsType(Class)`). [4]
- FeatherweightOCL types may be arbitrarily nested. For example, the expression `Set{Set{1,2}} = Set{Set{2` is legal and true.

[3] As side-effect free language, OCL has no object-constructors, but with `OclIsNew()`, the effect of object creation can be expressed in a declarative way.

[4] The details of such a pre-processing are described in [4].

- All objects types are represented in an object universe[5]. The universe construction also gives semantics to type casts, dynamic type tests, as well as functions such as `oclAllInstances()`, or `oclIsNew()`. The object universe onstruction is conceptually described and demonstrated at an example.

- As part of the OCL logic, FeatherweightOCL develops the theory of equality in UML/OCL. This includes the standard equality, which is a computable strict equality using the object references for comparison, and the not necessarily computable logical equality, which expresses the Leibniz principle that 'equals may be replaced by equals' in OCL terms.

- Technically, FeatherweightOCL is a *semantic embedding* into a powerful semantic meta-language and environment, namely Isabelle/HOL [25]. It is a so-called *shallow embedding* in HOL; this means that types in OCL were *injectively* represented by types in Isabelle/HOL. Ill-typed OCL specifications cannot therefore not be represented in FeatherweightOCL and a type in FeatherweightOCL contains exactly the values that are possible in OCL .

Context. This document stands in a more than fifteen years tradition of giving a formal semantics to the core of UML and its annotation language OCL, starting from Richters [30] and [17, 20, 24], leading to a number of formal, machine-checked versions, most notably HOL-OCL [5, 6, 9] and more recent approaches [14]. All of them have in common the attempt to reconcile the conflicting demands of an industrially used specification language and its various stakeholders, the needs of OMG standardization process and the desire for sufficient logical precision for tool-implementors, in particular from the Formal Methods research community.

FiXme: *Something like this ? Shorten Paragraph !*

To discuss the future directions of the standard, several OCL experts met in November 2013 in Aachen to discuss possible mid-term improvements of OCL, strategies of standardization of OCL within the OMG, and a vision for possible long-term developments of the language [13]. During this meeting, a Request for Proposals (RFP) for OCL 2.5 was finalized and meanwhile proposed. In particular, this RFP requires that the future OCL 2.5 standard document shall be generated from a machine-checked source. This will ensure

- the absence of syntax errors,

- the consistency of the formal semantics,

- a suite of corner-cases relevant for OCL tool implementors.

Organization of this document. This document is organized as follows. After a brief background section introducing a running example and basic knowledge on Isabelle/HOL and its formal notations, we present the formal semantics of FeatherweightOCL introducing:

1. A conceptual description of the formal semantics, highlighting the essentials and avoiding the definitions in detail.

2. A detailed formal description. This covers:
    a) OCL Types and their presentation in Isabelle/HOL,

[5] following the tradition of HOL-OCL [6]

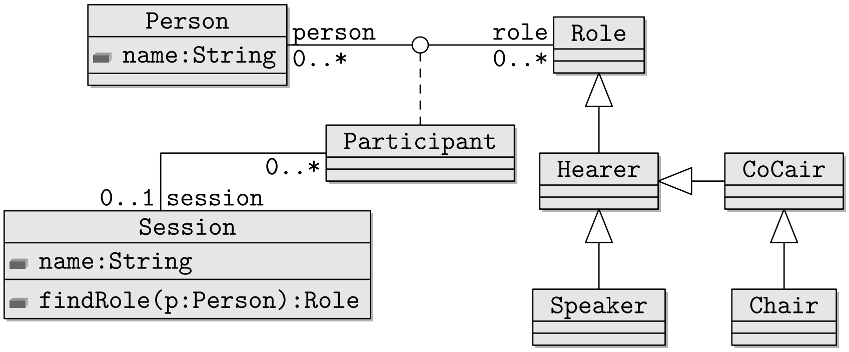


Figure A.1.: A simple UML class model representing a conference system for organizing conference sessions: persons can participate, in different roles, in a session.

b) OCL Terms, i. e. the semantics of library operators, together with definitions, lemmas, and test cases for the implementor,

c) UML/OCL Constructs, i. e. a core of UML class models plus user-defined constructions on them such as class-invariants and oepration constracts.

3. Since the latter, i. e. the construction of UML class models, has to be done on the meta-level (so not *inside* HOL, rather on the level of a pre-compiler), we will describe this process with two larger examples, namely formalizations of our running example.

A.2. Background

A.2.1. A Running Example for UML/OCL

FiXme: *REWRITE THIS FOR THE ANNEX A: SHORTEN!*

The Unified Modeling Language (UML) [26, 27] comprises a variety of model types for describing static (e. g., class models, object models) and dynamic (e. g., state-machines, activity graphs) system properties. One of the more prominent model types of the UML is the *class model* (visualized as *class diagram*) for modeling the underlying data model of a system in an object-oriented manner. As a running example, we model a part of a conference management system. Such a system usually supports the conference organizing process, e. g., creating a conference Website, reviewing submissions, registering attendees, organizing the different sessions and tracks, and indexing and producing the resulting proceedings. In this example, we constrain ourselves to the process of organizing conference sessions; Figure A.1 shows the class model. We model the hierarchy of roles of our system as a hierarchy of classes (e. g., `Hearer`, `Speaker`, or `Chair`) using an *inheritance* relation (also called *generalization*). In particular, *inheritance* establishes a *subtyping* relationship, i. e., every `Speaker` (*subclass*) is also a `Hearer` (*superclass*).

A class does not only describe a set of *instances* (called *objects*), i. e., record-like data consisting of *attributes* such as `name` of class `Session`, but also *operations* defined over them. For example, for the class `Session`, representing a conference session, we model an operation `findRole(p:Person):Role` that should return the role of a `Person` in the context of a specific session; later, we will describe the behavior of this operation in more detail using UML . In the following, the term object describes a (run-time) instance of a class or one of its subclasses.

Relations between classes (called *associations* in UML) can be represented in a class diagram by connecting lines, e. g., `Participant` and `Session` or `Person` and `Role`. Associations may be labeled by a particular constraint called *multiplicity*, e. g., `0..*` or `0..1`, which means that in a relation between participants and sessions, each `Participant` object is associated to at most one `Session` object, while each `Session` object may be associated to arbitrarily many `Participant` objects. Furthermore, associations may be labeled by projection functions like `person` and `role`; these implicit function definitions allow for OCL-expressions like `self.person`, where `self` is a variable of the class `Role`. The expression `self.person` denotes persons being related to the specific object `self` of type role. A particular feature of the UML are *association classes* (`Participant` in our example) which represent a concrete tuple of the relation within a system state as an object; i. e., associations classes allow also for defining attributes and operations for such tuples. In a class diagram, association classes are represented by a dotted line connecting the class with the association. Associations classes can take part in other associations. Moreover, UML supports also *n*-ary associations (not shown in our example).

We refine this data model using the Object Constraint Language (OCL) for specifying additional invariants, preconditions and postconditions of operations. For example, we specify that objects of the class `Person` are uniquely determined by the value of the `name` attribute and that the attribute `name` is not equal to the empty string (denoted by `''`):

```
context Person
  inv: name <> '' and
       Person::allInstances()->isUnique(p:Person | p.name)
```

Moreover, we specify that every session has exactly one chair by the following invariant (called `onlyOneChair`) of the class `Session`:

```
context Session
  inv onlyOneChair: self.participants->one( p:Participant |
                                      p.role.oclIsTypeOf(Chair))
```

where `p.role.oclIsTypeOf(Chair)` evaluates to true, if `p.role` is of *dynamic type* `Chair`. Besides the usual *static types* (i. e., the types inferred by a static type inference), objects in UML and other object-oriented languages have a second *dynamic* type concept. This is a consequence of a family of *casting functions* (written $o_{[C]}$ for an object *o* into another class type *C*) that allows for converting the static type of objects along the class hierarchy. The dynamic type of an object can be understood as its "initial static type" and is unchanged by casts. We complete our example by describing the behavior of the operation `findRole` as follows:

```
context Session::findRole(person:Person):Role
  pre:  self.participates.person->includes(person)
  post: result=self.participants->one(p:Participant |
                                p.person = person ).role
        and self.participants = self.participants@pre
        and self.name = self.name@pre
```

where in post-conditions, the operator `@pre` allows for accessing the previous state.

In UML, classes can contain attributes of the type of the defining class. Thus, UML can represent (mutually) recursive datatypes. Moreover, OCL introduces also recursively specified operations.

A key idea of defining the semantics of UML and extensions like SecureUML [10] is to translate the diagrammatic UML features into a combination of more elementary features of UML and OCL expressions [19]. For example, associations are usually represented by collection-valued class attributes together with OCL constraints expressing the multiplicity. Thus, having a semantics for a subset of UML and OCL is tantamount for the foundation of the entire method.

A.2.2. Formal Foundation

Isabelle

Isabelle [25] is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and natural deduction inference rules. Among other logics, Isabelle supports first-order logic, Zermelo-Fraenkel set theory and the instance for Church's higher-order logic (HOL).

Isabelle's inference rules are based on the built-in meta-level implication $\_\Longrightarrow\_$ allowing to form constructs like $A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow A_{n+1}$, which are viewed as a *rule* of the form "from assumptions A_1 to A_n, infer conclusion A_{n+1}" and which is written in Isabelle as

$$[\![A_1;\ldots;A_n]\!] \Longrightarrow A_{n+1} \qquad \text{or, in mathematical notation,} \quad \frac{A_1 \qquad \cdots \qquad A_n}{A_{n+1}}\,. \tag{A.2}$$

The built-in meta-level quantification $\bigwedge x.\ x$ captures the usual side-constraints "x must not occur free in the assumptions" for quantifier rules; meta-quantified variables can be considered as "fresh" free variables. Meta-level quantification leads to a generalization of Horn-clauses of the form:

$$\bigwedge x_1,\ldots,x_m.\ [\![A_1;\ldots;A_n]\!] \Longrightarrow A_{n+1}\,. \tag{A.3}$$

Isabelle supports forward- and backward reasoning on rules. For backward-reasoning, a *proof-state* can be initialized and further transformed into others. For example, a proof of ϕ, using the Isar [33] language, will look as follows in Isabelle:

```
lemma label:   φ
  apply(case_tac)
  apply(simp_all)
done
```

(A.4)

This proof script instructs Isabelle to prove ϕ by case distinction followed by a simplification of the resulting proof state. Such a proof state is an implicitly conjoint sequence of generalized Horn-clauses (called *subgoals*) $\phi_1,\ldots,\phi_n$ and a *goal* ϕ. Proof states were usually denoted by:

$$\begin{array}{rl} \text{label}: & \phi \\ 1. & \phi_1 \\ & \vdots \\ n. & \phi_n \end{array} \tag{A.5}$$

Subgoals and goals may be extracted from the proof state into theorems of the form $[\![\phi_1;\ldots;\phi_n]\!] \Longrightarrow \phi$ at any time; this mechanism helps to generate test theorems. Further, Isabelle supports meta-variables (written $?x, ?y, \ldots$), which can be seen as "holes in a term" that can still be substituted. Meta-variables are instantiated by Isabelle's built-in higher-order unification.

Higher-order Logic (HOL)

Higher-order logic (HOL) [1, 15] is a classical logic based on a simple type system. It provides the usual logical connectives like $\_ \wedge \_$, $\_ \rightarrow \_$, $\neg \_$ as well as the object-logical quantifiers $\forall x.\ P\,x$ and $\exists x.\ P\,x$; in contrast to first-order logic, quantifiers may range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centered around extensional equality $\_ = \_ :: \alpha \Rightarrow \alpha \Rightarrow \text{bool}$. HOL is more expressive than first-order logic, since, e. g., induction schemes can be expressed inside the logic. Being based on some polymorphically typed λ-calculus, HOL can be viewed as a combination of a programming language like SML or Haskell and a specification language providing powerful logical quantifiers ranging over elementary and function types.

Isabelle/HOL is a logical embedding of HOL into Isabelle. The (original) simple-type system underlying HOL has been extended by Hindley-Milner style polymorphism with type-classes similar to Haskell. While Isabelle/HOL is usually seen as proof assistant, we use it as symbolic computation environment. Implementations on top of Isabelle/HOL can re-use existing powerful deduction mechanisms such as higher-order resolution, tableaux-based reasoners, rewriting procedures, Presburger arithmetic, and via various integration mechanisms, also external provers such as Vampire [29] and the SMT-solver Z3 [18].

Isabelle/HOL offers support for a particular methodology to extend given theories in a logically safe way: A theory-extension is *conservative* if the extended theory is consistent provided that the original theory was consistent. Conservative extensions can be *constant definitions*, *type definitions*, *datatype definitions*, *primitive recursive definitions* and *wellfounded recursive definitions*.

For instance, the library includes the type constructor $\tau_\bot := \bot \mid \lfloor\_\rfloor : \alpha$ that assigns to each type τ a type $\tau_\bot$ *disjointly extended* by the exceptional element $\bot$. The function $\ulcorner\_\urcorner : \alpha_\bot \rightarrow \alpha$ is the inverse of $\lfloor\_\rfloor$ (unspecified for $\bot$). Partial functions $\alpha \rightharpoonup \beta$ are defined as functions $\alpha \Rightarrow \beta_\bot$ supporting the usual concepts of domain (dom _) and range (ran _).

As another example of a conservative extension, typed sets were built in the Isabelle libraries conservatively on top of the kernel of HOL as functions to bool; consequently, the constant definitions for membership is as follows:[6]

$$\begin{array}{lrll}
\text{types} & \alpha\,\text{set} & = \alpha \Rightarrow \text{bool} & \\
\text{definition} & \text{Collect} & :: (\alpha \Rightarrow \text{bool}) \Rightarrow \alpha\,\text{set} & \text{— set comprehension} \\
\text{where} & \text{Collect}\ S & \equiv S & \\
\text{definition} & \text{member} & :: \alpha \Rightarrow \alpha \Rightarrow \text{bool} & \text{— membership test} \\
\text{where} & \text{member}\ s\,S & \equiv S s &
\end{array} \tag{A.6}$$

Isabelle's syntax engine is instructed to accept the notation $\{x \mid P\}$ for Collect $\lambda x.\ P$ and the notation $s \in S$ for member $s\,S$. As can be inferred from the example, constant definitions are axioms that introduce a fresh constant symbol by some closed, non-recursive expressions; this type of axiom is logically safe since it works like an abbreviation. The syntactic side conditions of this axiom are mechanically checked, of course. It is

[6] To increase readability, we use a slightly simplified presentation.

straightforward to express the usual operations on sets like $\_\cup\_, \_\cap\_ :: \alpha\,\mathrm{set} \Rightarrow \alpha\,\mathrm{set} \Rightarrow \alpha\,\mathrm{set}$ as conservative extensions, too, while the rules of typed set theory were derived by proofs from these definitions.

Similarly, a logical compiler is invoked for the following statements introducing the types option and list:

$$\begin{array}{lll} \textbf{datatype} & \mathrm{option} & = \mathrm{None} \mid \mathrm{Some}\,\alpha \\ \textbf{datatype} & \alpha\,\mathrm{list} & = \mathrm{Nil} \mid \mathrm{Cons}\,a\,l \end{array} \tag{A.7}$$

Here, $[]$ or $a\#l$ are an alternative syntax for Nil or Cons $a\ l$; moreover, $[a,b,c]$ is defined as alternative syntax for $a\#b\#c\#[]$. These (recursive) statements were internally represented in by internal type and constant definitions. Besides the *constructors* None, Some, $[]$ and Cons, there is the match operation

$$\mathrm{case}\ x\ \mathrm{of}\ \ \mathrm{None} \Rightarrow F\ \mid \mathrm{Some}\,a \Rightarrow G\,a \tag{A.8}$$

respectively

$$\mathrm{case}\ x\ \mathrm{of}\ \ [] \Rightarrow F\ \mid \mathrm{Cons}\ a\,r \Rightarrow G\,a\,r\,. \tag{A.9}$$

From the internal definitions (not shown here) several properties were automatically derived. We show only the case for lists:

$$\begin{array}{ll} (\mathrm{case}\ []\ \mathrm{of}\ [] \Rightarrow F \mid (a\#r) \Rightarrow G\,a\,r) = F & \\ (\mathrm{case}\ b\#t\ \mathrm{of}\ [] \Rightarrow F \mid (a\#r) \Rightarrow G\,a\,r) = G\ b\ t & \\ [] \neq a\#t & \text{– distinctness} \\ [\![a = [] \to P; \exists\, x\, t\,.\ a = x\#t \to P]\!] \Longrightarrow P & \text{– exhaust} \\ [\![P[]; \forall\, at\,.\ Pt \to P(a\#t)]\!] \Longrightarrow Px & \text{– induct} \end{array} \tag{A.10}$$

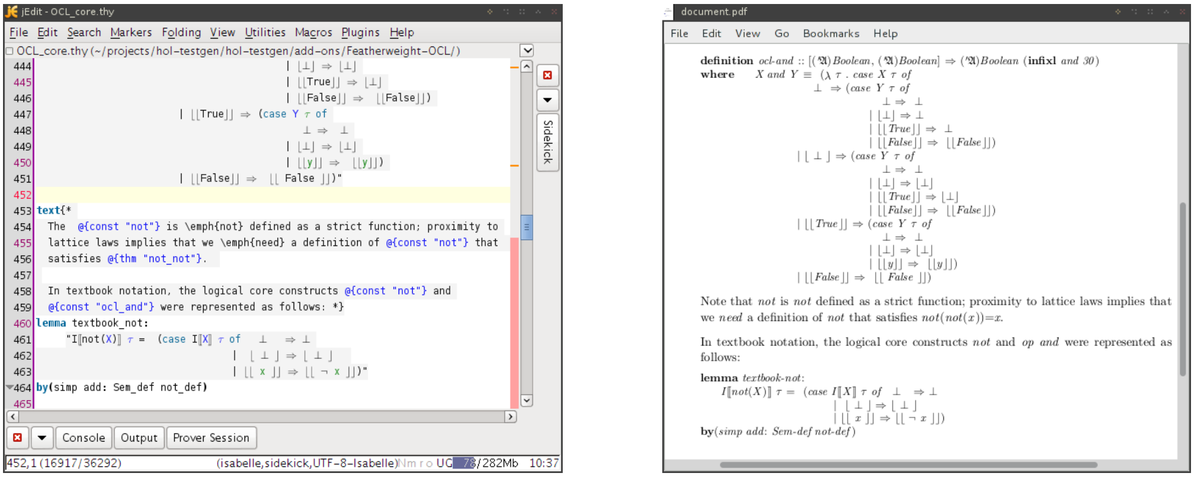
Finally, there is a compiler for primitive and wellfounded recursive function definitions. For example, we may define the sort operation of our running test example by:

$$\begin{array}{lll} \textbf{fun} & \mathrm{ins} & :: [\alpha :: \mathrm{linorder}, \alpha\,\mathrm{list}] \Rightarrow \alpha\,\mathrm{list} \\ \textbf{where} & \mathrm{ins}\ x\ [\,] & = [x] \\ & \mathrm{ins}\ x\ (y\#ys) & = \mathrm{if}\, x < y\, \mathrm{then}\, x\#y\#ys\, \mathrm{else}\, y\#(\mathrm{ins}\ x\,ys) \end{array} \tag{A.11}$$

$$\begin{array}{lll} \textbf{fun} & \mathrm{sort} & :: (\alpha :: \mathrm{linorder})\ \mathrm{list} \Rightarrow \alpha\,\mathrm{list} \\ \textbf{where} & \mathrm{sort}\ [\,] & = [\,] \\ & \mathrm{sort}(x\#xs) & = \mathrm{ins}\ x\ (\mathrm{sort}\ xs) \end{array} \tag{A.12}$$

The internal (non-recursive) constant definition for these operations is quite involved; however, the logical compiler will finally derive all the equations in the statements above from this definition and make them available for automated simplification.

Thus, Isabelle/HOL also provides a large collection of theories like sets, lists, multisets, orderings, and various arithmetic theories which only contain rules derived from conservative definitions. In particular, Isabelle manages a set of *executable types and operators*, i. e., types and operators for which a compilation to SML, OCaml or Haskell is possible. Setups for arithmetic types such as int have been done; moreover any datatype and any recursive function were included in this executable set (providing that they only consist of executable operators). Similarly, Isabelle manages a large set of (higher-order) rewrite rules into which recursive function definitions were included. Provided that this rule set represents a terminating and confluent rewrite system, the Isabelle simplifier provides also a highly potent decision procedure for many fragments of theories underlying the constraints to be processed when constructing test theorems.


jEdit - OCL_core.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
OCL_core.thy (~/projects/hol-testgen/hol-testgen/add-ons/Featherweight-OCL/)
Sidekick
Console
Output
Prover Session
452,1 (16917/36292)
(isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 78/282Mb 10:37

```
                                    | ⌊⊥⌋ ⇒ ⌊⊥⌋
                                    | ⌊⌊True⌋⌋ ⇒ ⌊⊥⌋
                                    | ⌊⌊False⌋⌋ ⇒  ⌊⌊False⌋⌋)
                       | ⌊⌊True⌋⌋ ⇒ (case Y τ of
                                      ⊥ ⇒  ⊥
                                    | ⌊⊥⌋ ⇒ ⌊⊥⌋
                                    | ⌊⌊y⌋⌋ ⇒  ⌊⌊y⌋⌋)
                       | ⌊⌊False⌋⌋ ⇒  ⌊⌊ False ⌋⌋)"

text{*
  The  @{const "not"} is \emph{not} defined as a strict function; proximity to
  lattice laws implies that we \emph{need} a definition of @{const "not"} that
  satisfies @{thm "not_not"}.

  In textbook notation, the logical core constructs @{const "not"} and
  @{const "ocl_and"} were represented as follows: *}
lemma textbook_not:
     "I⟦not(X)⟧ τ =  (case I⟦X⟧ τ of  ⊥   ⇒ ⊥
                                  |  ⌊ ⊥ ⌋ ⇒ ⌊ ⊥ ⌋
                                  | ⌊⌊ x ⌋⌋ ⇒ ⌊⌊ ¬ x ⌋⌋)"
by(simp add: Sem_def not_def)
```

(a) The Isabelle jEdit environment.

```
definition ocl-and :: [('𝔄)Boolean, ('𝔄)Boolean] ⇒ ('𝔄)Boolean (infixl and 30)
where    X and Y ≡ (λ τ . case X τ of
              ⊥  ⇒ (case Y τ of
                        ⊥ ⇒  ⊥
                      | ⌊⊥⌋ ⇒ ⊥
                      | ⌊⌊True⌋⌋ ⇒  ⊥
                      | ⌊⌊False⌋⌋ ⇒  ⌊⌊False⌋⌋)
            | ⌊ ⊥ ⌋ ⇒ (case Y τ of
                        ⊥ ⇒  ⊥
                      | ⌊⊥⌋ ⇒ ⌊⊥⌋
                      | ⌊⌊True⌋⌋ ⇒ ⌊⊥⌋
                      | ⌊⌊False⌋⌋ ⇒  ⌊⌊False⌋⌋)
            | ⌊⌊True⌋⌋ ⇒ (case Y τ of
                        ⊥ ⇒  ⊥
                      | ⌊⊥⌋ ⇒ ⌊⊥⌋
                      | ⌊⌊y⌋⌋ ⇒  ⌊⌊y⌋⌋)
            | ⌊⌊False⌋⌋ ⇒  ⌊⌊ False ⌋⌋)
```

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.

In textbook notation, the logical core constructs *not* and *op and* were represented as follows:

```
lemma textbook-not:
   I⟦not(X)⟧ τ =  (case I⟦X⟧ τ of  ⊥   ⇒ ⊥
                             |  ⌊ ⊥ ⌋ ⇒ ⌊ ⊥ ⌋
                             | ⌊⌊ x ⌋⌋ ⇒ ⌊⌊ ¬ x ⌋⌋)
by(simp add: Sem-def not-def)
```

(b) The generated formal document.

Figure A.2.: Generating documents with guaranteed syntactical and semantical consistency.

A.2.3. How this Annex A was Generated from Isabelle/HOL Theories

FiXme: *Here ? Or in chap 2 ?*

Isabelle, as a framework for building formal tools [32], provides the means for generating *formal documents*. With formal documents (such as the one you are currently reading) we refer to documents that are machine-generated and ensure certain formal guarantees. In particular, all formal content (e. g., definitions, formulae, types) are checked for consistency during the document generation.

For writing documents, Isabelle supports the embedding of informal texts using a LaTeX-based markup language within the theory files. To ensure the consistency, Isabelle supports to use, within these informal texts, *antiquotations* that refer to the formal parts and that are checked while generating the actual document as PDF. For example, in an informal text, the antiquotation @{thm "not_not"} will instruct Isabelle to lock-up the (formally proven) theorem of name ocl_not_not and to replace the antiquotation with the actual theorem, i. e., `not (not x) = x`.

Figure A.2 illustrates this approach: Figure A.2a shows the jEdit-based development environment of Isabelle with an excerpt of one of the core theories of FeatherweightOCL . Figure A.2b shows the generated PDF document where all antiquotations are replaced. Moreover, the document generation tools allows for defining syntactic sugar as well as skipping technical details of the formalization.

Thus, applying the FeatherweightOCL approach to writing an updated Annex A that provides a formal semantics of the most fundamental concepts of OCL would ensure

1. that all formal context is syntactically correct and well-typed, and
2. all formal definitions and the derived logical rules are semantically consistent.

Overall, this would contribute to one of the main goals of the OCL 2.5 RFP, as discussed at the OCL meeting in Aachen [13].

A.3. The Essence of UML-OCL Semantics

A.3.1. The Theory Organization

FiXme: *Generate this chapter from Isabelle theories ? Just for principle ?*

The semantic theory is organized in a quite conventional manner in three layers. The first layer, called the *denotational semantics* comprises a set of definitions of the operators of the language. Presented as *definitional axioms* inside Isabelle/HOL, this part assures the logically consistency of the overall construction. The denotational definitions of types, constants and operations, and OCL contracts represent the "gold standard" of the semantics. The second layer, called *logical layer*, is derived from the former and centered around the notion of validity of an OCL formula P for a state-transition from pre-state σ to post-state σ', validity statements were written $(\sigma, \sigma') \models P$. Its major purpose is to logically establish facts (lemmas and theorems) about the denotational definitions. The third layer, called *algebraic layer*, also derived from the former layers, tries to establish algebraic laws of the form $P = P'$; such laws are amenable to equational reasoning and also help for automated reasoning and code-generation. For an implementor of an OCL compiler, these consequences are of most interest.

For space reasons, we will restrict ourselves in this paper to a few operators and make a traversal through all three layers to give a high-level description of our formalization. Especially, the details of the semantic construction for sets and the handling of objects and object universes were excluded from a presentation here.

Denotational Semantics of Types

The syntactic material for type expressions, called $\text{TYPES}(C)$, is inductively defined as follows:

- $C \subseteq \text{TYPES}(C)$
- Boolean, Integer, Real, Void, ... are elements of $\text{TYPES}(C)$
- $\text{Sequence}(X)$, $\text{Set}(X)$, et $\text{Pair}(X,Y)$ (as example for a Tuple-type) are in $\text{TYPES}(C)$ (if $X, Y \in \text{TYPES}(C)$).

FiXme: *should we use expliucit definitions ?*

Types were directly represented in FeatherweightOCL by types in HOL; consequently, any FeatherweightOCL type must provide elements for a bottom element (also denoted $\perp$) and a null element; this is enforced in Isabelle by a type-class null that contains two distinguishable elements bot and null (see Section B.1.1 for the details of the construction).

Moreover, the representation mapping from OCL types to FeatherweightOCL is one-to-one (i. e. injective), and the corresponding FeatherweightOCL types were constructed to represent *exactly* the elements ("no junk, no confucion elements") of their OCL counterparts. The corresponding FeatherweightOCL types were constructed in two stages: First, a *base type* is constructed whose carrier set contains exactly the elements of the OCL type. Secondly, this base type is lifted to a *valuation* type that we use for type-checking FeatherweightOCL constants, operations, and expressions. The valuation type takes into account that some UML-OCL functions of its OCL type (namely: accessors in path-expressions) depend on a pre- and a post-state.

For most base types like $\text{Boolean}_{\text{base}}$ or $\text{Integer}_{\text{base}}$, it suffices to double-lift a HOL library type:

$$\text{type}_\text{s}\text{ynonym} \qquad \text{Boolean}_{\text{base}} := \mathit{bool}_{\perp\perp} \tag{A.13}$$

As a consequence of this definition of the type, we have the elements $\perp, \llcorner\perp\lrcorner, \llcorner\llcorner\text{true}\lrcorner\lrcorner, \llcorner\llcorner\text{false}\lrcorner\lrcorner$ in the carrier-set of $\text{Boolean}_{\text{base}}$. We can therefore use the element $\perp$ to define the generic type class element $\perp$ and $\llcorner\perp\lrcorner$ for the generic type class null. For collection types and object types this definition is more evolved (see Section B.1.1).

FiXme: *why does backslash null not work here ?*

For object base types, we assume a typed universe $\mathfrak{A}$ of objects to be discussed later, for the moment we will refer it by its polymorphic variable.

With respect the valuation types for OCL expression in general and Boolean expressions in particular, they depend on the pair (σ, σ') of pre-and post-state. Thus, we define valuation types by the synonym:

$$\text{type}_\text{s}\text{ynonym} \qquad V_{\mathfrak{A}}(\alpha) := \mathit{state}(\mathfrak{A}) \times \mathit{state}(\mathfrak{A}) \to \alpha :: \text{null} \,. \tag{A.14}$$

The valuation type for boolean,integer, etc. OCL terms is therefore defined as:

$$\begin{gathered} \text{type}_\text{s}\text{ynonym} \qquad \text{Boolean}_{\mathfrak{A}} := V_{\mathfrak{A}}(\text{Boolean}_\text{base}) \\ \text{type}_\text{s}\text{ynonym} \qquad \text{Integer}_{\mathfrak{A}} := V_{\mathfrak{A}}(\text{Integer}_\text{base}) \\ \dots \end{gathered}$$

the other cases are analogous. In the subsequent subsections, we will drop the index $\mathfrak{A}$ since it is constant in all formulas and expressions except for operations related to the object universe construction in **??**

The rules of the logical layer (there are no algebraic rules related to the semantics of types), and more details can be found in Section B.1.1.

A.3.2. Denotational Semantics of Constants and Operations

We use the notation $I[\![E]\!]\tau$ for the semantic interpretation function as commonly used in mathematical textbooks and the variable τ standing for pairs of pre- and post state (σ, σ'). OCL provides for all OCL types the constants `invalid` for the exceptional computation result and `null` for the non-existing value. Thus we define:

$$I[\![\texttt{invalid} :: V(\alpha)]\!]\tau \equiv \text{bot} \qquad I[\![\texttt{null} :: V(\alpha)]\!]\tau \equiv \text{null}$$

For the concrete `Boolean`-type, we define similarly the boolean constants `true` and `false` as well as the fundamental tests for definedness and validity (generically defined for all types):

$$\begin{gathered} I[\![\texttt{true} :: \texttt{Boolean}]\!]\tau = \llcorner\llcorner\text{true}\lrcorner\lrcorner \qquad I[\![\texttt{false}]\!]\tau = \llcorner\llcorner\text{false}\lrcorner\lrcorner \\ I[\![X\texttt{.oclIsUndefined()}]\!]\tau = (\text{if}\, I[\![X]\!]\tau \in \{\text{bot}, \text{null}\}\, \text{then}\, I[\![\texttt{true}]\!]\tau\, \text{else}\, I[\![\texttt{false}]\!]\tau) \\ I[\![X\texttt{.oclIsInvalid()}]\!]\tau = (\text{if}\, I[\![X]\!]\tau = \text{bot}\, \text{then}\, I[\![\texttt{true}]\!]\tau\, \text{else}\, I[\![\texttt{false}]\!]\tau) \end{gathered}$$

For reasons of conciseness, we will write $\delta\, X$ for `not`$(X$`.oclIsUndefined())` and $\upsilon\, X$ for `not`$(X$`.oclIsInvalic` throughout this document.

Due to the used style of semantic representation (a shallow embedding) I is in fact superfluous and defined semantically as the identity $\lambda\, x.x$; instead of:

$$I[\![\texttt{true} :: \texttt{Boolean}]\!]\tau = \llcorner\llcorner\text{true}\lrcorner\lrcorner$$

we can therefore write:

$$\texttt{true} :: \texttt{Boolean} = \lambda\, \tau. \llcorner\llcorner\text{true}\lrcorner\lrcorner$$

In Isabelle theories, this particular presentation of definitions paves the way for an automatic check that the underlying equation has the form of an *axiomatic definition* and is therefore logically safe.

On this basis, one can define the core logical operators not and and as follows:

$$
\begin{array}{ll}
I[\![\texttt{not}\ X]\!]\tau & = (\text{case}\, I[\![X]\!]\tau\, \text{of} \\
 & \quad \begin{array}{ll}
 \bot & \Rightarrow \bot \\
 |\lfloor\bot\rfloor & \Rightarrow \lfloor\bot\rfloor \\
 |\lfloor\lfloor x\rfloor\rfloor & \Rightarrow \lfloor\lfloor\neg x\rfloor\rfloor)
 \end{array}
\end{array}
$$

FiXme: *we must uniformize the list - vs. lfloor notation. Either the one or the other.*

$$
\begin{array}{ll}
I[\![X\ \texttt{and}\ Y]\!]\tau & = (\text{case}\, I[\![X]\!]\tau\, \text{of} \\
 & \quad \begin{array}{ll}
 \bot & \Rightarrow (\text{case}\, I[\![Y]\!]\tau\, \text{of} \\
 & \quad \begin{array}{ll}
 \bot & \Rightarrow \bot \\
 |\lfloor\bot\rfloor & \Rightarrow \bot \\
 |\lfloor\lfloor\text{true}\rfloor\rfloor & \Rightarrow \bot \\
 |\lfloor\lfloor\text{false}\rfloor\rfloor & \Rightarrow \lfloor\lfloor\text{false}\rfloor\rfloor)
 \end{array} \\
 |\lfloor\bot\rfloor & \Rightarrow (\text{case}\, I[\![Y]\!]\tau\, \text{of} \\
 & \quad \begin{array}{ll}
 \bot & \Rightarrow \bot \\
 |\lfloor\bot\rfloor & \Rightarrow \lfloor\bot\rfloor \\
 |\lfloor\lfloor\text{true}\rfloor\rfloor & \Rightarrow \lfloor\bot\rfloor \\
 |\lfloor\lfloor\text{false}\rfloor\rfloor & \Rightarrow \lfloor\lfloor\text{false}\rfloor\rfloor)
 \end{array} \\
 |\lfloor\lfloor\text{true}\rfloor\rfloor & \Rightarrow (\text{case}\, I[\![Y]\!]\tau\, \text{of} \\
 & \quad \begin{array}{ll}
 \bot & \Rightarrow \bot \\
 |\lfloor\bot\rfloor & \Rightarrow \lfloor\bot\rfloor \\
 |\lfloor\lfloor y\rfloor\rfloor & \Rightarrow \lfloor\lfloor y\rfloor\rfloor)
 \end{array} \\
 |\lfloor\lfloor\text{false}\rfloor\rfloor & \Rightarrow \lfloor\lfloor\text{false}\rfloor\rfloor)
 \end{array}
\end{array}
$$

These non-strict operations were used to define the other logical connectives in the usual classical way: $X\ \texttt{or}\ Y \equiv (\texttt{not}\ X)\ \texttt{and}\ (\texttt{not}\ Y)$ or $X\ \texttt{implies}\ Y \equiv (\texttt{not}\ X)\ \texttt{or}\ Y$.

The default semantics for an OCL library operator is strict semantics; this means that the result of an operation f is invalid if one of its arguments is +invalid+ or +null+. The definition of the addition for integers as default variant reads as follows:

$$
\begin{array}{ll}
I[\![x+y]\!]\tau = & \text{if}\, I[\![\delta\ x]\!]\tau = I[\![\texttt{true}]\!]\tau \wedge I[\![\delta\ y]\!]\tau = I[\![\texttt{true}]\!]\tau \\
 & \text{then}\, \lfloor\lfloor\lceil\lceil I[\![x]\!]\tau\rceil\rceil + \lceil\lceil I[\![y]\!]\tau\rceil\rceil\rfloor\rfloor \\
 & \text{else}\, \bot
\end{array}
$$

where the operator "+" on the left-hand side of the equation denotes the OCL addition of type $\texttt{Integer} \Rightarrow \texttt{Integer} \Rightarrow \texttt{Integer}$ while the "+" on the right-hand side of the equation of type $[\text{int}, \text{int}] \Rightarrow \text{int}$ denotes the integer-addition from the HOL library.

There are cases where strictness is handled differently: For example, since Set's may contain the null-element, it is necessary to allow null as argument for _->including():

$$
\begin{array}{ll}
I[\![S\ \texttt{->including}(y)]\!]\tau = & \text{if}\, I[\![\delta\ S]\!]\tau = I[\![\texttt{true}]\!]\tau \wedge I[\![\upsilon\ y]\!]\tau = I[\![\texttt{true}]\!]\tau \\
 & \text{then}\, \texttt{Abs\_Set}_{\texttt{base}}{}_{\llcorner\lrcorner}{}^{\ulcorner\urcorner}\texttt{Rep\_Set}_{\texttt{base}} I[\![S]\!]\tau^{\ulcorner\urcorner} \cup \{I[\![y]\!]\tau\}_{\llcorner\lrcorner} \\
 & \text{else}\, \bot
\end{array}
$$

Here, the operator $\_\cup\_$ stems from the HOL set theory, together with the set inclusion $\{\_\}$. The operator `Abs_Set`$_{\texttt{base}}$ is the constructor for the FeatherweightOCL Set type, whereas `Rep_Set`$_{\texttt{base}}$ is its destructor (see Section B.1.1 for details). There is even one more variant of a strict basic OCL operation: the referential equality _=_. Since the comparison with must be possible and since the referential equality should be symmetric, should be allowed for *both* arguments and the expression:

$$\texttt{null = null} \tag{A.15}$$

should be valid and true. The details were discussed in the next session.

Logical Layer

The topmost goal of the logic for OCL is to define the *validity statement*:

$$(\sigma,\sigma') \vDash P,$$

where σ is the pre-state and σ' the post-state of the underlying system and P is a formula, i. e. and OCL expression of type `Boolean`. Informally, a formula P is valid if and only if its evaluation in (σ,σ') (i. e., τ for short) yields true. Formally this means:

$$\tau \vDash P \equiv \big(I[\![P]\!]\tau = {\llcorner\llcorner}\mathrm{true}{\lrcorner\lrcorner}\big)\,.$$

On this basis, classical, two-valued inference rules can be established for reasoning over the logical connectives, the different notions of equality, definedness and validity. Generally speaking, rules over logical validity can relate bits and pieces in various OCL terms and allow—via strong logical equality discussed below—the replacement of semantically equivalent sub-expressions. The core inference rules are:

$$\tau \models \texttt{true} \qquad \neg(\tau \models \texttt{false}) \qquad \neg(\tau \models \texttt{invalid}) \qquad \neg(\tau \models \texttt{null})$$
$$\tau \models \texttt{not}\ P \Longrightarrow \neg(\tau \models P)$$
$$\tau \models P\ \texttt{and}\ Q \Longrightarrow \tau \models P \qquad \tau \models P\ \texttt{and}\ Q \Longrightarrow \tau \models Q$$
$$\tau \models P \Longrightarrow \tau \models P\ \texttt{or}\ Q \qquad \tau \models Q\tau \Longrightarrow \models P\ \texttt{or}\ Q$$
$$\tau \models P \Longrightarrow (\texttt{if}\ P\ \texttt{then}\ B_1\ \texttt{else}\ B_2\ \texttt{endif})\tau = B_1\,\tau$$
$$\tau \models \texttt{not}\ P \Longrightarrow (\texttt{if}\ P\ \texttt{then}\ B_1\ \texttt{else}\ B_2\ \texttt{endif})\tau = B_2\,\tau$$
$$\tau \models P \Longrightarrow \tau \models \delta\, P \qquad \tau \models \delta\, X \Longrightarrow \tau \models \upsilon\, X$$

By the latter two properties it can be inferred that any valid property P (so for example: a valid invariant) is defined, which allows to infer for terms composed by strict operations that their arguments and finally the variables occurring in it are valid or defined.

The mandatory part of the OCL standard refers to an equality (written `x = y` or `x <> y` for its negation), which is intended to be a strict operation (thus: `invalid = y` evaluates to `invalid`) and which uses the references of objects in a state when comparing objects, similarly to C++ or Java. In order to avoid confusions, we will use the following notations for equality:

1. The symbol $\_ = \_$ remains to be reserved to the HOL equality, i. e. the equality of our semantic meta-language,

2. The symbol _ $\triangleq$ _ will be used for the *strong logical equality*, which follows the general logical principle that "equals can be replaced by equals," [7] and is at the heart of the OCL logic,

3. The symbol _ $\doteq$ _ is used for the strict referential equality, i. e. the equality the mandatory part of the OCL standard refers to by the _ = _- symbol.

The strong logical equality is a polymorphic concept which is defined polymorphically for all OCL types by:

$$I[\![X \triangleq Y]\!]\tau \equiv \lfloor\lfloor I[\![X]\!]\tau = I[\![Y]\!]\tau \rfloor\rfloor$$

It enjoys nearly the laws of a congruence:

$$\tau \models (x \triangleq x)$$
$$\tau \models (x \triangleq y) \Longrightarrow \tau \models (y \triangleq x)$$
$$\tau \models (x \triangleq y) \Longrightarrow \tau \models (y \triangleq z) \Longrightarrow \tau \models (x \triangleq z)$$
$$\mathrm{cp}\, P \Longrightarrow \tau \models (x \triangleq y) \Longrightarrow \tau \models (P\, x) \Longrightarrow \tau \models (P\, y)$$

where the predicate cp stands for *context-passing*, a property that is true for all pure OCL expressions (but not arbitrary mixtures of OCL and HOL) in FeatherweightOCL . The necessary side-calculus for establishing cp can be fully automated; the reader interested in the details is referred to Section B.2.1.

The strong logical equality of FeatherweightOCL give rise to a number of further rules and derived properties, that clarify the role of strong logical equality and the boolean constants in OCL specifications:

$$\tau \models \delta\, x \vee \tau \models x \triangleq \texttt{invalid} \vee \tau \models x \triangleq \texttt{null},$$
$$(\tau \models A \triangleq \texttt{invalid}) = (\tau \models \texttt{not}(\upsilon A))$$
$$(\tau \models A \triangleq \texttt{true}) = (\tau \models A) \qquad (\tau \models A \triangleq \texttt{false}) = (\tau \models \texttt{not} A)$$
$$(\tau \models \texttt{not}(\delta x)) = (\neg \tau \models \delta x) \qquad (\tau \models \texttt{not}(\upsilon x)) = (\neg \tau \models \upsilon x)$$

The logical layer of the FeatherweightOCL rules gives also a means to convert an OCL formula living in its four-valued world into a representation that is classically two-valued and can be processed by standard SMT solvers such as CVC3 [2] or Z3 [18]. δ-closure rules for all logical connectives have the following format, e. g.:

$$\tau \models \delta\, x \Longrightarrow (\tau \models \texttt{not}\, x) = (\neg(\tau \models x))$$
$$\tau \models \delta\, x \Longrightarrow \tau \models \delta\, y \Longrightarrow (\tau \models x \,\texttt{and}\, y) = (\tau \models x \wedge \tau \models y)$$
$$\begin{aligned} &\tau \models \delta\, x \Longrightarrow \tau \models \delta\, y \\ &\quad \Longrightarrow (\tau \models (x \,\texttt{implies}\, y)) = ((\tau \models x) \longrightarrow (\tau \models y)) \end{aligned}$$

Together with the already mentioned general case-distinction

$$\tau \models \delta\, x \vee \tau \models x \triangleq \texttt{invalid} \vee \tau \models x \triangleq \texttt{null}$$

which is possible for any OCL type, a case distinction on the variables in a formula can be performed; due to strictness rules, formulae containing somewhere a variable x that is known to be `invalid` or `null` reduce

[7] Strong logical equality is also referred as "Leibniz"-equality.

usually quickly to contradictions. For example, we can infer from an invariant $\tau \models x \doteq y - 3$ that we have $\tau \models x \doteq y - 3 \wedge \tau \models \delta\, x \wedge \tau \models \delta\, y$. We call the latter formula the δ-closure of the former. Now, we can convert a formula like $\tau \models x > 0 \;\mathtt{or}\; 3 * y > x * x$ into the equivalent formula $\tau \models x > 0 \vee \tau \models 3 * y > x * x$ and thus internalize the OCL-logic into a classical (and more tool-conform) logic. This works—for the price of a potential, but due to the usually “rich” δ-closures of invariants rare—exponential blow-up of the formula for all OCL formulas.

Algebraic Layer

Based on the logical layer, we build a system with simpler rules which are amenable to automated reasoning. We restrict ourselves to pure equations on OCL expressions.

Our denotational definitions on `not` and `and` can be re-formulated in the following ground equations:

$$
\begin{array}{cc}
\upsilon\,\mathtt{invalid} = \mathtt{false} & \upsilon\,\mathtt{null} = \mathtt{true} \\
\upsilon\,\mathtt{true} = \mathtt{true} & \upsilon\,\mathtt{false} = \mathtt{true} \\
\delta\,\mathtt{invalid} = \mathtt{false} & \delta\,\mathtt{null} = \mathtt{false} \\
\delta\,\mathtt{true} = \mathtt{true} & \delta\,\mathtt{false} = \mathtt{true} \\
\mathtt{not}\,\mathtt{invalid} = \mathtt{invalid} & \mathtt{not}\,\mathtt{null} = \mathtt{null} \\
\mathtt{not}\,\mathtt{true} = \mathtt{false} & \mathtt{not}\,\mathtt{false} = \mathtt{true} \\
(\mathtt{null}\;\mathtt{and}\;\mathtt{true}) = \mathtt{null} & (\mathtt{null}\;\mathtt{and}\;\mathtt{false}) = \mathtt{false} \\
(\mathtt{null}\;\mathtt{and}\;\mathtt{null}) = \mathtt{null} & (\mathtt{null}\;\mathtt{and}\;\mathtt{invalid}) = \mathtt{invalid} \\
(\mathtt{false}\;\mathtt{and}\;\mathtt{true}) = \mathtt{false} & (\mathtt{false}\;\mathtt{and}\;\mathtt{false}) = \mathtt{false} \\
(\mathtt{false}\;\mathtt{and}\;\mathtt{null}) = \mathtt{false} & (\mathtt{false}\;\mathtt{and}\;\mathtt{invalid}) = \mathtt{false} \\
(\mathtt{true}\;\mathtt{and}\;\mathtt{true}) = \mathtt{true} & (\mathtt{true}\;\mathtt{and}\;\mathtt{false}) = \mathtt{false} \\
(\mathtt{true}\;\mathtt{and}\;\mathtt{null}) = \mathtt{null} & (\mathtt{true}\;\mathtt{and}\;\mathtt{invalid}) = \mathtt{invalid}
\end{array}
$$

$$
\begin{gathered}
(\mathtt{invalid}\;\mathtt{and}\;\mathtt{true}) = \mathtt{invalid} \\
(\mathtt{invalid}\;\mathtt{and}\;\mathtt{false}) = \mathtt{false} \\
(\mathtt{invalid}\;\mathtt{and}\;\mathtt{null}) = \mathtt{invalid} \\
(\mathtt{invalid}\;\mathtt{and}\;\mathtt{invalid}) = \mathtt{invalid}
\end{gathered}
$$

On this core, the structure of a conventional lattice arises:

$$
\begin{array}{cc}
X\;\mathtt{and}\;X = X & X\;\mathtt{and}\;Y = Y\;\mathtt{and}\;X \\
\mathtt{false}\;\mathtt{and}\;X = \mathtt{false} & X\;\mathtt{and}\;\mathtt{false} = \mathtt{false} \\
\mathtt{true}\;\mathtt{and}\;X = X & X\;\mathtt{and}\;\mathtt{true} = X
\end{array}
$$

$$
X\;\mathtt{and}\;(Y\;\mathtt{and}\;Z) = X\;\mathtt{and}\;Y\;\mathtt{and}\;Z
$$

as well as the dual equalities for `_ or _` and the De Morgan rules. This wealth of algebraic properties makes the understanding of the logic easier as well as automated analysis possible: it allows for, for example, computing a DNF of invariant systems (by clever term-rewriting techniques) which are a prerequisite for δ-closures.

The above equations explain the behavior for the most-important non-strict operations. The clarification of the exceptional behaviors is of key-importance for a semantic definition of the standard and the major deviation point from HOL-OCL [5, 7], to FeatherweightOCL as presented here. Expressed in algebraic equations, "strictness-principles" boil down to:

$$\begin{aligned}
\texttt{invalid} + X &= \texttt{invalid} & X + \texttt{invalid} &= \texttt{invalid}\\
\texttt{invalid->including(}X\texttt{)} &= \texttt{invalid} & \texttt{null->including(}X\texttt{)} &= \texttt{invalid}\\
X \doteq \texttt{invalid} &= \texttt{invalid} & \texttt{invalid} \doteq X &= \texttt{invalid}
\end{aligned}$$

$$\begin{aligned}
\texttt{S->including(invalid)} &= \texttt{invalid}\\
X \doteq X &= (\texttt{if}\ \upsilon\, x\ \texttt{then trueelse invalid endif})
\end{aligned}$$

$$\begin{aligned}
\texttt{1 / 0} &= \texttt{invalid} & \texttt{1 / null} &= \texttt{null}\\
\texttt{invalid->isEmpty()} &= \texttt{invalid} & \texttt{null->isEmpty()} &= \texttt{null}
\end{aligned}$$

Algebraic rules are also the key for execution and compilation of FeatherweightOCL expressions. We derived, e. g.:

$$\begin{aligned}
\delta\ \texttt{Set\{\}} &= \texttt{true}\\
\delta\ (X\texttt{->including(}x\texttt{)}) &= \delta\, X\ \texttt{and}\ \upsilon\, x\\
\texttt{Set\{\}->includes(}x\texttt{)} &= (\texttt{if}\ \upsilon\, x\ \texttt{then false}\\
&\qquad\qquad \texttt{else invalid endif})\\
(X\texttt{->including(}x\texttt{)->includes(}y\texttt{)}) &=\\
&(\texttt{if}\ \delta\, X\\
&\ \ \texttt{then if}\ x \doteq y\\
&\qquad\quad \texttt{then}\,\texttt{true}\\
&\qquad\quad \texttt{else}\,X\texttt{->includes(}y\texttt{)}\\
&\qquad\quad \texttt{endif}\\
&\ \ \texttt{else invalid}\\
&\ \ \texttt{endif})
\end{aligned}$$

As Set{1,2} is only syntactic sugar for

```
Set{}->including(1)->including(2)
```

an expression like Set{1,2}->includes(null) becomes decidable in FeatherweightOCL by applying these algebaric laws (which can give rise to efficient compilations). The reader interested in the list of "test-statements" like:

value "$\tau \models$(Set{Set{2,null}} $\doteq$ Set{Set{null,2}})"

make consult Section B.2.8; these test-statements have been machine-checked and proven consistent with the denotational and logic semantics of FeatherweightOCL.

A.3.3. Object-oriented Datatype Theories

In the following, we will refine the concepts of a user-defined data-model implied by a *class-model* (*visualized* by a class-*diagram*) as well as the notion of state used in the previous section to much more detail. UML class models represent in a compact and visual manner quite complex, object-oriented data-types with a surprisingly rich theory. In this section, this theory is made explicit and corner cases were pointed out.

A UML class model underlying a given OCL invariant or operation contract produces several implicit operations which become accessible via appropriate OCL syntax. A class model is a four-tuple $(C, \_ < \_, Attrib, Assoc)$ where:

FiXme: *TODO*

1. C is a set of class names (written as $\{C_1,\dots,C_n\}$). To each class name a type of data in OCL is associated. Moreover, class names declare two projector functions to the set of all objects in a state: C_i`.allInstances()` and C_i`.allInstances@pre()`,

2. $\_ < \_$ is an inheritance relation on classes,

3. $Attrib(C_i)$ is a collection of attributes associated to classes C_i. It declares two wo families of accessors; for each attribute $a \in Attrib(C_i)$ in a class definition C_i (denoted $X.a :: C_i \to A$ and $X.a$`@pre` $:: C_i \to A$ for $A \in TYPES(C)$),

4. $Assoc(C_i,C_j)$ is a collection of associations. [8] An association $(n, rn_{from}, rn_{to}) \in Assoc(C_i,C_j)$ between to classes C_i and C_j is a triple consisting of a (unique) association name n, and the rolenames rn_{to} and rn_{from}. To each rolename belong two families of accessors denoted $X.a :: C_i \to A$ and $X.a$`@pre` $:: C_i \to A$ for $A \in TYPES(C)$),

5. for each pair $C_i < C_j$ $(C_i, C_j < C)$, there is a cast operation of type $C_j \to C_i$ that can change the static type of an object of type C_i: $obj :: C_i$`.oclAsType(`C_j`)`,

6. for each class $C_i \in C$, there are two dynamic type tests (X`.oclIsTypeOf(`C_i`)` and X`.oclIsKindOf(`C_i`)`),

7. and last but not least, for each class name $C_i \in C$ there is an instance of the overloaded referential equality (written $\_ \doteq \_$).

Assuming a strong static type discipline in the sense of Hindley-Milner types, FeatherweightOCL has no "syntactic subtyping." In contrast, subtyping can be expressed *semantically* in FeatherweightOCL; by adding suitable casts which do have a formal semantics, subtyping becomes an issue of the front-end that can make implicit type-coersions explicit by introducing explicit type-casts. Our perspective shifts the emphasis on the semantic properties of casting, and the necessary universe of object representations (induced by a class model) that allows to establish them.

As a pre-requisite of a denotational semantics for these operations induced by a class-model, we need an *object-universe* in which these operations can be defined in a denotational manner and from which the necessary properties can be derived. A concrete universe constructed from a class model will be used to instantiate the implicit type parameter $\mathfrak{A}$ of all OCL operations discussed so far.

[8] Given the fact that there is at present no consensus on the semantics of n-ary associations, FeatherweightOCL restricts itself to binary associations.

A Denotational Space for Class-Models: Object Universes

It is natural to construct system states by a set of partial functions f that map object identifiers oid to some representations of objects:

$$\text{typedef} \qquad \alpha \text{ state} := \{\sigma :: \text{oid} \rightharpoonup \alpha \mid \text{inv}_\sigma(\sigma)\} \tag{A.16}$$

where inv_σ is a to be discussed invariant on states.

The key point is that we need a common type α for the set of all possible *object representations*. Object representations model "a piece of typed memory," i. e., a kind of record comprising administration information and the information for all attributes of an object; here, the primitive types as well as collections over them are stored directly in the object representations, class types and collections over them are represented by oid's (respectively lifted collections over them).

In a shallow embedding which must represent UML types injectively by HOL types, there are two fundamentally different ways to construct such a set of object representations, which we call an *object universe* $\mathfrak{A}$:

1. an object universe can be constructed from a given class model, leading to *closed world semantics*, and

2. an object universe can be constructed for a given class model *and all its extensions by new classes added into the leaves of the class hierarchy*, leading to an *open world semantics*.

For the sake of simplicity, the present semantics chose the first option for FeatherweightOCL, while HOL-OCL [6] used an involved construction allowing the latter.

A naïve attempt to construct $\mathfrak{A}$ would look like this: the class type C_i induced by a class will be the type of such an object representation: $C_i := (\text{oid} \times A_{i_1} \times \cdots \times A_{i_k})$ where the types $A_{i_1}, \ldots, A_{i_k}$ are the attribute types (including inherited attributes) with class types substituted by oid. The function OidOf projects the first component, the oid, out of an object representation. Then the object universe will be constructed by the type definition:

$$\mathfrak{A} := C_1 + \cdots + C_n\,. \tag{A.17}$$

It is possible to define constructors, accessors, and the referential equality on this object universe. However, the treatment of type casts and type tests cannot be faithful with common object-oriented semantics, be it in UML or Java: casting up along the class hierarchy can only be implemented by loosing information, such that casting up and casting down will *not* give the required identity:

$$X\texttt{.oclIsTypeOf(}C_k\texttt{) implies } X\texttt{.oclAsType(}C_i\texttt{).oclAsType(}C_k\texttt{)} \doteq X \tag{A.18}$$

$$\text{whenever } C_k < C_i \text{ and } X \text{ is valid.} \tag{A.19}$$

To overcome this limitation, we introduce an auxiliary type $C_{i\text{ext}}$ for *class type extension*; together, they were inductively defined for a given class diagram:

Let C_i be a class with a possibly empty set of subclasses $\{C_{j_1}, \ldots, C_{j_m}\}$.

- Then the *class type extension* $C_{i\text{ext}}$ associated to C_i is $A_{i_1} \times \cdots \times A_{i_n} \times (C_{j_1\text{ext}} + \cdots + C_{j_m\text{ext}})_\perp$ where A_{i_k} ranges over the local attribute types of C_i and $C_{j_l\text{ext}}$ ranges over all class type extensions of the subclass C_j of C_i.

- Then the *class type* for C_i is $oid \times A_{i_1} \times \cdots \times A_{i_n} \times (C_{j_1\text{ext}} + \cdots + C_{j_m\text{ext}})_\perp$ where A_{i_k} ranges over the inherited *and* local attribute types of C_i and $C_{j_l\text{ext}}$ ranges over all class type extensions of the subclass C_j of C_i.

Example instances of this scheme—outlining a compiler—can be found in Section B.4 and Section B.5.

This construction can *not* be done in HOL itself since it involves quantifications and iterations over the "set of class-types"; rather, it is a meta-level construction. Technically, this means that we need a compiler to be done in SML on the syntactic "meta-model"-level of a class model.

With respect to our semantic construction here, which above all means is intended to be type-safe, this has the following consequences:

- there is a generic theory of states, which must be formulated independently from a concrete object universe,

- there is a principle of translation (captured by the inductive scheme for class type extensions and class types above) that converts a given class model into an concrete object universe,

- there are fixed principles that allow to derive the semantic theory of any concrete object universe, called the *object-oriented datatype theory.*

We will work out concrete examples for the construction of the object-universes in Section B.4 and Section B.5 and the derivation of the respective datatype theories. While an automatization is clearly possible and desirable for concrete applications of FeatherweightOCL, we consider this out of the scope of this paper which has a focus on the semantic construction and its presentation.

Denotational Semantics of Accessors on Objects and Associations

Our choice to use a shallow embedding of OCL in HOL and, thus having an injective mapping from OCL types to HOL types, results in type-safety of FeatherweightOCL . Arguments and results of accessors are based on type-safe object representations and *not* oid's. This implies the following scheme for an accessor:

- The *evaluation and extraction* phase. If the argument evaluation results in an object representation, the oid is extracted, if not, exceptional cases like `invalid` are reported.

- The *dereferentiation* phase. The oid is interpreted in the pre- or post-state, the resulting object is casted to the expected format. The exceptional case of nonexistence in this state must be treated.

- The *selection* phase. The corresponding attribute is extracted from the object representation.

- The *re-construction* phase. The resulting value has to be embedded in the adequate HOL type. If an attribute has the type of an object (not value), it is represented by an optional (set of) oid, which must be converted via dereferentiation in one of the states to produce an object representation again. The exceptional case of nonexistence in this state must be treated.

The first phase directly translates into the following formalization:

definition

$$\begin{aligned}\text{eval\_extract} X\, f = (\lambda\, \tau.\ \text{case} X\, \tau\, \text{of}\ & \bot && \Rightarrow \texttt{invalid}\,\tau && \text{exception}\\ |\ & \llcorner\bot\lrcorner && \Rightarrow \texttt{invalid}\,\tau && \text{deref. null}\\ |\ & \llcorner\llcorner obj\lrcorner\lrcorner && \Rightarrow f\,(\text{oid\_of}\ obj)\,\tau)\end{aligned} \tag{A.20}$$

For each class C, we introduce the dereferentiation phase of this form:

$$\begin{aligned}&\text{definition deref\_oid}_C\, \mathit{fst\_snd}\, f\, \mathit{oid} = (\lambda\, \tau.\ \text{case}\,(\text{heap}\,(\mathit{fst\_snd}\,\tau))\,\mathit{oid}\ \text{of}\\ &\qquad\qquad\qquad\qquad\qquad\qquad \llcorner\text{in}_C\, obj\lrcorner \quad \Rightarrow f\, obj\,\tau\\ &\qquad\qquad\qquad\qquad\qquad\quad |\ \_ \qquad\qquad \Rightarrow \texttt{invalid}\,\tau)\end{aligned} \tag{A.21}$$

The operation yields undefined if the oid is uninterpretable in the state or referencing an object representation not conforming to the expected type.

We turn to the selection phase: for each class C in the class model with at least one attribute, and each attribute a in this class, we introduce the selection phase of this form:

$$\begin{aligned}\text{definition select}_a\, f = (\lambda\ & \text{mk}_C\ \mathit{oid} && \cdots\bot\cdots && C\mathcal{X}_{\text{ext}} && \Rightarrow \texttt{null}\\ |\ & \text{mk}_C\ \mathit{oid} && \cdots\llcorner a\lrcorner\cdots && C\mathcal{X}_{\text{ext}} && \Rightarrow f\,(\lambda\, x\,\_.\ \llcorner\llcorner x\lrcorner\lrcorner)\, a)\end{aligned} \tag{A.22}$$

This works for definitions of basic values as well as for object references in which the a is of type oid. To increase readability, we introduce the functions:

$$\begin{aligned}&\text{definition} && \text{in\_pre\_state} && = \text{fst} && \text{first component}\\ &\text{definition} && \text{in\_post\_state} && = \text{snd} && \text{second component}\\ &\text{definition} && \text{reconst\_basetype} && = \text{id} && \text{identity function}\end{aligned} \tag{A.23}$$

Let $\_\,.\texttt{getBase}$ be an accessor of class C yielding a value of base-type A_{base}. Then its definition is of the form:

$$\begin{aligned}&\text{definition} && \_\,.\texttt{getBase} && :: C \Rightarrow A_{base}\\ &\text{where} && X\,.\texttt{getBase} && = \text{eval\_extract}\ X\,(\text{deref\_oid}_C\ \text{in\_post\_state}\\ &&&&& \quad(\text{select}_{\text{getBase}}\ \text{reconst\_basetype}))\end{aligned} \tag{A.24}$$

Let $\_\,.\texttt{getObject}$ be an accessor of class C yielding a value of object-type A_{object}. Then its definition is of the form:

$$\begin{aligned}&\text{definition} && \_\,.\texttt{getObject} && :: C \Rightarrow A_{object}\\ &\text{where} && X\,.\texttt{getObject} && = \text{eval\_extract}\ X\,(\text{deref\_oid}_C\ \text{in\_post\_state}\\ &&&&& \quad(\text{select}_{\text{getObject}}\,(\text{deref\_oid}_C\ \text{in\_post\_state})))\end{aligned} \tag{A.25}$$

The variant for an accessor yielding a collection is omitted here; its construction follows by the application of the principles of the former two. The respective variants $\_.a$ `@pre` were produced when in_post_state is replaced by in_pre_state.

Examples for the construction of accessors via associations can be found in Section B.4.8, the construction of accessors via attributes in Section B.5.8. The construction of casts and type tests `->oclIsTypeOf()` and `->oclIsKindOf()` is similarly.

In the following, we discuss the role of multiplicities on the types of the accessors. Depending on the specified multiplicity, the evaluation of an attribute can yield just a value (multiplicity `0..1` or `1`) or a collection type like Set or Sequence of values (otherwise). A multiplicity defines a lower bound as well as a possibly infinite upper bound on the cardinality of the attribute's values.

Single-Valued Attributes If the upper bound specified by the attribute's multiplicity is one, then an evaluation of the attribute yields a single value. Thus, the evaluation result is *not* a collection. If the lower bound specified by the multiplicity is zero, the evaluation is not required to yield a non-null value. In this case an evaluation of the attribute can return `null` to indicate an absence of value.

To facilitate accessing attributes with multiplicity `0..1`, the OCL standard states that single values can be used as sets by calling collection operations on them. This implicit conversion of a value to a `Set` is not defined by the standard. We argue that the resulting set cannot be constructed the same way as when evaluating a `Set` literal. Otherwise, `null` would be mapped to the singleton set containing `null`, but the standard demands that the resulting set is empty in this case. The conversion should instead be defined as follows:

```
context OclAny::asSet():T
  post: if self = null then result = Set{}
        else result = Set{self} endif
```

Collection-Valued Attributes If the upper bound specified by the attribute's multiplicity is larger than one, then an evaluation of the attribute yields a collection of values. This raises the question whether `null` can belong to this collection. The OCL standard states that `null` can be owned by collections. However, if an attribute can evaluate to a collection containing `null`, it is not clear how multiplicity constraints should be interpreted for this attribute. The question arises whether the `null` element should be counted or not when determining the cardinality of the collection. Recall that `null` denotes the absence of value in the case of a cardinality upper bound of one, so we would assume that `null` is not counted. On the other hand, the operation `size` defined for collections in OCL does count `null`.

We propose to resolve this dilemma by regarding multiplicities as optional. This point of view complies with the UML standard, that does not require lower and upper bounds to be defined for multiplicities.[9] In case a multiplicity is specified for an attribute, i. e., a lower and an upper bound are provided, we require any collection the attribute evaluates to not contain `null`. This allows for a straightforward interpretation of the multiplicity constraint. If bounds are not provided for an attribute, we consider the attribute values to not be restricted in any way. Because in particular the cardinality of the attribute's values is not bounded, the result of an evaluation of the attribute is of collection type. As the range of values that the attribute can assume is not restricted, the attribute can evaluate to a collection containing `null`. The attribute can also evaluate to `invalid`. Allowing multiplicities to be optional in this way gives the modeler the freedom to define attributes that can assume the full ranges of values provided by their types. However, we do not permit the omission of multiplicities for

[9] We are however aware that a well-formedness rule of the UML standard does define a default bound of one in case a lower or upper bound is not specified.

association ends, since the values of association ends are not only restricted by multiplicities, but also by other constraints enforcing the semantics of associations. Hence, the values of association ends cannot be completely unrestricted.

The Precise Meaning of Multiplicity Constraints We are now ready to define the meaning of multiplicity constraints by giving equivalent invariants written in OCL . Let a be an attribute of a class C with a multiplicity specifying a lower bound m and an upper bound n. Then we can define the multiplicity constraint on the values of attribute a to be equivalent to the following invariants written in OCL:

```
context C inv lowerBound: a->size() >= m
          inv upperBound: a->size() <= n
          inv notNull: not a->includes(null)
```

If the upper bound n is infinite, the second invariant is omitted. For the definition of these invariants we are making use of the conversion of single values to sets described in Section A.3.3. If $n \leq 1$, the attribute a evaluates to a single value, which is then converted to a Set on which the size operation is called.

If a value of the attribute a includes a reference to a non-existent object, the attribute call evaluates to invalid. As a result, the entire expressions evaluate to invalid, and the invariants are not satisfied. Thus, references to non-existent objects are ruled out by these invariants. We believe that this result is appropriate, since we argue that the presence of such references in a system state is usually not intended and likely to be the result of an error. If the modeler wishes to allow references to non-existent objects, she can make use of the possibility described above to omit the multiplicity.

Logic Properties of Class-Models

In this section, we assume to be $C_z, C_i, C_j \in C$ and $C_i < C_j$. Let C_z be a smallest element wit h respect to the class hierarchy _ < _. The operations induced from a class-model have the following properties:

```
 \<tau> \<Turnstile> X .oclAsType(C_i) \<triangleq> X
 \<tau> \<Turnstile> invalid .oclAsType(C_i) \<triangleq> invalid
 \<tau> \<Turnstile> null .oclAsType(C_i) \<triangleq> null
 \<tau> \<Turnstile> ((X::C_i) .oclAsType(C_j) .oclAsType(C_i) \<triangleq> X)
 \<tau> \<Turnstile> X .oclAsType(C_j) .oclAsType(C_i) \<triangleq> X
 \<tau> \<Turnstile> \<upsilon> (X :: C_i) \<Longrightarrow> \<tau> \<Turnstile> (X .oclIs
 \<tau> \<Turnstile> (X::OclAny) .oclAsType(OclAny) \<triangleq> X
 \<tau> \<Turnstile> \<upsilon> (X :: C_i) \<Longrightarrow> \<tau> \<Turnstile> (X .oclIs
 \<tau> \<Turnstile> \<delta> X \<Longrightarrow> \<tau> \<Turnstile> X .oclAsType(C_j) .o
 \<tau> \<Turnstile> \<upsilon> X \<Longrightarrow> \<tau> \<Turnstile> X .oclIsTypeOf(C_i
 \<tau> \<Turnstile> X .oclIsTypeOf(C_j) \<Longrightarrow> \<tau> \<Turnstile> \<delta> X
 \<tau> \<Turnstile> invalid .oclIsTypeOf(C_i) \<triangleq> invalid
 \<tau> \<Turnstile> null .oclIsTypeOf(C_i) \<triangleq> true
 \<tau> \<Turnstile> (Person .allInstances()->forAll(X|X .oclIsTypeOf(C_z)))
 \<tau> \<Turnstile> (Person .allInstances@pre()->forAll(X|X .oclIsTypeOf(C_z)))
 \<tau> \<Turnstile> (Person .allInstances()->forAll(X|X .oclIsKindOf(C_i)))
 \<tau> \<Turnstile> (Person .allInstances@pre()->forAll(X|X .oclIsKindOf(C_i)))
 \<tau> \<Turnstile> (X::C_i).oclIsTypeOf(C_j) \<Longrightarrow> \<tau> \<Turnstile> (X::C_
(\<tau> \<Turnstile> (X::C_j) \<doteq> X) = (\<tau> \<Turnstile> if \<upsilon> X then true
 \<tau> \<Turnstile> (X::C_j) \<doteq> Y \<Longrightarrow>  \<tau> \<Turnstile> Y \<doteq>
 \<tau> \<Turnstile> (X::C_j) \<doteq> Y \<Longrightarrow>  \<tau> \<Turnstile> Y \<doteq>
\<Longrightarrow>  \<tau> \<Turnstile> X \<doteq> Z
```

Algebraic Properties of the Class-Models

In this section, we assume to be $C_i, C_j \in C$ and $C_i < C_j$. The operations induced from a class-model have the following properties:

$$\begin{array}{lll}
\mathtt{invalid.oclIsTypeOf}\,(C_i) = \mathtt{invalid} & \mathtt{null.oclIsTypeOf}\,(C_i) = \mathtt{true} & \\
\mathtt{invalid.oclIsKindOf}\,(C_i) = \mathtt{invalid} & \mathtt{null.oclIsKindOf}\,(C_i) = \mathtt{true} & (X :: C_i) \doteq X = \mathtt{if}\, \upsilon\, X \mathtt{t} \\
(X :: C_i)\mathtt{.oclAsType}\,(C_i) = X & \mathtt{invalid.oclAsType}\,(C_i) = \mathtt{invalid} & \\
\mathtt{null.oclAsType}\,(C_i) = \mathtt{null} & ((X :: C_i)\mathtt{.oclAsType}\,(C_j)\ \mathtt{.oclAsType}\,(C_i) = X) &
\end{array} \tag{A.26}$$

With respect to attributes _. a or _. a @pre and role-ends _. r or _. r @pre we have

$$\begin{array}{rl}
\mathtt{invalid.a} = \mathtt{invalid} & \mathtt{null.a} = \mathtt{invalid} \\
\mathtt{invalid.a@pre} = \mathtt{invalid} & \mathtt{null.a@pre} = \mathtt{invalid} \\
\mathtt{invalid.r} = \mathtt{invalid} & \mathtt{null.r} = \mathtt{invalid} \\
\mathtt{invalid.r@pre} = \mathtt{invalid} & \mathtt{null.r@pre} = \mathtt{invalid}
\end{array}$$

Other Operations on States

Defining _`.allInstances()` is straight-forward; the only difference is the property T `.allInstances()->exclude`
which is a consequence of the fact that `null`'s are values and do not "live" in the state. OCL semantics admits states with "dangling references,"; it is the semantics of accessors or roles which maps these references to `invalid`, which makes it possible to rule out these situations in invariants.

OCL does not guarantee that an operation only modifies the path-expressions mentioned in the postcondition, i. e., it allows arbitrary relations from pre-states to post-states. This framing problem is well-known (one of the suggested solutions is [21]). We define

```
(S:Set(OclAny))->oclIsModifiedOnly():Boolean
```

where `S` is a set of object representations, encoding a set of oid's. The semantics of this operator is defined such that for any object whose oid is *not* represented in `S` and that is defined in pre and post state, the corresponding object representation will not change in the state transition. A simplified presentation is as follows:

$$I[\![X\mathtt{->oclIsModifiedOnly()}]\!](\sigma,\sigma') \equiv \begin{cases} \bot & \text{if } X' = \bot \vee \text{null} \in X' \\ \llcorner \forall i{\in}M.\ \sigma\ i = \sigma'\ i \lrcorner & \text{otherwise}\,. \end{cases}$$

where $X' = I[\![X]\!](\sigma,\sigma')$ and $M = (\mathrm{dom}\ \sigma \cap \mathrm{dom}\ \sigma') - \{\mathrm{OidOf}\, x |\ x \in \ulcorner X' \urcorner\}$. Thus, if we require in a postcondition `Set{}->oclIsModifiedOnly()` and exclude via _`.oclIsNew()` and _`.oclIsDeleted()`

the existence of new or deleted objects, the operation is a query in the sense of the OCL standard, i. e., the `isQuery` property is true. So, whenever we have $\tau \vDash X\texttt{->excluding}(s.a)\texttt{->oclIsModifiedOnly()}$ and $\tau \vDash X\texttt{->forAll}\,(x\text{not}|(x \doteq s.a))$, we can infer that $\tau \vDash s.a \triangleq s.a\,\texttt{@pre}$.

A.3.4. Data Invariants

Since the present OCL semantics uses one interpretation function [10], we express the effect of OCL terms occuring in preconditions and invariants by a syntactic transformation $\_{}_{\text{pre}}$ which replaces:

- all accessor functions $\_\,.\,\text{a}$ from the class model $a \in \mathit{Attrib}(C)$ by their counterparts $\_\,.\,\text{i}\,\texttt{@pre}$. For example, $(\mathit{self}\,.\,\text{salary} > 500)_{\text{pre}}$ is transformed to $(\mathit{self}\,.\,\text{salary}\,\texttt{@pre} > 500)$.
- all role accessor functions $\_\,.\,\text{rn}_{\text{from}}$ or $\_\,.\,\text{rn}_{\text{to}}$ within the class model (i. e. $(\mathit{id}, \mathit{rn}_{\mathit{from}}, \mathit{rn}_{\mathit{to}}) \in \mathit{Assoc}(C_i, C_j)$) were replaced by their counterparts $\_\,.\,\text{rn}\,\texttt{@pre}$. For example, $(\mathit{self}\,.\,\text{boss} = \mathit{null})_{\text{pre}}$ is transformed to $\mathit{self}\,.\,\text{boss}\,\texttt{@pre} = \mathit{null}$.
- The operation $\_\texttt{.allInstances()}$ is also substituted by its `@pre` counterpart.

Thus, we formulate the semantics of the invariant specification as follows:

$$
\begin{aligned}
&I[\![\texttt{context}\ c : C_i\ \texttt{inv}\ n : \phi(c)]\!]\tau \equiv \\
&\qquad \tau \vDash (C_i\,\texttt{.allInstances()->forall}(x|\phi(x))) \wedge \\
&\qquad \tau \vDash (C_i\,\texttt{.allInstances()->forall}(x|\phi(x)))_{\text{pre}}
\end{aligned}
\tag{A.27}
$$

Recall that expressions containing `@pre` constructs in invariants or preconditions are syntactically forbidden; thus, mixed forms cannot arise.

A.3.5. Operation Contracts

Since operations have strict semantics in OCL, we have to distinguish for a specification of an operation *op* with the arguments $a_1, \ldots, a_n$ the two cases where all arguments are valid and additionally, *self* is non-null (i. e. it must be defined), or not. In former case, a method call can be replaced by a *result* that satisfies the contract, in the latter case the result is `invalid`. This is reflected by the following definition of the contract semantics:

FiXme: *Should we add in our notion of Class-Model also the Operations ?*

$$
\begin{aligned}
&I[\![\texttt{context}\ C :: \mathit{op}(a_1, \ldots, a_n) : T \\
&\qquad \texttt{pre}\ \phi(\mathit{self}, a_1, \ldots, a_n) \\
&\qquad \texttt{post}\ \psi(\mathit{self}, a_1, \ldots, a_n, \mathit{result})]\!] \equiv \\
&\qquad\quad \lambda\, s, x_1, \ldots, x_n, \tau. \\
&\qquad\qquad \text{if}\ \ \tau \vDash \partial\, s \wedge \tau \vDash \upsilon\, x_1 \wedge \ldots \wedge \tau \vDash \upsilon\, x_n \\
&\qquad\qquad \text{then SOME}\ \mathit{result}.\quad \tau \vDash \phi(s, x_1, \ldots, x_n)_{\text{pre}} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge \tau \vDash \psi(s, x_1, \ldots, x_n, \mathit{result})) \\
&\qquad\qquad \text{else}\ \bot
\end{aligned}
\tag{A.28}
$$

[10] This has been handled differently in previous versions of the Annex A.

where SOME x. $P(x)$ is the Hilbert-Choice Operator that chooses an arbitrary element satisfying P; if such an element does not exist, it chooses an arbitrary one[11]. Thus, using the Hilbert-Choice Operator, a contract can be associated to a function definition:

$$f_{op} \equiv I[\![\texttt{context}\ C\ :: op(a_1, \dots, a_n) : T \dots]\!] \tag{A.29}$$

provided that neither ϕ nor ψ contain recursive method calls of op. In the case of a query operation (i. e. τ must have the form: (σ, σ), which means that query operations do not change the state; c.f. `oclIsModifiedOnly()` in Section A.3.3), this constraint can be relaxed: the above equation is then stated as *axiom*. Note however, that the consistency of the overall theory is for recursive query constracts left to the user (it can be shown, for example, by a proof of termination, i. e. by showing that all recursive calls were applied to argument vectors that are smaller wrt. to a well-founded ordering). Under this condition, an f_{op} resulting from recursive query operations can be used safely inside pre- and post-conditions of other contracts.

For the general case of a user-defined contract, the following rule can be established that reduces the proof of a property E over a method call f_{op} to a proof of $E(res)$ (where res must be one of the values that satisfy the post-condition ψ):

$$\frac{\begin{array}{c}\left[\tau \vDash \psi\ self\ a_1 \dots a_n\ res\right]_{res} \\ \vdots \\ \tau \vDash E(res)\end{array}}{\tau \vDash E(f_{op}\ self\ a_1 \dots a_n)} \tag{A.30}$$

under the conditions:

- E must be an OCL term and
- *self* must be defined, and the arguments valid in τ: $\vDash \partial\ self \wedge \tau \vDash \upsilon\ x_1 \wedge \dots \wedge \tau \vDash \upsilon\ x_n$
- the post-condition must be satisfiable ("the operation must be implementable"): $\exists res. \tau \vDash \psi\ self\ a_1 \dots a_n\ res$.

For the special case of a (recursive) query method, this rule can be specialized to the following executable "unfolding principle":

$$\frac{\tau \vDash \phi\ self\ a_1 \dots a_n}{(\tau \vDash E(f_{op}\ self\ a_1 \dots a_n)) = (\tau \vDash E(BODY\ self\ a_1 ... a_n))} \tag{A.31}$$

where

- E must be an OCL term.
- *self* must be defined, and the arguments valid in τ: $\tau \vDash \partial\ self \wedge \tau \vDash \upsilon\ x_1 \wedge \dots \wedge \tau \vDash \upsilon\ x_n$

[11] In HOL, the Hilbert-Choice operator is a first-class element of the logical language.

- the postcondition $\psi\ self\ a_1\ \ldots\ a_n\ result$ must be decomposable into:
  $\psi'\ self\ a_1\ \ldots a_n$ and $result \triangleq BODY\ self\ a_1\ \ldots\ a_n$.

We do not model *overriding* of operations as in Java or C++ explicitly in FeatherweightOCL. However, it is easy expressed in this core-language by adding `self.oclIsKindOf(C)` in the pre-condition ϕ (assuming that, as in the schema above, `C` is the context to which the contract is referring to). In order to avoid logical contradictions (inconsistencies) between different instances of an overriden operation, the user has to prove Liskov's principle for these situations: pre-conditions of the superclass must imply pre-conditions of the subclass, and post-conditions of a subclass must imply post-conditions of the superclass. **FiXme**: *correct?*

B. Formal Semantics of OCL

B.1. Formalization I: OCL Types and Core Definitions

theory *UML-Types*
imports *Transcendental*
keywords *Assert* :: *thy-decl*
  and *Assert-local* :: *thy-decl*
begin

B.1.1. Preliminaries

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 like a textbook:

no-notation *ceiling* $(\lceil \text{-} \rceil)$
no-notation *floor* $(\lfloor \text{-} \rfloor)$

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

The following function (corresponding to *the* in the Isabelle/HOL library) is defined as the inverse of the injection *Some*.

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

The definitions for the constants and operations based on functions will be geared towards a format that Isabelle can check to be a "conservative" (i. e., logically safe) axiomatic definition. By introducing an explicit interpretation function (which happens to be defined just as the identity since we are using a shallow embedding of OCL into HOL), all these definitions can be rewritten into the conventional semantic textbook format. To say it in other words: The interpretation function *Sem* as defined below is just a textual marker for presentation purposes, i.e. intended for readers used to conventional textbook notations on semantics. Since we use a "shallow embedding", i.e. since we represent the syntax of OCL directly by HOL constants, the interpretation function is semantically not only superfluous, but from an Isabelle perspective strictly in the way for certain consistency checks performed by the definitional packages.

definition *Sem* :: $'a \Rightarrow 'a$ $(I[\![\text{-}]\!])$
where $I[\![x]\!] \equiv x$

Common Infrastructure for all 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 not comparable with the raw-types (consisting of nested option type constructions), such that the data-invariant must 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 (defined by ⌊ ⊥ ⌋ on *'a option option*) to a *null* element, which may have an arbitrary semantic structure, and an undefinedness element ⊥ to an abstract undefinedness element *bot* (also written ⊥ whenever no confusion arises). As a consequence, it is necessary to redefine the notions of invalid, defined, valuation etc. on top of this interface.

