

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 existence 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.Second()` 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 rules². And obviously, only a mechanized check of these definitions, following a rigorous 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

¹An 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

²The 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 created³. 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.

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

(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.

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

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

- All objects types are represented in an object universe⁵. 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.

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

FixMe:
*Something
 like this ?
 Shorten
 Para-
 graph
 !*

- 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,

⁵following the tradition of HOL-OCL [6]

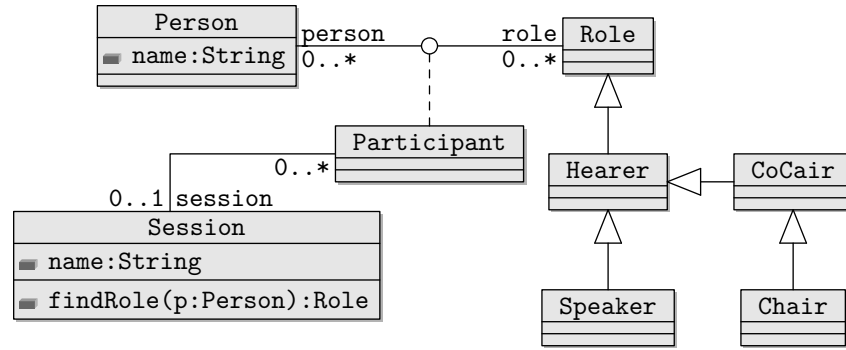


Figure A.1.: A simple UML class model representing a conference system for organizing conference sessions: persons can participate, in different roles, in a session.

- 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 operation constructs.
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

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

FiXme:
REWRITE
THIS
FOR THE
ANNEX
A:
SHORTEN!

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 $A_1 \Longrightarrow \dots \Longrightarrow A_n \Longrightarrow A_{n+1}$, 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

$$\llbracket A_1; \dots; A_n \rrbracket \Longrightarrow A_{n+1} \quad \text{or, in mathematical notation,} \quad \frac{A_1 \quad \dots \quad A_n}{A_{n+1}}. \quad (\text{A.2})$$

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:

$$\bigwedge x_1, \dots, x_m. \llbracket A_1; \dots; A_n \rrbracket \Longrightarrow A_{n+1}. \quad (\text{A.3})$$

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:  $\phi$ 
  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*) ϕ_1, \dots, ϕ_n and a *goal* ϕ . Proof states were usually denoted by:

$$\begin{array}{l} \text{label : } \phi \\ 1. \quad \phi_1 \\ \vdots \\ n. \quad \phi_n \end{array} \quad (\text{A.5})$$

Subgoals and goals may be extracted from the proof state into theorems of the form $\llbracket \phi_1; \dots; \phi_n \rrbracket \Longrightarrow \phi$ at any time; this mechanism helps to generate test theorems. Further, Isabelle supports meta-variables (written $\text{?}x, \text{?}y, \dots$), 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 $\forall x. Px$ and $\exists x. Px$; 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 $\tau_\perp := \perp \mid _ _$: α that assigns to each type τ a type τ_\perp *disjointly extended* by the exceptional element \perp . The function $_ _ : \alpha_\perp \rightarrow \alpha$ is the inverse of $_ _$ (unspecified for \perp). Partial functions $\alpha \rightarrow \beta$ are defined as functions $\alpha \Rightarrow \beta_\perp$ supporting the usual concepts of domain ($\text{dom } _$) and range ($\text{ran } _$).

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

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

Isabelle’s syntax engine is instructed to accept the notation $\{x \mid P\}$ for $\text{Collect } \lambda x. P$ and the notation $s \in S$ for $\text{member } s S$. 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

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

straightforward to express the usual operations on sets like $_ \cup _, _ \cap _ :: \alpha \text{ set} \Rightarrow \alpha \text{ set} \Rightarrow \alpha \text{ set}$ 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:

$$\begin{aligned} \text{datatype option} &= \text{None} \mid \text{Some } \alpha \\ \text{datatype } \alpha \text{ list} &= \text{Nil} \mid \text{Cons } a \, l \end{aligned} \quad (\text{A.7})$$

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

$$\text{case } x \text{ of } \text{None} \Rightarrow F \mid \text{Some } a \Rightarrow G a \quad (\text{A.8})$$

respectively

$$\text{case } x \text{ of } [] \Rightarrow F \mid \text{Cons } a \, r \Rightarrow G a \, r. \quad (\text{A.9})$$

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

$$\begin{aligned} &(\text{case } [] \text{ of } [] \Rightarrow F \mid (a\#r) \Rightarrow G a \, r) = F \\ &(\text{case } b\#t \text{ of } [] \Rightarrow F \mid (a\#r) \Rightarrow G a \, r) = G b \, t \\ &[] \neq a\#t \quad \text{-- distinctness} \\ &\llbracket a = [] \rightarrow P; \exists x \, t. a = x\#t \rightarrow P \rrbracket \Longrightarrow P \quad \text{-- exhaust} \\ &\llbracket P[]; \forall a \, t. P t \rightarrow P(a\#t) \rrbracket \Longrightarrow P x \quad \text{-- induct} \end{aligned} \quad (\text{A.10})$$

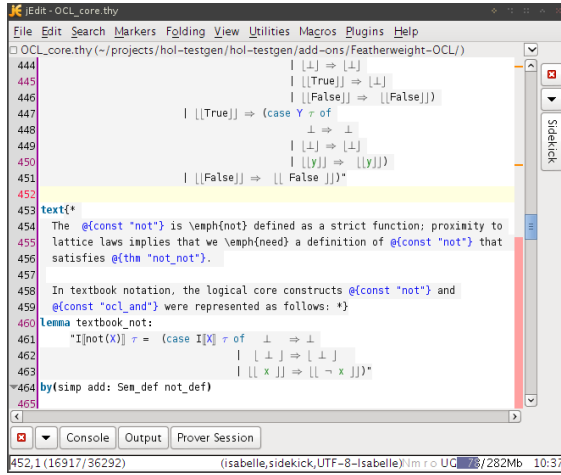
Finally, there is a compiler for primitive and wellfounded recursive function definitions. For example, we may define the sort operation of our running test example by:

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

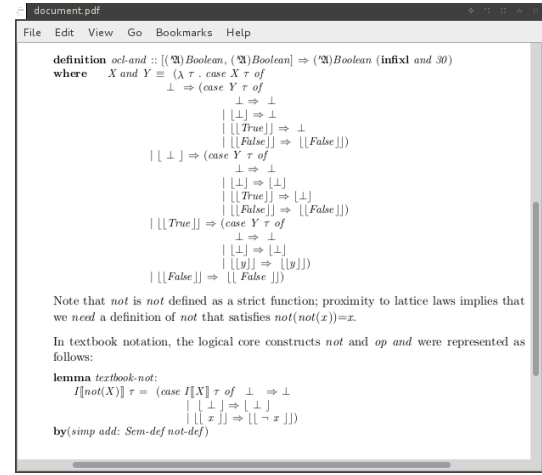
$$\begin{aligned} \text{fun sort} &:: (\alpha :: \text{linorder}) \text{ list} \Rightarrow \alpha \text{ list} \\ \text{where sort } [] &= [] \\ \text{sort } (x\#xs) &= \text{ins } x \, (\text{sort } xs) \end{aligned} \quad (\text{A.12})$$

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.



(a) The Isabelle jEdit environment.



(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

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., $\text{not } (\text{not } x) = x$.

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].

FiXme:
Here ? Or
in chap 2
?

A.3. The Essence of UML-OCL Semantics

A.3.1. The Theory Organization

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 logical 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 $\text{TYPES}(C)$
- Sequence(X), Set(X), et Pair(X, Y) (as example for a Tuple-type) are in $\text{TYPES}(C)$ (if $X, Y \in \text{TYPES}(C)$).

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 confusion 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:

$$\text{type_synonym} \quad \text{Boolean}_{\text{base}} := \text{bool}_{\perp\perp} \quad (\text{A.13})$$

As a consequence of this definition of the type, we have the elements $\perp, \perp_{\perp}, \perp_{\text{true}}, \perp_{\text{false}}$ 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 \perp_{\perp} for the generic type class `null`. For collection types and object types this definition is more evolved (see Section B.1.1).

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

FiXme:
should we
use
explicit
definitions
?

FiXme:
why does
backslash
null not
work here
?

For object base types, we assume a typed universe $??$ 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:

$$\text{type}_{\text{synonym}} \quad V_{??}(\alpha) := \text{state}(??) \times \text{state}(??) \rightarrow \alpha :: \text{null} . \quad (\text{A.14})$$

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

$$\begin{aligned} \text{type}_{\text{synonym}} \quad \text{Boolean}_{??} &:= V_{??}(\text{Boolean}_{\text{base}}) \\ \text{type}_{\text{synonym}} \quad \text{Integer}_{??} &:= V_{??}(\text{Integer}_{\text{base}}) \\ &\dots \end{aligned}$$

the other cases are analogous. In the subsequent subsections, we will drop the index $??$ 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 $I\llbracket E \rrbracket \tau$ 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:

$$I\llbracket \text{invalid} :: V(\alpha) \rrbracket \tau \equiv \text{bot} \quad I\llbracket \text{null} :: V(\alpha) \rrbracket \tau \equiv \text{null}$$

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):

$$\begin{aligned} I\llbracket \text{true} :: \text{Boolean} \rrbracket \tau &= \underline{\text{true}} \\ I\llbracket \text{false} \rrbracket \tau &= \underline{\text{false}} \\ I\llbracket X.\text{oclIsUndefined}() \rrbracket \tau &= (\text{if } I\llbracket X \rrbracket \tau \in \{\text{bot}, \text{null}\} \text{ then } I\llbracket \text{true} \rrbracket \tau \text{ else } I\llbracket \text{false} \rrbracket \tau) \\ I\llbracket X.\text{oclIsValid}() \rrbracket \tau &= (\text{if } I\llbracket X \rrbracket \tau = \text{bot} \text{ then } I\llbracket \text{true} \rrbracket \tau \text{ else } I\llbracket \text{false} \rrbracket \tau) \end{aligned}$$

For reasons of conciseness, we will write δX for $\text{not}(X.\text{oclIsUndefined}())$ and $v X$ for $\text{not}(X.\text{oclIsValid}())$ 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:

$$I\llbracket \text{true} :: \text{Boolean} \rrbracket \tau = \underline{\text{true}}$$

we can therefore write:

$$\text{true} :: \text{Boolean} = \lambda \tau. \underline{\text{true}}$$

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:

$$I[\text{not } X]\tau = (\text{case } I[X]\tau \text{ of} \\ \perp \Rightarrow \perp \\ |[\perp] \Rightarrow [\perp] \\ |[\![x]\!] \Rightarrow [\![\neg x]\!])$$

$$I[X \text{ and } Y]\tau = (\text{case } I[X]\tau \text{ of} \\ \perp \Rightarrow (\text{case } I[Y]\tau \text{ of} \\ \perp \Rightarrow \perp \\ |[\perp] \Rightarrow \perp \\ |[\![\text{true}]\!] \Rightarrow \perp \\ |[\![\text{false}]\!] \Rightarrow [\![\text{false}]\!]]) \\ |[\perp] \Rightarrow (\text{case } I[Y]\tau \text{ of} \\ \perp \Rightarrow \perp \\ |[\perp] \Rightarrow [\perp] \\ |[\![\text{true}]\!] \Rightarrow [\perp] \\ |[\![\text{false}]\!] \Rightarrow [\![\text{false}]\!]]) \\ |[\![\text{true}]\!] \Rightarrow (\text{case } I[Y]\tau \text{ of} \\ \perp \Rightarrow \perp \\ |[\perp] \Rightarrow [\perp] \\ |[\![y]\!] \Rightarrow [\![y]\!]]) \\ |[\![\text{false}]\!] \Rightarrow [\![\text{false}]\!])$$

FixMe:
we must
uni-
formize
the list -
vs. `lfloor`
notation.
Either the
one or the
other.

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

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:

$$I[x + y]\tau = \text{if } I[\delta x]\tau = I[\text{true}]\tau \wedge I[\delta y]\tau = I[\text{true}]\tau \\ \text{then } [\![\![I[x]\tau]\!] + \![I[y]\tau]\!]\!] \\ \text{else } \perp$$

where the operator “+” on the left-hand side of the equation denotes the OCL addition of type `Integer` \Rightarrow `Integer` while the “+” on the right-hand side of the equation of type `[int,int]` \Rightarrow `int` denotes the integer-addition from the HOL library.

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

$$I[S \text{-->including}(y)]\tau = \text{if } I[\delta S]\tau = I[\text{true}]\tau \wedge I[v y]\tau = I[\text{true}]\tau \\ \text{then } \text{Abs_Set}_{\text{base}} \ulcorner \text{Rep_Set}_{\text{base}} I[S]\tau \urcorner \cup \{I[y]\tau\} \\ \text{else } \perp$$

Here, the operator $_ \cup _$ stems from the HOL set theory, together with the set inclusion $\{ _ \}$. The operator Abs_Set_base is the constructor for the FeatherweightOCL Set type, whereas Rep_Set_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 must be possible and since the referential equality should be symmetric, should be allowed for *both* arguments and the expression:

$$\text{null} = \text{null} \quad (\text{A.15})$$

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*:

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

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:

$$\tau \models P \equiv (I[P]\tau = \underline{\text{true}}).$$

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:

$$\begin{aligned} \tau \models \text{true} \quad & \neg(\tau \models \text{false}) \quad \neg(\tau \models \text{invalid}) \quad \neg(\tau \models \text{null}) \\ \tau \models \text{not } P & \implies \neg(\tau \models P) \\ \tau \models P \text{ and } Q & \implies \tau \models P \quad \tau \models P \text{ and } Q \implies \tau \models Q \\ \tau \models P \implies \tau \models P \text{ or } Q & \quad \tau \models Q \implies \tau \models P \text{ or } Q \\ \tau \models P \implies (\text{if } P \text{ then } B_1 \text{ else } B_2 \text{ endif}) \tau & = B_1 \tau \\ \tau \models \text{not } P \implies (\text{if } P \text{ then } B_1 \text{ else } B_2 \text{ endif}) \tau & = B_2 \tau \\ \tau \models P \implies \tau \models \delta P \quad \tau \models \delta X \implies \tau \models v X \end{aligned}$$

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:

$$I[X \triangleq Y] \tau \equiv \llbracket I[X] \tau \rrbracket = I[Y] \tau$$

It enjoys nearly the laws of a congruence:

$$\begin{aligned} \tau &\models (x \triangleq x) \\ \tau &\models (x \triangleq y) \implies \tau \models (y \triangleq x) \\ \tau &\models (x \triangleq y) \implies \tau \models (y \triangleq z) \implies \tau \models (x \triangleq z) \\ \text{cp } P &\implies \tau \models (x \triangleq y) \implies \tau \models (Px) \implies \tau \models (Py) \end{aligned}$$

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:

$$\begin{aligned} \tau &\models \delta x \vee \tau \models x \triangleq \text{invalid} \vee \tau \models x \triangleq \text{null}, \\ (\tau \models A \triangleq \text{invalid}) &= (\tau \models \text{not}(vA)) \\ (\tau \models A \triangleq \text{true}) &= (\tau \models A) \quad (\tau \models A \triangleq \text{false}) = (\tau \models \text{not}A) \\ (\tau \models \text{not}(\delta x)) &= (\neg \tau \models \delta x) \quad (\tau \models \text{not}(vx)) = (\neg \tau \models vx) \end{aligned}$$

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.:

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

Together with the already mentioned general case-distinction

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

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

⁷Strong logical equality is also referred as “Leibniz”-equality.

usually quickly to contradictions. For example, we can infer from an invariant $\tau \models x \dot{=} y - 3$ that we have $\tau \models x \dot{=} y - 3 \wedge \tau \models \delta x \wedge \tau \models \delta y$. We call the latter formula the δ -closure of the former. Now, we can convert a formula like $\tau \models x > 0 \text{ or } 3 * y > x * x$ 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:

$$\begin{array}{ll}
v \text{ invalid} = \text{false} & v \text{ null} = \text{true} \\
v \text{ true} = \text{true} & v \text{ false} = \text{true} \\
\delta \text{ invalid} = \text{false} & \delta \text{ null} = \text{false} \\
\delta \text{ true} = \text{true} & \delta \text{ false} = \text{true} \\
\text{not invalid} = \text{invalid} & \text{not null} = \text{null} \\
\text{not true} = \text{false} & \text{not false} = \text{true} \\
(\text{null and true}) = \text{null} & (\text{null and false}) = \text{false} \\
(\text{null and null}) = \text{null} & (\text{null and invalid}) = \text{invalid} \\
(\text{false and true}) = \text{false} & (\text{false and false}) = \text{false} \\
(\text{false and null}) = \text{false} & (\text{false and invalid}) = \text{false} \\
(\text{true and true}) = \text{true} & (\text{true and false}) = \text{false} \\
(\text{true and null}) = \text{null} & (\text{true and invalid}) = \text{invalid} \\
(\text{invalid and true}) = \text{invalid} & \\
(\text{invalid and false}) = \text{false} & \\
(\text{invalid and null}) = \text{invalid} & \\
(\text{invalid and invalid}) = \text{invalid} &
\end{array}$$

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

$$\begin{array}{ll}
X \text{ and } X = X & X \text{ and } Y = Y \text{ and } X \\
\text{false and } X = \text{false} & X \text{ and false} = \text{false} \\
\text{true and } X = X & X \text{ and true} = X \\
X \text{ and } (Y \text{ and } Z) = X \text{ and } Y \text{ and } Z &
\end{array}$$

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:

$$\begin{aligned}
& \text{invalid} + X = \text{invalid} & X + \text{invalid} = \text{invalid} \\
& \text{invalid} \rightarrow \text{including}(X) = \text{invalid} & \text{null} \rightarrow \text{including}(X) = \text{invalid} \\
& X \dot{=} \text{invalid} = \text{invalid} & \text{invalid} \dot{=} X = \text{invalid} \\
& S \rightarrow \text{including}(\text{invalid}) = \text{invalid} \\
& X \dot{=} X = (\text{if } \mathbf{v} x \text{ then true else invalid endif}) \\
& 1 / 0 = \text{invalid} & 1 / \text{null} = \text{null} \\
& \text{invalid} \rightarrow \text{isEmpty}() = \text{invalid} & \text{null} \rightarrow \text{isEmpty}() = \text{null}
\end{aligned}$$

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

$$\begin{aligned}
& \delta \text{Set}\{\} = \text{true} \\
& \delta (X \rightarrow \text{including}(x)) = \delta X \text{ and } \mathbf{v} x \\
& \text{Set}\{\} \rightarrow \text{includes}(x) = (\text{if } \mathbf{v} x \text{ then false} \\
& \hspace{15em} \text{else invalid endif}) \\
& (X \rightarrow \text{including}(x) \rightarrow \text{includes}(y)) = \\
& \hspace{4em} (\text{if } \delta X \\
& \hspace{6em} \text{then if } x \dot{=} y \\
& \hspace{8em} \text{then true} \\
& \hspace{8em} \text{else } X \rightarrow \text{includes}(y) \\
& \hspace{6em} \text{endif} \\
& \hspace{4em} \text{else invalid} \\
& \hspace{4em} \text{endif})
\end{aligned}$$

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

$$\text{Set}\{\} \rightarrow \text{including}(1) \rightarrow \text{including}(2)$$

an expression like $\text{Set}\{1, 2\} \rightarrow \text{includes}(\text{null})$ becomes decidable in FeatherweightOCL by applying these algebraic laws (which can give rise to efficient compilations). The reader interested in the list of “test-statements” like:

$$\text{value } \tau \models (\text{Set}\{\text{Set}\{2, \text{null}\}\} \dot{=} \text{Set}\{\text{Set}\{\text{null}, 2\}\})$$

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 $(C, _ < _ , \text{Attrib}, \text{Assoc})$ 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: $C_i.\text{allInstances}()$ and $C_i.\text{allInstances}@\text{pre}()$,
2. $_ < _$ is an inheritance relation on classes,
3. $\text{Attrib}(C_i)$ is a collection of attributes associated to classes C_i . It declares two families of accessors; for each attribute $a \in \text{Attrib}(C_i)$ in a class definition C_i (denoted $X.a :: C_i \rightarrow A$ and $X.a@\text{pre} :: C_i \rightarrow A$ for $A \in \text{TYPES}(C)$),
4. $\text{Assoc}(C_i, C_j)$ is a collection of associations.⁸ An association $(n, rn_{from}, rn_{to}) \in \text{Assoc}(C_i, C_j)$ 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 rolenome belong two families of accessors denoted $X.a :: C_i \rightarrow A$ and $X.a@\text{pre} :: C_i \rightarrow A$ for $A \in \text{TYPES}(C)$,
5. for each pair $C_i < C_j$ ($C_i, C_j < C$), there is a cast operation of type $C_j \rightarrow C_i$ that can change the static type of an object of type C_i : $obj :: C_i.\text{oclAsType}(C_j)$,
6. for each class $C_i \in C$, there are two dynamic type tests ($X.\text{oclIsTypeOf}(C_i)$ and $X.\text{oclIsKindOf}(C_i)$),
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 ?? of all OCL operations discussed so far.

⁸Given 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:

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

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}, \dots, 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:

$$\mathfrak{A} := C_1 + \cdots + C_n. \quad (\text{A.17})$$

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:

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

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

To overcome this limitation, we introduce an auxiliary type C_{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 $\{C_{j_1}, \dots, C_{j_m}\}$.

- Then the *class type extension* $C_{i\text{ext}}$ associated to C_i is $A_{i_1} \times \dots \times A_{i_n} \times (C_{j_1\text{ext}} + \dots + C_{j_m\text{ext}})_\perp$ 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 $\text{oid} \times A_{i_1} \times \dots \times A_{i_n} \times (C_{j_1\text{ext}} + \dots + C_{j_m\text{ext}})_\perp$ 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

$$\begin{aligned} \text{eval_extract } X \ f = (\lambda \tau. \text{ case } X \ \tau \text{ of } & \perp \quad \Rightarrow \text{invalid } \tau \quad \text{exception} \\ & | \ \underline{\perp} \quad \Rightarrow \text{invalid } \tau \quad \text{deref. null} \\ & | \ \underline{\text{obj}} \quad \Rightarrow f(\text{oid_of } \text{obj}) \ \tau) \end{aligned} \quad (\text{A.20})$$

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

definition $\text{deref_oid}_C \text{ fst_snd } f \text{ oid} = (\lambda \tau. \text{ case } (\text{heap } (\text{fst_snd } \tau)) \text{ oid of}$

$$\begin{aligned} & \underline{\text{in}_C \text{ obj}} \quad \Rightarrow f \text{ obj } \tau \\ & | \ _ \quad \Rightarrow \text{invalid } \tau) \end{aligned} \quad (\text{A.21})$$

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:

$$\begin{aligned} \text{definition } \text{select}_a \ f = (\lambda \quad & \text{mk}_C \text{ oid} \quad \dots \perp \dots \quad C_{\text{Xext}} \Rightarrow \text{null} \\ & | \ \text{mk}_C \text{ oid} \quad \dots \underline{a} \dots \quad C_{\text{Xext}} \Rightarrow f(\lambda x \ \underline{x} \ \underline{x}) \ a) \end{aligned} \quad (\text{A.22})$$

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:

$$\begin{aligned} \text{definition} \quad & \text{in_pre_state} = \text{fst} \quad \text{first component} \\ \text{definition} \quad & \text{in_post_state} = \text{snd} \quad \text{second component} \\ \text{definition} \quad & \text{reconst_basetype} = \text{id} \quad \text{identity function} \end{aligned} \quad (\text{A.23})$$

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

$$\begin{aligned} \text{definition} \quad & _.\text{getBase} \quad :: C \Rightarrow A_{\text{base}} \\ \text{where} \quad & X.\text{getBase} = \text{eval_extract } X \ (\text{deref_oid}_C \text{ in_post_state} \\ & \quad (\text{select}_{\text{getBase}} \text{ reconst_basetype})) \end{aligned} \quad (\text{A.24})$$

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

$$\begin{aligned} \text{definition} \quad & _.\text{getObject} \quad :: C \Rightarrow A_{\text{object}} \\ \text{where} \quad & X.\text{getObject} = \text{eval_extract } X \ (\text{deref_oid}_C \text{ in_post_state} \\ & \quad (\text{select}_{\text{getObject}} (\text{deref_oid}_C \text{ in_post_state}))) \end{aligned} \quad (\text{A.25})$$

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

⁹We 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 with 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> \<epsilon> (X :: C_i) \<Longrightarrow> \<tau> \<Turnstile> (X .oclIsTypeOf(C_j))
\<tau> \<Turnstile> (X::OclAny) .oclAsType(OclAny) \<triangleq> X
\<tau> \<Turnstile> \<epsilon> (X :: C_i) \<Longrightarrow> \<tau> \<Turnstile> (X .oclIsTypeOf(C_j))
\<tau> \<Turnstile> \<delta> X \<Longrightarrow> \<tau> \<Turnstile> X .oclAsType(C_j) .oclAsType(C_i) \<triangleq> X
\<tau> \<Turnstile> \<epsilon> X \<Longrightarrow> \<tau> \<Turnstile> X .oclIsTypeOf(C_i) \<triangleq> X
\<tau> \<Turnstile> X .oclIsTypeOf(C_j) \<Longrightarrow> \<tau> \<Turnstile> \<delta> X \<Longrightarrow> \<tau> \<Turnstile> X .oclIsTypeOf(C_i) \<triangleq> 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_i) .oclIsTypeOf(C_j)
```

```

(\<tau> \<Turnstile> (X::C_j) \<doteq> X) = (\<tau> \<Turnstile> if \<epsilon> 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:

$$\begin{aligned}
& \text{invalid.oclIsTypeOf}(C_i) = \text{invalid} & \text{null.oclIsTypeOf}(C_i) = \text{true} \\
& \text{invalid.oclIsKindOf}(C_i) = \text{invalid} & \text{null.oclIsKindOf}(C_i) = \text{true} \\
& (X :: C_i).oclAsType(C_i) = X & \text{invalid.oclAsType}(C_i) = \text{invalid} & (X :: C_i) \doteq X = \text{if } \forall X \text{ t} \\
& \text{null.oclAsType}(C_i) = \text{null} & ((X :: C_i).oclAsType(C_j) \text{ .oclAsType}(C_i) = X)
\end{aligned} \tag{A.26}$$

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

$$\begin{aligned}
& \text{invalid}.a = \text{invalid} & \text{null}.a = \text{invalid} \\
& \text{invalid}.a@pre = \text{invalid} & \text{null}.a@pre = \text{invalid} \\
& \text{invalid}.r = \text{invalid} & \text{null}.r = \text{invalid} \\
& \text{invalid}.r@pre = \text{invalid} & \text{null}.r@pre = \text{invalid}
\end{aligned}$$

Other Operations on States

Defining $_ .allInstances()$ is straight-forward; the only difference is the property $T.allInstances() \rightarrow \text{exclude}$ 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:

$$I[X \rightarrow \text{oclIsModifiedOnly}()](\sigma, \sigma') \equiv \begin{cases} \perp & \text{if } X' = \perp \vee \text{null} \in X' \\ \lfloor \forall i \in M. \sigma i = \sigma' i \rfloor & \text{otherwise.} \end{cases}$$

where $X' = I[X](\sigma, \sigma')$ and $M = (\text{dom } \sigma \cap \text{dom } \sigma') - \{\text{OidOf } x \mid x \in X'\}$. Thus, if we require in a postcondition $\text{Set}\{\} \rightarrow \text{oclIsModifiedOnly}()$ 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 $\tau \models X \rightarrow \text{excluding}(s.a) \rightarrow \text{oclIsModifiedOnly}()$ and $\tau \models X \rightarrow \text{forAll}(x \text{ not } (x \doteq s.a))$, we can infer that $\tau \models s.a \triangleq s.a @pre$.

A.3.4. Data Invariants

Since the present OCL semantics uses one interpretation function¹⁰, we express the effect of OCL terms occurring in preconditions and invariants by a syntactic transformation $_pre$ which replaces:

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

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

$$\begin{aligned} I[\![\text{context } c : C_i \text{ inv } n : \phi(c)]\!] \tau \equiv \\ \tau \models (C_i.allInstances() \rightarrow \text{forall}(x | \phi(x))) \wedge \\ \tau \models (C_i.allInstances() \rightarrow \text{forall}(x | \phi(x)))_{pre} \end{aligned} \quad (\text{A.27})$$

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, \dots, 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:

$$\begin{aligned} I[\![\text{context } C :: op(a_1, \dots, a_n) : T \\ \text{pre } \phi(self, a_1, \dots, a_n) \\ \text{post } \psi(self, a_1, \dots, a_n, result)]\!] \equiv \\ \lambda s, x_1, \dots, x_n, \tau. \\ \text{if } \tau \models s \wedge \tau \models x_1 \wedge \dots \wedge \tau \models x_n \\ \text{then SOME } result. \quad \tau \models \phi(s, x_1, \dots, x_n)_{pre} \\ \quad \wedge \tau \models \psi(s, x_1, \dots, x_n, result)) \\ \text{else } \perp \end{aligned} \quad (\text{A.28})$$

FixMe:
Should we
add in our
notion of
Class-
Model
also the
Opera-
tions
?

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

where $\text{SOME } x. P(x)$ is the Hilbert-Choice Operator that chooses an arbitrary element satisfying P ; if such an element does not exist, it chooses an arbitrary one¹¹. Thus, using the Hilbert-Choice Operator, a contract can be associated to a function definition:

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

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 constructs 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):

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

under the conditions:

- E must be an OCL term and
- self must be defined, and the arguments valid in τ :
 $\tau \models \partial \text{ self} \wedge \tau \models v_{x_1} \wedge \dots \wedge \tau \models v_{x_n}$
- the post-condition must be satisfiable (“the operation must be implementable”): $\exists res. \tau \models \psi \text{ self } a_1 \dots a_n \text{ res}$.

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

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

where

- E must be an OCL term.
- self must be defined, and the arguments valid in τ :
 $\tau \models \partial \text{ self} \wedge \tau \models v_{x_1} \wedge \dots \wedge \tau \models v_{x_n}$

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

- the postcondition $\psi \text{ self } a_1 \dots a_n \text{ result}$ must be decomposable into a $\psi' \text{ self } a_1 \dots a_n$ and $\text{result} \triangleq \text{BODY self } a_1 \dots a_n$.

We do not model *overriding* of operations as in Java or C++ explicitly in FeatherweightOCL. However, it is easily 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 overridden 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 ( $\lceil \cdot \rceil$ )
no-notation floor ( $\lfloor \cdot \rfloor$ )
```

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

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

```
fun drop :: 'a option  $\Rightarrow$  'a ( $\lceil (-) \rceil$ )
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 ( $I \llbracket \cdot \rrbracket$ )
where  $I \llbracket x \rrbracket \equiv x$ 
```

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 $\lfloor \perp \rfloor$ on $'a\ option\ option$) to a *null* element, which may have an arbitrary semantic structure, and an undefinedness element \perp to an abstract undefinedness element *bot* (also written \perp whenever no confusion arises). As a consequence, it is necessary to redefine the notions of invalid, defined, valuation etc. on top of this interface.

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

```
class bot =  
  fixes bot :: 'a  
  assumes nonEmpty :  $\exists x. x \neq bot$ 
```

```
class null = bot +  
  fixes null :: 'a  
  assumes null-is-valid :  $null \neq bot$ 
```

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:  $(bot::'a\ option) \equiv (None::'a\ option)$   
  instance proof show  $\exists x::'a\ option. x \neq bot$   
    by(rule-tac x=Some x in exI, simp add:bot-option-def)  
  qed  
end
```

```
instantiation option :: (bot)null  
begin  
  definition null-option-def:  $(null::'a::bot\ option) \equiv \lfloor bot \rfloor$   
  instance proof show  $(null::'a::bot\ option) \neq bot$   
    by(simp add : null-option-def bot-option-def)
```

```

      qed
end

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

  instance proof show  $\exists (x::'a \Rightarrow 'b). x \neq bot$ 
    apply(rule-tac x= $\lambda -. (SOME y. y \neq bot)$  in exI, auto)
    apply(drule-tac x=x in fun-cong, auto simp:bot-fun-def)
    apply(erule contrapos-pp, simp)
    apply(rule some-eq-ex[THEN iffD2])
    apply(simp add: nonEmpty)
  done
  qed
end

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

  instance proof
    show (null::'a  $\Rightarrow$  'b::null)  $\neq bot$ 
    apply(auto simp: null-fun-def bot-fun-def)
    apply(drule-tac x=x in fun-cong)
    apply(erule contrapos-pp, simp add: null-is-valid)
  done
  qed
end

```

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

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 hierarchy.

For the moment, we formalize the most common notions of objects, in particular the existence 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 $'\mathcal{A}$.

record ($'\mathcal{A}$)*state* =
 heap :: $oid \rightarrow '\mathcal{A}$
 assocs :: $oid \rightarrow ((oid\ list)\ list)\ list$

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 ($'\mathcal{A}$)*st* = $'\mathcal{A}\ state \times '\mathcal{A}\ state$

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:

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

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

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

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

instantiation *option* :: (*object*)*object*
begin
 definition *oid-of-option-def*: $oid-of\ x = oid-of\ (the\ x)$
 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 $'\mathcal{A}$) to an arbitrary null-type (i. e., containing at least a distinguished *null* and *invalid* element).

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

FixMe:
 Get Appropriate
 Reference!