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

```
class  bot =
  fixes  bot :: 'a
  assumes nonEmpty : ∃ x. x ≠ bot
```

```
class    null = bot +
  fixes  null :: 'a
  assumes null-is-valid : null ≠ bot
```

Accommodation of Basic Types to the Abstract Interface

In the following it is shown that the "option-option" 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`, `Real`, ...).

```
instantiation  option  :: (type)bot
begin
  definition bot-option-def: (bot::'a option) ≡ (None::'a option)
  instance proof show      ∃x::'a option. x ≠ 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) ≡ ⌊ bot ⌋
  instance proof  show      (null::'a::bot option) ≠ bot
              by( simp add : null-option-def bot-option-def)
```

```
        qed
end


instantiation fun  :: (type,bot) bot
begin
  definition bot-fun-def: bot ≡ (λ x. bot)

  instance proof  show ∃(x::'a ⇒ 'b). x ≠ bot
              apply(rule-tac x=λ -. (SOME y. y ≠ 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 ⇒ 'b::null) ≡ (λ x. null)

 instance proof
          show (null::'a ⇒ 'b::null) ≠ 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).

The Common Infrastructure of Object Types (Class Types) and States.

Recall that OCL is a textual extension of the UML; in particular, we use OCL as means to annotate UML class models. Thus, OCL inherits a notion of *data* in the UML: UML class models provide classes, inheritance, types of objects, and subtypes connecting them along the inheritance hierarchie.

For the moment, we formalize the most common notions of objects, in particular the existance of object-identifiers (oid) for each object under which it can be referenced in a *state*.

type-synonym *oid* = *nat*

We refrained from the alternative:

type-synonym *oid* = *ind*

which is slightly more abstract but non-executable.

States in UML/OCL are a pair of

- a partial map from oid's to elements of an *object universe*, i. e. the set of all possible object representations.
- and an oid-indexed family of *associations*, i. e. finite relations between objects living in a state. These relations can be n-ary which we model by nested lists.

For the moment we do not have to describe the concrete structure of the object universe and denote it by the polymorphic variable $'\mathfrak{A}$.

record $('\mathfrak{A})\mathit{state} =$
    $\mathit{heap} \;::\; \mathit{oid} \rightharpoonup {}'\mathfrak{A}$
    $\mathit{assocs} :: \mathit{oid} \rightharpoonup ((\mathit{oid}\ \mathit{list})\ \mathit{list})\ \mathit{list}$

In general, OCL operations are functions implicitly depending on a pair of pre- and post-state, i. e. *state transitions*. Since this will be reflected in our representation of OCL Types within HOL, we need to introduce the foundational concept of an object id (oid), which is just some infinite set, and some abstract notion of state.

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

We will require for all objects that there is a function that projects the oid of an object in the state (we will settle the question how to define this function later). We will use the Isabelle type class mechanism [**?**] to capture this:

FiXme: *Get Appropriate Reference!*

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

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

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

The major instance needed are instances constructed over options: once an object, options of objects are also objects.

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

Common Infrastructure for all OCL Types (II): Valuations as OCL Types

Since OCL operations in general depend on pre- and post-states, we will represent OCL types as *functions* from pre- and post-state to some HOL raw-type that contains exactly the data in the OCL type — see below. This gives rise to the idea that we represent OCL types by *Valuations*.

Valuations are 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 $('\mathfrak{A},'\alpha)\ \mathit{val} = {}'\mathfrak{A}\ \mathit{st} \Rightarrow {}'\alpha{::}\mathit{null}$

The definitions for the constants and operations based on valuations will be geared towards a format that Isabelle can check to be a "conservative" (i. e., logically safe) axiomatic definition. By introducing an explicit interpretation function (which happens to be defined just as the identity since we are using a shallow embedding of OCL into HOL), all these definitions can be rewritten into the conventional semantic textbook format as follows:

The fundamental constants 'invalid' and 'null' in all OCL Types

As a consequence of semantic domain definition, any OCL type will have the two semantic constants *invalid* (for exceptional, aborted computation) and *null*:

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

This conservative Isabelle definition of the polymorphic constant *invalid* is equivalent with the textbook definition:

lemma *textbook-invalid*: $I[\![\mathit{invalid}]\!]\tau = \mathit{bot}$
by(*simp add*: *invalid-def Sem-def*)

Note that the definition :

```
definition  null     ::  "('𝔄,'α::null)  val"
where       "null     ≡λ τ.  null"
```

is not necessary since we defined the entire function space over null types again as null-types; the crucial definition is $\mathit{null} \equiv \lambda x.\ \mathit{null}$. Thus, the polymorphic constant *null* is simply the result of a general type class construction. Nevertheless, we can derive the semantic textbook definition for the OCL null constant based on the abstract null:

lemma *textbook-null-fun*: $I[\![\mathit{null}{::}('\mathfrak{A},'\alpha{::}\mathit{null})\ \mathit{val}]\!]\ \tau = (\mathit{null}{::}('\alpha{::}\mathit{null}))$
by(*simp add*: *null-fun-def Sem-def*)

B.1.2. Basic OCL Value Types

The semantic domain of the (basic) boolean type is now defined as the Standard: the space of valuation to *bool option option*, i. e. the Boolean base type:

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

Because of the previous class definitions, Isabelle type-inference establishes that $'\mathfrak{A}$ *Boolean* lives actually both in the type class *UML-Types.bot-class.bot* and *null*; this type is sufficiently rich to contain at least these two elements. Analogously we build:

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

type-synonym $\mathit{String}_{\mathit{base}}$ = *string option option*
type-synonym $('\mathfrak{A})\mathit{String} = ('\mathfrak{A},\mathit{String}_{\mathit{base}})\ \mathit{val}$

type-synonym $\mathit{Real}_{base} = \mathit{real\ option\ option}$
type-synonym $('\mathfrak{A})\mathit{Real} = ('\mathfrak{A},\mathit{Real}_{base})\ \mathit{val}$

Since *Real* is again a basic type, we define its semantic domain as the valuations over *real option option* — i.e. the mathematical type of real numbers. The HOL-theory for *real* "Real" transcendental numbers such as π and e as well as infrastructure to reason over infinite convergent Cauchy-sequences (it is thus possible, in principle, to reason in Featherweight OCL that the sum of inverted two-s exponentials is actually 2.

If needed, a code-generator to compile *Real* to floating-point numbers can be added; this allows for mapping reals to an efficient machine representation; of course, this feature would be logically unsafe.

For technical reasons related to the Isabelle type inference for type-classes (we don't get the properties in the right order that class instantiation provides them, if we would follow the previous scheme), we give a slightly atypic definition:

typedef $\mathit{Void}_{base} = \{X{::}\mathit{unit\ option\ option}.\ X = \mathit{bot} \lor X = \mathit{null}\ \}$ **by**$(\mathit{rule\text{-}tac\ x{=}bot}$ **in** $\mathit{exI},\ \mathit{simp})$

type-synonym $('\mathfrak{A})\mathit{Void} = ('\mathfrak{A},\mathit{Void}_{base})\ \mathit{val}$

B.1.3. Some OCL Collection Types

The construction of collection types is sligtly more involved: We need to define an concrete type, constrain it via a kind of data-invariant to "legitimate elements" (i. e. in our type will be "no junk, no confusion"), and abstract it to a new type constructor.

The Construction of the Pair Type (Tuples)

The core of an own type construction is done via a type definition which provides the base-type $('\alpha,\ '\beta)\ \mathit{Pair}_{base}$. It is shown that this type "fits" indeed into the abstract type interface discussed in the previous section.

typedef $('\alpha,\ '\beta)\ \mathit{Pair}_{base} = \{X{::}('\alpha{::}\mathit{null} \times\ '\beta{::}\mathit{null})\ \mathit{option\ option}.$
$\quad X = \mathit{bot} \lor X = \mathit{null} \lor (\mathit{fst}\lceil\lceil X\rceil\rceil \neq \mathit{bot} \land \mathit{snd}\lceil\lceil X\rceil\rceil \neq \mathit{bot})\}$
by $(\mathit{rule\text{-}tac\ x{=}bot}$ **in** $\mathit{exI},\ \mathit{simp})$

We "carve" out from the concrete type $('\alpha \times\ '\beta)$ *option option* the new fully abstract type, which will not contain representations like $\lfloor\lfloor(\bot,\ a)\rfloor\rfloor$ or $\lfloor\lfloor(b,\ \bot)\rfloor\rfloor$. The type constuctor $\mathit{Pair}\{x,y\}$ to be defined later will identify these with *invalid*.

instantiation Pair_{base} :: $(\mathit{null},\mathit{null})\mathit{bot}$
begin
  definition $\mathit{bot\text{-}Pair}_{base}\mathit{\text{-}def}$: $(\mathit{bot\text{-}class.bot} :: ('a{::}\mathit{null},'b{::}\mathit{null})\ \mathit{Pair}_{base}) \equiv \mathit{Abs\text{-}Pair}_{base}\ \mathit{None}$

  instance proof show $\exists x{::}('a,'b)\ \mathit{Pair}_{base}.\ x \neq \mathit{bot}$
    apply$(\mathit{rule\text{-}tac\ x{=}Abs\text{-}Pair}_{base}\ \lfloor\mathit{None}\rfloor$ **in** $\mathit{exI})$
    by$(\mathit{simp\ add}{:}\ \mathit{bot\text{-}Pair}_{base}\mathit{\text{-}def}\ \ \mathit{Abs\text{-}Pair}_{base}\mathit{\text{-}inject}\ \ \mathit{null\text{-}option\text{-}def\ bot\text{-}option\text{-}def})$
  qed
end

instantiation Pair_{base} :: $(\mathit{null},\mathit{null})\mathit{null}$
begin

definition $null\text{-}Pair_{base}\text{-}def$: $(null::('a::null,'b::null)\ Pair_{base}) \equiv Abs\text{-}Pair_{base}\ \lfloor None \rfloor$

instance proof show $(null::('a::null,'b::null)\ Pair_{base}) \neq bot$
  by$(simp\ add\text{:}\ bot\text{-}Pair_{base}\text{-}def\ null\text{-}Pair_{base}\text{-}def\ Abs\text{-}Pair_{base}\text{-}inject$
    $null\text{-}option\text{-}def\ bot\text{-}option\text{-}def)$
 qed
end

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

type-synonym $('\mathfrak{A},'\alpha,'\beta)\ Pair\ = ('\mathfrak{A}, ('\alpha,'\beta)\ Pair_{base})\ val$

The Construction of the Set Type

The core of an own type construction is done via a type definition which provides the raw-type $'\alpha\ Set_{base}$. It is shown that this type "fits" indeed into the abstract type interface discussed in the previous section. Note that we make no restriction whatsoever to *finite* sets; the type constructor of Featherweight OCL is in fact infinite.

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

instantiation Set_{base} :: $(null)bot$
begin

definition $bot\text{-}Set_{base}\text{-}def$: $(bot::('a::null)\ Set_{base}) \equiv Abs\text{-}Set_{base}\ None$

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

instantiation Set_{base} :: $(null)null$
begin

definition $null\text{-}Set_{base}\text{-}def$: $(null::('a::null)\ Set_{base}) \equiv Abs\text{-}Set_{base}\ \lfloor None \rfloor$

instance proof show $(null::('a::null)\ Set_{base}) \neq bot$
  by$(simp\ add\text{:}null\text{-}Set_{base}\text{-}def\ bot\text{-}Set_{base}\text{-}def\ Abs\text{-}Set_{base}\text{-}inject$
    $null\text{-}option\text{-}def\ bot\text{-}option\text{-}def)$
 qed
end

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

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

The Construction of the Sequence Type

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

typedef $'\alpha$ *Sequence*$_{base}$ $=\{X::('\alpha::\mathit{null})$ *list option option*.
$X = \mathit{bot} \lor X = \mathit{null} \lor (\forall x{\in}\mathit{set}\ \lceil\lceil X\rceil\rceil.\ x \neq \mathit{bot})\}$
by (*rule-tac x=bot* **in** *exI*, *simp*)

instantiation *Sequence*$_{base}$ :: (*null*)*bot*
begin

definition *bot-Sequence*$_{base}$*-def*: $(\mathit{bot}::('a::\mathit{null})$ *Sequence*$_{base}$$) \equiv$ *Abs-Sequence*$_{base}$ *None*

instance proof show $\exists x::'a$ *Sequence*$_{base}$. $x \neq \mathit{bot}$
apply(*rule-tac x=Abs-Sequence*$_{base}$ $\lfloor$*None*$\rfloor$ **in** *exI*)
by(*auto simp*:*bot-Sequence*$_{base}$*-def Abs-Sequence*$_{base}$*-inject*
null-option-def bot-option-def)
qed
end

instantiation *Sequence*$_{base}$ :: (*null*)*null*
begin

definition *null-Sequence*$_{base}$*-def*: $(\mathit{null}::('a::\mathit{null})$ *Sequence*$_{base}$$) \equiv$ *Abs-Sequence*$_{base}$ $\lfloor$ *None* $\rfloor$

instance proof show $(\mathit{null}::('a::\mathit{null})$ *Sequence*$_{base}$$) \neq \mathit{bot}$
by(*auto simp*:*bot-Sequence*$_{base}$*-def null-Sequence*$_{base}$*-def Abs-Sequence*$_{base}$*-inject*
null-option-def bot-option-def)
qed
end

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

type-synonym $('\mathfrak{A},'\alpha)$ *Sequence* $= ('\mathfrak{A},\ '\alpha$ *Sequence*$_{base}$$)$ *val*

Discussion: The Representation of UML/OCL Types in Featherweight OCL

In the introduction, we mentioned that there is an “injective representation mapping” between the types of OCL and the types of Featherweight OCL (and its meta-language: HOL). This injectivity is at the heart of our representation technique — a so-called *shallow embedding* — and means: OCL types were mapped one-to-one to types in HOL, ruling out a resentation where everything is mapped on some common HOL-type, say “OCL-expression”, in which we would have to sort out the typing of OCL and its impact on the semantic representation function in an own, quite heavy side-calculus.

After the previous sections, we are now able to exemplify this representation as follows:
We do not formalize the representation map here; however, its principles are quite straight-forward:

1. cartesion products of arguments were curried,

2. constants of type `T` were mapped to valuations over the HOL-type for `T`,

3. functions `T -> T'` were mapped to functions in HOL, where `T` and `T'` were mapped to the valuations for them, and

OCL Type	HOL Type
`Boolean`	$'\mathfrak{A}\ \mathit{Boolean}$
`Boolean -> Boolean`	$'\mathfrak{A}\ \mathit{Boolean} \Rightarrow {}'\mathfrak{A}\ \mathit{Boolean}$
`(Integer,Integer) -> Boolean`	$'\mathfrak{A}\ \mathit{Integer} \Rightarrow {}'\mathfrak{A}\ \mathit{Integer} \Rightarrow {}'\mathfrak{A}\ \mathit{Boolean}$
`Set(Integer)`	$('\mathfrak{A}, \mathit{Integer}_{base})\ \mathit{Set}$
`Set(Integer)-> Real`	$('\mathfrak{A}, \mathit{Integer}_{base})\ \mathit{Set} \Rightarrow {}'\mathfrak{A}\ \mathit{Real}$
`Set(Pair(Integer,Boolean))`	$('\mathfrak{A}, (\mathit{Integer}_{base}, \mathit{Boolean}_{base})\ \mathit{Pair}_{base})\ \mathit{Set}$
`Set(<T>)`	$('\mathfrak{A}, '\alpha)\ \mathit{Set}$

Table B.1.: Basic semantic constant definitions of the logic (except *null*)

4. the arguments of type constructors `Set(T)` remain corresponding HOL base-types.

Note, furthermore, that our construction of “fully abstract types” (no junk, no confusion) assures that the logical equality to be defined in the next section works correctly and comes as element of the “lingua franca”, i. e. HOL.

end

B.2. Formalization II: OCL Terms and Library Operations

theory *UML-Logic*
imports *UML-Types*
begin

B.2.1. The Operations of the Boolean Type and the OCL Logic

Basic Constants

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

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

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

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

lemma *bool-split-0*: $X\ \tau = invalid\ \tau \lor X\ \tau = null\ \tau \lor$
$\quad X\ \tau = true\ \tau \quad \lor X\ \tau = false\ \tau$
apply(*simp add*: *invalid-def null-def true-def false-def*)
apply(*case-tac X* τ,*simp-all add*: *null-fun-def null-option-def bot-option-def*)
apply(*case-tac a*,*simp*)
apply(*case-tac aa*,*simp*)
apply *auto*
done

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

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

lemma *textbook-true*: $I[\![true]\!]\ \tau = \lfloor\lfloor True \rfloor\rfloor$
by(*simp add*: *Sem-def true-def*)

lemma *textbook-false*: $I[\![false]\!]\ \tau = \lfloor\lfloor False \rfloor\rfloor$
by(*simp add*: *Sem-def false-def*)

Name	Theorem
textbook-invalid	$I[\![invalid]\!]\ \tau = UML\text{-}Types.bot\text{-}class.bot$
textbook-null-fun	$I[\![null]\!]\ \tau = null$
textbook-true	$I[\![true]\!]\ \tau = \lfloor\lfloor True \rfloor\rfloor$
textbook-false	$I[\![false]\!]\ \tau = \lfloor\lfloor False \rfloor\rfloor$

Table B.2.: Basic semantic constant definitions of the logic (except *null*)

Validity and Definedness

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

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

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

lemma *valid2*[*simp*]: υ *null = true*

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

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

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

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

definition *defined* :: $('\mathfrak{A},'a::null)val \Rightarrow ('\mathfrak{A})Boolean$ (δ - [*100*]*100*)
where $\delta\ X \equiv\ \lambda\ \tau\ .$ *if* $X\ \tau = bot\ \tau\ \vee X\ \tau = null\ \tau$ *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*]: δ *true* = *true*
by(*rule ext*,*simp add*: *defined-def bot-fun-def bot-option-def null-is-valid null-option-def*
null-fun-def invalid-def true-def false-def)

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

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

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

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

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

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

The definitions above for the constants *defined* and *valid* can be rewritten into the conventional semantic "textbook" format as follows:

lemma *textbook-defined*: $I[\![\delta(X)]\!]\ \tau = (\mathit{if}\ I[\![X]\!]\ \tau = I[\![\mathit{bot}]\!]\ \tau\ \vee I[\![X]\!]\ \tau = I[\![\mathit{null}]\!]\ \tau$
$\mathit{then}\ I[\![\mathit{false}]\!]\ \tau$
$\mathit{else}\ I[\![\mathit{true}]\!]\ \tau)$
by(*simp add*: *Sem-def defined-def*)

lemma *textbook-valid*: $I[\![\upsilon(X)]\!]\ \tau = (\mathit{if}\ I[\![X]\!]\ \tau = I[\![\mathit{bot}]\!]\ \tau$
$\mathit{then}\ I[\![\mathit{false}]\!]\ \tau$
$\mathit{else}\ I[\![\mathit{true}]\!]\ \tau)$
by(*simp add*: *Sem-def valid-def*)

Table B.3 and Table B.4 summarize the results of this section.

Name	Theorem
textbook-defined	$I[\![\delta\ X]\!]\ \tau = (\mathit{if}\ I[\![X]\!]\ \tau = I[\![\mathit{UML\text{-}Types.bot\text{-}class.bot}]\!]\ \tau \vee I[\![X]\!]\ \tau = I[\![\mathit{null}]\!]\ \tau\ \mathit{then}\ I[\![\mathit{false}]\!]\ \tau\ \mathit{else}\ I[\![\mathit{true}]\!]\ \tau)$
textbook-valid	$I[\![\upsilon\ X]\!]\ \tau = (\mathit{if}\ I[\![X]\!]\ \tau = I[\![\mathit{UML\text{-}Types.bot\text{-}class.bot}]\!]\ \tau\ \mathit{then}\ I[\![\mathit{false}]\!]\ \tau\ \mathit{else}\ I[\![\mathit{true}]\!]\ \tau)$

Table B.3.: Basic predicate definitions of the logic.

The Equalities of OCL

The OCL contains a particular version of equality, written in Standard documents _ = _ and _ <> _ for its negation, which is referred as *weak referential equality* hereafter and for which we use the symbol _ $\doteq$ _ throughout the formal part of this document. Its semantics is motivated by the desire of fast execution, and similarity to languages like Java and C, but does not satisfy the needs of logical reasoning over OCL expressions and specifications. We therefore introduce a second equality, referred as *strong equality* or *logical equality* and written _ $\triangleq$ _ which is not present in the current standard but was discussed in prior texts on OCL like the Amsterdam Manifesto [17] and was identified as desirable extension of OCL in the Aachen Meeting [13] in

Name	Theorem
defined1	$\delta\ \mathit{invalid} = \mathit{false}$
defined2	$\delta\ \mathit{null} = \mathit{false}$
defined3	$\delta\ \mathit{true} = \mathit{true}$
defined4	$\delta\ \mathit{false} = \mathit{true}$
defined5	$\delta\ \delta\ X = \mathit{true}$
defined6	$\delta\ \upsilon\ X = \mathit{true}$

Table B.4.: Laws of the basic predicates of the logic.

the future 2.5 OCL Standard. The purpose of strong equality is to define and reason over OCL. It is therefore a natural task in Featherweight OCL to formally investigate the somewhat quite complex relationship between these two.

Strong equality has two motivations: a pragmatic one and a fundamental one.

1. The pragmatic reason is fairly simple: users of object-oriented languages want something like a "shallow object value equality". You will want to say a.boss $\triangleq$ b.boss@pre instead of

   a.boss $\doteq$ b.boss@pre **and** *(∗ just the pointers are equal! ∗)*
   a.boss.name $\doteq$ b.boss@pre.name@pre **and**
   a.boss.age $\doteq$ b.boss@pre.age@pre

   Breaking a shallow-object equality down to referential equality of attributes is cumbersome, error-prone, and makes specifications difficult to extend (add for example an attribute sex to your class, and check in your OCL specification everywhere that you did it right with your simulation of strong equality). Therefore, languages like Java offer facilities to handle two different equalities, and it is problematic even in an execution oriented specification language to ignore shallow object equality because it is so common in the code.

2. The fundamental reason goes as follows: whatever you do to reason consistently over a language, you need the concept of equality: you need to know what expressions can be replaced by others because they *mean the same thing.* People call this also "Leibniz Equality" because this philosopher brought this principle first explicitly to paper and shed some light over it. It is the theoretic foundation of what you do in an optimizing compiler: you replace expressions by *equal* ones, which you hope are easier to evaluate. In a typed language, strong equality exists uniformly over all types, it is "polymorphic" $\_ = \_ :: \alpha * \alpha \to \mathit{bool}$—this is the way that equality is defined in HOL itself. We can express Leibniz principle as one logical rule of surprising simplicity and beauty:

   $$s = t \Longrightarrow P(s) = P(t) \tag{B.1}$$

   "Whenever we know, that s is equal to t, we can replace the sub-expression s in a term P by t and we have that the replacement is equal to the original."

While weak referential equality is defined to be strict in the OCL standard, we will define strong equality as non-strict. It is quite nasty (but not impossible) to define the logical equality in a strict way (the substitutivity rule above would look more complex), however, whenever references were used, strong equality is needed since references refer to particular states (pre or post), and that they mean the same thing can therefore not be taken for granted.

Definition 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.

We define strong equality extremely generic, even for types that contain a *null* or $\bot$ element. Strong equality is simply polymorphic in Featherweight OCL, i. e., is defined identical for all types in OCL and HOL.

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

From this follow already elementary properties like:

lemma [*simp*,*code-unfold*]: (*true* $\triangleq$ *false*) = *false*
by(*rule ext*, *auto simp*: *StrongEq-def*)

lemma [*simp*,*code-unfold*]: (*false* $\triangleq$ *true*) = *false*
by(*rule ext*, *auto simp*: *StrongEq-def*)

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

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

lemma *StrongEq-sym*: ($X \triangleq Y$) = ($Y \triangleq X$)
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 Featherweight 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 post state 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: ⋀X. P(X)τ = P(λ -. X τ)τ
 and     eq: (X ≜ Y)τ = true τ
 shows   (P X ≜ P Y)τ = true τ
 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
```

```
lemma defined7[simp]: δ (X ≜ Y) = true
 by(rule ext,
   auto simp: defined-def          true-def false-def StrongEq-def
          bot-fun-def bot-option-def null-option-def null-fun-def)
```

```
lemma valid7[simp]: υ (X ≜ 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 ≜ Y) τ = ((λ -. X τ) ≜ (λ -. Y τ)) τ
by(simp add: StrongEq-def)
```

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 normalization 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 OclNot :: ('𝔄)Boolean ⇒ ('𝔄)Boolean (not)
where    not X ≡ λ τ . case X τ of
                      ⊥   ⇒ ⊥
                   | ⌊ ⊥ ⌋  ⇒ ⌊ ⊥ ⌋
                   | ⌊⌊ x ⌋⌋ ⇒ ⌊⌊ ¬ x ⌋⌋
```

```
lemma cp-OclNot: (not X)τ = (not (λ -. X τ)) τ
by(simp add: OclNot-def)
```

```
lemma OclNot1[simp]: not invalid = invalid
  by(rule ext,simp add: OclNot-def null-def invalid-def true-def false-def bot-option-def)

lemma OclNot2[simp]: not null = null
  by(rule ext,simp add: OclNot-def null-def invalid-def true-def false-def
                    bot-option-def null-fun-def null-option-def )

lemma OclNot3[simp]: not true = false
  by(rule ext,simp add: OclNot-def null-def invalid-def true-def false-def)

lemma OclNot4[simp]: not false = true
  by(rule ext,simp add: OclNot-def null-def invalid-def true-def false-def)


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

lemma OclNot-inject: ⋀ x y. not x = not y ⟹ x = y
  by(subst OclNot-not[THEN sym], simp)

definition OclAnd :: [('𝔄)Boolean, ('𝔄)Boolean] ⇒ ('𝔄)Boolean (infixl and 30)
where    X and Y ≡  (λ τ . case X τ of
                   ⌊⌊False⌋⌋ ⇒              ⌊⌊False⌋⌋
                 | ⊥        ⇒ (case Y τ of
                             ⌊⌊False⌋⌋ ⇒ ⌊⌊False⌋⌋
                           | -        ⇒ ⊥)
                 | ⌊⊥⌋      ⇒ (case Y τ of
                             ⌊⌊False⌋⌋ ⇒ ⌊⌊False⌋⌋
                           | ⊥        ⇒ ⊥
                           | -        ⇒ ⌊⊥⌋)
                 | ⌊⌊True⌋⌋  ⇒              Y τ)
```

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*.

In textbook notation, the logical core constructs *not* and *op and* were represented as follows:

```
lemma textbook-OclNot:
    I⟦not(X)⟧ τ =  (case I⟦X⟧ τ of  ⊥   ⇒ ⊥
                             |  ⌊ ⊥ ⌋ ⇒ ⌊ ⊥ ⌋
                             | ⌊⌊ x ⌋⌋ ⇒ ⌊⌊ ¬ x ⌋⌋)
by(simp add: Sem-def OclNot-def)

lemma textbook-OclAnd:
    I⟦X and Y⟧ τ = (case I⟦X⟧ τ of
                  ⊥  ⇒ (case I⟦Y⟧ τ of
                         ⊥ ⇒ ⊥
                       | ⌊⊥⌋ ⇒ ⊥
                       | ⌊⌊True⌋⌋ ⇒ ⊥
                       | ⌊⌊False⌋⌋ ⇒ ⌊⌊False⌋⌋)
           | ⌊ ⊥ ⌋ ⇒ (case I⟦Y⟧ τ of
                         ⊥ ⇒ ⊥
                       | ⌊⊥⌋ ⇒ ⌊⊥⌋
                       | ⌊⌊True⌋⌋ ⇒ ⌊⊥⌋
                       | ⌊⌊False⌋⌋ ⇒ ⌊⌊False⌋⌋)
           | ⌊⌊True⌋⌋ ⇒ (case I⟦Y⟧ τ of
                         ⊥ ⇒ ⊥
                       | ⌊⊥⌋ ⇒ ⌊⊥⌋
                       | ⌊⌊y⌋⌋ ⇒ ⌊⌊y⌋⌋)
           | ⌊⌊False⌋⌋ ⇒ ⌊⌊ False ⌋⌋)
by(simp add: OclAnd-def Sem-def split: option.split bool.split)

definition OclOr :: [('𝔄)Boolean, ('𝔄)Boolean] ⇒ ('𝔄)Boolean          (infixl or 25)
where   X or Y ≡ not(not X and not Y)

definition OclImplies :: [('𝔄)Boolean, ('𝔄)Boolean] ⇒ ('𝔄)Boolean      (infixl implies 25)
where   X implies Y ≡ not X or Y



lemma cp-OclAnd:(X and Y) τ = ((λ -. X τ) and (λ -. Y τ)) τ
by(simp add: OclAnd-def)

lemma cp-OclOr:((X::('𝔄)Boolean) or Y) τ = ((λ -. X τ) or (λ -. Y τ)) τ
apply(simp add: OclOr-def)
apply(subst cp-OclNot[of not (λ-. X τ) and not (λ-. Y τ)])
apply(subst cp-OclAnd[of not (λ-. X τ) not (λ-. Y τ)])
by(simp add: cp-OclNot[symmetric] cp-OclAnd[symmetric] )


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

lemma OclAnd1[simp]: (invalid and true) = invalid
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def)
lemma OclAnd2[simp]: (invalid and false) = false
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def)
lemma OclAnd3[simp]: (invalid and null) = invalid
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def
                  null-fun-def null-option-def)
lemma OclAnd4[simp]: (invalid and invalid) = invalid
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def)
```

```
lemma OclAnd5[simp]: (null and true) = null
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def
                    null-fun-def null-option-def)
lemma OclAnd6[simp]: (null and false) = false
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def
                    null-fun-def null-option-def)
lemma OclAnd7[simp]: (null and null) = null
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def
                    null-fun-def null-option-def)
lemma OclAnd8[simp]: (null and invalid) = invalid
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def
                    null-fun-def null-option-def)

lemma OclAnd9[simp]: (false and true) = false
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def)
lemma OclAnd10[simp]: (false and false) = false
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def)
lemma OclAnd11[simp]: (false and null) = false
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def)
lemma OclAnd12[simp]: (false and invalid) = false
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def)

lemma OclAnd13[simp]: (true and true) = true
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def)
lemma OclAnd14[simp]: (true and false) = false
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def)
lemma OclAnd15[simp]: (true and null) = null
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def
                    null-fun-def null-option-def)
lemma OclAnd16[simp]: (true and invalid) = invalid
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def
                    null-fun-def null-option-def)

lemma OclAnd-idem[simp]: (X and X) = X
  apply(rule ext,simp add: OclAnd-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 OclAnd-commute: (X and Y) = (Y and X)
  by(rule ext,auto simp:true-def false-def OclAnd-def invalid-def
                split: option.split option.split-asm
                       bool.split bool.split-asm)


lemma OclAnd-false1[simp]: (false and X) = false
```

```
 apply(rule ext, simp add: OclAnd-def)
 apply(auto simp:true-def false-def invalid-def
          split: option.split option.split-asm)
 done

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


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

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

lemma OclAnd-bot1[simp]: ⋀τ. X τ ≠ false τ ⟹ (bot and X) τ = bot τ
 apply(simp add: OclAnd-def)
 apply(auto simp:true-def false-def bot-fun-def bot-option-def
          split: option.split option.split-asm)
done

lemma OclAnd-bot2[simp]: ⋀τ. X τ ≠ false τ ⟹ (X and bot) τ = bot τ
 by(simp add: OclAnd-commute)

lemma OclAnd-null1[simp]: ⋀τ. X τ ≠ false τ ⟹ X τ ≠ bot τ ⟹ (null and X) τ = null τ
 apply(simp add: OclAnd-def)
 apply(auto simp:true-def false-def bot-fun-def bot-option-def null-fun-def null-option-def
          split: option.split option.split-asm)
done

lemma OclAnd-null2[simp]: ⋀τ. X τ ≠ false τ ⟹ X τ ≠ bot τ ⟹ (X and null) τ = null τ
 by(simp add: OclAnd-commute)

lemma OclAnd-assoc: (X and (Y and Z)) = (X and Y and Z)
 apply(rule ext, simp add: OclAnd-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 OclOr1[simp]: (invalid or true) = true
by(rule ext, simp add: OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def
                  bot-option-def)
lemma OclOr2[simp]: (invalid or false) = invalid
```

by(*rule ext*, *simp add*: *OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def*
bot-option-def)
lemma *OclOr3*[*simp*]: (*invalid or null*) = *invalid*
by(*rule ext*, *simp add*: *OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def*
bot-option-def null-fun-def null-option-def)
lemma *OclOr4*[*simp*]: (*invalid or invalid*) = *invalid*
by(*rule ext*, *simp add*: *OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def*
bot-option-def)

lemma *OclOr5*[*simp*]: (*null or true*) = *true*
by(*rule ext*, *simp add*: *OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def*
bot-option-def null-fun-def null-option-def)
lemma *OclOr6*[*simp*]: (*null or false*) = *null*
by(*rule ext*, *simp add*: *OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def*
bot-option-def null-fun-def null-option-def)
lemma *OclOr7*[*simp*]: (*null or null*) = *null*
by(*rule ext*, *simp add*: *OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def*
bot-option-def null-fun-def null-option-def)
lemma *OclOr8*[*simp*]: (*null or invalid*) = *invalid*
by(*rule ext*, *simp add*: *OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def*
bot-option-def null-fun-def null-option-def)

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

lemma *OclOr-commute*: (*X or Y*) = (*Y or X*)
by(*simp add*: *OclOr-def OclAnd-commute*)

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

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

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

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

lemma *OclOr-bot1*[*simp*]: $\bigwedge\tau.\ X\ \tau \neq \mathit{true}\ \tau \Longrightarrow (\mathit{bot\ or\ X})\ \tau = \mathit{bot}\ \tau$
apply(*simp add*: *OclOr-def OclAnd-def OclNot-def*)
apply(*auto simp*:*true-def false-def bot-fun-def bot-option-def*
split: *option.split option.split-asm*)
done

lemma *OclOr-bot2*[*simp*]: $\bigwedge\tau.\ X\ \tau \neq \mathit{true}\ \tau \Longrightarrow (\mathit{X\ or\ bot})\ \tau = \mathit{bot}\ \tau$
by(*simp add*: *OclOr-commute*)

lemma *OclOr-null1*[*simp*]: $\bigwedge\tau.\ X\ \tau \neq true\ \tau \Longrightarrow X\ \tau \neq bot\ \tau \Longrightarrow (null\ or\ X)\ \tau = null\ \tau$
  apply(*simp add*: *OclOr-def OclAnd-def OclNot-def*)
  apply(*auto simp*:*true-def false-def bot-fun-def bot-option-def null-fun-def null-option-def*
      split: *option.split option.split-asm*)
  apply (*metis* (*full-types*) *bool.simps*(*3*) *bot-option-def null-is-valid null-option-def*)
by (*metis* (*full-types*) *bool.simps*(*3*) *option.distinct*(*1*) *the.simps*)

lemma *OclOr-null2*[*simp*]: $\bigwedge\tau.\ X\ \tau \neq true\ \tau \Longrightarrow X\ \tau \neq bot\ \tau \Longrightarrow (X\ or\ null)\ \tau = null\ \tau$
  by(*simp add*: *OclOr-commute*)

lemma *OclOr-assoc*: $(X\ or\ (Y\ or\ Z)) = (X\ or\ Y\ or\ Z)$
  by(*simp add*: *OclOr-def OclAnd-assoc*)

lemma *OclImplies-true*: $(X\ implies\ true) = true$
  by (*simp add*: *OclImplies-def OclOr-true2*)

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

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

A Standard Logical Calculus for OCL

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

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

lemma *transform1-rev*: $\forall\ \tau.\ \tau \models P \Longrightarrow P = true$
by(*rule ext*, *auto simp*: *OclValid-def true-def*)

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

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

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

lemma
assumes $H : P = \mathit{true} \Longrightarrow Q = \mathit{true}$
shows $\tau \models P \Longrightarrow \tau \models Q$
apply(*simp add*: *OclValid-def*)
apply(*rule H*[*THEN fun-cong*])
apply(*rule ext*)
oops

Local Validity and Meta-logic

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

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

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

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

lemma *bool-split*[*simp*]:
$(\tau \models (x \triangleq \mathit{invalid})) \lor (\tau \models (x \triangleq \mathit{null})) \lor (\tau \models (x \triangleq \mathit{true})) \lor (\tau \models (x \triangleq \mathit{false}))$
apply(*insert bool-split-0*[*of x* τ], *auto*)
apply(*simp-all add*: *OclValid-def StrongEq-def true-def null-def invalid-def*)
done

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

lemma *valid-bool-split*: $(\tau \models \upsilon\ A) = ((\tau \models A \triangleq \mathit{null}) \lor (\tau \models A) \lor\ (\tau \models \mathit{not}\ A))$
by(*auto simp*:*valid-def true-def false-def invalid-def null-def OclNot-def*
StrongEq-def OclValid-def bot-fun-def bot-option-def null-option-def null-fun-def)

lemma *defined-bool-split*: $(\tau \models \delta\ A) = ((\tau \models A) \lor (\tau \models \mathit{not}\ A))$
by(*auto simp*:*defined-def true-def false-def invalid-def null-def OclNot-def*
StrongEq-def OclValid-def bot-fun-def bot-option-def null-option-def null-fun-def)

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

lemma *foundation6*:

$\tau \models P \Longrightarrow \tau \models \delta\ P$
by(*simp add*: *OclNot-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*]:
$(\tau \models not\ (\delta\ x)) = (\neg\ (\tau \models \delta\ x))$
by(*simp add*: *OclNot-def OclValid-def true-def false-def defined-def*
split: *option.split option.split-asm*)

lemma *foundation7'*[*simp*]:
$(\tau \models not\ (\upsilon\ x)) = (\neg\ (\tau \models \upsilon\ x))$
by(*simp add*: *OclNot-def OclValid-def true-def false-def valid-def*
split: *option.split option.split-asm*)

Key theorem for the δ-closure: either an expression is defined, or it can be replaced (substituted via *StrongEq-L-subst2*; see below) by *invalid* or *null*. Strictness-reduction rules will usually reduce these substituted terms drastically.

lemma *foundation8*:
$(\tau \models \delta\ x) \vee (\tau \models (x \triangleq invalid)) \vee (\tau \models (x \triangleq null))$
proof –
have *1* : $(\tau \models \delta\ x) \vee (\neg(\tau \models \delta\ x))$ **by** *auto*
have *2* : $(\neg(\tau \models \delta\ x)) = ((\tau \models (x \triangleq invalid)) \vee (\tau \models (x \triangleq null)))$
by(*simp only*: *defined-split*, *simp*)
show *?thesis* **by**(*insert 1*, *simp add*:*2*)
qed

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

lemma *foundation9'*:
$\tau \models not\ x \Longrightarrow \neg\ (\tau \models x)$
by(*auto simp*: *foundation6 foundation9*)

lemma *foundation9''*:
$\tau \models not\ x \Longrightarrow \tau \models \delta\ x$
by(*metis OclNot3 OclNot-not OclValid-def cp-OclNot cp-defined defined4*)

lemma *foundation10*:
$\tau \models \delta\ x \Longrightarrow \tau \models \delta\ y \Longrightarrow (\tau \models (x\ and\ y)) = (\ (\tau \models x) \wedge (\tau \models y))$
apply(*simp add*: *defined-split*)
by(*auto simp*: *OclAnd-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 *foundation10'*: $(\tau \models (A\ \mathit{and}\ B)) = ((\tau \models A) \wedge (\tau \models B))$
by(*auto dest*:*foundation5 simp*:*foundation6 foundation10*)

lemma *foundation11*:
$\tau \models \delta\ x \Longrightarrow\ \tau \models \delta\ y \Longrightarrow (\tau \models (x\ \mathit{or}\ y)) = (\ (\tau \models x) \vee (\tau \models y))$
apply(*simp add*: *defined-split*)
by(*auto simp*: *OclNot-def OclOr-def OclAnd-def OclValid-def invalid-def*
true-def null-def StrongEq-def null-fun-def null-option-def bot-option-def
split:*bool.split-asm bool.split*)

lemma *foundation12*:
$\tau \models \delta\ x \Longrightarrow (\tau \models (x\ \mathit{implies}\ y)) = (\ (\tau \models x) \longrightarrow (\tau \models y))$
apply(*simp add*: *defined-split*)
by(*auto simp*: *OclNot-def OclOr-def OclAnd-def OclImplies-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 option.split-asm*)

lemma *foundation13*:$(\tau \models A \triangleq \mathit{true}) \quad = (\tau \models A)$
by(*auto simp*: *OclNot-def OclValid-def invalid-def true-def null-def StrongEq-def*
split:*bool.split-asm bool.split*)

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

lemma *foundation15*:$(\tau \models A \triangleq \mathit{invalid}) = (\tau \models \mathit{not}(\upsilon\ A))$
by(*auto simp*: *OclNot-def OclValid-def valid-def invalid-def false-def true-def null-def*
StrongEq-def bot-option-def null-fun-def null-option-def bot-option-def bot-fun-def
split:*bool.split-asm bool.split option.split*)

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

lemma *foundation16''*: $\neg(\tau \models (\delta\ X)) = ((\tau \models (X \triangleq \mathit{invalid})) \vee (\tau \models (X \triangleq \mathit{null})))$
apply(*simp add*: *foundation16*)
by(*auto simp*:*defined-def false-def true-def bot-fun-def null-fun-def OclValid-def StrongEq-def invalid-def*)

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

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

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

lemma *foundation18″*: $(\tau \models (\upsilon\ X)\ ) =\ (\neg(\tau \models (X \triangleq \mathit{invalid})))$
by(*auto simp*:*foundation15*)

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

lemma *foundation21*: $(\mathit{not}\ A \triangleq \mathit{not}\ B) = (A \triangleq B)$
by(*rule ext*, *auto simp*: *OclNot-def StrongEq-def*
        split: *bool.split-asm HOL.split-if-asm option.split*)

lemma *foundation22*: $(\tau \models (X \triangleq Y)) = (X\ \tau = Y\ \tau)$
by(*auto simp*: *StrongEq-def OclValid-def true-def*)

lemma *foundation23*: $(\tau \models P) = (\tau \models (\lambda\ \text{-}\ .\ P\ \tau))$
by(*auto simp*: *OclValid-def true-def*)

lemma *foundation24*:$(\tau \models \mathit{not}(X \triangleq Y)) = (X\ \tau \neq Y\ \tau)$
by(*simp add*: *StrongEq-def OclValid-def OclNot-def true-def*)

lemma *foundation25*: $\tau \models P \Longrightarrow \tau \models (P\ \mathit{or}\ Q)$
by(*simp add*: *OclOr-def OclNot-def OclAnd-def OclValid-def true-def*)

lemma *foundation25′*: $\tau \models Q \Longrightarrow \tau \models (P\ \mathit{or}\ Q)$
by(*subst OclOr-commute*, *simp add*:*foundation25*)

lemma *foundation26*:
assumes *defP*: $\tau \models \delta\ P$
assumes *defQ*: $\tau \models \delta\ Q$
assumes *H*: $\tau \models (P\ \mathit{or}\ Q)$
assumes *P*: $\tau \models P \Longrightarrow R$
assumes *Q*: $\tau \models Q \Longrightarrow R$
shows *R*

by(*insert H*, *subst* (*asm*) *foundation11*[*OF defP defQ*], *erule disjE*, *simp-all add*: *P Q*)

lemma *foundation27*: $(\tau \models (A\ and\ B)) = ((\tau \models A) \wedge (\tau \models B))$
by(*auto dest*:*foundation5 simp*:*foundation6 foundation10*)

lemma *defined-not-I* : $\tau \models \delta\ (x) \Longrightarrow \tau \models \delta\ (not\ x)$
  by(*auto simp*: *OclNot-def null-def invalid-def defined-def valid-def OclValid-def*
      true-def false-def bot-option-def null-option-def null-fun-def bot-fun-def
    split: *option.split-asm HOL.split-if-asm*)

lemma *valid-not-I* : $\tau \models \upsilon\ (x) \Longrightarrow \tau \models \upsilon\ (not\ x)$
  by(*auto simp*: *OclNot-def null-def invalid-def defined-def valid-def OclValid-def*
      true-def false-def bot-option-def null-option-def null-fun-def bot-fun-def
    split: *option.split-asm option.split HOL.split-if-asm*)

lemma *defined-and-I* : $\tau \models \delta\ (x) \Longrightarrow\ \tau \models \delta\ (y) \Longrightarrow \tau \models \delta\ (x\ and\ y)$
  apply(*simp add*: *OclAnd-def null-def invalid-def defined-def valid-def OclValid-def*
      true-def false-def bot-option-def null-option-def null-fun-def bot-fun-def
    split: *option.split-asm HOL.split-if-asm*)
  apply(*auto simp*: *null-option-def split*: *bool.split*)
  by(*case-tac ya*,*simp-all*)

lemma *valid-and-I* : $\tau \models \upsilon\ (x) \Longrightarrow\ \tau \models \upsilon\ (y) \Longrightarrow \tau \models \upsilon\ (x\ and\ y)$
  apply(*simp add*: *OclAnd-def null-def invalid-def defined-def valid-def OclValid-def*
      true-def false-def bot-option-def null-option-def null-fun-def bot-fun-def
    split: *option.split-asm HOL.split-if-asm*)
  by(*auto simp*: *null-option-def split*: *option.split bool.split*)

lemma *defined-or-I* : $\tau \models \delta\ (x) \Longrightarrow\ \tau \models \delta\ (y) \Longrightarrow \tau \models \delta\ (x\ or\ y)$
by(*simp add*: *OclOr-def defined-and-I defined-not-I*)

lemma *valid-or-I* : $\tau \models \upsilon\ (x) \Longrightarrow\ \tau \models \upsilon\ (y) \Longrightarrow \tau \models \upsilon\ (x\ or\ y)$
by(*simp add*: *OclOr-def valid-and-I valid-not-I*)

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

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

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

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

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

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

lemma *StrongEq-L-subst1*: $\bigwedge\ \tau.\ cp\ P \Longrightarrow \tau \models (x \triangleq y) \Longrightarrow \tau \models (P\ x \triangleq P\ y)$
by(*auto simp*: *OclValid-def StrongEq-def true-def cp-def*)

lemma *StrongEq-L-subst2*:
$\bigwedge\ \tau.\ \ cp\ P \Longrightarrow \tau \models (x \triangleq y) \Longrightarrow \tau \models (P\ x) \Longrightarrow \tau \models (P\ y)$
by(*auto simp*: *OclValid-def StrongEq-def true-def cp-def*)

lemma *StrongEq-L-subst2-rev*: $\tau \models y \triangleq x \Longrightarrow cp\ P \Longrightarrow \tau \models P\ x \Longrightarrow \tau \models P\ y$
apply(*erule StrongEq-L-subst2*)
apply(*erule StrongEq-L-sym*)
by *assumption*

lemma *StrongEq-L-subst3*:
assumes *cp*: *cp P*
and *eq*: $\tau \models (x \triangleq y)$
shows $(\tau \models P\ x) = (\tau \models P\ y)$
apply(*rule iffI*)
apply(*rule StrongEq-L-subst2*[*OF cp*,*OF eq*],*simp*)
apply(*rule StrongEq-L-subst2*[*OF cp*,*OF eq*[*THEN StrongEq-L-sym*]],*simp*)
done

lemma *StrongEq-L-subst3-rev*:
assumes *eq*: $\tau \models (x \triangleq y)$
and *cp*: *cp P*
shows $(\tau \models P\ x) = (\tau \models P\ y)$
by(*insert cp*, *erule StrongEq-L-subst3*, *rule eq*)

lemma *StrongEq-L-subst4-rev*:
assumes *eq*: $\tau \models (x \triangleq y)$
and *cp*: *cp P*
shows $(\neg(\tau \models P\ x)) = (\neg(\tau \models P\ y))$
thm *arg-cong*[*of - - Not*]
apply(*rule arg-cong*[*of - - Not*])
by(*insert cp*, *erule StrongEq-L-subst3*, *rule eq*)

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

lemma *cpI2*:

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

lemma *cpI3*:
$(\forall\ X\ Y\ Z\ \tau.\ f\ X\ Y\ Z\ \tau = f(\lambda\text{-}.\ X\ \tau)(\lambda\text{-}.\ Y\ \tau)(\lambda\text{-}.\ Z\ \tau)\ \tau) \Longrightarrow$
$\ cp\ P \Longrightarrow cp\ Q \Longrightarrow cp\ R \Longrightarrow cp(\lambda X.\ f\ (P\ X)\ (Q\ X)\ (R\ X))$
apply(*auto simp*: *cp-def*)
apply(*rule exI*, (*rule allI*)+)
by(*erule-tac x=P X* **in** *allE*, *auto*)

lemma *cpI4*:
$(\forall\ W\ X\ Y\ Z\ \tau.\ f\ W\ X\ Y\ Z\ \tau = f(\lambda\text{-}.\ W\ \tau)(\lambda\text{-}.\ X\ \tau)(\lambda\text{-}.\ Y\ \tau)(\lambda\text{-}.\ Z\ \tau)\ \tau) \Longrightarrow$
$\ cp\ P \Longrightarrow cp\ Q \Longrightarrow cp\ R \Longrightarrow cp\ S \Longrightarrow cp(\lambda X.\ f\ (P\ X)\ (Q\ X)\ (R\ X)\ (S\ X))$
apply(*auto simp*: *cp-def*)
apply(*rule exI*, (*rule allI*)+)
by(*erule-tac x=P X* **in** *allE*, *auto*)

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

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

lemmas *cp-intro*[*intro*!,*simp*,*code-unfold*] =
  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-OclNot[*THEN allI*[*THEN allI*[*THEN cpI1*], *of not*]]
  cp-OclAnd[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of op and*]]
  cp-OclOr[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of op or*]]
  cp-OclImplies[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of op implies*]]
  cp-StrongEq[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]],
    of StrongEq]]

OCL's if then else endif

definition *OclIf* :: $[({}'\mathfrak{A})Boolean\ ,\ ({}'\mathfrak{A},{}'\alpha{::}null)\ val,\ ({}'\mathfrak{A},{}'\alpha)\ val] \Rightarrow ({}'\mathfrak{A},{}'\alpha)\ val$
    (*if* (-) *then* (-) *else* (-) *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 *cp-OclIf*:((*if C then* B_1 *else* B_2 *endif*) τ =
(*if* (λ -. *C* τ) *then* (λ -. B_1 τ) *else* (λ -. B_2 τ) *endif*) τ)
by(*simp only*: *OclIf-def*, *subst cp-defined*, *rule refl*)

lemmas *cp-intro*′[*intro*!,*simp*,*code-unfold*] =
cp-intro
cp-OclIf[*THEN allI*[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI3*]]]], *of OclIf*]]

lemma *OclIf-invalid* [*simp*]: (*if invalid then* B_1 *else* B_2 *endif*) = *invalid*
by(*rule ext*, *auto simp*: *OclIf-def*)

lemma *OclIf-null* [*simp*]: (*if null then* B_1 *else* B_2 *endif*) = *invalid*
by(*rule ext*, *auto simp*: *OclIf-def*)

lemma *OclIf-true* [*simp*]: (*if true then* B_1 *else* B_2 *endif*) = B_1
by(*rule ext*, *auto simp*: *OclIf-def*)

lemma *OclIf-true*′ [*simp*]: $\tau \models P \Longrightarrow$ (*if P then* B_1 *else* B_2 *endif*)$\tau = B_1\ \tau$
apply(*subst cp-OclIf*,*auto simp*: *OclValid-def*)
by(*simp add*:*cp-OclIf*[*symmetric*])

lemma *OclIf-true*″ [*simp*]: $\tau \models P \Longrightarrow \tau \models$ (*if P then* B_1 *else* B_2 *endif*) $\triangleq B_1$
by(*subst OclValid-def*, *simp add*: *StrongEq-def true-def*)

lemma *OclIf-false* [*simp*]: (*if false then* B_1 *else* B_2 *endif*) = B_2
by(*rule ext*, *auto simp*: *OclIf-def*)

lemma *OclIf-false*′ [*simp*]: $\tau \models$ *not P* $\Longrightarrow$ (*if P then* B_1 *else* B_2 *endif*)$\tau = B_2\ \tau$
apply(*subst cp-OclIf*)
apply(*auto simp*: *foundation14*[*symmetric*] *foundation22*)
by(*auto simp*: *cp-OclIf*[*symmetric*])

lemma *OclIf-idem1*[*simp*]:(*if* δ *X then A else A endif*) = *A*
by(*rule ext*, *auto simp*: *OclIf-def*)

lemma *OclIf-idem2*[*simp*]:(*if* υ *X then A else A endif*) = *A*
by(*rule ext*, *auto simp*: *OclIf-def*)

lemma *OclNot-if*[*simp*]:
not(*if P then C else E endif*) = (*if P then not C else not E endif*)

apply(*rule OclNot-inject*, *simp*)
apply(*rule ext*)
apply(*subst cp-OclNot*, *simp add*: *OclIf-def*)
apply(*subst cp-OclNot*[*symmetric*])+
by *simp*

Fundamental Predicates on Basic Types: Strict (Referential) Equality

In contrast to logical equality, the OCL standard defines an equality operation which we call “strict referential equality”. It behaves differently for all types—on value types, it is basically a strict version of strong equality, for defined values it behaves identical. But on object types it will compare their references within the store. We introduce strict referential equality as an *overloaded* concept and will handle it for each type instance individually.

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

with term "not" we can express the notation:

syntax
notequal :: $('\mathfrak{A})Boolean \Rightarrow ('\mathfrak{A})Boolean \Rightarrow ('\mathfrak{A})Boolean$ (**infix** <> *40*)
translations
$a <> b == CONST\ OclNot(a \doteq b)$

We will define instances of this equality in a case-by-case basis.

Laws to Establish Definedness (δ-closure)

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

lemma *OclNot-defargs*:
$\tau \models (not\ P) \Longrightarrow \tau \models \delta\ P$
by(*auto simp*: *OclNot-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*)

lemma *OclNot-contrapos-nn*:
assumes *A*: $\tau \models \delta\ A$
assumes *B*: $\tau \models not\ B$
assumes *C*: $\tau \models A \Longrightarrow \tau \models B$
shows $\tau \models not\ A$
proof –
have *D* : $\tau \models \delta\ B$ **by**(*rule B*[*THEN OclNot-defargs*])
show *?thesis*
apply(*insert B*,*simp add*: *A D foundation9*)
by(*erule contrapos-nn*, *auto intro*: *C*)
qed

A Side-calculus for Constant Terms

definition $const\ X \equiv \forall\ \tau\ \tau'.\ X\ \tau = X\ \tau'$

lemma *const-charn*: $const\ X \Longrightarrow X\ \tau = X\ \tau'$
by(*auto simp*: *const-def*)

lemma *const-subst*:

```
 assumes const-X: const X
    and const-Y: const Y
    and eq :    X τ = Y τ
    and cp-P:   cp P
    and pp :    P Y τ = P Y τ'
  shows P X τ = P X τ'
proof –
  have A: ⋀Y. P Y τ = P (λ-. Y τ) τ
     apply(insert cp-P, unfold cp-def)
     apply(elim exE, erule-tac x=Y in allE', erule-tac x=τ in allE)
     apply(erule-tac x=(λ-. Y τ) in allE, erule-tac x=τ in allE)
     by simp
  have B: ⋀Y. P Y τ' = P (λ-. Y τ') τ'
     apply(insert cp-P, unfold cp-def)
     apply(elim exE, erule-tac x=Y in allE', erule-tac x=τ' in allE)
     apply(erule-tac x=(λ-. Y τ') in allE, erule-tac x=τ' in allE)
     by simp
  have C: X τ' = Y τ'
     apply(rule trans, subst const-charn[OF const-X],rule eq)
     by(rule const-charn[OF const-Y])
  show ?thesis
     apply(subst A, subst B, simp add: eq C)
     apply(subst A[symmetric],subst B[symmetric])
     by(simp add:pp)
qed


lemma const-imply2 :
 assumes ⋀τ τ'. P τ = P τ' ⟹ Q τ = Q τ'
 shows const P ⟹ const Q
by(simp add: const-def, insert assms, blast)

lemma const-imply3 :
 assumes ⋀τ τ'. P τ = P τ' ⟹ Q τ = Q τ' ⟹ R τ = R τ'
 shows const P ⟹ const Q ⟹ const R
by(simp add: const-def, insert assms, blast)

lemma const-imply4 :
 assumes ⋀τ τ'. P τ = P τ' ⟹ Q τ = Q τ' ⟹ R τ = R τ' ⟹ S τ = S τ'
 shows const P ⟹ const Q ⟹ const R ⟹ const S
by(simp add: const-def, insert assms, blast)

lemma const-lam : const (λ-. e)
by(simp add: const-def)


lemma const-true[simp] : const true
by(simp add: const-def true-def)
```

```
lemma const-false[simp] : const false
by(simp add: const-def false-def)

lemma const-null[simp] : const null
by(simp add: const-def null-fun-def)

lemma const-invalid [simp]: const invalid
by(simp add: const-def invalid-def)

lemma const-bot[simp] : const bot
by(simp add: const-def bot-fun-def)



lemma const-defined :
 assumes const X
 shows   const (δ X)
by(rule const-imply2[OF - assms],
   simp add: defined-def false-def true-def bot-fun-def bot-option-def null-fun-def null-option-def)

lemma const-valid :
 assumes const X
 shows   const (υ X)
by(rule const-imply2[OF - assms],
   simp add: valid-def false-def true-def bot-fun-def null-fun-def assms)


lemma const-OclAnd :
  assumes const X
  assumes const X′
  shows   const (X and X′)
by(rule const-imply3[OF - assms], subst (1 2) cp-OclAnd, simp add: assms OclAnd-def)


lemma const-OclNot :
    assumes const X
    shows   const (not X)
by(rule const-imply2[OF - assms],subst cp-OclNot,simp add: assms OclNot-def)

lemma const-OclOr :
  assumes const X
  assumes const X′
  shows   const (X or X′)
by(simp add: assms OclOr-def const-OclNot const-OclAnd)

lemma const-OclImplies :
  assumes const X
  assumes const X′
  shows   const (X implies X′)
by(simp add: assms OclImplies-def const-OclNot const-OclOr)

lemma const-StrongEq:
  assumes const X
  assumes const X′
  shows   const(X ≜ X′)
  apply(simp only: StrongEq-def const-def, intro allI)
  apply(subst assms(1)[THEN const-charn])
  apply(subst assms(2)[THEN const-charn])
  by simp


lemma const-OclIf :
  assumes const B
      and const C1
      and const C2
    shows const (if B then C1 else C2 endif)
 apply(rule const-imply4[OF - assms],
       subst (1 2) cp-OclIf, simp only: OclIf-def cp-defined[symmetric])
 apply(simp add: const-defined[OF assms(1), simplified const-def, THEN spec, THEN spec]
              const-true[simplified const-def, THEN spec, THEN spec]
              assms[simplified const-def, THEN spec, THEN spec]
              const-invalid[simplified const-def, THEN spec, THEN spec])
by (metis (no-types) bot-fun-def OclValid-def const-def const-true defined-def
              foundation16[THEN iffD1,standard]  null-fun-def)



lemma const-OclValid1:
 assumes const x
 shows   (τ ⊨ δ x) = (τ′ ⊨ δ x)
 apply(simp add: OclValid-def)
 apply(subst const-defined[OF assms, THEN const-charn])
 by(simp add: true-def)

lemma const-OclValid2:
 assumes const x
 shows   (τ ⊨ υ x) = (τ′ ⊨ υ x)
 apply(simp add: OclValid-def)
 apply(subst const-valid[OF assms, THEN const-charn])
 by(simp add: true-def)


lemma const-HOL-if : const C ⟹ const D ⟹ const F ⟹ const (λτ. if C τ then D τ else F τ)
      by(auto simp: const-def)
lemma const-HOL-and: const C ⟹ const D ⟹ const (λτ. C τ ∧ D τ)
```

```
  by(auto simp: const-def)
lemma const-HOL-eq : const C ⟹ const D ⟹ const (λ τ. C τ = D τ)
  apply(auto simp: const-def)
  apply(erule-tac x=τ in allE)
  apply(erule-tac x=τ in allE)
  apply(erule-tac x=τ' in allE)
  apply(erule-tac x=τ' in allE)
  apply simp
  apply(erule-tac x=τ in allE)
  apply(erule-tac x=τ in allE)
  apply(erule-tac x=τ' in allE)
  apply(erule-tac x=τ' in allE)
  by simp


lemmas const-ss = const-bot const-null  const-invalid  const-false  const-true  const-lam
            const-defined const-valid const-StrongEq const-OclNot const-OclAnd
            const-OclOr const-OclImplies const-OclIf
            const-HOL-if const-HOL-and const-HOL-eq
```

Miscellaneous: Overloading the syntax of “bottom”

```
notation bot (⊥)

end
```

```
theory UML-PropertyProfiles
imports  UML-Logic
begin
```

B.2.2. Property Profiles for OCL Operators via Isabelle Locales

We use the Isabelle mechanism of a *Locale* to generate the common lemmas for each type and operator; Locales can be seen as a functor that takes a local theory and generates a number of theorems. In our case, we will instantiate later these locales by the local theory of an operator definition and obtain the common rules for strictness, definedness propagation, context-passingness and constance in a systematic way.

Property Profiles for Monadic Operators

```
locale profile-mono-scheme =
  fixes f :: ('𝔄,'α::null)val ⇒ ('𝔄,'β::null)val
  fixes g
  assumes def-scheme: (f x) ≡ λ τ. if (δ x) τ = true τ then g (x τ) else invalid τ

locale profile-mono2 = profile-mono-scheme +
  assumes ⋀ x. x ≠ bot ⟹ x ≠ null ⟹ g x ≠ bot
begin
  lemma strict[simp,code-unfold]: f invalid = invalid
  by(rule ext, simp add: def-scheme true-def false-def)

  lemma null-strict[simp,code-unfold]: f null = invalid
  by(rule ext, simp add: def-scheme true-def false-def)

  lemma cp0 : f X τ = f (λ -. X τ) τ
  by(simp add: def-scheme  cp-defined[symmetric])

  lemma cp[simp,code-unfold] :  cp P ⟹ cp (λX. f (P X) )
  by(rule cpI1[of f], intro allI, rule cp0, simp-all)

  lemma const[simp,code-unfold] :
        assumes C1 :const X
        shows      const(f X)
     proof −
       have const-g : const (λτ. g (X τ))  by(insert C1, auto simp:const-def, metis)
       show ?thesis   by(simp-all add : def-scheme const-ss C1 const-g)
     qed
end

locale profile-mono0 = profile-mono-scheme +
  assumes def-body:  ⋀ x. x ≠ bot ⟹ x ≠ null ⟹ g x ≠ bot ∧ g x ≠ null

sublocale profile-mono0 < profile-mono2
by(unfold-locales, simp add: def-scheme, simp add: def-body)

context profile-mono0
begin
  lemma def-homo[simp,code-unfold]: δ(f x) = (δ x)
  apply(rule ext, rename-tac τ,subst foundation22[symmetric])
  apply(case-tac ¬(τ ⊨ δ x), simp add:defined-split, elim disjE)
    apply(erule StrongEq-L-subst2-rev, simp,simp)
   apply(erule StrongEq-L-subst2-rev, simp,simp)
  apply(simp)
  apply(rule foundation13[THEN iffD2,THEN StrongEq-L-subst2-rev, where y =δ x])
    apply(simp-all add:def-scheme)
  apply(simp add: OclValid-def)
  by(auto simp:foundation13 StrongEq-def false-def true-def defined-def bot-fun-def null-fun-def def-body
         split: split-if-asm)

  lemma def-valid-then-def: υ(f x) = (δ(f x))
  apply(rule ext, rename-tac τ,subst foundation22[symmetric])
  apply(case-tac ¬(τ ⊨ δ x), simp add:defined-split, elim disjE)
   apply(erule StrongEq-L-subst2-rev, simp,simp)
  apply(erule StrongEq-L-subst2-rev, simp,simp)
  apply simp
  apply(simp-all add:def-scheme)
  apply(simp add: OclValid-def valid-def, subst cp-StrongEq)
  apply(subst (2) cp-defined, simp, simp add: cp-defined[symmetric])
  by(auto simp:foundation13 StrongEq-def false-def true-def defined-def bot-fun-def null-fun-def def-body
        split: split-if-asm)
end
```

Property Profiles for Single

```
locale profile-single =
  fixes d:: ('𝔄,'a::null)val ⇒ '𝔄 Boolean
  assumes d-strict[simp,code-unfold]: d invalid = false
  assumes d-cp0: d X τ = d (λ -. X τ) τ
  assumes d-const[simp,code-unfold]: const X ⟹ const (d X)
```

Property Profiles for Binary Operators

```
definition bin' f g dₓ d_y X Y =
                    (f X Y = (λ τ. if (dₓ X) τ = true τ ∧ (d_y Y) τ = true τ
                                   then g X Y τ
                                   else invalid τ ))

definition bin f g = bin' f (λX Y τ. g (X τ) (Y τ))

lemmas [simp,code-unfold] = bin'-def bin-def

locale profile-bin-scheme =
  fixes dₓ:: ('𝔄,'a::null)val ⇒ '𝔄 Boolean
  fixes d_y:: ('𝔄,'b::null)val ⇒ '𝔄 Boolean
  fixes f::('𝔄,'a::null)val ⇒ ('𝔄,'b::null)val ⇒ ('𝔄,'c::null)val
  fixes g
  assumes dₓ' : profile-single dₓ
  assumes d_y' : profile-single d_y
  assumes dₓ-d_y-homo[simp,code-unfold]: cp (f X) ⟹
                      cp (λx. f x Y) ⟹
                      f X invalid = invalid ⟹
                      f invalid Y = invalid ⟹
                      (¬ (τ ⊨ dₓ X) ∨ ¬ (τ ⊨ d_y Y)) ⟹
                      τ ⊨ (δ f X Y ≜ (dₓ X and d_y Y))
  assumes def-scheme''[simplified]: bin f g dₓ d_y X Y
  assumes 1: τ ⊨ dₓ X ⟹ τ ⊨ d_y Y ⟹ τ ⊨ δ f X Y
begin
    interpretation dₓ : profile-single dₓ by (rule dₓ')
    interpretation d_y : profile-single d_y by (rule d_y')
    lemma strict1[simp,code-unfold]: f invalid y = invalid
    by(rule ext, simp add: def-scheme'' true-def false-def)

    lemma strict2[simp,code-unfold]: f x invalid = invalid
    by(rule ext, simp add: def-scheme'' true-def false-def)

    lemma cp0 : f X Y τ = f (λ -. X τ) (λ -. Y τ) τ
    by(simp add: def-scheme'' d_x.d-cp0[symmetric] d_y.d-cp0[symmetric] cp-defined[symmetric])

    lemma cp[simp,code-unfold] : cp P ⟹ cp Q ⟹ cp (λX. f (P X) (Q X))
    by(rule cpI2[of f], intro allI, rule cp0, simp-all)

    lemma def-homo[simp,code-unfold]: δ(f x y) = (d_x x and d_y y)
      apply(rule ext, rename-tac τ,subst foundation22[symmetric])
      apply(case-tac ¬(τ ⊨ d_x x), simp)
      apply(case-tac ¬(τ ⊨ d_y y), simp)
      apply(simp)
      apply(rule foundation13[THEN iffD2,THEN StrongEq-L-subst2-rev, where y =d_x x])
       apply(simp-all)
      apply(rule foundation13[THEN iffD2,THEN StrongEq-L-subst2-rev, where y =d_y y])
       apply(simp-all add: 1 foundation13)
      done

    lemma def-valid-then-def: υ(f x y) = (δ(f x y))
      apply(rule ext, rename-tac τ)
      apply(simp-all add: valid-def defined-def def-scheme''
                    true-def false-def invalid-def
                    null-def null-fun-def null-option-def bot-fun-def)
      by (metis 1 OclValid-def def-scheme'' foundation16 true-def)

    lemma defined-args-valid: (τ ⊨ δ (f x y)) = ((τ ⊨ d_x x) ∧ (τ ⊨ d_y y))
      by(simp add: foundation27)

    lemma const[simp,code-unfold] :
       assumes C1 :const X and C2 : const Y
       shows      const(f X Y)
    proof –
       have const-g : const (λτ. g (X τ) (Y τ))
             by(insert C1 C2, auto simp:const-def, metis)
      show ?thesis
      by(simp-all add : def-scheme'' const-ss C1 C2 const-g)
    qed
end
```

In our context, we will use Locales as "Property Profiles" for OCL operators; if an operator *f* is of profile *profile-bin-scheme defined f g* we know that it satisfies a number of properties like *strict1* or *strict2* i. e. *f invalid y* = *invalid* and *f null y* = *invalid*. Since some of the more advanced Locales come with 10 - 15 theorems, property profiles represent a major structuring mechanism for the OCL library.

```
locale profile-bin-scheme-defined =
  fixes d_y:: ('𝔄,'b::null)val ⇒ '𝔄 Boolean
  fixes f::('𝔄,'a::null)val ⇒ ('𝔄,'b::null)val ⇒ ('𝔄,'c::null)val
  fixes g
  assumes d_y : profile-single d_y
  assumes d_y-homo[simp,code-unfold]: cp (f X) ⟹
                    f X invalid = invalid ⟹
                    ¬ τ ⊨ d_y Y ⟹
                    τ ⊨ δ f X Y ≜ (δ X and d_y Y)
  assumes def-scheme'[simplified]: bin f g defined d_y X Y
  assumes def-body': ⋀ x y τ. x≠bot ⟹ x≠null ⟹ (d_y y) τ = true τ ⟹ g x (y τ) ≠ bot ∧ g x (y τ) ≠ null
begin
    lemma strict3[simp,code-unfold]: f null y = invalid
    by(rule ext, simp add: def-scheme' true-def false-def)
end

sublocale profile-bin-scheme-defined < profile-bin-scheme defined
proof −
      interpret d_y : profile-single d_y by (rule d_y)
 show profile-bin-scheme defined d_y f g
 apply(unfold-locales)
      apply(simp)+
     apply(subst cp-defined, simp)
    apply(rule const-defined, simp)
   apply(simp add:defined-split, elim disjE)
     apply(erule StrongEq-L-subst2-rev, simp, simp)+
   apply(simp)
  apply(simp add: def-scheme')
 apply(simp add: defined-def OclValid-def false-def true-def
             bot-fun-def null-fun-def def-scheme' split: split-if-asm, rule def-body')
 by(simp add: true-def)+
qed

locale profile-bin1 =
  fixes f::('𝔄,'a::null)val ⇒ ('𝔄,'b::null)val ⇒ ('𝔄,'c::null)val
  fixes g
  assumes def-scheme[simplified]: bin f g defined defined X Y
  assumes def-body: ⋀ x y. g x y ≠ bot ∧ g x y ≠ null
begin
    lemma strict4[simp,code-unfold]: f x null = invalid
    by(rule ext, simp add: def-scheme true-def false-def)
end

sublocale profile-bin1 < profile-bin-scheme-defined defined
 apply(unfold-locales)
      apply(simp)+
    apply(subst cp-defined, simp)+
   apply(rule const-defined, simp)+
  apply(simp add:defined-split, elim disjE)
   apply(erule StrongEq-L-subst2-rev, simp, simp)+
 apply(simp add: def-scheme)
by(simp add: defined-def OclValid-def false-def true-def
          bot-fun-def null-fun-def def-scheme def-body)

locale profile-bin2 =
  fixes f::('𝔄,'a::null)val ⇒ ('𝔄,'b::null)val ⇒ ('𝔄,'c::null)val
  fixes g
  assumes def-scheme[simplified]: bin f g defined valid X Y
  assumes def-body:  ⋀ x y. x≠bot ⟹ x≠null ⟹ y≠bot ⟹ g x y ≠ bot ∧ g x y ≠ null

sublocale profile-bin2 < profile-bin-scheme-defined valid
 apply(unfold-locales)
     apply(simp)
    apply(subst cp-valid, simp)
   apply(rule const-valid, simp)
  apply(simp add:foundation18′′)
  apply(erule StrongEq-L-subst2-rev, simp, simp)
 apply(simp add: def-scheme)
by (metis OclValid-def def-body foundation18′)

locale profile-bin3 =
  fixes f :: ('𝔄,'α::null)val ⇒ ('𝔄,'α::null)val ⇒ ('𝔄) Boolean
  assumes def-scheme[simplified]: bin′ f StrongEq valid valid X Y

sublocale profile-bin3 < profile-bin-scheme valid valid f λx y. ⌊⌊x = y⌋⌋
 apply(unfold-locales)
     apply(simp)
    apply(subst cp-valid, simp)
   apply (simp add: const-valid)
   apply (metis (hide-lams, mono-tags) OclValid-def def-scheme defined5 defined6 defined-and-I foundation1 foundation10′ foundation16′ foundation18 foundation21 foundation22 foundation9)
 apply(simp add: def-scheme, subst StrongEq-def, simp)
by (metis OclValid-def def-scheme defined7 foundation16)

context profile-bin3
  begin
    lemma idem[simp,code-unfold]:  f null null = true
    by(rule ext, simp add: def-scheme true-def false-def)


    lemma defargs: τ ⊨ f x y ⟹ (τ ⊨ υ x) ∧ (τ ⊨ υ y)
      by(simp add: def-scheme OclValid-def true-def invalid-def valid-def bot-option-def
           split: bool.split-asm HOL.split-if-asm)
    lemma defined-args-valid' : δ (f x y) = (υ x and υ y)
    by(auto intro!: transform2-rev defined-and-I simp:foundation10 defined-args-valid)


    lemma refl-ext[simp,code-unfold] : (f x x) = (if (υ x) then true else invalid endif)
      by(rule ext, simp add: def-scheme OclIf-def)

    lemma sym : τ ⊨ (f x y) ⟹ τ ⊨ (f y x)
      apply(case-tac τ ⊨ υ x)
       apply(auto simp: def-scheme OclValid-def)
      by(fold OclValid-def, erule StrongEq-L-sym)

    lemma symmetric : (f x y) = (f y x)
      by(rule ext, rename-tac τ, auto simp: def-scheme StrongEq-sym)

    lemma trans : τ ⊨ (f x y) ⟹ τ ⊨ (f y z) ⟹ τ ⊨ (f x z)
      apply(case-tac τ ⊨ υ x)
       apply(case-tac τ ⊨ υ y)
        apply(auto simp: def-scheme OclValid-def)
      by(fold OclValid-def, auto elim: StrongEq-L-trans)

    lemma StrictRefEq-vs-StrongEq: τ ⊨(υ x) ⟹ τ ⊨(υ y) ⟹ (τ ⊨ ((f x y) ≜ (x ≜ y)))
      apply(simp add: def-scheme OclValid-def)
      apply(subst cp-StrongEq[of - (x ≜ y)])
      by simp

  end


locale profile-bin4 =
  fixes f :: ('𝔄,'α::null)val ⇒ ('𝔄,'β::null)val ⇒ ('𝔄,'γ::null)val
  fixes g
  assumes def-scheme[simplified]: bin f g valid valid X Y
  assumes def-body:  ⋀ x y. x≠bot ⟹ y≠bot ⟹ g x y ≠ bot ∧ g x y ≠ null

sublocale profile-bin4 < profile-bin-scheme valid valid
 apply(unfold-locales)
      apply(simp, subst cp-valid, simp, rule const-valid, simp)+
  apply (metis (hide-lams, mono-tags) OclValid-def def-scheme defined5 defined6 defined-and-I
      foundation1 foundation10' foundation16' foundation18 foundation21 foundation22 foundation9)
  apply(simp add: def-scheme)
 apply(simp add: defined-def OclValid-def false-def true-def
          bot-fun-def null-fun-def def-scheme split: split-if-asm, rule def-body)
 by (metis OclValid-def foundation18' true-def)+