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* :: ($\mathcal{A}, \alpha :: bot$) val

where *invalid* $\equiv \lambda \tau. bot$

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

lemma *textbook-invalid*: $I\llbracket invalid \rrbracket \tau = bot$

by(*simp add: invalid-def Sem-def*)

Note that the definition :

definition *null* :: ($\mathcal{A}, \alpha :: null$) val

where "*null*" $\equiv \lambda \tau. null$

is not necessary since we defined the entire function space over null types again as null-types; the crucial definition is $null \equiv \lambda x. null$. 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*: $I\llbracket null :: (\mathcal{A}, \alpha :: null) val \rrbracket \tau = (null :: (\alpha :: null))$

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 *Boolean_{base}* = *bool option option*

type-synonym (\mathcal{A})*Boolean* = ($\mathcal{A}, Boolean_{base}$) val

Because of the previous class definitions, Isabelle type-inference establishes that $\mathcal{A} Boolean$ 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 *Integer_{base}* = *int option option*

type-synonym (\mathcal{A})*Integer* = ($\mathcal{A}, Integer_{base}$) val

type-synonym *String_{base}* = *string option option*

type-synonym (\mathcal{A})*String* = ($\mathcal{A}, String_{base}$) val

type-synonym $Real_{base} = real\ option\ option$

type-synonym $(\mathcal{A})Real = (\mathcal{A}, Real_{base})\ val$

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 $Void_{base} = \{X::unit\ option\ option.\ X = bot \vee X = null\}$ **by**(*rule-tac* $x=bot$ **in** *exI*, *simp*)

type-synonym $(\mathcal{A})Void = (\mathcal{A}, Void_{base})\ val$

B.1.3. Some OCL Collection Types

The construction of collection types is slightly 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 $(\alpha, \beta)\ Pair_{base}$. It is shown that this type “fits” indeed into the abstract type interface discussed in the previous section.

typedef $(\alpha, \beta)\ Pair_{base} = \{X::(\alpha::null \times \beta::null)\ option\ option.\$
 $X = bot \vee X = null \vee (fst[[X]] \neq bot \wedge snd[[X]] \neq bot)\}$
by (*rule-tac* $x=bot$ **in** *exI*, *simp*)

We “carve” out from the concrete type $(\alpha \times \beta)\ option\ option$ the new fully abstract type, which will not contain representations like $[(\perp, a)]$ or $[(b, \perp)]$. The type constructor $Pair\{x,y\}$ to be defined later will identify these with *invalid*.

instantiation $Pair_{base} :: (null, null)bot$

begin

definition $bot-Pair_{base}-def: (bot-class.bot :: (\alpha::null, \beta::null)\ Pair_{base}) \equiv Abs-Pair_{base}\ None$

instance proof show $\exists x::(\alpha, \beta)\ Pair_{base}.\ x \neq bot$

apply(*rule-tac* $x=Abs-Pair_{base}\ [None]$ **in** *exI*)

by(*simp add: bot-Pair_{base}-def Abs-Pair_{base}-inject null-option-def bot-option-def*)

qed

end

instantiation $Pair_{base} :: (null, null)null$

begin

definition $\text{null-Pair}_{base-def}$: $(\text{null}::('a::\text{null}, 'b::\text{null}) \text{Pair}_{base}) \equiv \text{Abs-Pair}_{base} \text{ [None]}$

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

qed

end

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

type-synonym $(\mathcal{A}, 'a, 'b) \text{Pair} = (\mathcal{A}, ('a, 'b) \text{Pair}_{base}) \text{ val}$

The Construction of the Set Type

The core of an own type construction is done via a type definition which provides the raw-type $'a \text{Set}_{base}$. 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 $'a \text{Set}_{base} = \{X::('a::\text{null}) \text{set option option. } X = \text{bot} \vee X = \text{null} \vee (\forall x \in [X]. x \neq \text{bot})\}$
by ($\text{rule-tac } x=\text{bot} \text{ in exI, simp}$)

instantiation $\text{Set}_{base} :: (\text{null})\text{bot}$

begin

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

instance proof show $\exists x::'a \text{Set}_{base}. x \neq \text{bot}$
apply ($\text{rule-tac } x=\text{Abs-Set}_{base} \text{ [None] in exI}$)
by ($\text{simp add: bot-Set}_{base-def} \text{Abs-Set}_{base-inject} \text{null-option-def bot-option-def}$)

qed

end

instantiation $\text{Set}_{base} :: (\text{null})\text{null}$

begin

definition $\text{null-Set}_{base-def}$: $(\text{null}::('a::\text{null}) \text{Set}_{base}) \equiv \text{Abs-Set}_{base} \text{ [None]}$

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

qed

end

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

type-synonym $(\mathcal{A}, 'a) \text{Set} = (\mathcal{A}, 'a \text{Set}_{base}) \text{ val}$

The Construction of the Sequence Type

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

```

typedef 'α Sequencebase = {X::('α::null) list option option.
    X = bot ∨ X = null ∨ (∀x∈set [X]. x ≠ bot)}
    by (rule-tac x=bot in exI, simp)

instantiation Sequencebase :: (null)bot
begin

    definition bot-Sequencebase-def: (bot::('α::null) Sequencebase) ≡ Abs-Sequencebase None

    instance proof show ∃x::'α Sequencebase. x ≠ bot
        apply (rule-tac x=Abs-Sequencebase [None] in exI)
        by (auto simp:bot-Sequencebase-def Abs-Sequencebase-inject
            null-option-def bot-option-def)
    qed
end

instantiation Sequencebase :: (null)null
begin

    definition null-Sequencebase-def: (null::('α::null) Sequencebase) ≡ Abs-Sequencebase [ None ]

    instance proof show (null::('α::null) Sequencebase) ≠ bot
        by (auto simp:bot-Sequencebase-def null-Sequencebase-def Abs-Sequencebase-inject
            null-option-def bot-option-def)
    qed
end

    ... and lifting this type to the format of a valuation gives us:
type-synonym ('A,'α) Sequence = ('A, 'α Sequencebase) 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 resenation 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. cartesian products of arguments were curried,
2. constants of type T were mapped to valuations over the HOL-type for T ,
3. functions $T \rightarrow T'$ were mapped to functions in HOL, where T and T' were mapped to the valuations for them, and

OCL Type	HOL Type
Boolean	$'\mathcal{A} \text{ Boolean}$
Boolean \rightarrow Boolean	$'\mathcal{A} \text{ Boolean} \Rightarrow '\mathcal{A} \text{ Boolean}$
(Integer, Integer) \rightarrow Boolean	$'\mathcal{A} \text{ Integer} \Rightarrow '\mathcal{A} \text{ Integer} \Rightarrow '\mathcal{A} \text{ Boolean}$
Set (Integer)	$('\mathcal{A}, \text{Integer}_{base}) \text{ Set}$
Set (Integer) \rightarrow Real	$('\mathcal{A}, \text{Integer}_{base}) \text{ Set} \Rightarrow '\mathcal{A} \text{ Real}$
Set (Pair (Integer, Boolean))	$('\mathcal{A}, (\text{Integer}_{base}, \text{Boolean}_{base}) \text{ Pair}_{base}) \text{ Set}$
Set (<T>)	$('\mathcal{A}, '\alpha) \text{ Set}$

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

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

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

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

```

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

```

```

lemma [simp]: false (a, b) =  $\llbracket \text{False} \rrbracket$ 
by(simp add: false-def)

```

```

lemma [simp]: true (a, b) =  $\llbracket \text{True} \rrbracket$ 
by(simp add: true-def)

```

```

lemma textbook-true:  $I\llbracket \text{true} \rrbracket \tau = \llbracket \text{True} \rrbracket$ 
by(simp add: Sem-def true-def)

```

```

lemma textbook-false:  $I\llbracket \text{false} \rrbracket \tau = \llbracket \text{False} \rrbracket$ 
by(simp add: Sem-def false-def)

```

Name	Theorem
<i>textbook-invalid</i>	$I\llbracket \text{invalid} \rrbracket \tau = \text{UML-Types.bot-class.bot}$
<i>textbook-null-fun</i>	$I\llbracket \text{null} \rrbracket \tau = \text{null}$
<i>textbook-true</i>	$I\llbracket \text{true} \rrbracket \tau = \llbracket \text{True} \rrbracket$
<i>textbook-false</i>	$I\llbracket \text{false} \rrbracket \tau = \llbracket \text{False} \rrbracket$

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

```

```

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

```

```

lemma valid2[simp]:  $\text{v null} = \text{true}$ 

```

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]: $\forall true = true$

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]: $\forall false = true$

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: $(\forall X) \tau = (\forall (\lambda -. X \tau)) \tau$

by(simp add: valid-def)

definition defined :: ($\mathcal{A}, 'a::null$)val \Rightarrow (\mathcal{A})Boolean ($\delta - [100]100$)

where $\delta X \equiv \lambda \tau . \text{if } X \tau = \text{bot } \tau \vee X \tau = \text{null } \tau \text{ then false } \tau \text{ else true } \tau$

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

lemma defined1[simp]: $\delta \text{ invalid} = \text{false}$

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

lemma defined2[simp]: $\delta \text{ null} = \text{false}$

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

lemma defined3[simp]: $\delta \text{ true} = \text{true}$

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

lemma defined4[simp]: $\delta \text{ false} = \text{true}$

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

lemma defined5[simp]: $\delta \delta X = \text{true}$

by(rule ext,
auto simp: defined-def true-def false-def
bot-fun-def bot-option-def null-option-def null-fun-def)

lemma defined6[simp]: $\delta \forall X = \text{true}$

by(rule ext,
auto simp: valid-def defined-def true-def false-def
bot-fun-def bot-option-def null-option-def null-fun-def)

lemma *valid5*[simp]: $\forall v X = \text{true}$
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]: $\forall \delta X = \text{true}$
by(rule ext,
 auto simp: *valid-def* *defined-def* *true-def* *false-def*
bot-fun-def *bot-option-def* *null-option-def* *null-fun-def*)

lemma *cp-defined*: $(\delta X) \tau = (\delta (\lambda \cdot X \tau)) \tau$
by(simp add: *defined-def*)

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

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

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

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

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

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
<i>defined1</i>	$\delta \text{ invalid} = \text{false}$
<i>defined2</i>	$\delta \text{ null} = \text{false}$
<i>defined3</i>	$\delta \text{ true} = \text{true}$
<i>defined4</i>	$\delta \text{ false} = \text{true}$
<i>defined5</i>	$\delta \delta X = \text{true}$
<i>defined6</i>	$\delta \vee X = \text{true}$

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 $a.\text{boss} \triangleq b.\text{boss@pre}$ instead of

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

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” $_ = _ :: \alpha * \alpha \rightarrow \text{bool}$ —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:

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

“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 \perp element. Strong equality is simply polymorphic in Featherweight OCL, i. e., is defined identical for all types in OCL and HOL.

definition *StrongEq*:: $[\lambda st \Rightarrow 'a, \lambda st \Rightarrow 'a] \Rightarrow ('a)Boolean$ (**infixl** $\triangleq 30$)
where $X \triangleq Y \equiv \lambda \tau. \llbracket X \tau = Y \tau \rrbracket$

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 refers to, is passed constantly and unmodified to the sub-expressions, i. e., all sub-expressions inside an OCL expression refer to the same context. Expressed formally, this boils down to:

lemma *StrongEq-subst* :
assumes *cp*: $\bigwedge X. P(X)\tau = P(\lambda -. X \tau)\tau$
and *eq*: $(X \triangleq Y)\tau = \text{true } \tau$
shows $(P X \triangleq P Y)\tau = \text{true } \tau$
apply(*insert cp eq*)
apply(*simp add: null-def invalid-def true-def false-def StrongEq-def*)
apply(*subst cp[of X]*)
apply(*subst cp[of Y]*)
by *simp*

lemma *defined7[simp]*: $\delta (X \triangleq Y) = \text{true}$
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]*: $\nu (X \triangleq Y) = \text{true}$
by(*rule ext,*
auto simp: valid-def true-def false-def StrongEq-def
bot-fun-def bot-option-def null-option-def null-fun-def)

lemma *cp-StrongEq*: $(X \triangleq Y) \tau = ((\lambda -. X \tau) \triangleq (\lambda -. Y \tau)) \tau$
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 than having the truth-table aside when reading fails its purpose in our view.

Practically, this means that we want to give a definition to the core operations to be as close as possible to the lattice laws; this makes also powerful symbolic 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* :: $(^{\mathcal{A}})\text{Boolean} \Rightarrow (^{\mathcal{A}})\text{Boolean} \text{ (not)}$

where $\text{not } X \equiv \lambda \tau. \text{case } X \tau \text{ of}$
 $\quad \perp \Rightarrow \perp$
 $\quad | [\perp] \Rightarrow [\perp]$
 $\quad | [[x]] \Rightarrow [[\neg x]]$

lemma *cp-OclNot*: $(\text{not } X)\tau = (\text{not } (\lambda -. X \tau)) \tau$
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*: $\bigwedge x y. \text{not } x = \text{not } y \implies x = y$
by(subst *OclNot-not*[*THEN sym*], simp)

definition *OclAnd* :: $((\lambda) \text{Boolean}, (\lambda) \text{Boolean}) \Rightarrow (\lambda) \text{Boolean}$ (**infixl** and 30)

where $X \text{ and } Y \equiv (\lambda \tau. \text{case } X \tau \text{ of}$
 $\quad \lfloor \lfloor \text{False} \rfloor \rfloor \Rightarrow \lfloor \lfloor \text{False} \rfloor \rfloor$
 $\quad \mid \perp \Rightarrow (\text{case } Y \tau \text{ of}$
 $\quad \quad \lfloor \lfloor \text{False} \rfloor \rfloor \Rightarrow \lfloor \lfloor \text{False} \rfloor \rfloor$
 $\quad \quad \mid - \Rightarrow \perp)$
 $\quad \mid \lfloor \perp \rfloor \Rightarrow (\text{case } Y \tau \text{ of}$
 $\quad \quad \lfloor \lfloor \text{False} \rfloor \rfloor \Rightarrow \lfloor \lfloor \text{False} \rfloor \rfloor$
 $\quad \quad \mid \perp \Rightarrow \perp$
 $\quad \quad \mid - \Rightarrow \lfloor \perp \rfloor)$
 $\quad \mid \lfloor \lfloor \text{True} \rfloor \rfloor \Rightarrow Y \tau)$

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

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

lemma *textbook-OclNot*:
 $I[\text{not}(X)] \tau = (\text{case } I[X] \tau \text{ of } \perp \Rightarrow \perp$
 $\quad \mid \lfloor \perp \rfloor \Rightarrow \lfloor \perp \rfloor$
 $\quad \mid \lfloor \lfloor x \rfloor \rfloor \Rightarrow \lfloor \lfloor \neg x \rfloor \rfloor)$
by(simp add: *Sem-def OclNot-def*)

lemma *textbook-OclAnd*:
 $I[X \text{ and } Y] \tau = (\text{case } I[X] \tau \text{ of}$
 $\quad \perp \Rightarrow (\text{case } I[Y] \tau \text{ of}$

$$\begin{array}{l}
\perp \Rightarrow \perp \\
| \lfloor \perp \rfloor \Rightarrow \perp \\
| \lfloor \text{True} \rfloor \Rightarrow \perp \\
| \lfloor \text{False} \rfloor \Rightarrow \lfloor \text{False} \rfloor \\
| \lfloor \perp \rfloor \Rightarrow (\text{case } I[Y] \tau \text{ of} \\
\quad \perp \Rightarrow \perp \\
\quad | \lfloor \perp \rfloor \Rightarrow \lfloor \perp \rfloor \\
\quad | \lfloor \text{True} \rfloor \Rightarrow \lfloor \perp \rfloor \\
\quad | \lfloor \text{False} \rfloor \Rightarrow \lfloor \text{False} \rfloor) \\
| \lfloor \text{True} \rfloor \Rightarrow (\text{case } I[Y] \tau \text{ of} \\
\quad \perp \Rightarrow \perp \\
\quad | \lfloor \perp \rfloor \Rightarrow \lfloor \perp \rfloor \\
\quad | \lfloor y \rfloor \Rightarrow \lfloor y \rfloor) \\
| \lfloor \text{False} \rfloor \Rightarrow \lfloor \text{False} \rfloor)
\end{array}$$

by(simp add: OclAnd-def Sem-def split: option.split bool.split)

definition *OclOr* :: $((\mathcal{A})\text{Boolean}, (\mathcal{A})\text{Boolean}) \Rightarrow (\mathcal{A})\text{Boolean}$ (**infixl** or 25)
where $X \text{ or } Y \equiv \text{not}(\text{not } X \text{ and } \text{not } Y)$

definition *OclImplies* :: $((\mathcal{A})\text{Boolean}, (\mathcal{A})\text{Boolean}) \Rightarrow (\mathcal{A})\text{Boolean}$ (**infixl** implies 25)
where $X \text{ implies } Y \equiv \text{not } X \text{ or } Y$

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

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

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

lemma *OclAnd1*[simp]: $(\text{invalid and true}) = \text{invalid}$
by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def)
lemma *OclAnd2*[simp]: $(\text{invalid and false}) = \text{false}$
by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def)
lemma *OclAnd3*[simp]: $(\text{invalid and null}) = \text{invalid}$
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]: $(\text{invalid and invalid}) = \text{invalid}$
by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def)

lemma *OclAnd5[simp]*: $(\text{null and true}) = \text{null}$
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]*: $(\text{null and false}) = \text{false}$
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]*: $(\text{null and null}) = \text{null}$
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]*: $(\text{null and invalid}) = \text{invalid}$
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]*: $(\text{false and true}) = \text{false}$
by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def)
lemma *OclAnd10[simp]*: $(\text{false and false}) = \text{false}$
by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def)
lemma *OclAnd11[simp]*: $(\text{false and null}) = \text{false}$
by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def)
lemma *OclAnd12[simp]*: $(\text{false and invalid}) = \text{false}$
by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def)

lemma *OclAnd13[simp]*: $(\text{true and true}) = \text{true}$
by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def)
lemma *OclAnd14[simp]*: $(\text{true and false}) = \text{false}$
by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def)
lemma *OclAnd15[simp]*: $(\text{true and null}) = \text{null}$
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]*: $(\text{true and invalid}) = \text{invalid}$
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]*: $(X \text{ and } X) = X$
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*: $(X \text{ and } Y) = (Y \text{ and } X)$
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-falseI[simp]*: $(\text{false and } X) = \text{false}$

```

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]:  $(X \text{ and } \text{false}) = \text{false}$ 
by(simp add: OclAnd-commute)

lemma OclAnd-true1[simp]:  $(\text{true} \text{ and } X) = X$ 
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]:  $(X \text{ and } \text{true}) = X$ 
by(simp add: OclAnd-commute)

lemma OclAnd-bot1[simp]:  $\bigwedge \tau. X \tau \neq \text{false} \tau \implies (\text{bot} \text{ and } X) \tau = \text{bot} \tau$ 
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]:  $\bigwedge \tau. X \tau \neq \text{false} \tau \implies (X \text{ and } \text{bot}) \tau = \text{bot} \tau$ 
by(simp add: OclAnd-commute)

lemma OclAnd-null1[simp]:  $\bigwedge \tau. X \tau \neq \text{false} \tau \implies X \tau \neq \text{bot} \tau \implies (\text{null} \text{ and } X) \tau = \text{null} \tau$ 
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]:  $\bigwedge \tau. X \tau \neq \text{false} \tau \implies X \tau \neq \text{bot} \tau \implies (X \text{ and } \text{null}) \tau = \text{null} \tau$ 
by(simp add: OclAnd-commute)

lemma OclAnd-assoc:  $(X \text{ and } (Y \text{ and } Z)) = (X \text{ and } Y \text{ and } Z)$ 
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]:  $(\text{invalid} \text{ or } \text{true}) = \text{true}$ 
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]:  $(\text{invalid} \text{ or } \text{false}) = \text{invalid}$ 

```

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]: $\bigwedge \tau. X \tau \neq \text{true} \tau \implies (\text{bot or } X) \tau = \text{bot } \tau$
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]: $\bigwedge \tau. X \tau \neq \text{true} \tau \implies (X \text{ or bot}) \tau = \text{bot } \tau$
by(simp add: OclOr-commute)

lemma *OclOr-null1*[simp]: $\bigwedge \tau. X \tau \neq \text{true} \tau \implies X \tau \neq \text{bot} \tau \implies (\text{null or } X) \tau = \text{null} \tau$
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]: $\bigwedge \tau. X \tau \neq \text{true} \tau \implies X \tau \neq \text{bot} \tau \implies (X \text{ or null}) \tau = \text{null} \tau$
by(simp add: *OclOr-commute*)

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

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

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

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

A Standard Logical Calculus for OCL

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

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

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

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

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

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

```

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

```

Local Validity and Meta-logic **lemma** *foundation1*[simp]: $\tau \models \text{true}$
by(auto simp: OclValid-def)

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

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

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

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

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

lemma *valid-bool-split*: $(\tau \models \vee \ A) = ((\tau \models A \triangleq \text{null}) \vee (\tau \models A) \vee (\tau \models \text{not } A))$
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*: $(\tau \models \delta \ A) = ((\tau \models A) \vee (\tau \models \text{not } A))$
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*:
 $\tau \models (P \text{ and } Q) \implies (\tau \models P) \wedge (\tau \models Q)$
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*:

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

lemma foundation7'[simp]:
 $(\tau \models \text{not } (v x)) = (\neg (\tau \models v x))$
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:
 $(\tau \models \delta x) \vee (\tau \models (x \triangleq \text{invalid})) \vee (\tau \models (x \triangleq \text{null}))$
proof –
have 1 : $(\tau \models \delta x) \vee (\neg (\tau \models \delta x))$ **by** auto
have 2 : $(\neg (\tau \models \delta x)) = ((\tau \models (x \triangleq \text{invalid})) \vee (\tau \models (x \triangleq \text{null})))$
by(simp only: defined-split, simp)
show ?thesis **by**(insert 1, simp add:2)
qed

lemma foundation9:
 $\tau \models \delta x \implies (\tau \models \text{not } x) = (\neg (\tau \models x))$
apply(simp add: 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':
 $\tau \models \text{not } x \implies \neg (\tau \models x)$
by(auto simp: foundation6 foundation9)

lemma foundation9'':
 $\tau \models \text{not } x \implies \tau \models \delta x$
by(metis OclNot3 OclNot-not OclValid-def cp-OclNot cp-defined defined4)

lemma foundation10:
 $\tau \models \delta x \implies \tau \models \delta y \implies (\tau \models (x \text{ and } y)) = ((\tau \models x) \wedge (\tau \models y))$
apply(simp add: 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': $(\tau \models (A \text{ and } B)) = ((\tau \models A) \wedge (\tau \models B))$

by(*auto dest: foundation5 simp: foundation6 foundation10*)

lemma foundation11:

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

apply(*simp add: 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:

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

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

by(*auto simp: OclNot-def OclValid-def invalid-def true-def null-def StrongEq-def
split:bool.split-asm bool.split*)

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

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

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

by(*auto simp: OclValid-def defined-def false-def true-def bot-fun-def null-fun-def
split:split-if-asm*)

lemma foundation16'': $\neg(\tau \models (\delta X)) = ((\tau \models (X \triangleq \text{invalid})) \vee (\tau \models (X \triangleq \text{null})))$

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

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

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

lemma *foundation18''*: $(\tau \models (\vee X)) = (\neg(\tau \models (X \triangleq \text{invalid})))$
by(*auto simp: foundation15*)

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

lemma *foundation21*: $(\text{not } A \triangleq \text{not } B) = (A \triangleq B)$
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*: $(\tau \models P) = (\tau \models (\lambda _ . P \tau))$
by(*auto simp: OclValid-def true-def*)

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

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

lemma *foundation25'*: $\tau \models Q \implies \tau \models (P \text{ or } Q)$
by(*subst OclOr-commute, simp add: foundation25*)

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

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

lemma foundation27: $(\tau \models (A \text{ and } B)) = ((\tau \models A) \wedge (\tau \models B))$

by(auto dest: foundation5 simp: foundation6 foundation10)

lemma defined-not-I : $\tau \models \delta(x) \implies \tau \models \delta(\text{not } x)$

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 : $\tau \models v(x) \implies \tau \models v(\text{not } x)$

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

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 : $\tau \models v(x) \implies \tau \models v(y) \implies \tau \models v(x \text{ and } y)$

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

by(simp add: OclOr-def defined-and-I defined-not-I)

lemma valid-or-I : $\tau \models v(x) \implies \tau \models v(y) \implies \tau \models v(x \text{ or } y)$

by(simp add: OclOr-def valid-and-I valid-not-I)

Local Judgements and Strong Equality **lemma** StrongEq-L-reft: $\tau \models (x \triangle x)$

by(simp add: OclValid-def StrongEq-def)

lemma StrongEq-L-sym: $\tau \models (x \triangle y) \implies \tau \models (y \triangle x)$

by(simp add: StrongEq-sym)

lemma StrongEq-L-trans: $\tau \models (x \triangle y) \implies \tau \models (y \triangle z) \implies \tau \models (x \triangle z)$

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 $cp :: ((\lambda\alpha, \alpha) val \Rightarrow (\lambda\alpha, \beta) val) \Rightarrow bool$
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*:
 $(\forall X \tau. f X \tau = f(\lambda\cdot. X \tau) \tau) \Longrightarrow cp P \Longrightarrow cp(\lambda X. f (P X))$
apply(*auto simp: true-def cp-def*)
apply(*rule exI, (rule allI)+*)
by(*erule-tac x=P X in allE, auto*)

lemma *cpI2*:

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

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

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

lemma *cp-const* : $cp(\lambda-. c)$
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* :: $[(^{\mathfrak{A}}) \text{Boolean}, (^{\mathfrak{A}}, ' \alpha :: \text{null}) \text{val}, (^{\mathfrak{A}}, ' \alpha) \text{val}] \Rightarrow (^{\mathfrak{A}}, ' \alpha) \text{val}$
 $(\text{if } (-) \text{ then } (-) \text{ else } (-) \text{ endif } [10, 10, 10] 50)$

where (*if C then B₁ else B₂ endif*) = $(\lambda \tau. \text{if } (\delta C) \tau = \text{true } \tau$
 $\text{then } (\text{if } (C \tau) = \text{true } \tau$
 $\text{then } B_1 \tau$
 $\text{else } B_2 \tau)$
 $\text{else invalid } \tau)$

lemma *cp-OclIf*: $((\text{if } C \text{ then } B_1 \text{ else } B_2 \text{ endif}) \tau =$
 $(\text{if } (\lambda -. C \tau) \text{ then } (\lambda -. B_1 \tau) \text{ else } (\lambda -. B_2 \tau) \text{ endif}) \tau)$
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*]: $(\text{if invalid then } B_1 \text{ else } B_2 \text{ endif}) = \text{invalid}$
by(*rule ext*, *auto simp*: *OclIf-def*)

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

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

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

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

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

lemma *OclIf-false'* [*simp*]: $\tau \models \text{not } P \implies (\text{if } P \text{ then } B_1 \text{ else } B_2 \text{ endif}) \tau = B_2 \tau$
apply(*subst cp-OclIf*)
apply(*auto simp*: *foundationI4*[*symmetric*] *foundation22*)
by(*auto simp*: *cp-OclIf*[*symmetric*])

lemma *OclIf-idem1* [*simp*]: $(\text{if } \delta X \text{ then } A \text{ else } A \text{ endif}) = A$
by(*rule ext*, *auto simp*: *OclIf-def*)

lemma *OclIf-idem2* [*simp*]: $(\text{if } \vee X \text{ then } A \text{ else } A \text{ endif}) = A$
by(*rule ext*, *auto simp*: *OclIf-def*)

lemma *OclNot-if* [*simp*]:
 $\text{not}(\text{if } P \text{ then } C \text{ else } E \text{ endif}) = (\text{if } P \text{ then not } C \text{ else not } E \text{ endif})$
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* :: $[(\lambda a, 'a)val, (\lambda a, 'a)val] \Rightarrow (\lambda a)Boolean$ (**infixl** \doteq 30)

with term "not" we can express the notation:

syntax

notequal :: $(\lambda a)Boolean \Rightarrow (\lambda a)Boolean \Rightarrow (\lambda a)Boolean$ (**infix** \triangleq 40)

translations

$a \triangleq b == CONST OclNot(a \doteq b)$

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 $\tau \models P \implies \tau \models \delta P$ — the following facts:

lemma *OclNot-defargs*:

$\tau \models (not P) \implies \tau \models \delta P$

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*: $\tau \models A \implies \tau \models B$

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*: $const X \implies X \tau = X \tau'$

by(*auto simp: const-def*)

lemma *const-subst*:


```

assumes const-X: const X
  and const-Y: const Y
  and eq :  $X \tau = Y \tau$ 
  and cp-P: cp P
  and pp :  $P Y \tau = P Y \tau'$ 
shows  $P X \tau = P X \tau'$ 
proof –
  have A:  $\bigwedge Y. P Y \tau = P (\lambda -. Y \tau) \tau$ 
    apply(insert cp-P, unfold cp-def)
    apply(elim exE, erule-tac x=Y in allE', erule-tac x= $\tau$  in allE)
    apply(erule-tac x=( $\lambda -. Y \tau$ ) in allE, erule-tac x= $\tau$  in allE)
    by simp
  have B:  $\bigwedge Y. P Y \tau' = P (\lambda -. Y \tau') \tau'$ 
    apply(insert cp-P, unfold cp-def)
    apply(elim exE, erule-tac x=Y in allE', erule-tac x= $\tau'$  in allE)
    apply(erule-tac x=( $\lambda -. Y \tau'$ ) in allE, erule-tac x= $\tau'$  in allE)
    by simp
  have C:  $X \tau' = Y \tau'$ 
    apply(rule trans, subst const-chn[OF const-X], rule eq)
    by(rule const-chn[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-imp2 :
assumes  $\bigwedge \tau \tau'. P \tau = P \tau' \implies Q \tau = Q \tau'$ 
shows const P  $\implies$  const Q
by(simp add: const-def, insert assms, blast)

```

```

lemma const-imp3 :
assumes  $\bigwedge \tau \tau'. P \tau = P \tau' \implies Q \tau = Q \tau' \implies R \tau = R \tau'$ 
shows const P  $\implies$  const Q  $\implies$  const R
by(simp add: const-def, insert assms, blast)

```

```

lemma const-imp4 :
assumes  $\bigwedge \tau \tau'. P \tau = P \tau' \implies Q \tau = Q \tau' \implies R \tau = R \tau' \implies S \tau = S \tau'$ 
shows const P  $\implies$  const Q  $\implies$  const R  $\implies$  const S
by(simp add: const-def, insert assms, blast)

```

```

lemma const-lam : const ( $\lambda -. e$ )
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-imp2*[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 (\vee X)*
by(rule *const-imp2*[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-imp3*[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-imp2*[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 \triangleq 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-impl4[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 $(\tau \models \delta x) = (\tau' \models \delta x)$
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 $(\tau \models v x) = (\tau' \models v x)$
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 \implies const D \implies const F \implies const ($\lambda \tau. \text{if } C \ \tau \text{ then } D \ \tau \text{ else } F \ \tau$)*
by(*auto simp: const-def*)
lemma *const-HOL-and*: *const C \implies const D \implies const ($\lambda \tau. C \ \tau \wedge D \ \tau$)*

```

  by(auto simp: const-def)
lemma const-HOL-eq : const C  $\implies$  const D  $\implies$  const ( $\lambda \tau. C \tau = D \tau$ )
  apply(auto simp: const-def)
  apply(erule-tac x= $\tau$  in allE)
  apply(erule-tac x= $\tau$  in allE)
  apply(erule-tac x= $\tau'$  in allE)
  apply(erule-tac x= $\tau'$  in allE)
  apply simp
  apply(erule-tac x= $\tau$  in allE)
  apply(erule-tac x= $\tau$  in allE)
  apply(erule-tac x= $\tau'$  in allE)
  apply(erule-tac x= $\tau'$  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 ( $\perp$ )

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 :: ( $\mathcal{A}, 'a :: \text{null}$ )val  $\Rightarrow$  ( $\mathcal{A}, 'b :: \text{null}$ )val
  fixes g

```

```

assumes def-scheme:  $(f\ x) \equiv \lambda\ \tau. \text{if } (\delta\ x)\ \tau = \text{true}\ \tau \text{ then } g\ (x\ \tau) \text{ else invalid } \tau$ 

locale profile-mono2 = profile-mono-scheme +
  assumes  $\bigwedge x. x \neq \text{bot} \implies x \neq \text{null} \implies g\ x \neq \text{bot}$ 
begin
  lemma strict[simp,code-unfold]:  $f\ \text{invalid} = \text{invalid}$ 
  by(rule ext, simp add: def-scheme true-def false-def)

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

  lemma cp0 :  $f\ X\ \tau = f\ (\lambda\ -. X\ \tau)\ \tau$ 
  by(simp add: def-scheme cp-defined[symmetric])

  lemma cp[simp,code-unfold] :  $cp\ P \implies cp\ (\lambda X. f\ (P\ X))$ 
  by(rule cpII[of f], intro allI, rule cp0, simp-all)

  lemma const[simp,code-unfold] :
    assumes C1 :  $\text{const } X$ 
    shows  $\text{const}(f\ X)$ 
  proof –
    have const-g :  $\text{const } (\lambda\ \tau. g\ (X\ \tau))$  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:  $\bigwedge x. x \neq \text{bot} \implies x \neq \text{null} \implies g\ x \neq \text{bot} \wedge g\ x \neq \text{null}$ 

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]:  $\delta(f\ x) = (\delta\ x)$ 
  apply(rule ext, rename-tac  $\tau$ ,subst foundation22[symmetric])
  apply(case-tac  $\neg(\tau \models \delta\ x)$ , 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  $y = \delta\ x$ ])
  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:  $v(f\ x) = (\delta(f\ x))$ 
  apply(rule ext, rename-tac  $\tau$ ,subst foundation22[symmetric])

```

```

apply(case-tac  $\neg(\tau \models \delta x)$ , 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:: (' $\mathcal{A}$ , 'a::null)val  $\Rightarrow$  ' $\mathcal{A}$  Boolean
  assumes d-strict[simp,code-unfold]: d invalid = false
  assumes d-cp0: d X  $\tau$  = d ( $\lambda$  -. X  $\tau$ )  $\tau$ 
  assumes d-const[simp,code-unfold]: const X  $\Longrightarrow$  const (d X)

```

Property Profiles for Binary Operators

```

definition bin' f g dx dy X Y =
  (f X Y = ( $\lambda$   $\tau$ . if (dx X)  $\tau$  = true  $\tau \wedge$  (dy Y)  $\tau$  = true  $\tau$ 
    then g X Y  $\tau$ 
    else invalid  $\tau$  ))

```

```

definition bin f g = bin' f ( $\lambda X Y \tau$ . g (X  $\tau$ ) (Y  $\tau$ ))

```