end
```

theory *UML-Boolean*
imports *../UML-PropertyProfiles*
begin

Fundamental Predicates on Basic Types: Strict (Referential) Equality

Here is a first instance of a definition of strict value equality—for the special case of the type $'\mathfrak{A}$ *Boolean*, it is just the strict extension of the logical equality:

defs $\textit{StrictRefEq}_{\textit{Boolean}}[\textit{code-unfold}]$:
  $(x{::}('\mathfrak{A})\textit{Boolean}) \doteq y \equiv \lambda\ \tau.\ \textit{if}\ (\upsilon\ x)\ \tau = \textit{true}\ \tau \land (\upsilon\ y)\ \tau = \textit{true}\ \tau$
    $\textit{then}\ (x \triangleq y)\tau$
    $\textit{else}\ \textit{invalid}\ \tau$

which implies elementary properties like:

lemma [*simp*,*code-unfold*] : $(\textit{true} \doteq \textit{false}) = \textit{false}$
by(*simp add*:$\textit{StrictRefEq}_{\textit{Boolean}}$)
lemma [*simp*,*code-unfold*] : $(\textit{false} \doteq \textit{true}) = \textit{false}$
by(*simp add*:$\textit{StrictRefEq}_{\textit{Boolean}}$)

lemma *null-non-false* [*simp*,*code-unfold*]:$(\textit{null} \doteq \textit{false}) = \textit{false}$
 apply(*rule ext*, *simp add*: $\textit{StrictRefEq}_{\textit{Boolean}}$ *StrongEq-def false-def*)
 by (*metis drop.simps cp-valid false-def is-none-code*(2) *is-none-def valid4*
  bot-option-def null-fun-def null-option-def)

lemma *null-non-true* [*simp*,*code-unfold*]:$(\textit{null} \doteq \textit{true}) = \textit{false}$
 apply(*rule ext*, *simp add*: $\textit{StrictRefEq}_{\textit{Boolean}}$ *StrongEq-def false-def*)
 by(*simp add*: *true-def bot-option-def null-fun-def null-option-def*)

lemma *false-non-null* [*simp*,*code-unfold*]:$(\textit{false} \doteq \textit{null}) = \textit{false}$
 apply(*rule ext*, *simp add*: $\textit{StrictRefEq}_{\textit{Boolean}}$ *StrongEq-def false-def*)
 by(*metis drop.simps cp-valid false-def is-none-code*(2) *is-none-def valid4*
  bot-option-def null-fun-def null-option-def)

lemma *true-non-null* [*simp*,*code-unfold*]:$(\textit{true} \doteq \textit{null}) = \textit{false}$
 apply(*rule ext*, *simp add*: $\textit{StrictRefEq}_{\textit{Boolean}}$ *StrongEq-def false-def*)
 by(*simp add*: *true-def bot-option-def null-fun-def null-option-def*)

With respect to strictness properties and miscelleaneous side-calculi, strict referential equality behaves on booleans as described in the *profile-bin3*:

interpretation $\textit{StrictRefEq}_{\textit{Boolean}}$: *profile-bin3* $\lambda\ x\ y.\ (x{::}('\mathfrak{A})\textit{Boolean}) \doteq y$
  by *unfold-locales* (*auto simp*:$\textit{StrictRefEq}_{\textit{Boolean}}$)

In particular, it is strict, cp-preserving and const-preserving. In particular, it generates the simplifier rules for terms like:

lemma $(\textit{invalid} \doteq \textit{false}) = \textit{invalid}$ **by**(*simp*)

lemma $(\mathit{invalid} \doteq \mathit{true}) = \mathit{invalid}$ **by**(simp)

lemma $(\mathit{false} \doteq \mathit{invalid}) = \mathit{invalid}$ **by**(simp)

lemma $(\mathit{true} \doteq \mathit{invalid}) = \mathit{invalid}$ **by**(simp)

lemma $((\mathit{invalid}{::}(\mathfrak{'A})\mathit{Boolean}) \doteq \mathit{invalid}) = \mathit{invalid}$ **by**(simp)

Thus, the weak equality is *not* reflexive.

Test Statements on Boolean Operations.

Here follows a list of code-examples, that explain the meanings of the above definitions by compilation to code and execution to *True*.

Elementary computations on Boolean

Assert $\tau \models \upsilon(\mathit{true})$

Assert $\tau \models \delta(\mathit{false})$

Assert $\neg(\tau \models \delta(\mathit{null}))$

Assert $\neg(\tau \models \delta(\mathit{invalid}))$

Assert $\tau \models \upsilon((\mathit{null}{::}(\mathfrak{'A})\mathit{Boolean}))$

Assert $\neg(\tau \models \upsilon(\mathit{invalid}))$

Assert $\tau \models (\mathit{true}\ \mathit{and}\ \mathit{true})$

Assert $\tau \models (\mathit{true}\ \mathit{and}\ \mathit{true} \triangleq \mathit{true})$

Assert $\tau \models ((\mathit{null}\ \mathit{or}\ \mathit{null}) \triangleq \mathit{null})$

Assert $\tau \models ((\mathit{null}\ \mathit{or}\ \mathit{null}) \doteq \mathit{null})$

Assert $\tau \models ((\mathit{true} \triangleq \mathit{false}) \triangleq \mathit{false})$

Assert $\tau \models ((\mathit{invalid} \triangleq \mathit{false}) \triangleq \mathit{false})$

Assert $\tau \models ((\mathit{invalid} \doteq \mathit{false}) \triangleq \mathit{invalid})$

Assert $\tau \models (\mathit{true} <> \mathit{false})$

Assert $\tau \models (\mathit{false} <> \mathit{true})$

end

theory *UML-Void*
imports *../UML-PropertyProfiles*
begin

B.2.3. Basic Type Void

This *minimal* OCL type contains only two elements: *invalid* and *null*. *Void* could initially be defined as *unit option option*, however the cardinal of this type is more than two, so it would have the cost to consider *Some None* and *Some* (*Some* ()) seemingly everywhere.

Fundamental Properties on Basic Types: Strict Equality

Definition **instantiation** $\mathit{Void}_{\mathit{base}}$ :: *bot*

begin

  definition *bot-Void-def*: $(\mathit{bot\text{-}class.bot} :: \mathit{Void}_{base}) \equiv \mathit{Abs\text{-}Void}_{base}\ \mathit{None}$

  instance proof show $\exists x{::}\ \mathit{Void}_{base}.\ x \neq \mathit{bot}$

    apply$(\mathit{rule\text{-}tac}\ x{=}\mathit{Abs\text{-}Void}_{base}\ \lfloor \mathit{None} \rfloor$ **in** $\mathit{exI})$

    apply$(\mathit{simp\ add{:}bot\text{-}Void\text{-}def},\ \mathit{subst}\ \mathit{Abs\text{-}Void}_{base}\mathit{\text{-}inject})$

    apply$(\mathit{simp\text{-}all\ add}{:}\ \mathit{null\text{-}option\text{-}def}\ \mathit{bot\text{-}option\text{-}def})$

    done

   qed

end

instantiation $\mathit{Void}_{base} :: \mathit{null}$

begin

  definition *null-Void-def*: $(\mathit{null}{::}\mathit{Void}_{base}) \equiv \mathit{Abs\text{-}Void}_{base}\ \lfloor\ \mathit{None}\ \rfloor$

  instance proof show $(\mathit{null}{::}\ \mathit{Void}_{base}) \neq \mathit{bot}$

    apply$(\mathit{simp\ add{:}null\text{-}Void\text{-}def}\ \mathit{bot\text{-}Void\text{-}def},\ \mathit{subst}\ \mathit{Abs\text{-}Void}_{base}\mathit{\text{-}inject})$

    apply$(\mathit{simp\text{-}all\ add}{:}\ \mathit{null\text{-}option\text{-}def}\ \mathit{bot\text{-}option\text{-}def})$

    done

   qed

end

The last basic operation belonging to the fundamental infrastructure of a value-type in OCL is the weak equality, which is defined similar to the $'\mathfrak{A}$ *Void*-case as strict extension of the strong equality:

defs $\mathit{StrictRefEq}_{Void}[\mathit{code\text{-}unfold}]$:

$$(x{::}('\mathfrak{A})\mathit{Void}) \doteq y \equiv \lambda\ \tau.\ \mathit{if}\ (\upsilon\ x)\ \tau = \mathit{true}\ \tau \wedge (\upsilon\ y)\ \tau = \mathit{true}\ \tau$$
$$\mathit{then}\ (x \triangleq y)\ \tau$$
$$\mathit{else}\ \mathit{invalid}\ \tau$$

Property proof in terms of *profile-bin3*

interpretation $\mathit{StrictRefEq}_{Void} : \mathit{profile\text{-}bin3}\ \lambda\ x\ y.\ (x{::}('\mathfrak{A})\mathit{Void}) \doteq y$

  by $\mathit{unfold\text{-}locales}\ (\mathit{auto\ simp}{:}\ \ \mathit{StrictRefEq}_{Void})$

Test Statements

Assert $\tau \models ((\mathit{null}{::}('\mathfrak{A})\mathit{Void})\ \doteq \mathit{null})$

end

theory *UML-Integer*

imports *../UML-PropertyProfiles*

begin

B.2.4. Basic Type Integer: Operations

Basic Integer Constants

Although the remaining part of this library reasons about integers abstractly, we provide here as example some convenient shortcuts.

definition *OclInt0* ::($'\mathfrak{A}$)*Integer* (**0**)
where **0** = (λ - . ⌊⌊*0*::*int*⌋⌋)

definition *OclInt1* ::($'\mathfrak{A}$)*Integer* (**1**)
where **1** = (λ - . ⌊⌊*1*::*int*⌋⌋)

definition *OclInt2* ::($'\mathfrak{A}$)*Integer* (**2**)
where **2** = (λ - . ⌊⌊*2*::*int*⌋⌋)

definition *OclInt3* ::($'\mathfrak{A}$)*Integer* (**3**)
where **3** = (λ - . ⌊⌊*3*::*int*⌋⌋)

definition *OclInt4* ::($'\mathfrak{A}$)*Integer* (**4**)
where **4** = (λ - . ⌊⌊*4*::*int*⌋⌋)

definition *OclInt5* ::($'\mathfrak{A}$)*Integer* (**5**)
where **5** = (λ - . ⌊⌊*5*::*int*⌋⌋)

definition *OclInt6* ::($'\mathfrak{A}$)*Integer* (**6**)
where **6** = (λ - . ⌊⌊*6*::*int*⌋⌋)

definition *OclInt7* ::($'\mathfrak{A}$)*Integer* (**7**)
where **7** = (λ - . ⌊⌊*7*::*int*⌋⌋)

definition *OclInt8* ::($'\mathfrak{A}$)*Integer* (**8**)
where **8** = (λ - . ⌊⌊*8*::*int*⌋⌋)

definition *OclInt9* ::($'\mathfrak{A}$)*Integer* (**9**)
where **9** = (λ - . ⌊⌊*9*::*int*⌋⌋)

definition *OclInt10* ::($'\mathfrak{A}$)*Integer* (**10**)
where **10** = (λ - . ⌊⌊*10*::*int*⌋⌋)

Validity and Definedness Properties

lemma δ(*null*::($'\mathfrak{A}$)*Integer*) = *false* **by** *simp*
lemma υ(*null*::($'\mathfrak{A}$)*Integer*) = *true* **by** *simp*

lemma [*simp*,*code-unfold*]: δ (λ-. ⌊⌊*n*⌋⌋) = *true*
by(*simp add*:*defined-def true-def*
bot-fun-def bot-option-def null-fun-def null-option-def)

lemma [*simp*,*code-unfold*]: $\upsilon\ (\lambda\text{-}.\ \lfloor\lfloor n\rfloor\rfloor) = \mathit{true}$
by(*simp add*:*valid-def true-def*
    bot-fun-def bot-option-def)

lemma [*simp*,*code-unfold*]: $\delta\ \mathbf{0} = \mathit{true}$ **by**(*simp add*:*OclInt0-def*)
lemma [*simp*,*code-unfold*]: $\upsilon\ \mathbf{0} = \mathit{true}$ **by**(*simp add*:*OclInt0-def*)
lemma [*simp*,*code-unfold*]: $\delta\ \mathbf{1} = \mathit{true}$ **by**(*simp add*:*OclInt1-def*)
lemma [*simp*,*code-unfold*]: $\upsilon\ \mathbf{1} = \mathit{true}$ **by**(*simp add*:*OclInt1-def*)
lemma [*simp*,*code-unfold*]: $\delta\ \mathbf{2} = \mathit{true}$ **by**(*simp add*:*OclInt2-def*)
lemma [*simp*,*code-unfold*]: $\upsilon\ \mathbf{2} = \mathit{true}$ **by**(*simp add*:*OclInt2-def*)
lemma [*simp*,*code-unfold*]: $\delta\ \mathbf{6} = \mathit{true}$ **by**(*simp add*:*OclInt6-def*)
lemma [*simp*,*code-unfold*]: $\upsilon\ \mathbf{6} = \mathit{true}$ **by**(*simp add*:*OclInt6-def*)
lemma [*simp*,*code-unfold*]: $\delta\ \mathbf{8} = \mathit{true}$ **by**(*simp add*:*OclInt8-def*)
lemma [*simp*,*code-unfold*]: $\upsilon\ \mathbf{8} = \mathit{true}$ **by**(*simp add*:*OclInt8-def*)
lemma [*simp*,*code-unfold*]: $\delta\ \mathbf{9} = \mathit{true}$ **by**(*simp add*:*OclInt9-def*)
lemma [*simp*,*code-unfold*]: $\upsilon\ \mathbf{9} = \mathit{true}$ **by**(*simp add*:*OclInt9-def*)

Arithmetical Operations

Definition 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).

Note that we can not follow the lexis of the OCL Standard for Isabelle technical reasons; these operators are heavily overloaded in the HOL library that a further overloading would lead to heavy technical buzz in this document.

definition $\mathit{OclAdd}_{\mathit{Integer}}$ ::$('\mathfrak{A})\mathit{Integer} \Rightarrow ('\mathfrak{A})\mathit{Integer} \Rightarrow ('\mathfrak{A})\mathit{Integer}$ (**infix** $+_{\mathit{int}}$ *40*)
where $x +_{\mathit{int}} y \equiv \lambda\ \tau.\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau \wedge (\delta\ y)\ \tau = \mathit{true}\ \tau$
    $\mathit{then}\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil + \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
    $\mathit{else}\ \mathit{invalid}\ \tau$
interpretation $\mathit{OclAdd}_{\mathit{Integer}}$: *profile-bin1 op* $+_{\mathit{int}}$ $\lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil + \lceil\lceil y\rceil\rceil\rfloor\rfloor$
  by *unfold-locales* (*auto simp*:$\mathit{OclAdd}_{\mathit{Integer}}$*-def bot-option-def null-option-def*)

definition $\mathit{OclMinus}_{\mathit{Integer}}$ ::$('\mathfrak{A})\mathit{Integer} \Rightarrow ('\mathfrak{A})\mathit{Integer} \Rightarrow ('\mathfrak{A})\mathit{Integer}$ (**infix** $-_{\mathit{int}}$ *41*)
where $x -_{\mathit{int}} y \equiv \lambda\ \tau.\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau \wedge (\delta\ y)\ \tau = \mathit{true}\ \tau$
    $\mathit{then}\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil - \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
    $\mathit{else}\ \mathit{invalid}\ \tau$
interpretation $\mathit{OclMinus}_{\mathit{Integer}}$: *profile-bin1 op* $-_{\mathit{int}}$ $\lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil - \lceil\lceil y\rceil\rceil\rfloor\rfloor$
  by *unfold-locales* (*auto simp*:$\mathit{OclMinus}_{\mathit{Integer}}$*-def bot-option-def null-option-def*)

definition $\mathit{OclMult}_{\mathit{Integer}}$ ::$('\mathfrak{A})\mathit{Integer} \Rightarrow ('\mathfrak{A})\mathit{Integer} \Rightarrow ('\mathfrak{A})\mathit{Integer}$ (**infix** $*_{\mathit{int}}$ *45*)
where $x *_{\mathit{int}} y \equiv \lambda\ \tau.\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau \wedge (\delta\ y)\ \tau = \mathit{true}\ \tau$
    $\mathit{then}\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil * \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
    $\mathit{else}\ \mathit{invalid}\ \tau$
interpretation $\mathit{OclMult}_{\mathit{Integer}}$: *profile-bin1 op* $*_{\mathit{int}}$ $\lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil * \lceil\lceil y\rceil\rceil\rfloor\rfloor$
  by *unfold-locales* (*auto simp*:$\mathit{OclMult}_{\mathit{Integer}}$*-def bot-option-def null-option-def*)

Here is the special case of division, which is defined as invalid for division by zero.

definition $OclDivision_{Integer}$ ::$({}'\mathfrak{A})Integer \Rightarrow ({}'\mathfrak{A})Integer \Rightarrow ({}'\mathfrak{A})Integer$ (**infix** div_{int} *45*)
where $x\ div_{int}\ y \equiv \lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$\quad then\ if\ y\ \tau \neq OclInt0\ \tau\ then\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil\ div\ \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor\ else\ invalid\ \tau$
$\quad else\ invalid\ \tau$

definition $OclModulus_{Integer}$ ::$({}'\mathfrak{A})Integer \Rightarrow ({}'\mathfrak{A})Integer \Rightarrow ({}'\mathfrak{A})Integer$ (**infix** mod_{int} *45*)
where $x\ mod_{int}\ y \equiv \lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$\quad then\ if\ y\ \tau \neq OclInt0\ \tau\ then\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil\ mod\ \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor\ else\ invalid\ \tau$
$\quad else\ invalid\ \tau$

definition $OclLess_{Integer}$ ::$({}'\mathfrak{A})Integer \Rightarrow ({}'\mathfrak{A})Integer \Rightarrow ({}'\mathfrak{A})Boolean$ (**infix** $<_{int}$ *35*)
where $x <_{int} y \equiv \lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$\quad then\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil < \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\quad else\ invalid\ \tau$
interpretation $OclLess_{Integer}$: *profile-bin1 op* $<_{int}$ $\lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil < \lceil\lceil y\rceil\rceil\rfloor\rfloor$
by *unfold-locales* (*auto simp*:$OclLess_{Integer}$*-def bot-option-def null-option-def*)

definition $OclLe_{Integer}$ ::$({}'\mathfrak{A})Integer \Rightarrow ({}'\mathfrak{A})Integer \Rightarrow ({}'\mathfrak{A})Boolean$ (**infix** $\leq_{int}$ *35*)
where $x \leq_{int} y \equiv \lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$\quad then\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil \leq \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\quad else\ invalid\ \tau$
interpretation $OclLe_{Integer}$: *profile-bin1 op* $\leq_{int}$ $\lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil \leq \lceil\lceil y\rceil\rceil\rfloor\rfloor$
by *unfold-locales* (*auto simp*:$OclLe_{Integer}$*-def bot-option-def null-option-def*)

Basic Properties **lemma** $OclAdd_{Integer}$*-commute*: $(X +_{int} Y) = (Y +_{int} X)$
by(*rule ext*,*auto simp*:*true-def false-def* $OclAdd_{Integer}$*-def invalid-def*
split: *option.split option.split-asm*
bool.split bool.split-asm)

Execution with Invalid or Null or Zero as Argument **lemma** $OclAdd_{Integer}$*-zero1*[*simp*,*code-unfold*] :
$(x +_{int} \mathbf{0}) = (if\ \upsilon\ x\ and\ not\ (\delta\ x)\ then\ invalid\ else\ x\ endif)$
proof (*rule ext*, *rename-tac* τ, *case-tac* $(\upsilon\ x\ and\ not\ (\delta\ x))\ \tau = true\ \tau$)
fix τ **show** $(\upsilon\ x\ and\ not\ (\delta\ x))\ \tau = true\ \tau \Longrightarrow$
$\quad (x +_{int} \mathbf{0})\ \tau = (if\ \upsilon\ x\ and\ not\ (\delta\ x)\ then\ invalid\ else\ x\ endif)\ \tau$
apply(*subst OclIf-true'*, *simp add*: *OclValid-def*)
by (*metis* $OclAdd_{Integer}$*-def OclNot-defargs OclValid-def foundation5 foundation9*)
apply-end *assumption*
next fix τ
have A: $\bigwedge\tau.\ (\tau \models not\ (\upsilon\ x\ and\ not\ (\delta\ x))) = (x\ \tau = invalid\ \tau \vee \tau \models \delta\ x)$
by (*metis OclNot-not OclOr-def defined5 defined6 defined-not-I foundation11 foundation18'*
foundation6 foundation7 foundation9 invalid-def)
have B: $\tau \models \delta\ x \Longrightarrow \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil\rfloor\rfloor = x\ \tau$
apply(*cases x* τ, *metis bot-option-def foundation16*)

apply(*rename-tac x*$'$, *case-tac x*$'$, *metis bot-option-def foundation16 null-option-def*)
by(*simp*)
show $\tau \models$ *not* (υ *x and not* (δ *x*)) $\Longrightarrow$
($x +_{int}$ **0**) τ = (*if* υ *x and not* (δ *x*) *then invalid else x endif*) τ
apply(*subst OclIf-false*$'$, *simp*, *simp add*: *A*, *auto simp*: *OclAdd*$_{Integer}$*-def OclInt0-def*)

apply(*simp add*: *foundation16*$'$[*simplified OclValid-def*])
apply(*simp add*: *B*)
by(*simp add*: *OclValid-def*)
apply-end(*metis OclValid-def defined5 defined6 defined-and-I defined-not-I foundation9*)
qed

lemma *OclAdd*$_{Integer}$*-zero2*[*simp*,*code-unfold*] :
(**0** $+_{int}$ *x*) = (*if* υ *x and not* (δ *x*) *then invalid else x endif*)
by(*subst OclAdd*$_{Integer}$*-commute*, *simp*)

Test Statements Here follows a list of code-examples, that explain the meanings of the above definitions by compilation to code and execution to *True*.

Assert $\tau \models$ (**9** $\leq_{int}$ **10**)
Assert $\tau \models$ ((**4** $+_{int}$ **4**) $\leq_{int}$ **10**)
Assert $\neg$($\tau \models$ ((**4** $+_{int}$ (**4** $+_{int}$ **4**)) $<_{int}$ **10**))
Assert $\tau \models$ *not* (υ (*null* $+_{int}$ **1**))
Assert $\tau \models$ (((**9** $*_{int}$ **4**) *div*$_{int}$ **10**) $\leq_{int}$ **4**)
Assert $\tau \models$ *not* (δ (**1** *div*$_{int}$ **0**))
Assert $\tau \models$ *not* (υ (**1** *div*$_{int}$ **0**))

Fundamental Predicates on Integers: Strict Equality

Definition The last basic operation belonging to the fundamental infrastructure of a value-type in OCL is the weak equality, which is defined similar to the $'\mathfrak{A}$ *Boolean*-case as strict extension of the strong equality:

defs *StrictRefEq*$_{Integer}$[*code-unfold*] :
(*x*::($'\mathfrak{A}$)*Integer*) $\doteq$ *y* $\equiv$ λ τ. *if* (υ *x*) τ = *true* τ $\wedge$ (υ *y*) τ = *true* τ
then (*x* $\triangleq$ *y*) τ
else invalid τ

Property proof in terms of *profile-bin3*

interpretation *StrictRefEq*$_{Integer}$: *profile-bin3* λ *x y*. (*x*::($'\mathfrak{A}$)*Integer*) $\doteq$ *y*
by *unfold-locales* (*auto simp*: *StrictRefEq*$_{Integer}$)

lemma *integer-non-null* [*simp*]: ((λ-. $\lfloor\lfloor n\rfloor\rfloor$) $\doteq$ (*null*::($'\mathfrak{A}$)*Integer*)) = *false*
by(*rule ext*,*auto simp*: *StrictRefEq*$_{Integer}$ *valid-def*
bot-fun-def bot-option-def null-fun-def null-option-def StrongEq-def)

lemma *null-non-integer* [*simp*]: ((*null*::($'\mathfrak{A}$)*Integer*) $\doteq$ (λ-. $\lfloor\lfloor n\rfloor\rfloor$)) = *false*
by(*rule ext*,*auto simp*: *StrictRefEq*$_{Integer}$ *valid-def*

bot-fun-def bot-option-def null-fun-def null-option-def StrongEq-def)

lemma *OclInt0-non-null* [*simp*,*code-unfold*]: $(\mathbf{0} \doteq \mathit{null}) = \mathit{false}$ **by**(*simp add*: *OclInt0-def*)

lemma *null-non-OclInt0* [*simp*,*code-unfold*]: $(\mathit{null} \doteq \mathbf{0}) = \mathit{false}$ **by**(*simp add*: *OclInt0-def*)

lemma *OclInt1-non-null* [*simp*,*code-unfold*]: $(\mathbf{1} \doteq \mathit{null}) = \mathit{false}$ **by**(*simp add*: *OclInt1-def*)

lemma *null-non-OclInt1* [*simp*,*code-unfold*]: $(\mathit{null} \doteq \mathbf{1}) = \mathit{false}$ **by**(*simp add*: *OclInt1-def*)

lemma *OclInt2-non-null* [*simp*,*code-unfold*]: $(\mathbf{2} \doteq \mathit{null}) = \mathit{false}$ **by**(*simp add*: *OclInt2-def*)

lemma *null-non-OclInt2* [*simp*,*code-unfold*]: $(\mathit{null} \doteq \mathbf{2}) = \mathit{false}$ **by**(*simp add*: *OclInt2-def*)

lemma *OclInt6-non-null* [*simp*,*code-unfold*]: $(\mathbf{6} \doteq \mathit{null}) = \mathit{false}$ **by**(*simp add*: *OclInt6-def*)

lemma *null-non-OclInt6* [*simp*,*code-unfold*]: $(\mathit{null} \doteq \mathbf{6}) = \mathit{false}$ **by**(*simp add*: *OclInt6-def*)

lemma *OclInt8-non-null* [*simp*,*code-unfold*]: $(\mathbf{8} \doteq \mathit{null}) = \mathit{false}$ **by**(*simp add*: *OclInt8-def*)

lemma *null-non-OclInt8* [*simp*,*code-unfold*]: $(\mathit{null} \doteq \mathbf{8}) = \mathit{false}$ **by**(*simp add*: *OclInt8-def*)

lemma *OclInt9-non-null* [*simp*,*code-unfold*]: $(\mathbf{9} \doteq \mathit{null}) = \mathit{false}$ **by**(*simp add*: *OclInt9-def*)

lemma *null-non-OclInt9* [*simp*,*code-unfold*]: $(\mathit{null} \doteq \mathbf{9}) = \mathit{false}$ **by**(*simp add*: *OclInt9-def*)

Test Statements on Basic Integer

Here follows a list of code-examples, that explain the meanings of the above definitions by compilation to code and execution to *True*.

Elementary computations on Integer

Assert $\tau \models ((\mathbf{0} <_{int} \mathbf{2})\ \mathit{and}\ (\mathbf{0} <_{int} \mathbf{1}))$

Assert $\tau \models \mathbf{1} <> \mathbf{2}$

Assert $\tau \models \mathbf{2} <> \mathbf{1}$

Assert $\tau \models \mathbf{2} \doteq \mathbf{2}$

Assert $\tau \models \upsilon\ \mathbf{4}$

Assert $\tau \models \delta\ \mathbf{4}$

Assert $\tau \models \upsilon\ (\mathit{null}::({}'\mathfrak{A})\mathit{Integer})$

Assert $\tau \models (\mathit{invalid} \triangleq \mathit{invalid})$

Assert $\tau \models (\mathit{null} \triangleq \mathit{null})$

Assert $\tau \models (\mathbf{4} \triangleq \mathbf{4})$

Assert $\neg(\tau \models (\mathbf{9} \triangleq \mathbf{10}))$

Assert $\neg(\tau \models (\mathit{invalid} \triangleq \mathbf{10}))$

Assert $\neg(\tau \models (\mathit{null} \triangleq \mathbf{10}))$

Assert $\neg(\tau \models (\mathit{invalid} \doteq (\mathit{invalid}::({}'\mathfrak{A})\mathit{Integer})))$

Assert $\neg(\tau \models \upsilon\ (\mathit{invalid} \doteq (\mathit{invalid}::({}'\mathfrak{A})\mathit{Integer})))$

Assert $\neg(\tau \models (\mathit{invalid} <> (\mathit{invalid}::({}'\mathfrak{A})\mathit{Integer})))$

Assert $\neg(\tau \models \upsilon\ (\mathit{invalid} <> (\mathit{invalid}::({}'\mathfrak{A})\mathit{Integer})))$

Assert $\tau \models (\mathit{null} \doteq (\mathit{null}::({}'\mathfrak{A})\mathit{Integer})\ )$

Assert $\tau \models (\mathit{null} \doteq (\mathit{null}::({}'\mathfrak{A})\mathit{Integer})\ )$

Assert $\tau \models (\mathbf{4} \doteq \mathbf{4})$

Assert $\neg(\tau \models (\mathbf{4} <> \mathbf{4}))$

Assert $\neg(\tau \models (\mathbf{4} \doteq \mathbf{10}))$

Assert $\tau \models (\mathbf{4} <> \mathbf{10})$

Assert $\neg(\tau \models (\mathbf{0} <_{int} \mathit{null}))$

Assert $\neg(\tau \models (\delta\ (\mathbf{0} <_{int} \mathit{null})))$

end

theory *UML-Real*
imports *../UML-PropertyProfiles*
begin

B.2.5. Basic Type Real: Operations

Basic Real Constants

Although the remaining part of this library reasons about reals abstractly, we provide here as example some convenient shortcuts.

definition *OclReal0* ::$('\mathfrak{A})$*Real* (**0**.**0**)
where **0**.**0** = $(\lambda$ - . $\lfloor\lfloor$*0*::*real*$\rfloor\rfloor)$

definition *OclReal1* ::$('\mathfrak{A})$*Real* (**1**.**0**)
where **1**.**0** = $(\lambda$ - . $\lfloor\lfloor$*1*::*real*$\rfloor\rfloor)$

definition *OclReal2* ::$('\mathfrak{A})$*Real* (**2**.**0**)
where **2**.**0** = $(\lambda$ - . $\lfloor\lfloor$*2*::*real*$\rfloor\rfloor)$

definition *OclReal3* ::$('\mathfrak{A})$*Real* (**3**.**0**)
where **3**.**0** = $(\lambda$ - . $\lfloor\lfloor$*3*::*real*$\rfloor\rfloor)$

definition *OclReal4* ::$('\mathfrak{A})$*Real* (**4**.**0**)
where **4**.**0** = $(\lambda$ - . $\lfloor\lfloor$*4*::*real*$\rfloor\rfloor)$

definition *OclReal5* ::$('\mathfrak{A})$*Real* (**5**.**0**)
where **5**.**0** = $(\lambda$ - . $\lfloor\lfloor$*5*::*real*$\rfloor\rfloor)$

definition *OclReal6* ::$('\mathfrak{A})$*Real* (**6**.**0**)
where **6**.**0** = $(\lambda$ - . $\lfloor\lfloor$*6*::*real*$\rfloor\rfloor)$

definition *OclReal7* ::$('\mathfrak{A})$*Real* (**7**.**0**)
where **7**.**0** = $(\lambda$ - . $\lfloor\lfloor$*7*::*real*$\rfloor\rfloor)$

definition *OclReal8* ::$('\mathfrak{A})$*Real* (**8**.**0**)
where **8**.**0** = $(\lambda$ - . $\lfloor\lfloor$*8*::*real*$\rfloor\rfloor)$

definition *OclReal9* ::$('\mathfrak{A})$*Real* (**9**.**0**)
where **9**.**0** = $(\lambda$ - . $\lfloor\lfloor$*9*::*real*$\rfloor\rfloor)$

definition *OclReal10* ::$('\mathfrak{A})$*Real* (**10**.**0**)

where $\mathbf{10.0} = (\lambda\ \text{-}\ .\ \lfloor\lfloor \mathit{10{::}real} \rfloor\rfloor)$

definition *OclRealpi* $::('\mathfrak{A})\mathit{Real}\ (\pi)$
where $\pi = (\lambda\ \text{-}\ .\ \lfloor\lfloor \mathit{pi} \rfloor\rfloor)$

Validity and Definedness Properties

lemma $\delta(\mathit{null}{::}('\mathfrak{A})\mathit{Real}) = \mathit{false}$ **by** *simp*
lemma $\upsilon(\mathit{null}{::}('\mathfrak{A})\mathit{Real}) = \mathit{true}$ **by** *simp*

lemma [*simp*,*code-unfold*]: $\delta\ (\lambda\text{-}.\ \lfloor\lfloor n \rfloor\rfloor) = \mathit{true}$
by(*simp add*:*defined-def true-def*
bot-fun-def bot-option-def null-fun-def null-option-def)

lemma [*simp*,*code-unfold*]: $\upsilon\ (\lambda\text{-}.\ \lfloor\lfloor n \rfloor\rfloor) = \mathit{true}$
by(*simp add*:*valid-def true-def*
bot-fun-def bot-option-def)

lemma [*simp*,*code-unfold*]: $\delta\ \mathbf{0.0} = \mathit{true}$ **by**(*simp add*:*OclReal0-def*)
lemma [*simp*,*code-unfold*]: $\upsilon\ \mathbf{0.0} = \mathit{true}$ **by**(*simp add*:*OclReal0-def*)
lemma [*simp*,*code-unfold*]: $\delta\ \mathbf{1.0} = \mathit{true}$ **by**(*simp add*:*OclReal1-def*)
lemma [*simp*,*code-unfold*]: $\upsilon\ \mathbf{1.0} = \mathit{true}$ **by**(*simp add*:*OclReal1-def*)
lemma [*simp*,*code-unfold*]: $\delta\ \mathbf{2.0} = \mathit{true}$ **by**(*simp add*:*OclReal2-def*)
lemma [*simp*,*code-unfold*]: $\upsilon\ \mathbf{2.0} = \mathit{true}$ **by**(*simp add*:*OclReal2-def*)
lemma [*simp*,*code-unfold*]: $\delta\ \mathbf{6.0} = \mathit{true}$ **by**(*simp add*:*OclReal6-def*)
lemma [*simp*,*code-unfold*]: $\upsilon\ \mathbf{6.0} = \mathit{true}$ **by**(*simp add*:*OclReal6-def*)
lemma [*simp*,*code-unfold*]: $\delta\ \mathbf{8.0} = \mathit{true}$ **by**(*simp add*:*OclReal8-def*)
lemma [*simp*,*code-unfold*]: $\upsilon\ \mathbf{8.0} = \mathit{true}$ **by**(*simp add*:*OclReal8-def*)
lemma [*simp*,*code-unfold*]: $\delta\ \mathbf{9.0} = \mathit{true}$ **by**(*simp add*:*OclReal9-def*)
lemma [*simp*,*code-unfold*]: $\upsilon\ \mathbf{9.0} = \mathit{true}$ **by**(*simp add*:*OclReal9-def*)

Arithmetical Operations

Definition 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).

Note that we can not follow the lexis of the OCL Standard for Isabelle technical reasons; these operators are heavily overloaded in the HOL library that a further overloading would lead to heavy technical buzz in this document.

definition $\mathit{OclAdd}_{\mathit{Real}}\ ::('\mathfrak{A})\mathit{Real} \Rightarrow ('\mathfrak{A})\mathit{Real} \Rightarrow ('\mathfrak{A})\mathit{Real}$ (**infix** $+_{\mathit{real}}$ *40*)
where $x +_{\mathit{real}} y \equiv \lambda\ \tau.\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau \wedge (\delta\ y)\ \tau = \mathit{true}\ \tau$
$\mathit{then}\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil + \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\mathit{else}\ \mathit{invalid}\ \tau$
interpretation $\mathit{OclAdd}_{\mathit{Real}}$: *profile-bin1 op* $+_{\mathit{real}}\ \lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil + \lceil\lceil y\rceil\rceil\rfloor\rfloor$
by *unfold-locales* (*auto simp*:$\mathit{OclAdd}_{\mathit{Real}}$*-def bot-option-def null-option-def*)

definition $OclMinus_{Real}$ ::$('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real$ (**infix** $-_{real}$ *41*)
where $x -_{real} y \equiv \lambda\ \tau.\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau \wedge (\delta\ y)\ \tau = \mathit{true}\ \tau$
$\quad\quad \mathit{then}\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil - \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\quad\quad \mathit{else}\ \mathit{invalid}\ \tau$
interpretation $OclMinus_{Real}$: *profile-bin1 op* $-_{real}$ $\lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil - \lceil\lceil y\rceil\rceil\rfloor\rfloor$
by *unfold-locales* (*auto simp*:$OclMinus_{Real}$*-def bot-option-def null-option-def*)

definition $OclMult_{Real}$ ::$('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real$ (**infix** $*_{real}$ *45*)
where $x *_{real} y \equiv \lambda\ \tau.\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau \wedge (\delta\ y)\ \tau = \mathit{true}\ \tau$
$\quad\quad \mathit{then}\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil * \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\quad\quad \mathit{else}\ \mathit{invalid}\ \tau$
interpretation $OclMult_{Real}$: *profile-bin1 op* $*_{real}$ $\lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil * \lceil\lceil y\rceil\rceil\rfloor\rfloor$
by *unfold-locales* (*auto simp*:$OclMult_{Real}$*-def bot-option-def null-option-def*)

Here is the special case of division, which is defined as invalid for division by zero.

definition $OclDivision_{Real}$ ::$('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real$ (**infix** div_{real} *45*)
where $x\ div_{real}\ y \equiv \lambda\ \tau.\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau \wedge (\delta\ y)\ \tau = \mathit{true}\ \tau$
$\quad\quad \mathit{then}\ \mathit{if}\ y\ \tau \neq OclReal0\ \tau\ \mathit{then}\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil\ /\ \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor\ \mathit{else}\ \mathit{invalid}\ \tau$
$\quad\quad \mathit{else}\ \mathit{invalid}\ \tau$

definition *mod-float a b* $=$ *a* $-$ *real* (*floor* (*a* / *b*)) $*$ *b*
definition $OclModulus_{Real}$ ::$('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real$ (**infix** mod_{real} *45*)
where $x\ mod_{real}\ y \equiv \lambda\ \tau.\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau \wedge (\delta\ y)\ \tau = \mathit{true}\ \tau$
$\quad\quad \mathit{then}\ \mathit{if}\ y\ \tau \neq OclReal0\ \tau\ \mathit{then}\ \lfloor\lfloor \textit{mod-float}\ \lceil\lceil x\ \tau\rceil\rceil\ \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor\ \mathit{else}\ \mathit{invalid}\ \tau$
$\quad\quad \mathit{else}\ \mathit{invalid}\ \tau$

definition $OclLess_{Real}$ ::$('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Boolean$ (**infix** $<_{real}$ *35*)
where $x <_{real} y \equiv \lambda\ \tau.\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau \wedge (\delta\ y)\ \tau = \mathit{true}\ \tau$
$\quad\quad \mathit{then}\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil < \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\quad\quad \mathit{else}\ \mathit{invalid}\ \tau$
interpretation $OclLess_{Real}$: *profile-bin1 op* $<_{real}$ $\lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil < \lceil\lceil y\rceil\rceil\rfloor\rfloor$
by *unfold-locales* (*auto simp*:$OclLess_{Real}$*-def bot-option-def null-option-def*)

definition $OclLe_{Real}$ ::$('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Boolean$ (**infix** $\leq_{real}$ *35*)
where $x \leq_{real} y \equiv \lambda\ \tau.\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau \wedge (\delta\ y)\ \tau = \mathit{true}\ \tau$
$\quad\quad \mathit{then}\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil \leq \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\quad\quad \mathit{else}\ \mathit{invalid}\ \tau$
interpretation $OclLe_{Real}$: *profile-bin1 op* $\leq_{real}$ $\lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil \leq \lceil\lceil y\rceil\rceil\rfloor\rfloor$
by *unfold-locales* (*auto simp*:$OclLe_{Real}$*-def bot-option-def null-option-def*)

Basic Properties **lemma** $OclAdd_{Real}$*-commute*: $(X +_{real} Y) = (Y +_{real} X)$
by(*rule ext,auto simp*:*true-def false-def* $OclAdd_{Real}$*-def invalid-def*
split: *option.split option.split-asm*
bool.split bool.split-asm)

Execution with Invalid or Null or Zero as Argument **lemma** $\mathit{OclAdd}_{\mathit{Real}}\text{-}\mathit{zero1}[\mathit{simp},\mathit{code\text{-}unfold}]$:
$(x +_{real} \mathbf{0.0}) = (\mathit{if}\ \upsilon\ x\ \mathit{and\ not}\ (\delta\ x)\ \mathit{then\ invalid\ else}\ x\ \mathit{endif})$
proof $(\mathit{rule\ ext},\ \mathit{rename\text{-}tac}\ \tau,\ \mathit{case\text{-}tac}\ (\upsilon\ x\ \mathit{and\ not}\ (\delta\ x))\ \tau = \mathit{true}\ \tau)$
fix τ **show** $(\upsilon\ x\ \mathit{and\ not}\ (\delta\ x))\ \tau = \mathit{true}\ \tau \Longrightarrow$
$(x +_{real} \mathbf{0.0})\ \tau = (\mathit{if}\ \upsilon\ x\ \mathit{and\ not}\ (\delta\ x)\ \mathit{then\ invalid\ else}\ x\ \mathit{endif})\ \tau$
apply$(\mathit{subst\ OclIf\text{-}true}',\ \mathit{simp\ add}\colon \mathit{OclValid\text{-}def})$
by $(\mathit{metis}\ \mathit{OclAdd}_{\mathit{Real}}\text{-}\mathit{def}\ \mathit{OclNot\text{-}defargs}\ \mathit{OclValid\text{-}def}\ \mathit{foundation5}\ \mathit{foundation9})$
apply-end *assumption*
next fix τ
have $A\colon \bigwedge\tau.\ (\tau \models \mathit{not}\ (\upsilon\ x\ \mathit{and\ not}\ (\delta\ x))) = (x\ \tau = \mathit{invalid}\ \tau \vee \tau \models \delta\ x)$
by $(\mathit{metis\ OclNot\text{-}not\ OclOr\text{-}def\ defined5\ defined6\ defined\text{-}not\text{-}I\ foundation11\ foundation18}'$
$\mathit{foundation6\ foundation7\ foundation9\ invalid\text{-}def})$
have $B\colon \tau \models \delta\ x \Longrightarrow \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil\rfloor\rfloor = x\ \tau$
apply$(\mathit{cases}\ x\ \tau,\ \mathit{metis\ bot\text{-}option\text{-}def\ foundation16})$
apply$(\mathit{rename\text{-}tac}\ x',\ \mathit{case\text{-}tac}\ x',\ \mathit{metis\ bot\text{-}option\text{-}def\ foundation16\ null\text{-}option\text{-}def})$
by(simp)
show $\tau \models \mathit{not}\ (\upsilon\ x\ \mathit{and\ not}\ (\delta\ x)) \Longrightarrow$
$(x +_{real} \mathbf{0.0})\ \tau = (\mathit{if}\ \upsilon\ x\ \mathit{and\ not}\ (\delta\ x)\ \mathit{then\ invalid\ else}\ x\ \mathit{endif})\ \tau$
apply$(\mathit{subst\ OclIf\text{-}false}',\ \mathit{simp},\ \mathit{simp\ add}\colon A,\ \mathit{auto\ simp}\colon \mathit{OclAdd}_{\mathit{Real}}\text{-}\mathit{def}\ \mathit{OclReal0\text{-}def})$

apply$(\mathit{simp\ add}\colon \mathit{foundation16}'[\mathit{simplified\ OclValid\text{-}def}])$
apply$(\mathit{simp\ add}\colon B)$
by$(\mathit{simp\ add}\colon \mathit{OclValid\text{-}def})$
apply-end$(\mathit{metis\ OclValid\text{-}def\ defined5\ defined6\ defined\text{-}and\text{-}I\ defined\text{-}not\text{-}I\ foundation9})$
qed

lemma $\mathit{OclAdd}_{\mathit{Real}}\text{-}\mathit{zero2}[\mathit{simp},\mathit{code\text{-}unfold}]$:
$(\mathbf{0.0} +_{real} x) = (\mathit{if}\ \upsilon\ x\ \mathit{and\ not}\ (\delta\ x)\ \mathit{then\ invalid\ else}\ x\ \mathit{endif})$
by$(\mathit{subst}\ \mathit{OclAdd}_{\mathit{Real}}\text{-}\mathit{commute},\ \mathit{simp})$

Test Statements Here follows a list of code-examples, that explain the meanings of the above definitions by compilation to code and execution to *True*.

Assert $\tau \models (\ \mathbf{9.0} \le_{real} \mathbf{10.0}\ )$
Assert $\tau \models ((\ \mathbf{4.0} +_{real} \mathbf{4.0}\ ) \le_{real} \mathbf{10.0}\ )$
Assert $\neg(\tau \models ((\ \mathbf{4.0} +_{real} (\ \mathbf{4.0} +_{real} \mathbf{4.0}\ ))\ <_{real} \mathbf{10.0}\ ))$
Assert $\tau \models \mathit{not}\ (\upsilon\ (\mathit{null} +_{real} \mathbf{1.0}))$
Assert $\tau \models (((\mathbf{9.0} *_{real} \mathbf{4.0})\ \mathit{div}_{real}\ \mathbf{10.0}) \le_{real} \mathbf{4.0})$
Assert $\tau \models \mathit{not}\ (\delta\ (\mathbf{1.0}\ \mathit{div}_{real}\ \mathbf{0.0}))$
Assert $\tau \models \mathit{not}\ (\upsilon\ (\mathbf{1.0}\ \mathit{div}_{real}\ \mathbf{0.0}))$

Fundamental Predicates on Reals: Strict Equality

Definition The last basic operation belonging to the fundamental infrastructure of a value-type in OCL is the weak equality, which is defined similar to the $'\mathfrak{A}$ *Boolean*-case as strict extension of the strong equality:

defs $\mathit{StrictRefEq}_{\mathit{Real}}\ [\mathit{code\text{-}unfold}]$:
$(x{::}('\mathfrak{A})\mathit{Real}) \doteq y \equiv \lambda\ \tau.\ \mathit{if}\ (\upsilon\ x)\ \tau = \mathit{true}\ \tau \wedge (\upsilon\ y)\ \tau = \mathit{true}\ \tau$

then $(x \triangleq y)\ \tau$

else invalid τ

Property proof in terms of *profile-bin3*

interpretation $\mathit{StrictRefEq}_{\mathit{Real}}$: *profile-bin3* $\lambda\ x\ y.\ (x{::}({}'\mathfrak{A})\mathit{Real}) \doteq y$

by *unfold-locales* (*auto simp*: $\mathit{StrictRefEq}_{\mathit{Real}}$)

lemma *real-non-null* [*simp*]: $((\lambda\text{-}.\ \lfloor\lfloor n\rfloor\rfloor) \doteq (\mathit{null}{::}({}'\mathfrak{A})\mathit{Real})) = \mathit{false}$

by(*rule ext*,*auto simp*: $\mathit{StrictRefEq}_{\mathit{Real}}$ *valid-def*

bot-fun-def bot-option-def null-fun-def null-option-def StrongEq-def)

lemma *null-non-real* [*simp*]: $((\mathit{null}{::}({}'\mathfrak{A})\mathit{Real}) \doteq (\lambda\text{-}.\ \lfloor\lfloor n\rfloor\rfloor)) = \mathit{false}$

by(*rule ext*,*auto simp*: $\mathit{StrictRefEq}_{\mathit{Real}}$ *valid-def*

bot-fun-def bot-option-def null-fun-def null-option-def StrongEq-def)

lemma *OclReal0-non-null* [*simp*,*code-unfold*]: $(\mathbf{0.0} \doteq \mathit{null}) = \mathit{false}$ **by**(*simp add*: *OclReal0-def*)

lemma *null-non-OclReal0* [*simp*,*code-unfold*]: $(\mathit{null} \doteq \mathbf{0.0}) = \mathit{false}$ **by**(*simp add*: *OclReal0-def*)

lemma *OclReal1-non-null* [*simp*,*code-unfold*]: $(\mathbf{1.0} \doteq \mathit{null}) = \mathit{false}$ **by**(*simp add*: *OclReal1-def*)

lemma *null-non-OclReal1* [*simp*,*code-unfold*]: $(\mathit{null} \doteq \mathbf{1.0}) = \mathit{false}$ **by**(*simp add*: *OclReal1-def*)

lemma *OclReal2-non-null* [*simp*,*code-unfold*]: $(\mathbf{2.0} \doteq \mathit{null}) = \mathit{false}$ **by**(*simp add*: *OclReal2-def*)

lemma *null-non-OclReal2* [*simp*,*code-unfold*]: $(\mathit{null} \doteq \mathbf{2.0}) = \mathit{false}$ **by**(*simp add*: *OclReal2-def*)

lemma *OclReal6-non-null* [*simp*,*code-unfold*]: $(\mathbf{6.0} \doteq \mathit{null}) = \mathit{false}$ **by**(*simp add*: *OclReal6-def*)

lemma *null-non-OclReal6* [*simp*,*code-unfold*]: $(\mathit{null} \doteq \mathbf{6.0}) = \mathit{false}$ **by**(*simp add*: *OclReal6-def*)

lemma *OclReal8-non-null* [*simp*,*code-unfold*]: $(\mathbf{8.0} \doteq \mathit{null}) = \mathit{false}$ **by**(*simp add*: *OclReal8-def*)

lemma *null-non-OclReal8* [*simp*,*code-unfold*]: $(\mathit{null} \doteq \mathbf{8.0}) = \mathit{false}$ **by**(*simp add*: *OclReal8-def*)

lemma *OclReal9-non-null* [*simp*,*code-unfold*]: $(\mathbf{9.0} \doteq \mathit{null}) = \mathit{false}$ **by**(*simp add*: *OclReal9-def*)

lemma *null-non-OclReal9* [*simp*,*code-unfold*]: $(\mathit{null} \doteq \mathbf{9.0}) = \mathit{false}$ **by**(*simp add*: *OclReal9-def*)

Const **lemma** [*simp*,*code-unfold*]: *const*($\mathbf{0.0}$) **by**(*simp add*: *const-ss OclReal0-def*)

lemma [*simp*,*code-unfold*]: *const*($\mathbf{1.0}$) **by**(*simp add*: *const-ss OclReal1-def*)

lemma [*simp*,*code-unfold*]: *const*($\mathbf{2.0}$) **by**(*simp add*: *const-ss OclReal2-def*)

lemma [*simp*,*code-unfold*]: *const*($\mathbf{6.0}$) **by**(*simp add*: *const-ss OclReal6-def*)

lemma [*simp*,*code-unfold*]: *const*($\mathbf{8.0}$) **by**(*simp add*: *const-ss OclReal8-def*)

lemma [*simp*,*code-unfold*]: *const*($\mathbf{9.0}$) **by**(*simp add*: *const-ss OclReal9-def*)

Test Statements on Basic Real

Here follows a list of code-examples, that explain the meanings of the above definitions by compilation to code and execution to *True*.

Elementary computations on Real

Assert $\tau \models \mathbf{1.0} <> \mathbf{2.0}$

Assert $\tau \models \mathbf{2.0} <> \mathbf{1.0}$

Assert $\tau \models \mathbf{2.0} \doteq \mathbf{2.0}$

Assert $\tau \models \upsilon\ \mathbf{4.0}$

Assert $\tau \models \delta\ \mathbf{4.0}$

Assert $\quad\tau \models \upsilon\ (\mathit{null}::({}'\mathfrak{A})\mathit{Real})$
Assert $\quad\tau \models (\mathit{invalid} \triangleq \mathit{invalid})$
Assert $\quad\tau \models (\mathit{null} \triangleq \mathit{null})$
Assert $\quad\tau \models (\mathbf{4.0} \triangleq \mathbf{4.0})$
Assert $\neg(\tau \models (\mathbf{9.0} \triangleq \mathbf{10.0}))$
Assert $\neg(\tau \models (\mathit{invalid} \triangleq \mathbf{10.0}))$
Assert $\neg(\tau \models (\mathit{null} \triangleq \mathbf{10.0}))$
Assert $\neg(\tau \models (\mathit{invalid} \doteq (\mathit{invalid}::({}'\mathfrak{A})\mathit{Real})))$
Assert $\neg(\tau \models \upsilon\ (\mathit{invalid} \doteq (\mathit{invalid}::({}'\mathfrak{A})\mathit{Real})))$
Assert $\neg(\tau \models (\mathit{invalid} <> (\mathit{invalid}::({}'\mathfrak{A})\mathit{Real})))$
Assert $\neg(\tau \models \upsilon\ (\mathit{invalid} <> (\mathit{invalid}::({}'\mathfrak{A})\mathit{Real})))$
Assert $\quad\tau \models (\mathit{null} \doteq (\mathit{null}::({}'\mathfrak{A})\mathit{Real})\ )$
Assert $\quad\tau \models (\mathit{null} \doteq (\mathit{null}::({}'\mathfrak{A})\mathit{Real})\ )$
Assert $\quad\tau \models (\mathbf{4.0} \doteq \mathbf{4.0})$
Assert $\neg(\tau \models (\mathbf{4.0} <> \mathbf{4.0}))$
Assert $\neg(\tau \models (\mathbf{4.0} \doteq \mathbf{10.0}))$
Assert $\quad\tau \models (\mathbf{4.0} <> \mathbf{10.0})$
Assert $\neg(\tau \models (\mathbf{0.0} <_{real} \mathit{null}))$
Assert $\neg(\tau \models (\delta\ (\mathbf{0.0} <_{real} \mathit{null})))$

end

theory *UML-String*
imports *../UML-PropertyProfiles*
begin

B.2.6. Basic Type String: Operations

Basic String Constants

Although the remaining part of this library reasons about integers abstractly, we provide here as example some convenient shortcuts.

definition *OclStringa* $::({}'\mathfrak{A})\mathit{String}$ (a)
where $\quad \mathrm{a} = (\lambda\ \text{-}\ .\ \lfloor\lfloor{}''a''\rfloor\rfloor)$

definition *OclStringb* $::({}'\mathfrak{A})\mathit{String}$ (b)
where $\quad \mathrm{b} = (\lambda\ \text{-}\ .\ \lfloor\lfloor{}''b''\rfloor\rfloor)$

definition *OclStringc* $::({}'\mathfrak{A})\mathit{String}$ (c)
where $\quad \mathrm{c} = (\lambda\ \text{-}\ .\ \lfloor\lfloor{}''c''\rfloor\rfloor)$

Validity and Definedness Properties

lemma $\delta(\mathit{null}::({}'\mathfrak{A})\mathit{String}) = \mathit{false}$ **by** *simp*

lemma $\upsilon(\mathit{null}::({}^{\prime}\mathfrak{A})\mathit{String}) = \mathit{true}$ **by** *simp*

lemma [*simp*,*code-unfold*]: $\delta\ (\lambda\text{-}.\ \lfloor\lfloor n\rfloor\rfloor) = \mathit{true}$
by(*simp add*:*defined-def true-def*
bot-fun-def bot-option-def null-fun-def null-option-def)

lemma [*simp*,*code-unfold*]: $\upsilon\ (\lambda\text{-}.\ \lfloor\lfloor n\rfloor\rfloor) = \mathit{true}$
by(*simp add*:*valid-def true-def*
bot-fun-def bot-option-def)

lemma [*simp*,*code-unfold*]: δ a $= \mathit{true}$ **by**(*simp add*:*OclStringa-def*)
lemma [*simp*,*code-unfold*]: υ a $= \mathit{true}$ **by**(*simp add*:*OclStringa-def*)

String Operations

Definition 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).

Note that we can not follow the lexis of the OCL Standard for Isabelle technical reasons; these operators are heavily overloaded in the HOL library that a further overloading would lead to heavy technical buzz in this document.

definition $\mathit{OclAdd}_{\mathit{String}} ::({}^{\prime}\mathfrak{A})\mathit{String} \Rightarrow ({}^{\prime}\mathfrak{A})\mathit{String} \Rightarrow ({}^{\prime}\mathfrak{A})\mathit{String}$ (**infix** $+_{\mathit{string}}$ *40*)
where $x +_{\mathit{string}} y \equiv \lambda\ \tau.\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau \wedge (\delta\ y)\ \tau = \mathit{true}\ \tau$
$\mathit{then}\ \lfloor\lfloor \mathit{concat}\ [\lceil\lceil x\ \tau\rceil\rceil, \lceil\lceil y\ \tau\rceil\rceil]\rfloor\rfloor$
$\mathit{else}\ \mathit{invalid}\ \tau$
interpretation $\mathit{OclAdd}_{\mathit{String}}$: *profile-bin1 op* $+_{\mathit{string}}\ \lambda\ x\ y.\ \lfloor\lfloor \mathit{concat}\ [\lceil\lceil x\rceil\rceil, \lceil\lceil y\rceil\rceil]\rfloor\rfloor$
by *unfold-locales* (*auto simp*:$\mathit{OclAdd}_{\mathit{String}}$*-def bot-option-def null-option-def*)

Basic Properties **lemma** $\mathit{OclAdd}_{\mathit{String}}$*-not-commute*: $\exists X\ Y.\ (X +_{\mathit{string}} Y) \neq (Y +_{\mathit{string}} X)$
apply(*rule-tac x* $= \lambda\text{-}.\ \lfloor\lfloor{}''b''\rfloor\rfloor$ **in** *exI*)
apply(*rule-tac x* $= \lambda\text{-}.\ \lfloor\lfloor{}''a''\rfloor\rfloor$ **in** *exI*)
apply(*simp-all add*:$\mathit{OclAdd}_{\mathit{String}}$*-def*)
by(*auto*, *drule fun-cong*, *auto*)

Test Statements Here follows a list of code-examples, that explain the meanings of the above definitions by compilation to code and execution to *True*.

Fundamental Properties on Strings: Strict Equality

Definition The last basic operation belonging to the fundamental infrastructure of a value-type in OCL is the weak equality, which is defined similar to the ${}^{\prime}\mathfrak{A}$ *Boolean*-case as strict extension of the strong equality:

defs $\mathit{StrictRefEq}_{\mathit{String}}$[*code-unfold*] :
$(x::({}^{\prime}\mathfrak{A})\mathit{String}) \doteq y \equiv \lambda\ \tau.\ \mathit{if}\ (\upsilon\ x)\ \tau = \mathit{true}\ \tau \wedge (\upsilon\ y)\ \tau = \mathit{true}\ \tau$
$\mathit{then}\ (x \triangleq y)\ \tau$
$\mathit{else}\ \mathit{invalid}\ \tau$

Property proof in terms of *profile-bin3*

interpretation $StrictRefEq_{String}$: $profile\text{-}bin3\ \lambda\ x\ y.\ (x::({}'\mathfrak{A})String) \doteq y$
by *unfold-locales* (*auto simp*: $StrictRefEq_{String}$)

Test Statements on Basic String

Here follows a list of code-examples, that explain the meanings of the above definitions by compilation to code and execution to *True*.

Elementary computations on String

Assert $\tau \models \mathrm{a} <> \mathrm{b}$
Assert $\tau \models \mathrm{b} <> \mathrm{a}$
Assert $\tau \models \mathrm{b} \doteq \mathrm{b}$

Assert $\tau \models \upsilon\ \mathrm{a}$
Assert $\tau \models \delta\ \mathrm{a}$
Assert $\tau \models \upsilon\ (null::({}'\mathfrak{A})String)$
Assert $\tau \models (invalid \triangleq invalid)$
Assert $\tau \models (null \triangleq null)$
Assert $\tau \models (\mathrm{a} \triangleq \mathrm{a})$
Assert $\neg(\tau \models (\mathrm{a} \triangleq \mathrm{b}))$
Assert $\neg(\tau \models (invalid \triangleq \mathrm{b}))$
Assert $\neg(\tau \models (null \triangleq \mathrm{b}))$
Assert $\neg(\tau \models (invalid \doteq (invalid::({}'\mathfrak{A})String)))$
Assert $\neg(\tau \models \upsilon\ (invalid \doteq (invalid::({}'\mathfrak{A})String)))$
Assert $\neg(\tau \models (invalid <> (invalid::({}'\mathfrak{A})String)))$
Assert $\neg(\tau \models \upsilon\ (invalid <> (invalid::({}'\mathfrak{A})String)))$
Assert $\tau \models (null \doteq (null::({}'\mathfrak{A})String)\ )$
Assert $\tau \models (null \doteq (null::({}'\mathfrak{A})String)\ )$
Assert $\tau \models (\mathrm{b} \doteq \mathrm{b})$
Assert $\neg(\tau \models (\mathrm{b} <> \mathrm{b}))$
Assert $\neg(\tau \models (\mathrm{b} \doteq \mathrm{c}))$
Assert $\tau \models (\mathrm{b} <> \mathrm{c})$

end

theory *UML-Pair*
imports *../basic-types/UML-Boolean*
../basic-types/UML-Integer
begin

B.2.7. Collection Type Pairs: Operations

The OCL standard provides the concept of *Tuples*, i. e. a family of record-types with projection functions. In FeatherWeight OCL, only the theory of a special case is developped, namely the type of Pairs, which is, however, sufficient for all applications since it can be used to mimick all tuples. In particular, it can be used to express operations with multiple arguments, roles of n-ary associations, ...

Semantic Properties of the Type Constructor

lemma $A[simp]{:}Rep\text{-}Pair_{base}\ x \neq None \Longrightarrow Rep\text{-}Pair_{base}\ x \neq null \Longrightarrow (fst\ \lceil\lceil Rep\text{-}Pair_{base}\ x\rceil\rceil) \neq bot$
by$(insert\ Rep\text{-}Pair_{base}[of\ x], auto\ simp{:}null\text{-}option\text{-}def\ bot\text{-}option\text{-}def)$

lemma $A'[simp]{:}\ x \neq bot \Longrightarrow\ x \neq null \Longrightarrow (fst\ \lceil\lceil Rep\text{-}Pair_{base}\ x\rceil\rceil) \neq bot$
apply$(insert\ Rep\text{-}Pair_{base}[of\ x],\ simp\ add{:}\ bot\text{-}Pair_{base}\text{-}def\ null\text{-}Pair_{base}\text{-}def)$
apply$(auto\ simp{:}null\text{-}option\text{-}def\ bot\text{-}option\text{-}def)$
apply$(erule\ contrapos\text{-}np[of\ x = Abs\text{-}Pair_{base}\ None])$
apply$(subst\ Rep\text{-}Pair_{base}\text{-}inject[symmetric],\ simp)$
apply$(subst\ Pair_{base}.Abs\text{-}Pair_{base}\text{-}inverse,\ simp\text{-}all, simp\ add{:}\ bot\text{-}option\text{-}def)$
apply$(erule\ contrapos\text{-}np[of\ x = Abs\text{-}Pair_{base}\ \lfloor None\rfloor])$
apply$(subst\ Rep\text{-}Pair_{base}\text{-}inject[symmetric],\ simp)$
apply$(subst\ Pair_{base}.Abs\text{-}Pair_{base}\text{-}inverse,\ simp\text{-}all, simp\ add{:}\ null\text{-}option\text{-}def\ bot\text{-}option\text{-}def)$
done

lemma $B[simp]{:}Rep\text{-}Pair_{base}\ x \neq None \Longrightarrow Rep\text{-}Pair_{base}\ x \neq null \Longrightarrow (snd\ \lceil\lceil Rep\text{-}Pair_{base}\ x\rceil\rceil) \neq bot$
by$(insert\ Rep\text{-}Pair_{base}[of\ x], auto\ simp{:}null\text{-}option\text{-}def\ bot\text{-}option\text{-}def)$

lemma $B'[simp]{:}x \neq bot \Longrightarrow x \neq null \Longrightarrow (snd\ \lceil\lceil Rep\text{-}Pair_{base}\ x\rceil\rceil) \neq bot$
apply$(insert\ Rep\text{-}Pair_{base}[of\ x],\ simp\ add{:}\ bot\text{-}Pair_{base}\text{-}def\ null\text{-}Pair_{base}\text{-}def)$
apply$(auto\ simp{:}null\text{-}option\text{-}def\ bot\text{-}option\text{-}def)$
apply$(erule\ contrapos\text{-}np[of\ x = Abs\text{-}Pair_{base}\ None])$
apply$(subst\ Rep\text{-}Pair_{base}\text{-}inject[symmetric],\ simp)$
apply$(subst\ Pair_{base}.Abs\text{-}Pair_{base}\text{-}inverse,\ simp\text{-}all, simp\ add{:}\ bot\text{-}option\text{-}def)$
apply$(erule\ contrapos\text{-}np[of\ x = Abs\text{-}Pair_{base}\ \lfloor None\rfloor])$
apply$(subst\ Rep\text{-}Pair_{base}\text{-}inject[symmetric],\ simp)$
apply$(subst\ Pair_{base}.Abs\text{-}Pair_{base}\text{-}inverse,\ simp\text{-}all, simp\ add{:}\ null\text{-}option\text{-}def\ bot\text{-}option\text{-}def)$
done

Strict Equality

Definition After the part of foundational operations on sets, we detail here equality on sets. Strong equality is inherited from the OCL core, but we have to consider the case of the strict equality. We decide to overload strict equality in the same way we do for other value's in OCL:

defs $StrictRefEq_{Pair}$:

$$\begin{aligned}
((x{::}('\mathfrak{A},'\alpha{::}null,'\beta{::}null)Pair) \doteq y) \equiv (\lambda\ \tau.\ &if\ (\upsilon\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau\\
&then\ (x \triangleq y)\tau\\
&else\ invalid\ \tau)
\end{aligned}$$

Property proof in terms of *profile-bin3*

interpretation *StrictRefEq*$_{Pair}$: *profile-bin3* λ *x y*. (x::($'\mathfrak{A}$,$'\alpha$::*null*,$'\beta$::*null*)*Pair*) $\doteq$ *y*
by *unfold-locales* (*auto simp*: *StrictRefEq*$_{Pair}$)

Standard Operations

This part provides a collection of operators for the Pair type.

Definition: OclPair Constructor **definition** *OclPair*::($'\mathfrak{A}$, $'\alpha$) *val* $\Rightarrow$
($'\mathfrak{A}$, $'\beta$) *val* $\Rightarrow$
($'\mathfrak{A}$,$'\alpha$::*null*,$'\beta$::*null*) *Pair* (*Pair*{(-),(-)})
where *Pair*{*X*,*Y*} $\equiv$ (λ τ. *if* (υ *X*) τ = *true* τ $\wedge$ (υ *Y*) τ = *true* τ
then Abs-Pair$_{base}$ $\lfloor\lfloor$(*X* τ, *Y* τ)$\rfloor\rfloor$
else invalid τ)

interpretation *OclPair* : *profile-bin4*
OclPair λ *x y*. *Abs-Pair*$_{base}$ $\lfloor\lfloor$(*x*, *y*)$\rfloor\rfloor$
apply(*unfold-locales*, *auto simp*: *OclPair-def bot-Pair*$_{base}$*-def null-Pair*$_{base}$*-def*)
by(*auto simp*: *Abs-Pair*$_{base}$*-inject null-option-def bot-option-def*)

Definition: OclFst **definition** *OclFirst*:: ($'\mathfrak{A}$,$'\alpha$::*null*,$'\beta$::*null*) *Pair* $\Rightarrow$ ($'\mathfrak{A}$, $'\alpha$) *val* (- .*First*$'$($'$))
where *X* .*First*() $\equiv$ (λ τ. *if* (δ *X*) τ = *true* τ
then fst $\lceil\lceil$*Rep-Pair*$_{base}$ (*X* τ)$\rceil\rceil$
else invalid τ)

interpretation *OclFirst* : *profile-mono2 OclFirst* λ*x*. *fst* $\lceil\lceil$*Rep-Pair*$_{base}$ (*x*)$\rceil\rceil$
by *unfold-locales* (*auto simp*: *OclFirst-def*)

Definition: OclSnd **definition** *OclSecond*:: ($'\mathfrak{A}$,$'\alpha$::*null*,$'\beta$::*null*) *Pair* $\Rightarrow$ ($'\mathfrak{A}$, $'\beta$) *val* (- .*Second*$'$($'$))
where *X* .*Second*() $\equiv$ (λ τ. *if* (δ *X*) τ = *true* τ
then snd $\lceil\lceil$*Rep-Pair*$_{base}$ (*X* τ)$\rceil\rceil$
else invalid τ)

interpretation *OclSecond* : *profile-mono2 OclSecond* λ*x*. *snd* $\lceil\lceil$*Rep-Pair*$_{base}$ (*x*)$\rceil\rceil$
by *unfold-locales* (*auto simp*: *OclSecond-def*)

Logical Properties

lemma *1* : $\tau \models \upsilon\ Y \Longrightarrow \tau \models$ *Pair*{*X*,*Y*} .*First*() $\triangleq$ *X*
apply(*case-tac* $\neg(\tau \models \upsilon\ X)$)
apply(*erule foundation7*$'$[*THEN iffD2*, *THEN foundation15*[*THEN iffD2*,
THEN StrongEq-L-subst2-rev]],*simp-all add*:*foundation18*$'$)
apply(*auto simp*: *OclValid-def valid-def defined-def StrongEq-def OclFirst-def OclPair-def*
true-def false-def invalid-def bot-fun-def null-fun-def)
apply(*auto simp*: *Abs-Pair*$_{base}$*-inject null-option-def bot-option-def bot-Pair*$_{base}$*-def null-Pair*$_{base}$*-def*)
by(*simp add*: *Abs-Pair*$_{base}$*-inverse*)

lemma *2* : $\tau \models \upsilon\, X \Longrightarrow \tau \models$ *Pair*$\{X,Y\}$ *.Second*() $\triangleq Y$
apply(*case-tac* $\neg(\tau \models \upsilon\, Y)$)
apply(*erule foundation7′*[*THEN iffD2*, *THEN foundation15*[*THEN iffD2*,
THEN StrongEq-L-subst2-rev]],*simp-all add*:*foundation18′*)
apply(*auto simp*: *OclValid-def valid-def defined-def StrongEq-def OclSecond-def OclPair-def*
true-def false-def invalid-def bot-fun-def null-fun-def)
apply(*auto simp*: *Abs-Pair*$_{base}$*-inject null-option-def bot-option-def bot-Pair*$_{base}$*-def null-Pair*$_{base}$*-def*)
by(*simp add*: *Abs-Pair*$_{base}$*-inverse*)

Execution Properties

lemma *proj1-exec* [*simp*, *code-unfold*] : *Pair*$\{X,Y\}$ *.First*() = (*if* $(\upsilon\, Y)$ *then X else invalid endif*)
apply(*rule ext*, *rename-tac* τ, *simp add*: *foundation22*[*symmetric*])
apply(*case-tac* $\neg(\tau \models \upsilon\, Y)$)
apply(*erule foundation7′*[*THEN iffD2*, *THEN foundation15*[*THEN iffD2*,
THEN StrongEq-L-subst2-rev]],*simp-all*)
apply(*subgoal-tac* $\tau \models \upsilon\, Y$)
apply(*erule foundation13*[*THEN iffD2*, *THEN StrongEq-L-subst2-rev*], *simp-all*)
by(*erule 1*)

lemma *proj2-exec* [*simp*, *code-unfold*] : *Pair*$\{X,Y\}$ *.Second*() = (*if* $(\upsilon\, X)$ *then Y else invalid endif*)
apply(*rule ext*, *rename-tac* τ, *simp add*: *foundation22*[*symmetric*])
apply(*case-tac* $\neg(\tau \models \upsilon\, X)$)
apply(*erule foundation7′*[*THEN iffD2*, *THEN foundation15*[*THEN iffD2*,
THEN StrongEq-L-subst2-rev]],*simp-all*)
apply(*subgoal-tac* $\tau \models \upsilon\, X$)
apply(*erule foundation13*[*THEN iffD2*, *THEN StrongEq-L-subst2-rev*], *simp-all*)
by(*erule 2*)

Test Statements

Assert $\tau \models$ *invalid .First*() $\triangleq$ *invalid*
Assert $\tau \models$ *null .First*() $\triangleq$ *invalid*
Assert $\tau \models$ *null .Second*() $\triangleq$ *invalid .Second*()
Assert $\tau \models$ *Pair*{*invalid*, *true*} $\triangleq$ *invalid*
Assert $\tau \models \upsilon$(*Pair*{*null*, *true*}.*First*())
Assert $\tau \models$ (*Pair*{*null*, *true*}).*First*() $\triangleq$ *null*
Assert $\tau \models$ (*Pair*{*null*, *Pair*{*true*,*invalid*}}).*First*() $\triangleq$ *invalid*

end

theory *UML-Set*
imports *../basic-types/UML-Boolean*

../basic-types/UML-Integer
begin

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

B.2.8. Collection Type Set: Operations

As a Motivation for the (infinite) Type Construction: Type-Extensions as Sets

Our notion of typed set goes beyond the usual notion of a finite executable set and is powerful enough to capture *the extension of a type* in UML and OCL. This means we can have in Featherweight OCL Sets containing all possible elements of a type, not only those (finite) ones representable in a state. This holds for base types as well as class types, although the notion for class-types — involving object id's not occuring in a state — requires some care.

In a world with *invalid* and *null*, there are two notions extensions possible:

1. the set of all *defined* values of a type *T* (for which we will introduce the constant *T*)

2. the set of all *valid* values of a type *T*, so including *null* (for which we will introduce the constant T_{null}).

We define the set extensions for the base type *Integer* as follows:

definition *Integer* :: $('\mathfrak{A}, Integer_{base})$ *Set*
where *Integer* $\equiv$ $(\lambda\ \tau.\ (Abs\text{-}Set_{base}\ o\ Some\ o\ Some)\ \ ((Some\ o\ Some)\ `\ (UNIV{::}int\ set)))$

definition $Integer_{null}$ :: $('\mathfrak{A}, Integer_{base})$ *Set*
where $Integer_{null} \equiv (\lambda\ \tau.\ (Abs\text{-}Set_{base}\ o\ Some\ o\ Some)\ \ (Some\ `\ (UNIV{::}int\ option\ set)))$

lemma *Integer-defined* : δ *Integer* $=$ *true*
apply(*rule ext*, *auto simp*: *Integer-def defined-def false-def true-def*
bot-fun-def null-fun-def null-option-def)
by(*simp-all add*: $Abs\text{-}Set_{base}$*-inject bot-option-def* $bot\text{-}Set_{base}$*-def* $null\text{-}Set_{base}$*-def null-option-def*)

lemma $Integer_{null}$*-defined* : δ $Integer_{null}$ $=$ *true*
apply(*rule ext*, *auto simp*: $Integer_{null}$*-def defined-def false-def true-def*
bot-fun-def null-fun-def null-option-def)
by(*simp-all add*: $Abs\text{-}Set_{base}$*-inject bot-option-def* $bot\text{-}Set_{base}$*-def* $null\text{-}Set_{base}$*-def null-option-def*)

This allows the theorems:
$\tau \models \delta\ x \implies \tau \models (Integer{-}{>}includes(x))\ \tau \models \delta\ x \implies \tau \models Integer\ \triangleq (Integer{-}{>}including(x))$
and
$\tau \models \upsilon\ x \implies \tau \models (Integer_{null}{-}{>}includes(x))\ \tau \models \upsilon\ x \implies \tau \models Integer_{null}\ \triangleq (Integer_{null}{-}{>}including(x))$
which characterize the infiniteness of these sets by a recursive property on these sets.

Validity and Definedness Properties

Every element in a defined set is valid.

```
lemma Set-inv-lemma: τ ⊨ (δ X) ⟹ ∀x∈⌈⌈Rep-Set_base (X τ)⌉⌉. x ≠ bot
apply(insert Rep-Set_base [of X τ], simp)
apply(auto simp: OclValid-def defined-def false-def true-def cp-def
            bot-fun-def bot-Set_base-def null-Set_base-def null-fun-def
        split:split-if-asm)
 apply(erule contrapos-pp [of Rep-Set_base (X τ) = bot])
 apply(subst Abs-Set_base-inject[symmetric], rule Rep-Set_base, simp)
 apply(simp add: Rep-Set_base-inverse bot-Set_base-def bot-option-def)
apply(erule contrapos-pp [of Rep-Set_base (X τ) = null])
apply(subst Abs-Set_base-inject[symmetric], rule Rep-Set_base, simp)
apply(simp add: Rep-Set_base-inverse  null-option-def)
by (simp add: bot-option-def)

lemma Set-inv-lemma′ :
 assumes x-def : τ ⊨ δ X
    and e-mem : e ∈ ⌈⌈Rep-Set_base (X τ)⌉⌉
  shows τ ⊨ υ (λ-. e)
 apply(rule Set-inv-lemma[OF x-def, THEN ballE[where x = e]])
  apply(simp add: foundation18′)
by(simp add: e-mem)

lemma abs-rep-simp′ :
 assumes S-all-def : τ ⊨ δ S
   shows Abs-Set_base ⌊⌊⌈⌈Rep-Set_base (S τ)⌉⌉⌋⌋ = S τ
proof −
 have discr-eq-false-true : ⋀τ. (false τ = true τ) = False by(simp add: false-def true-def)
 show ?thesis
  apply(insert S-all-def, simp add: OclValid-def defined-def)
  apply(rule mp[OF Abs-Set_base-induct[where P = λS. (if S = ⊥ τ ∨ S = null τ
                                         then false τ else true τ) = true τ ⟶
                                        Abs-Set_base ⌊⌊⌈⌈Rep-Set_base S⌉⌉⌋⌋ = S]],
      rename-tac S′)
   apply(simp add: Abs-Set_base-inverse discr-eq-false-true)
   apply(case-tac S′) apply(simp add: bot-fun-def bot-Set_base-def)+
   apply(rename-tac S″, case-tac S″) apply(simp add: null-fun-def null-Set_base-def)+
 done
qed

lemma S-lift′ :
 assumes S-all-def : (τ :: 𝔄 st) ⊨ δ S
   shows ∃S′. (λa (-::𝔄 st). a) ‘ ⌈⌈Rep-Set_base (S τ)⌉⌉ = (λa (-::𝔄 st). ⌊a⌋) ‘ S′
  apply(rule-tac x = (λa. ⌈a⌉) ‘ ⌈⌈Rep-Set_base (S τ)⌉⌉ in exI)
  apply(simp only: image-comp[symmetric])
  apply(simp add: comp-def)
  apply(rule image-cong, fast)

  apply(drule Set-inv-lemma′[OF S-all-def])
by(case-tac x, (simp add: bot-option-def foundation18′)+)
```

lemma *invalid-set-OclNot-defined* [*simp*,*code-unfold*]:$\delta(invalid::({}'\mathfrak{A},{}'\alpha::null)\ Set) = false$ **by** *simp*
lemma *null-set-OclNot-defined* [*simp*,*code-unfold*]:$\delta(null::({}'\mathfrak{A},{}'\alpha::null)\ Set) = false$
by(*simp add*: *defined-def null-fun-def*)
lemma *invalid-set-valid* [*simp*,*code-unfold*]:$\upsilon(invalid::({}'\mathfrak{A},{}'\alpha::null)\ Set) = false$
by *simp*
lemma *null-set-valid* [*simp*,*code-unfold*]:$\upsilon(null::({}'\mathfrak{A},{}'\alpha::null)\ Set) = true$
apply(*simp add*: *valid-def null-fun-def bot-fun-def bot-Set*$_{base}$*-def null-Set*$_{base}$*-def*)
apply(*subst Abs-Set*$_{base}$*-inject*,*simp-all add*: *null-option-def bot-option-def*)
done

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

Constants on Sets

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

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

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

lemma *mtSet-rep-set*: $\lceil\lceil Rep\text{-}Set_{base}\ (Set\{\}\ \tau)\rceil\rceil = \{\}$
apply(*simp add*: *mtSet-def*, *subst Abs-Set*$_{base}$*-inverse*)
by(*simp add*: *bot-option-def*)+

lemma [*simp*,*code-unfold*]: *const Set*$\{\}$
by(*simp add*: *const-def mtSet-def*)

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

Operations

This part provides a collection of operators for the Set type.

Definition: OclIncluding **definition** *OclIncluding* :: $[('\mathfrak{A},'\alpha{::}null)\ Set,('\mathfrak{A},'\alpha)\ val] \Rightarrow ('\mathfrak{A},'\alpha)\ Set$
where $OclIncluding\ x\ y = (\lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau$
$\quad then\ Abs\text{-}Set_{base}\ \lfloor\lfloor\ \lceil\lceil Rep\text{-}Set_{base}\ (x\ \tau)\rceil\rceil\ \cup \{y\ \tau\}\ \rfloor\rfloor$
$\quad else\ invalid\ \tau\ )$
notation *OclIncluding* $(\text{-}\text{-}{>}including'(\text{-}'))$

interpretation *OclIncluding* : *profile-bin2 OclIncluding* $\lambda x\ y.\ Abs\text{-}Set_{base}\lfloor\lfloor\lceil\lceil Rep\text{-}Set_{base}\ x\rceil\rceil \cup \{y\}\rfloor\rfloor$
proof –
have $A : None \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$ **by**(*simp add*: *bot-option-def*)
have $B : \lfloor None\rfloor \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
  by(*simp add*: *null-option-def bot-option-def*)
have $C : \bigwedge x\ y.\ x \neq \bot \Longrightarrow x \neq null \Longrightarrow\ y \neq \bot\ \Longrightarrow$
  $\lfloor\lfloor insert\ y\ \lceil\lceil Rep\text{-}Set_{base}\ x\rceil\rceil\rfloor\rfloor \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
  by(*auto intro*!:*Set-inv-lemma*[*simplified OclValid-def*
    defined-def false-def true-def null-fun-def bot-fun-def])
  show *profile-bin2 OclIncluding* $(\lambda x\ y.\ Abs\text{-}Set_{base}\lfloor\lfloor\lceil\lceil Rep\text{-}Set_{base}\ x\rceil\rceil \cup \{y\}\rfloor\rfloor)$
  apply *unfold-locales*
  apply(*auto simp*:*OclIncluding-def bot-option-def null-option-def* $null\text{-}Set_{base}\text{-}def\ bot\text{-}Set_{base}\text{-}def$)
  apply(*erule-tac* $Q{=}Abs\text{-}Set_{base}\lfloor\lfloor insert\ y\ \lceil\lceil Rep\text{-}Set_{base}\ x\rceil\rceil\rfloor\rfloor = Abs\text{-}Set_{base}\ None$ **in** *contrapos-pp*)
  apply(*subst* $Abs\text{-}Set_{base}\text{-}inject$[*OF C A*])
   apply(*simp-all add*: $null\text{-}Set_{base}\text{-}def\ bot\text{-}Set_{base}\text{-}def$ *bot-option-def*)
  apply(*erule-tac* $Q{=}Abs\text{-}Set_{base}\lfloor\lfloor insert\ y\ \lceil\lceil Rep\text{-}Set_{base}\ x\rceil\rceil\rfloor\rfloor = Abs\text{-}Set_{base}\ \lfloor None\rfloor$ **in** *contrapos-pp*)
  apply(*subst* $Abs\text{-}Set_{base}\text{-}inject$[*OF C B*])
   apply(*simp-all add*: $null\text{-}Set_{base}\text{-}def\ bot\text{-}Set_{base}\text{-}def$ *bot-option-def*)
  done
qed

syntax
-OclFinset :: *args* $=>$ $('\mathfrak{A},'a{::}null)\ Set$ $(Set\{(\text{-})\})$
translations
$Set\{x, xs\} == CONST\ OclIncluding\ (Set\{xs\})\ x$
$Set\{x\}\quad == CONST\ OclIncluding\ (Set\{\})\ x$

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

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

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

notation *OclExcludes* $(\text{-}{-}{>}excludes'(\text{-}'))$

The case of the size definition is somewhat special, we admit explicitly in Featherweight 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 **definition** *OclSize* $:: ({}'\mathfrak{A},{}'\alpha::null)Set \Rightarrow {}'\mathfrak{A}\ Integer$

where $OclSize\ x = (\lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \land finite(\lceil\lceil Rep\text{-}Set_{base}\ (x\ \tau)\rceil\rceil)$
$\quad then\ \lfloor\lfloor\ int(card\ \lceil\lceil Rep\text{-}Set_{base}\ (x\ \tau)\rceil\rceil)\ \rfloor\rfloor$
$\quad else\ \bot\ )$

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

The following definition follows the requirement of the standard to treat null as neutral element of sets. It is a well-documented exception from the general strictness rule and the rule that the distinguished argument self should be non-null.

Definition: OclIsEmpty **definition** *OclIsEmpty* $:: ({}'\mathfrak{A},{}'\alpha::null)\ Set \Rightarrow {}'\mathfrak{A}\ Boolean$

where $OclIsEmpty\ x = ((\upsilon\ x\ and\ not\ (\delta\ x))\ or\ ((OclSize\ x) \doteq \mathbf{0}))$

notation *OclIsEmpty* $(\text{-}{-}{>}isEmpty'(')\ )$

Definition: OclNotEmpty **definition** *OclNotEmpty* $:: ({}'\mathfrak{A},{}'\alpha::null)\ Set \Rightarrow {}'\mathfrak{A}\ Boolean$

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

notation *OclNotEmpty* $(\text{-}{-}{>}notEmpty'(')\ )$

Definition: OclANY **definition** *OclANY* $:: [({}'\mathfrak{A},{}'\alpha::null)\ Set] \Rightarrow ({}'\mathfrak{A},{}'\alpha)\ val$

where $OclANY\ x = (\lambda\ \tau.\ if\ (\upsilon\ x)\ \tau = true\ \tau$
$\quad then\ if\ (\delta\ x\ and\ OclNotEmpty\ x)\ \tau = true\ \tau$
$\qquad then\ SOME\ y.\ y \in \lceil\lceil Rep\text{-}Set_{base}\ (x\ \tau)\rceil\rceil$
$\qquad else\ null\ \tau$
$\quad else\ \bot\ )$

notation *OclANY* $(\text{-}{-}{>}any'('))$

Definition: OclForall The definition of OclForall mimics the one of *op and*: OclForall is not a strict operation.

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

where $OclForall\ S\ P = (\lambda\ \tau.\ if\ (\delta\ S)\ \tau = true\ \tau$
$\quad then\ if\ (\exists x{\in}\lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil.\ P(\lambda\ \text{-}.\ x)\ \tau = false\ \tau)$
$\qquad then\ false\ \tau$
$\qquad else\ if\ (\exists x{\in}\lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil.\ P(\lambda\ \text{-}.\ x)\ \tau = invalid\ \tau)$
$\qquad\quad then\ invalid\ \tau$
$\qquad\quad else\ if\ (\exists x{\in}\lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil.\ P(\lambda\ \text{-}.\ x)\ \tau = null\ \tau)$
$\qquad\qquad then\ null\ \tau$
$\qquad\qquad else\ true\ \tau$
$\quad else\ \bot)$

syntax
$\text{-}OclForall :: [({}'\mathfrak{A},{}'\alpha::null)\ Set,id,({}'\mathfrak{A})Boolean] \Rightarrow {}'\mathfrak{A}\ Boolean \quad ((\text{-}){-}{>}forAll'(\text{-}|\text{-}'))$

translations

$X{-}{>}\mathit{forAll}(x \mid P) == \mathit{CONST\ OclForall\ X\ (\%x.\ P)}$

Definition: OclExists Like OclForall, OclExists is also not strict.

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

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

syntax

$\text{-}\mathit{OclExist} :: [({}'\mathfrak{A},{}'\alpha{::}\mathit{null})\ \mathit{Set},\mathit{id},({}'\mathfrak{A})\mathit{Boolean}] \Rightarrow {}'\mathfrak{A}\ \mathit{Boolean} \quad ((\text{-}){-}{>}\mathit{exists}'(\text{-}|\text{-}'))$

translations

$X{-}{>}\mathit{exists}(x \mid P) == \mathit{CONST\ OclExists\ X\ (\%x.\ P)}$

Definition: OclIterate **definition** $\mathit{OclIterate} :: [({}'\mathfrak{A},{}'\alpha{::}\mathit{null})\ \mathit{Set},({}'\mathfrak{A},{}'\beta{::}\mathit{null})\mathit{val},$
$({}'\mathfrak{A},{}'\alpha)\mathit{val}{\Rightarrow}({}'\mathfrak{A},{}'\beta)\mathit{val}{\Rightarrow}({}'\mathfrak{A},{}'\beta)\mathit{val}] \Rightarrow ({}'\mathfrak{A},{}'\beta)\mathit{val}$

where $\mathit{OclIterate\ S\ A\ F} = (\lambda\ \tau.\ \mathit{if}\ (\delta\ S)\ \tau = \mathit{true}\ \tau \wedge (\upsilon\ A)\ \tau = \mathit{true}\ \tau \wedge \mathit{finite}\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (S\ \tau)\rceil\rceil$
$\mathit{then}\ (\mathit{Finite\text{-}Set.fold}\ (F)\ (A)\ ((\lambda a\ \tau.\ a)\ `\ \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (S\ \tau)\rceil\rceil))\tau$
$\mathit{else}\ \bot)$

syntax

$\text{-}\mathit{OclIterate}\ :: [({}'\mathfrak{A},{}'\alpha{::}\mathit{null})\ \mathit{Set},\ \mathit{idt},\ \mathit{idt},\ {}'\alpha,\ {}'\beta] => ({}'\mathfrak{A},{}'\gamma)\mathit{val}$
$(\text{-}\ {-}{>}\mathit{iterate}'(\text{-};\text{-}{=}\text{-}\mid\text{-}')\ )$

translations

$X{-}{>}\mathit{iterate}(a;\ x = A \mid P) == \mathit{CONST\ OclIterate\ X\ A\ (\%a.\ (\%\ x.\ P))}$

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

where $\mathit{OclSelect\ S\ P} = (\lambda\tau.\ \mathit{if}\ (\delta\ S)\ \tau = \mathit{true}\ \tau$
$\mathit{then\ if}\ (\exists x{\in}\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (S\ \tau)\rceil\rceil.\ P(\lambda\ \text{-}.\ x)\ \tau = \mathit{invalid}\ \tau)$
$\mathit{then\ invalid}\ \tau$
$\mathit{else\ Abs\text{-}Set}_{base}\ \lfloor\lfloor\{x{\in}\lceil\lceil\ \mathit{Rep\text{-}Set}_{base}\ (S\ \tau)\rceil\rceil.\ P\ (\lambda\text{-}.\ x)\ \tau \neq \mathit{false}\ \tau\}\rfloor\rfloor$
$\mathit{else\ invalid}\ \tau)$

syntax

$\text{-}\mathit{OclSelect} :: [({}'\mathfrak{A},{}'\alpha{::}\mathit{null})\ \mathit{Set},\mathit{id},({}'\mathfrak{A})\mathit{Boolean}] \Rightarrow {}'\mathfrak{A}\ \mathit{Boolean} \quad ((\text{-}){-}{>}\mathit{select}'(\text{-}|\text{-}'))$

translations

$X{-}{>}\mathit{select}(x \mid P) == \mathit{CONST\ OclSelect\ X\ (\%\ x.\ P)}$

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

where $\mathit{OclReject\ S\ P} = \mathit{OclSelect\ S}\ (\mathit{not\ o\ P})$

syntax

$\text{-}\mathit{OclReject} :: [({}'\mathfrak{A},{}'\alpha{::}\mathit{null})\ \mathit{Set},\mathit{id},({}'\mathfrak{A})\mathit{Boolean}] \Rightarrow {}'\mathfrak{A}\ \mathit{Boolean} \quad ((\text{-}){-}{>}\mathit{reject}'(\text{-}|\text{-}'))$

translations

$X{-}{>}\mathit{reject}(x \mid P) == \mathit{CONST\ OclReject\ X\ (\%\ x.\ P)}$

Definition (futur operators) **consts**

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

$\mathit{OclSum}\quad ::\ ({}'\mathfrak{A},{}'\alpha{::}\mathit{null})\ \mathit{Set} \Rightarrow {}'\mathfrak{A}\ \mathit{Integer}$

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

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

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

notation
OclCount (-->*count*'(-'))
notation
OclSum (-->*sum*'('))
notation
OclIncludesAll (-->*includesAll*'(-'))
notation
OclExcludesAll (-->*excludesAll*'(-'))
notation
OclComplement (-->*complement*'('))
notation
OclUnion (-->*union*'(-'))
notation
OclIntersection(-->*intersection*'(-'))

Validity and Definedness Properties

OclIncluding

lemma *OclIncluding-defined-args-valid*:
$(\tau \models \delta(X{-}{>}\mathit{including}(x))) = ((\tau \models(\delta\ X)) \land (\tau \models(\upsilon\ x)))$
by(*simp add*: *foundation10'*)

lemma *OclIncluding-valid-args-valid*:
$(\tau \models \upsilon(X{-}{>}\mathit{including}(x))) = ((\tau \models(\delta\ X)) \land (\tau \models(\upsilon\ x)))$
by (*metis* (*hide-lams*, *no-types*) *OclIncluding.def-valid-then-def OclIncluding-defined-args-valid*)

lemma *OclIncluding-defined-args-valid'*[*simp*,*code-unfold*]:
$\delta(X{-}{>}\mathit{including}(x)) = ((\delta\ X)\ \mathit{and}\ (\upsilon\ x))$
by *simp*

lemma *OclIncluding-valid-args-valid''*[*simp*,*code-unfold*]:
$\upsilon(X{-}{>}\mathit{including}(x)) = ((\delta\ X)\ \mathit{and}\ (\upsilon\ x))$
by(*auto intro*!: *transform2-rev simp*:*OclIncluding-valid-args-valid foundation10 defined-and-I*)

OclExcluding

lemma *OclExcluding-defined-args-valid*:
$(\tau \models \delta(X{-}{>}\mathit{excluding}(x))) = ((\tau \models(\delta\ X)) \land (\tau \models(\upsilon\ x)))$
proof −
have $A : \bot \in \{X.\ X = \mathit{bot} \lor X = \mathit{null} \lor (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq \mathit{bot})\}$ **by**(*simp add*: *bot-option-def*)
have $B : \lfloor\bot\rfloor \in \{X.\ X = \mathit{bot} \lor X = \mathit{null} \lor (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq \mathit{bot})\}$
by(*simp add*: *null-option-def bot-option-def*)
have $C : (\tau \models(\delta\ X)) \Longrightarrow (\tau \models(\upsilon\ x)) \Longrightarrow$
$\lfloor\lfloor\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil - \{x\ \tau\}\rfloor\rfloor \in \{X.\ X = \mathit{bot} \lor X = \mathit{null} \lor (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq \mathit{bot})\}$

by(*frule Set-inv-lemma*, *simp add*: *foundation18 invalid-def*)
have *D*: $(\tau \models \delta(X{-}{>}\mathit{excluding}(x))) \Longrightarrow ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
by(*auto simp*: *OclExcluding-def OclValid-def true-def valid-def false-def StrongEq-def*
defined-def invalid-def bot-fun-def null-fun-def
split: *bool.split-asm HOL.split-if-asm option.split*)
have *E*: $(\tau \models(\delta\ X)) \Longrightarrow (\tau \models(\upsilon\ x)) \Longrightarrow (\tau \models \delta(X{-}{>}\mathit{excluding}(x)))$
apply(*subst OclExcluding-def*, *subst OclValid-def*, *subst defined-def*)
apply(*auto simp*: *OclValid-def null-Set*$_{base}$*-def bot-Set*$_{base}$*-def null-fun-def bot-fun-def*)
apply(*frule Abs-Set*$_{base}$*-inject*[*OF C A*, *simplified OclValid-def*, *THEN iffD1*],
simp-all add: *bot-option-def*)
apply(*frule Abs-Set*$_{base}$*-inject*[*OF C B*, *simplified OclValid-def*, *THEN iffD1*],
simp-all add: *bot-option-def*)
done
show *?thesis* **by**(*auto dest*:*D intro*:*E*)
qed

lemma *OclExcluding-valid-args-valid*:
$(\tau \models \upsilon(X{-}{>}\mathit{excluding}(x))) = ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
proof –
have *D*: $(\tau \models \upsilon(X{-}{>}\mathit{excluding}(x))) \Longrightarrow ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
by(*auto simp*: *OclExcluding-def OclValid-def true-def valid-def false-def StrongEq-def*
defined-def invalid-def bot-fun-def null-fun-def
split: *bool.split-asm HOL.split-if-asm option.split*)
have *E*: $(\tau \models(\delta\ X)) \Longrightarrow (\tau \models(\upsilon\ x)) \Longrightarrow (\tau \models \upsilon(X{-}{>}\mathit{excluding}(x)))$
by(*simp add*: *foundation20 OclExcluding-defined-args-valid*)
show *?thesis* **by**(*auto dest*:*D intro*:*E*)
qed

lemma *OclExcluding-valid-args-valid′*[*simp*,*code-unfold*]:
$\delta(X{-}{>}\mathit{excluding}(x)) = ((\delta\ X)\ \mathit{and}\ (\upsilon\ x))$
by(*auto intro*!: *transform2-rev simp*:*OclExcluding-defined-args-valid foundation10 defined-and-I*)

lemma *OclExcluding-valid-args-valid″*[*simp*,*code-unfold*]:
$\upsilon(X{-}{>}\mathit{excluding}(x)) = ((\delta\ X)\ \mathit{and}\ (\upsilon\ x))$
by(*auto intro*!: *transform2-rev simp*:*OclExcluding-valid-args-valid foundation10 defined-and-I*)

OclIncludes

lemma *OclIncludes-defined-args-valid*:
$(\tau \models \delta(X{-}{>}\mathit{includes}(x))) = ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
proof –
have *A*: $(\tau \models \delta(X{-}{>}\mathit{includes}(x))) \Longrightarrow ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
by(*auto simp*: *OclIncludes-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 *B*: $(\tau \models(\delta\ X)) \Longrightarrow (\tau \models(\upsilon\ x)) \Longrightarrow (\tau \models \delta(X{-}{>}\mathit{includes}(x)))$

```
        by(auto simp: OclIncludes-def OclValid-def true-def false-def StrongEq-def
                    defined-def invalid-def valid-def bot-fun-def null-fun-def
                    bot-option-def null-option-def
                split: bool.split-asm HOL.split-if-asm option.split)
show ?thesis by(auto dest:A intro:B)
qed

lemma OclIncludes-valid-args-valid:
(τ ⊨ υ(X−>includes(x))) = ((τ ⊨(δ X)) ∧ (τ ⊨(υ x)))
proof −
 have A: (τ ⊨ υ(X−>includes(x))) ⟹ ((τ ⊨(δ X)) ∧ (τ ⊨(υ x)))
        by(auto simp: OclIncludes-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 B: (τ ⊨(δ X)) ⟹ (τ ⊨(υ x)) ⟹ (τ ⊨ υ(X−>includes(x)))
        by(auto simp: OclIncludes-def OclValid-def true-def false-def StrongEq-def
                    defined-def invalid-def valid-def bot-fun-def null-fun-def
                    bot-option-def null-option-def
                split: bool.split-asm HOL.split-if-asm option.split)
show ?thesis by(auto dest:A intro:B)
qed

lemma OclIncludes-valid-args-valid'[simp,code-unfold]:
δ(X−>includes(x)) = ((δ X) and (υ x))
by(auto intro!: transform2-rev simp:OclIncludes-defined-args-valid foundation10 defined-and-I)

lemma OclIncludes-valid-args-valid''[simp,code-unfold]:
υ(X−>includes(x)) = ((δ X) and (υ x))
by(auto intro!: transform2-rev simp:OclIncludes-valid-args-valid foundation10 defined-and-I)
```

OclExcludes

```
lemma OclExcludes-defined-args-valid:
(τ ⊨ δ(X−>excludes(x))) = ((τ ⊨(δ X)) ∧ (τ ⊨(υ x)))
by (metis (hide-lams, no-types)
    OclExcludes-def OclAnd-idem OclOr-def OclOr-idem defined-not-I OclIncludes-defined-args-valid)

lemma OclExcludes-valid-args-valid:
(τ ⊨ υ(X−>excludes(x))) = ((τ ⊨(δ X)) ∧ (τ ⊨(υ x)))
by (metis (hide-lams, no-types)
    OclExcludes-def OclAnd-idem OclOr-def OclOr-idem valid-not-I OclIncludes-valid-args-valid)

lemma OclExcludes-valid-args-valid'[simp,code-unfold]:
δ(X−>excludes(x)) = ((δ X) and (υ x))
by(auto intro!: transform2-rev simp:OclExcludes-defined-args-valid foundation10 defined-and-I)

lemma OclExcludes-valid-args-valid''[simp,code-unfold]:
υ(X−>excludes(x)) = ((δ X) and (υ x))
by(auto intro!: transform2-rev simp:OclExcludes-valid-args-valid foundation10 defined-and-I)
```

OclSize

lemma *OclSize-defined-args-valid*: $\tau \models \delta\ (X{-}{>}size()) \Longrightarrow \tau \models \delta\ X$
by(*auto simp*: *OclSize-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*)

lemma *OclSize-infinite*:
assumes *non-finite*:$\tau \models not(\delta(S{-}{>}size()))$
shows $(\tau \models not(\delta(S))) \vee \neg$ *finite* $\lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil$
apply(*insert non-finite*, *simp*)
apply(*rule impI*)
apply(*simp add*: *OclSize-def OclValid-def defined-def*)
apply(*case-tac finite* $\lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil$,
simp-all add:*null-fun-def null-option-def bot-fun-def bot-option-def*)
done

lemma $\tau \models \delta\ X \Longrightarrow \neg$ *finite* $\lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil \Longrightarrow \neg\ \tau \models \delta\ (X{-}{>}size())$
by(*simp add*: *OclSize-def OclValid-def defined-def bot-fun-def false-def true-def*)

lemma *size-defined*:
assumes *X-finite*: $\bigwedge\tau.$ *finite* $\lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil$
shows $\delta\ (X{-}{>}size()) = \delta\ X$
apply(*rule ext*, *simp add*: *cp-defined*[*of* $X{-}{>}size()$] *OclSize-def*)
apply(*simp add*: *defined-def bot-option-def bot-fun-def null-option-def null-fun-def X-finite*)
done

lemma *size-defined'*:
assumes *X-finite*:*finite* $\lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil$
shows $(\tau \models \delta\ (X{-}{>}size())) = (\tau \models \delta\ X)$
apply(*simp add*: *cp-defined*[*of* $X{-}{>}size()$] *OclSize-def OclValid-def*)
apply(*simp add*: *defined-def bot-option-def bot-fun-def null-option-def null-fun-def X-finite*)
done

OclIsEmpty

lemma *OclIsEmpty-defined-args-valid*:$\tau \models \delta\ (X{-}{>}isEmpty()) \Longrightarrow \tau \models \upsilon\ X$
apply(*auto simp*: *OclIsEmpty-def OclValid-def defined-def valid-def false-def true-def*
bot-fun-def null-fun-def OclAnd-def OclOr-def OclNot-def
split: *split-if-asm*)
apply(*case-tac* $(X{-}{>}size() \doteq \mathbf{0})\ \tau$, *simp add*: *bot-option-def*, *simp*, *rename-tac x*)
apply(*case-tac x*, *simp add*: *null-option-def bot-option-def*, *simp*)
apply(*simp add*: *OclSize-def* $StrictRefEq_{Integer}$ *valid-def*)
by (*metis* (*hide-lams*, *no-types*)
bot-fun-def OclValid-def defined-def foundation2 invalid-def)

lemma $\tau \models \delta\ (null{-}{>}isEmpty())$
by(*auto simp*: *OclIsEmpty-def OclValid-def defined-def valid-def false-def true-def*
bot-fun-def null-fun-def OclAnd-def OclOr-def OclNot-def null-is-valid
split: *split-if-asm*)

lemma *OclIsEmpty-infinite*: $\tau \models \delta\ X \Longrightarrow \neg\ \mathit{finite}\ \lceil\lceil \mathit{Rep\text{-}Set}_{\mathit{base}}\ (X\ \tau)\rceil\rceil \Longrightarrow \neg\ \tau \models \delta\ (X{-}{>}\mathit{isEmpty}())$
apply(*auto simp*: *OclIsEmpty-def OclValid-def defined-def valid-def false-def true-def*
bot-fun-def null-fun-def OclAnd-def OclOr-def OclNot-def
split: *split-if-asm*)
apply(*case-tac* $(X{-}{>}\mathit{size}() \doteq \mathbf{0})\ \tau$, *simp add*: *bot-option-def*, *simp*, *rename-tac x*)
apply(*case-tac x*, *simp add*: *null-option-def bot-option-def*, *simp*)
by(*simp add*: *OclSize-def* $\mathit{StrictRefEq}_{\mathit{Integer}}$ *valid-def bot-fun-def false-def true-def invalid-def*)

OclNotEmpty

lemma *OclNotEmpty-defined-args-valid*:$\tau \models \delta\ (X{-}{>}\mathit{notEmpty}()) \Longrightarrow \tau \models \upsilon\ X$
by (*metis* (*hide-lams*, *no-types*) *OclNotEmpty-def OclNot-defargs OclNot-not foundation6 foundation9*
OclIsEmpty-defined-args-valid)

lemma $\tau \models \delta\ (\mathit{null}{-}{>}\mathit{notEmpty}())$
by (*metis* (*hide-lams*, *no-types*) *OclNotEmpty-def OclAnd-false1 OclAnd-idem OclIsEmpty-def*
OclNot3 OclNot4 OclOr-def defined2 defined4 transform1 valid2)

lemma *OclNotEmpty-infinite*: $\tau \models \delta\ X \Longrightarrow \neg\ \mathit{finite}\ \lceil\lceil \mathit{Rep\text{-}Set}_{\mathit{base}}\ (X\ \tau)\rceil\rceil \Longrightarrow \neg\ \tau \models \delta\ (X{-}{>}\mathit{notEmpty}())$
apply(*simp add*: *OclNotEmpty-def*)
apply(*drule OclIsEmpty-infinite*, *simp*)
by (*metis OclNot-defargs OclNot-not foundation6 foundation9*)

lemma *OclNotEmpty-has-elt* : $\tau \models \delta\ X \Longrightarrow$
$\tau \models X{-}{>}\mathit{notEmpty}() \Longrightarrow$
$\exists e.\ e \in \lceil\lceil \mathit{Rep\text{-}Set}_{\mathit{base}}\ (X\ \tau)\rceil\rceil$
apply(*simp add*: *OclNotEmpty-def OclIsEmpty-def deMorgan1 deMorgan2*, *drule foundation5*)
apply(*subst* (*asm*) (*2*) *OclNot-def*,
simp add: *OclValid-def* $\mathit{StrictRefEq}_{\mathit{Integer}}$ *StrongEq-def*
split: *split-if-asm*)
prefer *2*
apply(*simp add*: *invalid-def bot-option-def true-def*)
apply(*simp add*: *OclSize-def valid-def split*: *split-if-asm*,
simp-all add: *false-def true-def bot-option-def bot-fun-def OclInt0-def*)
by (*metis equals0I*)

OclANY

lemma *OclANY-defined-args-valid*: $\tau \models \delta\ (X{-}{>}\mathit{any}()) \Longrightarrow \tau \models \delta\ X$
by(*auto simp*: *OclANY-def OclValid-def true-def valid-def false-def StrongEq-def*
defined-def invalid-def bot-fun-def null-fun-def OclAnd-def
split: *bool.split-asm HOL.split-if-asm option.split*)

lemma $\tau \models \delta\ X \Longrightarrow \tau \models X{-}{>}\mathit{isEmpty}() \Longrightarrow \neg\ \tau \models \delta\ (X{-}{>}\mathit{any}())$
apply(*simp add*: *OclANY-def OclValid-def*)
apply(*subst cp-defined*, *subst cp-OclAnd*, *simp add*: *OclNotEmpty-def*, *subst* (*1 2*) *cp-OclNot*,
simp add: *cp-OclNot*[*symmetric*] *cp-OclAnd*[*symmetric*] *cp-defined*[*symmetric*],
simp add: *false-def true-def*)
by(*drule foundation20*[*simplified OclValid-def true-def*], *simp*)

```
lemma OclANY-valid-args-valid:
(τ ⊨ υ(X−>any())) = (τ ⊨ υ X)
proof −
 have A: (τ ⊨ υ(X−>any())) ⟹ ((τ ⊨(υ X)))
        by(auto simp: OclANY-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 B: (τ ⊨(υ X)) ⟹ (τ ⊨ υ(X−>any()))
         apply(auto simp: OclANY-def OclValid-def true-def false-def StrongEq-def
                      defined-def invalid-def valid-def bot-fun-def null-fun-def
                      bot-option-def null-option-def null-is-valid
                      OclAnd-def
                  split: bool.split-asm HOL.split-if-asm option.split)
         apply(frule Set-inv-lemma[OF foundation16[THEN iffD2], OF conjI], simp)
         apply(subgoal-tac (δ X) τ = true τ)
          prefer 2
          apply (metis (hide-lams, no-types) OclValid-def foundation16)
         apply(simp add: true-def,
              drule OclNotEmpty-has-elt[simplified OclValid-def true-def], simp)
        by(erule exE,
          insert someI2[where Q = λx. x ≠ ⊥ and P = λy. y ∈ ⌈⌈Rep-Set_base (X τ)⌉⌉],
          simp)
 show ?thesis by(auto dest:A intro:B)
qed

lemma OclANY-valid-args-valid''[simp,code-unfold]:
υ(X−>any()) = (υ X)
by(auto intro!: OclANY-valid-args-valid transform2-rev)
```

Execution with Invalid or Null or Infinite Set as Argument

OclIncluding

```
lemma OclIncluding-invalid[simp,code-unfold]:(invalid−>including(x)) = invalid
by(simp add: bot-fun-def OclIncluding-def invalid-def defined-def valid-def false-def true-def)

lemma OclIncluding-invalid-args[simp,code-unfold]:(X−>including(invalid)) = invalid
by(simp add: OclIncluding-def invalid-def bot-fun-def defined-def valid-def false-def true-def)

lemma OclIncluding-null[simp,code-unfold]:(null−>including(x)) = invalid
by(simp add: OclIncluding-def invalid-def bot-fun-def defined-def valid-def false-def true-def)
```

OclExcluding

```
lemma OclExcluding-invalid[simp,code-unfold]:(invalid−>excluding(x)) = invalid
by(simp add: bot-fun-def OclExcluding-def invalid-def defined-def valid-def false-def true-def)

lemma OclExcluding-invalid-args[simp,code-unfold]:(X−>excluding(invalid)) = invalid
by(simp add: OclExcluding-def invalid-def bot-fun-def defined-def valid-def false-def true-def)
```

lemma *OclExcluding-null*[*simp*,*code-unfold*]:(*null*−>*excluding*(*x*)) = *invalid*
by(*simp add*: *OclExcluding-def invalid-def bot-fun-def defined-def valid-def false-def true-def*)

OclIncludes

lemma *OclIncludes-invalid*[*simp*,*code-unfold*]:(*invalid*−>*includes*(*x*)) = *invalid*
by(*simp add*: *bot-fun-def OclIncludes-def invalid-def defined-def valid-def false-def true-def*)

lemma *OclIncludes-invalid-args*[*simp*,*code-unfold*]:(*X*−>*includes*(*invalid*)) = *invalid*
by(*simp add*: *OclIncludes-def invalid-def bot-fun-def defined-def valid-def false-def true-def*)

lemma *OclIncludes-null*[*simp*,*code-unfold*]:(*null*−>*includes*(*x*)) = *invalid*
by(*simp add*: *OclIncludes-def invalid-def bot-fun-def defined-def valid-def false-def true-def*)

OclExcludes

lemma *OclExcludes-invalid*[*simp*,*code-unfold*]:(*invalid*−>*excludes*(*x*)) = *invalid*
by(*simp add*: *OclExcludes-def OclNot-def*, *simp add*: *invalid-def bot-option-def*)

lemma *OclExcludes-invalid-args*[*simp*,*code-unfold*]:(*X*−>*excludes*(*invalid*)) = *invalid*
by(*simp add*: *OclExcludes-def OclNot-def*, *simp add*: *invalid-def bot-option-def*)

lemma *OclExcludes-null*[*simp*,*code-unfold*]:(*null*−>*excludes*(*x*)) = *invalid*
by(*simp add*: *OclExcludes-def OclNot-def*, *simp add*: *invalid-def bot-option-def*)

OclSize

lemma *OclSize-invalid*[*simp*,*code-unfold*]:(*invalid*−>*size*()) = *invalid*
by(*simp add*: *bot-fun-def OclSize-def invalid-def defined-def valid-def false-def true-def*)

lemma *OclSize-null*[*simp*,*code-unfold*]:(*null*−>*size*()) = *invalid*
by(*rule ext*,
  simp add: *bot-fun-def null-fun-def null-is-valid OclSize-def*
    invalid-def defined-def valid-def false-def true-def)

OclIsEmpty

lemma *OclIsEmpty-invalid*[*simp*,*code-unfold*]:(*invalid*−>*isEmpty*()) = *invalid*
by(*simp add*: *OclIsEmpty-def*)

lemma *OclIsEmpty-null*[*simp*,*code-unfold*]:(*null*−>*isEmpty*()) = *true*
by(*simp add*: *OclIsEmpty-def*)

OclNotEmpty

lemma *OclNotEmpty-invalid*[*simp*,*code-unfold*]:(*invalid*−>*notEmpty*()) = *invalid*
by(*simp add*: *OclNotEmpty-def*)

lemma *OclNotEmpty-null*[*simp*,*code-unfold*]:(*null*−>*notEmpty*()) = *false*
by(*simp add*: *OclNotEmpty-def*)

OclANY

lemma *OclANY-invalid*[*simp*,*code-unfold*]:(*invalid*−>*any*()) = *invalid*

by(*simp add*: *bot-fun-def OclANY-def invalid-def defined-def valid-def false-def true-def*)

lemma *OclANY-null*[*simp*,*code-unfold*]:(*null*−>*any*()) = *null*
by(*simp add*: *OclANY-def false-def true-def*)

OclForall

lemma *OclForall-invalid*[*simp*,*code-unfold*]:*invalid*−>*forAll*(*a*| *P a*) = *invalid*
by(*simp add*: *bot-fun-def invalid-def OclForall-def defined-def valid-def false-def true-def*)

lemma *OclForall-null*[*simp*,*code-unfold*]:*null*−>*forAll*(*a* | *P a*) = *invalid*
by(*simp add*: *bot-fun-def invalid-def OclForall-def defined-def valid-def false-def true-def*)

OclExists

lemma *OclExists-invalid*[*simp*,*code-unfold*]:*invalid*−>*exists*(*a*| *P a*) = *invalid*
by(*simp add*: *OclExists-def*)

lemma *OclExists-null*[*simp*,*code-unfold*]:*null*−>*exists*(*a* | *P a*) = *invalid*
by(*simp add*: *OclExists-def*)

OclIterate

lemma *OclIterate-invalid*[*simp*,*code-unfold*]:*invalid*−>*iterate*(*a*; *x* = *A* | *P a x*) = *invalid*
by(*simp add*: *bot-fun-def invalid-def OclIterate-def defined-def valid-def false-def true-def*)

lemma *OclIterate-null*[*simp*,*code-unfold*]:*null*−>*iterate*(*a*; *x* = *A* | *P a x*) = *invalid*
by(*simp add*: *bot-fun-def invalid-def OclIterate-def defined-def valid-def false-def true-def*)

lemma *OclIterate-invalid-args*[*simp*,*code-unfold*]:*S*−>*iterate*(*a*; *x* = *invalid* | *P a x*) = *invalid*
by(*simp add*: *bot-fun-def invalid-def OclIterate-def defined-def valid-def false-def true-def*)

An open question is this ...

lemma *S*−>*iterate*(*a*; *x* = *null* | *P a x*) = *invalid*
oops

lemma *OclIterate-infinite*:
assumes *non-finite*: τ ⊨ *not*(δ(*S*−>*size*()))
shows (*OclIterate S A F*) τ = *invalid* τ
apply(*insert non-finite* [*THEN OclSize-infinite*])
apply(*subst* (*asm*) *foundation9*, *simp*)
by(*metis OclIterate-def OclValid-def invalid-def*)

OclSelect

lemma *OclSelect-invalid*[*simp*,*code-unfold*]:*invalid*−>*select*(*a* | *P a*) = *invalid*
by(*simp add*: *bot-fun-def invalid-def OclSelect-def defined-def valid-def false-def true-def*)

lemma *OclSelect-null*[*simp*,*code-unfold*]:*null*−>*select*(*a* | *P a*) = *invalid*
by(*simp add*: *bot-fun-def invalid-def OclSelect-def defined-def valid-def false-def true-def*)

OclReject

lemma *OclReject-invalid*[*simp*,*code-unfold*]:*invalid*−>*reject*(*a* | *P a*) = *invalid*
by(*simp add*: *OclReject-def*)

lemma *OclReject-null*[*simp*,*code-unfold*]:*null*−>*reject*(*a* | *P a*) = *invalid*
by(*simp add*: *OclReject-def*)

Context Passing **lemma** *cp-OclIncluding*:
(*X*−>*including*(*x*)) τ = ((λ -. *X* τ)−>*including*(λ -. *x* τ)) τ
by(*auto simp*: *OclIncluding-def StrongEq-def invalid-def*
cp-defined[*symmetric*] *cp-valid*[*symmetric*])

lemma *cp-OclExcluding*:
(*X*−>*excluding*(*x*)) τ = ((λ -. *X* τ)−>*excluding*(λ -. *x* τ)) τ
by(*auto simp*: *OclExcluding-def StrongEq-def invalid-def*
cp-defined[*symmetric*] *cp-valid*[*symmetric*])

lemma *cp-OclIncludes*:
(*X*−>*includes*(*x*)) τ = ((λ -. *X* τ)−>*includes*(λ -. *x* τ)) τ
by(*auto simp*: *OclIncludes-def StrongEq-def invalid-def*
cp-defined[*symmetric*] *cp-valid*[*symmetric*])

lemma *cp-OclIncludes1*:
(*X*−>*includes*(*x*)) τ = (*X*−>*includes*(λ -. *x* τ)) τ
by(*auto simp*: *OclIncludes-def StrongEq-def invalid-def*
cp-defined[*symmetric*] *cp-valid*[*symmetric*])

lemma *cp-OclExcludes*:
(*X*−>*excludes*(*x*)) τ = ((λ -. *X* τ)−>*excludes*(λ -. *x* τ)) τ
by(*simp add*: *OclExcludes-def OclNot-def*, *subst cp-OclIncludes*, *simp*)

lemma *cp-OclSize*: *X*−>*size*() τ = ((λ-. *X* τ)−>*size*()) τ
by(*simp add*: *OclSize-def cp-defined*[*symmetric*])

lemma *cp-OclIsEmpty*: *X*−>*isEmpty*() τ = ((λ-. *X* τ)−>*isEmpty*()) τ
apply(*simp only*: *OclIsEmpty-def*)
apply(*subst* (2) *cp-OclOr*,
subst cp-OclAnd,
subst cp-OclNot,
subst $StrictRefEq_{Integer}$*.cp0*)
by(*simp add*: *cp-defined*[*symmetric*] *cp-valid*[*symmetric*] $StrictRefEq_{Integer}$*.cp0*[*symmetric*]
cp-OclSize[*symmetric*] *cp-OclNot*[*symmetric*] *cp-OclAnd*[*symmetric*] *cp-OclOr*[*symmetric*])

lemma *cp-OclNotEmpty*: *X*−>*notEmpty*() τ = ((λ-. *X* τ)−>*notEmpty*()) τ
apply(*simp only*: *OclNotEmpty-def*)
apply(*subst* (2) *cp-OclNot*)
by(*simp add*: *cp-OclNot*[*symmetric*] *cp-OclIsEmpty*[*symmetric*])

```
lemma cp-OclANY: X−>any() τ = ((λ-. X τ)−>any()) τ
 apply(simp only: OclANY-def)
 apply(subst (2) cp-OclAnd)
by(simp only: cp-OclAnd[symmetric] cp-defined[symmetric] cp-valid[symmetric]
          cp-OclNotEmpty[symmetric])

lemma cp-OclForall:
(S−>forAll(x | P x)) τ = ((λ -. S τ)−>forAll(x | P (λ -. x τ))) τ
by(simp add: OclForall-def cp-defined[symmetric])


lemma cp-OclForall1 [simp,intro!]:
cp S ⟹ cp (λX. ((S X)−>forAll(x | P x)))
apply(simp add: cp-def)
apply(erule exE, rule exI, intro allI)
apply(erule-tac x=X in allE)
by(subst cp-OclForall, simp)

lemma
cp (λX St x. P (λτ. x) X St) ⟹ cp S ⟹ cp (λX. (S X)−>forAll(x|P x X))
apply(simp only: cp-def)
oops

lemma
cp S ⟹
 (⋀ x. cp(P x)) ⟹
 cp(λX. ((S X)−>forAll(x | P x X)))
oops



lemma cp-OclExists:
(S−>exists(x | P x)) τ = ((λ -. S τ)−>exists(x | P (λ -. x τ))) τ
by(simp add: OclExists-def OclNot-def, subst cp-OclForall, simp)


lemma cp-OclExists1 [simp,intro!]:
cp S ⟹ cp (λX. ((S X)−>exists(x | P x)))
apply(simp add: cp-def)
apply(erule exE, rule exI, intro allI)
apply(erule-tac x=X in allE)
by(subst cp-OclExists,simp)

lemma cp-OclIterate: (X−>iterate(a; x = A | P a x)) τ =
          ((λ -. X τ)−>iterate(a; x = A | P a x)) τ
by(simp add: OclIterate-def cp-defined[symmetric])
```

```
lemma cp-OclSelect: (X−>select(a | P a)) τ =
          ((λ -. X τ)−>select(a | P a)) τ
by(simp add: OclSelect-def cp-defined[symmetric])

lemma cp-OclReject: (X−>reject(a | P a)) τ =
          ((λ -. X τ)−>reject(a | P a)) τ
by(simp add: OclReject-def, subst cp-OclSelect, simp)

lemmas cp-intro''_Set[intro!,simp,code-unfold] =
     cp-OclIncluding [THEN allI[THEN allI[THEN allI[THEN cpI2]], of OclIncluding]]
     cp-OclExcluding [THEN allI[THEN allI[THEN allI[THEN cpI2]], of OclExcluding]]
     cp-OclIncludes  [THEN allI[THEN allI[THEN allI[THEN cpI2]], of OclIncludes]]
     cp-OclExcludes  [THEN allI[THEN allI[THEN allI[THEN cpI2]], of OclExcludes]]
     cp-OclSize      [THEN allI[THEN allI[THEN cpI1], of OclSize]]
     cp-OclIsEmpty   [THEN allI[THEN allI[THEN cpI1], of OclIsEmpty]]
     cp-OclNotEmpty  [THEN allI[THEN allI[THEN cpI1], of OclNotEmpty]]
     cp-OclANY       [THEN allI[THEN allI[THEN cpI1], of OclANY]]