```

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

```

```

locale profile-bin-scheme =
  fixes dx:: (' $\mathcal{A}$ , 'a::null)val  $\Rightarrow$  ' $\mathcal{A}$  Boolean
  fixes dy:: (' $\mathcal{A}$ , 'b::null)val  $\Rightarrow$  ' $\mathcal{A}$  Boolean
  fixes f:: (' $\mathcal{A}$ , 'a::null)val  $\Rightarrow$  (' $\mathcal{A}$ , 'b::null)val  $\Rightarrow$  (' $\mathcal{A}$ , 'c::null)val
  fixes g
  assumes dx': profile-single dx
  assumes dy': profile-single dy
  assumes dx-dy-homo[simp,code-unfold]: cp (f X)  $\Longrightarrow$ 
    cp ( $\lambda x$ . f x Y)  $\Longrightarrow$ 
    f X invalid = invalid  $\Longrightarrow$ 
    f invalid Y = invalid  $\Longrightarrow$ 
    ( $\neg(\tau \models d_x X) \vee \neg(\tau \models d_y Y)$ )  $\Longrightarrow$ 
     $\tau \models (\delta f X Y \triangleq (d_x X \text{ and } d_y Y))$ 
  assumes def-scheme'[simplified]: bin f g dx dy X Y
  assumes I:  $\tau \models d_x X \Longrightarrow \tau \models d_y Y \Longrightarrow \tau \models \delta f X Y$ 
begin
  interpretation dx : profile-single dx by (rule dx')
  interpretation dy : profile-single dy by (rule dy')

```

```

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'' dx.d-cp0[symmetric] dy.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) = (dx x and dy y)
  apply(rule ext, rename-tac τ,subst foundation22[symmetric])
  apply(case-tac ¬(τ ⊨ dx x), simp)
  apply(case-tac ¬(τ ⊨ dy y), simp)
  apply(simp)
  apply(rule foundation13[THEN iffD2,THEN StrongEq-L-subst2-rev, where y = dx x])
  apply(simp-all)
  apply(rule foundation13[THEN iffD2,THEN StrongEq-L-subst2-rev, where y = dy y])
  apply(simp-all add: 1 foundation13)
  done

lemma def-valid-then-def: v(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 ⊨ dx x) ∧ (τ ⊨ dy 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. $f \text{ invalid } y = \text{invalid}$ and $f \text{ null } y = \text{invalid}$. 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  $d_y :: ('A, 'b :: \text{null}) \text{val} \Rightarrow 'A \text{ Boolean}$ 
  fixes  $f :: ('A, 'a :: \text{null}) \text{val} \Rightarrow ('A, 'b :: \text{null}) \text{val} \Rightarrow ('A, 'c :: \text{null}) \text{val}$ 
  fixes  $g$ 
  assumes  $d_y : \text{profile-single } d_y$ 
  assumes  $d_y\text{-homo}[\text{simp}, \text{code-unfold}] : \text{cp } (f X) \Longrightarrow$ 
     $f X \text{ invalid} = \text{invalid} \Longrightarrow$ 
     $\neg \tau \models d_y Y \Longrightarrow$ 
     $\tau \models \delta f X Y \triangleq (\delta X \text{ and } d_y Y)$ 
  assumes  $\text{def-scheme}'[\text{simplified}] : \text{bin } f g \text{ defined } d_y X Y$ 
  assumes  $\text{def-body}' : \bigwedge x y \tau. x \neq \text{bot} \Longrightarrow x \neq \text{null} \Longrightarrow (d_y y) \tau = \text{true} \tau \Longrightarrow g x (y \tau) \neq \text{bot} \wedge g x (y \tau) \neq \text{null}$ 
begin
  lemma  $\text{strict3}[\text{simp}, \text{code-unfold}] : f \text{ null } y = \text{invalid}$ 
  by (rule ext, simp add: def-scheme' true-def false-def)
end

```

sublocale *profile-bin-scheme-defined* < *profile-bin-scheme defined*

proof –

```

  interpret  $d_y : \text{profile-single } d_y$  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  $f :: ('A, 'a :: \text{null}) \text{val} \Rightarrow ('A, 'b :: \text{null}) \text{val} \Rightarrow ('A, 'c :: \text{null}) \text{val}$ 
  fixes  $g$ 
  assumes  $\text{def-scheme}[\text{simplified}] : \text{bin } f g \text{ defined defined } X Y$ 
  assumes  $\text{def-body} : \bigwedge x y. g x y \neq \text{bot} \wedge g x y \neq \text{null}$ 
begin
  lemma  $\text{strict4}[\text{simp}, \text{code-unfold}] : f x \text{ null} = \text{invalid}$ 
  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  $\Rightarrow$  ('A, 'b::null)val  $\Rightarrow$  ('A, 'c::null)val
  fixes g
  assumes def-scheme[simplified]: bin f g defined valid X Y
  assumes def-body:  $\bigwedge x y. x \neq \text{bot} \Rightarrow x \neq \text{null} \Rightarrow y \neq \text{bot} \Rightarrow g\ x\ y \neq \text{bot} \wedge g\ x\ y \neq \text{null}$ 

```

```

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  $\Rightarrow$  ('A, 'α::null)val  $\Rightarrow$  ('A) Boolean
  assumes def-scheme[simplified]: bin' f StrongEq valid valid X Y

```

```

sublocale profile-bin3 < profile-bin-scheme valid valid f  $\lambda x\ y. \llbracket x = y \rrbracket$ 
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 founda-
tion10' 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:  $\tau \models f\ x\ y \Rightarrow (\tau \models v\ x) \wedge (\tau \models v\ y)$ 
  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'* : $\delta (f x y) = (v x \text{ and } v y)$
by (*auto intro!*: *transform2-rev defined-and-I simp: foundation10 defined-args-valid*)

lemma *refl-ext*[*simp, code-unfold*] : $(f x x) = (\text{if } (v x) \text{ then true else invalid endif})$
by (*rule ext, simp add: def-scheme OclIf-def*)

lemma *sym* : $\tau \models (f x y) \implies \tau \models (f y x)$
apply (*case-tac* $\tau \models v x$)
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* : $\tau \models (f x y) \implies \tau \models (f y z) \implies \tau \models (f x z)$
apply (*case-tac* $\tau \models v x$)
apply (*case-tac* $\tau \models v y$)
apply (*auto simp: def-scheme OclValid-def*)
by (*fold OclValid-def, auto elim: StrongEq-L-trans*)

lemma *StrictRefEq-vs-StrongEq*: $\tau \models (v x) \implies \tau \models (v y) \implies (\tau \models ((f x y) \triangleq (x \triangleq y)))$
apply (*simp add: def-scheme OclValid-def*)
apply (*subst cp-StrongEq[of - (x \triangleq y)]*)
by *simp*

end

locale *profile-bin4* =
fixes *f* :: (' \mathcal{A} , ' α ::null)val \Rightarrow (' \mathcal{A} , ' β ::null)val \Rightarrow (' \mathcal{A} , ' γ ::null)val
fixes *g*
assumes *def-scheme*[*simplified*]: *bin f g valid valid X Y*
assumes *def-body*: $\bigwedge x y. x \neq \text{bot} \implies y \neq \text{bot} \implies g x y \neq \text{bot} \wedge g x y \neq \text{null}$

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 \mathcal{A} *Boolean*, it is just the strict extension of the logical equality:

```

defs StrictRefEqBoolean[code-unfold] :
  (x::( $\mathcal{A}$ )Boolean)  $\doteq$  y  $\equiv$   $\lambda$   $\tau$ . if ( $\vee$  x)  $\tau$  = true  $\tau$   $\wedge$  ( $\vee$  y)  $\tau$  = true  $\tau$ 
    then (x  $\triangleq$  y)  $\tau$ 
    else invalid  $\tau$ 

```

which implies elementary properties like:

```

lemma [simp,code-unfold] : (true  $\doteq$  false) = false

```

```

by(simp add:StrictRefEqBoolean)

```

```

lemma [simp,code-unfold] : (false  $\doteq$  true) = false

```

```

by(simp add:StrictRefEqBoolean)

```

```

lemma null-non-false [simp,code-unfold]:(null  $\doteq$  false) = false

```

```

apply(rule ext, simp add: StrictRefEqBoolean 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]:(null  $\doteq$  true) = false

```

```

apply(rule ext, simp add: StrictRefEqBoolean 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]:(false  $\doteq$  null) = false

```

```

apply(rule ext, simp add: StrictRefEqBoolean 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]:(true  $\doteq$  null) = false

```

```

apply(rule ext, simp add: StrictRefEqBoolean StrongEq-def false-def)

```

```

by(simp add: true-def bot-option-def null-fun-def null-option-def)

```

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

```

interpretation StrictRefEqBoolean : profile-bin3  $\lambda$  x y. (x::( $\mathcal{A}$ )Boolean)  $\doteq$  y
  by unfold-locales (auto simp:StrictRefEqBoolean)

```

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

```

lemma (invalid  $\doteq$  false) = invalid by(simp)

```

```

lemma ( $invalid \dot{=} true$ ) =  $invalid$  by( $simp$ )
lemma ( $false \dot{=} invalid$ ) =  $invalid$  by( $simp$ )
lemma ( $true \dot{=} invalid$ ) =  $invalid$  by( $simp$ )
lemma ( $((invalid::({\mathbb{A}})Boolean) \dot{=} invalid) = invalid$  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  $\tau \models v(true)$ 
Assert  $\tau \models \delta(false)$ 
Assert  $\neg(\tau \models \delta(null))$ 
Assert  $\neg(\tau \models \delta(invalid))$ 
Assert  $\tau \models v((null::({\mathbb{A}})Boolean))$ 
Assert  $\neg(\tau \models v(invalid))$ 
Assert  $\tau \models (true \text{ and } true)$ 
Assert  $\tau \models (true \text{ and } true \triangleq true)$ 
Assert  $\tau \models ((null \text{ or } null) \triangleq null)$ 
Assert  $\tau \models ((null \text{ or } null) \dot{=} null)$ 
Assert  $\tau \models ((true \triangleq false) \triangleq false)$ 
Assert  $\tau \models ((invalid \triangleq false) \triangleq false)$ 
Assert  $\tau \models ((invalid \dot{=} false) \triangleq invalid)$ 
Assert  $\tau \models (true <> false)$ 
Assert  $\tau \models (false <> true)$ 

```

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 $Void_{base} :: bot$

```

begin
  definition bot-Void-def: (bot-class.bot :: Voidbase) ≡ Abs-Voidbase None

  instance proof show ∃x:: Voidbase. x ≠ bot
    apply(rule-tac x=Abs-Voidbase [None] in exI)
    apply(simp add:bot-Void-def, subst Abs-Voidbase-inject)
    apply(simp-all add: null-option-def bot-option-def)
    done
  qed
end

instantiation Voidbase :: null
begin
  definition null-Void-def: (null::Voidbase) ≡ Abs-Voidbase [ None ]

  instance proof show (null:: Voidbase) ≠ bot
    apply(simp add:null-Void-def bot-Void-def, subst Abs-Voidbase-inject)
    apply(simp-all add: null-option-def bot-option-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 \mathcal{A} Void-case as strict extension of the strong equality:

```

defs StrictRefEqVoid[code-unfold] :
  (x::( $\mathcal{A}$ )Void) ≐ y ≡ λ τ. if (v x) τ = true τ ∧ (v y) τ = true τ
    then (x ≐ y) τ
    else invalid τ

```

Property proof in terms of *profile-bin3*

```

interpretation StrictRefEqVoid : profile-bin3 λ x y. (x::( $\mathcal{A}$ )Void) ≐ y
  by unfold-locales (auto simp: StrictRefEqVoid)

```

Test Statements

```

Assert τ |= ((null::( $\mathcal{A}$ )Void) ≐ 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* :: (^{'A})Integer (0)
where 0 = (λ - . $\llbracket 0::int \rrbracket$)

definition *OclInt1* :: (^{'A})Integer (1)
where 1 = (λ - . $\llbracket 1::int \rrbracket$)

definition *OclInt2* :: (^{'A})Integer (2)
where 2 = (λ - . $\llbracket 2::int \rrbracket$)

definition *OclInt3* :: (^{'A})Integer (3)
where 3 = (λ - . $\llbracket 3::int \rrbracket$)

definition *OclInt4* :: (^{'A})Integer (4)
where 4 = (λ - . $\llbracket 4::int \rrbracket$)

definition *OclInt5* :: (^{'A})Integer (5)
where 5 = (λ - . $\llbracket 5::int \rrbracket$)

definition *OclInt6* :: (^{'A})Integer (6)
where 6 = (λ - . $\llbracket 6::int \rrbracket$)

definition *OclInt7* :: (^{'A})Integer (7)
where 7 = (λ - . $\llbracket 7::int \rrbracket$)

definition *OclInt8* :: (^{'A})Integer (8)
where 8 = (λ - . $\llbracket 8::int \rrbracket$)

definition *OclInt9* :: (^{'A})Integer (9)
where 9 = (λ - . $\llbracket 9::int \rrbracket$)

definition *OclInt10* :: (^{'A})Integer (10)
where 10 = (λ - . $\llbracket 10::int \rrbracket$)

Validity and Definedness Properties

lemma $\delta(\text{null}::(\sup{'A})Integer) = \text{false}$ **by** *simp*

lemma $v(\text{null}::(\sup{'A})Integer) = \text{true}$ **by** *simp*

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

lemma $[simp, code-unfold]: v (\lambda -. \llbracket n \rrbracket) = true$
by (simp add: valid-def true-def
 bot-fun-def bot-option-def)

lemma $[simp, code-unfold]: \delta 0 = true$ **by** (simp add: OclInt0-def)
lemma $[simp, code-unfold]: v 0 = true$ **by** (simp add: OclInt0-def)
lemma $[simp, code-unfold]: \delta 1 = true$ **by** (simp add: OclInt1-def)
lemma $[simp, code-unfold]: v 1 = true$ **by** (simp add: OclInt1-def)
lemma $[simp, code-unfold]: \delta 2 = true$ **by** (simp add: OclInt2-def)
lemma $[simp, code-unfold]: v 2 = true$ **by** (simp add: OclInt2-def)
lemma $[simp, code-unfold]: \delta 6 = true$ **by** (simp add: OclInt6-def)
lemma $[simp, code-unfold]: v 6 = true$ **by** (simp add: OclInt6-def)
lemma $[simp, code-unfold]: \delta 8 = true$ **by** (simp add: OclInt8-def)
lemma $[simp, code-unfold]: v 8 = true$ **by** (simp add: OclInt8-def)
lemma $[simp, code-unfold]: \delta 9 = true$ **by** (simp add: OclInt9-def)
lemma $[simp, code-unfold]: v 9 = true$ **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 $OclAdd_{Integer} :: ('A)Integer \Rightarrow ('A)Integer \Rightarrow ('A)Integer$ (**infix** $+_{int}$ 40)
where $x +_{int} y \equiv \lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (\delta y) \tau = true \tau$
 then $\llbracket \llbracket x \rrbracket \tau \rrbracket + \llbracket \llbracket y \rrbracket \tau \rrbracket$
 else $invalid \tau$

interpretation $OclAdd_{Integer} : profile-bin1 \text{ op } +_{int} \lambda x y. \llbracket \llbracket x \rrbracket \rrbracket + \llbracket \llbracket y \rrbracket \rrbracket$
by *unfold-locales* (auto simp: OclAdd_{Integer}-def bot-option-def null-option-def)

definition $OclMinus_{Integer} :: ('A)Integer \Rightarrow ('A)Integer \Rightarrow ('A)Integer$ (**infix** $-_{int}$ 41)
where $x -_{int} y \equiv \lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (\delta y) \tau = true \tau$
 then $\llbracket \llbracket x \rrbracket \tau \rrbracket - \llbracket \llbracket y \rrbracket \tau \rrbracket$
 else $invalid \tau$

interpretation $OclMinus_{Integer} : profile-bin1 \text{ op } -_{int} \lambda x y. \llbracket \llbracket x \rrbracket \rrbracket - \llbracket \llbracket y \rrbracket \rrbracket$
by *unfold-locales* (auto simp: OclMinus_{Integer}-def bot-option-def null-option-def)

definition $OclMult_{Integer} :: ('A)Integer \Rightarrow ('A)Integer \Rightarrow ('A)Integer$ (**infix** $*_{int}$ 45)
where $x *_{int} y \equiv \lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (\delta y) \tau = true \tau$
 then $\llbracket \llbracket x \rrbracket \tau \rrbracket * \llbracket \llbracket y \rrbracket \tau \rrbracket$
 else $invalid \tau$

interpretation $OclMult_{Integer} : profile-bin1 \text{ op } *_{int} \lambda x y. \llbracket \llbracket x \rrbracket \rrbracket * \llbracket \llbracket y \rrbracket \rrbracket$
by *unfold-locales* (auto simp: OclMult_{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 $OclDivision_{Integer} :: (^{\mathcal{A}})Integer \Rightarrow (^{\mathcal{A}})Integer \Rightarrow (^{\mathcal{A}})Integer$ (**infix** div_{int} 45)
where $x \mathit{div}_{int} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \tau \wedge (\delta y) \tau = \text{true} \tau$
 then if $y \tau \neq OclInt0 \tau$ *then* $[[\lceil x \tau \rceil] \mathit{div} \lceil y \tau \rceil]]$ *else* *invalid* τ
 else *invalid* τ

definition $OclModulus_{Integer} :: (^{\mathcal{A}})Integer \Rightarrow (^{\mathcal{A}})Integer \Rightarrow (^{\mathcal{A}})Integer$ (**infix** mod_{int} 45)
where $x \mathit{mod}_{int} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \tau \wedge (\delta y) \tau = \text{true} \tau$
 then if $y \tau \neq OclInt0 \tau$ *then* $[[\lceil x \tau \rceil] \mathit{mod} \lceil y \tau \rceil]]$ *else* *invalid* τ
 else *invalid* τ

definition $OclLess_{Integer} :: (^{\mathcal{A}})Integer \Rightarrow (^{\mathcal{A}})Integer \Rightarrow (^{\mathcal{A}})Boolean$ (**infix** $<_{int}$ 35)
where $x <_{int} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \tau \wedge (\delta y) \tau = \text{true} \tau$
 then $[[\lceil x \tau \rceil] < \lceil y \tau \rceil]]$
 else *invalid* τ

interpretation $OclLess_{Integer} : \text{profile-bin1 } op <_{int} \lambda x y. [[\lceil x \rceil] < \lceil y \rceil]]$
by *unfold-locale* (*auto simp:OclLessInteger-def bot-option-def null-option-def*)

definition $OclLe_{Integer} :: (^{\mathcal{A}})Integer \Rightarrow (^{\mathcal{A}})Integer \Rightarrow (^{\mathcal{A}})Boolean$ (**infix** \leq_{int} 35)
where $x \leq_{int} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \tau \wedge (\delta y) \tau = \text{true} \tau$
 then $[[\lceil x \tau \rceil] \leq \lceil y \tau \rceil]]$
 else *invalid* τ

interpretation $OclLe_{Integer} : \text{profile-bin1 } op \leq_{int} \lambda x y. [[\lceil x \rceil] \leq \lceil y \rceil]]$
by *unfold-locale* (*auto simp:OclLeInteger-def bot-option-def null-option-def*)