Const   lemma const-OclIncluding[simp,code-unfold] :
 assumes const-x : const x
    and const-S : const S
  shows  const (S−>including(x))
  proof −
    have A:⋀τ τ'. ¬ (τ ⊨ υ x) ⟹ (S−>including(x) τ) = (S−>including(x) τ')
          apply(simp add: foundation18)
          apply(erule const-subst[OF const-x const-invalid],simp-all)
          by(rule const-charn[OF const-invalid])
    have B: ⋀ τ τ'. ¬ (τ ⊨ δ S) ⟹ (S−>including(x) τ) = (S−>including(x) τ')
          apply(simp add: foundation16', elim disjE)
          apply(erule const-subst[OF const-S const-invalid],simp-all)
          apply(rule const-charn[OF const-invalid])
          apply(erule const-subst[OF const-S const-null],simp-all)
          by(rule const-charn[OF const-invalid])
    show ?thesis
      apply(simp only: const-def,intro allI, rename-tac τ τ')
      apply(case-tac ¬ (τ ⊨ υ x), simp add: A)
      apply(case-tac ¬ (τ ⊨ δ S), simp-all add: B)
      apply(frule-tac τ'1= τ' in  const-OclValid2[OF const-x, THEN iffD1])
      apply(frule-tac τ'1= τ' in  const-OclValid1[OF const-S, THEN iffD1])
      apply(simp add: OclIncluding-def OclValid-def)
      apply(subst const-charn[OF const-x])
      apply(subst const-charn[OF const-S])
      by simp
qed
```

Strict Equality

Definition After the part of foundational operations on sets, we detail here equality on sets. Strong equality is inherited from the OCL core, but we have to consider the case of the strict equality. We decide to overload strict equality in the same way we do for other value's in OCL:

defs $\mathit{StrictRefEq}_{\mathit{Set}}$:

$(x::('\mathfrak{A},'\alpha::\mathit{null})\mathit{Set}) \doteq y \equiv \lambda\ \tau.\ \mathit{if}\ (\upsilon\ x)\ \tau = \mathit{true}\ \tau \wedge (\upsilon\ y)\ \tau = \mathit{true}\ \tau$

$\qquad \mathit{then}\ (x \triangleq y)\tau$

$\qquad \mathit{else}\ \mathit{invalid}\ \tau$

One might object here that for the case of objects, this is an empty definition. The answer is no, we will restrain later on states and objects such that any object has its oid stored inside the object (so the ref, under which an object can be referenced in the store will represented in the object itself). For such well-formed stores that satisfy this invariant (the WFF-invariant), the referential equality and the strong equality—and therefore the strict equality on sets in the sense above—coincides.

Property proof in terms of *profile-bin3*

interpretation $\mathit{StrictRefEq}_{\mathit{Set}}$: *profile-bin3* $\lambda\ x\ y.\ (x::('\mathfrak{A},'\alpha::\mathit{null})\mathit{Set}) \doteq y$

by *unfold-locales* (*auto simp*: $\mathit{StrictRefEq}_{\mathit{Set}}$)

Execution Rules on OclIncluding

lemma *OclIncluding-finite-rep-set* :

assumes *X-def* : $\tau \models \delta\ X$

and *x-val* : $\tau \models \upsilon\ x$

shows $\mathit{finite}\ \lceil\lceil \mathit{Rep\text{-}Set}_{\mathit{base}}\ (X{-}{>}\mathit{including}(x)\ \tau)\rceil\rceil = \mathit{finite}\ \lceil\lceil \mathit{Rep\text{-}Set}_{\mathit{base}}\ (X\ \tau)\rceil\rceil$

proof –

have C : $\lfloor\lfloor \mathit{insert}\ (x\ \tau)\ \lceil\lceil \mathit{Rep\text{-}Set}_{\mathit{base}}\ (X\ \tau)\rceil\rceil\rfloor\rfloor \in \{X.\ X = \mathit{bot} \vee X = \mathit{null} \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq \mathit{bot})\}$

by(*insert X-def x-val*, *frule Set-inv-lemma*, *simp add*: *foundation18 invalid-def*)

show *?thesis*

by(*insert X-def x-val*,

auto simp: *OclIncluding-def* $\mathit{Abs\text{-}Set}_{\mathit{base}}$*-inverse*[*OF C*]

dest: *foundation13*[*THEN iffD2*, *THEN foundation22*[*THEN iffD1*]])

qed

lemma *OclIncluding-rep-set*:

assumes *S-def*: $\tau \models \delta\ S$

shows $\lceil\lceil \mathit{Rep\text{-}Set}_{\mathit{base}}\ (S{-}{>}\mathit{including}(\lambda\text{-}.\ \lfloor\lfloor x\rfloor\rfloor)\ \tau)\rceil\rceil = \mathit{insert}\ \lfloor\lfloor x\rfloor\rfloor\ \lceil\lceil \mathit{Rep\text{-}Set}_{\mathit{base}}\ (S\ \tau)\rceil\rceil$

apply(*simp add*: *OclIncluding-def S-def*[*simplified OclValid-def*])

apply(*subst* $\mathit{Abs\text{-}Set}_{\mathit{base}}$*-inverse*, *simp add*: *bot-option-def null-option-def*)

apply(*insert Set-inv-lemma*[*OF S-def*], *metis bot-option-def not-Some-eq*)

by(*simp*)

lemma *OclIncluding-notempty-rep-set*:

assumes *X-def*: $\tau \models \delta\ X$

and *a-val*: $\tau \models \upsilon\ a$

shows $\lceil\lceil \mathit{Rep\text{-}Set}_{\mathit{base}}\ (X{-}{>}\mathit{including}(a)\ \tau)\rceil\rceil \neq \{\}$

apply(*simp add*: *OclIncluding-def X-def*[*simplified OclValid-def*] *a-val*[*simplified OclValid-def*])

apply(*subst* $\mathit{Abs\text{-}Set}_{\mathit{base}}$*-inverse*, *simp add*: *bot-option-def null-option-def*)

apply(*insert Set-inv-lemma*[*OF X-def*], *metis a-val foundation18*$'$)
by(*simp*)

lemma *OclIncluding-includes0*:
assumes $\tau \models X{-}{>}includes(x)$
shows $X{-}{>}including(x)\ \tau = X\ \tau$
proof –
have *includes-def*: $\tau \models X{-}{>}includes(x) \Longrightarrow \tau \models \delta\ X$
by (*metis bot-fun-def OclIncludes-def OclValid-def defined3 foundation16*)

have *includes-val*: $\tau \models X{-}{>}includes(x) \Longrightarrow \tau \models \upsilon\ x$
by (*metis* (*hide-lams*, *no-types*) *foundation6*
OclIncludes-valid-args-valid$'$ *OclIncluding-valid-args-valid OclIncluding-valid-args-valid*$''$)

show *?thesis*
apply(*insert includes-def*[*OF assms*] *includes-val*[*OF assms*] *assms*,
simp add: *OclIncluding-def OclIncludes-def OclValid-def true-def*)
apply(*drule insert-absorb*, *simp*, *subst abs-rep-simp*$'$)
by(*simp-all add*: *OclValid-def true-def*)
qed

lemma *OclIncluding-includes*:
assumes $\tau \models X{-}{>}includes(x)$
shows $\tau \models X{-}{>}including(x) \triangleq X$
by(*simp add*: *StrongEq-def OclValid-def true-def OclIncluding-includes0*[*OF assms*])

lemma *OclIncluding-commute0* :
assumes *S-def* : $\tau \models \delta\ S$
and *i-val* : $\tau \models \upsilon\ i$
and *j-val* : $\tau \models \upsilon\ j$
shows $\tau \models ((S :: ('\mathfrak{A}, 'a{::}null)\ Set){-}{>}including(i){-}{>}including(j) \triangleq (S{-}{>}including(j){-}{>}including(i)))$
proof –
have *A* : $\lfloor\lfloor insert\ (i\ \tau)\ \lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil\rfloor\rfloor \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
by(*insert S-def i-val*, *frule Set-inv-lemma*, *simp add*: *foundation18 invalid-def*)
have *B* : $\lfloor\lfloor insert\ (j\ \tau)\ \lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil\rfloor\rfloor \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
by(*insert S-def j-val*, *frule Set-inv-lemma*, *simp add*: *foundation18 invalid-def*)

have *G1* : $Abs\text{-}Set_{base}\ \lfloor\lfloor insert\ (i\ \tau)\ \lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil\rfloor\rfloor \neq Abs\text{-}Set_{base}\ None$
by(*insert A*, *simp add*: $Abs\text{-}Set_{base}$*-inject bot-option-def null-option-def*)
have *G2* : $Abs\text{-}Set_{base}\ \lfloor\lfloor insert\ (i\ \tau)\ \lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil\rfloor\rfloor \neq Abs\text{-}Set_{base}\ \lfloor None\rfloor$
by(*insert A*, *simp add*: $Abs\text{-}Set_{base}$*-inject bot-option-def null-option-def*)
have *G3* : $Abs\text{-}Set_{base}\ \lfloor\lfloor insert\ (j\ \tau)\ \lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil\rfloor\rfloor \neq Abs\text{-}Set_{base}\ None$
by(*insert B*, *simp add*: $Abs\text{-}Set_{base}$*-inject bot-option-def null-option-def*)
have *G4* : $Abs\text{-}Set_{base}\ \lfloor\lfloor insert\ (j\ \tau)\ \lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil\rfloor\rfloor \neq Abs\text{-}Set_{base}\ \lfloor None\rfloor$
by(*insert B*, *simp add*: $Abs\text{-}Set_{base}$*-inject bot-option-def null-option-def*)

have $*$: $(\delta\ (\lambda\text{-}.\ Abs\text{-}Set_{base}\ \lfloor\lfloor insert\ (i\ \tau)\ \lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil\rfloor\rfloor))\ \tau = \lfloor\lfloor True\rfloor\rfloor$
by(*auto simp*: *OclValid-def false-def defined-def null-fun-def true-def*

bot-fun-def bot-Set$_{base}$*-def null-Set*$_{base}$*-def S-def i-val G1 G2*)

have $**$: $(\delta\ (\lambda\text{-}.\ \mathit{Abs\text{-}Set_{base}}\ \lfloor\lfloor \mathit{insert}\ (j\ \tau)\ \lceil\lceil \mathit{Rep\text{-}Set_{base}}\ (S\ \tau)\rceil\rceil\rfloor\rfloor))\ \tau = \lfloor\lfloor \mathit{True}\rfloor\rfloor$
by(*auto simp*: *OclValid-def false-def defined-def null-fun-def true-def*
bot-fun-def bot-Set$_{base}$*-def null-Set*$_{base}$*-def S-def i-val G3 G4*)

have $***$: $\mathit{Abs\text{-}Set_{base}}\ \lfloor\lfloor \mathit{insert}(j\ \tau)\lceil\lceil \mathit{Rep\text{-}Set_{base}}(\mathit{Abs\text{-}Set_{base}}\lfloor\lfloor \mathit{insert}(i\ \tau)\lceil\lceil \mathit{Rep\text{-}Set_{base}}(S\ \tau)\rceil\rceil\rfloor\rfloor)\rceil\rceil\rfloor\rfloor =$
$\mathit{Abs\text{-}Set_{base}}\ \lfloor\lfloor \mathit{insert}(i\ \tau)\lceil\lceil \mathit{Rep\text{-}Set_{base}}(\mathit{Abs\text{-}Set_{base}}\lfloor\lfloor \mathit{insert}(j\ \tau)\lceil\lceil \mathit{Rep\text{-}Set_{base}}(S\ \tau)\rceil\rceil\rfloor\rfloor)\rceil\rceil\rfloor\rfloor$
by(*simp add*: *Abs-Set*$_{base}$*-inverse*[*OF A*] *Abs-Set*$_{base}$*-inverse*[*OF B*] *Set.insert-commute*)
show *?thesis*
apply(*simp add*: *OclIncluding-def S-def*[*simplified OclValid-def*]
i-val[*simplified OclValid-def*] *j-val*[*simplified OclValid-def*]
true-def OclValid-def StrongEq-def)
apply(*subst cp-defined*,
simp add: *S-def*[*simplified OclValid-def*]
i-val[*simplified OclValid-def*] *j-val*[*simplified OclValid-def*] *true-def* $*$)
apply(*subst cp-defined*,
simp add: *S-def*[*simplified OclValid-def*]
i-val[*simplified OclValid-def*] *j-val*[*simplified OclValid-def*] *true-def* $**$ $***$)
apply(*subst cp-defined*,
simp add: *S-def*[*simplified OclValid-def*]
i-val[*simplified OclValid-def*] *j-val*[*simplified OclValid-def*] *true-def* $*$)
apply(*subst cp-defined*,
simp add: *S-def*[*simplified OclValid-def*]
i-val[*simplified OclValid-def*] *j-val*[*simplified OclValid-def*] *true-def* $*$)
apply(*subst cp-defined*,
simp add: *S-def*[*simplified OclValid-def*]
i-val[*simplified OclValid-def*] *j-val*[*simplified OclValid-def*] *true-def* $*$ $**$)
done
qed

lemma *OclIncluding-commute*[*simp*,*code-unfold*]:
$((S :: ('\mathfrak{A}, 'a{::}null)\ Set){-}{>}including(i){-}{>}including(j) = (S{-}{>}including(j){-}{>}including(i)))$
proof −
have A: $\bigwedge\tau.\quad \tau \models (i \triangleq invalid) \quad\Longrightarrow (S{-}{>}including(i){-}{>}including(j))\ \tau = invalid\ \tau$
apply(*rule foundation22*[*THEN iffD1*])
by(*erule StrongEq-L-subst2-rev*, *simp*,*simp*)
have A': $\bigwedge\tau.\quad \tau \models (i \triangleq invalid) \quad\Longrightarrow (S{-}{>}including(j){-}{>}including(i))\ \tau = invalid\ \tau$
apply(*rule foundation22*[*THEN iffD1*])
by(*erule StrongEq-L-subst2-rev*, *simp*,*simp*)
have B:$\bigwedge\tau.\quad \tau \models (j \triangleq invalid) \quad\Longrightarrow (S{-}{>}including(i){-}{>}including(j))\ \tau = invalid\ \tau$
apply(*rule foundation22*[*THEN iffD1*])
by(*erule StrongEq-L-subst2-rev*, *simp*,*simp*)
have B':$\bigwedge\tau.\quad \tau \models (j \triangleq invalid) \quad\Longrightarrow (S{-}{>}including(j){-}{>}including(i))\ \tau = invalid\ \tau$
apply(*rule foundation22*[*THEN iffD1*])
by(*erule StrongEq-L-subst2-rev*, *simp*,*simp*)
have C: $\bigwedge\tau.\quad \tau \models (S \triangleq invalid) \quad\Longrightarrow (S{-}{>}including(i){-}{>}including(j))\ \tau = invalid\ \tau$

apply(*rule foundation22*[*THEN iffD1*])

by(*erule StrongEq-L-subst2-rev*, *simp*,*simp*)

have C': $\bigwedge \tau.\ \ \tau \models (S \triangleq invalid) \ \ \Longrightarrow (S{-}{>}including(j){-}{>}including(i))\ \tau = invalid\ \tau$

apply(*rule foundation22*[*THEN iffD1*])

by(*erule StrongEq-L-subst2-rev*, *simp*,*simp*)

have D: $\bigwedge \tau.\ \ \tau \models (S \triangleq null) \ \ \Longrightarrow (S{-}{>}including(i){-}{>}including(j))\ \tau = invalid\ \tau$

apply(*rule foundation22*[*THEN iffD1*])

by(*erule StrongEq-L-subst2-rev*, *simp*,*simp*)

have D': $\bigwedge \tau.\ \ \tau \models (S \triangleq null) \ \ \Longrightarrow (S{-}{>}including(j){-}{>}including(i))\ \tau = invalid\ \tau$

apply(*rule foundation22*[*THEN iffD1*])

by(*erule StrongEq-L-subst2-rev*, *simp*,*simp*)

show *?thesis*

apply(*rule ext*, *rename-tac* τ)

apply(*case-tac* $\tau \models (\upsilon\ i)$)

apply(*case-tac* $\tau \models (\upsilon\ j)$)

apply(*case-tac* $\tau \models (\delta\ S)$)

apply(*simp only*: *OclIncluding-commute0*[*THEN foundation22*[*THEN iffD1*]])

apply(*simp add*: *foundation16*$'$, *elim disjE*)

apply(*simp add*: *C*[*OF foundation22*[*THEN iffD2*]] C'[*OF foundation22*[*THEN iffD2*]])

apply(*simp add*: *D*[*OF foundation22*[*THEN iffD2*]] D'[*OF foundation22*[*THEN iffD2*]])

apply(*simp add*:*foundation18 B*[*OF foundation22*[*THEN iffD2*]] B'[*OF foundation22*[*THEN iffD2*]])

apply(*simp add*:*foundation18 A*[*OF foundation22*[*THEN iffD2*]] A'[*OF foundation22*[*THEN iffD2*]])

done

qed

Execution Rules on OclExcluding **lemma** *OclExcluding-finite-rep-set* :

assumes *X-def* : $\tau \models \delta\ X$

and *x-val* : $\tau \models \upsilon\ x$

shows *finite* $\lceil\lceil Rep\text{-}Set_{base}\ (X{-}{>}excluding(x)\ \tau)\rceil\rceil$ = *finite* $\lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil$

proof $-$

have *C* : $\lfloor\lfloor\lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil - \{x\ \tau\}\rfloor\rfloor \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$

apply(*insert X-def x-val*, *frule Set-inv-lemma*)

apply(*simp add*: *foundation18 invalid-def*)

done

show *?thesis*

by(*insert X-def x-val*,

auto simp: *OclExcluding-def Abs-Set*$_{base}$*-inverse*[*OF C*]

dest: *foundation13*[*THEN iffD2*, *THEN foundation22*[*THEN iffD1*]])

qed

lemma *OclExcluding-rep-set*:

assumes *S-def*: $\tau \models \delta\ S$

shows $\lceil\lceil Rep\text{-}Set_{base}\ (S{-}{>}excluding(\lambda\text{-}.\ \lfloor\lfloor x\rfloor\rfloor)\ \tau)\rceil\rceil = \lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil - \{\lfloor\lfloor x\rfloor\rfloor\}$

apply(*simp add*: *OclExcluding-def S-def*[*simplified OclValid-def*])

apply(*subst Abs-Set*$_{base}$*-inverse*, *simp add*: *bot-option-def null-option-def*)

apply(*insert Set-inv-lemma*[*OF S-def*], *metis Diff-iff bot-option-def not-None-eq*)

by(*simp*)

```
lemma OclExcluding-excludes0:
 assumes τ ⊨ X−>excludes(x)
   shows X−>excluding(x) τ = X τ
proof −
 have excludes-def: τ ⊨ X−>excludes(x) ⟹ τ ⊨ δ X
 by (metis (hide-lams, no-types) OclExcludes-defined-args-valid foundation6)

 have excludes-val: τ ⊨ X−>excludes(x) ⟹ τ ⊨ υ x
 by (metis (hide-lams, no-types) OclExcludes-def OclIncludes-defined-args-valid OclNot-defargs)

 show ?thesis
  apply(insert excludes-def[OF assms] excludes-val[OF assms] assms,
       simp add: OclExcluding-def OclExcludes-def OclIncludes-def OclNot-def OclValid-def true-def)
 by (metis (hide-lams, no-types) abs-rep-simp′ assms excludes-def)
qed

lemma OclExcluding-excludes:
 assumes τ ⊨ X−>excludes(x)
   shows τ ⊨ X−>excluding(x) ≜ X
by(simp add: StrongEq-def OclValid-def true-def OclExcluding-excludes0[OF assms])

lemma OclExcluding-charn0[simp]:
assumes val-x:τ ⊨ (υ x)
shows        τ ⊨ ((Set{}−>excluding(x))  ≜  Set{})
proof −
  have A : ⌊None⌋ ∈ {X. X = bot ∨ X = null ∨ (∀x∈⌈⌈X⌉⌉. x ≠ bot)}
  by(simp add: null-option-def bot-option-def)
  have B : ⌊⌊{}⌋⌋ ∈ {X. X = bot ∨ X = null ∨ (∀x∈⌈⌈X⌉⌉. x ≠ bot)} by(simp add: mtSet-def)

  show ?thesis using val-x
    apply(auto simp: OclValid-def OclIncludes-def OclNot-def false-def true-def StrongEq-def
                 OclExcluding-def mtSet-def defined-def bot-fun-def null-fun-def null-Set_base-def)
     apply(auto simp: mtSet-def Set_base.Abs-Set_base-inverse
                  Set_base.Abs-Set_base-inject[OF B A])
  done
qed

lemma OclExcluding-commute0 :
 assumes S-def : τ ⊨ δ S
     and i-val : τ ⊨ υ i
     and j-val : τ ⊨ υ j
   shows τ ⊨ ((S :: ('𝔄, 'a::null) Set)−>excluding(i)−>excluding(j) ≜ (S−>excluding(j)−>excluding(i)))
proof −
  have A : ⌊⌊⌈⌈Rep-Set_base (S τ)⌉⌉ − {i τ}⌋⌋ ∈ {X. X = bot ∨ X = null ∨ (∀x∈⌈⌈X⌉⌉. x ≠ bot)}
          by(insert S-def i-val, frule Set-inv-lemma, simp add: foundation18 invalid-def)
  have B : ⌊⌊⌈⌈Rep-Set_base (S τ)⌉⌉ − {j τ}⌋⌋ ∈ {X. X = bot ∨ X = null ∨ (∀x∈⌈⌈X⌉⌉. x ≠ bot)}
          by(insert S-def j-val, frule Set-inv-lemma, simp add: foundation18 invalid-def)
```

have *G1* : $Abs\text{-}Set_{base}\ \lfloor\lfloor\lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil - \{i\ \tau\}\rfloor\rfloor \neq Abs\text{-}Set_{base}\ None$
  by(*insert A*, *simp add*: *Abs-Set*$_{base}$*-inject bot-option-def null-option-def*)
have *G2* : $Abs\text{-}Set_{base}\ \lfloor\lfloor\lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil - \{i\ \tau\}\rfloor\rfloor \neq Abs\text{-}Set_{base}\ \lfloor None\rfloor$
  by(*insert A*, *simp add*: *Abs-Set*$_{base}$*-inject bot-option-def null-option-def*)
have *G3* : $Abs\text{-}Set_{base}\ \lfloor\lfloor\lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil - \{j\ \tau\}\rfloor\rfloor \neq Abs\text{-}Set_{base}\ None$
  by(*insert B*, *simp add*: *Abs-Set*$_{base}$*-inject bot-option-def null-option-def*)
have *G4* : $Abs\text{-}Set_{base}\ \lfloor\lfloor\lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil - \{j\ \tau\}\rfloor\rfloor \neq Abs\text{-}Set_{base}\ \lfloor None\rfloor$
  by(*insert B*, *simp add*: *Abs-Set*$_{base}$*-inject bot-option-def null-option-def*)

have $*$: $(\delta\ (\lambda\text{-}.\ Abs\text{-}Set_{base}\ \lfloor\lfloor\lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil - \{i\ \tau\}\rfloor\rfloor))\ \tau = \lfloor\lfloor True\rfloor\rfloor$
  by(*auto simp*: *OclValid-def false-def defined-def null-fun-def true-def*
    bot-fun-def bot-Set$_{base}$*-def null-Set*$_{base}$*-def S-def i-val G1 G2*)

have $**$: $(\delta\ (\lambda\text{-}.\ Abs\text{-}Set_{base}\ \lfloor\lfloor\lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil - \{j\ \tau\}\rfloor\rfloor))\ \tau = \lfloor\lfloor True\rfloor\rfloor$
  by(*auto simp*: *OclValid-def false-def defined-def null-fun-def true-def*
    bot-fun-def bot-Set$_{base}$*-def null-Set*$_{base}$*-def S-def i-val G3 G4*)

have $***$: $Abs\text{-}Set_{base}\ \lfloor\lfloor\lceil\lceil Rep\text{-}Set_{base}(Abs\text{-}Set_{base}\lfloor\lfloor\lceil\lceil Rep\text{-}Set_{base}(S\ \tau)\rceil\rceil-\{i\ \tau\}\rfloor\rfloor)\rceil\rceil-\{j\ \tau\}\rfloor\rfloor =$
  $Abs\text{-}Set_{base}\ \lfloor\lfloor\lceil\lceil Rep\text{-}Set_{base}(Abs\text{-}Set_{base}\lfloor\lfloor\lceil\lceil Rep\text{-}Set_{base}(S\ \tau)\rceil\rceil-\{j\ \tau\}\rfloor\rfloor)\rceil\rceil-\{i\ \tau\}\rfloor\rfloor$
  apply(*simp add*: *Abs-Set*$_{base}$*-inverse*[*OF A*] *Abs-Set*$_{base}$*-inverse*[*OF B*])
  by (*metis Diff-insert2 insert-commute*)
show *?thesis*
  apply(*simp add*: *OclExcluding-def S-def*[*simplified OclValid-def*]
    i-val[*simplified OclValid-def*] *j-val*[*simplified OclValid-def*]
    true-def OclValid-def StrongEq-def)
  apply(*subst cp-defined*,
    simp add: *S-def*[*simplified OclValid-def*]
      i-val[*simplified OclValid-def*] *j-val*[*simplified OclValid-def*] *true-def* $*$)
  apply(*subst cp-defined*,
    simp add: *S-def*[*simplified OclValid-def*]
      i-val[*simplified OclValid-def*] *j-val*[*simplified OclValid-def*] *true-def* $**$ $***$)
  apply(*subst cp-defined*,
    simp add: *S-def*[*simplified OclValid-def*]
      i-val[*simplified OclValid-def*] *j-val*[*simplified OclValid-def*] *true-def* $*$)
  apply(*subst cp-defined*,
    simp add: *S-def*[*simplified OclValid-def*]
      i-val[*simplified OclValid-def*] *j-val*[*simplified OclValid-def*] *true-def* $*$)
  apply(*subst cp-defined*,
    simp add: *S-def*[*simplified OclValid-def*]
      i-val[*simplified OclValid-def*] *j-val*[*simplified OclValid-def*] *true-def* $*$ $**$)
  done
qed

lemma *OclExcluding-commute*[*simp*,*code-unfold*]:
$((S :: ('\mathfrak{A}, 'a{::}null)\ Set){-}{>}excluding(i){-}{>}excluding(j) = (S{-}{>}excluding(j){-}{>}excluding(i)))$
proof –
have *A*: $\bigwedge\tau.\quad \tau \models i \triangleq invalid \implies (S{-}{>}excluding(i){-}{>}excluding(j))\ \tau = invalid\ \tau$

```
          apply(rule foundation22[THEN iffD1])
          by(erule StrongEq-L-subst2-rev, simp,simp)
  have A': ⋀ τ.  τ ⊨ i ≜ invalid  ⟹ (S−>excluding(j)−>excluding(i)) τ = invalid τ
          apply(rule foundation22[THEN iffD1])
          by(erule StrongEq-L-subst2-rev, simp,simp)
  have B:⋀ τ.  τ ⊨ j ≜ invalid  ⟹ (S−>excluding(i)−>excluding(j)) τ = invalid τ
          apply(rule foundation22[THEN iffD1])
          by(erule StrongEq-L-subst2-rev, simp,simp)
  have B':⋀ τ.  τ ⊨ j ≜ invalid  ⟹ (S−>excluding(j)−>excluding(i)) τ = invalid τ
          apply(rule foundation22[THEN iffD1])
          by(erule StrongEq-L-subst2-rev, simp,simp)
  have C: ⋀ τ.  τ ⊨ S ≜ invalid  ⟹ (S−>excluding(i)−>excluding(j)) τ = invalid τ
          apply(rule foundation22[THEN iffD1])
          by(erule StrongEq-L-subst2-rev, simp,simp)
  have C': ⋀ τ.  τ ⊨ S ≜ invalid  ⟹ (S−>excluding(j)−>excluding(i)) τ = invalid τ
          apply(rule foundation22[THEN iffD1])
          by(erule StrongEq-L-subst2-rev, simp,simp)
  have D: ⋀ τ.  τ ⊨ S ≜ null  ⟹ (S−>excluding(i)−>excluding(j)) τ = invalid τ
          apply(rule foundation22[THEN iffD1])
          by(erule StrongEq-L-subst2-rev, simp,simp)
  have D': ⋀ τ.  τ ⊨ S ≜ null  ⟹ (S−>excluding(j)−>excluding(i)) τ = invalid τ
          apply(rule foundation22[THEN iffD1])
          by(erule StrongEq-L-subst2-rev, simp,simp)
  show ?thesis
    apply(rule ext, rename-tac τ)
    apply(case-tac τ ⊨ (υ i))
     apply(case-tac τ ⊨ (υ j))
      apply(case-tac τ ⊨ (δ S))
       apply(simp only: OclExcluding-commute0[THEN foundation22[THEN iffD1]])
      apply(simp add: foundation16', elim disjE)
     apply(simp add: C[OF foundation22[THEN iffD2]] C'[OF foundation22[THEN iffD2]])
    apply(simp add: D[OF foundation22[THEN iffD2]] D'[OF foundation22[THEN iffD2]])
   apply(simp add:foundation18 B[OF foundation22[THEN iffD2]] B'[OF foundation22[THEN iffD2]])
  apply(simp add:foundation18 A[OF foundation22[THEN iffD2]] A'[OF foundation22[THEN iffD2]])
 done
qed


lemma OclExcluding-charn0-exec[simp,code-unfold]:
(Set{}−>excluding(x)) = (if (υ x) then Set{} else invalid endif)
proof −
  have A: ⋀ τ. (Set{}−>excluding(invalid)) τ = (if (υ invalid) then Set{} else invalid endif) τ
        by simp
  have B: ⋀ τ x. τ ⊨ (υ x) ⟹
                (Set{}−>excluding(x)) τ = (if (υ x) then Set{} else invalid endif) τ
        by(simp add: OclExcluding-charn0[THEN foundation22[THEN iffD1]])
  show ?thesis
    apply(rule ext, rename-tac τ)
   apply(case-tac τ ⊨ (υ x))
    apply(simp add: B)
   apply(simp add: foundation18)
   apply(subst cp-OclExcluding, simp)
   apply(simp add: cp-OclIf[symmetric] cp-OclExcluding[symmetric] cp-valid[symmetric] A)
  done
qed

lemma OclExcluding-charn1:
assumes def-X:τ ⊨ (δ X)
and     val-x:τ ⊨ (υ x)
and     val-y:τ ⊨ (υ y)
and     neq  :τ ⊨ not(x ≜ y)
shows        τ ⊨ ((X−>including(x))−>excluding(y)) ≜ ((X−>excluding(y))−>including(x))
proof −
 have C : ⌊⌊insert (x τ) ⌈⌈Rep-Set_base (X τ)⌉⌉⌋⌋ ∈ {X. X = bot ∨ X = null ∨ (∀x∈⌈⌈X⌉⌉. x ≠ bot)}
        by(insert def-X val-x, frule Set-inv-lemma, simp add: foundation18 invalid-def)
 have D : ⌊⌊⌈⌈Rep-Set_base (X τ)⌉⌉ − {y τ}⌋⌋ ∈ {X. X = bot ∨ X = null ∨ (∀x∈⌈⌈X⌉⌉. x ≠ bot)}
        by(insert def-X val-x, frule Set-inv-lemma, simp add: foundation18 invalid-def)
 have E : x τ ≠ y τ
        by(insert neq,
           auto simp: OclValid-def bot-fun-def OclIncluding-def OclIncludes-def
                      false-def true-def defined-def valid-def bot-Set_base-def
                      null-fun-def null-Set_base-def StrongEq-def OclNot-def)

 have G1 : Abs-Set_base ⌊⌊insert (x τ) ⌈⌈Rep-Set_base (X τ)⌉⌉⌋⌋ ≠ Abs-Set_base None
        by(insert C, simp add: Abs-Set_base-inject bot-option-def null-option-def)
 have G2 : Abs-Set_base ⌊⌊insert (x τ) ⌈⌈Rep-Set_base (X τ)⌉⌉⌋⌋ ≠ Abs-Set_base ⌊None⌋
        by(insert C, simp add: Abs-Set_base-inject bot-option-def null-option-def)
 have G : (δ (λ-. Abs-Set_base ⌊⌊insert (x τ) ⌈⌈Rep-Set_base (X τ)⌉⌉⌋⌋)) τ = true τ
        by(auto simp: OclValid-def false-def true-def defined-def
                      bot-fun-def bot-Set_base-def null-fun-def null-Set_base-def G1 G2)

 have H1 : Abs-Set_base ⌊⌊⌈⌈Rep-Set_base (X τ)⌉⌉ − {y τ}⌋⌋ ≠ Abs-Set_base None
        by(insert D, simp add: Abs-Set_base-inject bot-option-def null-option-def)
 have H2 : Abs-Set_base ⌊⌊⌈⌈Rep-Set_base (X τ)⌉⌉ − {y τ}⌋⌋ ≠ Abs-Set_base ⌊None⌋
        by(insert D, simp add: Abs-Set_base-inject bot-option-def null-option-def)
 have H : (δ (λ-. Abs-Set_base ⌊⌊⌈⌈Rep-Set_base (X τ)⌉⌉ − {y τ}⌋⌋)) τ = true τ
        by(auto simp: OclValid-def false-def true-def defined-def
                        bot-fun-def bot-Set_base-def null-fun-def null-Set_base-def H1 H2)

 have Z : insert (x τ) ⌈⌈Rep-Set_base (X τ)⌉⌉ − {y τ} = insert (x τ) (⌈⌈Rep-Set_base (X τ)⌉⌉ − {y τ})
        by(auto simp: E)
 show ?thesis
  apply(insert def-X[THEN foundation13[THEN iffD2]] val-x[THEN foundation13[THEN iffD2]]
             val-y[THEN foundation13[THEN iffD2]])
  apply(simp add: foundation22 OclIncluding-def OclExcluding-def def-X[THEN foundation16[THEN iffD1,standard]])
  apply(subst cp-defined, simp)+
```

apply(*simp add*: *G H Abs-Set$_{base}$-inverse*[*OF C*] *Abs-Set$_{base}$-inverse*[*OF D*] *Z*)
done
qed

lemma *OclExcluding-charn2*:
assumes *def-X*:$\tau \models (\delta\ X)$
and *val-x*:$\tau \models (\upsilon\ x)$
shows $\tau \models (((X{-}{>}including(x)){-}{>}excluding(x)) \triangleq (X{-}{>}excluding(x)))$
proof –
have *C* : $\lfloor\lfloor insert\ (x\ \tau)\ \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil\rfloor\rfloor \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
by(*insert def-X val-x*, *frule Set-inv-lemma*, *simp add*: *foundation18 invalid-def*)
have *G1* : $Abs\text{-}Set_{base}\ \lfloor\lfloor insert\ (x\ \tau)\ \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil\rfloor\rfloor \neq Abs\text{-}Set_{base}\ None$
by(*insert C*, *simp add*: *Abs-Set$_{base}$-inject bot-option-def null-option-def*)
have *G2* : $Abs\text{-}Set_{base}\ \lfloor\lfloor insert\ (x\ \tau)\ \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil\rfloor\rfloor \neq Abs\text{-}Set_{base}\ \lfloor None\rfloor$
by(*insert C*, *simp add*: *Abs-Set$_{base}$-inject bot-option-def null-option-def*)
show *?thesis*
apply(*insert def-X*[*THEN foundation16*[*THEN iffD1*,*standard*]]
val-x[*THEN foundation18*[*THEN iffD1*,*standard*]])
apply(*auto simp*: *OclValid-def bot-fun-def OclIncluding-def OclIncludes-def false-def true-def*
invalid-def defined-def valid-def bot-Set$_{base}$-def null-fun-def null-Set$_{base}$-def
StrongEq-def)
apply(*subst cp-OclExcluding*)
apply(*auto simp*:*OclExcluding-def*)
apply(*simp add*: *Abs-Set$_{base}$-inverse*[*OF C*])
apply(*simp-all add*: *false-def true-def defined-def valid-def*
null-fun-def bot-fun-def null-Set$_{base}$-def bot-Set$_{base}$-def
split: *bool.split-asm HOL.split-if-asm option.split*)
apply(*auto simp*: *G1 G2*)
done
qed

theorem *OclExcluding-charn3*: $((X{-}{>}including(x)){-}{>}excluding(x)) = (X{-}{>}excluding(x))$
proof –
have *A1* : $\bigwedge\tau.\ \tau \models (X \triangleq invalid) \Longrightarrow (X{-}{>}including(x){-}{>}excluding(x))\ \tau = invalid\ \tau$
apply(*rule foundation22*[*THEN iffD1*])
by(*erule StrongEq-L-subst2-rev*, *simp*,*simp*)
have *A1'*: $\bigwedge\tau.\ \tau \models (X \triangleq invalid) \Longrightarrow (X{-}{>}excluding(x))\ \tau = invalid\ \tau$
apply(*rule foundation22*[*THEN iffD1*])
by(*erule StrongEq-L-subst2-rev*, *simp*,*simp*)
have *A2* : $\bigwedge\tau.\ \tau \models (X \triangleq null) \Longrightarrow (X{-}{>}including(x){-}{>}excluding(x))\ \tau = invalid\ \tau$
apply(*rule foundation22*[*THEN iffD1*])
by(*erule StrongEq-L-subst2-rev*, *simp*,*simp*)
have *A2'*: $\bigwedge\tau.\ \tau \models (X \triangleq null) \Longrightarrow (X{-}{>}excluding(x))\ \tau = invalid\ \tau$

```
          apply(rule foundation22[THEN iffD1])
          by(erule StrongEq-L-subst2-rev, simp,simp)
 have A3 : ⋀τ. τ ⊨ (x ≜ invalid) ⟹ (X−>including(x)−>excluding(x)) τ = invalid τ
          apply(rule foundation22[THEN iffD1])
          by(erule StrongEq-L-subst2-rev, simp,simp)
 have A3′: ⋀τ. τ ⊨ (x ≜ invalid) ⟹ (X−>excluding(x)) τ = invalid τ
          apply(rule foundation22[THEN iffD1])
          by(erule StrongEq-L-subst2-rev, simp,simp)

 show ?thesis
 apply(rule ext, rename-tac τ)
 apply(case-tac τ ⊨ (υ x))
  apply(case-tac τ ⊨ (δ X))
   apply(simp only: OclExcluding-charn2[THEN foundation22[THEN iffD1]])
   apply(simp add: foundation16′, elim disjE)
   apply(simp add: A1[OF foundation22[THEN iffD2]] A1′[OF foundation22[THEN iffD2]])
  apply(simp add: A2[OF foundation22[THEN iffD2]] A2′[OF foundation22[THEN iffD2]])
 apply(simp add:foundation18 A3[OF foundation22[THEN iffD2]] A3′[OF foundation22[THEN iffD2]])
 done
qed
```

One would like a generic theorem of the form:

```
lemma OclExcluding_charn_exec:
         "(X−>including(x::('𝔄,'a::null) val)−>excluding(y)) =
          (if  δ X then  if  x ≐y
                         then  X−>excluding(y)
                         else  X−>excluding(y)−>including(x)
                         endif
                  else  invalid  endif)"
```

Unfortunately, this does not hold in general, since referential equality is an overloaded concept and has to be defined for each type individually. Consequently, it is only valid for concrete type instances for Boolean, Integer, and Sets thereof...

The computational law *OclExcluding-charn-exec* becomes generic since it uses strict equality which in itself is generic. It is possible to prove the following generic theorem and instantiate it later (using properties that link the polymorphic logical strong equality with the concrete instance of strict quality).

```
lemma OclExcluding-charn-exec:
 assumes strict1: (invalid ≐ y) = invalid
 and     strict2: (x ≐ invalid) = invalid
 and     StrictRefEq-valid-args-valid: ⋀ (x::('𝔄,'a::null)val) y τ.
                     (τ ⊨ δ (x ≐ y)) = ((τ ⊨ (υ x)) ∧ (τ ⊨ υ y))
 and     cp-StrictRefEq: ⋀ (X::('𝔄,'a::null)val) Y τ. (X ≐ Y) τ = ((λ-. X τ) ≐ (λ-. Y τ)) τ
 and     StrictRefEq-vs-StrongEq: ⋀ (x::('𝔄,'a::null)val) y τ.
                      τ ⊨ υ x ⟹ τ ⊨ υ y ⟹ (τ ⊨ ((x ≐ y) ≜ (x ≜ y)))
 shows (X−>including(x::('𝔄,'a::null)val)−>excluding(y)) =
       (if δ X then if x ≐ y
                then X−>excluding(y)
                else X−>excluding(y)−>including(x)
                endif
            else invalid endif)
proof −

 have A1: ⋀τ. τ ⊨ (X ≜ invalid) ⟹
          (X−>including(x)−>includes(y)) τ = invalid τ
          apply(rule foundation22[THEN iffD1])
          by(erule StrongEq-L-subst2-rev, simp,simp)

 have B1: ⋀τ. τ ⊨ (X ≜ null) ⟹
          (X−>including(x)−>includes(y)) τ = invalid  τ
          apply(rule foundation22[THEN iffD1])
          by(erule StrongEq-L-subst2-rev, simp,simp)

 have A2: ⋀τ. τ ⊨ (X ≜ invalid) ⟹ X−>including(x)−>excluding(y) τ = invalid τ
          apply(rule foundation22[THEN iffD1])
          by(erule StrongEq-L-subst2-rev, simp,simp)

 have B2: ⋀τ. τ ⊨ (X ≜ null) ⟹ X−>including(x)−>excluding(y) τ = invalid τ
          apply(rule foundation22[THEN iffD1])
          by(erule StrongEq-L-subst2-rev, simp,simp)

 note [simp] = cp-StrictRefEq [THEN allI[THEN allI[THEN allI[THEN cpI2]], of StrictRefEq]]

 have C: ⋀τ. τ ⊨ (x ≜ invalid) ⟹
         (X−>including(x)−>excluding(y)) τ =
         (if x ≐ y then X−>excluding(y) else X−>excluding(y)−>including(x) endif) τ
          apply(rule foundation22[THEN iffD1])
          apply(erule StrongEq-L-subst2-rev,simp,simp)
          by(simp add: strict1)

 have D: ⋀τ. τ ⊨ (y ≜ invalid) ⟹
         (X−>including(x)−>excluding(y)) τ =
         (if x ≐ y then X−>excluding(y) else X−>excluding(y)−>including(x) endif) τ
          apply(rule foundation22[THEN iffD1])
          apply(erule StrongEq-L-subst2-rev,simp,simp)
          by (simp add: strict2)

 have E: ⋀τ. τ ⊨ υ x ⟹ τ ⊨ υ y ⟹
           (if x ≐ y then X−>excluding(y) else X−>excluding(y)−>including(x) endif) τ =
           (if x ≜ y then X−>excluding(y) else X−>excluding(y)−>including(x) endif) τ
         apply(subst cp-OclIf)
         apply(subst StrictRefEq-vs-StrongEq[THEN foundation22[THEN iffD1]])
         by(simp-all add: cp-OclIf[symmetric])

 have F: ⋀τ. τ ⊨ δ X ⟹ τ ⊨ υ x ⟹ τ ⊨ (x ≜ y) ⟹
```

(*X*−>*including*(*x*)−>*excluding*(*y*) τ) = (*X*−>*excluding*(*y*) τ)
  apply(*drule StrongEq-L-sym*)
  apply(*rule foundation22*[*THEN iffD1*])
  apply(*erule StrongEq-L-subst2-rev*,*simp*)
  by(*simp add*: *OclExcluding-charn2*)

show *?thesis*
  apply(*rule ext*, *rename-tac* τ)
  apply(*case-tac* $\neg$ ($\tau \models$ (δ *X*)), *simp add*:*defined-split*,*elim disjE A1 B1 A2 B2*)
  apply(*case-tac* $\neg$ ($\tau \models$ (υ *x*)),
    simp add:*foundation18 foundation22*[*symmetric*],
    drule StrongEq-L-sym)
  apply(*simp add*: *foundation22 C*)
  apply(*case-tac* $\neg$ ($\tau \models$ (υ *y*)),
    simp add:*foundation18 foundation22*[*symmetric*],
    drule StrongEq-L-sym, *simp add*: *foundation22 D*, *simp*)
  apply(*subst E*,*simp-all*)
  apply(*case-tac* $\tau \models$ *not* ($x \triangleq y$))
  apply(*simp add*: *OclExcluding-charn1*[*simplified foundation22*]
      OclExcluding-charn2[*simplified foundation22*])
  apply(*simp add*: *foundation9 F*)
done
qed

schematic-lemma *OclExcluding-charn-exec*$_{Integer}$[*simp*,*code-unfold*]: *?X*
by(*rule OclExcluding-charn-exec*[*OF StrictRefEq*$_{Integer}$.*strict1 StrictRefEq*$_{Integer}$.*strict2*
      StrictRefEq$_{Integer}$.*defined-args-valid*
      StrictRefEq$_{Integer}$.*cp0 StrictRefEq*$_{Integer}$.*StrictRefEq-vs-StrongEq*], *simp-all*)

schematic-lemma *OclExcluding-charn-exec*$_{Boolean}$[*simp*,*code-unfold*]: *?X*
by(*rule OclExcluding-charn-exec*[*OF StrictRefEq*$_{Boolean}$.*strict1 StrictRefEq*$_{Boolean}$.*strict2*
      StrictRefEq$_{Boolean}$.*defined-args-valid*
      StrictRefEq$_{Boolean}$.*cp0 StrictRefEq*$_{Boolean}$.*StrictRefEq-vs-StrongEq*], *simp-all*)

schematic-lemma *OclExcluding-charn-exec*$_{Set}$[*simp*,*code-unfold*]: *?X*
by(*rule OclExcluding-charn-exec*[*OF StrictRefEq*$_{Set}$.*strict1 StrictRefEq*$_{Set}$.*strict2*
      StrictRefEq$_{Set}$.*defined-args-valid*
      StrictRefEq$_{Set}$.*cp0 StrictRefEq*$_{Set}$.*StrictRefEq-vs-StrongEq*], *simp-all*)

Execution Rules on OclIncludes **lemma** *OclIncludes-charn0*[*simp*]:
assumes *val-x*:$\tau \models$ (υ *x*)
shows $\tau \models$ *not*(*Set*{}−>*includes*(*x*))
using *val-x*
apply(*auto simp*: *OclValid-def OclIncludes-def OclNot-def false-def true-def*)
apply(*auto simp*: *mtSet-def Set*$_{base}$.*Abs-Set*$_{base}$*-inverse*)

```
done


lemma OclIncludes-charn0′[simp,code-unfold]:
Set{}−>includes(x) = (if υ x then false else invalid endif)
proof −
 have A: ⋀ τ. (Set{}−>includes(invalid)) τ = (if (υ invalid) then false else invalid endif) τ
       by simp
 have B: ⋀ τ x. τ ⊨ (υ x) ⟹ (Set{}−>includes(x)) τ = (if υ x then false else invalid endif) τ
       apply(frule OclIncludes-charn0, simp add: OclValid-def)
       apply(rule foundation21[THEN fun-cong, simplified StrongEq-def,simplified,
                 THEN iffD1, of - - false])
       by simp
 show ?thesis
   apply(rule ext, rename-tac τ)
   apply(case-tac τ ⊨ (υ x))
    apply(simp-all add: B foundation18)
   apply(subst cp-OclIncludes, simp add: cp-OclIncludes[symmetric] A)
 done
qed


lemma OclIncludes-charn1:
assumes def-X:τ ⊨ (δ X)
assumes val-x:τ ⊨ (υ x)
shows        τ ⊨ (X−>including(x)−>includes(x))
proof −
 have C : ⌊⌊insert (x τ) ⌈⌈Rep-Set_base (X τ)⌉⌉⌋⌋ ∈ {X. X = bot ∨ X = null ∨ (∀x∈⌈⌈X⌉⌉. x ≠ bot)}
       by(insert def-X val-x, frule Set-inv-lemma, simp add: foundation18 invalid-def)
 show ?thesis
  apply(subst OclIncludes-def, simp add: foundation10[simplified OclValid-def] OclValid-def
                       def-X[simplified OclValid-def] val-x[simplified OclValid-def])
  apply(simp add: OclIncluding-def def-X[simplified OclValid-def] val-x[simplified OclValid-def]
              Abs-Set_base-inverse[OF C] true-def)
 done
qed


lemma OclIncludes-charn2:
assumes def-X:τ ⊨ (δ X)
and    val-x:τ ⊨ (υ x)
and    val-y:τ ⊨ (υ y)
and    neq  :τ ⊨ not(x ≜ y)
shows        τ ⊨ (X−>including(x)−>includes(y)) ≜ (X−>includes(y))
proof −
 have C : ⌊⌊insert (x τ) ⌈⌈Rep-Set_base (X τ)⌉⌉⌋⌋ ∈ {X. X = bot ∨ X = null ∨ (∀x∈⌈⌈X⌉⌉. x ≠ bot)}
       by(insert def-X val-x, frule Set-inv-lemma, simp add: foundation18 invalid-def)
 show ?thesis
  apply(subst OclIncludes-def,
       simp add: def-X[simplified OclValid-def] val-x[simplified OclValid-def]
                val-y[simplified OclValid-def] foundation10[simplified OclValid-def]
                OclValid-def StrongEq-def)
  apply(simp add: OclIncluding-def OclIncludes-def def-X[simplified OclValid-def]
                val-x[simplified OclValid-def] val-y[simplified OclValid-def]
                Abs-Set_base-inverse[OF C] true-def)
 by(metis foundation22 foundation6 foundation9 neq)
qed
```

Here is again a generic theorem similar as above.

```
lemma OclIncludes-execute-generic:
assumes strict1: (invalid ≐ y) = invalid
and     strict2: (x ≐ invalid) = invalid
and     cp-StrictRefEq: ⋀ (X::('𝔄,'a::null)val) Y τ. (X ≐ Y) τ = ((λ-. X τ) ≐ (λ-. Y τ)) τ
and     StrictRefEq-vs-StrongEq: ⋀ (x::('𝔄,'a::null)val) y τ.
                              τ ⊨ υ x ⟹ τ ⊨ υ y ⟹ (τ ⊨ ((x ≐ y) ≜ (x ≜ y)))
shows
     (X->including(x::('𝔄,'a::null)val)->includes(y)) =
     (if δ X then if x ≐ y then true else X->includes(y) endif else invalid endif)
proof −
  have A: ⋀τ. τ ⊨ (X ≜ invalid) ⟹
          (X->including(x)->includes(y)) τ = invalid τ
          apply(rule foundation22[THEN iffD1])
          by(erule StrongEq-L-subst2-rev,simp,simp)
  have B: ⋀τ. τ ⊨ (X ≜ null) ⟹
          (X->including(x)->includes(y)) τ = invalid  τ
          apply(rule foundation22[THEN iffD1])
          by(erule StrongEq-L-subst2-rev,simp,simp)

  note [simp] = cp-StrictRefEq [THEN allI[THEN allI[THEN allI[THEN cpI2]], of StrictRefEq]]

  have C: ⋀τ. τ ⊨ (x ≜ invalid) ⟹
          (X->including(x)->includes(y)) τ =
          (if x ≐ y then true else X->includes(y) endif) τ
          apply(rule foundation22[THEN iffD1])
          apply(erule StrongEq-L-subst2-rev,simp,simp)
          by (simp add: strict1)
  have D:⋀τ. τ ⊨ (y ≜ invalid) ⟹
          (X->including(x)->includes(y)) τ =
          (if x ≐ y then true else X->includes(y) endif) τ
          apply(rule foundation22[THEN iffD1])
          apply(erule StrongEq-L-subst2-rev,simp,simp)
          by (simp add: strict2)
  have E: ⋀τ. τ ⊨ υ x ⟹ τ ⊨ υ y ⟹
            (if x ≐ y then true else X->includes(y) endif) τ =
            (if x ≜ y then true else X->includes(y) endif) τ
```

```
        apply(subst cp-OclIf)
        apply(subst StrictRefEq-vs-StrongEq[THEN foundation22[THEN iffD1]])
        by(simp-all add: cp-OclIf[symmetric])
  have F: ⋀τ. τ ⊨ (x ≜ y) ⟹
            (X−>including(x)−>includes(y)) τ = (X−>including(x)−>includes(x)) τ
        apply(rule foundation22[THEN iffD1])
        by(erule StrongEq-L-subst2-rev,simp, simp)
  show ?thesis
   apply(rule ext, rename-tac τ)
   apply(case-tac ¬ (τ ⊨ (δ X)), simp add:defined-split,elim disjE A B)
   apply(case-tac ¬ (τ ⊨ (υ x)),
        simp add:foundation18 foundation22[symmetric],
        drule StrongEq-L-sym)
    apply(simp add: foundation22 C)
   apply(case-tac ¬ (τ ⊨ (υ y)),
        simp add:foundation18 foundation22[symmetric],
        drule StrongEq-L-sym, simp add: foundation22 D, simp)
   apply(subst E,simp-all)
   apply(case-tac τ ⊨ not(x ≜ y))
    apply(simp add: OclIncludes-charn2[simplified foundation22])
   apply(simp add: foundation9 F
               OclIncludes-charn1[THEN foundation13[THEN iffD2],
                           THEN foundation22[THEN iffD1]])
  done
qed


schematic-lemma OclIncludes-execute_Integer[simp,code-unfold]: ?X
by(rule OclIncludes-execute-generic[OF StrictRefEq_Integer.strict1 StrictRefEq_Integer.strict2
                              StrictRefEq_Integer.cp0
                              StrictRefEq_Integer.StrictRefEq-vs-StrongEq], simp-all)


schematic-lemma OclIncludes-execute_Boolean[simp,code-unfold]: ?X
by(rule OclIncludes-execute-generic[OF StrictRefEq_Boolean.strict1 StrictRefEq_Boolean.strict2
                              StrictRefEq_Boolean.cp0
                              StrictRefEq_Boolean.StrictRefEq-vs-StrongEq], simp-all)


schematic-lemma OclIncludes-execute_Set[simp,code-unfold]: ?X
by(rule OclIncludes-execute-generic[OF StrictRefEq_Set.strict1 StrictRefEq_Set.strict2
                              StrictRefEq_Set.cp0
                              StrictRefEq_Set.StrictRefEq-vs-StrongEq], simp-all)

lemma OclIncludes-including-generic :
 assumes OclIncludes-execute-generic [simp] : ⋀X x y.
        (X−>including(x::('𝔄,'a::null)val)−>includes(y)) =
```

$(\mathit{if}\ \delta\ X\ \mathit{then\ if}\ x \doteq y\ \mathit{then\ true\ else}\ X{-}{>}\mathit{includes}(y)\ \mathit{endif\ else\ invalid\ endif})$
and *StrictRefEq-strict''* : $\bigwedge x\ y.\ \delta\ ((x{::}({}'\mathfrak{A},{}'a{::}\mathit{null})\mathit{val}) \doteq y) = (\upsilon(x)\ \mathit{and}\ \upsilon(y))$
and *a-val* : $\tau \models \upsilon\ a$
and *x-val* : $\tau \models \upsilon\ x$
and *S-incl* : $\tau \models (S){-}{>}\mathit{includes}((x{::}({}'\mathfrak{A},{}'a{::}\mathit{null})\mathit{val}))$
shows $\tau \models S{-}{>}\mathit{including}((a{::}({}'\mathfrak{A},{}'a{::}\mathit{null})\mathit{val})){-}{>}\mathit{includes}(x)$
proof –
have *discr-eq-bot1-true* : $\bigwedge\tau.\ (\bot\ \tau = \mathit{true}\ \tau) = \mathit{False}$
by (*metis bot-fun-def foundation1 foundation18' valid3*)
have *discr-eq-bot2-true* : $\bigwedge\tau.\ (\bot = \mathit{true}\ \tau) = \mathit{False}$
by (*metis bot-fun-def discr-eq-bot1-true*)
have *discr-neq-invalid-true* : $\bigwedge\tau.\ (\mathit{invalid}\ \tau \neq \mathit{true}\ \tau) = \mathit{True}$
by (*metis discr-eq-bot2-true invalid-def*)
have *discr-eq-invalid-true* : $\bigwedge\tau.\ (\mathit{invalid}\ \tau = \mathit{true}\ \tau) = \mathit{False}$
by (*metis bot-option-def invalid-def option.simps(2) true-def*)
show *?thesis*
apply(*simp*)
apply(*subgoal-tac* $\tau \models \delta\ S$)
prefer *2*
apply(*insert S-incl*[*simplified OclIncludes-def*], *simp add*: *OclValid-def*)
apply(*metis discr-eq-bot2-true*)
apply(*simp add*: *cp-OclIf*[*of* δ *S*] *OclValid-def OclIf-def x-val*[*simplified OclValid-def*]
discr-neq-invalid-true discr-eq-invalid-true)
by (*metis OclValid-def S-incl StrictRefEq-strict'' a-val foundation10 foundation6 x-val*)
qed

lemmas *OclIncludes-including*$_{Integer}$ =
OclIncludes-including-generic[*OF OclIncludes-execute*$_{Integer}$ *StrictRefEq*$_{Integer}$*.def-homo*]

Execution Rules on OclExcludes **lemma** *OclExcludes-charn1*:
assumes *def-X*:$\tau \models (\delta\ X)$
assumes *val-x*:$\tau \models (\upsilon\ x)$
shows $\tau \models (X{-}{>}\mathit{excluding}(x){-}{>}\mathit{excludes}(x))$
proof –
let $?\mathit{OclSet} = \lambda S.\ \lfloor\lfloor S\rfloor\rfloor \in \{X.\ X = \bot \lor X = \mathit{null} \lor (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq \bot)\}$
have *diff-in-Set*$_{base}$: $?\mathit{OclSet}\ (\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil - \{x\ \tau\})$
apply(*simp*, (*rule disjI2*)+)
by (*metis* (*hide-lams*, *no-types*) *Diff-iff Set-inv-lemma def-X*)

show *?thesis*
apply(*subst OclExcludes-def*, *simp add*: *foundation10*[*simplified OclValid-def*] *OclValid-def*
def-X[*simplified OclValid-def*] *val-x*[*simplified OclValid-def*])
apply(*subst OclIncludes-def*, *simp add*: *OclNot-def*)
apply(*simp add*: *OclExcluding-def def-X*[*simplified OclValid-def*] *val-x*[*simplified OclValid-def*]
Abs-Set$_{base}$*-inverse*[*OF diff-in-Set*$_{base}$] *true-def*)
by(*simp add*: *OclAnd-def def-X*[*simplified OclValid-def*] *val-x*[*simplified OclValid-def*] *true-def*)
qed

Execution Rules on OclSize **lemma** [*simp*,*code-unfold*]: $Set\{\} \;{-}{>}size() = \mathbf{0}$
apply(*rule ext*)
apply(*simp add*: *defined-def mtSet-def OclSize-def*
bot-Set$_{base}$*-def bot-fun-def*
null-Set$_{base}$*-def null-fun-def*)
apply(*subst Abs-Set*$_{base}$*-inject*, *simp-all add*: *bot-option-def null-option-def*) +
by(*simp add*: *Abs-Set*$_{base}$*-inverse bot-option-def null-option-def OclInt0-def*)

lemma *OclSize-including-exec*[*simp*,*code-unfold*]:
$((X \;{-}{>}including(x)) \;{-}{>}size()) = (if\ \delta\ X\ and\ \upsilon\ x\ then$
$X \;{-}{>}size() +_{int} if\ X \;{-}{>}includes(x)\ then\ \mathbf{0}\ else\ \mathbf{1}\ endif$
else
invalid
endif)
proof −

have *valid-inject-true* : $\bigwedge\tau\ P.\ (\upsilon\ P)\ \tau \neq true\ \tau \Longrightarrow (\upsilon\ P)\ \tau = false\ \tau$
apply(*simp add*: *valid-def true-def false-def bot-fun-def bot-option-def*
null-fun-def null-option-def)
by (*case-tac* $P\ \tau = \bot$, *simp-all add*: *true-def*)
have *defined-inject-true* : $\bigwedge\tau\ P.\ (\delta\ P)\ \tau \neq true\ \tau \Longrightarrow (\delta\ P)\ \tau = false\ \tau$
apply(*simp add*: *defined-def true-def false-def bot-fun-def bot-option-def*
null-fun-def null-option-def)
by (*case-tac* $P\ \tau = \bot \vee P\ \tau = null$, *simp-all add*: *true-def*)

show *?thesis*
apply(*rule ext*, *rename-tac* τ)
proof −
fix τ
have *includes-notin*: $\neg\ \tau \models X{-}{>}includes(x) \Longrightarrow (\delta\ X)\ \tau = true\ \tau \wedge (\upsilon\ x)\ \tau = true\ \tau \Longrightarrow$
$x\ \tau \notin \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil$
by(*simp add*: *OclIncludes-def OclValid-def true-def*)

have *includes-def*: $\tau \models X{-}{>}includes(x) \Longrightarrow \tau \models \delta\ X$
by (*metis bot-fun-def OclIncludes-def OclValid-def defined3 foundation16*)

have *includes-val*: $\tau \models X{-}{>}includes(x) \Longrightarrow \tau \models \upsilon\ x$
by (*metis* (*hide-lams*, *no-types*) *foundation6*
OclIncludes-valid-args-valid$'$ *OclIncluding-valid-args-valid OclIncluding-valid-args-valid*$''$)

have *ins-in-Set*$_{base}$: $\tau \models \delta\ X \Longrightarrow \tau \models \upsilon\ x \Longrightarrow$
$\lfloor\lfloor insert\ (x\ \tau)\ \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil\rfloor\rfloor \in \{X.\ X = \bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq \bot)\}$
apply(*simp add*: *bot-option-def null-option-def*)
by (*metis* (*hide-lams*, *no-types*) *Set-inv-lemma foundation18*$'$ *foundation5*)

have $m : \bigwedge\tau.\ (\lambda\text{-}.\ \bot) = (\lambda\text{-}.\ invalid\ \tau)$ **by**(*rule ext*, *simp add*:*invalid-def*)

show $X{-}{>}including(x){-}{>}size()\ \tau = (if\ \ \delta\ X\ and\ \upsilon\ x$

```
                          then X−>size() +_int if X−>includes(x) then 0 else 1 endif
                          else invalid endif) τ
  apply(case-tac τ ⊨ δ X and υ x, simp)
   apply(subst OclAdd_Integer.cp0)
   apply(case-tac τ ⊨ X−>includes(x), simp add: OclAdd_Integer.cp0[symmetric])
    apply(case-tac τ ⊨ ((υ (X−>size())) and not (δ (X−>size()))), simp)
     apply(drule foundation5[where P = υ X−>size()], erule conjE)
     apply(drule OclSize-infinite)
     apply(frule includes-def, drule includes-val, simp)
     apply(subst OclSize-def, subst OclIncluding-finite-rep-set, assumption+)
    apply (metis (hide-lams, no-types) invalid-def)

    apply(subst OclIf-false′,
          metis (hide-lams, no-types) defined5 defined6 defined-and-I defined-not-I
                              foundation1 foundation9)
   apply(subst cp-OclSize, simp add: OclIncluding-includes0 cp-OclSize[symmetric])

   apply(subst OclIf-false′, subst foundation9,
         metis (hide-lams, no-types) OclIncludes-valid-args-valid′, simp, simp add: OclSize-def)
   apply(drule foundation5)
   apply(subst (1 2) OclIncluding-finite-rep-set, fast+)
   apply(subst (1 2) cp-OclAnd, subst (1 2) OclAdd_Integer.cp0, simp)
   apply(rule conjI)
    apply(simp add: OclIncluding-def)
    apply(subst Abs-Set_base-inverse[OF ins-in-Set_base], fast+)
    apply(subst (asm) (2 3) OclValid-def, simp add: OclAdd_Integer-def OclInt1-def)
    apply(rule impI)
    apply(drule Finite-Set.card.insert[where x = x τ])
    apply(rule includes-notin, simp, simp)
    apply (metis Suc-eq-plus1 int-1 of-nat-add)

   apply(subst (1 2) m[of τ], simp only:   OclAdd_Integer.cp0[symmetric],simp, simp add:invalid-def)
   apply(subst OclIncluding-finite-rep-set, fast+, simp add: OclValid-def)

  apply(subst OclIf-false′, metis (hide-lams, no-types) defined6 foundation1 foundation9
                                    OclExcluding-valid-args-valid′′)
 by (metis cp-OclSize foundation18′ OclIncluding-valid-args-valid′′ invalid-def OclSize-invalid)
 qed
qed
```

Execution Rules on OclIsEmpty

```
lemma [simp,code-unfold]: Set{}−>isEmpty() = true
by(simp add: OclIsEmpty-def)

lemma OclIsEmpty-including [simp]:
assumes X-def: τ ⊨ δ X
   and X-finite: finite ⌈⌈Rep-Set_base (X τ)⌉⌉
   and a-val: τ ⊨ υ a
shows X−>including(a)−>isEmpty() τ = false τ
```

proof –
have *A1* : $\bigwedge\tau\, X.\ X\ \tau = true\ \tau \vee X\ \tau = false\ \tau \Longrightarrow (X\ and\ not\ X)\ \tau = false\ \tau$
by (*metis* (*no-types*) *OclAnd-false1 OclAnd-idem OclImplies-def OclNot3 OclNot-not OclOr-false1*
cp-OclAnd cp-OclNot deMorgan1 deMorgan2)

have *defined-inject-true* : $\bigwedge\tau\, P.\ (\delta\ P)\ \tau \neq true\ \tau \Longrightarrow (\delta\ P)\ \tau = false\ \tau$
apply(*simp add*: *defined-def true-def false-def bot-fun-def bot-option-def*
null-fun-def null-option-def)
by (*case-tac* $P\ \tau = \bot \vee P\ \tau = null$, *simp-all add*: *true-def*)

have *B* : $\bigwedge X\ \tau.\ \tau \models \upsilon\ X \Longrightarrow X\ \tau \neq \mathbf{0}\ \tau \Longrightarrow (X \doteq \mathbf{0})\ \tau = false\ \tau$
apply(*simp add*: *foundation22*[*symmetric*] *foundation14 foundation9*)
apply(*erule StrongEq-L-subst4-rev*[*THEN iffD2*, *OF* $StrictRefEq_{Integer}$*.StrictRefEq-vs-StrongEq*])
by(*simp-all*)

show *?thesis*
apply(*simp add*: *OclIsEmpty-def del*: *OclSize-including-exec*)
apply(*subst cp-OclOr*, *subst A1*)
apply(*metis* (*hide-lams*, *no-types*) *defined-inject-true OclExcluding-valid-args-valid*′)
apply(*simp add*: *cp-OclOr*[*symmetric*] *del*: *OclSize-including-exec*)
apply(*rule B*,
rule foundation20,
metis (*hide-lams*, *no-types*) *OclIncluding-defined-args-valid OclIncluding-finite-rep-set*
X-def X-finite a-val size-defined′)
apply(*simp add*: *OclSize-def OclIncluding-finite-rep-set*[*OF X-def a-val*] *X-finite OclInt0-def*)
by (*metis OclValid-def X-def a-val foundation10 foundation6*
OclIncluding-notempty-rep-set[*OF X-def a-val*])
qed

Execution Rules on OclNotEmpty **lemma** [*simp*,*code-unfold*]: *Set*{}−>*notEmpty*() = *false*
by(*simp add*: *OclNotEmpty-def*)

lemma *OclNotEmpty-including* [*simp*,*code-unfold*]:
assumes *X-def*: $\tau \models \delta\ X$
and *X-finite*: *finite* $\lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil$
and *a-val*: $\tau \models \upsilon\ a$
shows *X*−>*including*(*a*)−>*notEmpty*() $\tau = true\ \tau$
apply(*simp add*: *OclNotEmpty-def*)
apply(*subst cp-OclNot*, *subst OclIsEmpty-including*, *simp-all add*: *assms*)
by (*metis OclNot4 cp-OclNot*)

Execution Rules on OclANY **lemma** [*simp*,*code-unfold*]: *Set*{}−>*any*() = *null*
by(*rule ext*, *simp add*: *OclANY-def*, *simp add*: *false-def true-def*)

lemma *OclANY-singleton-exec*[*simp*,*code-unfold*]:
(*Set*{}−>*including*(*a*))−>*any*() = *a*
apply(*rule ext*, *rename-tac* τ, *simp add*: *mtSet-def OclANY-def*)
apply(*case-tac* $\tau \models \upsilon\ a$)

apply(*simp add*: *OclValid-def mtSet-defined*[*simplified mtSet-def*]
mtSet-valid[*simplified mtSet-def*] *mtSet-rep-set*[*simplified mtSet-def*])
apply(*subst* (*1 2*) *cp-OclAnd*,
subst (*1 2*) *OclNotEmpty-including*[**where** $X = Set\{\}$, *simplified mtSet-def*])
apply(*simp add*: *mtSet-defined*[*simplified mtSet-def*])
apply(*metis* (*hide-lams*, *no-types*) *finite.emptyI mtSet-def mtSet-rep-set*)
apply(*simp add*: *OclValid-def*)
apply(*simp add*: *OclIncluding-def*)
apply(*rule conjI*)
apply(*subst* (*1 2*) *Abs-Set*$_{base}$*-inverse*, *simp add*: *bot-option-def null-option-def*)
apply(*simp*, *metis OclValid-def foundation18′*)
apply(*simp*)
apply(*simp add*: *mtSet-defined*[*simplified mtSet-def*])

apply(*subgoal-tac* $a\ \tau = \bot$)
prefer *2*
apply(*simp add*: *OclValid-def valid-def bot-fun-def split*: *split-if-asm*)
apply(*simp*)
apply(*subst* (*1 2 3 4*) *cp-OclAnd*,
simp add: *mtSet-defined*[*simplified mtSet-def*] *valid-def bot-fun-def*)
by(*simp add*: *cp-OclAnd*[*symmetric*], *rule impI*, *simp add*: *false-def true-def*)

Execution Rules on OclForall **lemma** *OclForall-mtSet-exec*[*simp*,*code-unfold*] :((*Set*{})−>*forAll*(*z*| *P*(*z*))) = *true*
apply(*simp add*: *OclForall-def*)
apply(*subst mtSet-def*)+
apply(*subst Abs-Set*$_{base}$*-inverse*, *simp-all add*: *true-def*)+
done

The following rule is a main theorem of our approach: From a denotational definition that assures consistency, but may be — as in the case of the *OclForall X P* — dauntingly complex, we derive operational rules that can serve as a gold-standard for operational execution, since they may be evaluated in whatever situation and according to whatever strategy. In the case of *OclForall X P*, the operational rule gives immediately a way to evaluation in any finite (in terms of conventional OCL: denotable) set, although the rule also holds for the infinite case:

$Integer_{null}{-}{>}forAll(x|Integer_{null}{-}{>}forAll(y|x +_{int} y \triangleq y +_{int} x))$
or even:
$Integer{-}{>}forAll(x|Integer{-}{>}forAll(y|x +_{int} y \doteq y +_{int} x))$
are valid OCL statements in any context τ.

theorem *OclForall-including-exec*[*simp*,*code-unfold*] :
assumes *cp0* : *cp P*
shows ((*S*−>*including*(*x*))−>*forAll*(*z* | *P*(*z*))) = (*if* δ *S and* υ *x*
then P x and (*S*−>*forAll*(*z* | *P*(*z*)))
else invalid
endif)
proof −
have *cp*: $\bigwedge\tau.\ P\ x\ \tau = P\ (\lambda\text{-}.\ x\ \tau)\ \tau$ **by**(*insert cp0*, *auto simp*: *cp-def*)

```
have cp-eq : ⋀τ v. (P x τ = v) = (P (λ-. x τ) τ = v) by(subst cp, simp)

have cp-OclNot-eq : ⋀τ v. (P x τ ≠ v) = (P (λ-. x τ) τ ≠ v) by(subst cp, simp)

have insert-in-Set_base : ⋀τ. (τ ⊨(δ S)) ⟹ (τ ⊨(υ x)) ⟹
                    ⌊⌊insert (x τ) ⌈⌈Rep-Set_base (S τ)⌉⌉⌋⌋ ∈
                     {X. X = bot ∨ X = null ∨ (∀x∈⌈⌈X⌉⌉. x ≠ bot)}
      by(frule Set-inv-lemma, simp add: foundation18 invalid-def)

have forall-including-invert : ⋀τ f. (f x τ = f (λ -. x τ) τ) ⟹
                            τ ⊨ (δ S and υ x) ⟹
                            (∀x∈⌈⌈Rep-Set_base (S->including(x) τ)⌉⌉. f (λ-. x) τ) =
                             (f x τ ∧ (∀x∈⌈⌈Rep-Set_base (S τ)⌉⌉. f (λ-. x) τ))
      apply(drule foundation5, simp add: OclIncluding-def)
      apply(subst Abs-Set_base-inverse)
      apply(rule insert-in-Set_base, fast+)
      by(simp add: OclValid-def)

have exists-including-invert : ⋀τ f. (f x τ = f (λ -. x τ) τ) ⟹
                            τ ⊨ (δ S and υ x) ⟹
                            (∃x∈⌈⌈Rep-Set_base (S->including(x) τ)⌉⌉. f (λ-. x) τ) =
                             (f x τ ∨ (∃x∈⌈⌈Rep-Set_base (S τ)⌉⌉. f (λ-. x) τ))
      apply(subst arg-cong[where f = λx. ¬x,
                  OF forall-including-invert[where f = λx τ. ¬ (f x τ)],
                  simplified])
      by simp-all

have contradict-Rep-Set_base: ⋀τ S f. ∃x∈⌈⌈Rep-Set_base S⌉⌉. f (λ-. x) τ ⟹
                          (∀x∈⌈⌈Rep-Set_base S⌉⌉. ¬ (f (λ-. x) τ)) = False
      by(case-tac (∀x∈⌈⌈Rep-Set_base S⌉⌉. ¬ (f (λ-. x) τ)) = True, simp-all)

have bot-invalid : ⊥ = invalid  by(rule ext, simp add: invalid-def bot-fun-def)

have bot-invalid2 : ⋀τ. ⊥ = invalid τ  by(simp add: invalid-def)

have C1 : ⋀τ. P x τ = false τ ∨ (∃x∈⌈⌈Rep-Set_base (S τ)⌉⌉. P (λ-. x) τ = false τ) ⟹
          τ ⊨ (δ S and υ x) ⟹
          false τ = (P x and OclForall S P) τ
      apply(simp add: cp-OclAnd[of P x])
      apply(elim disjE, simp)
       apply(simp only: cp-OclAnd[symmetric], simp)
      apply(subgoal-tac OclForall S P τ = false τ)
       apply(simp only: cp-OclAnd[symmetric], simp)
      apply(simp add: OclForall-def)
      apply(fold OclValid-def, simp add:  foundation27)
      done

have C2 : ⋀τ. τ ⊨ (δ S and υ x) ⟹
```

```
           P x τ = null τ ∨ (∃x∈⌈⌈Rep-Set_base (S τ)⌉⌉. P (λ-. x) τ = null τ) ⟹
           P x τ = invalid τ ∨ (∃x∈⌈⌈Rep-Set_base (S τ)⌉⌉. P (λ-. x) τ = invalid τ) ⟹
           ∀x∈⌈⌈Rep-Set_base (S−>including(x) τ)⌉⌉. P (λ-. x) τ ≠ false τ ⟹
           invalid τ = (P x and OclForall S P) τ
      apply(subgoal-tac (δ S)τ = true τ)
       prefer 2 apply(simp add: foundation27, simp add: OclValid-def)
      apply(drule forall-including-invert[of λ x τ. P x τ ≠ false τ, OF cp-OclNot-eq, THEN iffD1])
       apply(assumption)
      apply(simp add: cp-OclAnd[of P x],elim disjE, simp-all)
         apply(simp add: invalid-def null-fun-def null-option-def bot-fun-def bot-option-def)
        apply(subgoal-tac OclForall S P τ = invalid τ)
         apply(simp only:cp-OclAnd[symmetric],simp,simp add:invalid-def bot-fun-def)
        apply(unfold OclForall-def, simp add: invalid-def false-def bot-fun-def,simp)
       apply(simp add:cp-OclAnd[symmetric],simp)
      apply(erule conjE)
      apply(subgoal-tac (P x τ = invalid τ) ∨ (P x τ = null τ) ∨ (P x τ = true τ) ∨ (P x τ = false τ))
       prefer 2 apply(rule bool-split-0)
      apply(elim disjE, simp-all)
       apply(simp only:cp-OclAnd[symmetric],simp)+
      done

have A : ⋀τ. τ ⊨ (δ S and υ x) ⟹
          OclForall (S−>including(x)) P τ = (P x and OclForall S P) τ
    proof − fix τ
          assume 0 : τ ⊨ (δ S and υ x)
          let ?S = λocl. P x τ ≠ ocl τ ∧ (∀x∈⌈⌈Rep-Set_base (S τ)⌉⌉. P (λ-. x) τ ≠ ocl τ)
          let ?S' = λocl. ∀x∈⌈⌈Rep-Set_base (S−>including(x) τ)⌉⌉. P (λ-. x) τ ≠ ocl τ
          let ?assms-1 = ?S' null
          let ?assms-2 = ?S' invalid
          let ?assms-3 = ?S' false
          have 4 : ?assms-3 ⟹ ?S false
             apply(subst forall-including-invert[of λ x τ. P x τ ≠ false τ,symmetric])
             by(simp-all add: cp-OclNot-eq 0)
          have 5 : ?assms-2 ⟹ ?S invalid
             apply(subst forall-including-invert[of λ x τ. P x τ ≠ invalid τ,symmetric])
             by(simp-all add: cp-OclNot-eq 0)
          have 6 : ?assms-1 ⟹ ?S null
             apply(subst forall-including-invert[of λ x τ. P x τ ≠ null τ,symmetric])
             by(simp-all add: cp-OclNot-eq 0)
          have 7 : (δ S) τ = true τ
             by(insert 0, simp add: foundation27, simp add: OclValid-def)
    show ?thesis τ
      apply(subst OclForall-def)
      apply(simp add: cp-OclAnd[THEN sym] OclValid-def contradict-Rep-Set_base)
      apply(intro conjI impI,fold OclValid-def)
      apply(simp-all add: exists-including-invert[where f = λ x τ. P x τ = null τ, OF cp-eq])
      apply(simp-all add: exists-including-invert[where f = λ x τ. P x τ = invalid τ, OF cp-eq])
      apply(simp-all add: exists-including-invert[where f = λ x τ. P x τ = false τ, OF cp-eq])
```

```
proof −
  assume 1 : P x τ = null τ ∨ (∃x∈⌈⌈Rep-Set_base (S τ)⌉⌉. P (λ-. x) τ = null τ)
  and   2 : ?assms-2
  and   3 : ?assms-3
  show  null τ = (P x and OclForall S P) τ
  proof −
    note 4 = 4[OF 3]
    note 5 = 5[OF 2]
    have 6 : P x τ = null τ ∨ P x τ = true τ
      by(metis 4 5 bool-split-0)
    show ?thesis
    apply(insert 6, elim disjE)
     apply(subst cp-OclAnd)
     apply(simp add: OclForall-def 7 4[THEN conjunct2] 5[THEN conjunct2])
     apply(simp-all add:cp-OclAnd[symmetric])
    apply(subst cp-OclAnd, simp-all add:cp-OclAnd[symmetric] OclForall-def)
    apply(simp add:4[THEN conjunct2] 5[THEN conjunct2] 0[simplified OclValid-def] 7)
    apply(insert 1, elim disjE, auto)
    done
  qed
next
  assume 1 : ?assms-1
  and   2 : P x τ = invalid τ ∨ (∃x∈⌈⌈Rep-Set_base (S τ)⌉⌉. P (λ-. x) τ = invalid τ)
  and   3 : ?assms-3
  show  invalid τ = (P x and OclForall S P) τ
  proof −
    note 4 = 4[OF 3]
    note 6 = 6[OF 1]
    have 5 : P x τ = invalid τ ∨ P x τ = true τ
      by(metis 4 6 bool-split-0)
    show ?thesis
    apply(insert 5, elim disjE)
     apply(subst cp-OclAnd)
     apply(simp add: OclForall-def 4[THEN conjunct2] 6[THEN conjunct2] 7)
     apply(simp-all add:cp-OclAnd[symmetric])
    apply(subst cp-OclAnd, simp-all add:cp-OclAnd[symmetric] OclForall-def)
    apply(insert 2, elim disjE, simp add: invalid-def true-def bot-option-def)
    apply(simp add: 0[simplified OclValid-def] 4[THEN conjunct2] 6[THEN conjunct2] 7)
    by(auto)
   qed
next
  assume 1 : ?assms-1
  and   2 : ?assms-2
  and   3 : ?assms-3
  show  true τ = (P x and OclForall S P) τ
  proof −
    note 4 = 4[OF 3]
    note 5 = 5[OF 2]
```

```
          note 6 = 6[OF 1]
          have 8 : P x τ = true τ
             by(metis 4 5 6 bool-split-0)
          show ?thesis
          apply(subst cp-OclAnd, simp add: 8 cp-OclAnd[symmetric])
          by(simp add: OclForall-def 4 5 6 7)
        qed
      apply-end( simp add: 0
            | rule C1, simp+
            | rule C2, simp add: 0 )+
      qed
    qed

  have B : ⋀τ. ¬ (τ ⊨ (δ S and υ x)) ⟹
          OclForall (S−>including(x)) P τ = invalid τ
      apply(rule foundation22[THEN iffD1])
      apply(simp only: foundation10′ de-Morgan-conj foundation18″, elim disjE)
       apply(simp add:  defined-split, elim disjE)
        apply(erule StrongEq-L-subst2-rev, simp+)+
      done

  show ?thesis
      apply(rule ext, rename-tac τ)
      apply(simp add: OclIf-def)
      apply(simp add: cp-defined[of δ S and υ x] cp-defined[THEN sym])
      apply(intro conjI impI)
      by(auto intro!: A B simp: OclValid-def)
qed
```

Execution Rules on OclExists

```
lemma OclExists-mtSet-exec[simp,code-unfold] :
((Set{})−>exists(z | P(z))) = false
by(simp add: OclExists-def)

lemma OclExists-including-exec[simp,code-unfold] :
 assumes cp: cp P
 shows ((S−>including(x))−>exists(z | P(z))) = (if δ S and υ x
                                     then P x or (S−>exists(z | P(z)))
                                     else invalid
                                     endif)
 by(simp add: OclExists-def OclOr-def  cp OclNot-inject)
```

Execution Rules on OclIterate

```
lemma OclIterate-empty[simp,code-unfold]: ((Set{})−>iterate(a; x = A | P a x))
= A
proof −
 have C : ⋀ τ. (δ (λτ. Abs-Set_base ⌊⌊{}⌋⌋)) τ = true τ
 by (metis (no-types) defined-def mtSet-def mtSet-defined null-fun-def)
 show ?thesis
    apply(simp add: OclIterate-def mtSet-def Abs-Set_base-inverse valid-def C)
```

apply(*rule ext*, *rename-tac* τ)
apply(*case-tac A* $\tau = \perp \tau$, *simp-all*, *simp add*:*true-def false-def bot-fun-def*)
apply(*simp add*: *Abs-Set*$_{base}$*-inverse*)
done
qed

In particular, this does hold for A = null.

lemma *OclIterate-including*:
assumes *S-finite*: $\tau \models \delta(S{-}{>}size())$
and *F-valid-arg*: $(\upsilon\ A)\ \tau = (\upsilon\ (F\ a\ A))\ \tau$
and *F-commute*: *comp-fun-commute F*
and *F-cp*: $\bigwedge x\ y\ \tau.\ F\ x\ y\ \tau = F\ (\lambda\ \text{-}.\ x\ \tau)\ y\ \tau$
shows $((S{-}{>}including(a)){-}{>}iterate(a; x = \quad A \mid F\ a\ x))\ \tau =$
$((S{-}{>}excluding(a)){-}{>}iterate(a; x = F\ a\ A \mid F\ a\ x))\ \tau$
proof –
have *insert-in-Set*$_{base}$: $\bigwedge\tau.\ (\tau \models(\delta\ S)) \Longrightarrow (\tau \models(\upsilon\ a)) \Longrightarrow$
$\lfloor\lfloor insert\ (a\ \tau)\ \lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil\rfloor\rfloor \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
by(*frule Set-inv-lemma*, *simp add*: *foundation18 invalid-def*)

have *insert-defined* : $\bigwedge\tau.\ (\tau \models(\delta\ S)) \Longrightarrow (\tau \models(\upsilon\ a)) \Longrightarrow$
$(\delta\ (\lambda\text{-}.\ Abs\text{-}Set_{base}\ \lfloor\lfloor insert\ (a\ \tau)\ \lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil\rfloor\rfloor))\ \tau = true\ \tau$
apply(*subst defined-def*)
apply(*simp add*: *bot-Set*$_{base}$*-def bot-fun-def null-Set*$_{base}$*-def null-fun-def*)
by(*subst Abs-Set*$_{base}$*-inject*,
rule insert-in-Set$_{base}$, *simp-all add*: *null-option-def bot-option-def*)+

have *remove-finite* : *finite* $\lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil \Longrightarrow$
finite $((\lambda a\ \tau.\ a)\ \text{‘}\ (\lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil - \{a\ \tau\}))$
by(*simp*)

have *remove-in-Set*$_{base}$: $\bigwedge\tau.\ (\tau \models(\delta\ S)) \Longrightarrow (\tau \models(\upsilon\ a)) \Longrightarrow$
$\lfloor\lfloor\lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil - \{a\ \tau\}\rfloor\rfloor \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
by(*frule Set-inv-lemma*, *simp add*: *foundation18 invalid-def*)

have *remove-defined* : $\bigwedge\tau.\ (\tau \models(\delta\ S)) \Longrightarrow (\tau \models(\upsilon\ a)) \Longrightarrow$
$(\delta\ (\lambda\text{-}.\ Abs\text{-}Set_{base}\ \lfloor\lfloor\lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil - \{a\ \tau\}\rfloor\rfloor))\ \tau = true\ \tau$
apply(*subst defined-def*)
apply(*simp add*: *bot-Set*$_{base}$*-def bot-fun-def null-Set*$_{base}$*-def null-fun-def*)
by(*subst Abs-Set*$_{base}$*-inject*,
rule remove-in-Set$_{base}$, *simp-all add*: *null-option-def bot-option-def*)+

have *abs-rep*: $\bigwedge x.\ \lfloor\lfloor x\rfloor\rfloor \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\} \Longrightarrow$
$\lceil\lceil Rep\text{-}Set_{base}\ (Abs\text{-}Set_{base}\ \lfloor\lfloor x\rfloor\rfloor)\rceil\rceil = x$
by(*subst Abs-Set*$_{base}$*-inverse*, *simp-all*)

have *inject* : *inj* $(\lambda a\ \tau.\ a)$
by(*rule inj-fun*, *simp*)

show *?thesis*
apply(*subst* (*1 2*) *cp-OclIterate*, *subst OclIncluding-def*, *subst OclExcluding-def*)
apply(*case-tac* $\neg$ ((δ *S*) τ = *true* τ $\wedge$ (υ *a*) τ = *true* τ), *simp add*: *invalid-def*)

apply(*subgoal-tac OclIterate* (λ-. $\bot$) *A F* τ = *OclIterate* (λ-. $\bot$) (*F a A*) *F* τ, *simp*)
apply(*rule conjI*, *blast*+)
apply(*simp add*: *OclIterate-def defined-def bot-option-def bot-fun-def false-def true-def*)

apply(*simp add*: *OclIterate-def*)
apply((*subst abs-rep*[*OF insert-in-Set$_{base}$*[*simplified OclValid-def*], *of* τ], *simp-all*)+,
(*subst abs-rep*[*OF remove-in-Set$_{base}$*[*simplified OclValid-def*], *of* τ], *simp-all*)+,
(*subst insert-defined*, *simp-all add*: *OclValid-def*)+,
(*subst remove-defined*, *simp-all add*: *OclValid-def*)+)

apply(*case-tac* $\neg$ ((υ *A*) τ = *true* τ), (*simp add*: *F-valid-arg*)+)
apply(*rule impI*,
subst Finite-Set.*comp-fun-commute*.*fold-fun-left-comm*[*symmetric*, *OF F-commute*],
rule remove-finite, *simp*)

apply(*subst image-set-diff*[*OF inject*], *simp*)
apply(*subgoal-tac Finite-Set*.*fold F A* (*insert* ($\lambda\tau'$. *a* τ) ((λa τ. *a*) ‘ $\lceil\lceil$*Rep-Set$_{base}$* (*S* τ)$\rceil\rceil$)) τ =
F ($\lambda\tau'$. *a* τ) (*Finite-Set*.*fold F A* ((λa τ. *a*) ‘ $\lceil\lceil$*Rep-Set$_{base}$* (*S* τ)$\rceil\rceil$ $-$ {$\lambda\tau'$. *a* τ})) τ)
apply(*subst F-cp*, *simp*)

by(*subst Finite-Set*.*comp-fun-commute*.*fold-insert-remove*[*OF F-commute*], *simp*+)
qed

Execution Rules on OclSelect **lemma** *OclSelect-mtSet-exec*[*simp*,*code-unfold*]: *OclSelect mtSet P* = *mtSet*
apply(*rule ext*, *rename-tac* τ)
apply(*simp add*: *OclSelect-def mtSet-def defined-def false-def true-def*
bot-Set$_{base}$-def bot-fun-def null-Set$_{base}$-def null-fun-def)
by((*subst* (*1 2 3 4 5*) *Abs-Set$_{base}$-inverse*
| *subst Abs-Set$_{base}$-inject*), (*simp add*: *null-option-def bot-option-def*)+)+

definition *OclSelect-body* :: - $\Rightarrow$ - $\Rightarrow$ - $\Rightarrow$ ($'\mathfrak{A}$, $'a$ *option option*) *Set*
$\equiv$ (λ*P x acc*. *if P x* $\doteq$ *false then acc else acc*$-$>*including*(*x*) *endif*)

theorem *OclSelect-including-exec*[*simp*,*code-unfold*]:
assumes *P-cp* : *cp P*
shows *OclSelect* (*X*$-$>*including*(*y*)) *P* = *OclSelect-body P y* (*OclSelect* (*X*$-$>*excluding*(*y*)) *P*)
(**is** - = *?select*)
proof $-$
have *P-cp*: $\bigwedge x$ τ. *P x* τ = *P* (λ-. *x* τ) τ **by**(*insert P-cp*, *auto simp*: *cp-def*)

have *ex-including* : $\bigwedge f$ *X y* τ. $\tau \models \delta$ *X* $\Longrightarrow$ $\tau \models \upsilon$ *y* $\Longrightarrow$
($\exists x\in\lceil\lceil$*Rep-Set$_{base}$* (*X*$-$>*including*(*y*) τ)$\rceil\rceil$. *f* (*P* (λ-. *x*)) τ) =
(*f* (*P* (λ-. *y* τ)) τ $\vee$ ($\exists x\in\lceil\lceil$*Rep-Set$_{base}$* (*X* τ)$\rceil\rceil$. *f* (*P* (λ-. *x*)) τ))
apply(*simp add*: *OclIncluding-def OclValid-def*)

apply(*subst Abs-Set*$_{base}$*-inverse*, *simp*, (*rule disjI2*)+)
by (*metis* (*hide-lams*, *no-types*) *OclValid-def Set-inv-lemma foundation18*$'$,*simp*)

have *al-including* : $\bigwedge f\, X\, y\, \tau.\ \tau \models \delta\ X \Longrightarrow \tau \models \upsilon\ y \Longrightarrow$
$(\forall x{\in}\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X{-}{>}\mathit{including}(y)\ \tau)\rceil\rceil.\ f\ (P\ (\lambda\text{-}.\ x))\ \tau) =$
$(f\ (P\ (\lambda\text{-}.\ y\ \tau))\ \tau \wedge (\forall x{\in}\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil.\ f\ (P\ (\lambda\text{-}.\ x))\ \tau))$
apply(*simp add*: *OclIncluding-def OclValid-def*)
apply(*subst Abs-Set*$_{base}$*-inverse*, *simp*, (*rule disjI2*)+)
by (*metis* (*hide-lams*, *no-types*) *OclValid-def Set-inv-lemma foundation18*$'$, *simp*)

have *ex-excluding1* : $\bigwedge f\, X\, y\, \tau.\ \tau \models \delta\ X \Longrightarrow \tau \models \upsilon\ y \Longrightarrow \neg\ (f\ (P\ (\lambda\text{-}.\ y\ \tau))\ \tau) \Longrightarrow$
$(\exists x{\in}\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil.\ f\ (P\ (\lambda\text{-}.\ x))\ \tau) =$
$(\exists x{\in}\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X{-}{>}\mathit{excluding}(y)\ \tau)\rceil\rceil.\ f\ (P\ (\lambda\text{-}.\ x))\ \tau)$
apply(*simp add*: *OclExcluding-def OclValid-def*)
apply(*subst Abs-Set*$_{base}$*-inverse*, *simp*, (*rule disjI2*)+)
by (*metis* (*no-types*) *Diff-iff OclValid-def Set-inv-lemma*) *auto*

have *al-excluding1* : $\bigwedge f\, X\, y\, \tau.\ \tau \models \delta\ X \Longrightarrow \tau \models \upsilon\ y \Longrightarrow f\ (P\ (\lambda\text{-}.\ y\ \tau))\ \tau \Longrightarrow$
$(\forall x{\in}\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil.\ f\ (P\ (\lambda\text{-}.\ x))\ \tau) =$
$(\forall x{\in}\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X{-}{>}\mathit{excluding}(y)\ \tau)\rceil\rceil.\ f\ (P\ (\lambda\text{-}.\ x))\ \tau)$
apply(*simp add*: *OclExcluding-def OclValid-def*)
apply(*subst Abs-Set*$_{base}$*-inverse*, *simp*, (*rule disjI2*)+)
by (*metis* (*no-types*) *Diff-iff OclValid-def Set-inv-lemma*) *auto*

have *in-including* : $\bigwedge f\, X\, y\, \tau.\ \tau \models \delta\ X \Longrightarrow \tau \models \upsilon\ y \Longrightarrow$
$\{x \in \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X{-}{>}\mathit{including}(y)\ \tau)\rceil\rceil.\ f\ (P\ (\lambda\text{-}.\ x)\ \tau)\} =$
$(\mathit{let}\ s = \{x \in \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil.\ f\ (P\ (\lambda\text{-}.\ x)\ \tau)\}\ \mathit{in}$
$\mathit{if}\ f\ (P\ (\lambda\text{-}.\ y\ \tau)\ \tau)\ \mathit{then}\ \mathit{insert}\ (y\ \tau)\ s\ \mathit{else}\ s)$
apply(*simp add*: *OclIncluding-def OclValid-def*)
apply(*subst Abs-Set*$_{base}$*-inverse*, *simp*, (*rule disjI2*)+)
apply (*metis* (*hide-lams*, *no-types*) *OclValid-def Set-inv-lemma foundation18*$'$)
by(*simp add*: *Let-def*, *auto*)

let $?\mathit{OclSet} = \lambda S.\ \lfloor\lfloor S\rfloor\rfloor \in \{X.\ X = \bot \vee X = \mathit{null} \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq \bot)\}$

have *diff-in-Set*$_{base}$: $\bigwedge\tau.\ (\delta\ X)\ \tau = \mathit{true}\ \tau \Longrightarrow\ ?\mathit{OclSet}\ (\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil - \{y\ \tau\})$
apply(*simp*, (*rule disjI2*)+)
by (*metis* (*mono-tags*) *Diff-iff OclValid-def Set-inv-lemma*)

have *ins-in-Set*$_{base}$: $\bigwedge\tau.\ (\delta\ X)\ \tau = \mathit{true}\ \tau \Longrightarrow (\upsilon\ y)\ \tau = \mathit{true}\ \tau \Longrightarrow$
$?\mathit{OclSet}\ (\mathit{insert}\ (y\ \tau)\ \{x \in \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil.\ P\ (\lambda\text{-}.\ x)\ \tau \neq \mathit{false}\ \tau\})$
apply(*simp*, (*rule disjI2*)+)
by (*metis* (*hide-lams*, *no-types*) *OclValid-def Set-inv-lemma foundation18*$'$)

have *ins-in-Set*$_{base}{}'$: $\bigwedge\tau.\ (\delta\ X)\ \tau = \mathit{true}\ \tau \Longrightarrow (\upsilon\ y)\ \tau = \mathit{true}\ \tau \Longrightarrow$
$?\mathit{OclSet}\ (\mathit{insert}\ (y\ \tau)\ \{x \in \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil.\ x \neq y\ \tau \wedge P\ (\lambda\text{-}.\ x)\ \tau \neq \mathit{false}\ \tau\})$
apply(*simp*, (*rule disjI2*)+)
by (*metis* (*hide-lams*, *no-types*) *OclValid-def Set-inv-lemma foundation18*$'$)

```
have ins-in-Set_base'' : ⋀τ. (δ X) τ = true τ ⟹
      ?OclSet {x ∈ ⌈⌈Rep-Set_base (X τ)⌉⌉. P (λ-. x) τ ≠ false τ}
   apply(simp, (rule disjI2)+)
   by (metis (hide-lams, no-types) OclValid-def Set-inv-lemma)

have ins-in-Set_base''' : ⋀τ. (δ X) τ = true τ ⟹
      ?OclSet {x ∈ ⌈⌈Rep-Set_base (X τ)⌉⌉. x ≠ y τ ∧ P (λ-. x) τ ≠ false τ}
   apply(simp, (rule disjI2)+)
   by(metis (hide-lams, no-types) OclValid-def Set-inv-lemma)

have if-same : ⋀a b c d τ. τ ⊨ δ a ⟹ b τ = d τ ⟹ c τ = d τ ⟹
                    (if a then b else c endif) τ = d τ
   by(simp add: OclIf-def OclValid-def)

have invert-including : ⋀P y τ. P τ = ⊥ ⟹ P->including(y) τ = ⊥
   by (metis (hide-lams, no-types) foundation16[THEN iffD1,standard]
         foundation18' OclIncluding-valid-args-valid)

have exclude-defined : ⋀τ. τ ⊨ δ X ⟹
       (δ(λ-. Abs-Set_base ⌊⌊{x∈⌈⌈Rep-Set_base (X τ)⌉⌉. x ≠ y τ ∧ P (λ-. x) τ≠false τ}⌋⌋)) τ = true τ
   apply(subst defined-def,
        simp add: false-def true-def bot-Set_base-def bot-fun-def null-Set_base-def null-fun-def)
   by(subst Abs-Set_base-inject[OF ins-in-Set_base'''[simplified false-def]],
      (simp add: OclValid-def bot-option-def null-option-def)+)+

have if-eq : ⋀x A B τ. τ ⊨ υ x ⟹ τ ⊨ ((if x ≐ false then A else B endif) ≜
                                  (if x ≜ false then A else B endif))
   apply(simp add: StrictRefEq_Boolean OclValid-def)
   apply(subst (2) StrongEq-def)
   by(subst cp-OclIf, simp add: cp-OclIf[symmetric] true-def)

have OclSelect-body-bot: ⋀τ. τ ⊨ δ X ⟹ τ ⊨ υ y ⟹ P y τ ≠ ⊥ ⟹
                    (∃x∈⌈⌈Rep-Set_base (X τ)⌉⌉. P (λ-. x) τ = ⊥) ⟹ ⊥ = ?select τ
   apply(drule ex-excluding1[where X = X and y = y and f = λx τ. x τ = ⊥],
        (simp add: P-cp[symmetric])+)
   apply(subgoal-tac τ ⊨ (⊥ ≜ ?select), simp add: OclValid-def StrongEq-def true-def bot-fun-def)
   apply(simp add: OclSelect-body-def)
   apply(subst StrongEq-L-subst3[OF - if-eq], simp, metis foundation18')
   apply(simp add: OclValid-def, subst StrongEq-def, subst true-def, simp)
   apply(subgoal-tac ∃x∈⌈⌈Rep-Set_base (X->excluding(y) τ)⌉⌉. P (λ-. x) τ = ⊥ τ)
    prefer 2 apply (metis bot-fun-def )
    apply(subst if-same[where d = ⊥])
     apply (metis defined7 transform1)
    apply(simp add: OclSelect-def bot-option-def bot-fun-def invalid-def)
   apply(subst invert-including)
   by(simp add: OclSelect-def bot-option-def bot-fun-def invalid-def)+
have d-and-v-inject : ⋀τ X y. (δ X and υ y) τ ≠ true τ ⟹ (δ X and υ y) τ = false τ
   apply(fold OclValid-def, subst foundation22[symmetric])
   apply(auto simp:foundation27  defined-split)
     apply(erule StrongEq-L-subst2-rev,simp,simp)
    apply(erule StrongEq-L-subst2-rev,simp,simp)
   by(erule foundation7'[THEN iffD2, THEN foundation15[THEN iffD2,
                             THEN StrongEq-L-subst2-rev]],simp,simp)




have OclSelect-body-bot': ⋀τ. (δ X and υ y) τ ≠ true τ ⟹ ⊥ = ?select τ
   apply(drule d-and-v-inject)
   apply(simp add: OclSelect-def OclSelect-body-def)
   apply(subst cp-OclIf, subst cp-OclIncluding, simp add: false-def true-def)
   apply(subst cp-OclIf[symmetric], subst cp-OclIncluding[symmetric])
   by (metis (lifting, no-types) OclIf-def foundation18 foundation18' invert-including)

have conj-split2 : ⋀a b c τ. ((a ≜ false) τ = false τ ⟶ b) ∧ ((a ≜ false) τ = true τ ⟶ c) ⟹
                    (a τ ≠ false τ ⟶ b) ∧ (a τ = false τ ⟶ c)
   by (metis OclValid-def defined7 foundation14 foundation22 foundation9)

have defined-inject-true : ⋀τ P. (δ P) τ ≠ true τ ⟹ (δ P) τ = false τ
   apply(simp add: defined-def true-def false-def bot-fun-def bot-option-def
               null-fun-def null-option-def)
   by (case-tac  P τ = ⊥ ∨ P τ = null, simp-all add: true-def)

have cp-OclSelect-body : ⋀τ. ?select τ = OclSelect-body P y (λ-.(OclSelect (X->excluding(y))P)τ)τ
   apply(simp add: OclSelect-body-def)
   by(subst (1 2) cp-OclIf, subst (1 2) cp-OclIncluding, blast)

have OclSelect-body-strict1 : OclSelect-body P y invalid = invalid
   by(rule ext, simp add: OclSelect-body-def OclIf-def)

have bool-invalid: ⋀(x::('𝔄)Boolean) y τ. ¬ (τ ⊨ υ x) ⟹ τ ⊨ ((x ≐ y) ≜ invalid)
   by(simp add: StrictRefEq_Boolean OclValid-def StrongEq-def true-def)

have conj-comm : ⋀p q r. (p ∧ q ∧ r) = ((p ∧ q) ∧ r) by blast

have inv-bot : ⋀τ. invalid τ = ⊥ τ by (metis bot-fun-def invalid-def)
have inv-bot' : ⋀τ. invalid τ = ⊥ by (simp add: invalid-def)

show ?thesis
 apply(rule ext, rename-tac τ)
 apply(subst OclSelect-def)
 apply(case-tac (δ (X->including(y))) τ = true τ, simp)
  apply(( subst ex-including | subst in-including),
   metis OclValid-def foundation5,
   metis OclValid-def foundation5)+
apply(simp add: Let-def inv-bot)
apply(subst (2 4 7 9) bot-fun-def)

apply(subst (4) false-def, subst (4) bot-fun-def, simp add: bot-option-def P-cp[symmetric])

apply(case-tac ¬ (τ ⊨ (υ P y)))
 apply(subgoal-tac P y τ ≠ false τ)
  prefer 2
  apply (metis (hide-lams, no-types) foundation1 foundation18′ valid4)
 apply(simp)

 apply(subst conj-comm, rule conjI)
  apply(drule-tac y = false in bool-invalid)
  apply(simp only: OclSelect-body-def,
        metis OclIf-def OclValid-def defined-def foundation2 foundation22
              bot-fun-def invalid-def)

 apply(drule foundation5[simplified OclValid-def],
       subst al-including[simplified OclValid-def],
       simp,
       simp)
 apply(simp add: P-cp[symmetric])
 apply (metis bot-fun-def foundation18′)

apply(simp add: foundation18′ bot-fun-def OclSelect-body-bot OclSelect-body-bot′)

apply(subst (1 2) al-including, metis OclValid-def foundation5, metis OclValid-def foundation5)
apply(simp add: P-cp[symmetric], subst (4) false-def, subst (4) bot-option-def, simp)

apply(simp add: OclSelect-def[simplified inv-bot′] OclSelect-body-def StrictRefEq_Boolean)
apply(subst (1 2 3 4) cp-OclIf,
   subst (1 2 3 4) foundation18′[THEN iffD2, simplified OclValid-def],
   simp,
   simp only: cp-OclIf[symmetric] refl if-True)
apply(subst (1 2) cp-OclIncluding, rule conj-split2, simp add: cp-OclIf[symmetric])
apply(subst (1 2 3 4 5 6 7 8) cp-OclIf[symmetric], simp)
apply(( subst ex-excluding1[symmetric]
   | subst al-excluding1[symmetric] ),
   metis OclValid-def foundation5,
   metis OclValid-def foundation5,
   simp add: P-cp[symmetric] bot-fun-def)+
apply(simp add: bot-fun-def)
apply(subst (1 2) invert-including, simp+)

apply(rule conjI, blast)
apply(intro impI conjI)
```

```
   apply(subst OclExcluding-def)
   apply(drule foundation5[simplified OclValid-def], simp)
   apply(subst Abs-Set_base-inverse[OF diff-in-Set_base], fast)
   apply(simp add: OclIncluding-def cp-valid[symmetric])
   apply((erule conjE)+, frule exclude-defined[simplified OclValid-def], simp)
   apply(subst Abs-Set_base-inverse[OF ins-in-Set_base'''], simp+)
   apply(subst Abs-Set_base-inject[OF ins-in-Set_base ins-in-Set_base'], fast+)

  apply(simp add: OclExcluding-def)
  apply(simp add: foundation10[simplified OclValid-def])
  apply(subst Abs-Set_base-inverse[OF diff-in-Set_base], simp+)
  apply(subst Abs-Set_base-inject[OF ins-in-Set_base'' ins-in-Set_base'''], simp+)
  apply(subgoal-tac P (λ-. y τ) τ = false τ)
   prefer 2
   apply(subst P-cp[symmetric], metis OclValid-def foundation22)
  apply(rule equalityI)
   apply(rule subsetI, simp, metis)
  apply(rule subsetI, simp)

 apply(drule defined-inject-true)
 apply(subgoal-tac ¬ (τ ⊨ δ X) ∨ ¬ (τ ⊨ υ y))
  prefer 2
  apply (metis bot-fun-def OclValid-def foundation18' OclIncluding-defined-args-valid valid-def)
 apply(subst cp-OclSelect-body, subst cp-OclSelect, subst OclExcluding-def)
 apply(simp add: OclValid-def false-def true-def, rule conjI, blast)
 apply(simp add: OclSelect-invalid[simplified invalid-def]
               OclSelect-body-strict1[simplified invalid-def]
               inv-bot')
done
qed
```

Execution Rules on OclReject **lemma** *OclReject-mtSet-exec*[*simp*,*code-unfold*]: *OclReject mtSet P* = *mtSet*
by(*simp add*: *OclReject-def*)

```
lemma OclReject-including-exec[simp,code-unfold]:
 assumes P-cp : cp P
 shows OclReject (X->including(y)) P = OclSelect-body (not o P) y (OclReject (X->excluding(y)) P)
 apply(simp add: OclReject-def comp-def, rule OclSelect-including-exec)
by (metis  assms cp-intro'(5))
```

Execution Rules Combining Previous Operators OclIncluding

```
lemma OclIncluding-idem0 :
 assumes τ ⊨ δ S
    and τ ⊨ υ i
  shows τ ⊨ (S->including(i)->including(i) ≜ (S->including(i)))
by(simp add: OclIncluding-includes OclIncludes-charn1 assms)
```

```
theorem OclIncluding-idem[simp,code-unfold]: ((S :: ('𝔄,'a::null)Set)−>including(i)−>including(i) = (S−>including(i)))
proof −
 have A: ⋀ τ.   τ ⊨ (i ≜ invalid)   ⟹ (S−>including(i)−>including(i)) τ = invalid τ
         apply(rule foundation22[THEN iffD1])
         by(erule StrongEq-L-subst2-rev, simp,simp)
 have A':⋀ τ.   τ ⊨ (i ≜ invalid)   ⟹ (S−>including(i)) τ = invalid τ
         apply(rule foundation22[THEN iffD1])
         by(erule StrongEq-L-subst2-rev, simp,simp)
 have C: ⋀ τ.   τ ⊨ (S ≜ invalid)   ⟹ (S−>including(i)−>including(i)) τ = invalid τ
         apply(rule foundation22[THEN iffD1])
         by(erule StrongEq-L-subst2-rev, simp,simp)
 have C': ⋀ τ.  τ ⊨ (S ≜ invalid)   ⟹ (S−>including(i)) τ = invalid τ
         apply(rule foundation22[THEN iffD1])
         by(erule StrongEq-L-subst2-rev, simp,simp)
 have D: ⋀ τ.   τ ⊨ (S ≜ null)   ⟹ (S−>including(i)−>including(i)) τ = invalid τ
         apply(rule foundation22[THEN iffD1])
         by(erule StrongEq-L-subst2-rev, simp,simp)
 have D': ⋀ τ.  τ ⊨ (S ≜ null)   ⟹ (S−>including(i)) τ = invalid τ
         apply(rule foundation22[THEN iffD1])
         by(erule StrongEq-L-subst2-rev, simp,simp)
 show ?thesis
  apply(rule ext, rename-tac τ)
  apply(case-tac τ ⊨ (υ i))
   apply(case-tac τ ⊨ (δ S))
    apply(simp only: OclIncluding-idem0[THEN foundation22[THEN iffD1]])
    apply(simp add: foundation16', elim disjE)
    apply(simp add: C[OF foundation22[THEN iffD2]] C'[OF foundation22[THEN iffD2]])
   apply(simp add: D[OF foundation22[THEN iffD2]] D'[OF foundation22[THEN iffD2]])
  apply(simp add:foundation18 A[OF foundation22[THEN iffD2]] A'[OF foundation22[THEN iffD2]])
 done
qed
```

OclExcluding

```
lemma OclExcluding-idem0 :
 assumes τ ⊨ δ S
    and τ ⊨ υ i
  shows τ ⊨ (S−>excluding(i)−>excluding(i) ≜ (S−>excluding(i)))
by(simp add: OclExcluding-excludes OclExcludes-charn1 assms)
```

```
theorem OclExcluding-idem[simp,code-unfold]: ((S−>excluding(i))−>excluding(i)) = (S−>excluding(i))
proof −
 have A: ⋀ τ.   τ ⊨ (i ≜ invalid)   ⟹ (S−>excluding(i)−>excluding(i)) τ = invalid τ
         apply(rule foundation22[THEN iffD1])
         by(erule StrongEq-L-subst2-rev, simp,simp)
 have A':⋀ τ.   τ ⊨ (i ≜ invalid)   ⟹ (S−>excluding(i)) τ = invalid τ
         apply(rule foundation22[THEN iffD1])
```

```
        by(erule StrongEq-L-subst2-rev, simp,simp)
  have C: ⋀ τ.  τ ⊨ (S ≜ invalid)  ⟹ (S−>excluding(i)−>excluding(i)) τ = invalid τ
        apply(rule foundation22[THEN iffD1])
        by(erule StrongEq-L-subst2-rev, simp,simp)
  have C′: ⋀ τ.  τ ⊨ (S ≜ invalid)  ⟹ (S−>excluding(i)) τ = invalid τ
        apply(rule foundation22[THEN iffD1])
        by(erule StrongEq-L-subst2-rev, simp,simp)
  have D: ⋀ τ.  τ ⊨ (S ≜ null)  ⟹ (S−>excluding(i)−>excluding(i)) τ = invalid τ
        apply(rule foundation22[THEN iffD1])
        by(erule StrongEq-L-subst2-rev, simp,simp)
  have D′: ⋀ τ.  τ ⊨ (S ≜ null)  ⟹ (S−>excluding(i)) τ = invalid τ
        apply(rule foundation22[THEN iffD1])
        by(erule StrongEq-L-subst2-rev, simp,simp)
  show ?thesis
   apply(rule ext, rename-tac τ)
   apply(case-tac τ ⊨ (υ i))
    apply(case-tac τ ⊨ (δ S))
     apply(simp only: OclExcluding-idem0[THEN foundation22[THEN iffD1]])
     apply(simp add: foundation16′, elim disjE)
     apply(simp add: C[OF foundation22[THEN iffD2]] C′[OF foundation22[THEN iffD2]])
    apply(simp add: D[OF foundation22[THEN iffD2]] D′[OF foundation22[THEN iffD2]])
   apply(simp add:foundation18 A[OF foundation22[THEN iffD2]] A′[OF foundation22[THEN iffD2]])
  done
qed
```

OclIncludes

```
lemma OclIncludes-any[simp,code-unfold]:
     X−>includes(X−>any()) = (if δ X then
                              if δ (X−>size()) then not(X−>isEmpty())
                              else X−>includes(null) endif
                            else invalid endif)
proof −
 have defined-inject-true : ⋀τ P. (δ P) τ ≠ true τ ⟹ (δ P) τ = false τ
      apply(simp add: defined-def true-def false-def bot-fun-def bot-option-def
                null-fun-def null-option-def)
      by (case-tac  P τ = ⊥ ∨ P τ = null, simp-all add: true-def)

 have valid-inject-true : ⋀τ P. (υ P) τ ≠ true τ ⟹ (υ P) τ = false τ
      apply(simp add: valid-def true-def false-def bot-fun-def bot-option-def
                null-fun-def null-option-def)
      by (case-tac P τ = ⊥, simp-all add: true-def)



 have notempty′: ⋀τ X. τ ⊨ δ X ⟹ finite ⌈⌈Rep-Set_base (X τ)⌉⌉ ⟹ not (X−>isEmpty()) τ ≠ true τ ⟹
                X τ = Set{} τ
  apply(case-tac X τ, rename-tac X′, simp add: mtSet-def Abs-Set_base-inject)
  apply(erule disjE, metis (hide-lams, mono-tags) bot-Set_base-def bot-option-def foundation16)
  apply(erule disjE, metis (hide-lams, no-types) bot-option-def
                                     null-Set_base-def null-option-def foundation16[THEN iffD1,standard])
  apply(case-tac X′, simp, metis (hide-lams, no-types) bot-Set_base-def foundation16[THEN iffD1,standard])
  apply(rename-tac X″, case-tac X″, simp)
   apply (metis (hide-lams, no-types) foundation16[THEN iffD1,standard] null-Set_base-def)
  apply(simp add: OclIsEmpty-def OclSize-def)
  apply(subst (asm) cp-OclNot, subst (asm) cp-OclOr, subst (asm) StrictRefEq_Integer.cp0,
       subst (asm) cp-OclAnd, subst (asm) cp-OclNot)
  apply(simp only: OclValid-def foundation20[simplified OclValid-def]
             cp-OclNot[symmetric] cp-OclAnd[symmetric] cp-OclOr[symmetric])
  apply(simp add: Abs-Set_base-inverse split: split-if-asm)
 by(simp add: true-def OclInt0-def OclNot-def StrictRefEq_Integer StrongEq-def)

 have B: ⋀X τ. ¬ finite ⌈⌈Rep-Set_base (X τ)⌉⌉ ⟹ (δ (X−>size())) τ = false τ
  apply(subst cp-defined)
  apply(simp add: OclSize-def)
 by (metis bot-fun-def defined-def)

 show ?thesis
  apply(rule ext, rename-tac τ, simp only: OclIncludes-def OclANY-def)
  apply(subst cp-OclIf, subst (2) cp-valid)
  apply(case-tac (δ X) τ = true τ,
       simp only: foundation20[simplified OclValid-def] cp-OclIf[symmetric], simp,
       subst (1 2) cp-OclAnd, simp add: cp-OclAnd[symmetric])
   apply(case-tac finite ⌈⌈Rep-Set_base (X τ)⌉⌉)
    apply(frule size-defined′[THEN iffD2, simplified OclValid-def], assumption)
    apply(subst (1 2 3 4) cp-OclIf, simp)
    apply(subst (1 2 3 4) cp-OclIf[symmetric], simp)
    apply(case-tac (X−>notEmpty()) τ = true τ, simp)
     apply(frule OclNotEmpty-has-elt[simplified OclValid-def], simp)
     apply(simp add: OclNotEmpty-def cp-OclIf[symmetric])
     apply(subgoal-tac (SOME y. y ∈ ⌈⌈Rep-Set_base (X τ)⌉⌉) ∈ ⌈⌈Rep-Set_base (X τ)⌉⌉, simp add: true-def)
      apply(metis OclValid-def Set-inv-lemma foundation18′ null-option-def true-def)
     apply(rule someI-ex, simp)
    apply(simp add: OclNotEmpty-def cp-valid[symmetric])
    apply(subgoal-tac ¬ (null τ ∈ ⌈⌈Rep-Set_base (X τ)⌉⌉), simp)
     apply(subst OclIsEmpty-def, simp add: OclSize-def)
     apply(subst cp-OclNot, subst cp-OclOr, subst StrictRefEq_Integer.cp0, subst cp-OclAnd,
          subst cp-OclNot, simp add: OclValid-def foundation20[simplified OclValid-def]
                           cp-OclNot[symmetric] cp-OclAnd[symmetric] cp-OclOr[symmetric])
     apply(frule notempty′[simplified OclValid-def],
          (simp add: mtSet-def Abs-Set_base-inverse OclInt0-def false-def)+)
    apply(drule notempty′[simplified OclValid-def], simp, simp)
    apply (metis (hide-lams, no-types) empty-iff mtSet-rep-set)

   apply(frule B)
   apply(subst (1 2 3 4) cp-OclIf, simp)
   apply(subst (1 2 3 4) cp-OclIf[symmetric], simp)
   apply(case-tac (X−>notEmpty()) τ = true τ, simp)
    apply(frule OclNotEmpty-has-elt[simplified OclValid-def], simp)
    apply(simp add: OclNotEmpty-def OclIsEmpty-def)
    apply(subgoal-tac X−>size() τ = ⊥)
     prefer 2
     apply (metis (hide-lams, no-types) OclSize-def)
    apply(subst (asm) cp-OclNot, subst (asm) cp-OclOr, subst (asm) StrictRefEq_Integer.cp0,
         subst (asm) cp-OclAnd, subst (asm) cp-OclNot)
    apply(simp add: OclValid-def foundation20[simplified OclValid-def]
                 cp-OclNot[symmetric] cp-OclAnd[symmetric] cp-OclOr[symmetric])
    apply(simp add: OclNot-def StrongEq-def StrictRefEq_Integer valid-def false-def true-def
                 bot-option-def bot-fun-def invalid-def)

   apply (metis bot-fun-def null-fun-def null-is-valid valid-def)
 by(drule defined-inject-true,
    simp add: false-def true-def OclIf-false[simplified false-def] invalid-def)
qed
```

OclSize

```
lemma [simp,code-unfold]: δ (Set{} −>size()) = true
by simp


lemma [simp,code-unfold]: δ ((X −>including(x)) −>size()) = (δ(X−>size()) and υ(x))
proof −
 have defined-inject-true : ⋀τ P. (δ P) τ ≠ true τ ⟹ (δ P) τ = false τ
     apply(simp add: defined-def true-def false-def bot-fun-def bot-option-def
                null-fun-def null-option-def)
     by (case-tac  P τ = ⊥ ∨ P τ = null, simp-all add: true-def)

 have valid-inject-true : ⋀τ P. (υ P) τ ≠ true τ ⟹ (υ P) τ = false τ
     apply(simp add: valid-def true-def false-def bot-fun-def bot-option-def
                null-fun-def null-option-def)
     by (case-tac P τ = ⊥, simp-all add: true-def)

 have OclIncluding-finite-rep-set : ⋀τ. (δ X and υ x) τ = true τ ⟹
            finite ⌈⌈Rep-Set_base (X−>including(x) τ)⌉⌉ = finite ⌈⌈Rep-Set_base (X τ)⌉⌉
  apply(rule OclIncluding-finite-rep-set)
 by(metis OclValid-def foundation5)+

 have card-including-exec : ⋀τ. (δ (λ-. ⌊⌊int (card ⌈⌈Rep-Set_base (X−>including(x) τ)⌉⌉)⌋⌋)) τ =
                        (δ (λ-. ⌊⌊int (card ⌈⌈Rep-Set_base (X τ)⌉⌉)⌋⌋)) τ
 by(simp add: defined-def bot-fun-def bot-option-def null-fun-def null-option-def)

 show ?thesis
  apply(rule ext, rename-tac τ)
  apply(case-tac (δ (X−>including(x)−>size())) τ = true τ, simp del: OclSize-including-exec)
   apply(subst cp-OclAnd, subst cp-defined, simp only: cp-defined[of X−>including(x)−>size()],
      simp add: OclSize-def)
   apply(case-tac ((δ X and υ x) τ = true τ ∧ finite ⌈⌈Rep-Set_base (X−>including(x) τ)⌉⌉), simp)
    apply(erule conjE,
        simp add: OclIncluding-finite-rep-set[simplified OclValid-def] card-including-exec
              cp-OclAnd[of δ X υ x]
              cp-OclAnd[of true, THEN sym])
    apply(subgoal-tac (δ X) τ = true τ ∧ (υ x) τ = true τ, simp)
    apply(rule foundation5[of - δ X υ x, simplified OclValid-def],
        simp only: cp-OclAnd[THEN sym])
   apply(simp, simp add: defined-def true-def false-def bot-fun-def bot-option-def)

  apply(drule defined-inject-true[of X−>including(x)−>size()],
       simp del: OclSize-including-exec,
       simp only: cp-OclAnd[of δ (X−>size()) υ x],
       simp add: cp-defined[of X−>including(x)−>size() ] cp-defined[of X−>size() ]
          del: OclSize-including-exec,
       simp add: OclSize-def card-including-exec
          del: OclSize-including-exec)
  apply(case-tac (δ X and υ x) τ = true τ ∧ finite ⌈⌈Rep-Set_base (X τ)⌉⌉,
       simp add: OclIncluding-finite-rep-set[simplified OclValid-def] card-including-exec,
       simp only: cp-OclAnd[THEN sym],
       simp add: defined-def bot-fun-def)

  apply(split split-if-asm)
   apply(simp add: OclIncluding-finite-rep-set[simplified OclValid-def] card-including-exec)+
  apply(simp only: cp-OclAnd[THEN sym], simp, rule impI, erule conjE)
  apply(case-tac (υ x) τ = true τ, simp add: cp-OclAnd[of δ X υ x])
 by(drule valid-inject-true[of x], simp add: cp-OclAnd[of - υ x])
qed

lemma [simp,code-unfold]: δ ((X −>excluding(x)) −>size()) = (δ(X−>size()) and υ(x))
proof −
 have defined-inject-true : ⋀τ P. (δ P) τ ≠ true τ ⟹ (δ P) τ = false τ
     apply(simp add: defined-def true-def false-def bot-fun-def bot-option-def
                 null-fun-def null-option-def)
     by (case-tac  P τ = ⊥ ∨ P τ = null, simp-all add: true-def)

 have valid-inject-true : ⋀τ P. (υ P) τ ≠ true τ ⟹ (υ P) τ = false τ
     apply(simp add: valid-def true-def false-def bot-fun-def bot-option-def
                 null-fun-def null-option-def)
     by (case-tac P τ = ⊥, simp-all add: true-def)

 have OclExcluding-finite-rep-set : ⋀τ. (δ X and υ x) τ = true τ ⟹
                           finite ⌈⌈Rep-Set_base (X−>excluding(x) τ)⌉⌉ =
                           finite ⌈⌈Rep-Set_base (X τ)⌉⌉
  apply(rule OclExcluding-finite-rep-set)
 by(metis OclValid-def foundation5)+
 have card-excluding-exec : ⋀τ. (δ (λ-. ⌊⌊int (card ⌈⌈Rep-Set_base (X−>excluding(x) τ)⌉⌉)⌋⌋)) τ =
                            (δ (λ-. ⌊⌊int (card ⌈⌈Rep-Set_base (X τ)⌉⌉)⌋⌋)) τ
 by(simp add: defined-def bot-fun-def bot-option-def null-fun-def null-option-def)

 show ?thesis
  apply(rule ext, rename-tac τ)
  apply(case-tac (δ (X−>excluding(x)−>size())) τ = true τ, simp)
   apply(subst cp-OclAnd, subst cp-defined, simp only: cp-defined[of X−>excluding(x)−>size()],
        simp add: OclSize-def)
   apply(case-tac ((δ X and υ x) τ = true τ ∧ finite ⌈⌈Rep-Set_base (X−>excluding(x) τ)⌉⌉), simp)
    apply(erule conjE,
          simp add: OclExcluding-finite-rep-set[simplified OclValid-def] card-excluding-exec
                   cp-OclAnd[of δ X υ x]
                   cp-OclAnd[of true, THEN sym])
    apply(subgoal-tac (δ X) τ = true τ ∧ (υ x) τ = true τ, simp)
    apply(rule foundation5[of - δ X υ x, simplified OclValid-def],
          simp only: cp-OclAnd[THEN sym])
   apply(simp, simp add: defined-def true-def false-def bot-fun-def bot-option-def)

  apply(drule defined-inject-true[of X−>excluding(x)−>size()],
        simp,
        simp only: cp-OclAnd[of δ (X−>size()) υ x],
        simp add: cp-defined[of X−>excluding(x)−>size() ] cp-defined[of X−>size() ],
        simp add: OclSize-def card-excluding-exec)
  apply(case-tac (δ X and υ x) τ = true τ ∧ finite ⌈⌈Rep-Set_base (X τ)⌉⌉,
        simp add: OclExcluding-finite-rep-set[simplified OclValid-def] card-excluding-exec,
        simp only: cp-OclAnd[THEN sym],
        simp add: defined-def bot-fun-def)

  apply(split split-if-asm)
   apply(simp add: OclExcluding-finite-rep-set[simplified OclValid-def] card-excluding-exec)+
  apply(simp only: cp-OclAnd[THEN sym], simp, rule impI, erule conjE)
  apply(case-tac (υ x) τ = true τ, simp add: cp-OclAnd[of δ X υ x])
 by(drule valid-inject-true[of x], simp add: cp-OclAnd[of - υ x])
qed

lemma [simp]:
 assumes X-finite: ⋀τ. finite ⌈⌈Rep-Set_base (X τ)⌉⌉
 shows δ ((X −>including(x)) −>size()) = (δ(X) and υ(x))
by(simp add: size-defined[OF X-finite] del: OclSize-including-exec)
```

OclForall

```
lemma OclForall-rep-set-false:
 assumes τ ⊨ δ X
 shows (OclForall X P τ = false τ) = (∃x ∈ ⌈⌈Rep-Set_base (X τ)⌉⌉. P (λτ. x) τ = false τ)
by(insert assms, simp add: OclForall-def OclValid-def false-def true-def invalid-def
                 bot-fun-def bot-option-def null-fun-def null-option-def)
```

lemma *OclForall-rep-set-true*:
 assumes $\tau \models \delta\ X$
 shows $(\tau \models \mathit{OclForall}\ X\ P) = (\forall x \in \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil.\ \tau \models P\ (\lambda\tau.\ x))$
proof $-$
 have *destruct-ocl* : $\bigwedge x\ \tau.\ x = \mathit{true}\ \tau \vee x = \mathit{false}\ \tau \vee x = \mathit{null}\ \tau \vee x = \bot\ \tau$
  apply(*case-tac x*) **apply** (*metis bot-Boolean-def*)
  apply(*rename-tac* x', *case-tac* x') **apply** (*metis null-Boolean-def*)
  apply(*rename-tac* x'', *case-tac* x'') **apply** (*metis* (*full-types*) *true-def*)
 by (*metis* (*full-types*) *false-def*)

 have *disjE4* : $\bigwedge$ *P1 P2 P3 P4 R*.
  $(P1 \vee P2 \vee P3 \vee P4) \Longrightarrow (P1 \Longrightarrow R) \Longrightarrow (P2 \Longrightarrow R) \Longrightarrow (P3 \Longrightarrow R) \Longrightarrow (P4 \Longrightarrow R) \Longrightarrow R$
 by *metis*
 show *?thesis*
  apply(*simp add*: *OclForall-def OclValid-def true-def false-def invalid-def*
   bot-fun-def bot-option-def null-fun-def null-option-def split: *split-if-asm*)
  apply(*rule conjI*, *rule impI*) **apply** (*metis drop.simps option.distinct*(*1*) *invalid-def*)
  apply(*rule impI*, *rule conjI*, *rule impI*) **apply** (*metis option.distinct*(*1*))
  apply(*rule impI*, *rule conjI*, *rule impI*) **apply** (*metis drop.simps*)
  apply(*intro conjI impI ballI*)
   proof $-$ **fix** *x* **show** $\forall x{\in}\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil.\ P\ (\lambda\text{-}.\ x)\ \tau \neq \lfloor \mathit{None} \rfloor \Longrightarrow$
    $\forall x{\in}\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil.\ \exists y.\ P\ (\lambda\text{-}.\ x)\ \tau = \lfloor y \rfloor \Longrightarrow$
    $\forall x{\in}\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil.\ P\ (\lambda\text{-}.\ x)\ \tau \neq \lfloor\lfloor \mathit{False} \rfloor\rfloor \Longrightarrow$
    $x \in \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil \Longrightarrow P\ (\lambda\tau.\ x)\ \tau = \lfloor\lfloor \mathit{True} \rfloor\rfloor$
   apply(*erule-tac* $x = x$ **in** *ballE*)+
   by(*rule disjE4*[*OF destruct-ocl*[*of P* $(\lambda\tau.\ x)\ \tau$]],
    (*simp add*: *true-def false-def null-fun-def null-option-def bot-fun-def bot-option-def*)+)
  apply-end(*simp add*: *assms*[*simplified OclValid-def true-def*])+
 qed
qed

lemma *OclForall-includes* :
 assumes *x-def* : $\tau \models \delta\ x$
  and *y-def* : $\tau \models \delta\ y$
  shows $(\tau \models \mathit{OclForall}\ x\ (\mathit{OclIncludes}\ y)) = (\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (x\ \tau)\rceil\rceil \subseteq \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (y\ \tau)\rceil\rceil)$
 apply(*simp add*: *OclForall-rep-set-true*[*OF x-def*],
   simp add: *OclIncludes-def OclValid-def y-def*[*simplified OclValid-def*])
 apply(*insert Set-inv-lemma*[*OF x-def*], *simp add*: *valid-def false-def true-def bot-fun-def*)
by(*rule iffI*, *simp add*: *subsetI*, *simp add*: *subsetD*)

lemma *OclForall-not-includes* :
 assumes *x-def* : $\tau \models \delta\ x$
  and *y-def* : $\tau \models \delta\ y$
  shows $(\mathit{OclForall}\ x\ (\mathit{OclIncludes}\ y)\ \tau = \mathit{false}\ \tau) = (\neg\ \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (x\ \tau)\rceil\rceil \subseteq \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (y\ \tau)\rceil\rceil)$
 apply(*simp add*: *OclForall-rep-set-false*[*OF x-def*],
   simp add: *OclIncludes-def OclValid-def y-def*[*simplified OclValid-def*])
 apply(*insert Set-inv-lemma*[*OF x-def*], *simp add*: *valid-def false-def true-def bot-fun-def*)
by(*rule iffI*, *metis set-rev-mp*, *metis subsetI*)

lemma *OclForall-iterate*:
assumes *S-finite*: *finite* $\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (S\ \tau)\rceil\rceil$
shows $S{-}{>}\mathit{forAll}(x \mid P\ x)\ \tau = (S{-}{>}\mathit{iterate}(x;\ acc = true \mid acc\ and\ P\ x))\ \tau$
proof $-$
have *and-comm* : *comp-fun-commute* $(\lambda x\ acc.\ acc\ and\ P\ x)$
apply(*simp add*: *comp-fun-commute-def comp-def*)
by (*metis OclAnd-assoc OclAnd-commute*)

have *ex-insert* : $\bigwedge x\ F\ P.\ (\exists x{\in}insert\ x\ F.\ P\ x) = (P\ x \vee (\exists x{\in}F.\ P\ x))$
by (*metis insert-iff*)

have *destruct-ocl* : $\bigwedge x\ \tau.\ x = true\ \tau \vee x = false\ \tau \vee x = null\ \tau \vee x = \bot\ \tau$
apply(*case-tac x*) **apply** (*metis bot-Boolean-def*)
apply(*rename-tac* x', *case-tac* x') **apply** (*metis null-Boolean-def*)
apply(*rename-tac* x'', *case-tac* x'') **apply** (*metis* (*full-types*) *true-def*)
by (*metis* (*full-types*) *false-def*)

have *disjE4* : $\bigwedge$ *P1 P2 P3 P4 R*.
$(P1 \vee P2 \vee P3 \vee P4) \Longrightarrow (P1 \Longrightarrow R) \Longrightarrow (P2 \Longrightarrow R) \Longrightarrow (P3 \Longrightarrow R) \Longrightarrow (P4 \Longrightarrow R) \Longrightarrow R$
by *metis*

let $?P\text{-}eq = \lambda x\ b\ \tau.\ P\ (\lambda\text{-}.\ x)\ \tau = b\ \tau$
let $?P = \lambda set\ b\ \tau.\ \exists x{\in}set.\ ?P\text{-}eq\ x\ b\ \tau$
let $?if = \lambda f\ b\ c.\ if\ f\ b\ \tau\ then\ b\ \tau\ else\ c$
let $?forall = \lambda P.\ ?if\ P\ false\ (?if\ P\ invalid\ (?if\ P\ null\ (true\ \tau)))$
show *?thesis*
apply(*simp only*: *OclForall-def OclIterate-def*)
apply(*case-tac* $\tau \models \delta\ S$, *simp only*: *OclValid-def*)
apply(*subgoal-tac let set* $= \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (S\ \tau)\rceil\rceil$ *in*
$?forall\ (?P\ set) =$
$Finite\text{-}Set.fold\ (\lambda x\ acc.\ acc\ and\ P\ x)\ true\ ((\lambda a\ \tau.\ a)\ `\ set)\ \tau$,
simp only: *Let-def*, *simp add*: *S-finite*, *simp only*: *Let-def*)
apply(*case-tac* $\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (S\ \tau)\rceil\rceil = \{\}$, *simp*)
apply(*rule finite-ne-induct*[*OF S-finite*], *simp*)

apply(*simp only*: *image-insert*)
apply(*subst comp-fun-commute.fold-insert*[*OF and-comm*], *simp*)
apply (*metis empty-iff image-empty*)
apply(*simp add*: *invalid-def*)
apply (*metis bot-fun-def destruct-ocl null-fun-def*)

apply(*simp only*: *image-insert*)
apply(*subst comp-fun-commute.fold-insert*[*OF and-comm*], *simp*)
apply (*metis* (*mono-tags*) *imageE*)

apply(*subst cp-OclAnd*) **apply**(*drule sym*, *drule sym*, *simp only*:, *drule sym*, *simp only*:)

```
  apply(simp only: ex-insert)
  apply(subgoal-tac ∃x. x∈F) prefer 2
   apply(metis all-not-in-conv)
  proof − fix x F show (δ S) τ = true τ ⟹ ∃x. x ∈ F ⟹
          ?forall (λb τ. ?P-eq x b τ ∨ ?P F b τ) =
          ((λ-. ?forall (?P F)) and (λ-. P (λτ. x) τ)) τ
   apply(rule disjE4[OF destruct-ocl[where x = P (λτ. x) τ]])
     apply(simp-all add: true-def false-def invalid-def OclAnd-def
                    null-fun-def null-option-def bot-fun-def bot-option-def)
  by (metis (lifting) option.distinct(1))+
  apply-end(simp add: OclValid-def)+
 qed
qed

lemma OclForall-cong:
 assumes ⋀x. x ∈ ⌈⌈Rep-Set_base (X τ)⌉⌉ ⟹ τ ⊨ P (λτ. x) ⟹ τ ⊨ Q (λτ. x)
 assumes P: τ ⊨ OclForall X P
 shows τ ⊨ OclForall X Q
proof −
 have def-X: τ ⊨ δ X
 by(insert P, simp add: OclForall-def OclValid-def bot-option-def true-def split: split-if-asm)
 show ?thesis
  apply(insert P)
  apply(subst (asm) OclForall-rep-set-true[OF def-X], subst OclForall-rep-set-true[OF def-X])
 by (simp add: assms)
qed

lemma OclForall-cong':
 assumes ⋀x. x ∈ ⌈⌈Rep-Set_base (X τ)⌉⌉ ⟹ τ ⊨ P (λτ. x) ⟹ τ ⊨ Q (λτ. x) ⟹ τ ⊨ R (λτ. x)
 assumes P: τ ⊨ OclForall X P
 assumes Q: τ ⊨ OclForall X Q
 shows τ ⊨ OclForall X R
proof −
 have def-X: τ ⊨ δ X
 by(insert P, simp add: OclForall-def OclValid-def bot-option-def true-def split: split-if-asm)
 show ?thesis
  apply(insert P Q)
  apply(subst (asm) (1 2) OclForall-rep-set-true[OF def-X], subst OclForall-rep-set-true[OF def-X])
 by (simp add: assms)
qed
```

Strict Equality

```
lemma StrictRefEq_Set-defined :
 assumes x-def: τ ⊨ δ x
 assumes y-def: τ ⊨ δ y
 shows ((x::('𝔄,'α::null)Set) ≐ y) τ =
            (x−>forAll(z| y−>includes(z)) and (y−>forAll(z| x−>includes(z)))) τ
proof −
```

```
have rep-set-inj : ⋀τ. (δ x) τ = true τ ⟹
                  (δ y) τ = true τ ⟹
                   x τ ≠ y τ ⟹
                   ⌈⌈Rep-Set_base (y τ)⌉⌉ ≠ ⌈⌈Rep-Set_base (x τ)⌉⌉
 apply(simp add: defined-def)
 apply(split split-if-asm, simp add: false-def true-def)+
 apply(simp add: null-fun-def null-Set_base-def bot-fun-def bot-Set_base-def)

 apply(case-tac x τ, rename-tac x')
 apply(case-tac x', simp-all, rename-tac x'')
 apply(case-tac x'', simp-all)

 apply(case-tac y τ, rename-tac y')
 apply(case-tac y', simp-all, rename-tac y'')
 apply(case-tac y'', simp-all)

 apply(simp add: Abs-Set_base-inverse)
by(blast)

show ?thesis
 apply(simp add: StrictRefEq_Set StrongEq-def
  foundation20[OF x-def, simplified OclValid-def]
  foundation20[OF y-def, simplified OclValid-def])
 apply(subgoal-tac ⌊⌊x τ = y τ⌋⌋ = true τ ∨ ⌊⌊x τ = y τ⌋⌋ = false τ)
  prefer 2
  apply(simp add: false-def true-def)

 apply(erule disjE)
  apply(simp add: true-def)

  apply(subgoal-tac (τ ⊨ OclForall x (OclIncludes y)) ∧ (τ ⊨ OclForall y (OclIncludes x)))
   apply(subst cp-OclAnd, simp add: true-def OclValid-def)
  apply(simp add: OclForall-includes[OF x-def y-def]
             OclForall-includes[OF y-def x-def])


 apply(simp)

 apply(subgoal-tac OclForall x (OclIncludes y) τ = false τ ∨
               OclForall y (OclIncludes x) τ = false τ)
  apply(subst cp-OclAnd, metis OclAnd-false1 OclAnd-false2 cp-OclAnd)
 apply(simp only: OclForall-not-includes[OF x-def y-def, simplified OclValid-def]
             OclForall-not-includes[OF y-def x-def, simplified OclValid-def],
      simp add: false-def)
by (metis OclValid-def rep-set-inj subset-antisym x-def y-def)
qed

lemma StrictRefEq_Set-exec[simp,code-unfold] :
```

```
((x::('𝔄,'α::null)Set) ≐ y) =
 (if δ x then (if δ y
          then ((x−>forAll(z| y−>includes(z)) and (y−>forAll(z| x−>includes(z)))))
          else if υ y
             then false (∗ x′−>includes = null ∗)
             else invalid
             endif
          endif)
      else if υ x (∗ null = ??? ∗)
         then if υ y then not(δ y) else invalid endif
         else invalid
         endif
      endif)
proof −
 have defined-inject-true : ⋀τ P. (¬ (τ ⊨ δ P)) = ((δ P) τ = false τ)
 by (metis bot-fun-def OclValid-def defined-def foundation16 null-fun-def)

 have valid-inject-true : ⋀τ P. (¬ (τ ⊨ υ P)) = ((υ P) τ = false τ)
 by (metis bot-fun-def OclIf-true′ OclIncludes-charn0 OclIncludes-charn0′ OclValid-def valid-def
      foundation6 foundation9)
 show ?thesis
  apply(rule ext, rename-tac τ)

  apply(simp add: OclIf-def
           defined-inject-true[simplified OclValid-def]
           valid-inject-true[simplified OclValid-def],
     subst false-def, subst true-def, simp)
  apply(subst (1 2) cp-OclNot, simp, simp add: cp-OclNot[symmetric])
  apply(simp add: StrictRefEq_Set-defined[simplified OclValid-def])
 by(simp add: StrictRefEq_Set StrongEq-def false-def true-def valid-def defined-def)
qed

lemma StrictRefEq_Set-L-subst1 : cp P ⟹ τ ⊨ υ x ⟹ τ ⊨ υ y ⟹ τ ⊨ υ P x ⟹ τ ⊨ υ P y ⟹
   τ ⊨ (x::('𝔄,'α::null)Set) ≐ y ⟹ τ ⊨ (P x ::('𝔄,'α::null)Set) ≐ P y
 apply(simp only: StrictRefEq_Set OclValid-def)
 apply(split split-if-asm)
  apply(simp add: StrongEq-L-subst1[simplified OclValid-def])
by (simp add: invalid-def bot-option-def true-def)

lemma OclIncluding-cong′ :
shows τ ⊨ δ s ⟹ τ ⊨ δ t ⟹ τ ⊨ υ x ⟹
   τ ⊨ ((s::('𝔄,'a::null)Set) ≐ t) ⟹ τ ⊨ (s−>including(x) ≐ (t−>including(x)))
proof −
 have cp: cp (λs. (s−>including(x)))
  apply(simp add: cp-def, subst cp-OclIncluding)
 by (rule-tac x = (λxab ab. ((λ-. xab)−>including(λ-. x ab)) ab) in exI, simp)

 show τ ⊨ δ s ⟹ τ ⊨ δ t ⟹ τ ⊨ υ x ⟹ τ ⊨ (s ≐ t) ⟹ ?thesis
```

apply$(\mathit{rule\text{-}tac}\ P = \lambda s.\ (s{-}{>}\mathit{including}(x))$ **in** $\mathit{StrictRefEq}_{\mathit{Set}}\mathit{\text{-}L\text{-}subst1})$
apply$(\mathit{rule}\ \mathit{cp})$
apply$(\mathit{simp}\ \mathit{add}\colon \mathit{foundation20})$ **apply**$(\mathit{simp}\ \mathit{add}\colon \mathit{foundation20})$
apply $(\mathit{simp}\ \mathit{add}\colon \mathit{foundation10}\ \mathit{foundation6}){+}$
done
qed

lemma $\mathit{OclIncluding\text{-}cong} : \bigwedge(s{::}({}'\mathfrak{A},{}'a{::}\mathit{null})\mathit{Set})\ t\ x\ y\ \tau.\ \tau \models \delta\ t \Longrightarrow \tau \models \upsilon\ y \Longrightarrow$
$\tau \models s \doteq t \Longrightarrow x = y \Longrightarrow \tau \models s{-}{>}\mathit{including}(x) \doteq (t{-}{>}\mathit{including}(y))$
apply$(\mathit{simp}\ \mathit{only}\colon)$
apply$(\mathit{rule}\ \mathit{OclIncluding\text{-}cong}',\ \mathit{simp\text{-}all}\ \mathit{only}\colon)$
by$(\mathit{auto}\ \mathit{simp}\colon \mathit{OclValid\text{-}def}\ \mathit{OclIf\text{-}def}\ \mathit{invalid\text{-}def}\ \mathit{bot\text{-}option\text{-}def}\ \mathit{OclNot\text{-}def}\ \mathit{split} : \mathit{split\text{-}if\text{-}asm})$

lemma $\mathit{const\text{-}StrictRefEq}_{\mathit{Set}}\mathit{\text{-}empty} : \mathit{const}\ X \Longrightarrow\ \mathit{const}\ (X \doteq \mathit{Set}\{\})$
apply$(\mathit{rule}\ \mathit{StrictRefEq}_{\mathit{Set}}.\mathit{const},\ \mathit{assumption})$
by(simp)

lemma $\mathit{const\text{-}StrictRefEq}_{\mathit{Set}}\mathit{\text{-}including}$:
$\mathit{const}\ a \Longrightarrow \mathit{const}\ S \Longrightarrow \mathit{const}\ X \Longrightarrow\ \mathit{const}\ (X \doteq S{-}{>}\mathit{including}(a))$
apply$(\mathit{rule}\ \mathit{StrictRefEq}_{\mathit{Set}}.\mathit{const},\ \mathit{assumption})$
by$(\mathit{rule}\ \mathit{const\text{-}OclIncluding})$

Test Statements

Assert $(\tau \models (\mathit{Set}\{\lambda\text{-}.\ \lfloor\lfloor x\rfloor\rfloor\} \doteq \mathit{Set}\{\lambda\text{-}.\ \lfloor\lfloor x\rfloor\rfloor\}))$
Assert $(\tau \models (\mathit{Set}\{\lambda\text{-}.\ \lfloor x\rfloor\} \doteq \mathit{Set}\{\lambda\text{-}.\ \lfloor x\rfloor\}))$

end

theory *UML-Sequence*
imports *../basic-types/UML-Boolean*
../basic-types/UML-Integer
begin

B.2.9. Collection Type Sequence: Operations

Constants: mtSequence

definition $\mathit{mtSequence}\ {::}({}'\mathfrak{A},{}'\alpha{::}\mathit{null})\ \mathit{Sequence}\ \ (\mathit{Sequence}\{\})$
where $\mathit{Sequence}\{\} \equiv (\lambda\ \tau.\ \mathit{Abs\text{-}Sequence}_{\mathit{base}}\ \lfloor\lfloor[]{::}{}'\alpha\ \mathit{list}\rfloor\rfloor\ )$

declare $\mathit{mtSequence\text{-}def}[\mathit{code\text{-}unfold}]$

lemma $\mathit{mtSequence\text{-}defined}[\mathit{simp},\mathit{code\text{-}unfold}]{:}\delta(\mathit{Sequence}\{\}) = \mathit{true}$
apply$(\mathit{rule}\ \mathit{ext},\ \mathit{auto}\ \mathit{simp}\colon \mathit{mtSequence\text{-}def}\ \mathit{defined\text{-}def}\ \mathit{null\text{-}Sequence}_{\mathit{base}}\mathit{\text{-}def}$

bot-Sequence$_{base}$*-def bot-fun-def null-fun-def*)
by(*simp-all add*: *Abs-Sequence*$_{base}$*-inject bot-option-def null-option-def*)

lemma *mtSequence-valid*[*simp*,*code-unfold*]:$\upsilon(Sequence\{\}) = true$
apply(*rule ext*,*auto simp*: *mtSequence-def valid-def null-Sequence*$_{base}$*-def*
bot-Sequence$_{base}$*-def bot-fun-def null-fun-def*)
by(*simp-all add*: *Abs-Sequence*$_{base}$*-inject bot-option-def null-option-def*)

lemma *mtSequence-rep-set*: $\lceil\lceil Rep\text{-}Sequence_{base}\ (Sequence\{\}\ \tau)\rceil\rceil = []$
apply(*simp add*: *mtSequence-def*, *subst Abs-Sequence*$_{base}$*-inverse*)
by(*simp add*: *bot-option-def*)+

lemma [*simp*,*code-unfold*]: *const Sequence*{}
by(*simp add*: *const-def mtSequence-def*)

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

lemmas $cp\text{-}intro''_{Sequence}$[*intro*!,*simp*,*code-unfold*] = *cp-intro′*

Properties of Sequence Type: Every element in a defined sequence is valid.

lemma *Sequence-inv-lemma*: $\tau \models (\delta\ X) \Longrightarrow \forall x{\in}set\ \lceil\lceil Rep\text{-}Sequence_{base}\ (X\ \tau)\rceil\rceil.\ x \neq bot$
apply(*insert Rep-Sequence*$_{base}$ [*of X* τ], *simp*)
apply(*auto simp*: *OclValid-def defined-def false-def true-def cp-def*
bot-fun-def bot-Sequence$_{base}$*-def null-Sequence*$_{base}$*-def null-fun-def*
split:*split-if-asm*)
apply(*erule contrapos-pp* [*of Rep-Sequence*$_{base}$ $(X\ \tau) = bot$])
apply(*subst Abs-Sequence*$_{base}$*-inject*[*symmetric*], *rule Rep-Sequence*$_{base}$, *simp*)
apply(*simp add*: *Rep-Sequence*$_{base}$*-inverse bot-Sequence*$_{base}$*-def bot-option-def*)
apply(*erule contrapos-pp* [*of Rep-Sequence*$_{base}$ $(X\ \tau) = null$])
apply(*subst Abs-Sequence*$_{base}$*-inject*[*symmetric*], *rule Rep-Sequence*$_{base}$, *simp*)
apply(*simp add*: *Rep-Sequence*$_{base}$*-inverse null-option-def*)
by (*simp add*: *bot-option-def*)

Strict Equality

Definition After the part of foundational operations on sets, we detail here equality on sets. Strong equality is inherited from the OCL core, but we have to consider the case of the strict equality. We decide to overload strict equality in the same way we do for other value's in OCL:

defs $StrictRefEq_{Sequence}$ [*code-unfold*]:
$((x{::}('\mathfrak{A},'\alpha{::}null)Sequence) \doteq y) \equiv (\lambda\ \tau.\ if\ (\upsilon\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau$
$then\ (x \triangleq y)\tau$
$else\ invalid\ \tau)$

Property proof in terms of *profile-bin3*

interpretation $StrictRefEq_{Sequence}$: *profile-bin3* $\lambda\ x\ y.\ (x{::}('\mathfrak{A},'\alpha{::}null)Sequence) \doteq y$
by *unfold-locales* (*auto simp*: $StrictRefEq_{Sequence}$)

Standard Operations

Definition: including **definition** *OclIncluding* :: $[('\mathfrak{A},'\alpha::null)\ Sequence,('\mathfrak{A},'\alpha)\ val] \Rightarrow ('\mathfrak{A},'\alpha)\ Sequence$

where $OclIncluding\ x\ y = (\lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau$

$then\ Abs\text{-}Sequence_{base}\ \lfloor\lfloor\ \lceil\lceil Rep\text{-}Sequence_{base}\ (x\ \tau)\rceil\rceil\ @\ [y\ \tau]\ \rfloor\rfloor$

$else\ invalid\ \tau\ )$

notation *OclIncluding* $(\text{-}{-}{>}including_{Seq}'(\text{-}'))$

interpretation *OclIncluding* :

$profile\text{-}bin2\ OclIncluding\ \lambda x\ y.\ Abs\text{-}Sequence_{base}\lfloor\lfloor\lceil\lceil Rep\text{-}Sequence_{base}\ x\rceil\rceil\ @\ [y]\rfloor\rfloor$

proof –

have $A : \bigwedge x\ y.\ x \neq bot \Longrightarrow x \neq null \Longrightarrow\ y \neq bot\ \Longrightarrow$

$\lfloor\lfloor\lceil\lceil Rep\text{-}Sequence_{base}\ x\rceil\rceil\ @\ [y]\rfloor\rfloor \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}set\ \lceil\lceil X\rceil\rceil.\ x \neq bot)\}$

by(*auto intro*!:*Sequence-inv-lemma*[*simplified OclValid-def defined-def false-def true-def null-fun-def bot-fun-def*])

show $profile\text{-}bin2\ OclIncluding\ (\lambda x\ y.\ Abs\text{-}Sequence_{base}\ \lfloor\lfloor\lceil\lceil Rep\text{-}Sequence_{base}\ x\rceil\rceil\ @\ [y]\rfloor\rfloor)$

apply *unfold-locales*

apply(*auto simp*:*OclIncluding-def bot-option-def null-option-def null-Sequence*$_{base}$*-def bot-Sequence*$_{base}$*-def*)

apply(*erule-tac* $Q{=}Abs\text{-}Sequence_{base}\ \lfloor\lfloor\lceil\lceil Rep\text{-}Sequence_{base}\ x\rceil\rceil\ @\ [y]\rfloor\rfloor = Abs\text{-}Sequence_{base}\ None$ **in** *contrapos-pp*)

apply(*subst Abs-Sequence*$_{base}$*-inject* [*OF A*])

apply(*simp-all add*: *null-Sequence*$_{base}$*-def bot-Sequence*$_{base}$*-def bot-option-def*)

apply(*erule-tac* $Q{=}Abs\text{-}Sequence_{base}\lfloor\lfloor\lceil\lceil Rep\text{-}Sequence_{base}\ x\rceil\rceil\ @\ [y]\rfloor\rfloor = Abs\text{-}Sequence_{base}\ \lfloor None\rfloor$ **in** *contrapos-pp*)

apply(*subst Abs-Sequence*$_{base}$*-inject*[*OF A*])

apply(*simp-all add*: *null-Sequence*$_{base}$*-def bot-Sequence*$_{base}$*-def bot-option-def null-option-def*)

done

qed

syntax

-OclFinsequence :: *args* => $('\mathfrak{A},'a::null)$ *Sequence* (*Sequence*{(-)})

translations

Sequence{*x*, *xs*} == *CONST OclIncluding* (*Sequence*{*xs*}) *x*

Sequence{*x*} == *CONST OclIncluding* (*Sequence*{}) *x*

typ *int*

typ *num*

Definition: excluding

Definition: union

Definition: append identical to including

Definition: prepend

Definition: subSequence

Definition: at

Definition: first

Definition: last

Definition: asSet

instantiation *Sequence*$_{base}$ :: (*equal*)*equal*
begin
  definition *HOL.equal k l* $\longleftrightarrow$ (*k*::(*'a*::*equal*)*Sequence*$_{base}$) = *l*
  instance **by** *default* (*rule equal-Sequence*$_{base}$*-def*)
end

lemma *equal-Sequence*$_{base}$*-code* [*code*]:
  HOL.equal k (*l*::(*'a*::{*equal*,*null*})*Sequence*$_{base}$) $\longleftrightarrow$ *Rep-Sequence*$_{base}$ *k* = *Rep-Sequence*$_{base}$ *l*
  by (*auto simp add*: *equal Sequence*$_{base}$.*Rep-Sequence*$_{base}$*-inject*)

Test Statements

Assert $(\tau \models (\mathit{Sequence}\{\} \doteq \mathit{Sequence}\{\}))$
Assert $\tau \models (\mathit{Sequence}\{\mathbf{1},\mathit{invalid},\mathbf{2}\} \triangleq \mathit{invalid})$

end

theory *UML-Library*
imports
    basic-types/*UML-Boolean*
    basic-types/*UML-Void*
    basic-types/*UML-Integer*
    basic-types/*UML-Real*
    basic-types/*UML-String*

    collection-types/*UML-Pair*
    collection-types/*UML-Set*
    collection-types/*UML-Sequence*
begin

B.2.10. Miscellaneous Stuff

Properties on Collection Types: Strict Equality

The structure of this chapter roughly follows the structure of Chapter 10 of the OCL standard [28], which introduces the OCL Library.

MOVE TEXT : Collection Types

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, and
2. we want a possibility to nest collection types (so, we want the potential to talking about $Set(Set(Sequences(Pairs(X,Y))$

The former principle 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.

lemmas *cp-intro''* [*intro*!,*simp*,*code-unfold*] =
 cp-intro'

 $cp\text{-}intro''_{Set}$
 $cp\text{-}intro''_{Sequence}$

MOVE TEXT: Test Statements

lemma *syntax-test*: $Set\{\mathbf{2},\mathbf{1}\} = (Set\{\}{-}{>}including(\mathbf{1}){-}{>}including(\mathbf{2}))$
by (*rule refl*)

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-test2*:
assumes $H{:}(Set\{\mathbf{2}\} \doteq null) = (false{::}('\mathfrak{A})Boolean)$
shows $(\tau{::}('\mathfrak{A})st) \models (Set\{Set\{\mathbf{2}\},null\}{-}{>}includes(null))$
by(*simp add*: $OclIncludes\text{-}execute_{Set}$ *H*)

lemma *short-cut'*[*simp*,*code-unfold*]: $(\mathbf{8} \doteq \mathbf{6}) = false$
 apply(*rule ext*)
 apply(*simp add*: $StrictRefEq_{Integer}$ *StrongEq-def OclInt8-def OclInt6-def*
  true-def false-def invalid-def bot-option-def)
done

lemma *short-cut''*[*simp*,*code-unfold*]: $(\mathbf{2} \doteq \mathbf{1}) = \mathit{false}$
apply(*rule ext*)
apply(*simp add*: *StrictRefEq*$_{\mathit{Integer}}$ *StrongEq-def OclInt2-def OclInt1-def*
true-def false-def invalid-def bot-option-def)
done
lemma *short-cut'''*[*simp*,*code-unfold*]: $(\mathbf{1} \doteq \mathbf{2}) = \mathit{false}$
apply(*rule ext*)
apply(*simp add*: *StrictRefEq*$_{\mathit{Integer}}$ *StrongEq-def OclInt2-def OclInt1-def*
true-def false-def invalid-def bot-option-def)
done

Elementary computations on Sets.

declare *OclSelect-body-def* [*simp*]

Assert $\neg\,(\tau \models \upsilon(\mathit{invalid}::('\mathfrak{A},'\alpha::\mathit{null})\ \mathit{Set}))$
Assert $\tau \models \upsilon(\mathit{null}::('\mathfrak{A},'\alpha::\mathit{null})\ \mathit{Set})$
Assert $\neg\,(\tau \models \delta(\mathit{null}::('\mathfrak{A},'\alpha::\mathit{null})\ \mathit{Set}))$
Assert $\tau \models \upsilon(\mathit{Set}\{\})$
Assert $\tau \models \upsilon(\mathit{Set}\{\mathit{Set}\{\mathbf{2}\},\mathit{null}\})$
Assert $\tau \models \delta(\mathit{Set}\{\mathit{Set}\{\mathbf{2}\},\mathit{null}\})$
Assert $\tau \models (\mathit{Set}\{\mathbf{2},\mathbf{1}\}{-}{>}\mathit{includes}(\mathbf{1}))$
Assert $\neg\,(\tau \models (\mathit{Set}\{\mathbf{2}\}{-}{>}\mathit{includes}(\mathbf{1})))$
Assert $\neg\,(\tau \models (\mathit{Set}\{\mathbf{2},\mathbf{1}\}{-}{>}\mathit{includes}(\mathit{null})))$
Assert $\tau \models (\mathit{Set}\{\mathbf{2},\mathit{null}\}{-}{>}\mathit{includes}(\mathit{null}))$
Assert $\tau \models (\mathit{Set}\{\mathit{null},\mathbf{2}\}{-}{>}\mathit{includes}(\mathit{null}))$

Assert $\tau \models ((\mathit{Set}\{\}){-}{>}\mathit{forAll}(z \mid \mathbf{0} <_{\mathit{int}} z))$

Assert $\tau \models ((\mathit{Set}\{\mathbf{2},\mathbf{1}\}){-}{>}\mathit{forAll}(z \mid \mathbf{0} <_{\mathit{int}} z))$
Assert $\tau \models (\mathbf{0} <_{\mathit{int}} \mathbf{2})\ \mathit{and}\ (\mathbf{0} <_{\mathit{int}} \mathbf{1})$
Assert $\neg\,(\tau \models ((\mathit{Set}\{\mathbf{2},\mathbf{1}\}){-}{>}\mathit{exists}(z \mid z <_{\mathit{int}} \mathbf{0})))$
Assert $\neg\,(\tau \models (\delta(\mathit{Set}\{\mathbf{2},\mathit{null}\}){-}{>}\mathit{forAll}(z \mid \mathbf{0} <_{\mathit{int}} z)))$
Assert $\neg\,(\tau \models ((\mathit{Set}\{\mathbf{2},\mathit{null}\}){-}{>}\mathit{forAll}(z \mid \mathbf{0} <_{\mathit{int}} z)))$
Assert $\tau \models ((\mathit{Set}\{\mathbf{2},\mathit{null}\}){-}{>}\mathit{exists}(z \mid \mathbf{0} <_{\mathit{int}} z))$

Assert $\neg\,(\tau \models (\mathit{Set}\{\mathit{null}::'a\ \mathit{Boolean}\} \doteq \mathit{Set}\{\}))$
Assert $\neg\,(\tau \models (\mathit{Set}\{\mathit{null}::'a\ \mathit{Integer}\} \doteq \mathit{Set}\{\}))$

Assert $\neg\,(\tau \models (\mathit{Set}\{\mathit{true}\} \doteq \mathit{Set}\{\mathit{false}\}))$
Assert $\neg\,(\tau \models (\mathit{Set}\{\mathit{true},\mathit{true}\} \doteq \mathit{Set}\{\mathit{false}\}))$
Assert $\neg\,(\tau \models (\mathit{Set}\{\mathbf{2}\} \doteq \mathit{Set}\{\mathbf{1}\}))$
Assert $\tau \models (\mathit{Set}\{\mathbf{2},\mathit{null},\mathbf{2}\} \doteq \mathit{Set}\{\mathit{null},\mathbf{2}\})$
Assert $\tau \models (\mathit{Set}\{\mathbf{1},\mathit{null},\mathbf{2}\} <> \mathit{Set}\{\mathit{null},\mathbf{2}\})$
Assert $\tau \models (\mathit{Set}\{\mathit{Set}\{\mathbf{2},\mathit{null}\}\} \doteq \mathit{Set}\{\mathit{Set}\{\mathit{null},\mathbf{2}\}\})$
Assert $\tau \models (\mathit{Set}\{\mathit{Set}\{\mathbf{2},\mathit{null}\}\} <> \mathit{Set}\{\mathit{Set}\{\mathit{null},\mathbf{2}\},\mathit{null}\})$
Assert $\tau \models (\mathit{Set}\{\mathit{null}\}{-}{>}\mathit{select}(x \mid \mathit{not}\ x) \doteq \mathit{Set}\{\mathit{null}\})$
Assert $\tau \models (\mathit{Set}\{\mathit{null}\}{-}{>}\mathit{reject}(x \mid \mathit{not}\ x) \doteq \mathit{Set}\{\mathit{null}\})$

lemma $\mathit{const}\ (\mathit{Set}\{\mathit{Set}\{\mathbf{2},\mathit{null}\},\ \mathit{invalid}\})$ **by**$(\mathit{simp\ add}\colon \mathit{const\text{-}ss})$

end

B.3. Formalization III: UML/OCL constructs: State Operations and Objects

theory *UML-State*
imports *UML-Library*
begin

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

B.3.1. Introduction: States over Typed Object Universes

In the following, we will refine the concepts of a user-defined data-model (implied by a class-diagram) as well as the notion of state used in the previous section to much more detail. Surprisingly, even without a concrete notion of an objects and a universe of object representation, the generic infrastructure of state-related operations is fairly rich.

Fundamental Properties on Objects: Core Referential Equality

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

definition $\mathit{StrictRefEq}_{Object} :: ('\mathfrak{A},'a{::}\{\mathit{object},\mathit{null}\})\mathit{val} \Rightarrow ('\mathfrak{A},'a)\mathit{val} \Rightarrow ('\mathfrak{A})\mathit{Boolean}$
where $\mathit{StrictRefEq}_{Object}\ x\ y$

$$\begin{aligned}
&\equiv \lambda\ \tau.\ \mathit{if}\ (\upsilon\ x)\ \tau = \mathit{true}\ \tau \wedge (\upsilon\ y)\ \tau = \mathit{true}\ \tau\\
&\quad\ \mathit{then\ if}\ x\ \tau = \mathit{null} \vee y\ \tau = \mathit{null}\\
&\quad\quad\ \mathit{then}\ \lfloor\lfloor x\ \tau = \mathit{null} \wedge y\ \tau = \mathit{null}\rfloor\rfloor\\
&\quad\quad\ \mathit{else}\ \lfloor\lfloor(\mathit{oid\text{-}of}\ (x\ \tau)) = (\mathit{oid\text{-}of}\ (y\ \tau))\ \rfloor\rfloor\\
&\quad\ \mathit{else\ invalid}\ \tau
\end{aligned}$$

Strictness and context passing **lemma** $\mathit{StrictRefEq}_{Object}\text{-}\mathit{strict1}[\mathit{simp},\mathit{code\text{-}unfold}]$:
$(\mathit{StrictRefEq}_{Object}\ x\ \mathit{invalid}) = \mathit{invalid}$
by$(\mathit{rule\ ext},\ \mathit{simp\ add}\colon \mathit{StrictRefEq}_{Object}\text{-}\mathit{def\ true\text{-}def\ false\text{-}def})$

lemma $\mathit{StrictRefEq}_{Object}\text{-}\mathit{strict2}[\mathit{simp},\mathit{code\text{-}unfold}]$:
$(\mathit{StrictRefEq}_{Object}\ \mathit{invalid}\ x) = \mathit{invalid}$
by$(\mathit{rule\ ext},\ \mathit{simp\ add}\colon \mathit{StrictRefEq}_{Object}\text{-}\mathit{def\ true\text{-}def\ false\text{-}def})$

lemma $\mathit{cp\text{-}StrictRefEq}_{Object}$:
$(\mathit{StrictRefEq}_{Object}\ x\ y\ \tau) = (\mathit{StrictRefEq}_{Object}\ (\lambda\text{-}.\ x\ \tau)\ (\lambda\text{-}.\ y\ \tau))\ \tau$
by$(\mathit{auto\ simp}\colon \mathit{StrictRefEq}_{Object}\text{-}\mathit{def\ cp\text{-}valid}[\mathit{symmetric}])$

lemmas *cp0-StrictRefEq*$_{Object}$= *cp-StrictRefEq*$_{Object}$[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]],
of StrictRefEq$_{Object}$]]

lemmas *cp-intro*$''$[*intro*!,*simp*,*code-unfold*] =
cp-intro$''$
cp-StrictRefEq$_{Object}$[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]],
of StrictRefEq$_{Object}$]]

Logic and Algebraic Layer on Object

Validity and Definedness Properties We derive the usual laws on definedness for (generic) object equality:

lemma *StrictRefEq*$_{Object}$*-defargs*:
$\tau \models$ (*StrictRefEq*$_{Object}$ *x* (*y*::($'\mathfrak{A}$,$'a$::{*null*,*object*})*val*))$\Longrightarrow$ ($\tau \models$(υ *x*)) $\wedge$ ($\tau \models$(υ *y*))
by(*simp add*: *StrictRefEq*$_{Object}$*-def OclValid-def true-def invalid-def bot-option-def*
split: *bool.split-asm HOL.split-if-asm*)

lemma *defined-StrictRefEq*$_{Object}$*-I*:
assumes *val-x* : $\tau \models \upsilon\ x$
assumes *val-x* : $\tau \models \upsilon\ y$
shows $\tau \models \delta$ (*StrictRefEq*$_{Object}$ *x y*)
apply(*insert assms*, *simp add*: *StrictRefEq*$_{Object}$*-def OclValid-def*)
by(*subst cp-defined*, *simp add*: *true-def*)

lemma *StrictRefEq*$_{Object}$*-def-homo* :
δ(*StrictRefEq*$_{Object}$ *x* (*y*::($'\mathfrak{A}$,$'a$::{*null*,*object*})*val*)) = ((υ *x*) *and* (υ *y*))
sorry

Symmetry **lemma** *StrictRefEq*$_{Object}$*-sym* :
assumes *x-val* : $\tau \models \upsilon\ x$
shows $\tau \models$ *StrictRefEq*$_{Object}$ *x x*
by(*simp add*: *StrictRefEq*$_{Object}$*-def true-def OclValid-def*
x-val[*simplified OclValid-def*])

Behavior vs StrongEq It remains to clarify the role of the state invariant $\mathrm{inv}_\sigma(\sigma)$ mentioned above that states the condition that there is a "one-to-one" correspondence between object representations and oid's: $\forall oid \in \operatorname{dom}\sigma.\ oid = \mathrm{OidOf}\ulcorner\sigma(oid)\urcorner$. This condition is also mentioned in [28, Annex A] and goes back to Richters [30]; however, we state this condition as an invariant on states rather than a global axiom. It can, therefore, not be taken for granted that an oid makes sense both in pre- and post-states of OCL expressions.

We capture this invariant in the predicate WFF :

definition *WFF* :: ($'\mathfrak{A}$::*object*)*st* $\Rightarrow$ *bool*
where *WFF* τ = (($\forall$ *x* $\in$ *ran*(*heap*(*fst* τ)). $\lceil$*heap*(*fst* τ) (*oid-of x*)$\rceil$ = *x*) $\wedge$
($\forall$ *x* $\in$ *ran*(*heap*(*snd* τ)). $\lceil$*heap*(*snd* τ) (*oid-of x*)$\rceil$ = *x*))

It turns out that WFF is 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.

We turn now to the generic definition of referential equality on objects: Equality on objects in a state is reduced to equality on the references to these objects. As in HOL-OCL [5, 7], 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-correspondence" 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.

theorem *StrictRefEq$_{Object}$-vs-StrongEq*:
assumes *WFF*: *WFF* τ
and *valid-x*: $\tau \models (\upsilon\ x)$
and *valid-y*: $\tau \models (\upsilon\ y)$
and *x-present-pre*: $x\ \tau \in \mathit{ran}\ (\mathit{heap}(\mathit{fst}\ \tau))$
and *y-present-pre*: $y\ \tau \in \mathit{ran}\ (\mathit{heap}(\mathit{fst}\ \tau))$
and *x-present-post*:$x\ \tau \in \mathit{ran}\ (\mathit{heap}(\mathit{snd}\ \tau))$
and *y-present-post*:$y\ \tau \in \mathit{ran}\ (\mathit{heap}(\mathit{snd}\ \tau))$

shows $(\tau \models (\mathit{StrictRefEq}_{Object}\ x\ y)) = (\tau \models (x \triangleq y))$
apply(*insert WFF valid-x valid-y x-present-pre y-present-pre x-present-post y-present-post*)
apply(*auto simp*: *StrictRefEq$_{Object}$-def OclValid-def WFF-def StrongEq-def true-def Ball-def*)
apply(*erule-tac x=x* τ **in** *allE$'$*, *simp-all*)
done

theorem *StrictRefEq$_{Object}$-vs-StrongEq$'$*:
assumes *WFF*: *WFF* τ
and *valid-x*: $\tau \models (\upsilon\ (x :: ('\mathfrak{A}::\mathit{object},'\alpha::\{\mathit{null},\mathit{object}\})\mathit{val}))$
and *valid-y*: $\tau \models (\upsilon\ y)$
and *oid-preserve*: $\bigwedge x.\ x \in \mathit{ran}\ (\mathit{heap}(\mathit{fst}\ \tau)) \lor x \in \mathit{ran}\ (\mathit{heap}(\mathit{snd}\ \tau)) \Longrightarrow$
$H\ x \neq \bot \Longrightarrow \mathit{oid\text{-}of}\ (H\ x) = \mathit{oid\text{-}of}\ x$
and *xy-together*: $x\ \tau \in H\ `\ \mathit{ran}\ (\mathit{heap}(\mathit{fst}\ \tau)) \land y\ \tau \in H\ `\ \mathit{ran}\ (\mathit{heap}(\mathit{fst}\ \tau)) \lor$
$x\ \tau \in H\ `\ \mathit{ran}\ (\mathit{heap}(\mathit{snd}\ \tau)) \land y\ \tau \in H\ `\ \mathit{ran}\ (\mathit{heap}(\mathit{snd}\ \tau))$

shows $(\tau \models (\mathit{StrictRefEq}_{Object}\ x\ y)) = (\tau \models (x \triangleq y))$
apply(*insert WFF valid-x valid-y xy-together*)
apply(*simp add*: *WFF-def*)
apply(*auto simp*: *StrictRefEq$_{Object}$-def OclValid-def WFF-def StrongEq-def true-def Ball-def*)
by (*metis foundation18$'$ oid-preserve valid-x valid-y*)+

So, if two object descriptions live in the same state (both pre or post), the referential equality on objects implies in a WFF state the logical equality.

B.3.2. Operations on Object

Initial States (for testing and code generation)

definition $\tau_0 :: ('\mathfrak{A})st$
where $\tau_0 \equiv ((\!|heap{=}Map.empty, assocs = Map.empty|\!),$
$(\!|heap{=}Map.empty, assocs = Map.empty|\!))$

OclAllInstances

To denote OCL types occurring in OCL expressions syntactically—as, for example, as "argument" of `oclAllInstances(` we use the inverses of the injection functions into the object universes; we show that this is a sufficient "characterization."

definition $\mathit{OclAllInstances\text{-}generic} :: (('\mathfrak{A}::object)\ st \Rightarrow '\mathfrak{A}\ state) \Rightarrow ('\mathfrak{A}::object \rightharpoonup '\alpha) \Rightarrow$
$('\mathfrak{A}, '\alpha\ option\ option)\ Set$
where $\mathit{OclAllInstances\text{-}generic\ fst\text{-}snd\ H} =$
$(\lambda\tau.\ \mathit{Abs\text{-}Set}_{base}\ \lfloor\lfloor\ \mathit{Some}\ `\ ((H\ `\ ran\ (heap\ (fst\text{-}snd\ \tau))) - \{\ None\ \})\ \rfloor\rfloor)$

lemma $\mathit{OclAllInstances\text{-}generic\text{-}defined}$: $\tau \models \delta\ (\mathit{OclAllInstances\text{-}generic\ pre\text{-}post\ H})$
apply(*simp add*: *defined-def OclValid-def OclAllInstances-generic-def false-def true-def*
bot-fun-def bot-Set$_{base}$-def null-fun-def null-Set$_{base}$-def)
apply(*rule conjI*)
apply(*rule notI*, *subst* (*asm*) *Abs-Set$_{base}$-inject*, *simp*,
(*rule disjI2*)+,
metis bot-option-def option.distinct(*1*),
(*simp add*: *bot-option-def null-option-def*)+)+
done

lemma *OclAllInstances-generic-init-empty*:
assumes [*simp*]: $\bigwedge x.\ pre\text{-}post\ (x, x) = x$
shows $\tau_0 \models \mathit{OclAllInstances\text{-}generic\ pre\text{-}post\ H} \triangleq \mathit{Set}\{\}$
by(*simp add*: *StrongEq-def OclAllInstances-generic-def OclValid-def* τ_0*-def mtSet-def*)

lemma *represented-generic-objects-nonnull*:
assumes A: $\tau \models ((\mathit{OclAllInstances\text{-}generic\ pre\text{-}post}\ (H::('\mathfrak{A}::object \rightharpoonup '\alpha)))\ {-}{>}includes(x))$
shows $\tau \models not(x \triangleq null)$
proof −
have B: $\tau \models \delta\ (\mathit{OclAllInstances\text{-}generic\ pre\text{-}post\ H})$
by(*insert A*[*THEN foundation6*,
simplified OclIncludes-defined-args-valid], *auto*)
have C: $\tau \models \upsilon\ x$
by(*insert A*[*THEN foundation6*,
simplified OclIncludes-defined-args-valid], *auto*)
show *?thesis*
apply(*insert A*)
apply(*simp add*: *StrongEq-def OclValid-def*
OclNot-def null-def true-def OclIncludes-def B[*simplified OclValid-def*]
C[*simplified OclValid-def*])

```
  apply(simp add:OclAllInstances-generic-def)
  apply(erule contrapos-pn)
  apply(subst Set_base.Abs-Set_base-inverse,
       auto simp: null-fun-def null-option-def bot-option-def)
  done
qed
```

```
lemma represented-generic-objects-defined:
assumes A: τ ⊨ ((OclAllInstances-generic pre-post (H::('𝔄::object ⇀ 'α))) −>includes(x))
shows    τ ⊨ δ (OclAllInstances-generic pre-post H) ∧ τ ⊨ δ x
apply(insert  A[THEN foundation6,
            simplified OclIncludes-defined-args-valid])
apply(simp add: foundation16 foundation18 invalid-def, erule conjE)
apply(insert A[THEN represented-generic-objects-nonnull])
by(simp add: foundation24 null-fun-def)
```

One way to establish the actual presence of an object representation in a state is:

```
lemma represented-generic-objects-in-state:
assumes A: τ ⊨ (OclAllInstances-generic pre-post H)−>includes(x)
shows    x τ ∈ (Some o H) ‘ ran (heap(pre-post τ))
proof −
  have B: (δ (OclAllInstances-generic pre-post H)) τ = true τ
       by(simp add: OclValid-def[symmetric] OclAllInstances-generic-defined)
  have C: (υ x) τ = true τ
       by(insert  A[THEN foundation6,
                 simplified OclIncludes-defined-args-valid],
          auto simp: OclValid-def)
  have F: Rep-Set_base (Abs-Set_base ⌊⌊Some ‘ (H ‘ ran (heap (pre-post τ)) − {None})⌋⌋) =
        ⌊⌊Some ‘ (H ‘ ran (heap (pre-post τ)) − {None})⌋⌋
       by(subst Set_base.Abs-Set_base-inverse,simp-all add: bot-option-def)
  show ?thesis
   apply(insert A)
   apply(simp add: OclIncludes-def OclValid-def ran-def B C image-def true-def)
   apply(simp add: OclAllInstances-generic-def)
   apply(simp add: F)
   apply(simp add: ran-def)
  by(fastforce)
qed
```

```
lemma state-update-vs-allInstances-generic-empty:
assumes [simp]: ⋀a. pre-post (mk a) = a
shows   (mk ⦇heap=empty, assocs=A⦈) ⊨ OclAllInstances-generic pre-post Type ≐ Set{}
proof −
 have state-update-vs-allInstances-empty:
  (OclAllInstances-generic pre-post Type) (mk ⦇heap=empty, assocs=A⦈) =
  Set{} (mk ⦇heap=empty, assocs=A⦈)
 by(simp add: OclAllInstances-generic-def mtSet-def)
 show ?thesis
  apply(simp only: OclValid-def, subst StrictRefEq_Set.cp0,
       simp only: state-update-vs-allInstances-empty StrictRefEq_Set.refl-ext)
  apply(simp add: OclIf-def valid-def mtSet-def defined-def
               bot-fun-def null-fun-def null-option-def bot-Set_base-def)
 by(subst Abs-Set_base-inject, (simp add: bot-option-def true-def)+)
qed
```

Here comes a couple of operational rules that allow to infer the value of oclAllInstances from the context τ. These rules are a special-case in the sense that they are the only rules that relate statements with *different* τ's. For that reason, new concepts like "constant contexts P" are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

```
lemma state-update-vs-allInstances-generic-including':
assumes [simp]: ⋀a. pre-post (mk a) = a
assumes ⋀x. σ' oid = Some x ⟹ x = Object
   and Type Object ≠ None
 shows (OclAllInstances-generic pre-post Type)
       (mk ⦇heap=σ'(oid↦Object), assocs=A⦈)
       =
       ((OclAllInstances-generic pre-post Type)->including(λ -. ⌊⌊ drop (Type Object) ⌋⌋))
       (mk ⦇heap=σ',assocs=A⦈)
proof -
 have drop-none : ⋀x. x ≠ None ⟹ ⌊⌈x⌉⌋ = x
 by(case-tac x, simp+)

 have insert-diff : ⋀x S. insert ⌊x⌋ (S - {None}) = (insert ⌊x⌋ S) - {None}
 by (metis insert-Diff-if option.distinct(1) singletonE)

 show ?thesis
  apply(simp add: UML-Set.OclIncluding-def OclAllInstances-generic-defined[simplified OclValid-def],
       simp add: OclAllInstances-generic-def)
  apply(subst Abs-Set_base-inverse, simp add: bot-option-def, simp add: comp-def,
       subst image-insert[symmetric],
       subst drop-none, simp add: assms)
  apply(case-tac Type Object, simp add: assms, simp only:,
       subst insert-diff, drule sym, simp)
  apply(subgoal-tac ran (σ'(oid ↦ Object)) = insert Object (ran σ'), simp)
  apply(case-tac ¬ (∃x. σ' oid = Some x))
   apply(rule ran-map-upd, simp)
  apply(simp, erule exE, frule assms, simp)
  apply(subgoal-tac Object ∈ ran σ') prefer 2
   apply(rule ranI, simp)
 by(subst insert-absorb, simp, metis fun-upd-apply)

qed
```

```
lemma state-update-vs-allInstances-generic-including:
assumes [simp]: ⋀a. pre-post (mk a) = a
assumes ⋀x. σ' oid = Some x ⟹ x = Object
   and Type Object ≠ None
shows   (OclAllInstances-generic pre-post Type)
        (mk ⦇heap=σ'(oid↦Object), assocs=A⦈)
        =
        ((λ-. (OclAllInstances-generic pre-post Type)
              (mk ⦇heap=σ', assocs=A⦈))−>including(λ -. ⌊⌊ drop (Type Object) ⌋⌋))
        (mk ⦇heap=σ'(oid↦Object), assocs=A⦈)
 apply(subst state-update-vs-allInstances-generic-including', (simp add: assms)+,
      subst cp-OclIncluding,
      simp add: UML-Set.OclIncluding-def)
 apply(subst (1 3) cp-defined[symmetric],
      simp add: OclAllInstances-generic-defined[simplified OclValid-def])

 apply(simp add: defined-def OclValid-def OclAllInstances-generic-def invalid-def
             bot-fun-def null-fun-def bot-Set_base-def null-Set_base-def)
 apply(subst (1 3) Abs-Set_base-inject)
by(simp add: bot-option-def null-option-def)+



lemma state-update-vs-allInstances-generic-noincluding':
assumes [simp]: ⋀a. pre-post (mk a) = a
assumes ⋀x. σ' oid = Some x ⟹ x = Object
   and Type Object = None
  shows (OclAllInstances-generic pre-post Type)
        (mk ⦇heap=σ'(oid↦Object), assocs=A⦈)
        =
        (OclAllInstances-generic pre-post Type)
        (mk ⦇heap=σ', assocs=A⦈)
proof −
 have drop-none : ⋀x. x ≠ None ⟹ ⌊⌈x⌉⌋ = x
 by(case-tac x, simp+)

 have insert-diff : ⋀x S. insert ⌊x⌋ (S − {None}) = (insert ⌊x⌋ S) − {None}
 by (metis insert-Diff-if option.distinct(1) singletonE)

 show ?thesis
  apply(simp add: OclIncluding-def OclAllInstances-generic-defined[simplified OclValid-def]
              OclAllInstances-generic-def)
  apply(subgoal-tac ran (σ'(oid ↦ Object)) = insert Object (ran σ'), simp add: assms)
  apply(case-tac ¬ (∃x. σ' oid = Some x))
   apply(rule ran-map-upd, simp)
  apply(simp, erule exE, frule assms, simp)
  apply(subgoal-tac Object ∈ ran σ′) prefer 2
   apply(rule ranI, simp)
  apply(subst insert-absorb, simp)
 by (metis fun-upd-apply)
qed

theorem state-update-vs-allInstances-generic-ntc:
assumes [simp]: ⋀a. pre-post (mk a) = a
assumes oid-def:   oid∉dom σ′
and  non-type-conform: Type Object = None
and  cp-ctxt:      cp P
and  const-ctxt:   ⋀X. const X ⟹ const (P X)
shows (mk ⦇heap=σ′(oid↦Object),assocs=A⦈ ⊨ P (OclAllInstances-generic pre-post Type)) =
      (mk ⦇heap=σ′, assocs=A⦈            ⊨ P (OclAllInstances-generic pre-post Type))
     (is (?τ ⊨ P ?φ) = (?τ′ ⊨ P ?φ))
proof −
 have P-cp  : ⋀x τ. P x τ = P (λ-. x τ) τ
          by (metis (full-types) cp-ctxt cp-def)
 have A     : const (P (λ-. ?φ ?τ))
          by(simp add: const-ctxt const-ss)
 have       (?τ ⊨ P ?φ) = (?τ ⊨ λ-. P ?φ ?τ)
          by(subst foundation23, rule refl)
 also have  ... = (?τ ⊨ λ-. P (λ-. ?φ ?τ)  ?τ)
          by(subst P-cp, rule refl)
 also have  ... = (?τ′ ⊨ λ-. P (λ-. ?φ ?τ)  ?τ′)
          apply(simp add: OclValid-def)
          by(subst A[simplified const-def], subst const-true[simplified const-def], simp)
 finally have X: (?τ ⊨ P ?φ) = (?τ′ ⊨ λ-. P (λ-. ?φ ?τ)  ?τ′)
          by simp
 show ?thesis
 apply(subst X) apply(subst foundation23[symmetric])
 apply(rule StrongEq-L-subst3[OF cp-ctxt])
 apply(simp add: OclValid-def StrongEq-def true-def)
 apply(rule state-update-vs-allInstances-generic-noincluding′)
 by(insert oid-def, auto simp: non-type-conform)
qed

theorem state-update-vs-allInstances-generic-tc:
assumes [simp]: ⋀a. pre-post (mk a) = a
assumes oid-def:   oid∉dom σ′
and  type-conform: Type Object ≠ None
and  cp-ctxt:      cp P
and  const-ctxt:   ⋀X. const X ⟹ const (P X)
shows (mk ⦇heap=σ′(oid↦Object),assocs=A⦈ ⊨ P (OclAllInstances-generic pre-post Type)) =
      (mk ⦇heap=σ′, assocs=A⦈            ⊨ P ((OclAllInstances-generic pre-post Type)
                                              −>including(λ -. ⌊(Type Object)⌋)))
      (is (?τ ⊨ P ?φ) = (?τ′ ⊨ P ?φ′))
proof −
```

```
 have P-cp  : ⋀x τ. P x τ = P (λ-. x τ) τ
          by (metis (full-types) cp-ctxt cp-def)
 have A     : const (P (λ-. ?φ ?τ))
          by(simp add: const-ctxt const-ss)
 have      (?τ ⊨ P ?φ) = (?τ ⊨ λ-. P ?φ ?τ)
          by(subst foundation23, rule refl)
 also have  ... = (?τ ⊨ λ-. P (λ-. ?φ ?τ)  ?τ)
          by(subst P-cp, rule refl)
 also have  ... = (?τ′ ⊨ λ-. P (λ-. ?φ ?τ)  ?τ′)
          apply(simp add: OclValid-def)
          by(subst A[simplified const-def], subst const-true[simplified const-def], simp)
 finally have X: (?τ ⊨ P ?φ) = (?τ′ ⊨ λ-. P (λ-. ?φ ?τ)  ?τ′)
          by simp
 let        ?allInstances = OclAllInstances-generic pre-post Type
 have        ?allInstances ?τ = λ-. ?allInstances ?τ′−>including(λ-.⌊⌊⌈Type Object⌉⌋⌋) ?τ
          apply(rule state-update-vs-allInstances-generic-including)
          by(insert oid-def, auto simp: type-conform)
 also have   ... = ((λ-. ?allInstances ?τ′)−>including(λ-. (λ-.⌊⌊⌈Type Object⌉⌋⌋) ?τ′) ?τ′)
          by(subst const-OclIncluding[simplified const-def], simp+)
 also have   ... = (?allInstances−>including(λ -. ⌊Type Object⌋) ?τ′)
          apply(subst cp-OclIncluding[symmetric])
          by(insert type-conform, auto)
 finally have Y : ?allInstances ?τ = (?allInstances−>including(λ -. ⌊Type Object⌋) ?τ′)
          by auto
 show ?thesis
    apply(subst X) apply(subst foundation23[symmetric])
    apply(rule StrongEq-L-subst3[OF cp-ctxt])
    apply(simp add: OclValid-def StrongEq-def Y true-def)
 done
qed

declare OclAllInstances-generic-def [simp]
```

OclAllInstances (@post)