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

Execution with Invalid or Null or Zero as Argument **lemma** $OclAdd_{Integer}\text{-zero1}[simp,code-unfold] :$
 $(x +_{int} 0) = (\text{if } \nu x \text{ and not } (\delta x) \text{ then invalid else } x \text{ endif})$
proof (*rule ext, rename-tac τ , case-tac $(\nu x \text{ and not } (\delta x)) \tau = \text{true } \tau$*)
fix τ **show** $(\nu x \text{ and not } (\delta x)) \tau = \text{true } \tau \implies$
 $(x +_{int} 0) \tau = (\text{if } \nu x \text{ and not } (\delta x) \text{ then invalid else } x \text{ endif}) \tau$
apply(*subst OclIf-true', simp add: OclValid-def*)
by (*metis OclAddInteger-def OclNot-defargs OclValid-def foundation5 foundation9*)
apply-end *assumption*
next fix τ
have $A: \bigwedge \tau. (\tau \models \text{not } (\nu x \text{ and not } (\delta x))) = (x \tau = \text{invalid } \tau \vee \tau \models \delta x)$
by (*metis OclNot-not OclOr-def defined5 defined6 defined-not-I foundation11 foundation18'*
 foundation6 foundation7 foundation9 invalid-def)
have $B: \tau \models \delta x \implies [[\lceil x \tau \rceil]] = x \tau$
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  $\tau \models \text{not } (\vee x \text{ and not } (\delta x)) \implies$ 
       $(x +_{\text{int}} \mathbf{0}) \tau = (\text{if } \vee x \text{ and not } (\delta x) \text{ then invalid else } x \text{ endif}) \tau$ 
apply(subst OclIf-false', simp, simp add: A, auto simp: OclAddInteger-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 OclAddInteger-zero2 [simp,code-unfold] :
   $(\mathbf{0} +_{\text{int}} x) = (\text{if } \vee x \text{ and not } (\delta x) \text{ then invalid else } x \text{ endif})$ 
by(subst OclAddInteger-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  $\tau \models (\mathbf{9} \leq_{\text{int}} \mathbf{10})$ 
Assert  $\tau \models ((\mathbf{4} +_{\text{int}} \mathbf{4}) \leq_{\text{int}} \mathbf{10})$ 
Assert  $\neg(\tau \models ((\mathbf{4} +_{\text{int}} (\mathbf{4} +_{\text{int}} \mathbf{4})) <_{\text{int}} \mathbf{10}))$ 
Assert  $\tau \models \text{not } (\vee (\text{null} +_{\text{int}} \mathbf{1}))$ 
Assert  $\tau \models (((\mathbf{9} *_{\text{int}} \mathbf{4}) \text{div}_{\text{int}} \mathbf{10}) \leq_{\text{int}} \mathbf{4})$ 
Assert  $\tau \models \text{not } (\delta (\mathbf{1} \text{div}_{\text{int}} \mathbf{0}))$ 
Assert  $\tau \models \text{not } (\vee (\mathbf{1} \text{div}_{\text{int}} \mathbf{0}))$ 

```

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 $^{\mathcal{A}}$ *Boolean*-case as strict extension of the strong equality:

```

defs StrictRefEqInteger [code-unfold] :
   $(x :: (^{\mathcal{A}})\text{Integer}) \doteq y \equiv \lambda \tau. \text{if } (\vee x) \tau = \text{true } \tau \wedge (\vee y) \tau = \text{true } \tau$ 
     $\text{then } (x \triangleq y) \tau$ 
     $\text{else invalid } \tau$ 

```

Property proof in terms of *profile-bin3*

```

interpretation StrictRefEqInteger : profile-bin3  $\lambda x y. (x :: (^{\mathcal{A}})\text{Integer}) \doteq y$ 
by unfold-locales (auto simp: StrictRefEqInteger)

```

```

lemma integer-non-null [simp]:  $((\lambda -. \lfloor \lfloor n \rfloor \rfloor) \doteq (\text{null} :: (^{\mathcal{A}})\text{Integer})) = \text{false}$ 
by(rule ext,auto simp: StrictRefEqInteger valid-def
    bot-fun-def bot-option-def null-fun-def null-option-def StrongEq-def)

```

```

lemma null-non-integer [simp]:  $((\text{null} :: (^{\mathcal{A}})\text{Integer}) \doteq (\lambda -. \lfloor \lfloor n \rfloor \rfloor)) = \text{false}$ 
by(rule ext,auto simp: StrictRefEqInteger valid-def)

```

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

```

lemma OclInt0-non-null [simp,code-unfold]: (0  $\doteq$  null) = false by(simp add: OclInt0-def)
lemma null-non-OclInt0 [simp,code-unfold]: (null  $\doteq$  0) = false by(simp add: OclInt0-def)
lemma OclInt1-non-null [simp,code-unfold]: (1  $\doteq$  null) = false by(simp add: OclInt1-def)
lemma null-non-OclInt1 [simp,code-unfold]: (null  $\doteq$  1) = false by(simp add: OclInt1-def)
lemma OclInt2-non-null [simp,code-unfold]: (2  $\doteq$  null) = false by(simp add: OclInt2-def)
lemma null-non-OclInt2 [simp,code-unfold]: (null  $\doteq$  2) = false by(simp add: OclInt2-def)
lemma OclInt6-non-null [simp,code-unfold]: (6  $\doteq$  null) = false by(simp add: OclInt6-def)
lemma null-non-OclInt6 [simp,code-unfold]: (null  $\doteq$  6) = false by(simp add: OclInt6-def)
lemma OclInt8-non-null [simp,code-unfold]: (8  $\doteq$  null) = false by(simp add: OclInt8-def)
lemma null-non-OclInt8 [simp,code-unfold]: (null  $\doteq$  8) = false by(simp add: OclInt8-def)
lemma OclInt9-non-null [simp,code-unfold]: (9  $\doteq$  null) = false by(simp add: OclInt9-def)
lemma null-non-OclInt9 [simp,code-unfold]: (null  $\doteq$  9) = 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 $\tau \models ((\mathbf{0} <_{int} \mathbf{2}) \text{ and } (\mathbf{0} <_{int} \mathbf{1}))$

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

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

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

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

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

Assert $\tau \models v \ (null::(^{\mathcal{A}})Integer)$

Assert $\tau \models (invalid \triangleq invalid)$

Assert $\tau \models (null \triangleq null)$

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

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

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

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

Assert $\neg(\tau \models (invalid \doteq (invalid::(^{\mathcal{A}})Integer)))$

Assert $\neg(\tau \models v \ (invalid \doteq (invalid::(^{\mathcal{A}})Integer)))$

Assert $\neg(\tau \models (invalid <> (invalid::(^{\mathcal{A}})Integer)))$

Assert $\neg(\tau \models v \ (invalid <> (invalid::(^{\mathcal{A}})Integer)))$

Assert $\tau \models (null \doteq (null::(^{\mathcal{A}})Integer))$

Assert $\tau \models (null \doteq (null::(^{\mathcal{A}})Integer))$

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

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

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 :: (ℵℚ)Real (0.0)
where 0.0 = (λ - . [[0::real]])
```

```
definition OclReal1 :: (ℵℚ)Real (1.0)
where 1.0 = (λ - . [[1::real]])
```

```
definition OclReal2 :: (ℵℚ)Real (2.0)
where 2.0 = (λ - . [[2::real]])
```

```
definition OclReal3 :: (ℵℚ)Real (3.0)
where 3.0 = (λ - . [[3::real]])
```

```
definition OclReal4 :: (ℵℚ)Real (4.0)
where 4.0 = (λ - . [[4::real]])
```

```
definition OclReal5 :: (ℵℚ)Real (5.0)
where 5.0 = (λ - . [[5::real]])
```

```
definition OclReal6 :: (ℵℚ)Real (6.0)
where 6.0 = (λ - . [[6::real]])
```

```
definition OclReal7 :: (ℵℚ)Real (7.0)
where 7.0 = (λ - . [[7::real]])
```

```
definition OclReal8 :: (ℵℚ)Real (8.0)
where 8.0 = (λ - . [[8::real]])
```

```
definition OclReal9 :: (ℵℚ)Real (9.0)
where 9.0 = (λ - . [[9::real]])
```

```
definition OclReal10 :: (ℵℚ)Real (10.0)
```

where $10.0 = (\lambda -. \llbracket 10::real \rrbracket)$

definition $OclRealpi :: (^{\mathcal{A}})Real (\pi)$

where $\pi = (\lambda -. \llbracket pi \rrbracket)$

Validity and Definedness Properties

lemma $\delta(null::(^{\mathcal{A}})Real) = false$ **by** *simp*

lemma $v(null::(^{\mathcal{A}})Real) = true$ **by** *simp*

lemma $[simp,code-unfold]: \delta(\lambda -. \llbracket n \rrbracket) = true$
by (*simp add:defined-def true-def*
bot-fun-def bot-option-def null-fun-def null-option-def)

lemma $[simp,code-unfold]: v(\lambda -. \llbracket n \rrbracket) = true$
by (*simp add:valid-def true-def*
bot-fun-def bot-option-def)

lemma $[simp,code-unfold]: \delta 0.0 = true$ **by** (*simp add:OclReal0-def*)

lemma $[simp,code-unfold]: v 0.0 = true$ **by** (*simp add:OclReal0-def*)

lemma $[simp,code-unfold]: \delta 1.0 = true$ **by** (*simp add:OclReal1-def*)

lemma $[simp,code-unfold]: v 1.0 = true$ **by** (*simp add:OclReal1-def*)

lemma $[simp,code-unfold]: \delta 2.0 = true$ **by** (*simp add:OclReal2-def*)

lemma $[simp,code-unfold]: v 2.0 = true$ **by** (*simp add:OclReal2-def*)

lemma $[simp,code-unfold]: \delta 6.0 = true$ **by** (*simp add:OclReal6-def*)

lemma $[simp,code-unfold]: v 6.0 = true$ **by** (*simp add:OclReal6-def*)

lemma $[simp,code-unfold]: \delta 8.0 = true$ **by** (*simp add:OclReal8-def*)

lemma $[simp,code-unfold]: v 8.0 = true$ **by** (*simp add:OclReal8-def*)

lemma $[simp,code-unfold]: \delta 9.0 = true$ **by** (*simp add:OclReal9-def*)

lemma $[simp,code-unfold]: v 9.0 = true$ **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 $OclAdd_{Real} :: (^{\mathcal{A}})Real \Rightarrow (^{\mathcal{A}})Real \Rightarrow (^{\mathcal{A}})Real$ (**infix** $+_{real}$ 40)

where $x +_{real} y \equiv \lambda \tau. \text{if } (\delta x) \tau = true \wedge (\delta y) \tau = true \text{ then } \llbracket \llbracket x \rrbracket \tau \rrbracket + \llbracket \llbracket y \rrbracket \tau \rrbracket$
else invalid τ

interpretation $OclAdd_{Real} : profile-bin1 \text{ op } +_{real} \lambda x y. \llbracket \llbracket x \rrbracket \rrbracket + \llbracket \llbracket y \rrbracket \rrbracket$
by *unfold-locales (auto simp:OclAdd_{Real}-def bot-option-def null-option-def)*

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

definition $OclMult_{Real} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Real$ (**infix** $*_{real}$ 45)
where $x *_{real} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
 then $[[[x \tau]] * [[y \tau]]]$
 else $\text{invalid } \tau$
interpretation $OclMult_{Real} : \text{profile-bin1 op } *_{real} \lambda x y. [[x] * [y]]$
 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 $OclDivision_{Real} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Real$ (**infix** div_{real} 45)
where $x \text{div}_{real} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
 then if $y \neq OclReal0 \tau$ *then* $[[[x \tau]] / [[y \tau]]]$ *else* $\text{invalid } \tau$
 else $\text{invalid } \tau$

definition $\text{mod-float } a b = a -_{real} (\text{floor } (a / b)) * b$
definition $OclModulus_{Real} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Real$ (**infix** mod_{real} 45)
where $x \text{mod}_{real} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
 then if $y \neq OclReal0 \tau$ *then* $[[\text{mod-float } [x \tau] [y \tau]]]$ *else* $\text{invalid } \tau$
 else $\text{invalid } \tau$

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

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

Basic Properties **lemma** $OclAdd_{Real}\text{-commute}: (X +_{real} Y) = (Y +_{real} X)$
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 *OclAdd_{Real}-zero1*[simp,code-unfold] :

```

(x +real 0.0) = (if v x and not (δ x) then invalid else x endif)
proof (rule ext, rename-tac τ, case-tac (v x and not (δ x)) τ = true τ)
fix τ show (v x and not (δ x)) τ = true τ  $\implies$ 
    (x +real 0.0) τ = (if v x and not (δ x) then invalid else x endif) τ
apply(subst OclIf-true', simp add: OclValid-def)
by (metis OclAddReal-def OclNot-defargs OclValid-def foundation5 foundation9)
apply-end assumption
next fix τ
have A:  $\bigwedge \tau. (\tau \models \text{not } (v x \text{ and not } (\delta x))) = (x \tau = \text{invalid } \tau \vee \tau \models \delta x)$ 
by (metis OclNot-not OclOr-def defined5 defined6 defined-not-I foundation11 foundation18'
    foundation6 foundation7 foundation9 invalid-def)
have B:  $\tau \models \delta x \implies \llbracket \tau \rrbracket = x \tau$ 
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  $\tau \models \text{not } (v x \text{ and not } (\delta x)) \implies$ 
    (x +real 0.0) τ = (if v x and not (δ x) then invalid else x endif) τ
apply(subst OclIf-false', simp, simp add: A, auto simp: OclAddReal-def OclReal0-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_{Real}-zero2*[simp,code-unfold] :
 (0.0 +_{real} x) = (if v x and not (δ x) then invalid else x endif)
by(subst OclAdd_{Real}-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 τ  $\models (9.0 \leq_{\text{real}} 10.0)$ 
Assert τ  $\models ((4.0 +_{\text{real}} 4.0) \leq_{\text{real}} 10.0)$ 
Assert  $\neg(\tau \models ((4.0 +_{\text{real}} (4.0 +_{\text{real}} 4.0)) <_{\text{real}} 10.0))$ 
Assert τ  $\models \text{not } (v (\text{null} +_{\text{real}} 1.0))$ 
Assert τ  $\models (((9.0 *_{\text{real}} 4.0) \text{div}_{\text{real}} 10.0) \leq_{\text{real}} 4.0)$ 
Assert τ  $\models \text{not } (\delta (1.0 \text{div}_{\text{real}} 0.0))$ 
Assert τ  $\models \text{not } (v (1.0 \text{div}_{\text{real}} 0.0))$ 

```

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 *Boolean*-case as strict extension of the strong equality:

```

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

```

then $(x \triangleq y) \tau$
 else invalid τ

Property proof in terms of *profile-bin3*

interpretation *StrictRefEqReal* : *profile-bin3* $\lambda x y. (x :: (^{\mathcal{A}})Real) \dot{=} y$
by *unfold-locales* (*auto simp: StrictRefEqReal*)

lemma *real-non-null* [*simp*]: $((\lambda -. \lfloor \lfloor n \rfloor \rfloor) \dot{=} (null :: (^{\mathcal{A}})Real)) = false$
by (*rule ext, auto simp: StrictRefEqReal valid-def*
bot-fun-def bot-option-def null-fun-def null-option-def StrongEq-def)

lemma *null-non-real* [*simp*]: $((null :: (^{\mathcal{A}})Real) \dot{=} (\lambda -. \lfloor \lfloor n \rfloor \rfloor)) = false$
by (*rule ext, auto simp: StrictRefEqReal valid-def*
bot-fun-def bot-option-def null-fun-def null-option-def StrongEq-def)

lemma *OclReal0-non-null* [*simp, code-unfold*]: $(0.0 \dot{=} null) = false$ **by** (*simp add: OclReal0-def*)
lemma *null-non-OclReal0* [*simp, code-unfold*]: $(null \dot{=} 0.0) = false$ **by** (*simp add: OclReal0-def*)
lemma *OclReal1-non-null* [*simp, code-unfold*]: $(1.0 \dot{=} null) = false$ **by** (*simp add: OclReal1-def*)
lemma *null-non-OclReal1* [*simp, code-unfold*]: $(null \dot{=} 1.0) = false$ **by** (*simp add: OclReal1-def*)
lemma *OclReal2-non-null* [*simp, code-unfold*]: $(2.0 \dot{=} null) = false$ **by** (*simp add: OclReal2-def*)
lemma *null-non-OclReal2* [*simp, code-unfold*]: $(null \dot{=} 2.0) = false$ **by** (*simp add: OclReal2-def*)
lemma *OclReal6-non-null* [*simp, code-unfold*]: $(6.0 \dot{=} null) = false$ **by** (*simp add: OclReal6-def*)
lemma *null-non-OclReal6* [*simp, code-unfold*]: $(null \dot{=} 6.0) = false$ **by** (*simp add: OclReal6-def*)
lemma *OclReal8-non-null* [*simp, code-unfold*]: $(8.0 \dot{=} null) = false$ **by** (*simp add: OclReal8-def*)
lemma *null-non-OclReal8* [*simp, code-unfold*]: $(null \dot{=} 8.0) = false$ **by** (*simp add: OclReal8-def*)
lemma *OclReal9-non-null* [*simp, code-unfold*]: $(9.0 \dot{=} null) = false$ **by** (*simp add: OclReal9-def*)
lemma *null-non-OclReal9* [*simp, code-unfold*]: $(null \dot{=} 9.0) = false$ **by** (*simp add: OclReal9-def*)

Const lemma [*simp, code-unfold*]: *const*(0.0) **by** (*simp add: const-ss OclReal0-def*)
lemma [*simp, code-unfold*]: *const*(1.0) **by** (*simp add: const-ss OclReal1-def*)
lemma [*simp, code-unfold*]: *const*(2.0) **by** (*simp add: const-ss OclReal2-def*)
lemma [*simp, code-unfold*]: *const*(6.0) **by** (*simp add: const-ss OclReal6-def*)
lemma [*simp, code-unfold*]: *const*(8.0) **by** (*simp add: const-ss OclReal8-def*)
lemma [*simp, code-unfold*]: *const*(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 $\tau \models 1.0 <> 2.0$
Assert $\tau \models 2.0 <> 1.0$
Assert $\tau \models 2.0 \dot{=} 2.0$

Assert $\tau \models v \ 4.0$
Assert $\tau \models \delta \ 4.0$

```

Assert  $\tau \models v \text{ (null::('A)Real)}$ 
Assert  $\tau \models (\text{invalid} \triangleq \text{invalid})$ 
Assert  $\tau \models (\text{null} \triangleq \text{null})$ 
Assert  $\tau \models (\mathbf{4.0} \triangleq \mathbf{4.0})$ 
Assert  $\neg(\tau \models (\mathbf{9.0} \triangleq \mathbf{10.0}))$ 
Assert  $\neg(\tau \models (\text{invalid} \triangleq \mathbf{10.0}))$ 
Assert  $\neg(\tau \models (\text{null} \triangleq \mathbf{10.0}))$ 
Assert  $\neg(\tau \models (\text{invalid} \doteq (\text{invalid::('A)Real})))$ 
Assert  $\neg(\tau \models v \text{ (invalid} \doteq (\text{invalid::('A)Real})))$ 
Assert  $\neg(\tau \models (\text{invalid} <> (\text{invalid::('A)Real})))$ 
Assert  $\neg(\tau \models v \text{ (invalid} <> (\text{invalid::('A)Real})))$ 
Assert  $\tau \models (\text{null} \doteq (\text{null::('A)Real}) )$ 
Assert  $\tau \models (\text{null} \doteq (\text{null::('A)Real}) )$ 
Assert  $\tau \models (\mathbf{4.0} \doteq \mathbf{4.0})$ 
Assert  $\neg(\tau \models (\mathbf{4.0} <> \mathbf{4.0}))$ 
Assert  $\neg(\tau \models (\mathbf{4.0} \doteq \mathbf{10.0}))$ 
Assert  $\tau \models (\mathbf{4.0} <> \mathbf{10.0})$ 
Assert  $\neg(\tau \models (\mathbf{0.0} <_{real} \text{null}))$ 
Assert  $\neg(\tau \models (\delta \text{ (}\mathbf{0.0} <_{real} \text{null})))$ 

```

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 ::('A)String (a)
where    a = ( $\lambda$  - .  $\llbracket \text{"a"} \rrbracket$ )

```

```

definition OclStringb ::('A)String (b)
where    b = ( $\lambda$  - .  $\llbracket \text{"b"} \rrbracket$ )

```

```

definition OclStringc ::('A)String (c)
where    c = ( $\lambda$  - .  $\llbracket \text{"c"} \rrbracket$ )

```

Validity and Definedness Properties

```

lemma  $\delta(\text{null::('A)String}) = \text{false}$  by simp

```


lemma $v(\text{null}::({}^{\mathcal{A}})\text{String}) = \text{true}$ **by** *simp*

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

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

lemma $[\text{simp}, \text{code-unfold}]: \delta a = \text{true}$ **by** (*simp add: OclStringa-def*)
lemma $[\text{simp}, \text{code-unfold}]: v a = \text{true}$ **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 $\text{OclAddString} :: ({}^{\mathcal{A}})\text{String} \Rightarrow ({}^{\mathcal{A}})\text{String} \Rightarrow ({}^{\mathcal{A}})\text{String}$ (**infix** $+_{\text{string}}$ 40)
where $x +_{\text{string}} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \wedge (\delta y) \tau = \text{true} \wedge$
 $\text{then } \llbracket \text{concat } [\llbracket x \tau \rrbracket, \llbracket y \tau \rrbracket] \rrbracket$
 $\text{else invalid } \tau$
interpretation $\text{OclAddString} : \text{profile-bin1 op } +_{\text{string}} \lambda x y. \llbracket \text{concat } [\llbracket x \rrbracket, \llbracket y \rrbracket] \rrbracket$
by *unfold-locales (auto simp: OclAddString-def bot-option-def null-option-def)*

Basic Properties **lemma** $\text{OclAddString-not-commute}: \exists X Y. (X +_{\text{string}} Y) \neq (Y +_{\text{string}} X)$
apply (*rule-tac x = \lambda-. \llbracket "b" \rrbracket in exI*)
apply (*rule-tac x = \lambda-. \llbracket "a" \rrbracket in exI*)
apply (*simp-all add: OclAddString-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 ${}^{\mathcal{A}}$ *Boolean*-case as strict extension of the strong equality:

defs $\text{StrictRefEqString}[\text{code-unfold}] :$
 $(x::({}^{\mathcal{A}})\text{String}) = y \equiv \lambda \tau. \text{if } (v x) \tau = \text{true} \wedge (v y) \tau = \text{true} \wedge$
 $\text{then } (x \triangleq y) \tau$
 $\text{else invalid } \tau$

Property proof in terms of *profile-bin3*

interpretation *StrictRefEqString* : *profile-bin3* $\lambda x y. (x::('A)String) \doteq y$
by *unfold-locales* (*auto simp: StrictRefEqString*)

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  $\tau \models a <> b$   
Assert  $\tau \models b <> a$   
Assert  $\tau \models b \doteq b$   
  
Assert  $\tau \models v\ a$   
Assert  $\tau \models \delta\ a$   
Assert  $\tau \models v\ (null::('A)String)$   
Assert  $\tau \models (invalid \triangleq invalid)$   
Assert  $\tau \models (null \triangleq null)$   
Assert  $\tau \models (a \triangleq a)$   
Assert  $\neg(\tau \models (a \triangleq b))$   
Assert  $\neg(\tau \models (invalid \triangleq b))$   
Assert  $\neg(\tau \models (null \triangleq b))$   
Assert  $\neg(\tau \models (invalid \doteq (invalid::('A)String)))$   
Assert  $\neg(\tau \models v\ (invalid \doteq (invalid::('A)String)))$   
Assert  $\neg(\tau \models (invalid <> (invalid::('A)String)))$   
Assert  $\neg(\tau \models v\ (invalid <> (invalid::('A)String)))$   
Assert  $\tau \models (null \doteq (null::('A)String))$   
Assert  $\tau \models (null \doteq (null::('A)String))$   
Assert  $\tau \models (b \doteq b)$   
Assert  $\neg(\tau \models (b <> b))$   
Assert  $\neg(\tau \models (b \doteq c))$   
Assert  $\tau \models (b <> c)$ 
```

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 developed, 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 $A[simp]: Rep-Pair_{base} x \neq None \implies Rep-Pair_{base} x \neq null \implies (fst \llbracket Rep-Pair_{base} x \rrbracket) \neq bot$
by (insert $Rep-Pair_{base}[of\ x]$, auto simp: null-option-def bot-option-def)

lemma $A'[simp]: x \neq bot \implies x \neq null \implies (fst \llbracket Rep-Pair_{base} x \rrbracket) \neq bot$
apply (insert $Rep-Pair_{base}[of\ x]$, simp add: bot-Pair_{base}-def null-Pair_{base}-def)
apply (auto simp: null-option-def bot-option-def)
apply (erule contrapos-np[$of\ x = Abs-Pair_{base}\ None$])
apply (subst $Rep-Pair_{base}$ -inject[symmetric], simp)
apply (subst $Pair_{base}.Abs-Pair_{base}$ -inverse, simp-all, simp add: bot-option-def)
apply (erule contrapos-np[$of\ x = Abs-Pair_{base}\ [None]$])
apply (subst $Rep-Pair_{base}$ -inject[symmetric], simp)
apply (subst $Pair_{base}.Abs-Pair_{base}$ -inverse, simp-all, simp add: null-option-def bot-option-def)
done

lemma $B[simp]: Rep-Pair_{base} x \neq None \implies Rep-Pair_{base} x \neq null \implies (snd \llbracket Rep-Pair_{base} x \rrbracket) \neq bot$
by (insert $Rep-Pair_{base}[of\ x]$, auto simp: null-option-def bot-option-def)

lemma $B'[simp]: x \neq bot \implies x \neq null \implies (snd \llbracket Rep-Pair_{base} x \rrbracket) \neq bot$
apply (insert $Rep-Pair_{base}[of\ x]$, simp add: bot-Pair_{base}-def null-Pair_{base}-def)
apply (auto simp: null-option-def bot-option-def)
apply (erule contrapos-np[$of\ x = Abs-Pair_{base}\ None$])
apply (subst $Rep-Pair_{base}$ -inject[symmetric], simp)
apply (subst $Pair_{base}.Abs-Pair_{base}$ -inverse, simp-all, simp add: bot-option-def)
apply (erule contrapos-np[$of\ x = Abs-Pair_{base}\ [None]$])
apply (subst $Rep-Pair_{base}$ -inject[symmetric], simp)
apply (subst $Pair_{base}.Abs-Pair_{base}$ -inverse, simp-all, simp add: null-option-def bot-option-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 $StrictRefEq_{Pair} :$
 $((x::('A, 'B::null) Pair) \doteq y) \equiv (\lambda \tau. \text{if } (v\ x)\ \tau = true\ \tau \wedge (v\ y)\ \tau = true\ \tau$
 $\quad \text{then } (x \triangleq y)\ \tau$
 $\quad \text{else invalid } \tau)$

Property proof in terms of *profile-bin3*

interpretation *StrictRefEqPair* : *profile-bin3* $\lambda x y. (x :: ('A, 'a :: null, 'b :: null) Pair) \doteq y$
by *unfold-locales* (*auto simp: StrictRefEqPair*)

Standard Operations

This part provides a collection of operators for the *Pair* type.

Definition: OclPair Constructor **definition** *OclPair*::('A, 'a) val \Rightarrow

(*'A*, *'a*) val \Rightarrow
(*'A*, *'a*::*null*, *'b*::*null*) *Pair* (*Pair*{(-),(-)})

where *Pair*{*X*,*Y*} $\equiv (\lambda \tau. \text{if } (\nu X) \tau = \text{true } \tau \wedge (\nu Y) \tau = \text{true } \tau$
then *Abs-Pair*_{base} $[[X \tau, Y \tau]]$
else *invalid* $\tau)$

interpretation *OclPair* : *profile-bin4*

OclPair $\lambda x y. \text{Abs-Pair}_{\text{base}} [[x, y]]$

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*::('A, 'a :: null, 'b :: null) *Pair* \Rightarrow ('A, 'a) val (*-* .*First*'())

where *X* .*First*() $\equiv (\lambda \tau. \text{if } (\delta X) \tau = \text{true } \tau$
then *fst* $[[Rep-Pair_{\text{base}} (X \tau)]]$
else *invalid* $\tau)$

interpretation *OclFirst* : *profile-mono2* *OclFirst* $\lambda x. \text{fst } [[Rep-Pair_{\text{base}} (x)]]$
by *unfold-locales* (*auto simp: OclFirst-def*)

Definition: OclSnd **definition** *OclSecond*::('A, 'a :: null, 'b :: null) *Pair* \Rightarrow ('A, 'b) val (*-* .*Second*'())

where *X* .*Second*() $\equiv (\lambda \tau. \text{if } (\delta X) \tau = \text{true } \tau$
then *snd* $[[Rep-Pair_{\text{base}} (X \tau)]]$
else *invalid* $\tau)$

interpretation *OclSecond* : *profile-mono2* *OclSecond* $\lambda x. \text{snd } [[Rep-Pair_{\text{base}} (x)]]$
by *unfold-locales* (*auto simp: OclSecond-def*)

Logical Properties

lemma *I* : $\tau \models \nu Y \implies \tau \models Pair\{X, Y\} . First() \triangleq X$

apply(*case-tac* $\neg(\tau \models \nu X)$)

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 :  $\tau \models v X \implies \tau \models \text{Pair}\{X,Y\}.\text{Second}() \triangleq Y$ 
apply(case-tac  $\neg(\tau \models v Y)$ )
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-Pairbase-inject null-option-def bot-option-def bot-Pairbase-def null-Pairbase-def)
by(simp add: Abs-Pairbase-inverse)

```

Execution Properties

```

lemma proj1-exec [simp, code-unfold] :  $\text{Pair}\{X,Y\}.\text{First}() = (\text{if } (v Y) \text{ then } X \text{ else invalid endif})$ 
apply(rule ext, rename-tac  $\tau$ , simp add: foundation22[symmetric])
apply(case-tac  $\neg(\tau \models v Y)$ )
apply(erule foundation7'[THEN iffD2, THEN foundation15[THEN iffD2,
    THEN StrongEq-L-subst2-rev]],simp-all)
apply(subgoal-tac  $\tau \models v Y$ )
apply(erule foundation13[THEN iffD2, THEN StrongEq-L-subst2-rev], simp-all)
by(erule 1)

```

```

lemma proj2-exec [simp, code-unfold] :  $\text{Pair}\{X,Y\}.\text{Second}() = (\text{if } (v X) \text{ then } Y \text{ else invalid endif})$ 
apply(rule ext, rename-tac  $\tau$ , simp add: foundation22[symmetric])
apply(case-tac  $\neg(\tau \models v X)$ )
apply(erule foundation7'[THEN iffD2, THEN foundation15[THEN iffD2,
    THEN StrongEq-L-subst2-rev]],simp-all)
apply(subgoal-tac  $\tau \models v X$ )
apply(erule foundation13[THEN iffD2, THEN StrongEq-L-subst2-rev], simp-all)
by(erule 2)

```

Test Statements

```

Assert  $\tau \models \text{invalid}.\text{First}() \triangleq \text{invalid}$ 
Assert  $\tau \models \text{null}.\text{First}() \triangleq \text{invalid}$ 
Assert  $\tau \models \text{null}.\text{Second}() \triangleq \text{invalid}.\text{Second}()$ 
Assert  $\tau \models \text{Pair}\{\text{invalid}, \text{true}\} \triangleq \text{invalid}$ 
Assert  $\tau \models v(\text{Pair}\{\text{null}, \text{true}\}.\text{First}())$ 
Assert  $\tau \models (\text{Pair}\{\text{null}, \text{true}\}.\text{First}()) \triangleq \text{null}$ 
Assert  $\tau \models (\text{Pair}\{\text{null}, \text{Pair}\{\text{true}, \text{invalid}\}\}.\text{First}()) \triangleq \text{invalid}$ 

```

end

```

theory UML-Set
imports ../basic-types/UML-Boolean

```

../basic-types/UML-Integer
begin

no-notation $\text{None} (\perp)$

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 occurring 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 $\text{Integer} :: (\mathcal{A}, \text{Integer}_{\text{base}}) \text{ Set}$
where $\text{Integer} \equiv (\lambda \tau. (\text{Abs-Set}_{\text{base}} \circ \text{Some} \circ \text{Some}) ((\text{Some} \circ \text{Some}) ' (\text{UNIV}::\text{int set})))$

definition $\text{Integer}_{\text{null}} :: (\mathcal{A}, \text{Integer}_{\text{base}}) \text{ Set}$
where $\text{Integer}_{\text{null}} \equiv (\lambda \tau. (\text{Abs-Set}_{\text{base}} \circ \text{Some} \circ \text{Some}) (\text{Some} ' (\text{UNIV}::\text{int option set})))$

lemma $\text{Integer-defined} : \delta \text{ Integer} = \text{true}$
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-Set_{base}-inject bot-option-def bot-Set_{base}-def null-Set_{base}-def null-option-def)

lemma $\text{Integer}_{\text{null}}\text{-defined} : \delta \text{ Integer}_{\text{null}} = \text{true}$
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-Set_{base}-inject bot-option-def bot-Set_{base}-def null-Set_{base}-def null-option-def)