```
definition OclAllInstances-at-post :: (′𝔄 :: object ⇀ ′α) ⇒ (′𝔄, ′α option option) Set
                      (- .allInstances′(′))
where  OclAllInstances-at-post =  OclAllInstances-generic snd

lemma OclAllInstances-at-post-defined: τ ⊨ δ (H .allInstances())
unfolding OclAllInstances-at-post-def
by(rule OclAllInstances-generic-defined)

lemma τ₀ ⊨ H .allInstances() ≜ Set{}
unfolding OclAllInstances-at-post-def
by(rule OclAllInstances-generic-init-empty, simp)


lemma represented-at-post-objects-nonnull:
assumes A: τ ⊨ (((H::(′𝔄::object ⇀ ′α)).allInstances()) −>includes(x))
```

shows $\tau \models not(x \triangleq null)$
by(*rule represented-generic-objects-nonnull*[*OF A*[*simplified OclAllInstances-at-post-def*]])

lemma *represented-at-post-objects-defined*:
assumes A: $\tau \models (((H::('\mathfrak{A}::object \rightharpoonup '\alpha)).allInstances()) \; {-}{>}includes(x))$
shows $\tau \models \delta \; (H \; .allInstances()) \wedge \tau \models \delta \; x$
unfolding *OclAllInstances-at-post-def*
by(*rule represented-generic-objects-defined*[*OF A*[*simplified OclAllInstances-at-post-def*]])

One way to establish the actual presence of an object representation in a state is:

lemma
assumes A: $\tau \models H \; .allInstances(){-}{>}includes(x)$
shows $x \; \tau \in (Some \; o \; H) \; ` \; ran \; (heap(snd \; \tau))$
by(*rule represented-generic-objects-in-state*[*OF A*[*simplified OclAllInstances-at-post-def*]])

lemma *state-update-vs-allInstances-at-post-empty*:
shows $(\sigma, (\!|heap{=}empty, assocs{=}A|\!)) \models Type \; .allInstances() \doteq Set\{\}$
unfolding *OclAllInstances-at-post-def*
by(*rule state-update-vs-allInstances-generic-empty*[*OF snd-conv*])

Here comes a couple of operational rules that allow to infer the value of oclAllInstances from the context τ. These rules are a special-case in the sense that they are the only rules that relate statements with *different* τ's. For that reason, new concepts like “constant contexts P” are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

lemma *state-update-vs-allInstances-at-post-including'*:
assumes $\bigwedge x. \; \sigma' \; oid = Some \; x \Longrightarrow x = Object$
  and $Type \; Object \neq None$
 shows $(Type \; .allInstances())$
    $(\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!))$
    $=$
    $((Type \; .allInstances()){-}{>}including(\lambda \text{ -. } \lfloor\lfloor \; drop \; (Type \; Object) \; \rfloor\rfloor))$
    $(\sigma, (\!|heap{=}\sigma', assocs{=}A|\!))$
unfolding *OclAllInstances-at-post-def*
by(*rule state-update-vs-allInstances-generic-including'*[*OF snd-conv*], *insert assms*)

lemma *state-update-vs-allInstances-at-post-including*:
assumes $\bigwedge x. \; \sigma' \; oid = Some \; x \Longrightarrow x = Object$
  and $Type \; Object \neq None$
shows $(Type \; .allInstances())$
    $(\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!))$
    $=$
    $((\lambda\text{-. } (Type \; .allInstances())$
      $(\sigma, (\!|heap{=}\sigma', assocs{=}A|\!))){-}{>}including(\lambda \text{ -. } \lfloor\lfloor \; drop \; (Type \; Object) \; \rfloor\rfloor))$
    $(\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!))$

unfolding *OclAllInstances-at-post-def*
by(*rule state-update-vs-allInstances-generic-including*[*OF snd-conv*], *insert assms*)

lemma *state-update-vs-allInstances-at-post-noincluding*$'$:
assumes $\bigwedge x.\ \sigma'\ oid = Some\ x \Longrightarrow x = Object$
  and $Type\ Object = None$
 shows $(Type\ .allInstances())$
   $(\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!))$
   $=$
   $(Type\ .allInstances())$
   $(\sigma, (\!|heap{=}\sigma', assocs{=}A|\!))$
unfolding *OclAllInstances-at-post-def*
by(*rule state-update-vs-allInstances-generic-noincluding*$'$[*OF snd-conv*], *insert assms*)

theorem *state-update-vs-allInstances-at-post-ntc*:
assumes *oid-def*: $oid{\notin}dom\ \sigma'$
and *non-type-conform*: $Type\ Object = None$
and *cp-ctxt*: $cp\ P$
and *const-ctxt*: $\bigwedge X.\ const\ X \Longrightarrow const\ (P\ X)$
shows $((\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object),assocs{=}A|\!)) \models (P(Type\ .allInstances()))) =$
   $((\sigma, (\!|heap{=}\sigma', assocs{=}A|\!))\qquad \models (P(Type\ .allInstances())))$
unfolding *OclAllInstances-at-post-def*
by(*rule state-update-vs-allInstances-generic-ntc*[*OF snd-conv*], *insert assms*)

theorem *state-update-vs-allInstances-at-post-tc*:
assumes *oid-def*: $oid{\notin}dom\ \sigma'$
and *type-conform*: $Type\ Object \neq None$
and *cp-ctxt*: $cp\ P$
and *const-ctxt*: $\bigwedge X.\ const\ X \Longrightarrow const\ (P\ X)$
shows $((\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object),assocs{=}A|\!)) \models (P(Type\ .allInstances()))) =$
   $((\sigma, (\!|heap{=}\sigma', assocs{=}A|\!))\qquad \models (P((Type\ .allInstances())$
              $->including(\lambda\ \text{-}.\ \lfloor(Type\ Object)\rfloor))))$
unfolding *OclAllInstances-at-post-def*
by(*rule state-update-vs-allInstances-generic-tc*[*OF snd-conv*], *insert assms*)

OclAllInstances (@pre) **definition** *OclAllInstances-at-pre* :: $('\mathfrak{A} :: object \rightharpoonup '\alpha) \Rightarrow ('\mathfrak{A}, '\alpha\ option\ option)\ Set$
      $(\text{-}\ .allInstances@pre'(')$)
where $OclAllInstances\text{-}at\text{-}pre = OclAllInstances\text{-}generic\ fst$

lemma *OclAllInstances-at-pre-defined*: $\tau \models \delta\ (H\ .allInstances@pre())$
unfolding *OclAllInstances-at-pre-def*
by(*rule OclAllInstances-generic-defined*)

lemma $\tau_0 \models H\ .allInstances@pre() \triangleq Set\{\}$
unfolding *OclAllInstances-at-pre-def*
by(*rule OclAllInstances-generic-init-empty*, *simp*)

lemma *represented-at-pre-objects-nonnull*:
assumes A: $\tau \models (((H{::}({}'\mathfrak{A}{::}\mathit{object} \rightharpoonup {}'\alpha)).\mathit{allInstances@pre}())\ {-}{>}\mathit{includes}(x))$
shows $\tau \models \mathit{not}(x \triangleq \mathit{null})$
by(*rule represented-generic-objects-nonnull*[*OF A*[*simplified OclAllInstances-at-pre-def*]])

lemma *represented-at-pre-objects-defined*:
assumes A: $\tau \models (((H{::}({}'\mathfrak{A}{::}\mathit{object} \rightharpoonup {}'\alpha)).\mathit{allInstances@pre}())\ {-}{>}\mathit{includes}(x))$
shows $\tau \models \delta\ (H\ .\mathit{allInstances@pre}()) \wedge \tau \models \delta\ x$
unfolding *OclAllInstances-at-pre-def*
by(*rule represented-generic-objects-defined*[*OF A*[*simplified OclAllInstances-at-pre-def*]])

One way to establish the actual presence of an object representation in a state is:

lemma
assumes A: $\tau \models H\ .\mathit{allInstances@pre}(){-}{>}\mathit{includes}(x)$
shows $x\ \tau \in (\mathit{Some}\ o\ H)\ `\ \mathit{ran}\ (\mathit{heap}(\mathit{fst}\ \tau))$
by(*rule represented-generic-objects-in-state*[*OF A*[*simplified OclAllInstances-at-pre-def*]])

lemma *state-update-vs-allInstances-at-pre-empty*:
shows $((\!|\mathit{heap}{=}\mathit{empty}, \mathit{assocs}{=}A|\!), \sigma) \models \mathit{Type}\ .\mathit{allInstances@pre}() \doteq \mathit{Set}\{\}$
unfolding *OclAllInstances-at-pre-def*
by(*rule state-update-vs-allInstances-generic-empty*[*OF fst-conv*])

Here comes a couple of operational rules that allow to infer the value of oclAllInstances@pre from the context τ. These rules are a special-case in the sense that they are the only rules that relate statements with *different* τ's. For that reason, new concepts like "constant contexts P" are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

lemma *state-update-vs-allInstances-at-pre-including*$'$:
assumes $\bigwedge x.\ \sigma'\ \mathit{oid} = \mathit{Some}\ x \Longrightarrow x = \mathit{Object}$
and $\mathit{Type}\ \mathit{Object} \neq \mathit{None}$
shows $(\mathit{Type}\ .\mathit{allInstances@pre}())$
$((\!|\mathit{heap}{=}\sigma'(\mathit{oid}{\mapsto}\mathit{Object}), \mathit{assocs}{=}A|\!), \sigma)$
$=$
$((\mathit{Type}\ .\mathit{allInstances@pre}()){-}{>}\mathit{including}(\lambda\ \text{-}.\ \lfloor\lfloor\ \mathit{drop}\ (\mathit{Type}\ \mathit{Object})\ \rfloor\rfloor))$
$((\!|\mathit{heap}{=}\sigma', \mathit{assocs}{=}A|\!), \sigma)$
unfolding *OclAllInstances-at-pre-def*
by(*rule state-update-vs-allInstances-generic-including*$'$[*OF fst-conv*], *insert assms*)

lemma *state-update-vs-allInstances-at-pre-including*:
assumes $\bigwedge x.\ \sigma'\ \mathit{oid} = \mathit{Some}\ x \Longrightarrow x = \mathit{Object}$
and $\mathit{Type}\ \mathit{Object} \neq \mathit{None}$
shows $(\mathit{Type}\ .\mathit{allInstances@pre}())$
$((\!|\mathit{heap}{=}\sigma'(\mathit{oid}{\mapsto}\mathit{Object}), \mathit{assocs}{=}A|\!), \sigma)$

$=$
$((\lambda\text{-}.\ (\mathit{Type}\ .\mathit{allInstances@pre}())$
$(\llparenthesis \mathit{heap}{=}\sigma',\ \mathit{assocs}{=}A\rrparenthesis,\ \sigma))-{>}\mathit{including}(\lambda\ \text{-}.\ \lfloor\lfloor\ \mathit{drop}\ (\mathit{Type\ Object})\ \rfloor\rfloor))$
$(\llparenthesis \mathit{heap}{=}\sigma'(\mathit{oid}{\mapsto}\mathit{Object}),\ \mathit{assocs}{=}A\rrparenthesis,\ \sigma)$
unfolding *OclAllInstances-at-pre-def*
by(*rule state-update-vs-allInstances-generic-including*[*OF fst-conv*], *insert assms*)

lemma *state-update-vs-allInstances-at-pre-noincluding*$'$:
assumes $\bigwedge x.\ \sigma'\ \mathit{oid} = \mathit{Some}\ x \Longrightarrow x = \mathit{Object}$
and $\mathit{Type\ Object} = \mathit{None}$
shows $(\mathit{Type}\ .\mathit{allInstances@pre}())$
$(\llparenthesis \mathit{heap}{=}\sigma'(\mathit{oid}{\mapsto}\mathit{Object}),\ \mathit{assocs}{=}A\rrparenthesis,\ \sigma)$
$=$
$(\mathit{Type}\ .\mathit{allInstances@pre}())$
$(\llparenthesis \mathit{heap}{=}\sigma',\ \mathit{assocs}{=}A\rrparenthesis,\ \sigma)$
unfolding *OclAllInstances-at-pre-def*
by(*rule state-update-vs-allInstances-generic-noincluding*$'$[*OF fst-conv*], *insert assms*)

theorem *state-update-vs-allInstances-at-pre-ntc*:
assumes *oid-def*: $\mathit{oid}{\notin}\mathit{dom}\ \sigma'$
and *non-type-conform*: $\mathit{Type\ Object} = \mathit{None}$
and *cp-ctxt*: $\mathit{cp}\ P$
and *const-ctxt*: $\bigwedge X.\ \mathit{const}\ X \Longrightarrow \mathit{const}\ (P\ X)$
shows $(((\llparenthesis \mathit{heap}{=}\sigma'(\mathit{oid}{\mapsto}\mathit{Object}),\mathit{assocs}{=}A\rrparenthesis,\ \sigma) \models (P(\mathit{Type}\ .\mathit{allInstances@pre}()))) =$
$(((\llparenthesis \mathit{heap}{=}\sigma',\ \mathit{assocs}{=}A\rrparenthesis,\ \sigma) \qquad \models (P(\mathit{Type}\ .\mathit{allInstances@pre}())))$
unfolding *OclAllInstances-at-pre-def*
by(*rule state-update-vs-allInstances-generic-ntc*[*OF fst-conv*], *insert assms*)

theorem *state-update-vs-allInstances-at-pre-tc*:
assumes *oid-def*: $\mathit{oid}{\notin}\mathit{dom}\ \sigma'$
and *type-conform*: $\mathit{Type\ Object} \neq \mathit{None}$
and *cp-ctxt*: $\mathit{cp}\ P$
and *const-ctxt*: $\bigwedge X.\ \mathit{const}\ X \Longrightarrow \mathit{const}\ (P\ X)$
shows $(((\llparenthesis \mathit{heap}{=}\sigma'(\mathit{oid}{\mapsto}\mathit{Object}),\mathit{assocs}{=}A\rrparenthesis,\ \sigma) \models (P(\mathit{Type}\ .\mathit{allInstances@pre}()))) =$
$(((\llparenthesis \mathit{heap}{=}\sigma',\ \mathit{assocs}{=}A\rrparenthesis,\ \sigma) \qquad \models (P((\mathit{Type}\ .\mathit{allInstances@pre}())$
$-{>}\mathit{including}(\lambda\ \text{-}.\ \lfloor(\mathit{Type\ Object})\rfloor))))$
unfolding *OclAllInstances-at-pre-def*
by(*rule state-update-vs-allInstances-generic-tc*[*OF fst-conv*], *insert assms*)

@post or @pre **theorem** *StrictRefEq*$_{\mathit{Object}}$*-vs-StrongEq*$''$:
assumes *WFF*: $\mathit{WFF}\ \tau$
and *valid-x*: $\tau \models (\upsilon\ (x :: ({}'\mathfrak{A}{::}\mathit{object},{}'\alpha{::}\mathit{object\ option\ option})\mathit{val}))$
and *valid-y*: $\tau \models (\upsilon\ y)$
and *oid-preserve*: $\bigwedge x.\ x \in \mathit{ran}\ (\mathit{heap}(\mathit{fst}\ \tau)) \vee x \in \mathit{ran}\ (\mathit{heap}(\mathit{snd}\ \tau)) \Longrightarrow$
$\mathit{oid\text{-}of}\ (H\ x) = \mathit{oid\text{-}of}\ x$
and *xy-together*: $\tau \models ((H\ .\mathit{allInstances}()-{>}\mathit{includes}(x)\ \mathit{and}\ H\ .\mathit{allInstances}()-{>}\mathit{includes}(y))\ \mathit{or}$

```
                  (H .allInstances@pre()−>includes(x) and H .allInstances@pre()−>includes(y)))

shows (τ ⊨ (StrictRefEq$_{Object}$ x y)) = (τ ⊨ (x $\triangleq$ y))
proof −
  have at-post-def : ⋀x. τ ⊨ υ x ⟹ τ ⊨ δ (H .allInstances()−>includes(x))
   apply(simp add: OclIncludes-def OclValid-def
              OclAllInstances-at-post-defined[simplified OclValid-def])
  by(subst cp-defined, simp)
  have at-pre-def : ⋀x. τ ⊨ υ x ⟹ τ ⊨ δ (H .allInstances@pre()−>includes(x))
   apply(simp add: OclIncludes-def OclValid-def
              OclAllInstances-at-pre-defined[simplified OclValid-def])
  by(subst cp-defined, simp)
  have F: Rep-Set$_{base}$ (Abs-Set$_{base}$ ⌊⌊Some ‘ (H ‘ ran (heap (fst τ)) − {None})⌋⌋) =
         ⌊⌊Some ‘ (H ‘ ran (heap (fst τ)) − {None})⌋⌋
        by(subst Set$_{base}$.Abs-Set$_{base}$-inverse,simp-all add: bot-option-def)
  have F′: Rep-Set$_{base}$ (Abs-Set$_{base}$ ⌊⌊Some ‘ (H ‘ ran (heap (snd τ)) − {None})⌋⌋) =
         ⌊⌊Some ‘ (H ‘ ran (heap (snd τ)) − {None})⌋⌋
        by(subst Set$_{base}$.Abs-Set$_{base}$-inverse,simp-all add: bot-option-def)
 show ?thesis
  apply(rule StrictRefEq$_{Object}$-vs-StrongEq′[OF WFF valid-x valid-y, where H = Some o H])
  apply(subst oid-preserve[symmetric], simp, simp add: oid-of-option-def)
  apply(insert xy-together,
       subst (asm) foundation11,
       metis at-post-def defined-and-I valid-x valid-y,
       metis at-pre-def defined-and-I valid-x valid-y)
  apply(erule disjE)
 by(drule foundation5,
    simp add: OclAllInstances-at-pre-def OclAllInstances-at-post-def
           OclValid-def OclIncludes-def true-def F F′
           valid-x[simplified OclValid-def] valid-y[simplified OclValid-def] bot-option-def
      split: split-if-asm,
    simp add: comp-def image-def, fastforce)+
qed
```

OclIsNew, OclIsDeleted, OclIsMaintained, OclIsAbsent

```
definition OclIsNew:: ('𝔄, 'α::{null,object})val ⇒ ('𝔄)Boolean   ((-).oclIsNew'('))
where X .oclIsNew() ≡ (λτ . if (δ X) τ = true τ
                        then ⌊⌊oid-of (X τ) ∉ dom(heap(fst τ)) ∧
                              oid-of (X τ) ∈ dom(heap(snd τ))⌋⌋
                        else invalid τ)
```

The following predicates — which are not part of the OCL standard descriptions — complete the goal of oclIsNew by describing where an object belongs.

```
definition OclIsDeleted:: ('𝔄, 'α::{null,object})val ⇒ ('𝔄)Boolean   ((-).oclIsDeleted'('))
where X .oclIsDeleted() ≡ (λτ . if (δ X) τ = true τ
                        then ⌊⌊oid-of (X τ) ∈ dom(heap(fst τ)) ∧
                              oid-of (X τ) ∉ dom(heap(snd τ))⌋⌋
```

else invalid τ)

definition *OclIsMaintained*:: $('\mathfrak{A},\ '\alpha{::}\{null,object\})val \Rightarrow ('\mathfrak{A})Boolean((\text{-}).oclIsMaintained'('))$
where $X\ .oclIsMaintained() \equiv (\lambda\tau\ .\ if\ (\delta\ X)\ \tau = true\ \tau$
$\quad then\ \lfloor\lfloor oid\text{-}of\ (X\ \tau) \in dom(heap(fst\ \tau)) \wedge$
$\quad\quad oid\text{-}of\ (X\ \tau) \in dom(heap(snd\ \tau))\rfloor\rfloor$
$\quad else\ invalid\ \tau)$

definition *OclIsAbsent*:: $('\mathfrak{A},\ '\alpha{::}\{null,object\})val \Rightarrow ('\mathfrak{A})Boolean\quad ((\text{-}).oclIsAbsent'('))$
where $X\ .oclIsAbsent() \equiv (\lambda\tau\ .\ if\ (\delta\ X)\ \tau = true\ \tau$
$\quad then\ \lfloor\lfloor oid\text{-}of\ (X\ \tau) \notin dom(heap(fst\ \tau)) \wedge$
$\quad\quad oid\text{-}of\ (X\ \tau) \notin dom(heap(snd\ \tau))\rfloor\rfloor$
$\quad else\ invalid\ \tau)$

lemma *state-split* : $\tau \models \delta\ X \Longrightarrow$
$\quad \tau \models (X\ .oclIsNew()) \vee \tau \models (X\ .oclIsDeleted()) \vee$
$\quad \tau \models (X\ .oclIsMaintained()) \vee \tau \models (X\ .oclIsAbsent())$
by(*simp add*: *OclIsDeleted-def OclIsNew-def OclIsMaintained-def OclIsAbsent-def*
OclValid-def true-def, *blast*)

lemma *notNew-vs-others* : $\tau \models \delta\ X \Longrightarrow$
$\quad (\neg\ \tau \models (X\ .oclIsNew())) = (\tau \models (X\ .oclIsDeleted()) \vee$
$\quad \tau \models (X\ .oclIsMaintained()) \vee \tau \models (X\ .oclIsAbsent()))$
by(*simp add*: *OclIsDeleted-def OclIsNew-def OclIsMaintained-def OclIsAbsent-def*
OclNot-def OclValid-def true-def, *blast*)

OclIsModifiedOnly

Definition The following predicate—which is not part of the OCL standard—provides a simple, but powerful means to describe framing conditions. For any formal approach, be it animation of OCL contracts, test-case generation or die-hard theorem proving, the specification of the part of a system transition that *does not change* is of primordial importance. The following operator establishes the equality between old and new objects in the state (provided that they exist in both states), with the exception of those objects.

definition *OclIsModifiedOnly* $::('\mathfrak{A}{::}object,'\alpha{::}\{null,object\})Set \Rightarrow\ '\mathfrak{A}\ Boolean$
$\quad (\text{-->}oclIsModifiedOnly'('))$
where $X\text{->}oclIsModifiedOnly() \equiv (\lambda(\sigma,\sigma').$
$\quad let\ X' = (oid\text{-}of\ `\ \lceil\lceil Rep\text{-}Set_{base}(X(\sigma,\sigma'))\rceil\rceil);$
$\quad\quad S = ((dom\ (heap\ \sigma) \cap dom\ (heap\ \sigma')) - X')$
$\quad in\ if\ (\delta\ X)\ (\sigma,\sigma') = true\ (\sigma,\sigma') \wedge (\forall x{\in}\lceil\lceil Rep\text{-}Set_{base}(X(\sigma,\sigma'))\rceil\rceil.\ x \neq null)$
$\quad\quad then\ \lfloor\lfloor\forall\ x \in S.\ (heap\ \sigma)\ x = (heap\ \sigma')\ x\rfloor\rfloor$
$\quad\quad else\ invalid\ (\sigma,\sigma'))$

Execution with Invalid or Null or Null Element as Argument **lemma** $invalid\text{->}oclIsModifiedOnly() = invalid$
by(*simp add*: *OclIsModifiedOnly-def*)

lemma $null\text{->}oclIsModifiedOnly() = invalid$
by(*simp add*: *OclIsModifiedOnly-def*)

lemma
assumes *X-null* : $\tau \models X{-}{>}includes(null)$
shows $\tau \models X{-}{>}oclIsModifiedOnly() \triangleq invalid$
apply(*insert X-null*,
  simp add : *OclIncludes-def OclIsModifiedOnly-def StrongEq-def OclValid-def true-def*)
apply(*case-tac* τ, *simp*)
apply(*simp split*: *split-if-asm*)
by(*simp add*: *null-fun-def*, *blast*)

Context Passing

lemma *cp-OclIsModifiedOnly* : $X{-}{>}oclIsModifiedOnly()\ \tau = (\lambda\text{-}.\ X\ \tau){-}{>}oclIsModifiedOnly()\ \tau$
by(*simp only*: *OclIsModifiedOnly-def*, *case-tac* τ, *simp only*:, *subst cp-defined*, *simp*)

OclSelf

The following predicate—which is not part of the OCL standard—explicitly retrieves in the pre or post state the original OCL expression given as argument.

definition [*simp*]: $OclSelf\ x\ H\ fst\text{-}snd = (\lambda\tau\ .\ if\ (\delta\ x)\ \tau = true\ \tau$
  $then\ if\ oid\text{-}of\ (x\ \tau) \in dom(heap(fst\ \tau)) \wedge oid\text{-}of\ (x\ \tau) \in dom(heap\ (snd\ \tau))$
    $then\ \ H\ \lceil(heap(fst\text{-}snd\ \tau))(oid\text{-}of\ (x\ \tau))\rceil$
    $else\ invalid\ \tau$
  $else\ invalid\ \tau)$

definition *OclSelf-at-pre* :: $('\mathfrak{A}{::}object,'\alpha{::}\{null,object\})val \Rightarrow$
  $('\mathfrak{A} \Rightarrow '\alpha) \Rightarrow$
  $('\mathfrak{A}{::}object,'\alpha{::}\{null,object\})val\ ((\text{-})@pre(\text{-}))$
where $x\ @pre\ H = OclSelf\ x\ H\ fst$

definition *OclSelf-at-post* :: $('\mathfrak{A}{::}object,'\alpha{::}\{null,object\})val \Rightarrow$
  $('\mathfrak{A} \Rightarrow '\alpha) \Rightarrow$
  $('\mathfrak{A}{::}object,'\alpha{::}\{null,object\})val\ ((\text{-})@post(\text{-}))$
where $x\ @post\ H = OclSelf\ x\ H\ snd$

Framing Theorem

lemma *all-oid-diff*:
assumes *def-x* : $\tau \models \delta\ x$
assumes *def-X* : $\tau \models \delta\ X$
assumes *def-X'* : $\bigwedge x.\ x \in \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil \Longrightarrow x \neq null$

defines $P \equiv (\lambda a.\ not\ (StrictRefEq_{Object}\ x\ a))$
shows $(\tau \models X{-}{>}forAll(a|\ P\ a)) = (oid\text{-}of\ (x\ \tau) \notin oid\text{-}of\ `\ \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil)$
proof −
have *P-null-bot*: $\bigwedge b.\ b = null \vee b = \bot \Longrightarrow$
  $\neg\ (\exists x{\in}\lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil.\ P\ (\lambda(\text{-}{::}\ 'a\ state \times\ 'a\ state).\ x)\ \tau = b\ \tau)$
apply(*erule disjE*)
apply(*simp*, *rule ballI*,

simp add: *P-def StrictRefEq*$_{Object}$*-def*, *rename-tac* x',
subst cp-OclNot, *simp*,
subgoal-tac $x\ \tau \neq null \wedge x' \neq null$, *simp*,
simp add: *OclNot-def null-fun-def null-option-def bot-option-def bot-fun-def invalid-def*,
(*metis def-X*$'$ *def-x foundation16*[*THEN iffD1*]
| (*metis bot-fun-def OclValid-def Set-inv-lemma def-X def-x defined-def valid-def*,
metis def-X$'$ *def-x foundation16*[*THEN iffD1*])))+
done

have *not-inj* : $\bigwedge x\ y.\ ((not\ x)\ \tau = (not\ y)\ \tau) = (x\ \tau = y\ \tau)$
by (*metis foundation21 foundation22*)

have *P-false* : $\exists x \in \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil.\ P\ (\lambda\text{-}.\ x)\ \tau = false\ \tau \Longrightarrow$
$oid\text{-}of\ (x\ \tau) \in oid\text{-}of\ `\ \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil$
apply(*erule bexE*, *rename-tac* x')
apply(*simp add*: *P-def*)
apply(*simp only*: *OclNot3*[*symmetric*], *simp only*: *not-inj*)
apply(*simp add*: *StrictRefEq*$_{Object}$*-def split*: *split-if-asm*)
apply(*subgoal-tac* $x\ \tau \neq null \wedge x' \neq null$, *simp*)
apply (*metis* (*mono-tags*) *drop.simps def-x foundation16*[*THEN iffD1*] *true-def*)
by(*simp add*: *invalid-def bot-option-def true-def*)+

have *P-true* : $\forall x \in \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil.\ P\ (\lambda\text{-}.\ x)\ \tau = true\ \tau \Longrightarrow$
$oid\text{-}of\ (x\ \tau) \notin oid\text{-}of\ `\ \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil$
apply(*subgoal-tac* $\forall x' \in \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil.\ oid\text{-}of\ x' \neq oid\text{-}of\ (x\ \tau)$)
apply (*metis imageE*)
apply(*rule ballI*, *drule-tac* $x = x'$ **in** *ballE*) **prefer** *3* **apply** *assumption*
apply(*simp add*: *P-def*)
apply(*simp only*: *OclNot4*[*symmetric*], *simp only*: *not-inj*)
apply(*simp add*: *StrictRefEq*$_{Object}$*-def false-def split*: *split-if-asm*)
apply(*subgoal-tac* $x\ \tau \neq null \wedge x' \neq null$, *simp*)
apply (*metis def-X*$'$ *def-x foundation16*[*THEN iffD1*])
by(*simp add*: *invalid-def bot-option-def false-def*)+

have *bool-split* : $\forall x \in \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil.\ P\ (\lambda\text{-}.\ x)\ \tau \neq null\ \tau \Longrightarrow$
$\forall x \in \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil.\ P\ (\lambda\text{-}.\ x)\ \tau \neq \bot\ \tau \Longrightarrow$
$\forall x \in \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil.\ P\ (\lambda\text{-}.\ x)\ \tau \neq false\ \tau \Longrightarrow$
$\forall x \in \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil.\ P\ (\lambda\text{-}.\ x)\ \tau = true\ \tau$
apply(*rule ballI*)
apply(*drule-tac* $x = x$ **in** *ballE*) **prefer** *3* **apply** *assumption*
apply(*drule-tac* $x = x$ **in** *ballE*) **prefer** *3* **apply** *assumption*
apply(*drule-tac* $x = x$ **in** *ballE*) **prefer** *3* **apply** *assumption*
apply (*metis* (*full-types*) *bot-fun-def OclNot4 OclValid-def foundation16*
foundation9 not-inj null-fun-def)
by(*fast*+)

show *?thesis*

```
  apply(subst OclForall-rep-set-true[OF def-X], simp add: OclValid-def)
  apply(rule iffI, simp add: P-true)
 by (metis P-false P-null-bot bool-split)
qed

theorem framing:
      assumes modifiesclause:τ ⊨ (X−>excluding(x))−>oclIsModifiedOnly()
      and oid-is-typerepr : τ ⊨ X−>forAll(a| not (StrictRefEq_Object x a))
      shows τ ⊨ (x @pre P  ≜  (x @post P))
 apply(case-tac τ ⊨ δ x)
 proof − show τ ⊨ δ x ⟹ ?thesis proof − assume def-x : τ ⊨ δ x show ?thesis proof −

 have def-X : τ ⊨ δ X
  apply(insert oid-is-typerepr, simp add: OclForall-def OclValid-def split: split-if-asm)
 by(simp add: bot-option-def true-def)

 have def-X′ : ⋀x. x ∈ ⌈⌈Rep-Set_base (X τ)⌉⌉ ⟹ x ≠ null
  apply(insert modifiesclause, simp add: OclIsModifiedOnly-def OclValid-def split: split-if-asm)
  apply(case-tac τ, simp split: split-if-asm)
   apply(simp add: OclExcluding-def split: split-if-asm)
    apply(subst (asm) (2) Abs-Set_base-inverse)
     apply(simp, (rule disjI2)+)
     apply (metis (hide-lams, mono-tags) Diff-iff Set-inv-lemma def-X)
    apply(simp)
    apply(erule ballE[where P = λx. x ≠ null]) apply(assumption)
    apply(simp)
    apply (metis (hide-lams, no-types) def-x  foundation16[THEN iffD1])
   apply (metis (hide-lams, no-types) OclValid-def def-X def-x foundation20
                               OclExcluding-valid-args-valid OclExcluding-valid-args-valid′′)
 by(simp add: invalid-def bot-option-def)

 have oid-is-typerepr : oid-of (x τ) ∉ oid-of ‘ ⌈⌈Rep-Set_base (X τ)⌉⌉
 by(rule all-oid-diff[THEN iffD1, OF def-x def-X def-X′ oid-is-typerepr])

 show ?thesis
  apply(simp add: StrongEq-def OclValid-def true-def OclSelf-at-pre-def OclSelf-at-post-def
              def-x[simplified OclValid-def])
  apply(rule conjI, rule impI)
   apply(rule-tac f = λx. P ⌈x⌉ in arg-cong)
   apply(insert modifiesclause[simplified OclIsModifiedOnly-def OclValid-def])
   apply(case-tac τ, rename-tac σ σ′, simp split: split-if-asm)
    apply(subst (asm) (2) OclExcluding-def)
    apply(drule foundation5[simplified OclValid-def true-def], simp)
    apply(subst (asm) Abs-Set_base-inverse, simp)
     apply(rule disjI2)+
     apply (metis (hide-lams, no-types) DiffD1 OclValid-def Set-inv-lemma def-x
                               foundation16 foundation18′)
    apply(simp)
   apply(erule-tac x = oid-of (x (σ, σ′)) in ballE) apply simp+
   apply (metis (hide-lams, no-types)
             DiffD1 image-iff image-insert insert-Diff-single insert-absorb oid-is-typerepr)
  apply(simp add: invalid-def bot-option-def)+
 by blast
 qed qed
 apply-end(simp add: OclSelf-at-post-def OclSelf-at-pre-def OclValid-def StrongEq-def true-def)+
qed
```

As corollary, the framing property can be expressed with only the strong equality as comparison operator.

```
theorem framing′:
 assumes wff : WFF τ
 assumes modifiesclause:τ ⊨ (X−>excluding(x))−>oclIsModifiedOnly()
 and oid-is-typerepr : τ ⊨ X−>forAll(a| not (x ≜ a))
 and oid-preserve: ⋀x. x ∈ ran (heap(fst τ)) ∨ x ∈ ran (heap(snd τ)) ⟹
                oid-of (H x) = oid-of x
 and xy-together:
 τ ⊨ X−>forAll(y | (H .allInstances()−>includes(x) and H .allInstances()−>includes(y)) or
              (H .allInstances@pre()−>includes(x) and H .allInstances@pre()−>includes(y)))
 shows τ ⊨ (x @pre P  ≜  (x @post P))
proof −
 have def-X : τ ⊨ δ X
  apply(insert oid-is-typerepr, simp add: OclForall-def OclValid-def split: split-if-asm)
 by(simp add: bot-option-def true-def)
 show ?thesis
  apply(case-tac τ ⊨ δ x, drule foundation20)
   apply(rule framing[OF modifiesclause])
   apply(rule OclForall-cong′[OF - oid-is-typerepr xy-together], rename-tac y)
   apply(cut-tac Set-inv-lemma′[OF def-X]) prefer 2 apply assumption
   apply(rule OclNot-contrapos-nn, simp add: StrictRefEq_Object-def)
    apply(simp add: OclValid-def, subst cp-defined, simp,
         assumption)
   apply(rule StrictRefEq_Object-vs-StrongEq′′[THEN iffD1, OF wff - - oid-preserve], assumption+)
 by(simp add: OclSelf-at-post-def OclSelf-at-pre-def OclValid-def StrongEq-def true-def)+
qed
```

Miscellaneous

```
lemma pre-post-new: τ ⊨ (x .oclIsNew()) ⟹ ¬ (τ ⊨ υ(x @pre H1)) ∧ ¬ (τ ⊨ υ(x @post H2))
by(simp add: OclIsNew-def OclSelf-at-pre-def OclSelf-at-post-def
          OclValid-def StrongEq-def true-def false-def
          bot-option-def invalid-def bot-fun-def valid-def
     split: split-if-asm)

lemma pre-post-old: τ ⊨ (x .oclIsDeleted()) ⟹ ¬ (τ ⊨ υ(x @pre H1)) ∧ ¬ (τ ⊨ υ(x @post H2))
by(simp add: OclIsDeleted-def OclSelf-at-pre-def OclSelf-at-post-def
          OclValid-def StrongEq-def true-def false-def
          bot-option-def invalid-def bot-fun-def valid-def
```

split: *split-if-asm*)

lemma *pre-post-absent*: $\tau \models (x\ .oclIsAbsent()) \Longrightarrow \neg\ (\tau \models \upsilon(x\ @pre\ H1)) \land \neg\ (\tau \models \upsilon(x\ @post\ H2))$
by(*simp add*: *OclIsAbsent-def OclSelf-at-pre-def OclSelf-at-post-def*
OclValid-def StrongEq-def true-def false-def
bot-option-def invalid-def bot-fun-def valid-def
split: *split-if-asm*)

lemma *pre-post-maintained*: $(\tau \models \upsilon(x\ @pre\ H1) \lor \tau \models \upsilon(x\ @post\ H2)) \Longrightarrow \tau \models (x\ .oclIsMaintained())$
by(*simp add*: *OclIsMaintained-def OclSelf-at-pre-def OclSelf-at-post-def*
OclValid-def StrongEq-def true-def false-def
bot-option-def invalid-def bot-fun-def valid-def
split: *split-if-asm*)

lemma *pre-post-maintained'*:
$\tau \models (x\ .oclIsMaintained()) \Longrightarrow (\tau \models \upsilon(x\ @pre\ (Some\ o\ H1)) \land \tau \models \upsilon(x\ @post\ (Some\ o\ H2)))$
by(*simp add*: *OclIsMaintained-def OclSelf-at-pre-def OclSelf-at-post-def*
OclValid-def StrongEq-def true-def false-def
bot-option-def invalid-def bot-fun-def valid-def
split: *split-if-asm*)

lemma *framing-same-state*: $(\sigma, \sigma) \models (x\ @pre\ H \triangleq (x\ @post\ H))$
by(*simp add*: *OclSelf-at-pre-def OclSelf-at-post-def OclValid-def StrongEq-def*)

end

theory *UML-Contracts*
imports *UML-State*
begin

Modeling of an operation contract for an operation with 2 arguments, (so depending on three parameters if one takes "self" into account).

locale *contract-scheme* =
fixes $f\text{-}\upsilon$
fixes *f-lam*
fixes $f\ ::\ ('\mathfrak{A},'\alpha 0::null)val \Rightarrow$
$'b \Rightarrow$
$('\mathfrak{A},'res::null)val$
fixes *PRE*
fixes *POST*
assumes *def-scheme'*: $f\ self\ x \equiv (\lambda\ \tau.\ if\ (\tau \models (\delta\ self)) \land f\text{-}\upsilon\ x\ \tau$
$then\ SOME\ res.\ (\tau \models PRE\ self\ x) \land$
$(\tau \models POST\ self\ x\ (\lambda\ \text{-}.\ res))$
$else\ invalid\ \tau)$
assumes *all-post'*: $\forall\ \sigma\ \sigma'\ \sigma''.\ ((\sigma,\sigma') \models PRE\ self\ x) = ((\sigma,\sigma'') \models PRE\ self\ x)$

```
  assumes cp_PRE': PRE (self) x τ = PRE (λ -. self τ) (f-lam x τ) τ

  assumes cp_POST':POST (self) x (res) τ = POST (λ -. self τ) (f-lam x τ) (λ -. res τ) τ
  assumes f-υ-val: ⋀a1. f-υ (f-lam a1 τ) τ = f-υ a1 τ
begin
  lemma strict0 [simp]: f invalid X = invalid
  by(rule ext, rename-tac τ, simp add: def-scheme')

  lemma nullstrict0[simp]: f null X = invalid
  by(rule ext, rename-tac τ, simp add: def-scheme')

  lemma cp0 : f self a1 τ = f (λ -. self τ) (f-lam a1 τ) τ
  proof −
    have A: (τ ⊨ δ (λ-. self τ)) = (τ ⊨ δ self) by(simp add: OclValid-def cp-defined[symmetric])
    have B: f-υ (f-lam a1 τ) τ = f-υ a1 τ by (rule f-υ-val)
    have D: (τ ⊨ PRE (λ-. self τ) (f-lam a1 τ)) = ( τ ⊨ PRE self a1 )
                                    by(simp add: OclValid-def cp_PRE'[symmetric])
    show ?thesis
      apply(auto simp: def-scheme' A B D)
      apply(simp add: OclValid-def)
      by(subst cp_POST', simp)
    qed

  theorem unfold' :
    assumes context-ok:   cp E
    and args-def-or-valid: (τ ⊨ δ self) ∧ f-υ a1 τ
    and pre-satisfied:    τ ⊨ PRE self a1
    and post-satisfiable:  ∃res. (τ ⊨ POST self a1 (λ -. res))
    and sat-for-sols-post: (⋀res. τ ⊨ POST self a1 (λ -. res) ⟹ τ ⊨ E (λ -. res))
    shows             τ ⊨ E(f self a1)
  proof −
    have cp0: ⋀ X τ. E X τ = E (λ-. X τ) τ by(insert context-ok[simplified cp-def], auto)
    show ?thesis
      apply(simp add: OclValid-def, subst cp0, fold OclValid-def)
      apply(simp add:def-scheme' args-def-or-valid pre-satisfied)
      apply(insert post-satisfiable, elim exE)
      apply(rule Hilbert-Choice.someI2, assumption)
      by(rule sat-for-sols-post, simp)
  qed

  lemma unfold2' :
    assumes context-ok:    cp E
    and args-def-or-valid:  (τ ⊨ δ self) ∧ (f-υ a1 τ)
    and pre-satisfied:     τ ⊨ PRE self a1
    and postsplit-satisfied: τ ⊨ POST' self a1
    and post-decomposable  : ⋀ res. (POST self a1 res) =
                          ((POST' self a1)  and (res ≜ (BODY self a1)))
    shows (τ ⊨ E(f self a1)) = (τ ⊨ E(BODY self a1))
  proof −
    have cp0: ⋀ X τ. E X τ = E (λ-. X τ) τ by(insert context-ok[simplified cp-def], auto)
    show ?thesis
      apply(simp add: OclValid-def, subst cp0, fold OclValid-def)
      apply(simp add:def-scheme′ args-def-or-valid pre-satisfied
                post-decomposable postsplit-satisfied foundation27)
      apply(subst some-equality)
      apply(simp add: OclValid-def StrongEq-def true-def)+
      by(subst (2) cp0, rule refl)
  qed
end


locale contract0 =
  fixes f  :: (′𝔄,′α0::null)val ⇒
            (′𝔄,′res::null)val
  fixes PRE
  fixes POST
  assumes def-scheme: f self ≡  (λ τ. if (τ ⊨ (δ self))
                                then SOME res. (τ ⊨ PRE self) ∧
                                          (τ ⊨ POST self (λ -. res))
                                else invalid τ)
  assumes all-post: ∀ σ σ′ σ″. ((σ,σ′) ⊨ PRE self) = ((σ,σ″) ⊨ PRE self)

  assumes cp_PRE: PRE (self)  τ = PRE (λ -. self τ) τ

  assumes cp_POST:POST (self) (res) τ = POST (λ -. self τ) (λ -. res τ) τ

sublocale contract0 < contract-scheme λ- -. True λx -. x λx -. f x λx -. PRE x λx -. POST x
 apply(unfold-locales)
   apply(simp add: def-scheme, rule all-post, rule cp_PRE, rule cp_POST)
by simp

context contract0
begin
  lemma cp-pre: cp self′ ⟹  cp (λX. PRE (self′ X)  )
  by(rule-tac f=PRE in cpI1, auto intro: cp_PRE)

  lemma cp-post: cp self′ ⟹ cp res′  ⟹ cp (λX. POST (self′ X) (res′ X))
  by(rule-tac f=POST in cpI2, auto intro: cp_POST)

  lemma cp [simp]:  cp self′ ⟹  cp res′ ⟹ cp (λX. f (self′ X) )
    by(rule-tac f=f in cpI1, auto intro:cp0)

  lemmas unfold = unfold′[simplified]

  lemma unfold2 :
```

assumes $cp\ E$
and $(\tau \models \delta\ self)$
and $\tau \models PRE\ self$
and $\tau \models POST'\ self$
and $\bigwedge res.\ (POST\ self\ res) =$
$((POST'\ self)\ \ and\ (res \triangleq (BODY\ self)))$
shows $(\tau \models E(f\ self)) = (\tau \models E(BODY\ self))$
apply$(rule\ unfold2'[simplified])$
by$((rule\ assms)+)$

end

locale $contract1 =$
fixes $f\ \ ::\ ('\mathfrak{A},'\alpha 0::null)val \Rightarrow$
$('\mathfrak{A},'\alpha 1::null)val \Rightarrow$
$('\mathfrak{A},'res::null)val$
fixes PRE
fixes $POST$
assumes $def\text{-}scheme: f\ self\ a1 \equiv$
$(\lambda\ \tau.\ if\ (\tau \models (\delta\ self)) \wedge\ (\tau \models \upsilon\ a1)$
$then\ SOME\ res.\ (\tau \models PRE\ self\ a1) \wedge$
$(\tau \models POST\ self\ a1\ (\lambda\ \text{-}.\ res))$
$else\ invalid\ \tau)$
assumes $all\text{-}post: \forall\ \sigma\ \sigma'\ \sigma''.\ \ ((\sigma,\sigma') \models PRE\ self\ a1) = ((\sigma,\sigma'') \models PRE\ self\ a1)$

assumes $cp_{PRE}: PRE\ (self)\ (a1)\ \ \tau = PRE\ (\lambda\ \text{-}.\ self\ \tau)\ (\lambda\ \text{-}.\ a1\ \tau)\ \tau$

assumes $cp_{POST}{:}POST\ (self)\ (a1)\ (res)\ \tau = POST\ (\lambda\ \text{-}.\ self\ \tau)(\lambda\ \text{-}.\ a1\ \tau)\ (\lambda\ \text{-}.\ res\ \tau)\ \tau$

sublocale $contract1 < contract\text{-}scheme\ \lambda a1\ \tau.\ (\tau \models \upsilon\ a1)\ \lambda a1\ \tau.\ (\lambda\ \text{-}.\ a1\ \tau)$
apply$(unfold\text{-}locales)$
apply$(rule\ def\text{-}scheme,\ rule\ all\text{-}post,\ rule\ cp_{PRE},\ rule\ cp_{POST})$
by$(simp\ add{:}\ OclValid\text{-}def\ cp\text{-}valid[symmetric])$

context $contract1$
begin
lemma $strict1[simp]{:}\ f\ self\ invalid = invalid$
by$(rule\ ext,\ rename\text{-}tac\ \tau,\ simp\ add{:}\ def\text{-}scheme)$

lemma $cp\text{-}pre{:}\ cp\ self' \Longrightarrow cp\ a1' \Longrightarrow\ cp\ (\lambda X.\ PRE\ (self'\ X)\ (a1'\ X)\ \ )$
by$(rule\text{-}tac\ f{=}PRE$ **in** $cpI2,\ auto\ intro{:}\ cp_{PRE})$

lemma $cp\text{-}post{:}\ cp\ self' \Longrightarrow cp\ a1' \Longrightarrow cp\ res'$
$\Longrightarrow cp\ (\lambda X.\ POST\ (self'\ X)\ (a1'\ X)\ (res'\ X))$
by$(rule\text{-}tac\ f{=}POST$ **in** $cpI3,\ auto\ intro{:}\ cp_{POST})$

lemma $cp\ [simp]{:}\ \ cp\ self' \Longrightarrow cp\ a1' \Longrightarrow\ \ cp\ res' \Longrightarrow cp\ (\lambda X.\ f\ (self'\ X)\ (a1'\ X))$
by$(rule\text{-}tac\ f{=}f$ **in** $cpI2,\ auto\ intro{:}cp0)$

```
  lemmas unfold = unfold′
  lemmas unfold2 = unfold2′
end

locale contract2 =
  fixes f  :: (′𝔄,′α0::null)val ⇒
              (′𝔄,′α1::null)val ⇒ (′𝔄,′α2::null)val ⇒
              (′𝔄,′res::null)val
  fixes PRE
  fixes POST
  assumes def-scheme: f self a1 a2 ≡
                   (λ τ. if (τ ⊨ (δ self)) ∧ (τ ⊨ υ a1) ∧ (τ ⊨ υ a2)
                        then SOME res. (τ ⊨ PRE self a1 a2) ∧
                                  (τ ⊨ POST self a1 a2 (λ -. res))
                        else invalid τ)
  assumes all-post: ∀ σ σ′ σ′′. ((σ,σ′) ⊨ PRE self a1 a2) = ((σ,σ′′) ⊨ PRE self a1 a2)

  assumes cp_PRE: PRE (self) (a1) (a2) τ = PRE (λ -. self τ) (λ -. a1 τ) (λ -. a2 τ) τ

  assumes cp_POST:⋀res. POST (self) (a1) (a2) (res) τ =
                POST (λ -. self τ)(λ -. a1 τ)(λ -. a2 τ) (λ -. res τ) τ


sublocale contract2 < contract-scheme λ(a1,a2) τ. (τ ⊨ υ a1) ∧ (τ ⊨ υ a2)
                               λ(a1,a2) τ. (λ -.a1 τ, λ -.a2 τ)
                               (λx (a,b). f x a b)
                               (λx (a,b). PRE x a b)
                               (λx (a,b). POST x a b)
 apply(unfold-locales)
    apply(auto simp add: def-scheme)
      apply (metis all-post, metis all-post)
     apply(subst cp_PRE, simp)
    apply(subst cp_POST, simp)
by(simp-all add: OclValid-def cp-valid[symmetric])

context contract2
begin
  lemma strict0[simp] : f invalid X Y = invalid
  by(insert strict0[of (X,Y)], simp)

  lemma nullstrict0[simp]: f null X Y = invalid
  by(insert nullstrict0[of (X,Y)], simp)

  lemma strict1[simp]: f self invalid Y = invalid
  by(rule ext, rename-tac τ, simp add: def-scheme)

  lemma strict2[simp]: f self X invalid = invalid
  by(rule ext, rename-tac τ, simp add: def-scheme)

  lemma cp-pre: cp self′ ⟹ cp a1′ ⟹ cp a2′ ⟹ cp (λX. PRE (self′ X) (a1′ X) (a2′ X) )
  by(rule-tac f=PRE in cpI3, auto intro: cp_PRE)

  lemma cp-post: cp self′ ⟹ cp a1′ ⟹ cp a2′ ⟹ cp res′
           ⟹ cp (λX. POST (self′ X) (a1′ X) (a2′ X) (res′ X))
  by(rule-tac f=POST in cpI4, auto intro: cp_POST)

  lemma cp0 : f self a1 a2 τ = f (λ -. self τ) (λ -. a1 τ) (λ -. a2 τ) τ
  by (rule cp0[of - (a1,a2), simplified])

  lemma cp [simp]:  cp self′ ⟹ cp a1′ ⟹ cp a2′ ⟹ cp res′
              ⟹ cp (λX. f (self′ X) (a1′ X) (a2′ X))
    by(rule-tac f=f in cpI3, auto intro:cp0)

  theorem unfold :
    assumes          cp E
    and             (τ ⊨ δ self) ∧ (τ ⊨ υ a1) ∧  (τ ⊨ υ a2)
    and             τ ⊨ PRE self a1 a2
    and             ∃res. (τ ⊨ POST self a1 a2 (λ -. res))
    and             (⋀res. τ ⊨ POST self a1 a2 (λ -. res)  ⟹ τ ⊨ E (λ -. res))
    shows            τ ⊨ E(f self a1 a2)
    apply(rule unfold′[of - - - (a1, a2), simplified])
    by((rule assms)+)

  lemma unfold2 :
    assumes          cp E
    and             (τ ⊨ δ self) ∧ (τ ⊨ υ a1) ∧  (τ ⊨ υ a2)
    and             τ ⊨ PRE self a1 a2
    and             τ ⊨ POST′ self a1 a2
    and             ⋀ res. (POST self a1 a2 res) =
                      ((POST′ self a1 a2)  and (res ≜ (BODY self a1 a2)))
    shows (τ ⊨ E(f self a1 a2)) = (τ ⊨ E(BODY self a1 a2))
    apply(rule unfold2′[of - - - (a1, a2), simplified])
    by((rule assms)+)
end

end
```

```
theory UML-Tools
imports UML-Logic
begin
```

```
lemmas substs1 = StrongEq-L-subst2-rev
            foundation15[THEN iffD2, THEN StrongEq-L-subst2-rev]
            foundation7'[THEN iffD2, THEN foundation15[THEN iffD2,
                           THEN StrongEq-L-subst2-rev]]
            foundation14[THEN iffD2, THEN StrongEq-L-subst2-rev]
            foundation13[THEN iffD2, THEN StrongEq-L-subst2-rev]

lemmas substs2 = StrongEq-L-subst3-rev
            foundation15[THEN iffD2, THEN StrongEq-L-subst3-rev]
            foundation7'[THEN iffD2, THEN foundation15[THEN iffD2,
                           THEN StrongEq-L-subst3-rev]]
            foundation14[THEN iffD2, THEN StrongEq-L-subst3-rev]
            foundation13[THEN iffD2, THEN StrongEq-L-subst3-rev]

lemmas substs4 = StrongEq-L-subst4-rev
            foundation15[THEN iffD2, THEN StrongEq-L-subst4-rev]
            foundation7'[THEN iffD2, THEN foundation15[THEN iffD2,
                           THEN StrongEq-L-subst4-rev]]
            foundation14[THEN iffD2, THEN StrongEq-L-subst4-rev]
            foundation13[THEN iffD2, THEN StrongEq-L-subst4-rev]


lemmas substs = substs1 substs2 substs4 [THEN iffD2] substs4
thm substs
ML⟨⟨
fun ocl-subst-asm-tac ctxt  = FIRST'(map (fn C => (etac C) THEN' (simp-tac ctxt))
                                @{thms substs})

val ocl-subst-asm = fn ctxt => SIMPLE-METHOD (ocl-subst-asm-tac ctxt 1);

val - = Theory.setup
          (Method.setup (Binding.name ocl-subst-asm)
          (Scan.succeed (ocl-subst-asm))
          ocl substition step)

⟩⟩

lemma test1 : τ ⊨ A ⟹ τ ⊨ (A and B ≜ B)
apply(tactic ocl-subst-asm-tac @{context} 1)
apply(simp)
done

lemma test2 : τ ⊨ A ⟹ τ ⊨ (A and B ≜ B)
by(ocl-subst-asm, simp)

lemma test3 : τ ⊨ A ⟹ τ ⊨ (A and A)
by(ocl-subst-asm, simp)
```

lemma *test4* : $\tau \models \mathit{not}\ A \Longrightarrow \tau \models (A\ \mathit{and}\ B \triangleq \mathit{false})$
by(*ocl-subst-asm*, *simp*)

lemma *test5* : $\tau \models (A \triangleq \mathit{null}) \Longrightarrow \tau \models (B \triangleq \mathit{null}) \Longrightarrow \neg\ (\tau \models (A\ \mathit{and}\ B))$
by(*ocl-subst-asm*,*ocl-subst-asm*,*simp*)

lemma *test6* : $\tau \models \mathit{not}\ A \Longrightarrow \neg\ (\tau \models (A\ \mathit{and}\ B))$
by(*ocl-subst-asm*, *simp*)

lemma *test7* : $\neg\ (\tau \models (\upsilon\ A)) \Longrightarrow \tau \models (\mathit{not}\ B) \Longrightarrow \neg\ (\tau \models (A\ \mathit{and}\ B))$
by(*ocl-subst-asm*,*ocl-subst-asm*,*simp*)

lemma *X*: $\neg\ (\tau \models (\mathit{invalid}\ \mathit{and}\ B))$
apply(*insert foundation8*[*of* τ *B*], *elim disjE*,
    simp add:*defined-bool-split*, *elim disjE*)
apply(*ocl-subst-asm*, *simp*)
apply(*ocl-subst-asm*, *simp*)
apply(*ocl-subst-asm*, *simp*)
apply(*ocl-subst-asm*, *simp*)
done

lemma X': $\neg\ (\tau \models (\mathit{invalid}\ \mathit{and}\ B))$
by(*simp add*:*foundation10*$'$)
lemma *Y*: $\neg\ (\tau \models (\mathit{null}\ \mathit{and}\ B))$
by(*simp add*: *foundation10*$'$)
lemma *Z*: $\neg\ (\tau \models (\mathit{false}\ \mathit{and}\ B))$
by(*simp add*: *foundation10*$'$)
lemma Z': $(\tau \models (\mathit{true}\ \mathit{and}\ B)) = (\tau \models B)$
by(*simp*)

end

theory *UML-Main*
imports *UML-Contracts UML-Tools*

begin

end

B.4. Example I : The Employee Analysis Model (UML)

theory
Analysis-UML
imports
../../../src/UML-Main
begin

B.4.1. Introduction

For certain concepts like classes and class-types, only a generic definition for its resulting semantics can be given. Generic means, there is a function outside HOL that "compiles" a concrete, closed-world class diagram into a "theory" of this data model, consisting of a bunch of definitions for classes, accessors, method, casts, and tests for actual types, as well as proofs for the fundamental properties of these operations in this concrete data model.

Such generic function or "compiler" can be implemented in Isabelle on the ML level. This has been done, for a semantics following the open-world assumption, for UML 2.0 in [4, 6]. In this paper, we follow another approach for UML 2.4: we define the concepts of the compilation informally, and present a concrete example which is verified in Isabelle/HOL.

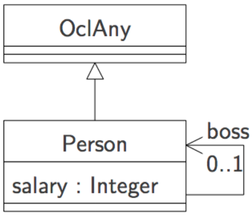


Figure B.1.: A simple UML class model drawn from Figure 7.3, page 20 of [28].

Outlining the Example

We are presenting here an “analysis-model” of the (slightly modified) example Figure 7.3, page 20 of the OCL standard [28]. Here, analysis model means that associations were really represented as relation on objects on the state—as is intended by the standard—rather by pointers between objects as is done in our “design model” (see Section B.5). To be precise, this theory contains the formalization of the data-part covered by the UML class model (see Figure B.1):

This means that the association (attached to the association class `EmployeeRanking`) with the association ends `boss` and `employees` is implemented by the attribute `boss` and the operation `employees` (to be discussed in the OCL part captured by the subsequent theory).

B.4.2. Example Data-Universe and its Infrastructure

Ideally, the following is generated automatically from a UML class model.

Our data universe consists in the concrete class diagram just of node’s, and implicitly of the class object. Each class implies the existence of a class type defined for the corresponding object representations as follows:

datatype $type_{Person} = mk_{Person}\ oid$
$\quad int\ option$

datatype $type_{OclAny} = mk_{OclAny}\ oid$
$\quad (int\ option)\ option$

Now, we construct a concrete “universe of OclAny types” by injection into a sum type containing the class types. This type of OclAny will be used as instance for all respective type-variables.

datatype $\mathfrak{A} = in_{Person}\ type_{Person} \mid in_{OclAny}\ type_{OclAny}$

Having fixed the object universe, we can introduce type synonyms that exactly correspond to OCL types. Again, we exploit that our representation of OCL is a “shallow embedding” with a one-to-one correspondance of OCL-types to types of the meta-language HOL.

type-synonym $Boolean \quad = \mathfrak{A}\ Boolean$

```
type-synonym Integer    = 𝔄 Integer
type-synonym Void       = 𝔄 Void
type-synonym OclAny     = (𝔄, type_OclAny option option) val
type-synonym Person     = (𝔄, type_Person option option) val
type-synonym Set-Integer = (𝔄, int option option) Set
type-synonym Set-Person  = (𝔄, type_Person option option) Set
```

Just a little check:

```
typ Boolean
```

To reuse key-elements of the library like referential equality, we have to show that the object universe belongs to the type class “oclany,” i. e., each class type has to provide a function *oid-of* yielding the object id (oid) of the object.

```
instantiation type_Person :: object
begin
  definition oid-of-type_Person-def: oid-of x = (case x of mk_Person oid - ⇒ oid)
  instance ..
end

instantiation type_OclAny :: object
begin
  definition oid-of-type_OclAny-def: oid-of x = (case x of mk_OclAny oid - ⇒ oid)
  instance ..
end

instantiation 𝔄 :: object
begin
  definition oid-of-𝔄-def: oid-of x = (case x of
                                in_Person person ⇒ oid-of person
                              | in_OclAny oclany ⇒ oid-of oclany)
  instance ..
end
```

B.4.3. Instantiation of the Generic Strict Equality

We instantiate the referential equality on *Person* and *OclAny*

```
defs(overloaded)  StrictRefEq_Object-Person  : (x::Person) ≐ y  ≡ StrictRefEq_Object x y
defs(overloaded)  StrictRefEq_Object-OclAny  : (x::OclAny) ≐ y  ≡ StrictRefEq_Object x y

lemmas
  cp-StrictRefEq_Object[of x::Person y::Person τ,
                simplified StrictRefEq_Object-Person[symmetric]]
  cp-intro(9)        [of P::Person ⇒PersonQ::Person ⇒Person,
                simplified StrictRefEq_Object-Person[symmetric] ]
  StrictRefEq_Object-def     [of x::Person y::Person,
                simplified StrictRefEq_Object-Person[symmetric]]
  StrictRefEq_Object-defargs  [of - x::Person y::Person,
```

$\quad\quad\quad \mathit{simplified}\ \mathit{StrictRefEq}_{Object\text{-}Person}[\mathit{symmetric}]]$
$\mathit{StrictRefEq}_{Object}\text{-}\mathit{strict1}$
$\quad\quad [\mathit{of}\ x{::}\mathit{Person},$
$\quad\quad\quad \mathit{simplified}\ \mathit{StrictRefEq}_{Object\text{-}Person}[\mathit{symmetric}]]$
$\mathit{StrictRefEq}_{Object}\text{-}\mathit{strict2}$
$\quad\quad [\mathit{of}\ x{::}\mathit{Person},$
$\quad\quad\quad \mathit{simplified}\ \mathit{StrictRefEq}_{Object\text{-}Person}[\mathit{symmetric}]]$

For each Class C, we will have a casting operation `.oclAsType(`C`)`, a test on the actual type `.oclIsTypeOf(`C`)` as well as its relaxed form `.oclIsKindOf(`C`)` (corresponding exactly to Java's `instanceof`-operator.

Thus, since we have two class-types in our concrete class hierarchy, we have two operations to declare and to provide two overloading definitions for the two static types.

B.4.4. OclAsType

Definition

consts $\mathit{OclAsType}_{OclAny} :: {}'\alpha \Rightarrow \mathit{OclAny}\ ((\text{-})\ .\mathit{oclAsType}'(\mathit{OclAny}'))$
consts $\mathit{OclAsType}_{Person} :: {}'\alpha \Rightarrow \mathit{Person}\ ((\text{-})\ .\mathit{oclAsType}'(\mathit{Person}'))$

definition $\mathit{OclAsType}_{OclAny}\text{-}\mathfrak{A} = (\lambda u.\ \lfloor \mathit{case}\ u\ \mathit{of}\ \mathit{in}_{OclAny}\ a \Rightarrow a$
$\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathit{in}_{Person}\ (\mathit{mk}_{Person}\ \mathit{oid}\ a) \Rightarrow \mathit{mk}_{OclAny}\ \mathit{oid}\ \lfloor a \rfloor \rfloor)$

lemma $\mathit{OclAsType}_{OclAny}\text{-}\mathfrak{A}\text{-}\mathit{some}$: $\mathit{OclAsType}_{OclAny}\text{-}\mathfrak{A}\ x \neq \mathit{None}$
by$(\mathit{simp}\ \mathit{add}\text{: } \mathit{OclAsType}_{OclAny}\text{-}\mathfrak{A}\text{-}\mathit{def})$

defs (**overloaded**) $\mathit{OclAsType}_{OclAny}\text{-}\mathit{OclAny}$:
$\quad (X{::}\mathit{OclAny})\ .\mathit{oclAsType}(\mathit{OclAny}) \equiv X$

defs (**overloaded**) $\mathit{OclAsType}_{OclAny}\text{-}\mathit{Person}$:
$\quad (X{::}\mathit{Person})\ .\mathit{oclAsType}(\mathit{OclAny}) \equiv$
$\quad\quad (\lambda\tau.\ \mathit{case}\ X\ \tau\ \mathit{of}$
$\quad\quad\quad\quad \bot\ \ \Rightarrow \mathit{invalid}\ \tau$
$\quad\quad\quad \mid \lfloor\bot\rfloor \Rightarrow \mathit{null}\ \tau$
$\quad\quad\quad \mid \lfloor\lfloor \mathit{mk}_{Person}\ \mathit{oid}\ a\ \rfloor\rfloor \Rightarrow\ \lfloor\lfloor\ (\mathit{mk}_{OclAny}\ \mathit{oid}\ \lfloor a\rfloor)\ \rfloor\rfloor)$

definition $\mathit{OclAsType}_{Person}\text{-}\mathfrak{A} = (\lambda u.\ \mathit{case}\ u\ \mathit{of}\ \mathit{in}_{Person}\ p \Rightarrow \lfloor p\rfloor$
$\quad\quad\quad\quad\quad\quad \mid \mathit{in}_{OclAny}\ (\mathit{mk}_{OclAny}\ \mathit{oid}\ \lfloor a\rfloor) \Rightarrow \lfloor \mathit{mk}_{Person}\ \mathit{oid}\ a\rfloor$
$\quad\quad\quad\quad\quad\quad \mid \text{-} \Rightarrow \mathit{None})$

defs (**overloaded**) $\mathit{OclAsType}_{Person}\text{-}\mathit{OclAny}$:
$\quad (X{::}\mathit{OclAny})\ .\mathit{oclAsType}(\mathit{Person}) \equiv$
$\quad\quad (\lambda\tau.\ \mathit{case}\ X\ \tau\ \mathit{of}$
$\quad\quad\quad\quad \bot\ \ \Rightarrow \mathit{invalid}\ \tau$
$\quad\quad\quad \mid \lfloor\bot\rfloor \Rightarrow \mathit{null}\ \tau$
$\quad\quad\quad \mid \lfloor\lfloor \mathit{mk}_{OclAny}\ \mathit{oid}\ \bot\ \rfloor\rfloor \Rightarrow\ \mathit{invalid}\ \tau\ \ (*\ \mathit{down}{-}\mathit{cast}\ \mathit{exception}\ *)$
$\quad\quad\quad \mid \lfloor\lfloor \mathit{mk}_{OclAny}\ \mathit{oid}\ \lfloor a\rfloor\ \rfloor\rfloor \Rightarrow\ \lfloor\lfloor \mathit{mk}_{Person}\ \mathit{oid}\ a\rfloor\rfloor)$

defs (**overloaded**) *OclAsType*$_{Person}$*-Person*:
  (*X*::*Person*) *.oclAsType*(*Person*) ≡ *X*

lemmas [*simp*] =
 OclAsType$_{OclAny}$*-OclAny*
 OclAsType$_{Person}$*-Person*

Context Passing

lemma *cp-OclAsType*$_{OclAny}$*-Person-Person*: *cp P* ⟹ *cp*(λ*X*. (*P* (*X*::*Person*)::*Person*) *.oclAsType*(*OclAny*))
by(*rule cpI1*, *simp-all add*: *OclAsType*$_{OclAny}$*-Person*)
lemma *cp-OclAsType*$_{OclAny}$*-OclAny-OclAny*: *cp P* ⟹ *cp*(λ*X*. (*P* (*X*::*OclAny*)::*OclAny*) *.oclAsType*(*OclAny*))
by(*rule cpI1*, *simp-all add*: *OclAsType*$_{OclAny}$*-OclAny*)
lemma *cp-OclAsType*$_{Person}$*-Person-Person*: *cp P* ⟹ *cp*(λ*X*. (*P* (*X*::*Person*)::*Person*) *.oclAsType*(*Person*))
by(*rule cpI1*, *simp-all add*: *OclAsType*$_{Person}$*-Person*)
lemma *cp-OclAsType*$_{Person}$*-OclAny-OclAny*: *cp P* ⟹ *cp*(λ*X*. (*P* (*X*::*OclAny*)::*OclAny*) *.oclAsType*(*Person*))
by(*rule cpI1*, *simp-all add*: *OclAsType*$_{Person}$*-OclAny*)

lemma *cp-OclAsType*$_{OclAny}$*-Person-OclAny*: *cp P* ⟹ *cp*(λ*X*. (*P* (*X*::*Person*)::*OclAny*) *.oclAsType*(*OclAny*))
by(*rule cpI1*, *simp-all add*: *OclAsType*$_{OclAny}$*-OclAny*)
lemma *cp-OclAsType*$_{OclAny}$*-OclAny-Person*: *cp P* ⟹ *cp*(λ*X*. (*P* (*X*::*OclAny*)::*Person*) *.oclAsType*(*OclAny*))
by(*rule cpI1*, *simp-all add*: *OclAsType*$_{OclAny}$*-Person*)
lemma *cp-OclAsType*$_{Person}$*-Person-OclAny*: *cp P* ⟹ *cp*(λ*X*. (*P* (*X*::*Person*)::*OclAny*) *.oclAsType*(*Person*))
by(*rule cpI1*, *simp-all add*: *OclAsType*$_{Person}$*-OclAny*)
lemma *cp-OclAsType*$_{Person}$*-OclAny-Person*: *cp P* ⟹ *cp*(λ*X*. (*P* (*X*::*OclAny*)::*Person*) *.oclAsType*(*Person*))
by(*rule cpI1*, *simp-all add*: *OclAsType*$_{Person}$*-Person*)

lemmas [*simp*] =
 cp-OclAsType$_{OclAny}$*-Person-Person*
 cp-OclAsType$_{OclAny}$*-OclAny-OclAny*
 cp-OclAsType$_{Person}$*-Person-Person*
 cp-OclAsType$_{Person}$*-OclAny-OclAny*

 cp-OclAsType$_{OclAny}$*-Person-OclAny*
 cp-OclAsType$_{OclAny}$*-OclAny-Person*
 cp-OclAsType$_{Person}$*-Person-OclAny*
 cp-OclAsType$_{Person}$*-OclAny-Person*

Execution with Invalid or Null as Argument

lemma *OclAsType*$_{OclAny}$*-OclAny-strict* : (*invalid*::*OclAny*) *.oclAsType*(*OclAny*) = *invalid*
by(*simp*)

lemma *OclAsType*$_{OclAny}$*-OclAny-nullstrict* : (*null*::*OclAny*) *.oclAsType*(*OclAny*) = *null*
by(*simp*)

```
lemma OclAsType$_{OclAny}$-Person-strict[simp] : (invalid::Person) .oclAsType(OclAny) = invalid
by(rule ext, simp add: bot-option-def invalid-def
                OclAsType$_{OclAny}$-Person)
```

```
lemma OclAsType$_{OclAny}$-Person-nullstrict[simp] : (null::Person) .oclAsType(OclAny) = null
by(rule ext, simp add: null-fun-def null-option-def bot-option-def
                OclAsType$_{OclAny}$-Person)
```

```
lemma OclAsType$_{Person}$-OclAny-strict[simp] : (invalid::OclAny) .oclAsType(Person) = invalid
by(rule ext, simp add: bot-option-def invalid-def
                OclAsType$_{Person}$-OclAny)
```

```
lemma OclAsType$_{Person}$-OclAny-nullstrict[simp] : (null::OclAny) .oclAsType(Person) = null
by(rule ext, simp add: null-fun-def null-option-def bot-option-def
                OclAsType$_{Person}$-OclAny)
```

```
lemma OclAsType$_{Person}$-Person-strict : (invalid::Person) .oclAsType(Person) = invalid
by(simp)
lemma OclAsType$_{Person}$-Person-nullstrict : (null::Person) .oclAsType(Person) = null
by(simp)
```

B.4.5. OclIsTypeOf

Definition

```
consts OclIsTypeOf$_{OclAny}$ :: 'α ⇒ Boolean ((-).oclIsTypeOf'(OclAny'))
consts OclIsTypeOf$_{Person}$ :: 'α ⇒ Boolean ((-).oclIsTypeOf'(Person'))
```

```
defs (overloaded) OclIsTypeOf$_{OclAny}$-OclAny:
    (X::OclAny) .oclIsTypeOf(OclAny) ≡
        (λτ. case X τ of
              ⊥  ⇒ invalid τ
            | ⌊⊥⌋ ⇒ true τ  (∗ invalid ?? ∗)
            | ⌊⌊mk$_{OclAny}$ oid ⊥ ⌋⌋ ⇒ true τ
            | ⌊⌊mk$_{OclAny}$ oid ⌊-⌋ ⌋⌋ ⇒ false τ)
```

```
defs (overloaded) OclIsTypeOf$_{OclAny}$-Person:
    (X::Person) .oclIsTypeOf(OclAny) ≡
        (λτ. case X τ of
              ⊥  ⇒ invalid τ
            | ⌊⊥⌋ ⇒ true τ    (∗ invalid ?? ∗)
            | ⌊⌊ - ⌋⌋ ⇒ false τ)
```

```
defs (overloaded) OclIsTypeOf$_{Person}$-OclAny:
    (X::OclAny) .oclIsTypeOf(Person) ≡
        (λτ. case X τ of
              ⊥  ⇒ invalid τ
```

$| \lfloor\bot\rfloor \Rightarrow \mathit{true}\ \tau$
$| \lfloor\lfloor \mathit{mk}_{OclAny}\ \mathit{oid}\ \bot\ \rfloor\rfloor \Rightarrow \mathit{false}\ \tau$
$| \lfloor\lfloor \mathit{mk}_{OclAny}\ \mathit{oid}\ \lfloor\text{-}\rfloor\ \rfloor\rfloor \Rightarrow \mathit{true}\ \tau)$

defs (**overloaded**) *OclIsTypeOf*$_{Person}$*-Person*:
$(X{::}\mathit{Person})\ .\mathit{oclIsTypeOf}(\mathit{Person}) \equiv$
$(\lambda\tau.\ \mathit{case}\ X\ \tau\ \mathit{of}$
$\bot \Rightarrow \mathit{invalid}\ \tau$
$|\ \text{-} \Rightarrow \mathit{true}\ \tau)$

Context Passing

lemma *cp-OclIsTypeOf*$_{OclAny}$*-Person-Person*: $\mathit{cp}\ P \Longrightarrow \mathit{cp}(\lambda X.(P(X{::}\mathit{Person}){::}\mathit{Person}).\mathit{oclIsTypeOf}(\mathit{OclAny}))$
by(*rule cpI1*, *simp-all add*: *OclIsTypeOf*$_{OclAny}$*-Person*)
lemma *cp-OclIsTypeOf*$_{OclAny}$*-OclAny-OclAny*: $\mathit{cp}\ P \Longrightarrow \mathit{cp}(\lambda X.(P(X{::}\mathit{OclAny}){::}\mathit{OclAny}).\mathit{oclIsTypeOf}(\mathit{OclAny}))$
by(*rule cpI1*, *simp-all add*: *OclIsTypeOf*$_{OclAny}$*-OclAny*)
lemma *cp-OclIsTypeOf*$_{Person}$*-Person-Person*: $\mathit{cp}\ P \Longrightarrow \mathit{cp}(\lambda X.(P(X{::}\mathit{Person}){::}\mathit{Person}).\mathit{oclIsTypeOf}(\mathit{Person}))$
by(*rule cpI1*, *simp-all add*: *OclIsTypeOf*$_{Person}$*-Person*)
lemma *cp-OclIsTypeOf*$_{Person}$*-OclAny-OclAny*: $\mathit{cp}\ P \Longrightarrow \mathit{cp}(\lambda X.(P(X{::}\mathit{OclAny}){::}\mathit{OclAny}).\mathit{oclIsTypeOf}(\mathit{Person}))$
by(*rule cpI1*, *simp-all add*: *OclIsTypeOf*$_{Person}$*-OclAny*)

lemma *cp-OclIsTypeOf*$_{OclAny}$*-Person-OclAny*: $\mathit{cp}\ P \Longrightarrow \mathit{cp}(\lambda X.(P(X{::}\mathit{Person}){::}\mathit{OclAny}).\mathit{oclIsTypeOf}(\mathit{OclAny}))$
by(*rule cpI1*, *simp-all add*: *OclIsTypeOf*$_{OclAny}$*-OclAny*)
lemma *cp-OclIsTypeOf*$_{OclAny}$*-OclAny-Person*: $\mathit{cp}\ P \Longrightarrow \mathit{cp}(\lambda X.(P(X{::}\mathit{OclAny}){::}\mathit{Person}).\mathit{oclIsTypeOf}(\mathit{OclAny}))$
by(*rule cpI1*, *simp-all add*: *OclIsTypeOf*$_{OclAny}$*-Person*)
lemma *cp-OclIsTypeOf*$_{Person}$*-Person-OclAny*: $\mathit{cp}\ P \Longrightarrow \mathit{cp}(\lambda X.(P(X{::}\mathit{Person}){::}\mathit{OclAny}).\mathit{oclIsTypeOf}(\mathit{Person}))$
by(*rule cpI1*, *simp-all add*: *OclIsTypeOf*$_{Person}$*-OclAny*)
lemma *cp-OclIsTypeOf*$_{Person}$*-OclAny-Person*: $\mathit{cp}\ P \Longrightarrow \mathit{cp}(\lambda X.(P(X{::}\mathit{OclAny}){::}\mathit{Person}).\mathit{oclIsTypeOf}(\mathit{Person}))$
by(*rule cpI1*, *simp-all add*: *OclIsTypeOf*$_{Person}$*-Person*)

lemmas [*simp*] =
cp-OclIsTypeOf$_{OclAny}$*-Person-Person*
cp-OclIsTypeOf$_{OclAny}$*-OclAny-OclAny*
cp-OclIsTypeOf$_{Person}$*-Person-Person*
cp-OclIsTypeOf$_{Person}$*-OclAny-OclAny*

cp-OclIsTypeOf$_{OclAny}$*-Person-OclAny*
cp-OclIsTypeOf$_{OclAny}$*-OclAny-Person*
cp-OclIsTypeOf$_{Person}$*-Person-OclAny*
cp-OclIsTypeOf$_{Person}$*-OclAny-Person*

Execution with Invalid or Null as Argument

lemma *OclIsTypeOf*$_{OclAny}$*-OclAny-strict1*[*simp*]:
$(\mathit{invalid}{::}\mathit{OclAny})\ .\mathit{oclIsTypeOf}(\mathit{OclAny}) = \mathit{invalid}$
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
OclIsTypeOf$_{OclAny}$*-OclAny*)

lemma *OclIsTypeOf*$_{OclAny}$*-OclAny-strict2*[*simp*]:
  (*null*::*OclAny*) .*oclIsTypeOf*(*OclAny*) = *true*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
  OclIsTypeOf$_{OclAny}$*-OclAny*)
lemma *OclIsTypeOf*$_{OclAny}$*-Person-strict1*[*simp*]:
  (*invalid*::*Person*) .*oclIsTypeOf*(*OclAny*) = *invalid*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
  OclIsTypeOf$_{OclAny}$*-Person*)
lemma *OclIsTypeOf*$_{OclAny}$*-Person-strict2*[*simp*]:
  (*null*::*Person*) .*oclIsTypeOf*(*OclAny*) = *true*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
  OclIsTypeOf$_{OclAny}$*-Person*)
lemma *OclIsTypeOf*$_{Person}$*-OclAny-strict1*[*simp*]:
  (*invalid*::*OclAny*) .*oclIsTypeOf*(*Person*) = *invalid*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
  OclIsTypeOf$_{Person}$*-OclAny*)
lemma *OclIsTypeOf*$_{Person}$*-OclAny-strict2*[*simp*]:
  (*null*::*OclAny*) .*oclIsTypeOf*(*Person*) = *true*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
  OclIsTypeOf$_{Person}$*-OclAny*)
lemma *OclIsTypeOf*$_{Person}$*-Person-strict1*[*simp*]:
  (*invalid*::*Person*) .*oclIsTypeOf*(*Person*) = *invalid*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
  OclIsTypeOf$_{Person}$*-Person*)
lemma *OclIsTypeOf*$_{Person}$*-Person-strict2*[*simp*]:
  (*null*::*Person*) .*oclIsTypeOf*(*Person*) = *true*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
  OclIsTypeOf$_{Person}$*-Person*)

Up Down Casting

lemma *actualType-larger-staticType*:
assumes *isdef*: $\tau \models (\delta\ X)$
shows $\tau \models (X{::}\mathit{Person})\ .\mathit{oclIsTypeOf}(\mathit{OclAny}) \triangleq \mathit{false}$
using *isdef*
by(*auto simp* : *null-option-def bot-option-def*
  OclIsTypeOf$_{OclAny}$*-Person foundation22 foundation16*)

lemma *down-cast-type*:
assumes *isOclAny*: $\tau \models (X{::}\mathit{OclAny})\ .\mathit{oclIsTypeOf}(\mathit{OclAny})$
and *non-null*: $\tau \models (\delta\ X)$
shows $\tau \models (X\ .\mathit{oclAsType}(\mathit{Person})) \triangleq \mathit{invalid}$
using *isOclAny non-null*
apply(*auto simp* : *bot-fun-def null-fun-def null-option-def bot-option-def null-def invalid-def*
  OclAsType$_{OclAny}$*-Person OclAsType*$_{Person}$*-OclAny foundation22 foundation16*
  split: *option.split type*$_{OclAny}$*.split type*$_{Person}$*.split*)
by(*simp add*: *OclIsTypeOf*$_{OclAny}$*-OclAny OclValid-def false-def true-def*)

lemma *down-cast-type′*:
assumes *isOclAny*: $\tau \models (X::OclAny)$ *.oclIsTypeOf*(*OclAny*)
and *non-null*: $\tau \models (\delta\ X)$
shows $\tau \models$ *not* (υ (*X .oclAsType*(*Person*)))
by(*rule foundation15*[*THEN iffD1*], *simp add*: *down-cast-type*[*OF assms*])

lemma *up-down-cast* :
assumes *isdef*: $\tau \models (\delta\ X)$
shows $\tau \models$ ((*X*::*Person*) *.oclAsType*(*OclAny*) *.oclAsType*(*Person*) $\triangleq$ *X*)
using *isdef*
by(*auto simp* : *null-fun-def null-option-def bot-option-def null-def invalid-def*
OclAsType$_{OclAny}$*-Person OclAsType*$_{Person}$*-OclAny foundation22 foundation16*
split: *option.split type*$_{Person}$*.split*)

lemma *up-down-cast-Person-OclAny-Person* [*simp*]:
shows ((*X*::*Person*) *.oclAsType*(*OclAny*) *.oclAsType*(*Person*) = *X*)
apply(*rule ext*, *rename-tac* τ)
apply(*rule foundation22*[*THEN iffD1*])
apply(*case-tac* $\tau \models (\delta\ X)$, *simp add*: *up-down-cast*)
apply(*simp add*: *defined-split*, *elim disjE*)
apply(*erule StrongEq-L-subst2-rev*, *simp*, *simp*)+
done

lemma *up-down-cast-Person-OclAny-Person′*:
assumes $\tau \models \upsilon\ X$
shows $\tau \models$ (((*X* :: *Person*) *.oclAsType*(*OclAny*) *.oclAsType*(*Person*)) $\doteq$ *X*)
apply(*simp only*: *up-down-cast-Person-OclAny-Person StrictRefEq*$_{Object}$*-*$_{Person}$)
by(*rule StrictRefEq*$_{Object}$*-sym*, *simp add*: *assms*)

lemma *up-down-cast-Person-OclAny-Person″*:
assumes $\tau \models \upsilon$ (*X* :: *Person*)
shows $\tau \models$ (*X .oclIsTypeOf*(*Person*) *implies* (*X .oclAsType*(*OclAny*) *.oclAsType*(*Person*)) $\doteq$ *X*)
apply(*simp add*: *OclValid-def*)
apply(*subst cp-OclImplies*)
apply(*simp add*: *StrictRefEq*$_{Object}$*-*$_{Person}$ *StrictRefEq*$_{Object}$*-sym*[*OF assms*, *simplified OclValid-def*])
apply(*subst cp-OclImplies*[*symmetric*])
by (*simp add*: *OclImplies-true*)

B.4.6. OclIsKindOf

Definition

consts *OclIsKindOf*$_{OclAny}$ :: $'\alpha \Rightarrow$ *Boolean* ((-)*.oclIsKindOf′*(*OclAny′*))
consts *OclIsKindOf*$_{Person}$ :: $'\alpha \Rightarrow$ *Boolean* ((-)*.oclIsKindOf′*(*Person′*))

defs (**overloaded**) *OclIsKindOf*$_{OclAny}$*-OclAny*:
(*X*::*OclAny*) *.oclIsKindOf*(*OclAny*) $\equiv$

$(\lambda\tau.\ case\ X\ \tau\ of$
$\bot \Rightarrow invalid\ \tau$
$\mid - \Rightarrow true\ \tau)$

defs (**overloaded**) *OclIsKindOf*$_{OclAny}$*-Person*:
$(X{::}Person)\ .oclIsKindOf(OclAny) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\bot \Rightarrow invalid\ \tau$
$\mid - \Rightarrow true\ \tau)$

defs (**overloaded**) *OclIsKindOf*$_{Person}$*-OclAny*:
$(X{::}OclAny)\ .oclIsKindOf(Person) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\bot\ \ \Rightarrow invalid\ \tau$
$\mid \lfloor\bot\rfloor \Rightarrow true\ \tau$
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \bot\ \rfloor\rfloor \Rightarrow false\ \tau$
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \lfloor\text{-}\rfloor\ \rfloor\rfloor \Rightarrow true\ \tau)$

defs (**overloaded**) *OclIsKindOf*$_{Person}$*-Person*:
$(X{::}Person)\ .oclIsKindOf(Person) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\bot \Rightarrow invalid\ \tau$
$\mid - \Rightarrow true\ \tau)$

Context Passing

lemma *cp-OclIsKindOf*$_{OclAny}$*-Person-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X{::}Person){::}Person).oclIsKindOf(OclAny))$
by(*rule cpI1*, *simp-all add*: *OclIsKindOf*$_{OclAny}$*-Person*)
lemma *cp-OclIsKindOf*$_{OclAny}$*-OclAny-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X{::}OclAny){::}OclAny).oclIsKindOf(OclAny))$
by(*rule cpI1*, *simp-all add*: *OclIsKindOf*$_{OclAny}$*-OclAny*)
lemma *cp-OclIsKindOf*$_{Person}$*-Person-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X{::}Person){::}Person).oclIsKindOf(Person))$
by(*rule cpI1*, *simp-all add*: *OclIsKindOf*$_{Person}$*-Person*)
lemma *cp-OclIsKindOf*$_{Person}$*-OclAny-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X{::}OclAny){::}OclAny).oclIsKindOf(Person))$
by(*rule cpI1*, *simp-all add*: *OclIsKindOf*$_{Person}$*-OclAny*)

lemma *cp-OclIsKindOf*$_{OclAny}$*-Person-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X{::}Person){::}OclAny).oclIsKindOf(OclAny))$
by(*rule cpI1*, *simp-all add*: *OclIsKindOf*$_{OclAny}$*-OclAny*)
lemma *cp-OclIsKindOf*$_{OclAny}$*-OclAny-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X{::}OclAny){::}Person).oclIsKindOf(OclAny))$
by(*rule cpI1*, *simp-all add*: *OclIsKindOf*$_{OclAny}$*-Person*)
lemma *cp-OclIsKindOf*$_{Person}$*-Person-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X{::}Person){::}OclAny).oclIsKindOf(Person))$
by(*rule cpI1*, *simp-all add*: *OclIsKindOf*$_{Person}$*-OclAny*)
lemma *cp-OclIsKindOf*$_{Person}$*-OclAny-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X{::}OclAny){::}Person).oclIsKindOf(Person))$
by(*rule cpI1*, *simp-all add*: *OclIsKindOf*$_{Person}$*-Person*)

lemmas [*simp*] =
cp-OclIsKindOf$_{OclAny}$*-Person-Person*

cp-OclIsKindOf$_{OclAny}$*-OclAny-OclAny*
cp-OclIsKindOf$_{Person}$*-Person-Person*
cp-OclIsKindOf$_{Person}$*-OclAny-OclAny*

cp-OclIsKindOf$_{OclAny}$*-Person-OclAny*
cp-OclIsKindOf$_{OclAny}$*-OclAny-Person*
cp-OclIsKindOf$_{Person}$*-Person-OclAny*
cp-OclIsKindOf$_{Person}$*-OclAny-Person*

Execution with Invalid or Null as Argument

lemma *OclIsKindOf*$_{OclAny}$*-OclAny-strict1*[*simp*] : (*invalid*::*OclAny*) .*oclIsKindOf*(*OclAny*) = *invalid*
by(*rule ext*, *simp add*: *invalid-def bot-option-def*
OclIsKindOf$_{OclAny}$*-OclAny*)

lemma *OclIsKindOf*$_{OclAny}$*-OclAny-strict2*[*simp*] : (*null*::*OclAny*) .*oclIsKindOf*(*OclAny*) = *true*
by(*rule ext*, *simp add*: *null-fun-def null-option-def*
OclIsKindOf$_{OclAny}$*-OclAny*)

lemma *OclIsKindOf*$_{OclAny}$*-Person-strict1*[*simp*] : (*invalid*::*Person*) .*oclIsKindOf*(*OclAny*) = *invalid*
by(*rule ext*, *simp add*: *bot-option-def invalid-def*
OclIsKindOf$_{OclAny}$*-Person*)

lemma *OclIsKindOf*$_{OclAny}$*-Person-strict2*[*simp*] : (*null*::*Person*) .*oclIsKindOf*(*OclAny*) = *true*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def*
OclIsKindOf$_{OclAny}$*-Person*)

lemma *OclIsKindOf*$_{Person}$*-OclAny-strict1*[*simp*]: (*invalid*::*OclAny*) .*oclIsKindOf*(*Person*) = *invalid*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
OclIsKindOf$_{Person}$*-OclAny*)

lemma *OclIsKindOf*$_{Person}$*-OclAny-strict2*[*simp*]: (*null*::*OclAny*) .*oclIsKindOf*(*Person*) = *true*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
OclIsKindOf$_{Person}$*-OclAny*)

lemma *OclIsKindOf*$_{Person}$*-Person-strict1*[*simp*]: (*invalid*::*Person*) .*oclIsKindOf*(*Person*) = *invalid*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
OclIsKindOf$_{Person}$*-Person*)

lemma *OclIsKindOf*$_{Person}$*-Person-strict2*[*simp*]: (*null*::*Person*) .*oclIsKindOf*(*Person*) = *true*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
OclIsKindOf$_{Person}$*-Person*)

Up Down Casting

lemma *actualKind-larger-staticKind*:
assumes *isdef*: $\tau \models (\delta\ X)$
shows $\tau \models$ ((*X*::*Person*) .*oclIsKindOf*(*OclAny*) $\triangleq$ *true*)

using *isdef*
by(*auto simp* : *bot-option-def*
OclIsKindOf$_{OclAny}$*-Person foundation22 foundation16*)

lemma *down-cast-kind*:
assumes *isOclAny*: $\neg$ ($\tau \models$ ((*X*::*OclAny*).*oclIsKindOf*(*Person*)))
and *non-null*: $\tau \models (\delta\ X)$
shows $\tau \models$ ((*X .oclAsType*(*Person*)) $\triangleq$ *invalid*)
using *isOclAny non-null*
apply(*auto simp* : *bot-fun-def null-fun-def null-option-def bot-option-def null-def invalid-def*
OclAsType$_{OclAny}$*-Person OclAsType*$_{Person}$*-OclAny foundation22 foundation16*
split: *option.split type*$_{OclAny}$*.split type*$_{Person}$*.split*)
by(*simp add*: *OclIsKindOf*$_{Person}$*-OclAny OclValid-def false-def true-def*)

B.4.7. OclAllInstances

To denote OCL-types occuring in OCL expressions syntactically—as, for example, as “argument” of oclAllInstances ()—we use the inverses of the injection functions into the object universes; we show that this is sufficient “characterization.”

definition *Person* $\equiv$ *OclAsType*$_{Person}$-$\mathfrak{A}$
definition *OclAny* $\equiv$ *OclAsType*$_{OclAny}$-$\mathfrak{A}$
lemmas [*simp*] = *Person-def OclAny-def*

lemma *OclAllInstances-generic*$_{OclAny}$*-exec*: *OclAllInstances-generic pre-post OclAny* =
($\lambda\tau$. *Abs-Set*$_{base}$ $\lfloor\lfloor$ *Some* ‘ *OclAny* ‘ *ran* (*heap* (*pre-post* τ)) $\rfloor\rfloor$)
proof –
let *?S1* = $\lambda\tau$. *OclAny* ‘ *ran* (*heap* (*pre-post* τ))
let *?S2* = $\lambda\tau$. *?S1* τ – {*None*}
have *B* : $\bigwedge\tau$. *?S2* $\tau \subseteq$ *?S1* τ **by** *auto*
have *C* : $\bigwedge\tau$. *?S1* $\tau \subseteq$ *?S2* τ **by**(*auto simp*: *OclAsType*$_{OclAny}$-$\mathfrak{A}$*-some*)

show *?thesis* **by**(*insert equalityI*[*OF B C*], *simp*)
qed

lemma *OclAllInstances-at-post*$_{OclAny}$*-exec*: *OclAny .allInstances*() =
($\lambda\tau$. *Abs-Set*$_{base}$ $\lfloor\lfloor$ *Some* ‘ *OclAny* ‘ *ran* (*heap* (*snd* τ)) $\rfloor\rfloor$)
unfolding *OclAllInstances-at-post-def*
by(*rule OclAllInstances-generic*$_{OclAny}$*-exec*)

lemma *OclAllInstances-at-pre*$_{OclAny}$*-exec*: *OclAny .allInstances@pre*() =
($\lambda\tau$. *Abs-Set*$_{base}$ $\lfloor\lfloor$ *Some* ‘ *OclAny* ‘ *ran* (*heap* (*fst* τ)) $\rfloor\rfloor$)
unfolding *OclAllInstances-at-pre-def*
by(*rule OclAllInstances-generic*$_{OclAny}$*-exec*)

OclIsTypeOf

lemma *OclAny-allInstances-generic-oclIsTypeOf*$_{OclAny}$*1*:

assumes [*simp*]: $\bigwedge x$. *pre-post* $(x, x) = x$
shows $\exists\,\tau$. ($\tau \models$ ((*OclAllInstances-generic pre-post OclAny*)$-$>*forAll*(*X*|*X .oclIsTypeOf*(*OclAny*))))
apply(*rule-tac x* = τ_0 **in** *exI*, *simp add*: τ_0-*def OclValid-def del*: *OclAllInstances-generic-def*)
apply(*simp only*: *assms OclForall-def refl if-True*
OclAllInstances-generic-defined[*simplified OclValid-def*])
apply(*simp only*: *OclAllInstances-generic-def*)
apply(*subst* (*1 2 3*) *Abs-Set*$_{base}$*-inverse*, *simp add*: *bot-option-def*)
by(*simp add*: *OclIsTypeOf*$_{OclAny}$*-OclAny*)

lemma *OclAny-allInstances-at-post-oclIsTypeOf*$_{OclAny}$*1*:
$\exists\,\tau$. ($\tau \models$ (*OclAny .allInstances*()$-$>*forAll*(*X*|*X .oclIsTypeOf*(*OclAny*))))
unfolding *OclAllInstances-at-post-def*
by(*rule OclAny-allInstances-generic-oclIsTypeOf*$_{OclAny}$*1*, *simp*)

lemma *OclAny-allInstances-at-pre-oclIsTypeOf*$_{OclAny}$*1*:
$\exists\,\tau$. ($\tau \models$ (*OclAny .allInstances@pre*()$-$>*forAll*(*X*|*X .oclIsTypeOf*(*OclAny*))))
unfolding *OclAllInstances-at-pre-def*
by(*rule OclAny-allInstances-generic-oclIsTypeOf*$_{OclAny}$*1*, *simp*)

lemma *OclAny-allInstances-generic-oclIsTypeOf*$_{OclAny}$*2*:
assumes [*simp*]: $\bigwedge x$. *pre-post* $(x, x) = x$
shows $\exists\,\tau$. ($\tau \models$ *not* ((*OclAllInstances-generic pre-post OclAny*)$-$>*forAll*(*X*|*X .oclIsTypeOf*(*OclAny*))))
proof $-$ **fix** *oid a* **let** *?t0* = (|*heap* = *empty*(*oid* $\mapsto$ *in*$_{OclAny}$ (*mk*$_{OclAny}$ *oid* $\lfloor a \rfloor$)),
assocs = *empty*|) **show** *?thesis*
apply(*rule-tac x* = (*?t0*, *?t0*) **in** *exI*, *simp add*: *OclValid-def del*: *OclAllInstances-generic-def*)
apply(*simp only*: *OclForall-def refl if-True*
OclAllInstances-generic-defined[*simplified OclValid-def*])
apply(*simp only*: *OclAllInstances-generic-def OclAsType*$_{OclAny}$-$\mathfrak{A}$*-def*)
apply(*subst* (*1 2 3*) *Abs-Set*$_{base}$*-inverse*, *simp add*: *bot-option-def*)
by(*simp add*: *OclIsTypeOf*$_{OclAny}$*-OclAny OclNot-def OclAny-def*)
qed

lemma *OclAny-allInstances-at-post-oclIsTypeOf*$_{OclAny}$*2*:
$\exists\,\tau$. ($\tau \models$ *not* (*OclAny .allInstances*()$-$>*forAll*(*X*|*X .oclIsTypeOf*(*OclAny*))))
unfolding *OclAllInstances-at-post-def*
by(*rule OclAny-allInstances-generic-oclIsTypeOf*$_{OclAny}$*2*, *simp*)

lemma *OclAny-allInstances-at-pre-oclIsTypeOf*$_{OclAny}$*2*:
$\exists\,\tau$. ($\tau \models$ *not* (*OclAny .allInstances@pre*()$-$>*forAll*(*X*|*X .oclIsTypeOf*(*OclAny*))))
unfolding *OclAllInstances-at-pre-def*
by(*rule OclAny-allInstances-generic-oclIsTypeOf*$_{OclAny}$*2*, *simp*)

lemma *Person-allInstances-generic-oclIsTypeOf*$_{Person}$:
$\tau \models$ ((*OclAllInstances-generic pre-post Person*)$-$>*forAll*(*X*|*X .oclIsTypeOf*(*Person*)))
apply(*simp add*: *OclValid-def del*: *OclAllInstances-generic-def*)
apply(*simp only*: *OclForall-def refl if-True*
OclAllInstances-generic-defined[*simplified OclValid-def*])
apply(*simp only*: *OclAllInstances-generic-def*)

apply(*subst* (*1 2 3*) *Abs-Set*$_{base}$*-inverse*, *simp add*: *bot-option-def*)
by(*simp add*: *OclIsTypeOf*$_{Person}$*-Person*)

lemma *Person-allInstances-at-post-oclIsTypeOf*$_{Person}$:
$\tau \models$ (*Person .allInstances*()*−>forAll*(*X*|*X .oclIsTypeOf*(*Person*)))
unfolding *OclAllInstances-at-post-def*
by(*rule Person-allInstances-generic-oclIsTypeOf*$_{Person}$)

lemma *Person-allInstances-at-pre-oclIsTypeOf*$_{Person}$:
$\tau \models$ (*Person .allInstances@pre*()*−>forAll*(*X*|*X .oclIsTypeOf*(*Person*)))
unfolding *OclAllInstances-at-pre-def*
by(*rule Person-allInstances-generic-oclIsTypeOf*$_{Person}$)

OclIsKindOf

lemma *OclAny-allInstances-generic-oclIsKindOf*$_{OclAny}$:
$\tau \models$ ((*OclAllInstances-generic pre-post OclAny*)*−>forAll*(*X*|*X .oclIsKindOf*(*OclAny*)))
apply(*simp add*: *OclValid-def del*: *OclAllInstances-generic-def*)
apply(*simp only*: *OclForall-def refl if-True*
OclAllInstances-generic-defined[*simplified OclValid-def*])
apply(*simp only*: *OclAllInstances-generic-def*)
apply(*subst* (*1 2 3*) *Abs-Set*$_{base}$*-inverse*, *simp add*: *bot-option-def*)
by(*simp add*: *OclIsKindOf*$_{OclAny}$*-OclAny*)

lemma *OclAny-allInstances-at-post-oclIsKindOf*$_{OclAny}$:
$\tau \models$ (*OclAny .allInstances*()*−>forAll*(*X*|*X .oclIsKindOf*(*OclAny*)))
unfolding *OclAllInstances-at-post-def*
by(*rule OclAny-allInstances-generic-oclIsKindOf*$_{OclAny}$)

lemma *OclAny-allInstances-at-pre-oclIsKindOf*$_{OclAny}$:
$\tau \models$ (*OclAny .allInstances@pre*()*−>forAll*(*X*|*X .oclIsKindOf*(*OclAny*)))
unfolding *OclAllInstances-at-pre-def*
by(*rule OclAny-allInstances-generic-oclIsKindOf*$_{OclAny}$)

lemma *Person-allInstances-generic-oclIsKindOf*$_{OclAny}$:
$\tau \models$ ((*OclAllInstances-generic pre-post Person*)*−>forAll*(*X*|*X .oclIsKindOf*(*OclAny*)))
apply(*simp add*: *OclValid-def del*: *OclAllInstances-generic-def*)
apply(*simp only*: *OclForall-def refl if-True*
OclAllInstances-generic-defined[*simplified OclValid-def*])
apply(*simp only*: *OclAllInstances-generic-def*)
apply(*subst* (*1 2 3*) *Abs-Set*$_{base}$*-inverse*, *simp add*: *bot-option-def*)
by(*simp add*: *OclIsKindOf*$_{OclAny}$*-Person*)

lemma *Person-allInstances-at-post-oclIsKindOf*$_{OclAny}$:
$\tau \models$ (*Person .allInstances*()*−>forAll*(*X*|*X .oclIsKindOf*(*OclAny*)))
unfolding *OclAllInstances-at-post-def*
by(*rule Person-allInstances-generic-oclIsKindOf*$_{OclAny}$)

lemma *Person-allInstances-at-pre-oclIsKindOf*$_{OclAny}$:
$\tau \models (\mathit{Person}\ .\mathit{allInstances@pre}()\!-\!\!>\!\mathit{forAll}(X|X\ .\mathit{oclIsKindOf}(\mathit{OclAny})))$
unfolding *OclAllInstances-at-pre-def*
by(*rule Person-allInstances-generic-oclIsKindOf*$_{OclAny}$)

lemma *Person-allInstances-generic-oclIsKindOf*$_{Person}$:
$\tau \models ((\mathit{OclAllInstances\text{-}generic\ pre\text{-}post\ Person})\!-\!\!>\!\mathit{forAll}(X|X\ .\mathit{oclIsKindOf}(\mathit{Person})))$
apply(*simp add*: *OclValid-def del*: *OclAllInstances-generic-def*)
apply(*simp only*: *OclForall-def refl if-True*
OclAllInstances-generic-defined[*simplified OclValid-def*])
apply(*simp only*: *OclAllInstances-generic-def*)
apply(*subst* (*1 2 3*) *Abs-Set*$_{base}$*-inverse*, *simp add*: *bot-option-def*)
by(*simp add*: *OclIsKindOf*$_{Person}$*-Person*)

lemma *Person-allInstances-at-post-oclIsKindOf*$_{Person}$:
$\tau \models (\mathit{Person}\ .\mathit{allInstances}()\!-\!\!>\!\mathit{forAll}(X|X\ .\mathit{oclIsKindOf}(\mathit{Person})))$
unfolding *OclAllInstances-at-post-def*
by(*rule Person-allInstances-generic-oclIsKindOf*$_{Person}$)

lemma *Person-allInstances-at-pre-oclIsKindOf*$_{Person}$:
$\tau \models (\mathit{Person}\ .\mathit{allInstances@pre}()\!-\!\!>\!\mathit{forAll}(X|X\ .\mathit{oclIsKindOf}(\mathit{Person})))$
unfolding *OclAllInstances-at-pre-def*
by(*rule Person-allInstances-generic-oclIsKindOf*$_{Person}$)

B.4.8. The Accessors (any, boss, salary)

Should be generated entirely from a class-diagram.

Definition (of the association Employee-Boss)

We start with a oid for the association; this oid can be used in presence of association classes to represent the association inside an object, pretty much similar to the Design_UML, where we stored an `oid` inside the class as "pointer."

definition $\mathit{oid}_{Person}\mathscr{BOSS}$ ::*oid* **where** $\mathit{oid}_{Person}\mathscr{BOSS} = 10$

From there on, we can already define an empty state which must contain for $\mathit{oid}_{Person}\mathscr{BOSS}$ the empty relation (encoded as association list, since there are associations with a Sequence-like structure).

definition *eval-extract* :: $('\mathfrak{A},('a{::}\mathit{object})\ \mathit{option\ option})\ \mathit{val}$
$\Rightarrow (\mathit{oid} \Rightarrow ('\mathfrak{A},'c{::}\mathit{null})\ \mathit{val})$
$\Rightarrow ('\mathfrak{A},'c{::}\mathit{null})\ \mathit{val}$
where $\mathit{eval\text{-}extract}\ X f = (\lambda\ \tau.\ \mathit{case}\ X\ \tau\ \mathit{of}$
$\bot \Rightarrow \mathit{invalid}\ \tau\ \ (*\ \mathit{exception\ propagation}\ *)$
$|\ \lfloor\ \bot\ \rfloor \Rightarrow \mathit{invalid}\ \tau\ (*\ \mathit{dereferencing\ null\ pointer}\ *)$
$|\ \lfloor\lfloor\ \mathit{obj}\ \rfloor\rfloor \Rightarrow f\ (\mathit{oid\text{-}of}\ \mathit{obj})\ \tau)$

definition *choose*$_2$*-1* = *fst*
definition *choose*$_2$*-2* = *snd*

definition *List-flatten* = (λ*l*. (*foldl* ((λ*acc*. (λ*l*. (*foldl* ((λ*acc*. (λ*l*. (*Cons* (*l*) (*acc*))))) (*acc*) ((*rev* (*l*))))))) (*Nil*) ((*rev* (*l*)))))
definition *deref-assocs*$_2$ :: ('𝔄 *state* × '𝔄 *state* ⇒ '𝔄 *state*)
⇒ (*oid list list* ⇒ *oid list* × *oid list*)
⇒ *oid*
⇒ (*oid list* ⇒ ('𝔄,'*f*)*val*)
⇒ *oid*
⇒ ('𝔄, '*f*::*null*)*val*
where *deref-assocs*$_2$ *pre-post to-from assoc-oid f oid* =
(λτ. *case* (*assocs* (*pre-post* τ)) *assoc-oid of*
⌊ *S* ⌋ ⇒ *f* (*List-flatten* (*map* (*choose*$_2$*-2* ∘ *to-from*)
(*filter* (λ *p*. *List.member* (*choose*$_2$*-1* (*to-from p*)) *oid*) *S*)))
τ
| - ⇒ *invalid* τ)

The *pre-post*-parameter is configured with *fst* or *snd*, the *to-from*-parameter either with the identity *id* or the following combinator *switch*:

definition *switch*$_2$*-1* = (λ[*x*,*y*]⇒ (*x*,*y*))
definition *switch*$_2$*-2* = (λ[*x*,*y*]⇒ (*y*,*x*))
definition *switch*$_3$*-1* = (λ[*x*,*y*,*z*]⇒ (*x*,*y*))
definition *switch*$_3$*-2* = (λ[*x*,*y*,*z*]⇒ (*x*,*z*))
definition *switch*$_3$*-3* = (λ[*x*,*y*,*z*]⇒ (*y*,*x*))
definition *switch*$_3$*-4* = (λ[*x*,*y*,*z*]⇒ (*y*,*z*))
definition *switch*$_3$*-5* = (λ[*x*,*y*,*z*]⇒ (*z*,*x*))
definition *switch*$_3$*-6* = (λ[*x*,*y*,*z*]⇒ (*z*,*y*))

definition *select-object* :: (('𝔄, '*b*::*null*)*val*)
⇒ (('𝔄,'*b*)*val* ⇒ ('𝔄,'*c*)*val* ⇒ ('𝔄, '*b*)*val*)
⇒ (('𝔄, '*b*)*val* ⇒ ('𝔄, '*d*)*val*)
⇒ (*oid* ⇒ ('𝔄,'*c*::*null*)*val*)
⇒ *oid list*
⇒ ('𝔄, '*d*)*val*
where *select-object mt incl smash deref l* = *smash*(*foldl incl mt* (*map deref l*))
(∗ *smash returns null with mt in input* (*in this case*, *object contains null pointer*) ∗)

The continuation *f* is usually instantiated with a smashing function which is either the identity *id* or, for `0..1` cardinalities of associations, the *OclANY*-selector which also handles the *null*-cases appropriately. A standard use-case for this combinator is for example:

term (*select-object mtSet UML-Set.OclIncluding OclANY f l oid*)::('𝔄, '*a*::*null*)*val*

definition *deref-oid*$_{Person}$ :: (𝔄 *state* × 𝔄 *state* ⇒ 𝔄 *state*)
⇒ (*type*$_{Person}$ ⇒ (𝔄, '*c*::*null*)*val*)
⇒ *oid*
⇒ (𝔄, '*c*::*null*)*val*
where *deref-oid*$_{Person}$ *fst-snd f oid* = (λτ. *case* (*heap* (*fst-snd* τ)) *oid of*

$\lfloor in_{Person}\ obj \rfloor \Rightarrow f\ obj\ \tau$
$|$ - $\Rightarrow invalid\ \tau)$

definition $deref\text{-}oid_{OclAny} :: (\mathfrak{A}\ state \times \mathfrak{A}\ state \Rightarrow \mathfrak{A}\ state)$
$\Rightarrow (type_{OclAny} \Rightarrow (\mathfrak{A},\ 'c{::}null)val)$
$\Rightarrow oid$
$\Rightarrow (\mathfrak{A},\ 'c{::}null)val$
where $deref\text{-}oid_{OclAny}\ fst\text{-}snd\ f\ oid = (\lambda\tau.\ case\ (heap\ (fst\text{-}snd\ \tau))\ oid\ of$
$\lfloor in_{OclAny}\ obj \rfloor \Rightarrow f\ obj\ \tau$
$|$ - $\Rightarrow invalid\ \tau)$

pointer undefined in state or not referencing a type conform object representation

definition $select_{OclAny}\mathscr{ANY}\ f = (\lambda\ X.\ case\ X\ of$
$(mk_{OclAny}$ - $\bot) \Rightarrow null$
$|\ (mk_{OclAny}$ - $\lfloor any \rfloor) \Rightarrow f\ (\lambda x$ -. $\lfloor\lfloor x \rfloor\rfloor)\ any)$

definition $select_{Person}\mathscr{BOSS}\ f = select\text{-}object\ mtSet\ UML\text{-}Set.OclIncluding\ OclANY\ (f\ (\lambda x$ -. $\lfloor\lfloor x \rfloor\rfloor))$

definition $select_{Person}\mathscr{SALARY}\ f = (\lambda\ X.\ case\ X\ of$
$(mk_{Person}$ - $\bot) \Rightarrow null$
$|\ (mk_{Person}$ - $\lfloor salary \rfloor) \Rightarrow f\ (\lambda x$ -. $\lfloor\lfloor x \rfloor\rfloor)\ salary)$

definition $deref\text{-}assocs_2\mathscr{BOSS}\ fst\text{-}snd\ f = (\lambda\ mk_{Person}\ oid$ - $\Rightarrow$
$deref\text{-}assocs_2\ fst\text{-}snd\ switch_2\text{-}1\ oid_{Person}\mathscr{BOSS}\ f\ oid)$

definition $in\text{-}pre\text{-}state = fst$
definition $in\text{-}post\text{-}state = snd$

definition $reconst\text{-}basetype = (\lambda\ convert\ x.\ convert\ x)$

definition $dot_{OclAny}\mathscr{ANY} :: OclAny \Rightarrow$ - $((1(\text{-}).any)\ 50)$
where $(X).any = eval\text{-}extract\ X$
$(deref\text{-}oid_{OclAny}\ in\text{-}post\text{-}state$
$(select_{OclAny}\mathscr{ANY}$
$reconst\text{-}basetype))$

definition $dot_{Person}\mathscr{BOSS} :: Person \Rightarrow Person\ \ ((1(\text{-}).boss)\ 50)$
where $(X).boss = eval\text{-}extract\ X$
$(deref\text{-}oid_{Person}\ in\text{-}post\text{-}state$
$(deref\text{-}assocs_2\mathscr{BOSS}\ in\text{-}post\text{-}state$
$(select_{Person}\mathscr{BOSS}$
$(deref\text{-}oid_{Person}\ in\text{-}post\text{-}state))))$

definition *dot*$_{Person}\mathcal{SALARY}$ :: *Person* ⇒ *Integer* ((*1*(-).*salary*) *50*)
where (X).*salary* = *eval-extract X*
(*deref-oid*$_{Person}$ *in-post-state*
(*select*$_{Person}\mathcal{SALARY}$
reconst-basetype))

definition *dot*$_{OclAny}\mathcal{ANY}$*-at-pre* :: *OclAny* ⇒ - ((*1*(-).*any@pre*) *50*)
where (X).*any@pre* = *eval-extract X*
(*deref-oid*$_{OclAny}$ *in-pre-state*
(*select*$_{OclAny}\mathcal{ANY}$
reconst-basetype))

definition *dot*$_{Person}\mathcal{BOSS}$*-at-pre*:: *Person* ⇒ *Person* ((*1*(-).*boss@pre*) *50*)
where (X).*boss@pre* = *eval-extract X*
(*deref-oid*$_{Person}$ *in-pre-state*
(*deref-assocs*$_2\mathcal{BOSS}$ *in-pre-state*
(*select*$_{Person}\mathcal{BOSS}$
(*deref-oid*$_{Person}$ *in-pre-state*))))

definition *dot*$_{Person}\mathcal{SALARY}$*-at-pre*:: *Person* ⇒ *Integer* ((*1*(-).*salary@pre*) *50*)
where (X).*salary@pre* = *eval-extract X*
(*deref-oid*$_{Person}$ *in-pre-state*
(*select*$_{Person}\mathcal{SALARY}$
reconst-basetype))

lemmas *dot-accessor* =
dot$_{OclAny}\mathcal{ANY}$*-def*
dot$_{Person}\mathcal{BOSS}$*-def*
dot$_{Person}\mathcal{SALARY}$*-def*
dot$_{OclAny}\mathcal{ANY}$*-at-pre-def*
dot$_{Person}\mathcal{BOSS}$*-at-pre-def*
dot$_{Person}\mathcal{SALARY}$*-at-pre-def*

Context Passing

lemmas [*simp*] = *eval-extract-def*

lemma *cp-dot*$_{OclAny}\mathcal{ANY}$: $((X).any)\ \tau = ((\lambda\text{-}.\ X\ \tau).any)\ \tau$ **by** (*simp add*: *dot-accessor*)
lemma *cp-dot*$_{Person}\mathcal{BOSS}$: $((X).boss)\ \tau = ((\lambda\text{-}.\ X\ \tau).boss)\ \tau$ **by** (*simp add*: *dot-accessor*)
lemma *cp-dot*$_{Person}\mathcal{SALARY}$: $((X).salary)\ \tau = ((\lambda\text{-}.\ X\ \tau).salary)\ \tau$ **by** (*simp add*: *dot-accessor*)

lemma *cp-dot*$_{OclAny}\mathcal{ANY}$*-at-pre*: $((X).any@pre)\ \tau = ((\lambda\text{-}.\ X\ \tau).any@pre)\ \tau$ **by** (*simp add*: *dot-accessor*)
lemma *cp-dot*$_{Person}\mathcal{BOSS}$*-at-pre*: $((X).boss@pre)\ \tau = ((\lambda\text{-}.\ X\ \tau).boss@pre)\ \tau$ **by** (*simp add*: *dot-accessor*)
lemma *cp-dot*$_{Person}\mathcal{SALARY}$*-at-pre*: $((X).salary@pre)\ \tau = ((\lambda\text{-}.\ X\ \tau).salary@pre)\ \tau$ **by** (*simp add*: *dot-accessor*)

lemmas *cp-dot*$_{OclAny}\mathcal{ANY}$*-I* [*simp*, *intro*!]=
cp-dot$_{OclAny}\mathcal{ANY}$[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

lemmas *cp-dot*$_{OclAny}\mathcal{ANY}$*-at-pre-I* [*simp*, *intro*!]=
  cp-dot$_{OclAny}\mathcal{ANY}$*-at-pre*[*THEN allI*[*THEN allI*],
    of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

lemmas *cp-dot*$_{Person}\mathcal{BOSS}$*-I* [*simp*, *intro*!]=
  cp-dot$_{Person}\mathcal{BOSS}$[*THEN allI*[*THEN allI*],
    of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]
lemmas *cp-dot*$_{Person}\mathcal{BOSS}$*-at-pre-I* [*simp*, *intro*!]=
  cp-dot$_{Person}\mathcal{BOSS}$*-at-pre*[*THEN allI*[*THEN allI*],
    of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

lemmas *cp-dot*$_{Person}\mathcal{SALARY}$*-I* [*simp*, *intro*!]=
  cp-dot$_{Person}\mathcal{SALARY}$[*THEN allI*[*THEN allI*],
    of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]
lemmas *cp-dot*$_{Person}\mathcal{SALARY}$*-at-pre-I* [*simp*, *intro*!]=
  cp-dot$_{Person}\mathcal{SALARY}$*-at-pre*[*THEN allI*[*THEN allI*],
    of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

Execution with Invalid or Null as Argument

lemma *dot*$_{OclAny}\mathcal{ANY}$*-nullstrict* [*simp*]: (*null*).*any* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma *dot*$_{OclAny}\mathcal{ANY}$*-at-pre-nullstrict* [*simp*] : (*null*).*any@pre* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma *dot*$_{OclAny}\mathcal{ANY}$*-strict* [*simp*] : (*invalid*).*any* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma *dot*$_{OclAny}\mathcal{ANY}$*-at-pre-strict* [*simp*] : (*invalid*).*any@pre* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)

lemma *dot*$_{Person}\mathcal{BOSS}$*-nullstrict* [*simp*]: (*null*).*boss* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma *dot*$_{Person}\mathcal{BOSS}$*-at-pre-nullstrict* [*simp*] : (*null*).*boss@pre* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma *dot*$_{Person}\mathcal{BOSS}$*-strict* [*simp*] : (*invalid*).*boss* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma *dot*$_{Person}\mathcal{BOSS}$*-at-pre-strict* [*simp*] : (*invalid*).*boss@pre* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)

lemma *dot*$_{Person}\mathcal{SALARY}$*-nullstrict* [*simp*]: (*null*).*salary* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma *dot*$_{Person}\mathcal{SALARY}$*-at-pre-nullstrict* [*simp*] : (*null*).*salary@pre* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma *dot*$_{Person}\mathcal{SALARY}$*-strict* [*simp*] : (*invalid*).*salary* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma *dot*$_{Person}\mathcal{SALARY}$*-at-pre-strict* [*simp*] : (*invalid*).*salary@pre* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)

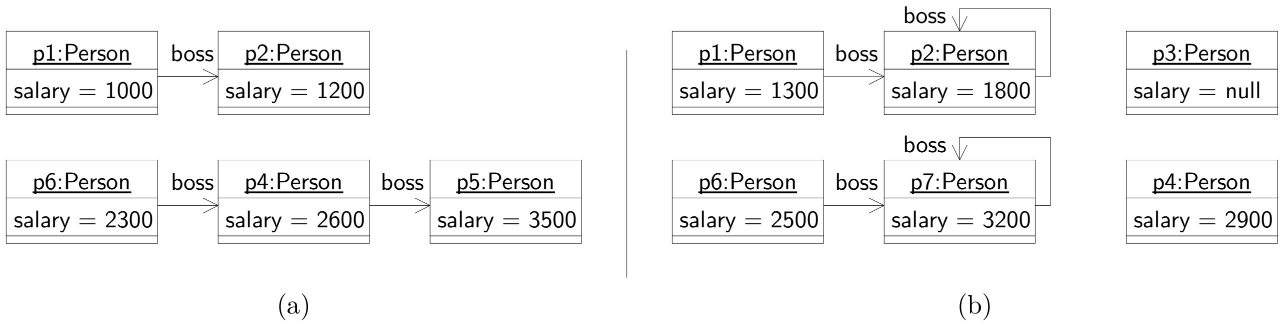


Figure B.2.: (a) pre-state σ_1 and (b) post-state σ_1'.

B.4.9. A Little Infra-structure on Example States

The example we are defining in this section comes from the figure B.2.

definition *OclInt1000* (**1000**) **where** *OclInt1000* = (λ - . $\lfloor\lfloor 1000 \rfloor\rfloor$)
definition *OclInt1200* (**1200**) **where** *OclInt1200* = (λ - . $\lfloor\lfloor 1200 \rfloor\rfloor$)
definition *OclInt1300* (**1300**) **where** *OclInt1300* = (λ - . $\lfloor\lfloor 1300 \rfloor\rfloor$)
definition *OclInt1800* (**1800**) **where** *OclInt1800* = (λ - . $\lfloor\lfloor 1800 \rfloor\rfloor$)
definition *OclInt2600* (**2600**) **where** *OclInt2600* = (λ - . $\lfloor\lfloor 2600 \rfloor\rfloor$)
definition *OclInt2900* (**2900**) **where** *OclInt2900* = (λ - . $\lfloor\lfloor 2900 \rfloor\rfloor$)
definition *OclInt3200* (**3200**) **where** *OclInt3200* = (λ - . $\lfloor\lfloor 3200 \rfloor\rfloor$)
definition *OclInt3500* (**3500**) **where** *OclInt3500* = (λ - . $\lfloor\lfloor 3500 \rfloor\rfloor$)

definition *oid0* ≡ *0*
definition *oid1* ≡ *1*
definition *oid2* ≡ *2*
definition *oid3* ≡ *3*
definition *oid4* ≡ *4*
definition *oid5* ≡ *5*
definition *oid6* ≡ *6*
definition *oid7* ≡ *7*
definition *oid8* ≡ *8*

definition *person1* ≡ mk_{Person} *oid0* $\lfloor 1300 \rfloor$
definition *person2* ≡ mk_{Person} *oid1* $\lfloor 1800 \rfloor$
definition *person3* ≡ mk_{Person} *oid2 None*
definition *person4* ≡ mk_{Person} *oid3* $\lfloor 2900 \rfloor$
definition *person5* ≡ mk_{Person} *oid4* $\lfloor 3500 \rfloor$
definition *person6* ≡ mk_{Person} *oid5* $\lfloor 2500 \rfloor$
definition *person7* ≡ mk_{OclAny} *oid6* $\lfloor\lfloor 3200 \rfloor\rfloor$
definition *person8* ≡ mk_{OclAny} *oid7 None*
definition *person9* ≡ mk_{Person} *oid8* $\lfloor 0 \rfloor$

definition
$\sigma_1 \equiv (\!|\ heap = empty(oid0 \mapsto in_{Person}\ (mk_{Person}\ oid0\ \lfloor 1000 \rfloor))$

$(oid1 \mapsto in_{Person}\ (mk_{Person}\ oid1\ \lfloor 1200 \rfloor))$
$(*oid2*)$
$(oid3 \mapsto in_{Person}\ (mk_{Person}\ oid3\ \lfloor 2600 \rfloor))$
$(oid4 \mapsto in_{Person}\ person5)$
$(oid5 \mapsto in_{Person}\ (mk_{Person}\ oid5\ \lfloor 2300 \rfloor))$
$(*oid6*)$
$(*oid7*)$
$(oid8 \mapsto in_{Person}\ person9)$,
$assocs = empty(oid_{Person}\mathcal{BOSS} \mapsto [[[oid0],[oid1]],[[oid3],[oid4]],[[oid5],[oid3]]])$ ⦈

definition
$\sigma_1{}' \equiv$ ⦇ $heap = empty(oid0 \mapsto in_{Person}\ person1)$
$(oid1 \mapsto in_{Person}\ person2)$
$(oid2 \mapsto in_{Person}\ person3)$
$(oid3 \mapsto in_{Person}\ person4)$
$(*oid4*)$
$(oid5 \mapsto in_{Person}\ person6)$
$(oid6 \mapsto in_{OclAny}\ person7)$
$(oid7 \mapsto in_{OclAny}\ person8)$
$(oid8 \mapsto in_{Person}\ person9)$,
$assocs = empty(oid_{Person}\mathcal{BOSS} \mapsto [[[oid0],[oid1]],[[oid1],[oid1]],[[oid5],[oid6]],[[oid6],[oid6]]])$ ⦈

definition $\sigma_0 \equiv$ ⦇ $heap = empty, assocs = empty$ ⦈

lemma *basic-τ-wff*: $WFF(\sigma_1,\sigma_1{}')$
by(*auto simp*: *WFF-def* σ_1*-def* $\sigma_1{}'$*-def*
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def
oid-of-𝔄-def $oid\text{-}of\text{-}type_{Person}$*-def* $oid\text{-}of\text{-}type_{OclAny}$*-def*
person1-def person2-def person3-def person4-def
person5-def person6-def person7-def person8-def person9-def)

lemma [*simp*,*code-unfold*]: $dom\ (heap\ \sigma_1) = \{oid0,oid1,(*,oid2*)oid3,oid4,oid5(*,oid6,oid7*),oid8\}$
by(*auto simp*: σ_1*-def*)

lemma [*simp*,*code-unfold*]: $dom\ (heap\ \sigma_1{}') = \{oid0,oid1,oid2,oid3,(*,oid4*)oid5,oid6,oid7,oid8\}$
by(*auto simp*: $\sigma_1{}'$*-def*)

definition $X_{Person}1 :: Person \equiv \lambda\ \text{-}\ .\lfloor\lfloor\ person1\ \rfloor\rfloor$
definition $X_{Person}2 :: Person \equiv \lambda\ \text{-}\ .\lfloor\lfloor\ person2\ \rfloor\rfloor$
definition $X_{Person}3 :: Person \equiv \lambda\ \text{-}\ .\lfloor\lfloor\ person3\ \rfloor\rfloor$
definition $X_{Person}4 :: Person \equiv \lambda\ \text{-}\ .\lfloor\lfloor\ person4\ \rfloor\rfloor$
definition $X_{Person}5 :: Person \equiv \lambda\ \text{-}\ .\lfloor\lfloor\ person5\ \rfloor\rfloor$
definition $X_{Person}6 :: Person \equiv \lambda\ \text{-}\ .\lfloor\lfloor\ person6\ \rfloor\rfloor$
definition $X_{Person}7 :: OclAny \equiv \lambda\ \text{-}\ .\lfloor\lfloor\ person7\ \rfloor\rfloor$
definition $X_{Person}8 :: OclAny \equiv \lambda\ \text{-}\ .\lfloor\lfloor\ person8\ \rfloor\rfloor$
definition $X_{Person}9 :: Person \equiv \lambda\ \text{-}\ .\lfloor\lfloor\ person9\ \rfloor\rfloor$

lemma [*code-unfold*]: $((x{::}Person) \doteq y) = StrictRefEq_{Object}\ x\ y$ **by**(*simp only*: $StrictRefEq_{Object}\text{-}_{Person}$)
lemma [*code-unfold*]: $((x{::}OclAny) \doteq y) = StrictRefEq_{Object}\ x\ y$ **by**(*simp only*: $StrictRefEq_{Object}\text{-}_{OclAny}$)

lemmas [*simp*,*code-unfold*] =
$OclAsType_{OclAny}$-*OclAny*
$OclAsType_{OclAny}$-*Person*
$OclAsType_{Person}$-*OclAny*
$OclAsType_{Person}$-*Person*

$OclIsTypeOf_{OclAny}$-*OclAny*
$OclIsTypeOf_{OclAny}$-*Person*
$OclIsTypeOf_{Person}$-*OclAny*
$OclIsTypeOf_{Person}$-*Person*

$OclIsKindOf_{OclAny}$-*OclAny*
$OclIsKindOf_{OclAny}$-*Person*
$OclIsKindOf_{Person}$-*OclAny*
$OclIsKindOf_{Person}$-*Person*

Assert $\bigwedge s_{pre} \quad . \quad (s_{pre},\sigma_1{}') \models \quad (X_{Person}1\ .salary \quad <> \mathbf{1000})$
Assert $\bigwedge s_{pre} \quad . \quad (s_{pre},\sigma_1{}') \models \quad (X_{Person}1\ .salary \quad \doteq \mathbf{1300})$
Assert $\bigwedge \quad s_{post}. \quad (\sigma_1,s_{post}) \models \quad (X_{Person}1\ .salary@pre \quad \doteq \mathbf{1000})$
Assert $\bigwedge \quad s_{post}. \quad (\sigma_1,s_{post}) \models \quad (X_{Person}1\ .salary@pre \quad <> \mathbf{1300})$

lemma $\qquad (\sigma_1,\sigma_1{}') \models \quad (X_{Person}1\ .oclIsMaintained())$
by(*simp add*: *OclValid-def OclIsMaintained-def*
σ_1-*def* $\sigma_1{}'$-*def*
$X_{Person}1$-*def person1-def*
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def
oid-of-option-def $oid\text{-}of\text{-}type_{Person}$-*def*)

lemma $\bigwedge s_{pre}\ s_{post}. \quad (s_{pre},s_{post}) \models \quad ((X_{Person}1\ .oclAsType(OclAny)\ .oclAsType(Person)) \doteq X_{Person}1)$
by(*rule up-down-cast-Person-OclAny-Person'*, *simp add*: $X_{Person}1$-*def*)
Assert $\bigwedge s_{pre}\ s_{post}. \quad (s_{pre},s_{post}) \models \quad (X_{Person}1\ .oclIsTypeOf(Person))$
Assert $\bigwedge s_{pre}\ s_{post}. \quad (s_{pre},s_{post}) \models not(X_{Person}1\ .oclIsTypeOf(OclAny))$
Assert $\bigwedge s_{pre}\ s_{post}. \quad (s_{pre},s_{post}) \models \quad (X_{Person}1\ .oclIsKindOf(Person))$
Assert $\bigwedge s_{pre}\ s_{post}. \quad (s_{pre},s_{post}) \models \quad (X_{Person}1\ .oclIsKindOf(OclAny))$
Assert $\bigwedge s_{pre}\ s_{post}. \quad (s_{pre},s_{post}) \models not(X_{Person}1\ .oclAsType(OclAny)\ .oclIsTypeOf(OclAny))$

Assert $\bigwedge s_{pre} \quad . \quad (s_{pre},\sigma_1{}') \models \quad (X_{Person}2\ .salary \quad \doteq \mathbf{1800})$
Assert $\bigwedge \quad s_{post}. \quad (\sigma_1,s_{post}) \models \quad (X_{Person}2\ .salary@pre \quad \doteq \mathbf{1200})$

lemma $\qquad (\sigma_1,\sigma_1{}') \models \quad (X_{Person}2\ .oclIsMaintained())$
by(*simp add*: *OclValid-def OclIsMaintained-def*
σ_1-*def* $\sigma_1{}'$-*def*
$X_{Person}2$-*def person2-def*

oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def
oid-of-option-def oid-of-type$_{Person}$-def)

Assert $\bigwedge s_{pre} \ . \ (s_{pre},\sigma_1{}') \models (X_{Person}3 \ .salary \doteq null)$
Assert $\bigwedge \ s_{post} . \ (\sigma_1,s_{post}) \models not(\upsilon(X_{Person}3 \ .salary@pre))$
lemma $(\sigma_1,\sigma_1{}') \models (X_{Person}3 \ .oclIsNew())$
by(*simp add*: *OclValid-def OclIsNew-def*
σ_1*-def* $\sigma_1{}'$*-def*
X_{Person}*3-def person3-def*
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid8-def
oid-of-option-def oid-of-type$_{Person}$-def)

lemma $(\sigma_1,\sigma_1{}') \models (X_{Person}4 \ .oclIsMaintained())$
by(*simp add*: *OclValid-def OclIsMaintained-def*
σ_1*-def* $\sigma_1{}'$*-def*
X_{Person}*4-def person4-def*
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def
oid-of-option-def oid-of-type$_{Person}$-def)

Assert $\bigwedge s_{pre} \ . \ (s_{pre},\sigma_1{}') \models not(\upsilon(X_{Person}5 \ .salary))$
Assert $\bigwedge \ s_{post} . \ (\sigma_1,s_{post}) \models (X_{Person}5 \ .salary@pre \doteq \mathbf{3500})$

lemma $(\sigma_1,\sigma_1{}') \models (X_{Person}5 \ .oclIsDeleted())$
by(*simp add*: *OclNot-def OclValid-def OclIsDeleted-def*
σ_1*-def* $\sigma_1{}'$*-def*
X_{Person}*5-def person5-def*
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def
oid-of-option-def oid-of-type$_{Person}$-def)

lemma $(\sigma_1,\sigma_1{}') \models (X_{Person}6 \ .oclIsMaintained())$
by(*simp add*: *OclValid-def OclIsMaintained-def*
σ_1*-def* $\sigma_1{}'$*-def*
X_{Person}*6-def person6-def*
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def
oid-of-option-def oid-of-type$_{Person}$-def)

Assert $\bigwedge s_{pre} \ s_{post} . \ (s_{pre},s_{post}) \models \upsilon(X_{Person}7 \ .oclAsType(Person))$

lemma $\bigwedge s_{pre}\ s_{post}.\ \ (s_{pre},s_{post}) \models\ \ ((X_{Person}7\ .oclAsType(Person)\ .oclAsType(OclAny)$
$.oclAsType(Person))$
$\doteq (X_{Person}7\ .oclAsType(Person)))$
by(*rule up-down-cast-Person-OclAny-Person′, simp add*: X_{Person}*7-def OclValid-def valid-def person7-def*)
lemma $(\sigma_1,\sigma_1{}') \models\ \ (X_{Person}7\ .oclIsNew())$
by(*simp add*: *OclValid-def OclIsNew-def*
σ_1*-def* $\sigma_1{}'$*-def*
X_{Person}*7-def person7-def*
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid8-def
oid-of-option-def oid-of-type$_{OclAny}$*-def*)

Assert $\bigwedge s_{pre}\ s_{post}.\ \ (s_{pre},s_{post}) \models\ \ (X_{Person}8\ <> X_{Person}7)$
Assert $\bigwedge s_{pre}\ s_{post}.\ \ (s_{pre},s_{post}) \models not(\upsilon(X_{Person}8\ .oclAsType(Person)))$
Assert $\bigwedge s_{pre}\ s_{post}.\ \ (s_{pre},s_{post}) \models\ \ (X_{Person}8\ .oclIsTypeOf(OclAny))$
Assert $\bigwedge s_{pre}\ s_{post}.\ \ (s_{pre},s_{post}) \models\ not(X_{Person}8\ .oclIsTypeOf(Person))$
Assert $\bigwedge s_{pre}\ s_{post}.\ \ (s_{pre},s_{post}) \models\ not(X_{Person}8\ .oclIsKindOf(Person))$
Assert $\bigwedge s_{pre}\ s_{post}.\ \ (s_{pre},s_{post}) \models\ \ (X_{Person}8\ .oclIsKindOf(OclAny))$

lemma σ*-modifiedonly*: $(\sigma_1,\sigma_1{}') \models (Set\{\ X_{Person}1\ .oclAsType(OclAny)$
$,X_{Person}2\ .oclAsType(OclAny)$
$(*, X_{Person}3\ .oclAsType(OclAny)*)$
$,X_{Person}4\ .oclAsType(OclAny)$
$(*, X_{Person}5\ .oclAsType(OclAny)*)$
$,X_{Person}6\ .oclAsType(OclAny)$
$(*, X_{Person}7\ .oclAsType(OclAny)*)$
$(*, X_{Person}8\ .oclAsType(OclAny)*)$
$(*, X_{Person}9\ .oclAsType(OclAny)*)\}{-}{>}oclIsModifiedOnly())$
apply(*simp add*: *OclIsModifiedOnly-def OclValid-def*
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def
X_{Person}*1-def* X_{Person}*2-def* X_{Person}*3-def* X_{Person}*4-def*
X_{Person}*5-def* X_{Person}*6-def* X_{Person}*7-def* X_{Person}*8-def* X_{Person}*9-def*
person1-def person2-def person3-def person4-def
person5-def person6-def person7-def person8-def person9-def
image-def)
apply(*simp add*: *OclIncluding-rep-set mtSet-rep-set null-option-def bot-option-def*)
apply(*simp add*: *oid-of-option-def oid-of-type*$_{OclAny}$*-def*, *clarsimp*)
apply(*simp add*: σ_1*-def* $\sigma_1{}'$*-def*
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)
done

lemma $(\sigma_1,\sigma_1{}') \models ((X_{Person}9\ @pre\ (\lambda x.\ \lfloor OclAsType_{Person}\text{-}\mathfrak{A}\ x\rfloor))\ \triangleq X_{Person}9)$
by(*simp add*: *OclSelf-at-pre-def* σ_1*-def oid-of-option-def oid-of-type*$_{Person}$*-def*
X_{Person}*9-def person9-def oid8-def OclValid-def StrongEq-def OclAsType*$_{Person}$-$\mathfrak{A}$*-def*)

lemma $(\sigma_1,\sigma_1{}') \models ((X_{Person}9\ @post\ (\lambda x.\ \lfloor OclAsType_{Person}\text{-}\mathfrak{A}\ x\rfloor))\ \triangleq X_{Person}9)$

by(*simp add*: *OclSelf-at-post-def* $\sigma_1{}'$*-def oid-of-option-def oid-of-type*$_{Person}$*-def*
    X_{Person}*9-def person9-def oid8-def OclValid-def StrongEq-def OclAsType*$_{Person}$*-*$\mathfrak{A}$*-def*)

lemma $(\sigma_1,\sigma_1{}') \models (((X_{Person}9\ .oclAsType(OclAny))\ @pre\ (\lambda x.\ \lfloor OclAsType_{OclAny}\text{-}\mathfrak{A}\ x\rfloor)) \triangleq$
    $((X_{Person}9\ .oclAsType(OclAny))\ @post\ (\lambda x.\ \lfloor OclAsType_{OclAny}\text{-}\mathfrak{A}\ x\rfloor)))$
proof −

have *including4* : $\bigwedge a\ b\ c\ d\ \tau.$
  $Set\{\lambda\tau.\ \lfloor\lfloor a\rfloor\rfloor, \lambda\tau.\ \lfloor\lfloor b\rfloor\rfloor, \lambda\tau.\ \lfloor\lfloor c\rfloor\rfloor, \lambda\tau.\ \lfloor\lfloor d\rfloor\rfloor\}\ \tau = Abs\text{-}Set_{base}\ \lfloor\lfloor\ \{\lfloor\lfloor a\rfloor\rfloor, \lfloor\lfloor b\rfloor\rfloor, \lfloor\lfloor c\rfloor\rfloor, \lfloor\lfloor d\rfloor\rfloor\}\ \rfloor\rfloor$
 apply(*subst abs-rep-simp*′[*symmetric*], *simp*)
 apply(*simp add*: *OclIncluding-rep-set mtSet-rep-set*)
 by(*rule arg-cong*[*of* - - $\lambda x.\ (Abs\text{-}Set_{base}(\lfloor\lfloor\ x\ \rfloor\rfloor))$], *auto*)

have *excluding1*: $\bigwedge S\ a\ b\ c\ d\ e\ \tau.$
    $(\lambda\text{-}.\ Abs\text{-}Set_{base}\ \lfloor\lfloor\ \{\lfloor\lfloor a\rfloor\rfloor, \lfloor\lfloor b\rfloor\rfloor, \lfloor\lfloor c\rfloor\rfloor, \lfloor\lfloor d\rfloor\rfloor\}\ \rfloor\rfloor){-}{>}excluding(\lambda\tau.\ \lfloor\lfloor e\rfloor\rfloor)\ \tau =$
    $Abs\text{-}Set_{base}\ \lfloor\lfloor\ \{\lfloor\lfloor a\rfloor\rfloor, \lfloor\lfloor b\rfloor\rfloor, \lfloor\lfloor c\rfloor\rfloor, \lfloor\lfloor d\rfloor\rfloor\} - \{\lfloor\lfloor e\rfloor\rfloor\}\ \rfloor\rfloor$
 apply(*simp add*: *OclExcluding-def*)
 apply(*simp add*: *defined-def OclValid-def false-def true-def*
    bot-fun-def bot-Set$_{base}$*-def null-fun-def null-Set*$_{base}$*-def*)
 apply(*rule conjI*)
  apply(*rule impI*, *subst* (*asm*) *Abs-Set*$_{base}$*-inject*) **apply**(*simp add*: *bot-option-def*)+
 apply(*rule conjI*)
  apply(*rule impI*, *subst* (*asm*) *Abs-Set*$_{base}$*-inject*) **apply**(*simp add*: *bot-option-def null-option-def*)+
 apply(*subst Abs-Set*$_{base}$*-inverse*, *simp add*: *bot-option-def*, *simp*)
done

show *?thesis*
 apply(*rule framing*[**where** $X = Set\{\ X_{Person}1\ .oclAsType(OclAny)$
     $,\ X_{Person}2\ .oclAsType(OclAny)$
    $(*,\ X_{Person}3\ .oclAsType(OclAny)*)$
     $,\ X_{Person}4\ .oclAsType(OclAny)$
    $(*,\ X_{Person}5\ .oclAsType(OclAny)*)$
     $,\ X_{Person}6\ .oclAsType(OclAny)$
    $(*,\ X_{Person}7\ .oclAsType(OclAny)*)$
    $(*,\ X_{Person}8\ .oclAsType(OclAny)*)$
    $(*,\ X_{Person}9\ .oclAsType(OclAny)*)\}$])
  apply(*cut-tac* σ*-modifiedonly*)
  apply(*simp only*: *OclValid-def*
    X_{Person}*1-def* X_{Person}*2-def* X_{Person}*3-def* X_{Person}*4-def*
    X_{Person}*5-def* X_{Person}*6-def* X_{Person}*7-def* X_{Person}*8-def* X_{Person}*9-def*
    person1-def person2-def person3-def person4-def
    person5-def person6-def person7-def person8-def person9-def
    OclAsType$_{OclAny}$*-Person*)
  apply(*subst cp-OclIsModifiedOnly*, *subst cp-OclExcluding*,
   subst (*asm*) *cp-OclIsModifiedOnly*, *simp add*: *including4 excluding1*)

 apply(*simp only*: X_{Person}*1-def* X_{Person}*2-def* X_{Person}*3-def* X_{Person}*4-def*
    X_{Person}*5-def* X_{Person}*6-def* X_{Person}*7-def* X_{Person}*8-def* X_{Person}*9-def*

person1-def person2-def person3-def person4-def
person5-def person6-def person7-def person8-def person9-def)
apply(*simp add*: *OclIncluding-rep-set mtSet-rep-set*
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)
apply(*simp add*: *StrictRefEq*$_{Object}$*-def oid-of-option-def oid-of-type*$_{OclAny}$*-def OclNot-def OclValid-def*
null-option-def bot-option-def)
done
qed

lemma *perm-*$\sigma_1{}'$: $\sigma_1{}'$ = ⦇ *heap* = *empty*
(*oid8* ↦ *in*$_{Person}$ *person9*)
(*oid7* ↦ *in*$_{OclAny}$ *person8*)
(*oid6* ↦ *in*$_{OclAny}$ *person7*)
(*oid5* ↦ *in*$_{Person}$ *person6*)
(∗*oid4*∗)
(*oid3* ↦ *in*$_{Person}$ *person4*)
(*oid2* ↦ *in*$_{Person}$ *person3*)
(*oid1* ↦ *in*$_{Person}$ *person2*)
(*oid0* ↦ *in*$_{Person}$ *person1*)
, *assocs* = *assocs* $\sigma_1{}'$ ⦈
proof −
note *P* = *fun-upd-twist*
show *?thesis*
apply(*simp add*: $\sigma_1{}'$*-def*
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)
apply(*subst* (*1*) *P*, *simp*)
apply(*subst* (*2*) *P*, *simp*) **apply**(*subst* (*1*) *P*, *simp*)
apply(*subst* (*3*) *P*, *simp*) **apply**(*subst* (*2*) *P*, *simp*) **apply**(*subst* (*1*) *P*, *simp*)
apply(*subst* (*4*) *P*, *simp*) **apply**(*subst* (*3*) *P*, *simp*) **apply**(*subst* (*2*) *P*, *simp*) **apply**(*subst* (*1*) *P*, *simp*)
apply(*subst* (*5*) *P*, *simp*) **apply**(*subst* (*4*) *P*, *simp*) **apply**(*subst* (*3*) *P*, *simp*) **apply**(*subst* (*2*) *P*, *simp*) **apply**(*subst* (*1*) *P*, *simp*)
apply(*subst* (*6*) *P*, *simp*) **apply**(*subst* (*5*) *P*, *simp*) **apply**(*subst* (*4*) *P*, *simp*) **apply**(*subst* (*3*) *P*, *simp*) **apply**(*subst* (*2*) *P*, *simp*) **apply**(*subst* (*1*) *P*, *simp*)
apply(*subst* (*7*) *P*, *simp*) **apply**(*subst* (*6*) *P*, *simp*) **apply**(*subst* (*5*) *P*, *simp*) **apply**(*subst* (*4*) *P*, *simp*) **apply**(*subst* (*3*) *P*, *simp*) **apply**(*subst* (*2*) *P*, *simp*) **apply**(*subst* (*1*) *P*, *simp*)
by(*simp*)
qed

declare *const-ss* [*simp*]

lemma $\bigwedge\sigma_1$.
$(\sigma_1,\sigma_1{}') \models (Person\ .allInstances() \doteq Set\{\ X_{Person}1, X_{Person}2, X_{Person}3, X_{Person}4(*, X_{Person}5*), X_{Person}6,$
$X_{Person}7\ .oclAsType(Person)(*, X_{Person}8*), X_{Person}9\ \})$
apply(*subst perm-*$\sigma_1{}'$)
apply(*simp only*: *oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def*
X_{Person}*1-def* X_{Person}*2-def* X_{Person}*3-def* X_{Person}*4-def*
X_{Person}*5-def* X_{Person}*6-def* X_{Person}*7-def* X_{Person}*8-def* X_{Person}*9-def*
person7-def)

apply(*subst state-update-vs-allInstances-at-post-tc*, *simp*, *simp add*: *OclAsType*$_{Person}$-$\mathfrak{A}$-*def*, *simp*, *rule const-StrictRefEq*$_{Set}$-*including*,
simp, *simp*, *simp*, *rule OclIncluding-cong*, *simp*, *simp*)
apply(*subst state-update-vs-allInstances-at-post-tc*, *simp*, *simp add*: *OclAsType*$_{Person}$-$\mathfrak{A}$-*def*, *simp*, *rule const-StrictRefEq*$_{Set}$-*including*
simp, *simp*, *simp*, *rule OclIncluding-cong*, *simp*, *simp*)
apply(*subst state-update-vs-allInstances-at-post-tc*, *simp*, *simp add*: *OclAsType*$_{Person}$-$\mathfrak{A}$-*def*, *simp*, *rule const-StrictRefEq*$_{Set}$-*including*
simp, *simp*, *simp*, *rule OclIncluding-cong*, *simp*, *simp*)
apply(*subst state-update-vs-allInstances-at-post-tc*, *simp*, *simp add*: *OclAsType*$_{Person}$-$\mathfrak{A}$-*def*, *simp*, *rule const-StrictRefEq*$_{Set}$-*includin*
simp, *simp*, *simp*, *rule OclIncluding-cong*, *simp*, *simp*)
apply(*subst state-update-vs-allInstances-at-post-tc*, *simp*, *simp add*: *OclAsType*$_{Person}$-$\mathfrak{A}$-*def*, *simp*, *rule const-StrictRefEq*$_{Set}$-*includin*
simp, *simp*, *simp*, *rule OclIncluding-cong*, *simp*, *simp*)
apply(*subst state-update-vs-allInstances-at-post-tc*, *simp*, *simp add*: *OclAsType*$_{Person}$-$\mathfrak{A}$-*def*, *simp*, *rule const-StrictRefEq*$_{Set}$-*includin*
simp, *simp*, *simp*, *rule OclIncluding-cong*, *simp*, *simp*)
apply(*subst state-update-vs-allInstances-at-post-ntc*, *simp*, *simp add*: *OclAsType*$_{Person}$-$\mathfrak{A}$-*def*
person8-def, *simp*, *rule const-StrictRefEq*$_{Set}$-*including*, *simp*, *simp*, *simp*)
apply(*subst state-update-vs-allInstances-at-post-tc*, *simp*, *simp add*: *OclAsType*$_{Person}$-$\mathfrak{A}$-*def*, *simp*, *rule const-StrictRefEq*$_{Set}$-*includi*
simp, *simp*, *simp*, *rule OclIncluding-cong*, *simp*, *simp*)
apply(*rule state-update-vs-allInstances-at-post-empty*)
by(*simp-all add*: *OclAsType*$_{Person}$-$\mathfrak{A}$-*def*)

lemma $\bigwedge\sigma_1$.
$(\sigma_1,\sigma_1{}') \models$ (*OclAny .allInstances*() $\doteq$ *Set*{ X_{Person}*1 .oclAsType*(*OclAny*), X_{Person}*2 .oclAsType*(*OclAny*),
X_{Person}*3 .oclAsType*(*OclAny*), X_{Person}*4 .oclAsType*(*OclAny*)
(∗, X_{Person}*5*∗), X_{Person}*6 .oclAsType*(*OclAny*),
X_{Person}*7*, X_{Person}*8*, X_{Person}*9 .oclAsType*(*OclAny*) })
apply(*subst perm*-$\sigma_1{}'$)
apply(*simp only*: *oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def*
X_{Person}*1-def* X_{Person}*2-def* X_{Person}*3-def* X_{Person}*4-def* X_{Person}*5-def* X_{Person}*6-def* X_{Person}*7-def* X_{Person}*8-def*
X_{Person}*9-def*
person1-def person2-def person3-def person4-def person5-def person6-def person9-def)
apply(*subst state-update-vs-allInstances-at-post-tc*, *simp*, *simp add*: *OclAsType*$_{OclAny}$-$\mathfrak{A}$-*def*, *simp*, *rule const-StrictRefEq*$_{Set}$-*including*
simp, *simp*, *simp*, *rule OclIncluding-cong*, *simp*, *simp*)+
apply(*rule state-update-vs-allInstances-at-post-empty*)
by(*simp-all add*: *OclAsType*$_{OclAny}$-$\mathfrak{A}$-*def*)

end

theory
Analysis-OCL
imports
Analysis-UML
begin

B.4.10. OCL Part: Standard State Infrastructure

Ideally, these definitions are automatically generated from the class model.

B.4.11. Invariant

These recursive predicates can be defined conservatively by greatest fix-point constructions—automatically. See [4, 5] for details. For the purpose of this example, we state them as axioms here.

```
context Person
  inv label : self .boss <> null implies (self .salary  \<le>  ((self .boss) .salary))
```

definition $\mathit{Person\text{-}label}_{inv} :: \mathit{Person} \Rightarrow \mathit{Boolean}$
where $\mathit{Person\text{-}label}_{inv}\ (\mathit{self}) \equiv$
$(\mathit{self}\ .\mathit{boss} <> \mathit{null\ implies}\ (\mathit{self}\ .\mathit{salary}\ \leq_{int}\ ((\mathit{self}\ .\mathit{boss})\ .\mathit{salary})))$

definition $\mathit{Person\text{-}label}_{invATpre} :: \mathit{Person} \Rightarrow \mathit{Boolean}$
where $\mathit{Person\text{-}label}_{invATpre}\ (\mathit{self}) \equiv$
$(\mathit{self}\ .\mathit{boss@pre} <> \mathit{null\ implies}\ (\mathit{self}\ .\mathit{salary@pre} \leq_{int} ((\mathit{self}\ .\mathit{boss@pre})\ .\mathit{salary@pre})))$

definition $\mathit{Person\text{-}label}_{globalinv} :: \mathit{Boolean}$
where $\mathit{Person\text{-}label}_{globalinv} \equiv (\mathit{Person}\ .\mathit{allInstances}()\text{--}{>}\mathit{forAll}(x \mid \mathit{Person\text{-}label}_{inv}\ (x))\ \mathit{and}$
$(\mathit{Person}\ .\mathit{allInstances@pre}()\text{--}{>}\mathit{forAll}(x \mid \mathit{Person\text{-}label}_{invATpre}\ (x))))$

lemma $\tau \models \delta\ (X\ .\mathit{boss}) \Longrightarrow \tau \models \mathit{Person}\ .\mathit{allInstances}()\text{--}{>}\mathit{includes}(X\ .\mathit{boss}) \wedge$
$\tau \models \mathit{Person}\ .\mathit{allInstances}()\text{--}{>}\mathit{includes}(X)$
sorry

lemma $\mathit{REC\text{-}pre} : \tau \models \mathit{Person\text{-}label}_{globalinv}$
$\Longrightarrow \tau \models \mathit{Person}\ .\mathit{allInstances}()\text{--}{>}\mathit{includes}(X)$ (* *X represented object in state* *)
$\Longrightarrow \exists\ \mathit{REC}.\ \ \tau \models \mathit{REC}(X)\ \triangleq (\mathit{Person\text{-}label}_{inv}\ (X)\ \mathit{and}\ (X\ .\mathit{boss} <> \mathit{null\ implies}\ \mathit{REC}(X\ .\mathit{boss})))$
sorry

This allows to state a predicate:

axiomatization $\mathit{inv}_{Person\text{-}label} :: \mathit{Person} \Rightarrow \mathit{Boolean}$
where $\mathit{inv}_{Person\text{-}label}\text{-}\mathit{def}$:
$(\tau \models \mathit{Person}\ .\mathit{allInstances}()\text{--}{>}\mathit{includes}(\mathit{self})) \Longrightarrow$
$(\tau \models (\mathit{inv}_{Person\text{-}label}(\mathit{self}) \triangleq\ (\mathit{self}\ .\mathit{boss} <> \mathit{null\ implies}$
$(\mathit{self}\ .\mathit{salary}\ \leq_{int}\ ((\mathit{self}\ .\mathit{boss})\ .\mathit{salary}))\ \mathit{and}$
$\mathit{inv}_{Person\text{-}label}(\mathit{self}\ .\mathit{boss}))))$

axiomatization $\mathit{inv}_{Person\text{-}labelATpre} :: \mathit{Person} \Rightarrow \mathit{Boolean}$
where $\mathit{inv}_{Person\text{-}labelATpre}\text{-}\mathit{def}$:
$(\tau \models \mathit{Person}\ .\mathit{allInstances@pre}()\text{--}{>}\mathit{includes}(\mathit{self})) \Longrightarrow$
$(\tau \models (\mathit{inv}_{Person\text{-}labelATpre}(\mathit{self}) \triangleq (\mathit{self}\ .\mathit{boss@pre} <> \mathit{null\ implies}$

$(self\ .salary@pre\ \leq_{int}\ ((self\ .boss@pre)\ .salary@pre))\ and$
$inv_{Person\text{-}labelATpre}(self\ .boss@pre))))$

lemma *inv-1* :
$(\tau \models Person\ .allInstances()->includes(self)) \Longrightarrow$
$(\tau \models inv_{Person\text{-}label}(self) = ((\tau \models (self\ .boss \doteq null)) \vee$
$(\ \tau \models (self\ .boss <> null) \wedge$
$\tau \models ((self\ .salary)\ \leq_{int}\ (self\ .boss\ .salary))\ \wedge$
$\tau \models (inv_{Person\text{-}label}(self\ .boss)))))$
sorry

lemma *inv-2* :
$(\tau \models Person\ .allInstances@pre()->includes(self)) \Longrightarrow$
$(\tau \models inv_{Person\text{-}labelATpre}(self)) =\ ((\tau \models (self\ .boss@pre \doteq null)) \vee$
$(\tau \models (self\ .boss@pre <> null) \wedge$
$(\tau \models (self\ .boss@pre\ .salary@pre \leq_{int} self\ .salary@pre))\ \wedge$
$(\tau \models (inv_{Person\text{-}labelATpre}(self\ .boss@pre)))))$
sorry

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 $inv :: Person \Rightarrow (\mathfrak{A})st \Rightarrow bool$ **where**
$(\tau \models (\delta\ self)) \Longrightarrow ((\tau \models (self\ .boss \doteq null)) \vee$
$(\tau \models (self\ .boss <> null) \wedge (\tau \models (self\ .boss\ .salary \leq_{int} self\ .salary))\ \wedge$
$(\ (inv(self\ .boss))\tau\ )))$
$\Longrightarrow (\ inv\ self\ \tau)$

B.4.12. The Contract of a Recursive Query

The original specification of a recursive query :

```
context Person::contents():Set(Integer)
pre:   true
post:  result = if self.boss = null
                then Set{i}
                else self.boss.contents()->including(i)
                endif
```

For the case of recursive queries, we use at present just axiomatizations:

axiomatization *contents* :: *Person* ⇒ *Set-Integer* *((1(-).contents'(')) 50)*
where *contents-def*:
$(self\ .contents()) = (\lambda\ \tau.\ (if\ \tau \models (\delta\ self)$
$then\ SOME\ res.((\tau \models true) \wedge$
$(\tau \models (\lambda\text{-}\ .\ res) \triangleq if\ (self\ .boss \doteq null)$
$then\ (Set\{self\ .salary\})$

```
                                    else (self .boss .contents()
                                          −>including(self .salary))
                                    endif))
                 else invalid τ))

interpretation contents : contract0 contents λ self. true
             λ self res.  res ≜ if (self .boss ≐ null)
                                    then (Set{self .salary})
                                    else (self .boss .contents()
                                          −>including(self .salary))
                                    endif
    proof (unfold-locales)
      show ⋀self τ. true τ = true τ by auto
    next
      show ⋀self. ∀σ σ' σ''. ((σ, σ') ⊨ true) = ((σ, σ'') ⊨ true) by auto
    next
      show ⋀self. self .contents() ≡
                λτ. if τ ⊨ δ self
                   then SOME res.
                         τ ⊨ true ∧
                         τ ⊨ (λ-. res) ≜ (if self .boss ≐ null then Set{self .salary}
                                 else self .boss .contents()−>including(self .salary)
                                 endif)
                   else invalid τ
           by(auto simp: contents-def )
    next
      have A:⋀self τ. ((λ-. self τ) .boss ≐ null) τ = (λ-. (self .boss ≐ null) τ) τ  sorry
      have B:⋀self τ. (λ-. Set{(λ-. self τ) .salary} τ) = (λ-. Set{self .salary} τ) sorry
      have C:⋀self τ. ((λ-. self τ).boss .contents()−>including((λ-. self τ).salary) τ) =
                  (self .boss .contents() −>including(self .salary) τ) sorry
      show ⋀self res τ.
           (res ≜ if (self .boss) ≐ null then Set{self .salary}
                 else self .boss .contents()−>including(self .salary) endif) τ =
           ((λ-. res τ) ≜ if (λ-. self τ) .boss ≐ null then Set{(λ-. self τ) .salary}
                       else(λ-. self τ) .boss .contents()−>including((λ-. self τ) .salary) endif) τ
     apply(subst cp-StrongEq)
     apply(subst (2) cp-StrongEq)
     apply(subst cp-OclIf)
     apply(subst (2)cp-OclIf)
     by(simp add: A B C)
    qed
```

Specializing $[\![cp\ E;\ \tau \models \delta\ self;\ \tau \models true;\ \tau \models POST'\ self;\ \bigwedge res.\ (res \triangleq if\ self.boss \doteq null\ then\ Set\{self.salary\}\ else\ self.boss.contents()\!-\!\!>\!including(self.salary)\ endif) = (POST'\ self\ and\ (res \triangleq BODY\ self))]\!] \Longrightarrow (\tau \models E\ (self.contents())) = (\tau \models E\ (BODY\ self))$, one gets the following more practical rewrite rule that is amenable to symbolic evaluation:

```
theorem unfold-contents :
  assumes cp E
  and     τ ⊨ δ self
  shows  (τ ⊨ E (self .contents())) =
         (τ ⊨ E (if self .boss ≐ null
              then Set{self .salary}
              else self .boss .contents()−>including(self .salary) endif))
by(rule contents.unfold2[of - - - λ X. true], simp-all add: assms)
```

Since we have only one interpretation function, we need the corresponding operation on the pre-state:

```
consts contentsATpre :: Person ⇒ Set-Integer  ((1(-).contents@pre'(')) 50)

axiomatization where contentsATpre-def:
 (self).contents@pre() = (λ τ.
    (if τ ⊨ (δ self)
     then SOME res.((τ ⊨ true) ∧                              (∗ pre ∗)
                (τ ⊨ ((λ-. res) ≜ if (self).boss@pre ≐ null  (∗ post ∗)
                                   then Set{(self).salary@pre}
                                   else (self).boss@pre .contents@pre()
                                          −>including(self .salary@pre)
                                   endif)))
     else invalid τ))

interpretation contentsATpre : contract0 contentsATpre λ self. true
                     λ self res.  res ≜ if (self .boss@pre ≐ null)
                                         then (Set{self .salary@pre})
                                         else (self .boss@pre .contents@pre()
                                                −>including(self .salary@pre))
                                         endif
     proof (unfold-locales)
       show ⋀self τ. true τ = true τ by auto
     next
       show ⋀self. ∀σ σ' σ''. ((σ, σ') ⊨ true) = ((σ, σ'') ⊨ true) by auto
     next
       show ⋀self. self .contents@pre() ≡
                λτ. if τ ⊨ δ self
                    then SOME res.
                           τ ⊨ true ∧
                           τ ⊨ (λ-. res) ≜ (if self .boss@pre ≐ null then Set{self .salary@pre}
                               else self .boss@pre .contents@pre()−>including(self .salary@pre)
                               endif)
                    else invalid τ
           by(auto simp: contentsATpre-def)
     next
       have A:⋀self τ. ((λ-. self τ) .boss@pre ≐ null) τ = (λ-. (self .boss@pre ≐ null) τ) τ sorry
       have B:⋀self τ. (λ-. Set{(λ-. self τ) .salary@pre} τ) = (λ-. Set{self .salary@pre} τ) sorry
       have C:⋀self τ. ((λ-. self τ).boss@pre .contents@pre()−>including((λ-. self τ).salary@pre) τ) =
                  (self .boss@pre .contents@pre() −>including(self .salary@pre) τ) sorry
       show ⋀self res τ.
            (res ≜ if (self .boss@pre) ≐ null then Set{self .salary@pre}
```

else self .boss@pre .contents@pre()−>including(self .salary@pre) endif) $\tau =$
$((\lambda\text{-}.\ \mathit{res}\ \tau) \triangleq \mathit{if}\ (\lambda\text{-}.\ \mathit{self}\ \tau)\ \mathit{.boss@pre} \doteq \mathit{null\ then\ Set}\{(\lambda\text{-}.\ \mathit{self}\ \tau)\ \mathit{.salary@pre}\}$
$\mathit{else}(\lambda\text{-}.\ \mathit{self}\ \tau)\ \mathit{.boss@pre\ .contents@pre()}{-}{>}\mathit{including}((\lambda\text{-}.\ \mathit{self}\ \tau)\ \mathit{.salary@pre})\ \mathit{endif})\ \tau$
apply(*subst cp-StrongEq*)
apply(*subst* (2) *cp-StrongEq*)
apply(*subst cp-OclIf*)
apply(*subst* (2)*cp-OclIf*)
by(*simp add*: *A B C*)
qed

Again, we derive via *contents.unfold2* a Knaster-Tarski like Fixpoint rule that is amenable to symbolic evaluation:

theorem *unfold-contentsATpre* :
assumes *cp E*
and $\tau \models \delta\ \mathit{self}$
shows $(\tau \models E\ (\mathit{self\ .contents@pre}())) =$
$(\tau \models E\ (\mathit{if\ self\ .boss@pre} \doteq \mathit{null}$
then Set{*self .salary@pre*}
else self .boss@pre .contents@pre()−>including(self .salary@pre) endif))
by(*rule contentsATpre.unfold2*[*of - - -* λ *X. true*], *simp-all add*: *assms*)

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

B.4.13. The Contract of a User-defined Method

The example specification in high-level OCL input syntax reads as follows:

```
context Person::insert(x:Integer)
pre: true
post: contents():Set(Integer)
contents() = contents@pre()->including(x)
```

This boils down to:

definition *insert* :: *Person* ⇒*Integer* ⇒ *Void* ((*1*(-).*insert*′(-′)) *50*)
where *self .insert*(*x*) ≡
$(\lambda\ \tau.\ \mathit{if}\ (\tau \models (\delta\ \mathit{self})) \wedge\ (\tau \models \upsilon\ x)$
$\mathit{then\ SOME\ res}.\ (\tau \models \mathit{true} \wedge$
$(\tau \models ((\mathit{self}).\mathit{contents}() \triangleq (\mathit{self}).\mathit{contents@pre}(){-}{>}\mathit{including}(x))))$
$\mathit{else\ invalid}\ \tau)$

The semantic consequences of this definition were computed inside this locale interpretation:

interpretation *insert* : *contract1 insert* λ *self x. true*
$\lambda\ \mathit{self\ x\ res}.\ ((\mathit{self\ .contents}()) \triangleq$
$(\mathit{self\ .contents@pre}(){-}{>}\mathit{including}(x)))$
apply *unfold-locales* **apply**(*auto simp*:*insert-def*)
apply(*subst cp-StrongEq*) **apply**(*subst* (2) *cp-StrongEq*)
apply(*subst contents.cp0*)
apply(*subst UML-Set.OclIncluding.cp0*)

apply(*subst* (2) *UML-Set.OclIncluding.cp0*)
apply(*subst contentsATpre.cp0*)
by(*simp*)

The result of this locale interpretation for our *Analysis-OCL.insert* contract is the following set of properties, which serves as basis for automated deduction on them:

Name	Theorem
insert.strict0	$(\mathit{invalid.insert}(X)) = \mathit{invalid}$
insert.nullstrict0	$(\mathit{null.insert}(X)) = \mathit{invalid}$
insert.strict1	$(\mathit{self.insert}(\mathit{invalid})) = \mathit{invalid}$
$\mathit{insert.cp}_{PRE}$	$\mathit{true}\ \tau = \mathit{true}\ \tau$
$\mathit{insert.cp}_{POST}$	$(\mathit{self.contents}() \triangleq \mathit{self.contents@pre}()-{>}\mathit{including}(\mathit{a1.0}))\ \tau = (\lambda\text{-}.\ \mathit{self}\ \tau.\mathit{contents}() \triangleq \lambda\text{-}.\ \mathit{self}\ \tau.\mathit{contents@pre}()-{>}\mathit{including}(\lambda\text{-}.\ \mathit{a1.0}\ \tau))\ \tau$
insert.cp-pre	$\llbracket \mathit{cp}\ \mathit{self}';\ \mathit{cp}\ \mathit{a1}' \rrbracket \Longrightarrow \mathit{cp}\ (\lambda X.\ \mathit{true})$
insert.cp-post	$\llbracket \mathit{cp}\ \mathit{self}';\ \mathit{cp}\ \mathit{a1}';\ \mathit{cp}\ \mathit{res}' \rrbracket \Longrightarrow \mathit{cp}\ (\lambda X.\ \mathit{self}'\ X.\mathit{contents}() \triangleq \mathit{self}'\ X.\mathit{contents@pre}()-{>}\mathit{including}(\mathit{a1}'\ X))$
insert.cp	$\llbracket \mathit{cp}\ \mathit{self}';\ \mathit{cp}\ \mathit{a1}';\ \mathit{cp}\ \mathit{res}' \rrbracket \Longrightarrow \mathit{cp}\ (\lambda X.\ \mathit{self}'\ X.\mathit{insert}(\mathit{a1}'\ X))$
insert.cp0	$(\mathit{self.insert}(\mathit{a1.0}))\ \tau = (\lambda\text{-}.\ \mathit{self}\ \tau.\mathit{insert}(\lambda\text{-}.\ \mathit{a1.0}\ \tau))\ \tau$
insert.def-scheme	$\mathit{self.insert}(\mathit{a1.0}) \equiv \lambda\tau.\ \mathit{if}\ \tau \models \delta\ \mathit{self} \land \tau \models \upsilon\ \mathit{a1.0}\ \mathit{then}\ \mathit{SOME}\ \mathit{res}.\ \tau \models \mathit{true} \land \tau \models \mathit{self.contents}() \triangleq \mathit{self.contents@pre}()-{>}\mathit{including}(\mathit{a1.0})\ \mathit{else}\ \mathit{invalid}\ \tau$
insert.unfold	$\llbracket \mathit{cp}\ E;\ \tau \models \delta\ \mathit{self} \land \tau \models \upsilon\ \mathit{a1.0};\ \tau \models \mathit{true};\ \exists \mathit{res}.\ \tau \models \mathit{self.contents}() \triangleq \mathit{self.contents@pre}()-{>}\mathit{including}(\mathit{a1.0});\ \bigwedge \mathit{res}.\ \tau \models \mathit{self.contents}() \triangleq \mathit{self.contents@pre}()-{>}\mathit{including}(\mathit{a1.0}) \Longrightarrow \tau \models E\ (\lambda\text{-}.\ \mathit{res}) \rrbracket \Longrightarrow \tau \models E\ (\mathit{self.insert}(\mathit{a1.0}))$
insert.unfold2	$\llbracket \mathit{cp}\ E;\ \tau \models \delta\ \mathit{self} \land \tau \models \upsilon\ \mathit{a1.0};\ \tau \models \mathit{true};\ \tau \models \mathit{POST}'\ \mathit{self}\ \mathit{a1.0};\ \bigwedge \mathit{res}.\ (\mathit{self.contents}() \triangleq \mathit{self.contents@pre}()-{>}\mathit{including}(\mathit{a1.0})) = (\mathit{POST}'\ \mathit{self}\ \mathit{a1.0}\ \mathit{and}\ (\mathit{res} \triangleq \mathit{BODY}\ \mathit{self}\ \mathit{a1.0})) \rrbracket \Longrightarrow (\tau \models E\ (\mathit{self.insert}(\mathit{a1.0}))) = (\tau \models E\ (\mathit{BODY}\ \mathit{self}\ \mathit{a1.0}))$

Table B.5.: Semantic properties resulting from a user-defined operation contract.

end

B.5. Example II: The Employee Design Model (UML)

theory
Design-UML
imports
../../../src/UML-Main
begin

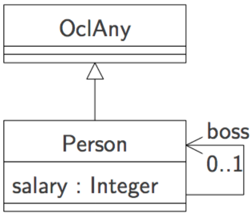


Figure B.3.: A simple UML class model drawn from Figure 7.3, page 20 of [28].

B.5.1. Introduction

For certain concepts like classes and class-types, only a generic definition for its resulting semantics can be given. Generic means, there is a function outside HOL that "compiles" a concrete, closed-world class diagram into a "theory" of this data model, consisting of a bunch of definitions for classes, accessors, method, casts, and tests for actual types, as well as proofs for the fundamental properties of these operations in this concrete data model.

Such generic function or "compiler" can be implemented in Isabelle on the ML level. This has been done, for a semantics following the open-world assumption, for UML 2.0 in [4, 6]. In this paper, we follow another approach for UML 2.4: we define the concepts of the compilation informally, and present a concrete example which is verified in Isabelle/HOL.

Outlining the Example

We are presenting here a "design-model" of the (slightly modified) example Figure 7.3, page 20 of the OCL standard [28]. To be precise, this theory contains the formalization of the data-part covered by the UML class model (see Figure B.3):

This means that the association (attached to the association class `EmployeeRanking`) with the association ends `boss` and `employees` is implemented by the attribute `boss` and the operation `employees` (to be discussed in the OCL part captured by the subsequent theory).

B.5.2. Example Data-Universe and its Infrastructure

Ideally, the following is generated automatically from a UML class model.

Our data universe consists in the concrete class diagram just of node's, and implicitly of the class object. Each class implies the existence of a class type defined for the corresponding object representations as follows:

datatype $type_{Person} = mk_{Person}\ oid$
$\quad int\ option$
$\quad oid\ option$

datatype $type_{OclAny} = mk_{OclAny}\ oid$
$\qquad (int\ option \times oid\ option)\ option$

Now, we construct a concrete "universe of OclAny types" by injection into a sum type containing the class types. This type of OclAny will be used as instance for all respective type-variables.

datatype $\mathfrak{A} = in_{Person}\ type_{Person} \mid in_{OclAny}\ type_{OclAny}$

Having fixed the object universe, we can introduce type synonyms that exactly correspond to OCL types. Again, we exploit that our representation of OCL is a "shallow embedding" with a one-to-one correspondance of OCL-types to types of the meta-language HOL.

type-synonym *Boolean* = $\mathfrak{A}$ *Boolean*
type-synonym *Integer* = $\mathfrak{A}$ *Integer*
type-synonym *Void* = $\mathfrak{A}$ *Void*
type-synonym *OclAny* = $(\mathfrak{A},\ type_{OclAny}\ option\ option)\ val$
type-synonym *Person* = $(\mathfrak{A},\ type_{Person}\ option\ option)\ val$
type-synonym *Set-Integer* = $(\mathfrak{A},\ int\ option\ option)\ Set$
type-synonym *Set-Person* = $(\mathfrak{A},\ type_{Person}\ option\ option)\ Set$

Just a little check:

typ *Boolean*

To reuse key-elements of the library like referential equality, we have to show that the object universe belongs to the type class "oclany," i. e., each class type has to provide a function *oid-of* yielding the object id (oid) of the object.

instantiation $type_{Person}$ :: *object*
begin
  definition *oid-of-type*$_{Person}$*-def*: $oid\text{-}of\ x = (case\ x\ of\ mk_{Person}\ oid\ \text{-}\ \text{-} \Rightarrow oid)$
  instance ..
end

instantiation $type_{OclAny}$ :: *object*
begin
  definition *oid-of-type*$_{OclAny}$*-def*: $oid\text{-}of\ x = (case\ x\ of\ mk_{OclAny}\ oid\ \text{-} \Rightarrow oid)$
  instance ..
end

instantiation $\mathfrak{A}$ :: *object*
begin
  definition *oid-of-*$\mathfrak{A}$*-def*: $oid\text{-}of\ x = (case\ x\ of$
    $in_{Person}\ person \Rightarrow oid\text{-}of\ person$
    $\mid in_{OclAny}\ oclany \Rightarrow oid\text{-}of\ oclany)$
  instance ..
end

B.5.3. Instantiation of the Generic Strict Equality

We instantiate the referential equality on *Person* and *OclAny*

defs(**overloaded**) $StrictRefEq_{Object}\text{-}_{Person}$: $(x::Person) \doteq y \ \equiv StrictRefEq_{Object}\ x\ y$
defs(**overloaded**) $StrictRefEq_{Object}\text{-}_{OclAny}$: $(x::OclAny) \doteq y \ \equiv StrictRefEq_{Object}\ x\ y$

lemmas
$cp\text{-}StrictRefEq_{Object}[of\ x::Person\ y::Person\ \tau,$
$simplified\ StrictRefEq_{Object}\text{-}_{Person}[symmetric]]$
$cp\text{-}intro(9)\quad [of\ P::Person \Rightarrow Person Q::Person \Rightarrow Person,$
$simplified\ StrictRefEq_{Object}\text{-}_{Person}[symmetric]\ ]$
$StrictRefEq_{Object}\text{-}def\quad [of\ x::Person\ y::Person,$
$simplified\ StrictRefEq_{Object}\text{-}_{Person}[symmetric]]$
$StrictRefEq_{Object}\text{-}defargs\ [of\ \text{-}\ x::Person\ y::Person,$
$simplified\ StrictRefEq_{Object}\text{-}_{Person}[symmetric]]$
$StrictRefEq_{Object}\text{-}strict1$
$[of\ x::Person,$
$simplified\ StrictRefEq_{Object}\text{-}_{Person}[symmetric]]$
$StrictRefEq_{Object}\text{-}strict2$
$[of\ x::Person,$
$simplified\ StrictRefEq_{Object}\text{-}_{Person}[symmetric]]$

For each Class *C*, we will have a casting operation `.oclAsType(`*C*`)`, a test on the actual type `.oclIsTypeOf(`*C*`)` as well as its relaxed form `.oclIsKindOf(`*C*`)` (corresponding exactly to Java's `instanceof`-operator.

Thus, since we have two class-types in our concrete class hierarchy, we have two operations to declare and to provide two overloading definitions for the two static types.

B.5.4. OclAsType

Definition

consts $OclAsType_{OclAny} :: {}'\alpha \Rightarrow OclAny\ ((\text{-})\ .oclAsType'(OclAny'))$
consts $OclAsType_{Person} :: {}'\alpha \Rightarrow Person\ ((\text{-})\ .oclAsType'(Person'))$

definition $OclAsType_{OclAny}\text{-}\mathfrak{A} = (\lambda u.\ \lfloor case\ u\ of\ in_{OclAny}\ a \Rightarrow a$
$\mid in_{Person}\ (mk_{Person}\ oid\ a\ b) \Rightarrow mk_{OclAny}\ oid\ \lfloor(a,b)\rfloor\rfloor)$

lemma $OclAsType_{OclAny}\text{-}\mathfrak{A}\text{-}some$: $OclAsType_{OclAny}\text{-}\mathfrak{A}\ x \neq None$
by$(simp\ add{:}\ OclAsType_{OclAny}\text{-}\mathfrak{A}\text{-}def)$

defs (**overloaded**) $OclAsType_{OclAny}\text{-}OclAny$:
$(X::OclAny)\ .oclAsType(OclAny) \equiv X$

defs (**overloaded**) $OclAsType_{OclAny}\text{-}Person$:
$(X::Person)\ .oclAsType(OclAny) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\perp\ \Rightarrow invalid\ \tau$

$\mid \lfloor\bot\rfloor \Rightarrow null\ \tau$
$\mid \lfloor\lfloor mk_{Person}\ oid\ a\ b\ \rfloor\rfloor \Rightarrow\ \lfloor\lfloor\ (mk_{OclAny}\ oid\ \lfloor(a,b)\rfloor)\ \rfloor\rfloor)$

definition $OclAsType_{Person}\text{-}\mathfrak{A} = (\lambda u.\ case\ u\ of\ in_{Person}\ p \Rightarrow \lfloor p\rfloor$
$\mid in_{OclAny}\ (mk_{OclAny}\ oid\ \lfloor(a,b)\rfloor) \Rightarrow \lfloor mk_{Person}\ oid\ a\ b\rfloor$
$\mid \text{-} \Rightarrow None)$

defs (**overloaded**) $OclAsType_{Person}\text{-}OclAny$:
$(X{::}OclAny)\ .oclAsType(Person) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\bot\ \ \Rightarrow invalid\ \tau$
$\mid \lfloor\bot\rfloor \Rightarrow null\ \tau$
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \bot\ \rfloor\rfloor \Rightarrow\ invalid\ \tau\quad (*\ down{-}cast\ exception\ *)$
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \lfloor(a,b)\rfloor\ \rfloor\rfloor \Rightarrow\ \lfloor\lfloor mk_{Person}\ oid\ a\ b\ \rfloor\rfloor)$

defs (**overloaded**) $OclAsType_{Person}\text{-}Person$:
$(X{::}Person)\ .oclAsType(Person) \equiv X$

lemmas [*simp*] =
$OclAsType_{OclAny}\text{-}OclAny$
$OclAsType_{Person}\text{-}Person$

Context Passing

lemma $cp\text{-}OclAsType_{OclAny}\text{-}Person\text{-}Person$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X{::}Person){::}Person)\ .oclAsType(OclAny))$
by(*rule cpI1*, *simp-all add*: $OclAsType_{OclAny}\text{-}Person$)
lemma $cp\text{-}OclAsType_{OclAny}\text{-}OclAny\text{-}OclAny$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X{::}OclAny){::}OclAny)\ .oclAsType(OclAny))$
by(*rule cpI1*, *simp-all add*: $OclAsType_{OclAny}\text{-}OclAny$)
lemma $cp\text{-}OclAsType_{Person}\text{-}Person\text{-}Person$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X{::}Person){::}Person)\ .oclAsType(Person))$
by(*rule cpI1*, *simp-all add*: $OclAsType_{Person}\text{-}Person$)
lemma $cp\text{-}OclAsType_{Person}\text{-}OclAny\text{-}OclAny$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X{::}OclAny){::}OclAny)\ .oclAsType(Person))$
by(*rule cpI1*, *simp-all add*: $OclAsType_{Person}\text{-}OclAny$)

lemma $cp\text{-}OclAsType_{OclAny}\text{-}Person\text{-}OclAny$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X{::}Person){::}OclAny)\ .oclAsType(OclAny))$
by(*rule cpI1*, *simp-all add*: $OclAsType_{OclAny}\text{-}OclAny$)
lemma $cp\text{-}OclAsType_{OclAny}\text{-}OclAny\text{-}Person$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X{::}OclAny){::}Person)\ .oclAsType(OclAny))$
by(*rule cpI1*, *simp-all add*: $OclAsType_{OclAny}\text{-}Person$)
lemma $cp\text{-}OclAsType_{Person}\text{-}Person\text{-}OclAny$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X{::}Person){::}OclAny)\ .oclAsType(Person))$
by(*rule cpI1*, *simp-all add*: $OclAsType_{Person}\text{-}OclAny$)
lemma $cp\text{-}OclAsType_{Person}\text{-}OclAny\text{-}Person$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X{::}OclAny){::}Person)\ .oclAsType(Person))$
by(*rule cpI1*, *simp-all add*: $OclAsType_{Person}\text{-}Person$)

lemmas [*simp*] =
$cp\text{-}OclAsType_{OclAny}\text{-}Person\text{-}Person$
$cp\text{-}OclAsType_{OclAny}\text{-}OclAny\text{-}OclAny$
$cp\text{-}OclAsType_{Person}\text{-}Person\text{-}Person$
$cp\text{-}OclAsType_{Person}\text{-}OclAny\text{-}OclAny$

cp-OclAsType$_{OclAny}$*-Person-OclAny*
cp-OclAsType$_{OclAny}$*-OclAny-Person*
cp-OclAsType$_{Person}$*-Person-OclAny*
cp-OclAsType$_{Person}$*-OclAny-Person*

Execution with Invalid or Null as Argument

lemma *OclAsType*$_{OclAny}$*-OclAny-strict* : (*invalid*::*OclAny*) .*oclAsType*(*OclAny*) = *invalid*
by(*simp*)

lemma *OclAsType*$_{OclAny}$*-OclAny-nullstrict* : (*null*::*OclAny*) .*oclAsType*(*OclAny*) = *null*
by(*simp*)

lemma *OclAsType*$_{OclAny}$*-Person-strict*[*simp*] : (*invalid*::*Person*) .*oclAsType*(*OclAny*) = *invalid*
by(*rule ext*, *simp add*: *bot-option-def invalid-def*
OclAsType$_{OclAny}$*-Person*)

lemma *OclAsType*$_{OclAny}$*-Person-nullstrict*[*simp*] : (*null*::*Person*) .*oclAsType*(*OclAny*) = *null*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def*
OclAsType$_{OclAny}$*-Person*)

lemma *OclAsType*$_{Person}$*-OclAny-strict*[*simp*] : (*invalid*::*OclAny*) .*oclAsType*(*Person*) = *invalid*
by(*rule ext*, *simp add*: *bot-option-def invalid-def*
OclAsType$_{Person}$*-OclAny*)

lemma *OclAsType*$_{Person}$*-OclAny-nullstrict*[*simp*] : (*null*::*OclAny*) .*oclAsType*(*Person*) = *null*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def*
OclAsType$_{Person}$*-OclAny*)

lemma *OclAsType*$_{Person}$*-Person-strict* : (*invalid*::*Person*) .*oclAsType*(*Person*) = *invalid*
by(*simp*)
lemma *OclAsType*$_{Person}$*-Person-nullstrict* : (*null*::*Person*) .*oclAsType*(*Person*) = *null*
by(*simp*)

B.5.5. OclIsTypeOf

Definition

consts *OclIsTypeOf*$_{OclAny}$ :: $'\alpha \Rightarrow$ *Boolean* ((-).*oclIsTypeOf*'(*OclAny*'))
consts *OclIsTypeOf*$_{Person}$ :: $'\alpha \Rightarrow$ *Boolean* ((-).*oclIsTypeOf*'(*Person*'))

defs (**overloaded**) *OclIsTypeOf*$_{OclAny}$*-OclAny*:
(*X*::*OclAny*) .*oclIsTypeOf*(*OclAny*) $\equiv$
($\lambda\tau$. *case X* τ *of*
$\bot \ \Rightarrow$ *invalid* τ
$|\ \lfloor\bot\rfloor \Rightarrow$ *true* τ ($*$ *invalid ??* $*$)
$|\ \lfloor\lfloor mk_{OclAny}\ oid\ \bot\ \rfloor\rfloor \Rightarrow$ *true* τ

$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \lfloor\text{-}\rfloor\ \rfloor\rfloor \Rightarrow false\ \tau)$

defs (**overloaded**) *OclIsTypeOf*$_{OclAny}$*-Person*:
$(X::Person)\ .oclIsTypeOf(OclAny) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\bot\ \ \Rightarrow invalid\ \tau$
$\mid \lfloor\bot\rfloor \Rightarrow true\ \tau \quad (*\ invalid\ ??\ *)$
$\mid \lfloor\lfloor\ \text{-}\ \rfloor\rfloor \Rightarrow false\ \tau)$

defs (**overloaded**) *OclIsTypeOf*$_{Person}$*-OclAny*:
$(X::OclAny)\ .oclIsTypeOf(Person) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\bot\ \ \Rightarrow invalid\ \tau$
$\mid \lfloor\bot\rfloor \Rightarrow true\ \tau$
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \bot\ \rfloor\rfloor \Rightarrow false\ \tau$
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \lfloor\text{-}\rfloor\ \rfloor\rfloor \Rightarrow true\ \tau)$

defs (**overloaded**) *OclIsTypeOf*$_{Person}$*-Person*:
$(X::Person)\ .oclIsTypeOf(Person) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\bot \Rightarrow invalid\ \tau$
$\mid \text{-} \Rightarrow true\ \tau)$

Context Passing

lemma *cp-OclIsTypeOf*$_{OclAny}$*-Person-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::Person)::Person).oclIsTypeOf(OclAny))$
by(*rule cpI1*, *simp-all add*: *OclIsTypeOf*$_{OclAny}$*-Person*)
lemma *cp-OclIsTypeOf*$_{OclAny}$*-OclAny-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::OclAny).oclIsTypeOf(OclAny))$
by(*rule cpI1*, *simp-all add*: *OclIsTypeOf*$_{OclAny}$*-OclAny*)
lemma *cp-OclIsTypeOf*$_{Person}$*-Person-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::Person)::Person).oclIsTypeOf(Person))$
by(*rule cpI1*, *simp-all add*: *OclIsTypeOf*$_{Person}$*-Person*)
lemma *cp-OclIsTypeOf*$_{Person}$*-OclAny-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::OclAny).oclIsTypeOf(Person))$
by(*rule cpI1*, *simp-all add*: *OclIsTypeOf*$_{Person}$*-OclAny*)

lemma *cp-OclIsTypeOf*$_{OclAny}$*-Person-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::Person)::OclAny).oclIsTypeOf(OclAny))$
by(*rule cpI1*, *simp-all add*: *OclIsTypeOf*$_{OclAny}$*-OclAny*)
lemma *cp-OclIsTypeOf*$_{OclAny}$*-OclAny-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::Person).oclIsTypeOf(OclAny))$
by(*rule cpI1*, *simp-all add*: *OclIsTypeOf*$_{OclAny}$*-Person*)
lemma *cp-OclIsTypeOf*$_{Person}$*-Person-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::Person)::OclAny).oclIsTypeOf(Person))$
by(*rule cpI1*, *simp-all add*: *OclIsTypeOf*$_{Person}$*-OclAny*)
lemma *cp-OclIsTypeOf*$_{Person}$*-OclAny-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::Person).oclIsTypeOf(Person))$
by(*rule cpI1*, *simp-all add*: *OclIsTypeOf*$_{Person}$*-Person*)

lemmas [*simp*] =
cp-OclIsTypeOf$_{OclAny}$*-Person-Person*
cp-OclIsTypeOf$_{OclAny}$*-OclAny-OclAny*

cp-OclIsTypeOf$_{Person}$*-Person-Person*
cp-OclIsTypeOf$_{Person}$*-OclAny-OclAny*

cp-OclIsTypeOf$_{OclAny}$*-Person-OclAny*
cp-OclIsTypeOf$_{OclAny}$*-OclAny-Person*
cp-OclIsTypeOf$_{Person}$*-Person-OclAny*
cp-OclIsTypeOf$_{Person}$*-OclAny-Person*

Execution with Invalid or Null as Argument

lemma *OclIsTypeOf*$_{OclAny}$*-OclAny-strict1*[*simp*]:
  (*invalid*::*OclAny*) .*oclIsTypeOf*(*OclAny*) = *invalid*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
    OclIsTypeOf$_{OclAny}$*-OclAny*)
lemma *OclIsTypeOf*$_{OclAny}$*-OclAny-strict2*[*simp*]:
  (*null*::*OclAny*) .*oclIsTypeOf*(*OclAny*) = *true*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
    OclIsTypeOf$_{OclAny}$*-OclAny*)
lemma *OclIsTypeOf*$_{OclAny}$*-Person-strict1*[*simp*]:
  (*invalid*::*Person*) .*oclIsTypeOf*(*OclAny*) = *invalid*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
    OclIsTypeOf$_{OclAny}$*-Person*)
lemma *OclIsTypeOf*$_{OclAny}$*-Person-strict2*[*simp*]:
  (*null*::*Person*) .*oclIsTypeOf*(*OclAny*) = *true*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
    OclIsTypeOf$_{OclAny}$*-Person*)
lemma *OclIsTypeOf*$_{Person}$*-OclAny-strict1*[*simp*]:
  (*invalid*::*OclAny*) .*oclIsTypeOf*(*Person*) = *invalid*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
    OclIsTypeOf$_{Person}$*-OclAny*)
lemma *OclIsTypeOf*$_{Person}$*-OclAny-strict2*[*simp*]:
  (*null*::*OclAny*) .*oclIsTypeOf*(*Person*) = *true*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
    OclIsTypeOf$_{Person}$*-OclAny*)
lemma *OclIsTypeOf*$_{Person}$*-Person-strict1*[*simp*]:
  (*invalid*::*Person*) .*oclIsTypeOf*(*Person*) = *invalid*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
    OclIsTypeOf$_{Person}$*-Person*)
lemma *OclIsTypeOf*$_{Person}$*-Person-strict2*[*simp*]:
  (*null*::*Person*) .*oclIsTypeOf*(*Person*) = *true*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
    OclIsTypeOf$_{Person}$*-Person*)

Up Down Casting

lemma *actualType-larger-staticType*:
assumes *isdef*: $\tau \models (\delta\ X)$
shows $\tau \models$ (*X*::*Person*) .*oclIsTypeOf*(*OclAny*) $\triangleq$ *false*

using *isdef*
by(*auto simp* : *null-option-def bot-option-def*
OclIsTypeOf$_{OclAny}$*-Person foundation22 foundation16*)

lemma *down-cast-type*:
assumes *isOclAny*: $\tau \models$ (*X*::*OclAny*) *.oclIsTypeOf*(*OclAny*)
and *non-null*: $\tau \models (\delta\ X)$
shows $\tau \models$ (*X .oclAsType*(*Person*)) $\triangleq$ *invalid*
using *isOclAny non-null*
apply(*auto simp* : *bot-fun-def null-fun-def null-option-def bot-option-def null-def invalid-def*
OclAsType$_{OclAny}$*-Person OclAsType*$_{Person}$*-OclAny foundation22 foundation16*
split: *option.split type*$_{OclAny}$*.split type*$_{Person}$*.split*)
by(*simp add*: *OclIsTypeOf*$_{OclAny}$*-OclAny OclValid-def false-def true-def*)

lemma *down-cast-type*$'$:
assumes *isOclAny*: $\tau \models$ (*X*::*OclAny*) *.oclIsTypeOf*(*OclAny*)
and *non-null*: $\tau \models (\delta\ X)$
shows $\tau \models$ *not* (υ (*X .oclAsType*(*Person*)))
by(*rule foundation15*[*THEN iffD1*], *simp add*: *down-cast-type*[*OF assms*])

lemma *up-down-cast* :
assumes *isdef*: $\tau \models (\delta\ X)$
shows $\tau \models$ ((*X*::*Person*) *.oclAsType*(*OclAny*) *.oclAsType*(*Person*) $\triangleq$ *X*)
using *isdef*
by(*auto simp* : *null-fun-def null-option-def bot-option-def null-def invalid-def*
OclAsType$_{OclAny}$*-Person OclAsType*$_{Person}$*-OclAny foundation22 foundation16*
split: *option.split type*$_{Person}$*.split*)

lemma *up-down-cast-Person-OclAny-Person* [*simp*]:
shows ((*X*::*Person*) *.oclAsType*(*OclAny*) *.oclAsType*(*Person*) = *X*)
apply(*rule ext*, *rename-tac* τ)
apply(*rule foundation22*[*THEN iffD1*])
apply(*case-tac* $\tau \models (\delta\ X)$, *simp add*: *up-down-cast*)
apply(*simp add*: *defined-split*, *elim disjE*)
apply(*erule StrongEq-L-subst2-rev*, *simp*, *simp*)+
done

lemma *up-down-cast-Person-OclAny-Person*$'$: **assumes** $\tau \models \upsilon\ X$
shows $\tau \models$ (((*X* :: *Person*) *.oclAsType*(*OclAny*) *.oclAsType*(*Person*)) $\doteq$ *X*)
apply(*simp only*: *up-down-cast-Person-OclAny-Person StrictRefEq*$_{Object}$*-*$_{Person}$)
by(*rule StrictRefEq*$_{Object}$*-sym*, *simp add*: *assms*)

lemma *up-down-cast-Person-OclAny-Person*$''$: **assumes** $\tau \models \upsilon$ (*X* :: *Person*)
shows $\tau \models$ (*X .oclIsTypeOf*(*Person*) *implies* (*X .oclAsType*(*OclAny*) *.oclAsType*(*Person*)) $\doteq$ *X*)
apply(*simp add*: *OclValid-def*)
apply(*subst cp-OclImplies*)
apply(*simp add*: *StrictRefEq*$_{Object}$*-*$_{Person}$ *StrictRefEq*$_{Object}$*-sym*[*OF assms*, *simplified OclValid-def*])

```
 apply(subst cp-OclImplies[symmetric])
by (simp add: OclImplies-true)
```

B.5.6. OclIsKindOf

Definition

```
consts OclIsKindOf_OclAny :: 'α ⇒ Boolean ((-).oclIsKindOf'(OclAny'))
consts OclIsKindOf_Person :: 'α ⇒ Boolean ((-).oclIsKindOf'(Person'))

defs (overloaded) OclIsKindOf_OclAny-OclAny:
    (X::OclAny) .oclIsKindOf(OclAny) ≡
            (λτ. case X τ of
                     ⊥ ⇒ invalid τ
                   | - ⇒ true τ)

defs (overloaded) OclIsKindOf_OclAny-Person:
    (X::Person) .oclIsKindOf(OclAny) ≡
            (λτ. case X τ of
                     ⊥ ⇒ invalid τ
                   | -⇒ true τ)



defs (overloaded) OclIsKindOf_Person-OclAny:
    (X::OclAny) .oclIsKindOf(Person) ≡
            (λτ. case X τ of
                     ⊥   ⇒ invalid τ
                   | ⌊⊥⌋ ⇒ true τ
                   | ⌊⌊mk_OclAny oid ⊥ ⌋⌋ ⇒ false τ
                   | ⌊⌊mk_OclAny oid ⌊-⌋ ⌋⌋ ⇒ true τ)

defs (overloaded) OclIsKindOf_Person-Person:
    (X::Person) .oclIsKindOf(Person) ≡
            (λτ. case X τ of
                     ⊥ ⇒ invalid τ
                   | - ⇒ true τ)
```

Context Passing

```
lemma cp-OclIsKindOf_OclAny-Person-Person: cp P ⟹ cp(λX.(P(X::Person)::Person).oclIsKindOf(OclAny))
by(rule cpI1, simp-all add: OclIsKindOf_OclAny-Person)
lemma cp-OclIsKindOf_OclAny-OclAny-OclAny: cp P ⟹ cp(λX.(P(X::OclAny)::OclAny).oclIsKindOf(OclAny))
by(rule cpI1, simp-all add: OclIsKindOf_OclAny-OclAny)
lemma cp-OclIsKindOf_Person-Person-Person: cp P ⟹ cp(λX.(P(X::Person)::Person).oclIsKindOf(Person))
by(rule cpI1, simp-all add: OclIsKindOf_Person-Person)
lemma cp-OclIsKindOf_Person-OclAny-OclAny: cp P ⟹ cp(λX.(P(X::OclAny)::OclAny).oclIsKindOf(Person))
by(rule cpI1, simp-all add: OclIsKindOf_Person-OclAny)
```

lemma *cp-OclIsKindOf*$_{OclAny}$*-Person-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(OclAny))$
by(*rule cpI1*, *simp-all add*: *OclIsKindOf*$_{OclAny}$*-OclAny*)
lemma *cp-OclIsKindOf*$_{OclAny}$*-OclAny-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(OclAny))$
by(*rule cpI1*, *simp-all add*: *OclIsKindOf*$_{OclAny}$*-Person*)
lemma *cp-OclIsKindOf*$_{Person}$*-Person-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(Person))$
by(*rule cpI1*, *simp-all add*: *OclIsKindOf*$_{Person}$*-OclAny*)
lemma *cp-OclIsKindOf*$_{Person}$*-OclAny-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(Person))$
by(*rule cpI1*, *simp-all add*: *OclIsKindOf*$_{Person}$*-Person*)

lemmas [*simp*] =
cp-OclIsKindOf$_{OclAny}$*-Person-Person*
cp-OclIsKindOf$_{OclAny}$*-OclAny-OclAny*
cp-OclIsKindOf$_{Person}$*-Person-Person*
cp-OclIsKindOf$_{Person}$*-OclAny-OclAny*

cp-OclIsKindOf$_{OclAny}$*-Person-OclAny*
cp-OclIsKindOf$_{OclAny}$*-OclAny-Person*
cp-OclIsKindOf$_{Person}$*-Person-OclAny*
cp-OclIsKindOf$_{Person}$*-OclAny-Person*

Execution with Invalid or Null as Argument

lemma *OclIsKindOf*$_{OclAny}$*-OclAny-strict1*[*simp*] : $(invalid::OclAny)\ .oclIsKindOf(OclAny) = invalid$
by(*rule ext*, *simp add*: *invalid-def bot-option-def*
OclIsKindOf$_{OclAny}$*-OclAny*)

lemma *OclIsKindOf*$_{OclAny}$*-OclAny-strict2*[*simp*] : $(null::OclAny)\ .oclIsKindOf(OclAny) = true$
by(*rule ext*, *simp add*: *null-fun-def null-option-def*
OclIsKindOf$_{OclAny}$*-OclAny*)

lemma *OclIsKindOf*$_{OclAny}$*-Person-strict1*[*simp*] : $(invalid::Person)\ .oclIsKindOf(OclAny) = invalid$
by(*rule ext*, *simp add*: *bot-option-def invalid-def*
OclIsKindOf$_{OclAny}$*-Person*)

lemma *OclIsKindOf*$_{OclAny}$*-Person-strict2*[*simp*] : $(null::Person)\ .oclIsKindOf(OclAny) = true$
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def*
OclIsKindOf$_{OclAny}$*-Person*)

lemma *OclIsKindOf*$_{Person}$*-OclAny-strict1*[*simp*]: $(invalid::OclAny)\ .oclIsKindOf(Person) = invalid$
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
OclIsKindOf$_{Person}$*-OclAny*)

lemma *OclIsKindOf*$_{Person}$*-OclAny-strict2*[*simp*]: $(null::OclAny)\ .oclIsKindOf(Person) = true$
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
OclIsKindOf$_{Person}$*-OclAny*)

lemma *OclIsKindOf*$_{Person}$*-Person-strict1*[*simp*]: $(invalid::Person)\ .oclIsKindOf(Person) = invalid$

by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
$OclIsKindOf_{Person}$*-Person*)

lemma $OclIsKindOf_{Person}$*-Person-strict2*[*simp*]: (*null*::*Person*) .*oclIsKindOf*(*Person*) = *true*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def null-def invalid-def*
$OclIsKindOf_{Person}$*-Person*)

Up Down Casting

lemma *actualKind-larger-staticKind*:
assumes *isdef*: $\tau \models (\delta\ X)$
shows $\tau \models ((X::Person)\ .oclIsKindOf(OclAny) \triangleq true)$
using *isdef*
by(*auto simp* : *bot-option-def*
$OclIsKindOf_{OclAny}$*-Person foundation22 foundation16*)

lemma *down-cast-kind*:
assumes *isOclAny*: $\neg\ (\tau \models ((X::OclAny).oclIsKindOf(Person)))$
and *non-null*: $\tau \models (\delta\ X)$
shows $\tau \models ((X\ .oclAsType(Person)) \triangleq invalid)$
using *isOclAny non-null*
apply(*auto simp* : *bot-fun-def null-fun-def null-option-def bot-option-def null-def invalid-def*
$OclAsType_{OclAny}$*-Person* $OclAsType_{Person}$*-OclAny foundation22 foundation16*
split: *option.split* $type_{OclAny}$*.split* $type_{Person}$*.split*)
by(*simp add*: $OclIsKindOf_{Person}$*-OclAny OclValid-def false-def true-def*)

B.5.7. OclAllInstances

To denote OCL-types occuring in OCL expressions syntactically—as, for example, as "argument" of oclAllInstances ()—we use the inverses of the injection functions into the object universes; we show that this is sufficient "characterization."

definition *Person* ≡ $OclAsType_{Person}$-𝔄
definition *OclAny* ≡ $OclAsType_{OclAny}$-𝔄
lemmas [*simp*] = *Person-def OclAny-def*

lemma $OclAllInstances\text{-}generic_{OclAny}$*-exec*: *OclAllInstances-generic pre-post OclAny* =
$(\lambda\tau.\ Abs\text{-}Set_{base}\ \lfloor\lfloor\ Some\ \text{`}\ OclAny\ \text{`}\ ran\ (heap\ (pre\text{-}post\ \tau))\ \rfloor\rfloor)$
proof –
let $?S1 = \lambda\tau.\ OclAny\ \text{`}\ ran\ (heap\ (pre\text{-}post\ \tau))$
let $?S2 = \lambda\tau.\ ?S1\ \tau - \{None\}$
have $B : \bigwedge\tau.\ ?S2\ \tau \subseteq ?S1\ \tau$ **by** *auto*
have $C : \bigwedge\tau.\ ?S1\ \tau \subseteq ?S2\ \tau$ **by**(*auto simp*: $OclAsType_{OclAny}$-𝔄*-some*)

show *?thesis* **by**(*insert equalityI*[*OF B C*], *simp*)
qed

lemma $OclAllInstances\text{-}at\text{-}post_{OclAny}$*-exec*: *OclAny* .*allInstances*() =

($\lambda\tau$. *Abs-Set*$_{base}$ ⌊⌊ *Some* ‘ *OclAny* ‘ *ran* (*heap* (*snd* τ)) ⌋⌋)
unfolding *OclAllInstances-at-post-def*
by(*rule OclAllInstances-generic*$_{OclAny}$*-exec*)

lemma *OclAllInstances-at-pre*$_{OclAny}$*-exec*: *OclAny .allInstances@pre*() =
($\lambda\tau$. *Abs-Set*$_{base}$ ⌊⌊ *Some* ‘ *OclAny* ‘ *ran* (*heap* (*fst* τ)) ⌋⌋)
unfolding *OclAllInstances-at-pre-def*
by(*rule OclAllInstances-generic*$_{OclAny}$*-exec*)

OclIsTypeOf

lemma *OclAny-allInstances-generic-oclIsTypeOf*$_{OclAny}$*1*:
assumes [*simp*]: $\bigwedge x$. *pre-post* (*x*, *x*) = *x*
shows $\exists\,\tau$. ($\tau \models$ ((*OclAllInstances-generic pre-post OclAny*)−>*forAll*(*X*|*X .oclIsTypeOf*(*OclAny*))))
apply(*rule-tac x* = τ_0 **in** *exI*, *simp add*: τ_0*-def OclValid-def del*: *OclAllInstances-generic-def*)
apply(*simp only*: *assms OclForall-def refl if-True*
OclAllInstances-generic-defined[*simplified OclValid-def*])
apply(*simp only*: *OclAllInstances-generic-def*)
apply(*subst* (*1 2 3*) *Abs-Set*$_{base}$*-inverse*, *simp add*: *bot-option-def*)
by(*simp add*: *OclIsTypeOf*$_{OclAny}$*-OclAny*)

lemma *OclAny-allInstances-at-post-oclIsTypeOf*$_{OclAny}$*1*:
$\exists\,\tau$. ($\tau \models$ (*OclAny .allInstances*()−>*forAll*(*X*|*X .oclIsTypeOf*(*OclAny*))))
unfolding *OclAllInstances-at-post-def*
by(*rule OclAny-allInstances-generic-oclIsTypeOf*$_{OclAny}$*1*, *simp*)

lemma *OclAny-allInstances-at-pre-oclIsTypeOf*$_{OclAny}$*1*:
$\exists\,\tau$. ($\tau \models$ (*OclAny .allInstances@pre*()−>*forAll*(*X*|*X .oclIsTypeOf*(*OclAny*))))
unfolding *OclAllInstances-at-pre-def*
by(*rule OclAny-allInstances-generic-oclIsTypeOf*$_{OclAny}$*1*, *simp*)

lemma *OclAny-allInstances-generic-oclIsTypeOf*$_{OclAny}$*2*:
assumes [*simp*]: $\bigwedge x$. *pre-post* (*x*, *x*) = *x*
shows $\exists\,\tau$. ($\tau \models$ *not* ((*OclAllInstances-generic pre-post OclAny*)−>*forAll*(*X*|*X .oclIsTypeOf*(*OclAny*))))
proof − **fix** *oid a* **let** *?t0* = ⦇*heap* = *empty*(*oid* ↦ *in*$_{OclAny}$ (*mk*$_{OclAny}$ *oid* ⌊*a*⌋)),
assocs = *empty*⦈ **show** *?thesis*
apply(*rule-tac x* = (*?t0*, *?t0*) **in** *exI*, *simp add*: *OclValid-def del*: *OclAllInstances-generic-def*)
apply(*simp only*: *OclForall-def refl if-True*
OclAllInstances-generic-defined[*simplified OclValid-def*])
apply(*simp only*: *OclAllInstances-generic-def OclAsType*$_{OclAny}$-$\mathfrak{A}$*-def*)
apply(*subst* (*1 2 3*) *Abs-Set*$_{base}$*-inverse*, *simp add*: *bot-option-def*)
by(*simp add*: *OclIsTypeOf*$_{OclAny}$*-OclAny OclNot-def OclAny-def*)
qed

lemma *OclAny-allInstances-at-post-oclIsTypeOf*$_{OclAny}$*2*:
$\exists\,\tau$. ($\tau \models$ *not* (*OclAny .allInstances*()−>*forAll*(*X*|*X .oclIsTypeOf*(*OclAny*))))
unfolding *OclAllInstances-at-post-def*
by(*rule OclAny-allInstances-generic-oclIsTypeOf*$_{OclAny}$*2*, *simp*)

lemma *OclAny-allInstances-at-pre-oclIsTypeOf*$_{OclAny}$*2*:
$\exists\,\tau.$ ($\tau \models$ *not* (*OclAny .allInstances@pre*()$-$>*forAll*(*X*|*X .oclIsTypeOf*(*OclAny*))))
unfolding *OclAllInstances-at-pre-def*
by(*rule OclAny-allInstances-generic-oclIsTypeOf*$_{OclAny}$*2*, *simp*)

lemma *Person-allInstances-generic-oclIsTypeOf*$_{Person}$:
$\tau \models$ ((*OclAllInstances-generic pre-post Person*)$-$>*forAll*(*X*|*X .oclIsTypeOf*(*Person*)))
 apply(*simp add*: *OclValid-def del*: *OclAllInstances-generic-def*)
 apply(*simp only*: *OclForall-def refl if-True*
  OclAllInstances-generic-defined[*simplified OclValid-def*])
 apply(*simp only*: *OclAllInstances-generic-def*)
 apply(*subst* (*1 2 3*) *Abs-Set*$_{base}$*-inverse*, *simp add*: *bot-option-def*)
by(*simp add*: *OclIsTypeOf*$_{Person}$*-Person*)

lemma *Person-allInstances-at-post-oclIsTypeOf*$_{Person}$:
$\tau \models$ (*Person .allInstances*()$-$>*forAll*(*X*|*X .oclIsTypeOf*(*Person*)))
unfolding *OclAllInstances-at-post-def*
by(*rule Person-allInstances-generic-oclIsTypeOf*$_{Person}$)

lemma *Person-allInstances-at-pre-oclIsTypeOf*$_{Person}$:
$\tau \models$ (*Person .allInstances@pre*()$-$>*forAll*(*X*|*X .oclIsTypeOf*(*Person*)))
unfolding *OclAllInstances-at-pre-def*
by(*rule Person-allInstances-generic-oclIsTypeOf*$_{Person}$)

OclIsKindOf

lemma *OclAny-allInstances-generic-oclIsKindOf*$_{OclAny}$:
$\tau \models$ ((*OclAllInstances-generic pre-post OclAny*)$-$>*forAll*(*X*|*X .oclIsKindOf*(*OclAny*)))
 apply(*simp add*: *OclValid-def del*: *OclAllInstances-generic-def*)
 apply(*simp only*: *OclForall-def refl if-True*
  OclAllInstances-generic-defined[*simplified OclValid-def*])
 apply(*simp only*: *OclAllInstances-generic-def*)
 apply(*subst* (*1 2 3*) *Abs-Set*$_{base}$*-inverse*, *simp add*: *bot-option-def*)
by(*simp add*: *OclIsKindOf*$_{OclAny}$*-OclAny*)

lemma *OclAny-allInstances-at-post-oclIsKindOf*$_{OclAny}$:
$\tau \models$ (*OclAny .allInstances*()$-$>*forAll*(*X*|*X .oclIsKindOf*(*OclAny*)))
unfolding *OclAllInstances-at-post-def*
by(*rule OclAny-allInstances-generic-oclIsKindOf*$_{OclAny}$)

lemma *OclAny-allInstances-at-pre-oclIsKindOf*$_{OclAny}$:
$\tau \models$ (*OclAny .allInstances@pre*()$-$>*forAll*(*X*|*X .oclIsKindOf*(*OclAny*)))
unfolding *OclAllInstances-at-pre-def*
by(*rule OclAny-allInstances-generic-oclIsKindOf*$_{OclAny}$)

lemma *Person-allInstances-generic-oclIsKindOf*$_{OclAny}$:
$\tau \models$ ((*OclAllInstances-generic pre-post Person*)$-$>*forAll*(*X*|*X .oclIsKindOf*(*OclAny*)))

apply(*simp add*: *OclValid-def del*: *OclAllInstances-generic-def*)
apply(*simp only*: *OclForall-def refl if-True*
OclAllInstances-generic-defined[*simplified OclValid-def*])
apply(*simp only*: *OclAllInstances-generic-def*)
apply(*subst* (*1 2 3*) *Abs-Set*$_{base}$*-inverse*, *simp add*: *bot-option-def*)
by(*simp add*: *OclIsKindOf*$_{OclAny}$*-Person*)

lemma *Person-allInstances-at-post-oclIsKindOf*$_{OclAny}$:
$\tau \models$ (*Person .allInstances*()$-$>*forAll*(X|X *.oclIsKindOf*(*OclAny*)))
unfolding *OclAllInstances-at-post-def*
by(*rule Person-allInstances-generic-oclIsKindOf*$_{OclAny}$)

lemma *Person-allInstances-at-pre-oclIsKindOf*$_{OclAny}$:
$\tau \models$ (*Person .allInstances@pre*()$-$>*forAll*(X|X *.oclIsKindOf*(*OclAny*)))
unfolding *OclAllInstances-at-pre-def*
by(*rule Person-allInstances-generic-oclIsKindOf*$_{OclAny}$)

lemma *Person-allInstances-generic-oclIsKindOf*$_{Person}$:
$\tau \models$ ((*OclAllInstances-generic pre-post Person*)$-$>*forAll*(X|X *.oclIsKindOf*(*Person*)))
apply(*simp add*: *OclValid-def del*: *OclAllInstances-generic-def*)
apply(*simp only*: *OclForall-def refl if-True*
OclAllInstances-generic-defined[*simplified OclValid-def*])
apply(*simp only*: *OclAllInstances-generic-def*)
apply(*subst* (*1 2 3*) *Abs-Set*$_{base}$*-inverse*, *simp add*: *bot-option-def*)
by(*simp add*: *OclIsKindOf*$_{Person}$*-Person*)

lemma *Person-allInstances-at-post-oclIsKindOf*$_{Person}$:
$\tau \models$ (*Person .allInstances*()$-$>*forAll*(X|X *.oclIsKindOf*(*Person*)))
unfolding *OclAllInstances-at-post-def*
by(*rule Person-allInstances-generic-oclIsKindOf*$_{Person}$)

lemma *Person-allInstances-at-pre-oclIsKindOf*$_{Person}$:
$\tau \models$ (*Person .allInstances@pre*()$-$>*forAll*(X|X *.oclIsKindOf*(*Person*)))
unfolding *OclAllInstances-at-pre-def*
by(*rule Person-allInstances-generic-oclIsKindOf*$_{Person}$)

B.5.8. The Accessors (any, boss, salary)

Should be generated entirely from a class-diagram.

Definition

definition *eval-extract* :: ($'\mathfrak{A}$,($'a$::*object*) *option option*) *val*
$\Rightarrow$ (*oid* $\Rightarrow$ ($'\mathfrak{A}$,$'c$::*null*) *val*)
$\Rightarrow$ ($'\mathfrak{A}$,$'c$::*null*) *val*
where *eval-extract X f* = (λ τ. *case X* τ *of*
$\perp \Rightarrow$ *invalid* τ ($*$ *exception propagation* $*$)

$\mid \lfloor\ \bot\ \rfloor \Rightarrow \textit{invalid}\ \tau$ (∗ *dereferencing null pointer* ∗)
$\mid \lfloor\lfloor\ \textit{obj}\ \rfloor\rfloor \Rightarrow f\ (\textit{oid-of obj})\ \tau)$

definition $\textit{deref-oid}_{Person} :: (\mathfrak{A}\ \textit{state} \times \mathfrak{A}\ \textit{state} \Rightarrow \mathfrak{A}\ \textit{state})$
$\Rightarrow (\textit{type}_{Person} \Rightarrow (\mathfrak{A},\ 'c{::}\textit{null})\textit{val})$
$\Rightarrow \textit{oid}$
$\Rightarrow (\mathfrak{A},\ 'c{::}\textit{null})\textit{val}$
where $\textit{deref-oid}_{Person}\ \textit{fst-snd}\ f\ \textit{oid} = (\lambda\tau.\ \textit{case}\ (\textit{heap}\ (\textit{fst-snd}\ \tau))\ \textit{oid of}$
$\lfloor\ \textit{in}_{Person}\ \textit{obj}\ \rfloor \Rightarrow f\ \textit{obj}\ \tau$
$\mid \text{-} \quad \Rightarrow \textit{invalid}\ \tau)$

definition $\textit{deref-oid}_{OclAny} :: (\mathfrak{A}\ \textit{state} \times \mathfrak{A}\ \textit{state} \Rightarrow \mathfrak{A}\ \textit{state})$
$\Rightarrow (\textit{type}_{OclAny} \Rightarrow (\mathfrak{A},\ 'c{::}\textit{null})\textit{val})$
$\Rightarrow \textit{oid}$
$\Rightarrow (\mathfrak{A},\ 'c{::}\textit{null})\textit{val}$
where $\textit{deref-oid}_{OclAny}\ \textit{fst-snd}\ f\ \textit{oid} = (\lambda\tau.\ \textit{case}\ (\textit{heap}\ (\textit{fst-snd}\ \tau))\ \textit{oid of}$
$\lfloor\ \textit{in}_{OclAny}\ \textit{obj}\ \rfloor \Rightarrow f\ \textit{obj}\ \tau$
$\mid \text{-} \quad \Rightarrow \textit{invalid}\ \tau)$

pointer undefined in state or not referencing a type conform object representation

definition $\textit{select}_{OclAny}\mathscr{ANY}\ f = (\lambda\ X.\ \textit{case}\ X\ \textit{of}$
$(\textit{mk}_{OclAny}\ \text{-}\ \bot) \Rightarrow \textit{null}$
$\mid (\textit{mk}_{OclAny}\ \text{-}\ \lfloor \textit{any} \rfloor) \Rightarrow f\ (\lambda x\ \text{-}.\ \lfloor\lfloor x \rfloor\rfloor)\ \textit{any})$

definition $\textit{select}_{Person}\mathscr{BOSS}\ f = (\lambda\ X.\ \textit{case}\ X\ \textit{of}$
$(\textit{mk}_{Person}\ \text{-}\ \text{-}\ \bot) \Rightarrow \textit{null}$ (∗ *object contains null pointer* ∗)
$\mid (\textit{mk}_{Person}\ \text{-}\ \text{-}\ \lfloor \textit{boss} \rfloor) \Rightarrow f\ (\lambda x\ \text{-}.\ \lfloor\lfloor x \rfloor\rfloor)\ \textit{boss})$

definition $\textit{select}_{Person}\mathscr{SALARY}\ f = (\lambda\ X.\ \textit{case}\ X\ \textit{of}$
$(\textit{mk}_{Person}\ \text{-}\ \bot\ \text{-}) \Rightarrow \textit{null}$
$\mid (\textit{mk}_{Person}\ \text{-}\ \lfloor \textit{salary} \rfloor\ \text{-}) \Rightarrow f\ (\lambda x\ \text{-}.\ \lfloor\lfloor x \rfloor\rfloor)\ \textit{salary})$

definition $\textit{in-pre-state} = \textit{fst}$
definition $\textit{in-post-state} = \textit{snd}$

definition $\textit{reconst-basetype} = (\lambda\ \textit{convert}\ x.\ \textit{convert}\ x)$

definition $\textit{dot}_{OclAny}\mathscr{ANY} :: \textit{OclAny} \Rightarrow \text{-}\ \ ((1(\text{-}).\textit{any})\ 50)$
where $(X).\textit{any} = \textit{eval-extract}\ X$
$(\textit{deref-oid}_{OclAny}\ \textit{in-post-state}$
$(\textit{select}_{OclAny}\mathscr{ANY}$

reconst-basetype))

definition $dot_{Person}\mathcal{BOSS}$ :: *Person* ⇒ *Person* ((*1*(-).*boss*) *50*)
where (X).*boss* = *eval-extract X*
(*deref-oid*$_{Person}$ *in-post-state*
(*select*$_{Person}\mathcal{BOSS}$
(*deref-oid*$_{Person}$ *in-post-state*)))

definition $dot_{Person}\mathcal{SALARY}$ :: *Person* ⇒ *Integer* ((*1*(-).*salary*) *50*)
where (X).*salary* = *eval-extract X*
(*deref-oid*$_{Person}$ *in-post-state*
(*select*$_{Person}\mathcal{SALARY}$
reconst-basetype))

definition $dot_{OclAny}\mathcal{ANY}$*-at-pre* :: *OclAny* ⇒ - ((*1*(-).*any@pre*) *50*)
where (X).*any@pre* = *eval-extract X*
(*deref-oid*$_{OclAny}$ *in-pre-state*
(*select*$_{OclAny}\mathcal{ANY}$
reconst-basetype))

definition $dot_{Person}\mathcal{BOSS}$*-at-pre*:: *Person* ⇒ *Person* ((*1*(-).*boss@pre*) *50*)
where (X).*boss@pre* = *eval-extract X*
(*deref-oid*$_{Person}$ *in-pre-state*
(*select*$_{Person}\mathcal{BOSS}$
(*deref-oid*$_{Person}$ *in-pre-state*)))

definition $dot_{Person}\mathcal{SALARY}$*-at-pre*:: *Person* ⇒ *Integer* ((*1*(-).*salary@pre*) *50*)
where (X).*salary@pre* = *eval-extract X*
(*deref-oid*$_{Person}$ *in-pre-state*
(*select*$_{Person}\mathcal{SALARY}$
reconst-basetype))

lemmas *dot-accessor* =
$dot_{OclAny}\mathcal{ANY}$*-def*
$dot_{Person}\mathcal{BOSS}$*-def*
$dot_{Person}\mathcal{SALARY}$*-def*
$dot_{OclAny}\mathcal{ANY}$*-at-pre-def*
$dot_{Person}\mathcal{BOSS}$*-at-pre-def*
$dot_{Person}\mathcal{SALARY}$*-at-pre-def*

Context Passing

lemmas [*simp*] = *eval-extract-def*

lemma *cp-*$dot_{OclAny}\mathcal{ANY}$: (($X$).*any*) τ = ((λ-. X τ).*any*) τ **by** (*simp add*: *dot-accessor*)
lemma *cp-*$dot_{Person}\mathcal{BOSS}$: (($X$).*boss*) τ = ((λ-. X τ).*boss*) τ **by** (*simp add*: *dot-accessor*)
lemma *cp-*$dot_{Person}\mathcal{SALARY}$: (($X$).*salary*) τ = ((λ-. X τ).*salary*) τ **by** (*simp add*: *dot-accessor*)

lemma *cp-dot*$_{OclAny}\mathcal{ANY}$*-at-pre*: ((*X*).*any@pre*) τ = ((λ-. *X* τ).*any@pre*) τ **by** (*simp add*: *dot-accessor*)
lemma *cp-dot*$_{Person}\mathcal{BOSS}$*-at-pre*: ((*X*).*boss@pre*) τ = ((λ-. *X* τ).*boss@pre*) τ **by** (*simp add*: *dot-accessor*)
lemma *cp-dot*$_{Person}\mathcal{SALARY}$*-at-pre*: ((*X*).*salary@pre*) τ = ((λ-. *X* τ).*salary@pre*) τ **by** (*simp add*: *dot-accessor*)

lemmas *cp-dot*$_{OclAny}\mathcal{ANY}$*-I* [*simp*, *intro*!]=
cp-dot$_{OclAny}\mathcal{ANY}$[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]
lemmas *cp-dot*$_{OclAny}\mathcal{ANY}$*-at-pre-I* [*simp*, *intro*!]=
cp-dot$_{OclAny}\mathcal{ANY}$*-at-pre*[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

lemmas *cp-dot*$_{Person}\mathcal{BOSS}$*-I* [*simp*, *intro*!]=
cp-dot$_{Person}\mathcal{BOSS}$[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]
lemmas *cp-dot*$_{Person}\mathcal{BOSS}$*-at-pre-I* [*simp*, *intro*!]=
cp-dot$_{Person}\mathcal{BOSS}$*-at-pre*[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

lemmas *cp-dot*$_{Person}\mathcal{SALARY}$*-I* [*simp*, *intro*!]=
cp-dot$_{Person}\mathcal{SALARY}$[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]
lemmas *cp-dot*$_{Person}\mathcal{SALARY}$*-at-pre-I* [*simp*, *intro*!]=
cp-dot$_{Person}\mathcal{SALARY}$*-at-pre*[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

Execution with Invalid or Null as Argument

lemma *dot*$_{OclAny}\mathcal{ANY}$*-nullstrict* [*simp*]: (*null*).*any* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma *dot*$_{OclAny}\mathcal{ANY}$*-at-pre-nullstrict* [*simp*] : (*null*).*any@pre* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma *dot*$_{OclAny}\mathcal{ANY}$*-strict* [*simp*] : (*invalid*).*any* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma *dot*$_{OclAny}\mathcal{ANY}$*-at-pre-strict* [*simp*] : (*invalid*).*any@pre* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)

lemma *dot*$_{Person}\mathcal{BOSS}$*-nullstrict* [*simp*]: (*null*).*boss* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma *dot*$_{Person}\mathcal{BOSS}$*-at-pre-nullstrict* [*simp*] : (*null*).*boss@pre* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma *dot*$_{Person}\mathcal{BOSS}$*-strict* [*simp*] : (*invalid*).*boss* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma *dot*$_{Person}\mathcal{BOSS}$*-at-pre-strict* [*simp*] : (*invalid*).*boss@pre* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)

lemma *dot*$_{Person}\mathcal{SALARY}$*-nullstrict* [*simp*]: (*null*).*salary* = *invalid*

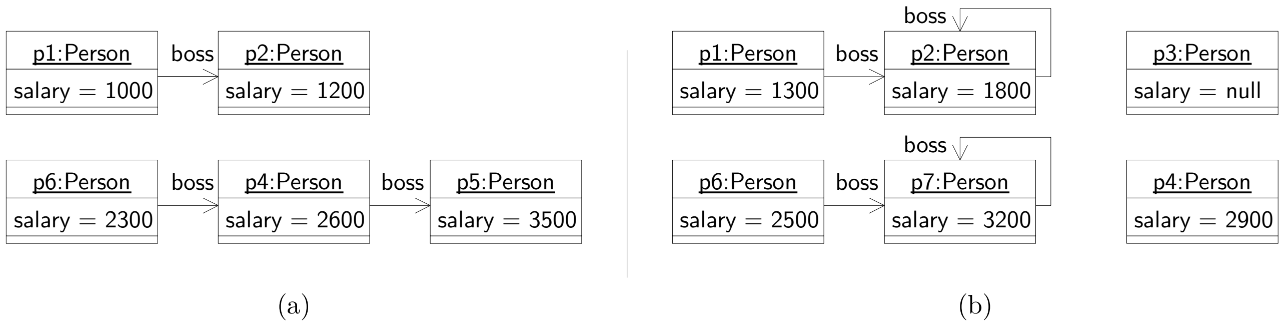


Figure B.4.: (a) pre-state σ_1 and (b) post-state σ_1'.

by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma $dot_{Person}\mathcal{SALARY}$*-at-pre-nullstrict* [*simp*] : (*null*).*salary@pre* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma $dot_{Person}\mathcal{SALARY}$*-strict* [*simp*] : (*invalid*).*salary* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)
lemma $dot_{Person}\mathcal{SALARY}$*-at-pre-strict* [*simp*] : (*invalid*).*salary@pre* = *invalid*
by(*rule ext*, *simp add*: *dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def*)

B.5.9. A Little Infra-structure on Example States

The example we are defining in this section comes from the figure B.4.

definition *OclInt1000* (**1000**) **where** *OclInt1000* = $(\lambda$ - . $\lfloor\lfloor 1000\rfloor\rfloor)$
definition *OclInt1200* (**1200**) **where** *OclInt1200* = $(\lambda$ - . $\lfloor\lfloor 1200\rfloor\rfloor)$
definition *OclInt1300* (**1300**) **where** *OclInt1300* = $(\lambda$ - . $\lfloor\lfloor 1300\rfloor\rfloor)$
definition *OclInt1800* (**1800**) **where** *OclInt1800* = $(\lambda$ - . $\lfloor\lfloor 1800\rfloor\rfloor)$
definition *OclInt2600* (**2600**) **where** *OclInt2600* = $(\lambda$ - . $\lfloor\lfloor 2600\rfloor\rfloor)$
definition *OclInt2900* (**2900**) **where** *OclInt2900* = $(\lambda$ - . $\lfloor\lfloor 2900\rfloor\rfloor)$
definition *OclInt3200* (**3200**) **where** *OclInt3200* = $(\lambda$ - . $\lfloor\lfloor 3200\rfloor\rfloor)$
definition *OclInt3500* (**3500**) **where** *OclInt3500* = $(\lambda$ - . $\lfloor\lfloor 3500\rfloor\rfloor)$

definition *oid0* ≡ *0*
definition *oid1* ≡ *1*
definition *oid2* ≡ *2*
definition *oid3* ≡ *3*
definition *oid4* ≡ *4*
definition *oid5* ≡ *5*
definition *oid6* ≡ *6*
definition *oid7* ≡ *7*
definition *oid8* ≡ *8*

definition *person1* ≡ mk_{Person} *oid0* $\lfloor 1300\rfloor$ $\lfloor oid1\rfloor$
definition *person2* ≡ mk_{Person} *oid1* $\lfloor 1800\rfloor$ $\lfloor oid1\rfloor$
definition *person3* ≡ mk_{Person} *oid2 None None*

definition $person4 \equiv mk_{Person}\ oid3\ \lfloor 2900 \rfloor\ None$
definition $person5 \equiv mk_{Person}\ oid4\ \lfloor 3500 \rfloor\ None$
definition $person6 \equiv mk_{Person}\ oid5\ \lfloor 2500 \rfloor\ \lfloor oid6 \rfloor$
definition $person7 \equiv mk_{OclAny}\ oid6\ \lfloor (\lfloor 3200 \rfloor,\ \lfloor oid6 \rfloor) \rfloor$
definition $person8 \equiv mk_{OclAny}\ oid7\ None$
definition $person9 \equiv mk_{Person}\ oid8\ \lfloor 0 \rfloor\ None$

definition
$\sigma_1\ \equiv (\!|\ heap = empty(oid0 \mapsto in_{Person}\ (mk_{Person}\ oid0\ \lfloor 1000 \rfloor\ \lfloor oid1 \rfloor))$
$(oid1 \mapsto in_{Person}\ (mk_{Person}\ oid1\ \lfloor 1200 \rfloor\ \ None))$
$(*oid2*)$
$(oid3 \mapsto in_{Person}\ (mk_{Person}\ oid3\ \lfloor 2600 \rfloor\ \lfloor oid4 \rfloor))$
$(oid4 \mapsto in_{Person}\ person5)$
$(oid5 \mapsto in_{Person}\ (mk_{Person}\ oid5\ \lfloor 2300 \rfloor\ \lfloor oid3 \rfloor))$
$(*oid6*)$
$(*oid7*)$
$(oid8 \mapsto in_{Person}\ person9),$
$assocs = empty\ |\!)$

definition
$\sigma_1{}' \equiv (\!|\ heap = empty(oid0 \mapsto in_{Person}\ person1)$
$(oid1 \mapsto in_{Person}\ person2)$
$(oid2 \mapsto in_{Person}\ person3)$
$(oid3 \mapsto in_{Person}\ person4)$
$(*oid4*)$
$(oid5 \mapsto in_{Person}\ person6)$
$(oid6 \mapsto in_{OclAny}\ person7)$
$(oid7 \mapsto in_{OclAny}\ person8)$
$(oid8 \mapsto in_{Person}\ person9),$
$assocs = empty\ |\!)$

definition $\sigma_0 \equiv (\!|\ heap = empty,\ assocs = empty\ |\!)$

lemma $basic\text{-}\tau\text{-}wff$: $WFF(\sigma_1,\sigma_1{}')$
by$($*auto simp*: *WFF-def* σ_1*-def* $\sigma_1{}'$*-def*
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def
oid-of-$\mathfrak{A}$*-def oid-of-type*$_{Person}$*-def oid-of-type*$_{OclAny}$*-def*
person1-def person2-def person3-def person4-def
person5-def person6-def person7-def person8-def person9-def$)$

lemma $[simp,code\text{-}unfold]$: $dom\ (heap\ \sigma_1) = \{oid0,oid1,(*,oid2*)oid3,oid4,oid5(*,oid6,oid7*),oid8\}$
by$($*auto simp*: σ_1*-def*$)$

lemma $[simp,code\text{-}unfold]$: $dom\ (heap\ \sigma_1{}') = \{oid0,oid1,oid2,oid3,(*,oid4*)oid5,oid6,oid7,oid8\}$
by$($*auto simp*: $\sigma_1{}'$*-def*$)$

definition $X_{Person}1 :: Person \equiv \lambda\ \text{-}\ .\lfloor\lfloor\ person1\ \rfloor\rfloor$

definition $X_{Person}2$ :: *Person* $\equiv \lambda$ - .$\lfloor\lfloor$ *person2* $\rfloor\rfloor$

definition $X_{Person}3$ :: *Person* $\equiv \lambda$ - .$\lfloor\lfloor$ *person3* $\rfloor\rfloor$

definition $X_{Person}4$ :: *Person* $\equiv \lambda$ - .$\lfloor\lfloor$ *person4* $\rfloor\rfloor$

definition $X_{Person}5$ :: *Person* $\equiv \lambda$ - .$\lfloor\lfloor$ *person5* $\rfloor\rfloor$

definition $X_{Person}6$ :: *Person* $\equiv \lambda$ - .$\lfloor\lfloor$ *person6* $\rfloor\rfloor$

definition $X_{Person}7$ :: *OclAny* $\equiv \lambda$ - .$\lfloor\lfloor$ *person7* $\rfloor\rfloor$

definition $X_{Person}8$ :: *OclAny* $\equiv \lambda$ - .$\lfloor\lfloor$ *person8* $\rfloor\rfloor$

definition $X_{Person}9$ :: *Person* $\equiv \lambda$ - .$\lfloor\lfloor$ *person9* $\rfloor\rfloor$

lemma [*code-unfold*]: $((x::Person) \doteq y) = StrictRefEq_{Object}\ x\ y$ **by**(*simp only*: $StrictRefEq_{Object}\text{-}_{Person}$)

lemma [*code-unfold*]: $((x::OclAny) \doteq y) = StrictRefEq_{Object}\ x\ y$ **by**(*simp only*: $StrictRefEq_{Object}\text{-}_{OclAny}$)

lemmas [*simp*,*code-unfold*] =

$OclAsType_{OclAny}$-*OclAny*
$OclAsType_{OclAny}$-*Person*
$OclAsType_{Person}$-*OclAny*
$OclAsType_{Person}$-*Person*

$OclIsTypeOf_{OclAny}$-*OclAny*
$OclIsTypeOf_{OclAny}$-*Person*
$OclIsTypeOf_{Person}$-*OclAny*
$OclIsTypeOf_{Person}$-*Person*

$OclIsKindOf_{OclAny}$-*OclAny*
$OclIsKindOf_{OclAny}$-*Person*
$OclIsKindOf_{Person}$-*OclAny*
$OclIsKindOf_{Person}$-*Person*

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models (X_{Person}1\ .salary\ <> \mathbf{1000})$

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models (X_{Person}1\ .salary\ \doteq \mathbf{1300})$

Assert $\bigwedge\ s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}1\ .salary@pre\ \doteq \mathbf{1000})$

Assert $\bigwedge\ s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}1\ .salary@pre\ <> \mathbf{1300})$

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models (X_{Person}1\ .boss\ <> X_{Person}1)$

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models (X_{Person}1\ .boss\ .salary\ \doteq \mathbf{1800})$

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models (X_{Person}1\ .boss\ .boss\ <> X_{Person}1)$

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models (X_{Person}1\ .boss\ .boss\ \doteq X_{Person}2)$

Assert $(\sigma_1,\sigma_1{}') \models (X_{Person}1\ .boss@pre\ .salary\ \doteq \mathbf{1800})$

Assert $\bigwedge\ s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}1\ .boss@pre\ .salary@pre\ \doteq \mathbf{1200})$

Assert $\bigwedge\ s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}1\ .boss@pre\ .salary@pre\ <> \mathbf{1800})$

Assert $\bigwedge\ s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}1\ .boss@pre\ \doteq X_{Person}2)$

Assert $(\sigma_1,\sigma_1{}') \models (X_{Person}1\ .boss@pre\ .boss\ \doteq X_{Person}2)$

Assert $\bigwedge\ s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}1\ .boss@pre\ .boss@pre\ \doteq null)$

Assert $\bigwedge\ s_{post}.\ (\sigma_1,s_{post}) \models not(\upsilon(X_{Person}1\ .boss@pre\ .boss@pre\ .boss@pre))$

lemma $(\sigma_1,\sigma_1{}') \models (X_{Person}1\ .oclIsMaintained())$
by(*simp add*: *OclValid-def OclIsMaintained-def*
σ_1-*def* $\sigma_1{}'$-*def*

$X_{Person}1$-def person1-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def
oid-of-option-def oid-of-type$_{Person}$-def)

lemma $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models ((X_{Person}1\ .oclAsType(OclAny)\ .oclAsType(Person)) \doteq X_{Person}1)$
by(*rule up-down-cast-Person-OclAny-Person′, simp add*: *$X_{Person}1$-def*)
Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models (X_{Person}1\ .oclIsTypeOf(Person))$
Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models not(X_{Person}1\ .oclIsTypeOf(OclAny))$
Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models (X_{Person}1\ .oclIsKindOf(Person))$
Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models (X_{Person}1\ .oclIsKindOf(OclAny))$
Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models not(X_{Person}1\ .oclAsType(OclAny)\ .oclIsTypeOf(OclAny))$

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1') \models (X_{Person}2\ .salary \doteq \mathbf{1800})$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}2\ .salary@pre \doteq \mathbf{1200})$
Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1') \models (X_{Person}2\ .boss \doteq X_{Person}2)$
Assert $(\sigma_1,\sigma_1') \models (X_{Person}2\ .boss\ .salary@pre \doteq \mathbf{1200})$
Assert $(\sigma_1,\sigma_1') \models (X_{Person}2\ .boss\ .boss@pre \doteq null)$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}2\ .boss@pre \doteq null)$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}2\ .boss@pre\ <>\ X_{Person}2)$
Assert $(\sigma_1,\sigma_1') \models (X_{Person}2\ .boss@pre\ <>\ (X_{Person}2\ .boss))$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models not(\upsilon(X_{Person}2\ .boss@pre\ .boss))$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models not(\upsilon(X_{Person}2\ .boss@pre\ .salary@pre))$
lemma $(\sigma_1,\sigma_1') \models (X_{Person}2\ .oclIsMaintained())$
by(*simp add*: *OclValid-def OclIsMaintained-def*
σ_1-def σ_1'-def
$X_{Person}2$-def person2-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def
oid-of-option-def oid-of-type$_{Person}$-def)

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1') \models (X_{Person}3\ .salary \doteq null)$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models not(\upsilon(X_{Person}3\ .salary@pre))$
Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1') \models (X_{Person}3\ .boss \doteq null)$
Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1') \models not(\upsilon(X_{Person}3\ .boss\ .salary))$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models not(\upsilon(X_{Person}3\ .boss@pre))$
lemma $(\sigma_1,\sigma_1') \models (X_{Person}3\ .oclIsNew())$
by(*simp add*: *OclValid-def OclIsNew-def*
σ_1-def σ_1'-def
$X_{Person}3$-def person3-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid8-def
oid-of-option-def oid-of-type$_{Person}$-def)

Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}4\ .boss@pre \doteq X_{Person}5)$
Assert $(\sigma_1,\sigma_1') \models not(\upsilon(X_{Person}4\ .boss@pre\ .salary))$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}4\ .boss@pre\ .salary@pre \doteq \mathbf{3500})$
lemma $(\sigma_1,\sigma_1') \models (X_{Person}4\ .oclIsMaintained())$

by(*simp add*: *OclValid-def OclIsMaintained-def*
    σ_1*-def* $\sigma_1{}'$*-def*
    X_{Person}*4-def person4-def*
    oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def
    oid-of-option-def oid-of-type$_{Person}$*-def*)

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models not(\upsilon(X_{Person}5\ .salary))$

Assert $\bigwedge\ s_{post}.\ (\sigma_1,s_{post}) \models\ (X_{Person}5\ .salary@pre\ \doteq \mathbf{3500})$

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models not(\upsilon(X_{Person}5\ .boss))$

lemma $(\sigma_1,\sigma_1{}') \models\ (X_{Person}5\ .oclIsDeleted())$

by(*simp add*: *OclNot-def OclValid-def OclIsDeleted-def*
    σ_1*-def* $\sigma_1{}'$*-def*
    X_{Person}*5-def person5-def*
    oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def
    oid-of-option-def oid-of-type$_{Person}$*-def*)

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models not(\upsilon(X_{Person}6\ .boss\ .salary@pre))$

Assert $\bigwedge\ s_{post}.\ (\sigma_1,s_{post}) \models\ (X_{Person}6\ .boss@pre\ \doteq X_{Person}4)$

Assert $(\sigma_1,\sigma_1{}') \models\ (X_{Person}6\ .boss@pre\ .salary\ \doteq \mathbf{2900})$

Assert $\bigwedge\ s_{post}.\ (\sigma_1,s_{post}) \models\ (X_{Person}6\ .boss@pre\ .salary@pre\ \doteq \mathbf{2600})$

Assert $\bigwedge\ s_{post}.\ (\sigma_1,s_{post}) \models\ (X_{Person}6\ .boss@pre\ .boss@pre\ \doteq X_{Person}5)$

lemma $(\sigma_1,\sigma_1{}') \models\ (X_{Person}6\ .oclIsMaintained())$

by(*simp add*: *OclValid-def OclIsMaintained-def*
    σ_1*-def* $\sigma_1{}'$*-def*
    X_{Person}*6-def person6-def*
    oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def
    oid-of-option-def oid-of-type$_{Person}$*-def*)

Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models\ \upsilon(X_{Person}7\ .oclAsType(Person))$

Assert $\bigwedge\ s_{post}.\ (\sigma_1,s_{post}) \models not(\upsilon(X_{Person}7\ .oclAsType(Person)\ .boss@pre))$

lemma $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models\ ((X_{Person}7\ .oclAsType(Person)\ .oclAsType(OclAny)$
$.oclAsType(Person))$
$\doteq (X_{Person}7\ .oclAsType(Person)))$

by(*rule up-down-cast-Person-OclAny-Person*$'$, *simp add*: X_{Person}*7-def OclValid-def valid-def person7-def*)

lemma $(\sigma_1,\sigma_1{}') \models\ (X_{Person}7\ .oclIsNew())$

by(*simp add*: *OclValid-def OclIsNew-def*
    σ_1*-def* $\sigma_1{}'$*-def*
    X_{Person}*7-def person7-def*
    oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid8-def
    oid-of-option-def oid-of-type$_{OclAny}$*-def*)

Assert $\bigwedge s_{pre}\ s_{post}.\quad (s_{pre},s_{post}) \models \quad (X_{Person}\mathit{8}\ <>\ X_{Person}\mathit{7})$
Assert $\bigwedge s_{pre}\ s_{post}.\quad (s_{pre},s_{post}) \models \mathit{not}(\upsilon(X_{Person}\mathit{8}\ .\mathit{oclAsType}(\mathit{Person})))$
Assert $\bigwedge s_{pre}\ s_{post}.\quad (s_{pre},s_{post}) \models \quad (X_{Person}\mathit{8}\ .\mathit{oclIsTypeOf}(\mathit{OclAny}))$
Assert $\bigwedge s_{pre}\ s_{post}.\quad (s_{pre},s_{post}) \models\ \mathit{not}(X_{Person}\mathit{8}\ .\mathit{oclIsTypeOf}(\mathit{Person}))$
Assert $\bigwedge s_{pre}\ s_{post}.\quad (s_{pre},s_{post}) \models\ \mathit{not}(X_{Person}\mathit{8}\ .\mathit{oclIsKindOf}(\mathit{Person}))$
Assert $\bigwedge s_{pre}\ s_{post}.\quad (s_{pre},s_{post}) \models \quad (X_{Person}\mathit{8}\ .\mathit{oclIsKindOf}(\mathit{OclAny}))$

lemma σ*-modifiedonly*: $(\sigma_1,\sigma_1{}') \models (\mathit{Set}\{\ X_{Person}\mathit{1}\ .\mathit{oclAsType}(\mathit{OclAny})$
$,\ X_{Person}\mathit{2}\ .\mathit{oclAsType}(\mathit{OclAny})$
$(*,\ X_{Person}\mathit{3}\ .\mathit{oclAsType}(\mathit{OclAny})*)$
$,\ X_{Person}\mathit{4}\ .\mathit{oclAsType}(\mathit{OclAny})$
$(*,\ X_{Person}\mathit{5}\ .\mathit{oclAsType}(\mathit{OclAny})*)$
$,\ X_{Person}\mathit{6}\ .\mathit{oclAsType}(\mathit{OclAny})$
$(*,\ X_{Person}\mathit{7}\ .\mathit{oclAsType}(\mathit{OclAny})*)$
$(*,\ X_{Person}\mathit{8}\ .\mathit{oclAsType}(\mathit{OclAny})*)$
$(*,\ X_{Person}\mathit{9}\ .\mathit{oclAsType}(\mathit{OclAny})*)\}{-}{>}\mathit{oclIsModifiedOnly}())$
apply(*simp add*: *OclIsModifiedOnly-def OclValid-def*
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def
X_{Person}*1-def* X_{Person}*2-def* X_{Person}*3-def* X_{Person}*4-def*
X_{Person}*5-def* X_{Person}*6-def* X_{Person}*7-def* X_{Person}*8-def* X_{Person}*9-def*
person1-def person2-def person3-def person4-def
person5-def person6-def person7-def person8-def person9-def
image-def)
apply(*simp add*: *OclIncluding-rep-set mtSet-rep-set null-option-def bot-option-def*)
apply(*simp add*: *oid-of-option-def oid-of-type*$_{OclAny}$*-def*, *clarsimp*)
apply(*simp add*: σ_1*-def* $\sigma_1{}'$*-def*
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)
done

lemma $(\sigma_1,\sigma_1{}') \models ((X_{Person}\mathit{9}\ @\mathit{pre}\ (\lambda x.\ \lfloor \mathit{OclAsType}_{Person}\text{-}\mathfrak{A}\ x\rfloor))\ \triangleq X_{Person}\mathit{9})$
by(*simp add*: *OclSelf-at-pre-def* σ_1*-def oid-of-option-def oid-of-type*$_{Person}$*-def*
X_{Person}*9-def person9-def oid8-def OclValid-def StrongEq-def OclAsType*$_{Person}$*-*$\mathfrak{A}$*-def*)

lemma $(\sigma_1,\sigma_1{}') \models ((X_{Person}\mathit{9}\ @\mathit{post}\ (\lambda x.\ \lfloor \mathit{OclAsType}_{Person}\text{-}\mathfrak{A}\ x\rfloor))\ \triangleq X_{Person}\mathit{9})$
by(*simp add*: *OclSelf-at-post-def* $\sigma_1{}'$*-def oid-of-option-def oid-of-type*$_{Person}$*-def*
X_{Person}*9-def person9-def oid8-def OclValid-def StrongEq-def OclAsType*$_{Person}$*-*$\mathfrak{A}$*-def*)

lemma $(\sigma_1,\sigma_1{}') \models (((X_{Person}\mathit{9}\ .\mathit{oclAsType}(\mathit{OclAny}))\ @\mathit{pre}\ (\lambda x.\ \lfloor \mathit{OclAsType}_{OclAny}\text{-}\mathfrak{A}\ x\rfloor)) \triangleq$
$((X_{Person}\mathit{9}\ .\mathit{oclAsType}(\mathit{OclAny}))\ @\mathit{post}\ (\lambda x.\ \lfloor \mathit{OclAsType}_{OclAny}\text{-}\mathfrak{A}\ x\rfloor)))$
proof –

have *including4* : $\bigwedge a\ b\ c\ d\ \tau.$
$\mathit{Set}\{\lambda\tau.\ \lfloor\lfloor a\rfloor\rfloor,\ \lambda\tau.\ \lfloor\lfloor b\rfloor\rfloor,\ \lambda\tau.\ \lfloor\lfloor c\rfloor\rfloor,\ \lambda\tau.\ \lfloor\lfloor d\rfloor\rfloor\}\ \tau = \mathit{Abs\text{-}Set}_{base}\ \lfloor\lfloor\ \{\lfloor\lfloor a\rfloor\rfloor,\ \lfloor\lfloor b\rfloor\rfloor,\ \lfloor\lfloor c\rfloor\rfloor,\ \lfloor\lfloor d\rfloor\rfloor\}\ \rfloor\rfloor$
apply(*subst abs-rep-simp*$'$[*symmetric*], *simp*)
apply(*simp add*: *OclIncluding-rep-set mtSet-rep-set*)
by(*rule arg-cong*[*of - -* $\lambda x.\ (\mathit{Abs\text{-}Set}_{base}(\lfloor\lfloor\ x\ \rfloor\rfloor))$], *auto*)

have *excluding1*: $\bigwedge S\ a\ b\ c\ d\ e\ \tau.$
$(\lambda\text{-}.\ \mathit{Abs\text{-}Set}_{base}\ \lfloor\lfloor\ \{\lfloor\lfloor a\rfloor\rfloor,\ \lfloor\lfloor b\rfloor\rfloor,\ \lfloor\lfloor c\rfloor\rfloor,\ \lfloor\lfloor d\rfloor\rfloor\}\ \rfloor\rfloor){-}{>}\mathit{excluding}(\lambda\tau.\ \lfloor\lfloor e\rfloor\rfloor)\ \tau =$
$\mathit{Abs\text{-}Set}_{base}\ \lfloor\lfloor\ \{\lfloor\lfloor a\rfloor\rfloor,\ \lfloor\lfloor b\rfloor\rfloor,\ \lfloor\lfloor c\rfloor\rfloor,\ \lfloor\lfloor d\rfloor\rfloor\} - \{\lfloor\lfloor e\rfloor\rfloor\}\ \rfloor\rfloor$

apply(*simp add*: *OclExcluding-def*)

apply(*simp add*: *defined-def OclValid-def false-def true-def*
bot-fun-def bot-Set$_{base}$*-def null-fun-def null-Set*$_{base}$*-def*)

apply(*rule conjI*)

apply(*rule impI*, *subst* (*asm*) *Abs-Set*$_{base}$*-inject*) **apply**(*simp add*: *bot-option-def*)+

apply(*rule conjI*)

apply(*rule impI*, *subst* (*asm*) *Abs-Set*$_{base}$*-inject*) **apply**(*simp add*: *bot-option-def null-option-def*)+

apply(*subst Abs-Set*$_{base}$*-inverse*, *simp add*: *bot-option-def*, *simp*)

done

show *?thesis*

apply(*rule framing*[**where** X = *Set*{ X_{Person}*1 .oclAsType*(*OclAny*)
, X_{Person}*2 .oclAsType*(*OclAny*)
($*$, X_{Person}*3 .oclAsType*(*OclAny*)$*$)
, X_{Person}*4 .oclAsType*(*OclAny*)
($*$, X_{Person}*5 .oclAsType*(*OclAny*)$*$)
, X_{Person}*6 .oclAsType*(*OclAny*)
($*$, X_{Person}*7 .oclAsType*(*OclAny*)$*$)
($*$, X_{Person}*8 .oclAsType*(*OclAny*)$*$)
($*$, X_{Person}*9 .oclAsType*(*OclAny*)$*$)}])

apply(*cut-tac* σ*-modifiedonly*)

apply(*simp only*: *OclValid-def*
X_{Person}*1-def* X_{Person}*2-def* X_{Person}*3-def* X_{Person}*4-def*
X_{Person}*5-def* X_{Person}*6-def* X_{Person}*7-def* X_{Person}*8-def* X_{Person}*9-def*
person1-def person2-def person3-def person4-def
person5-def person6-def person7-def person8-def person9-def
OclAsType$_{OclAny}$*-Person*)

apply(*subst cp-OclIsModifiedOnly*, *subst cp-OclExcluding*,
subst (*asm*) *cp-OclIsModifiedOnly*, *simp add*: *including4 excluding1*)

apply(*simp only*: X_{Person}*1-def* X_{Person}*2-def* X_{Person}*3-def* X_{Person}*4-def*
X_{Person}*5-def* X_{Person}*6-def* X_{Person}*7-def* X_{Person}*8-def* X_{Person}*9-def*
person1-def person2-def person3-def person4-def
person5-def person6-def person7-def person8-def person9-def)

apply(*simp add*: *OclIncluding-rep-set mtSet-rep-set*
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)

apply(*simp add*: *StrictRefEq*$_{Object}$*-def oid-of-option-def oid-of-type*$_{OclAny}$*-def OclNot-def OclValid-def*
null-option-def bot-option-def)

done

qed

lemma *perm-*$\sigma_1{}'$: $\sigma_1{}'$ = (| *heap* = *empty*
(*oid8* $\mapsto$ *in*$_{Person}$ *person9*)
(*oid7* $\mapsto$ *in*$_{OclAny}$ *person8*)

($oid6 \mapsto in_{OclAny}$ *person7*)
($oid5 \mapsto in_{Person}$ *person6*)
(∗*oid4*∗)
($oid3 \mapsto in_{Person}$ *person4*)
($oid2 \mapsto in_{Person}$ *person3*)
($oid1 \mapsto in_{Person}$ *person2*)
($oid0 \mapsto in_{Person}$ *person1*)
, *assocs* = *assocs* $\sigma_1{}'$ ⦈

proof −
note *P* = *fun-upd-twist*
show *?thesis*
apply(*simp add*: $\sigma_1{}'$*-def*
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)
apply(*subst* (*1*) *P*, *simp*)
apply(*subst* (*2*) *P*, *simp*) **apply**(*subst* (*1*) *P*, *simp*)
apply(*subst* (*3*) *P*, *simp*) **apply**(*subst* (*2*) *P*, *simp*) **apply**(*subst* (*1*) *P*, *simp*)
apply(*subst* (*4*) *P*, *simp*) **apply**(*subst* (*3*) *P*, *simp*) **apply**(*subst* (*2*) *P*, *simp*) **apply**(*subst* (*1*) *P*, *simp*)
apply(*subst* (*5*) *P*, *simp*) **apply**(*subst* (*4*) *P*, *simp*) **apply**(*subst* (*3*) *P*, *simp*) **apply**(*subst* (*2*) *P*, *simp*) **apply**(*subst* (*1*) *P*, *simp*)
apply(*subst* (*6*) *P*, *simp*) **apply**(*subst* (*5*) *P*, *simp*) **apply**(*subst* (*4*) *P*, *simp*) **apply**(*subst* (*3*) *P*, *simp*) **apply**(*subst* (*2*) *P*, *simp*) **apply**(*subst* (*1*) *P*, *simp*)
apply(*subst* (*7*) *P*, *simp*) **apply**(*subst* (*6*) *P*, *simp*) **apply**(*subst* (*5*) *P*, *simp*) **apply**(*subst* (*4*) *P*, *simp*) **apply**(*subst* (*3*) *P*, *simp*) **apply**(*subst* (*2*) *P*, *simp*) **apply**(*subst* (*1*) *P*, *simp*)
by(*simp*)
qed

declare *const-ss* [*simp*]

lemma $\bigwedge\sigma_1$.
(σ_1,$\sigma_1{}'$) ⊨ (*Person .allInstances*() ≐ *Set*{ X_{Person}*1*, X_{Person}*2*, X_{Person}*3*, X_{Person}*4*(∗, X_{Person}*5*∗), X_{Person}*6*,
X_{Person}*7 .oclAsType*(*Person*)(∗, X_{Person}*8*∗), X_{Person}*9* })
apply(*subst perm*-$\sigma_1{}'$)
apply(*simp only*: *oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def*
X_{Person}*1-def* X_{Person}*2-def* X_{Person}*3-def* X_{Person}*4-def*
X_{Person}*5-def* X_{Person}*6-def* X_{Person}*7-def* X_{Person}*8-def* X_{Person}*9-def*
person7-def)
apply(*subst state-update-vs-allInstances-at-post-tc*, *simp*, *simp add*: $OclAsType_{Person}$-𝔄*-def*, *simp*, *rule const-StrictRefEq*$_{Set}$*-including*,
simp, *simp*, *simp*, *rule OclIncluding-cong*, *simp*, *simp*)
apply(*subst state-update-vs-allInstances-at-post-tc*, *simp*, *simp add*: $OclAsType_{Person}$-𝔄*-def*, *simp*, *rule const-StrictRefEq*$_{Set}$*-including*
simp, *simp*, *simp*, *rule OclIncluding-cong*, *simp*, *simp*)
apply(*subst state-update-vs-allInstances-at-post-tc*, *simp*, *simp add*: $OclAsType_{Person}$-𝔄*-def*, *simp*, *rule const-StrictRefEq*$_{Set}$*-including*
simp, *simp*, *simp*, *rule OclIncluding-cong*, *simp*, *simp*)
apply(*subst state-update-vs-allInstances-at-post-tc*, *simp*, *simp add*: $OclAsType_{Person}$-𝔄*-def*, *simp*, *rule const-StrictRefEq*$_{Set}$*-including*
simp, *simp*, *simp*, *rule OclIncluding-cong*, *simp*, *simp*)
apply(*subst state-update-vs-allInstances-at-post-tc*, *simp*, *simp add*: $OclAsType_{Person}$-𝔄*-def*, *simp*, *rule const-StrictRefEq*$_{Set}$*-includin*
simp, *simp*, *simp*, *rule OclIncluding-cong*, *simp*, *simp*)
apply(*subst state-update-vs-allInstances-at-post-tc*, *simp*, *simp add*: $OclAsType_{Person}$-𝔄*-def*, *simp*, *rule const-StrictRefEq*$_{Set}$*-includi*
simp, *simp*, *simp*, *rule OclIncluding-cong*, *simp*, *simp*)

apply(*subst state-update-vs-allInstances-at-post-ntc*, *simp*, *simp add*: $OclAsType_{Person}$-$\mathfrak{A}$-*def*
person8-def, *simp*, *rule const-StrictRefEq*$_{Set}$-*including*, *simp*, *simp*, *simp*)
apply(*subst state-update-vs-allInstances-at-post-tc*, *simp*, *simp add*: $OclAsType_{Person}$-$\mathfrak{A}$-*def*, *simp*, *rule const-StrictRefEq*$_{Set}$-*includi*
simp, *simp*, *simp*, *rule OclIncluding-cong*, *simp*, *simp*)
apply(*rule state-update-vs-allInstances-at-post-empty*)
by(*simp-all add*: $OclAsType_{Person}$-$\mathfrak{A}$-*def*)

lemma $\bigwedge\sigma_1$.
$(\sigma_1,\sigma_1{}') \models (OclAny\ .allInstances() \doteq Set\{\ X_{Person}1\ .oclAsType(OclAny), X_{Person}2\ .oclAsType(OclAny),$
$X_{Person}3\ .oclAsType(OclAny), X_{Person}4\ .oclAsType(OclAny)$
$(*, X_{Person}5*), X_{Person}6\ .oclAsType(OclAny),$
$X_{Person}7, X_{Person}8, X_{Person}9\ .oclAsType(OclAny)\ \})$
apply(*subst perm-*$\sigma_1{}'$)
apply(*simp only*: *oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def*
$X_{Person}1$-*def* $X_{Person}2$-*def* $X_{Person}3$-*def* $X_{Person}4$-*def* $X_{Person}5$-*def* $X_{Person}6$-*def* $X_{Person}7$-*def* $X_{Person}8$-*def*
$X_{Person}9$-*def*
person1-def person2-def person3-def person4-def person5-def person6-def person9-def)
apply(*subst state-update-vs-allInstances-at-post-tc*, *simp*, *simp add*: $OclAsType_{OclAny}$-$\mathfrak{A}$-*def*, *simp*, *rule const-StrictRefEq*$_{Set}$-*including*
simp, *simp*, *simp*, *rule OclIncluding-cong*, *simp*, *simp*)+
apply(*rule state-update-vs-allInstances-at-post-empty*)
by(*simp-all add*: $OclAsType_{OclAny}$-$\mathfrak{A}$-*def*)

end

theory
Design-OCL
imports
Design-UML
begin

B.5.10. OCL Part: Standard State Infrastructure

Ideally, these definitions are automatically generated from the class model.

B.5.11. Invariant

These recursive predicates can be defined conservatively by greatest fix-point constructions—automatically. See [4, 5] for details. For the purpose of this example, we state them as axioms here.

```
context Person
  inv label : self .boss <> null implies (self .salary  \<le>  ((self .boss) .salary))
```

definition $Person\text{-}label_{inv}$ :: *Person* ⇒ *Boolean*
where $Person\text{-}label_{inv}\ (self) \equiv$

$(\mathit{self}\ .\mathit{boss} <> \mathit{null}\ \mathit{implies}\ (\mathit{self}\ .\mathit{salary}\ \leq_{\mathit{int}}\ ((\mathit{self}\ .\mathit{boss})\ .\mathit{salary})))$

definition $\mathit{Person\text{-}label}_{\mathit{invAT\,pre}} :: \mathit{Person} \Rightarrow \mathit{Boolean}$
where $\mathit{Person\text{-}label}_{\mathit{invAT\,pre}}\ (\mathit{self}) \equiv$
$(\mathit{self}\ .\mathit{boss@pre} <> \mathit{null}\ \mathit{implies}\ (\mathit{self}\ .\mathit{salary@pre} \leq_{\mathit{int}} ((\mathit{self}\ .\mathit{boss@pre})\ .\mathit{salary@pre})))$

definition $\mathit{Person\text{-}label}_{\mathit{global\,inv}} :: \mathit{Boolean}$
where $\mathit{Person\text{-}label}_{\mathit{global\,inv}} \equiv (\mathit{Person}\ .\mathit{allInstances}()\text{-}{>}\mathit{forAll}(x \mid \mathit{Person\text{-}label}_{\mathit{inv}}\ (x))\ \mathit{and}$
$(\mathit{Person}\ .\mathit{allInstances@pre}()\text{-}{>}\mathit{forAll}(x \mid \mathit{Person\text{-}label}_{\mathit{invAT\,pre}}\ (x))))$

lemma $\tau \models \delta\ (X\ .\mathit{boss}) \Longrightarrow \tau \models \mathit{Person}\ .\mathit{allInstances}()\text{-}{>}\mathit{includes}(X\ .\mathit{boss}) \wedge$
$\tau \models \mathit{Person}\ .\mathit{allInstances}()\text{-}{>}\mathit{includes}(X)$
sorry

lemma $\mathit{REC\text{-}pre} : \tau \models \mathit{Person\text{-}label}_{\mathit{global\,inv}}$
$\Longrightarrow \tau \models \mathit{Person}\ .\mathit{allInstances}()\text{-}{>}\mathit{includes}(X)\ (*\ X\ \mathit{represented\ object\ in\ state}\ *)$
$\Longrightarrow \exists\ \mathit{REC}.\ \ \tau \models \mathit{REC}(X)\ \triangleq (\mathit{Person\text{-}label}_{\mathit{inv}}\ (X)\ \mathit{and}\ (X\ .\mathit{boss} <> \mathit{null}\ \mathit{implies}\ \mathit{REC}(X\ .\mathit{boss})))$
sorry

This allows to state a predicate:

axiomatization $\mathit{inv}_{\mathit{Person\text{-}label}} :: \mathit{Person} \Rightarrow \mathit{Boolean}$
where $\mathit{inv}_{\mathit{Person\text{-}label}}\text{-}\mathit{def}$:
$(\tau \models \mathit{Person}\ .\mathit{allInstances}()\text{-}{>}\mathit{includes}(\mathit{self})) \Longrightarrow$
$(\tau \models (\mathit{inv}_{\mathit{Person\text{-}label}}(\mathit{self}) \triangleq\ (\mathit{self}\ .\mathit{boss} <> \mathit{null}\ \mathit{implies}$
$(\mathit{self}\ .\mathit{salary}\ \leq_{\mathit{int}}\ ((\mathit{self}\ .\mathit{boss})\ .\mathit{salary}))\ \mathit{and}$
$\mathit{inv}_{\mathit{Person\text{-}label}}(\mathit{self}\ .\mathit{boss}))))$

axiomatization $\mathit{inv}_{\mathit{Person\text{-}labelAT\,pre}} :: \mathit{Person} \Rightarrow \mathit{Boolean}$
where $\mathit{inv}_{\mathit{Person\text{-}labelAT\,pre}}\text{-}\mathit{def}$:
$(\tau \models \mathit{Person}\ .\mathit{allInstances@pre}()\text{-}{>}\mathit{includes}(\mathit{self})) \Longrightarrow$
$(\tau \models (\mathit{inv}_{\mathit{Person\text{-}labelAT\,pre}}(\mathit{self}) \triangleq (\mathit{self}\ .\mathit{boss@pre} <> \mathit{null}\ \mathit{implies}$
$(\mathit{self}\ .\mathit{salary@pre}\ \leq_{\mathit{int}}\ ((\mathit{self}\ .\mathit{boss@pre})\ .\mathit{salary@pre}))\ \mathit{and}$
$\mathit{inv}_{\mathit{Person\text{-}labelAT\,pre}}(\mathit{self}\ .\mathit{boss@pre}))))$

lemma $\mathit{inv\text{-}1}$:
$(\tau \models \mathit{Person}\ .\mathit{allInstances}()\text{-}{>}\mathit{includes}(\mathit{self})) \Longrightarrow$
$(\tau \models \mathit{inv}_{\mathit{Person\text{-}label}}(\mathit{self}) = ((\tau \models (\mathit{self}\ .\mathit{boss} \doteq \mathit{null})) \vee$
$(\ \tau \models (\mathit{self}\ .\mathit{boss} <> \mathit{null}) \wedge$
$\tau \models ((\mathit{self}\ .\mathit{salary})\ \leq_{\mathit{int}}\ (\mathit{self}\ .\mathit{boss}\ .\mathit{salary}))\ \wedge$
$\tau \models (\mathit{inv}_{\mathit{Person\text{-}label}}(\mathit{self}\ .\mathit{boss})))))$
sorry

lemma $\mathit{inv\text{-}2}$:

$(\tau \models Person\ .allInstances@pre() -> includes(self)) \Longrightarrow$
$\quad (\tau \models inv_{Person\text{-}labelATpre}(self)) = \ ((\tau \models (self\ .boss@pre \doteq null)) \lor$
$\qquad\qquad (\tau \models (self\ .boss@pre <> null) \land$
$\qquad\qquad (\tau \models (self\ .boss@pre\ .salary@pre \leq_{int} self\ .salary@pre)) \ \land$
$\qquad\qquad (\tau \models (inv_{Person\text{-}labelATpre}(self\ .boss@pre)))))$
sorry

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 $inv :: Person \Rightarrow (\mathfrak{A})st \Rightarrow bool$ **where**
$(\tau \models (\delta\ self)) \Longrightarrow ((\tau \models (self\ .boss \doteq null)) \lor$
$\qquad (\tau \models (self\ .boss <> null) \land (\tau \models (self\ .boss\ .salary \leq_{int} self\ .salary)) \ \land$
$\qquad ((inv(self\ .boss))\tau\ )))$
$\qquad \Longrightarrow (\ inv\ self\ \tau)$

B.5.12. The Contract of a Recursive Query

This part is analogous to the Analysis Model and skipped here.

end

C. Conclusion

C.1. Lessons Learned and Contributions

We provided a typed and type-safe shallow embedding of the core of UML [26, 27] and OCL [28]. Shallow embedding means that types of OCL were injectively, i. e., mapped by the embedding one-to-one to types in Isabelle/HOL [25]. We followed the usual methodology to build up the theory uniquely by conservative extensions of all operators in a denotational style and to derive logical and algebraic (execution) rules from them; thus, we can guarantee the logical consistency of the library and instances of the class model construction, i. e., closed-world object-oriented datatype theories, as long as it follows the described methodology.[1] Moreover, all derived execution rules are by construction type-safe (which would be an issue, if we had chosen to use an object universe construction in Zermelo-Fraenkel set theory as an alternative approach to subtyping.). In more detail, our theory gives answers and concrete solutions to a number of open major issues for the UML/OCL standardization:

1. the role of the two exception elements invalid and null, the former usually assuming strict evaluation while the latter ruled by non-strict evaluation.
2. the functioning of the resulting four-valued logic, together with safe rules (for example foundation9 – foundation12 in Section B.2.1) that allow a reduction to two-valued reasoning as required for many automated provers. The resulting logic still enjoys the rules of a strong Kleene Logic in the spirit of the Amsterdam Manifesto [17].
3. the complicated life resulting from the two necessary equalities: the standard's "strict weak referential equality" as default (written $\_ \doteq \_$ throughout this document) and the strong equality (written $\_ \triangleq \_$), which follows the logical Leibniz principle that "equals can be replaced by equals." Which is not necessarily the case if invalid or objects of different states are involved.
4. a type-safe representation of objects and a clarification of the old idea of a one-to-one correspondence between object representations and object-id's, which became a state invariant.
5. a simple concept of state-framing via the novel operator `_->oclIsModifiedOnly()` and its consequences for strong and weak equality.
6. a semantic view on subtyping clarifying the role of static and dynamic type (aka *apparent* and *actual* type in Java terminology), and its consequences for casts, dynamic type-tests, and static types.

[1] Our two examples of Employee_AnalysisModel and Employee_DesignModel (see Section B.4 and Figure B as well as Section B.5 and Figure B) sketch how this construction can be captured by an automated process.

7. a semantic view on path expressions, that clarify the role of invalid and null as well as the tricky issues related to de-referentiation in pre- and post state.

8. an optional extension of the OCL semantics by *infinite* sets that provide means to represent "the set of potential objects or values" to state properties over them (this will be an important feature if OCL is intended to become a full-blown code annotation language in the spirit of JML [23] for semi-automated code verification, and has been considered desirable in the Aachen Meeting [13]).

Moreover, we managed to make our theory in large parts executable, which allowed us to include mechanically checked value-statements that capture numerous corner-cases relevant for OCL implementors. Among many minor issues, we thus pin-pointed the behavior of `null` in collections as well as in casts and the desired `isKindOf`-semantics of `allInstances()`.

C.2. Lessons Learned

While our paper and pencil arguments, given in [11], turned out to be essentially correct, there had also been a lesson to be learned: If the logic is not defined as a Kleene-Logic, having a structure similar to a complete partial order (CPO), reasoning becomes complicated: several important algebraic laws break down which makes reasoning in OCL inherent messy and a semantically clean compilation of OCL formulae to a two-valued presentation, that is amenable to animators like KodKod [31] or SMT-solvers like Z3 [18] completely impractical. Concretely, if the expression `not(null)` is defined `invalid` (as is the case in the present standard [28]), than standard involution does not hold, i. e., `not(not(A)) = A` does not hold universally. Similarly, if `null and null` is `invalid`, then not even idempotence `X and X = X` holds. We strongly argue in favor of a lattice-like organization, where `null` represents "more information" than `invalid` and the logical operators are monotone with respect to this semantical "information ordering."

A similar experience with prior paper and pencil arguments was our investigation of the object-oriented data-models, in particular path-expressions [14]. The final presentation is again essentially correct, but the technical details concerning exception handling lead finally to a continuation-passing style of the (in future generated) definitions for accessors, casts and tests. Apparently, OCL semantics (as many other "real" programming and specification languages) is meanwhile too complex to be treated by informal arguments solely.

Featherweight OCL makes several minor deviations from the standard and showed how the previous constructions can be made correct and consistent, and the DNF-normalization as well as δ-closure laws (necessary for a transition into a two-valued presentation of OCL specifications ready for interpretation in SMT solvers (see [12] for details)) are valid in Featherweight OCL.

C.3. Conclusion and Future Work

Featherweight OCL concentrates on formalizing the semantics of a core subset of OCL in general and in particular on formalizing the consequences of a four-valued logic (i. e., OCL versions that support, besides the truth values `true` and `false` also the two exception values `invalid` and `null`).

In the following, we outline the necessary steps for turning Featherweight OCL into a fully fledged tool for OCL, e. g., similar to HOL-OCL as well as for supporting test case generation similar to HOL-TestGen [8]. There are essentially five extensions necessary:

- extension of the library to support all OCL data types, e. g., `OrderedSet(T)` or `Sequence(T).` This formalization of the OCL standard library can be used for checking the consistency of the formal semantics (known as "Annex A") with the informal and semi-formal requirements in the normative part of the OCL standard.

- development of a compiler that compiles a textual or CASE tool representation (e. g., using XMI or the textual syntax of the USE tool [30]) of class models. Such compiler could also generate the necessary casts when converting standard OCL to Featherweight OCL as well as providing "normalizations" such as converting multiplicities of class attributes to into OCL class invariants.

- a setup for translating Featherweight OCL into a two-valued representation as described in [12]. As, in real-world scenarios, large parts of UML/OCL specifications are defined (e. g., from the default multiplicity `1` of an attributes `x`, we can directly infer that for all valid states `x` is neither `invalid` nor `null`), such a translation enables an efficient test case generation approach.

- a setup in Featherweight OCL of the Nitpick animator [3]. It remains to be shown that the standard, Kodkod [31] based animator in Isabelle can give a similar quality of animation as the OCLexec Tool [22]

- a code-generator setup for Featherweight OCL for Isabelle's code generator. For example, the Isabelle code generator supports the generation of F#, which would allow to use OCL specifications for testing arbitrary .net-based applications.

The first two extensions are sufficient to provide a formal proof environment for OCL 2.5 similar to HOL-OCL while the remaining extensions are geared towards increasing the degree of proof automation and usability as well as providing a tool-supported test methodology for UML/OCL.

Our work shows that developing a machine-checked formal semantics of recent OCL standards still reveals significant inconsistencies—even though this type of research is not new. In fact, we started our work already with the 1.x series of OCL. The reasons for this ongoing consistency problems of OCL standard are manifold. For example, the consequences of adding an additional exception value to OCL 2.2 are widespread across the whole language and many of them are also quite subtle. Here, a machine-checked formal semantics is of great value, as one is forced to formalize all details and subtleties. Moreover, the standardization process of the OMG, in which standards (e. g., the UML infrastructure and the OCL standard) that need to be aligned closely are developed quite independently, are prone to ad-hoc changes that attempt to align these standards. And, even worse, updating a standard document by voting on the acceptance (or rejection) of isolated text changes does not help either. Here, a tool for the editor of the standard that helps to check the consistency of the whole standard after each and every modifications can be of great value as well.

Bibliography

[1] P. B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Kluwer Academic Publishers, Dordrecht, 2nd edition, 2002. ISBN 1-402-00763-9.

[2] C. Barrett and C. Tinelli. Cvc3. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 2007. ISBN 978-3-540-73367-6. doi: 10.1007/978-3-540-73368-3_34.

[3] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. C. Paulson, editors, *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer-Verlag, 2010. ISBN 978-3-642-14051-8. doi: 10.1007/978-3-642-14052-5_11.

[4] A. D. Brucker. *An Interactive Proof Environment for Object-oriented Specifications*. PhD thesis, ETH Zurich, Mar. 2007. URL http://www.brucker.ch/bibliography/abstract/brucker-interactive-2007. ETH Dissertation No. 17097.

[5] A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-ocl-book-2006.

[6] A. D. Brucker and B. Wolff. An extensible encoding of object-oriented data models in hol. *Journal of Automated Reasoning*, 41:219–249, 2008. ISSN 0168-7433. doi: 10.1007/s10817-008-9108-3. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-extensible-2008-b.

[7] A. D. Brucker and B. Wolff. HOL-OCL – A Formal Proof Environment for UML/OCL. In J. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering (FASE08)*, number 4961 in Lecture Notes in Computer Science, pages 97–100. Springer-Verlag, Heidelberg, 2008. doi: 10.1007/978-3-540-78743-3_8. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-ocl-2008.

[8] A. D. Brucker and B. Wolff. HOL-TestGen: An interactive test-case generation framework. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering (FASE09)*, number 5503 in Lecture Notes in Computer Science, pages 417–420. Springer-Verlag, Heidelberg, 2009. doi: 10.1007/978-3-642-00593-0_28. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-testgen-2009.

[9] A. D. Brucker, J. Doser, and B. Wolff. Semantic issues of OCL: Past, present, and future. *Electronic Communications of the EASST*, 5, 2006. ISSN 1863-2122. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-semantic-2006-b.

[10] A. D. Brucker, J. Doser, and B. Wolff. A model transformation semantics and analysis methodology for SecureUML. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS 2006: Model*

Driven Engineering Languages and Systems, number 4199 in Lecture Notes in Computer Science, pages 306–320. Springer-Verlag, Heidelberg, 2006. doi: 10.1007/11880240_22. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-transformation-2006. An extended version of this paper is available as ETH Technical Report, no. 524.

[11] A. D. Brucker, M. P. Krieger, and B. Wolff. Extending OCL with null-references. In S. Gosh, editor, *Models in Software Engineering*, number 6002 in Lecture Notes in Computer Science, pages 261–275. Springer-Verlag, Heidelberg, 2009. doi: 10.1007/978-3-642-12261-3_25. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-ocl-null-2009. Selected best papers from all satellite events of the MoDELS 2009 conference.

[12] A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff. A specification-based test case generation method for UML/OCL. In J. Dingel and A. Solberg, editors, *MoDELS Workshops*, number 6627 in Lecture Notes in Computer Science, pages 334–348. Springer-Verlag, Heidelberg, 2010. ISBN 978-3-642-21209-3. doi: 10.1007/978-3-642-21210-9_33. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-ocl-testing-2010. Selected best papers from all satellite events of the MoDELS 2010 conference. Workshop on OCL and Textual Modelling.

[13] A. D. Brucker, D. Chiorean, T. Clark, B. Demuth, M. Gogolla, D. Plotnikov, B. Rumpe, E. D. Willink, and B. Wolff. Report on the Aachen OCL meeting. In J. Cabot, M. Gogolla, I. Rath, and E. Willink, editors, *Proceedings of the MODELS 2013 OCL Workshop (OCL 2013)*, volume 1092 of *CEUR Workshop Proceedings*, pages 103–111. CEUR-WS.org, 2013. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-summary-aachen-2013.

[14] A. D. Brucker, D. Longuet, F. Tuong, and B. Wolff. On the semantics of object-oriented data structures and path expressions. In *OCL@MoDELS*, pages 23–32, 2013.

[15] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[16] T. Clark and J. Warmer, editors. *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *Lecture Notes in Computer Science*, Heidelberg, 2002. Springer-Verlag. ISBN 3-540-43169-1.

[17] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The amsterdam manifesto on OCL. In Clark and Warmer [16], pages 115–149. ISBN 3-540-43169-1.

[18] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-78799-0. doi: 10.1007/978-3-540-78800-3_24.

[19] M. Gogolla and M. Richters. Expressing UML class diagrams properties with OCL. In Clark and Warmer [16], pages 85–114. ISBN 3-540-43169-1.

[20] A. Hamie, F. Civello, J. Howse, S. Kent, and R. Mitchell. Reflections on the Object Constraint Language. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language. «UML»'98: Beyond the Notation*,

volume 1618 of *Lecture Notes in Computer Science*, pages 162–172, Heidelberg, 1998. Springer-Verlag. ISBN 3-540-66252-9. doi: 10.1007/b72309.

[21] P. Kosiuczenko. Specification of invariability in OCL. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *Lecture Notes in Computer Science*, pages 676–691, Heidelberg, 2006. Springer-Verlag. ISBN 978-3-540-45772-5. doi: 10.1007/11880240_47.

[22] M. P. Krieger, A. Knapp, and B. Wolff. Generative programming and component engineering. In E. Visser and J. Järvi, editors, *International Conference on Generative Programming and Component Engineering (GPCE 2010)*, pages 53–62. ACM, Oct. 2010. ISBN 978-1-4503-0154-1.

[23] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML reference manual (revision 1.2), Feb. 2007. Available from http://www.jmlspecs.org.

[24] L. Mandel and M. V. Cengarle. On the expressive power of OCL. In J. M. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems (FM)*, volume 1708 of *Lecture Notes in Computer Science*, pages 854–874, Heidelberg, 1999. Springer-Verlag. ISBN 3-540-66587-0.

[25] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2002. doi: 10.1007/3-540-45949-9.

[26] Object Management Group. UML 2.4.1: Infrastructure specification, Aug. 2011. Available as OMG document formal/2011-08-05.

[27] Object Management Group. UML 2.4.1: Superstructure specification, Aug. 2011. Available as OMG document formal/2011-08-06.

[28] Object Management Group. UML 2.3.1 OCL specification, Feb. 2012. Available as OMG document formal/2012-01-01.

[29] A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *CADE*, volume 1632 of *Lecture Notes in Computer Science*, pages 292–296. Springer-Verlag, 1999. ISBN 3-540-66222-7. doi: 10.1007/3-540-48660-7_26.

[30] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[31] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-71208-4. doi: 10.1007/978-3-540-71209-1_49.

[32] M. Wenzel and B. Wolff. Building formal method tools in the Isabelle/Isar framework. In K. Schneider and J. Brandt, editors, *TPHOLs 2007*, number 4732 in Lecture Notes in Computer Science, pages 352–367. Springer-Verlag, Heidelberg, 2007. doi: 10.1007/978-3-540-74591-4_26.

[33] M. M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, TU München, München, Feb. 2002. URL http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html.

Contents