This allows the theorems:

$$\tau \models \delta x \implies \tau \models (\text{Integer} \text{--} \text{>includes}(x)) \quad \tau \models \delta x \implies \tau \models \text{Integer} \triangleq (\text{Integer} \text{--} \text{>including}(x))$$

and

$$\tau \models \nu x \implies \tau \models (\text{Integer}_{\text{null}} \text{--} \text{>includes}(x)) \quad \tau \models \nu x \implies \tau \models \text{Integer}_{\text{null}} \triangleq (\text{Integer}_{\text{null}} \text{--} \text{>including}(x))$$

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*: $\tau \models (\delta X) \implies \forall x \in \llbracket \text{Rep-Set}_{\text{base}} (X \tau) \rrbracket. x \neq \text{bot}$
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* : $\tau \models \delta X$
and *e-mem* : $e \in \llbracket \text{Rep-Set}_{\text{base}} (X \tau) \rrbracket$
shows $\tau \models v (\lambda \cdot. e)$
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* : $\tau \models \delta S$
shows *Abs-Set_{base}* $\llbracket \llbracket \text{Rep-Set}_{\text{base}} (S \tau) \rrbracket \rrbracket = S \tau$
proof –
have *discr-eq-false-true* : $\wedge \tau. (\text{false } \tau = \text{true } \tau) = \text{False}$ **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** $P = \lambda S. (\text{if } S = \perp \ \tau \vee S = \text{null } \tau$
 $\text{then false } \tau \text{ else true } \tau) = \text{true } \tau \longrightarrow$
 $\text{Abs-Set}_{\text{base}} \llbracket \llbracket \text{Rep-Set}_{\text{base}} S \rrbracket \rrbracket = S$]],
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* : $(\tau :: 'A \text{ st}) \models \delta S$
shows $\exists S'. (\lambda a. (-::'A \text{ st}). a) ' \llbracket \text{Rep-Set}_{\text{base}} (S \tau) \rrbracket = (\lambda a. (-::'A \text{ st}). [a]) ' S'$
apply(rule-tac $x = (\lambda a. [a]) ' \llbracket \text{Rep-Set}_{\text{base}} (S \tau) \rrbracket$ **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]: $\delta(\text{invalid}::('A, 'A::\text{null}) \text{Set}) = \text{false}$  by simp
lemma null-set-OclNot-defined [simp,code-unfold]: $\delta(\text{null}::('A, 'A::\text{null}) \text{Set}) = \text{false}$ 
by (simp add: defined-def null-fun-def)
lemma invalid-set-valid [simp,code-unfold]: $v(\text{invalid}::('A, 'A::\text{null}) \text{Set}) = \text{false}$ 
by simp
lemma null-set-valid [simp,code-unfold]: $v(\text{null}::('A, 'A::\text{null}) \text{Set}) = \text{true}$ 
apply (simp add: valid-def null-fun-def bot-fun-def bot-Setbase-def null-Setbase-def)
apply (subst Abs-Setbase-inject, simp-all add: null-option-def bot-option-def)
done

```

... which means that we can have a type $(A, (A, (A \text{ Integer}) \text{Set}) \text{Set})$ corresponding exactly to $\text{Set}(\text{Set}(\text{Integer}))$ in OCL notation. Note that the parameter 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::('A, 'A::\text{null}) \text{Set} (\text{Set}\{\})
where Set\{\}  $\equiv (\lambda \tau. \text{Abs-Set}_{\text{base}} \llbracket \{\}::'A \text{ set} \rrbracket)$ 

```

```

lemma mtSet-defined[simp,code-unfold]: $\delta(\text{Set}\{\}) = \text{true}$ 
apply (rule ext, auto simp: mtSet-def defined-def null-Setbase-def
    bot-Setbase-def bot-fun-def null-fun-def)
by (simp-all add: Abs-Setbase-inject bot-option-def null-Setbase-def null-option-def)

```

```

lemma mtSet-valid[simp,code-unfold]: $v(\text{Set}\{\}) = \text{true}$ 
apply (rule ext, auto simp: mtSet-def valid-def null-Setbase-def
    bot-Setbase-def bot-fun-def null-fun-def)
by (simp-all add: Abs-Setbase-inject bot-option-def null-Setbase-def null-option-def)

```

```

lemma mtSet-rep-set:  $\llbracket \llbracket \text{Rep-Set}_{\text{base}} (\text{Set}\{\} \tau) \rrbracket \rrbracket = \{\}$ 
apply (simp add: mtSet-def, subst Abs-Setbase-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* :: [$(\mathcal{A}, \alpha :: \text{null}) \text{ Set}, (\mathcal{A}, \alpha) \text{ val}] \Rightarrow (\mathcal{A}, \alpha) \text{ Set}$
where *OclIncluding* $x\ y = (\lambda\ \tau. \text{ if } (\delta\ x)\ \tau = \text{true } \tau \wedge (\vee\ y)\ \tau = \text{true } \tau$
 $\text{ then } \text{Abs-Set}_{\text{base}} \llbracket \llbracket \text{Rep-Set}_{\text{base}} (x\ \tau) \rrbracket \cup \{y\ \tau\} \rrbracket$
 $\text{ else } \text{invalid } \tau)$
notation *OclIncluding* $(-->\text{including}'(-'))$

interpretation *OclIncluding* : *profile-bin2* *OclIncluding* $\lambda x\ y. \text{Abs-Set}_{\text{base}} \llbracket \llbracket \text{Rep-Set}_{\text{base}} x \rrbracket \cup \{y\} \rrbracket$

proof –

have $A : \text{None} \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$ **by** (*simp add: bot-option-def*)
have $B : \llbracket \text{None} \rrbracket \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$
by (*simp add: null-option-def bot-option-def*)
have $C : \bigwedge x\ y. x \neq \perp \implies x \neq \text{null} \implies y \neq \perp \implies$
 $\llbracket \llbracket \text{insert } y\ \llbracket \text{Rep-Set}_{\text{base}} x \rrbracket \rrbracket \rrbracket \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$
by (*auto intro!: Set-inv-lemma[simplified OclValid-def*
 $\text{ defined-def false-def true-def null-fun-def bot-fun-def}]$)
show *profile-bin2* *OclIncluding* $(\lambda x\ y. \text{Abs-Set}_{\text{base}} \llbracket \llbracket \text{Rep-Set}_{\text{base}} x \rrbracket \cup \{y\} \rrbracket)$
apply *unfold-locales*
apply (*auto simp: OclIncluding-def bot-option-def null-option-def null-Set_{base}-def bot-Set_{base}-def*)
apply (*erule-tac Q=Abs-Set_{base} [insert y [Rep-Set_{base} x]] = Abs-Set_{base} None in contrapos-pp*)
apply (*subst Abs-Set_{base}-inject[OF C A]*)
apply (*simp-all add: null-Set_{base}-def bot-Set_{base}-def bot-option-def*)
apply (*erule-tac Q=Abs-Set_{base} [insert y [Rep-Set_{base} x]] = Abs-Set_{base} [None] in contrapos-pp*)
apply (*subst Abs-Set_{base}-inject[OF C B]*)
apply (*simp-all add: null-Set_{base}-def bot-Set_{base}-def bot-option-def*)
done

qed

syntax

-OclFinset :: $\text{args} \Rightarrow (\mathcal{A}, \alpha :: \text{null}) \text{ Set} \quad (\text{Set}\{-\})$

translations

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

$\text{Set}\{x\} == \text{CONST } \text{OclIncluding} (\text{Set}\{x\})\ x$

Definition: OclExcluding **definition** *OclExcluding* :: [$(\mathcal{A}, \alpha :: \text{null}) \text{ Set}, (\mathcal{A}, \alpha) \text{ val}] \Rightarrow (\mathcal{A}, \alpha) \text{ Set}$

where *OclExcluding* $x\ y = (\lambda\ \tau. \text{ if } (\delta\ x)\ \tau = \text{true } \tau \wedge (\vee\ y)\ \tau = \text{true } \tau$
 $\text{ then } \text{Abs-Set}_{\text{base}} \llbracket \llbracket \text{Rep-Set}_{\text{base}} (x\ \tau) \rrbracket - \{y\ \tau\} \rrbracket$
 $\text{ else } \perp)$

notation *OclExcluding* $(-->\text{excluding}'(-'))$

Definition: OclIncludes **definition** *OclIncludes* :: [$(\mathcal{A}, \alpha :: \text{null}) \text{ Set}, (\mathcal{A}, \alpha) \text{ val}] \Rightarrow \mathcal{A} \text{ Boolean}$

where *OclIncludes* $x\ y = (\lambda\ \tau. \text{ if } (\delta\ x)\ \tau = \text{true } \tau \wedge (\vee\ y)\ \tau = \text{true } \tau$
 $\text{ then } \llbracket (y\ \tau) \in \llbracket \text{Rep-Set}_{\text{base}} (x\ \tau) \rrbracket \rrbracket$
 $\text{ else } \perp)$

notation *OclIncludes* $(-->\text{includes}'(-'))$

Definition: OclExcludes **definition** *OclExcludes* :: [$(\mathcal{A}, \alpha :: \text{null}) \text{ Set}, (\mathcal{A}, \alpha) \text{ val}] \Rightarrow \mathcal{A} \text{ Boolean}$

where *OclExcludes* $x\ y = (\text{not } (\text{OclIncludes } x\ y))$

notation $OclExcludes \quad (\dashrightarrow excludes'('))$

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** $OclSize \quad :: ('A, 'A::null) Set \Rightarrow 'A Integer$
where $OclSize \ x = (\lambda \ \tau. \text{if } (\delta \ x) \ \tau = true \ \tau \wedge finite(\llbracket Rep\text{-}Set_{base} \ (x \ \tau) \rrbracket))$
 $\text{then } \llbracket int(card \ \llbracket Rep\text{-}Set_{base} \ (x \ \tau) \rrbracket) \rrbracket$
 $\text{else } \perp)$

notation

$OclSize \quad (\dashrightarrow size'('))$

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** $OclIsEmpty \quad :: ('A, 'A::null) Set \Rightarrow 'A Boolean$
where $OclIsEmpty \ x = ((\vee \ x \text{ and not } (\delta \ x)) \text{ or } ((OclSize \ x) \doteq 0))$
notation $OclIsEmpty \quad (\dashrightarrow isEmpty'('))$

Definition: OclNotEmpty **definition** $OclNotEmpty \quad :: ('A, 'A::null) Set \Rightarrow 'A Boolean$
where $OclNotEmpty \ x = not(OclIsEmpty \ x)$
notation $OclNotEmpty \quad (\dashrightarrow notEmpty'('))$

Definition: OclANY **definition** $OclANY \quad :: (('A, 'A::null) Set) \Rightarrow ('A, 'A) val$
where $OclANY \ x = (\lambda \ \tau. \text{if } (\vee \ x) \ \tau = true \ \tau$
 $\text{then if } (\delta \ x \text{ and } OclNotEmpty \ x) \ \tau = true \ \tau$
 $\text{then } SOME \ y. y \in \llbracket Rep\text{-}Set_{base} \ (x \ \tau) \rrbracket$
 $\text{else } null \ \tau$
 $\text{else } \perp)$
notation $OclANY \quad (\dashrightarrow any'('))$

Definition: OclForall The definition of OclForall mimics the one of *op and*: OclForall is not a strict operation.

definition $OclForall \quad :: (('A, 'A::null) Set, ('A, 'A) val \Rightarrow ('A) Boolean) \Rightarrow 'A Boolean$
where $OclForall \ S \ P = (\lambda \ \tau. \text{if } (\delta \ S) \ \tau = true \ \tau$
 $\text{then if } (\exists x \in \llbracket Rep\text{-}Set_{base} \ (S \ \tau) \rrbracket. P(\lambda \ -. \ x) \ \tau = false \ \tau)$
 $\text{then } false \ \tau$
 $\text{else if } (\exists x \in \llbracket Rep\text{-}Set_{base} \ (S \ \tau) \rrbracket. P(\lambda \ -. \ x) \ \tau = invalid \ \tau)$
 $\text{then } invalid \ \tau$
 $\text{else if } (\exists x \in \llbracket Rep\text{-}Set_{base} \ (S \ \tau) \rrbracket. P(\lambda \ -. \ x) \ \tau = null \ \tau)$
 $\text{then } null \ \tau$
 $\text{else } true \ \tau$
 $\text{else } \perp)$

syntax

$-OclForall \ :: (('A, 'A::null) Set, id, ('A) Boolean) \Rightarrow 'A Boolean \quad ((-) \dashrightarrow forAll'(-|'))$

translations

$$X \rightarrow \text{forall}(x \mid P) == \text{CONST OclForall } X \ (\%x. P)$$

Definition: OclExists Like OclForall, OclExists is also not strict.

definition *OclExists* :: $[(\mathcal{A}, \alpha :: \text{null}) \text{ Set}, (\mathcal{A}, \alpha) \text{ val} \Rightarrow (\mathcal{A}) \text{ Boolean}] \Rightarrow \mathcal{A} \text{ Boolean}$

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

syntax

$\text{-OclExist} :: [(\mathcal{A}, \alpha :: \text{null}) \text{ Set}, \text{id}, (\mathcal{A}) \text{ Boolean}] \Rightarrow \mathcal{A} \text{ Boolean} \quad ((-) \rightarrow \text{exists}'(-|-'))$

translations

$$X \rightarrow \text{exists}(x \mid P) == \text{CONST OclExists } X \ (\%x. P)$$

Definition: OclIterate **definition** *OclIterate* :: $[(\mathcal{A}, \alpha :: \text{null}) \text{ Set}, (\mathcal{A}, \beta :: \text{null}) \text{ val}, (\mathcal{A}, \alpha) \text{ val} \Rightarrow (\mathcal{A}, \beta) \text{ val} \Rightarrow (\mathcal{A}, \beta) \text{ val}] \Rightarrow (\mathcal{A}, \beta) \text{ val}$

where $\text{OclIterate } S \ A \ F = (\lambda \tau. \text{if } (\delta \ S) \ \tau = \text{true} \ \tau \wedge (v \ A) \ \tau = \text{true} \ \tau \wedge \text{finite} \llbracket \text{Rep-Set}_{\text{base}}(S \ \tau) \rrbracket$
 $\text{then } (\text{Finite-Set.fold } (F) \ (A) \ ((\lambda a \ \tau. a) \ ' \llbracket \text{Rep-Set}_{\text{base}}(S \ \tau) \rrbracket)) \tau$
 $\text{else } \perp)$

syntax

$\text{-OclIterate} :: [(\mathcal{A}, \alpha :: \text{null}) \text{ Set}, \text{id}, \text{id}, \alpha, \beta] \Rightarrow (\mathcal{A}, \gamma) \text{ val}$
 $(- \rightarrow \text{iterate}'(-; - \mid -'))$

translations

$$X \rightarrow \text{iterate}(a; x = A \mid P) == \text{CONST OclIterate } X \ A \ (\%a. (\%x. P))$$

Definition: OclSelect **definition** *OclSelect* :: $[(\mathcal{A}, \alpha :: \text{null}) \text{ Set}, (\mathcal{A}, \alpha) \text{ val} \Rightarrow (\mathcal{A}) \text{ Boolean}] \Rightarrow (\mathcal{A}, \alpha) \text{ Set}$

where $\text{OclSelect } S \ P = (\lambda \tau. \text{if } (\delta \ S) \ \tau = \text{true} \ \tau$
 $\text{then if } (\exists x \in \llbracket \text{Rep-Set}_{\text{base}}(S \ \tau) \rrbracket. P(\lambda -. x) \ \tau = \text{invalid } \tau)$
 $\text{then invalid } \tau$
 $\text{else Abs-Set}_{\text{base}} \llbracket \{x \in \llbracket \text{Rep-Set}_{\text{base}}(S \ \tau) \rrbracket. P(\lambda -. x) \ \tau \neq \text{false } \tau\} \rrbracket$
 $\text{else invalid } \tau)$

syntax

$\text{-OclSelect} :: [(\mathcal{A}, \alpha :: \text{null}) \text{ Set}, \text{id}, (\mathcal{A}) \text{ Boolean}] \Rightarrow \mathcal{A} \text{ Boolean} \quad ((-) \rightarrow \text{select}'(-|-'))$

translations

$$X \rightarrow \text{select}(x \mid P) == \text{CONST OclSelect } X \ (\%x. P)$$

Definition: OclReject **definition** *OclReject* :: $[(\mathcal{A}, \alpha :: \text{null}) \text{ Set}, (\mathcal{A}, \alpha) \text{ val} \Rightarrow (\mathcal{A}) \text{ Boolean}] \Rightarrow (\mathcal{A}, \alpha :: \text{null}) \text{ Set}$

where $\text{OclReject } S \ P = \text{OclSelect } S \ (\text{not } o \ P)$

syntax

$\text{-OclReject} :: [(\mathcal{A}, \alpha :: \text{null}) \text{ Set}, \text{id}, (\mathcal{A}) \text{ Boolean}] \Rightarrow \mathcal{A} \text{ Boolean} \quad ((-) \rightarrow \text{reject}'(-|-'))$

translations

$$X \rightarrow \text{reject}(x \mid P) == \text{CONST OclReject } X \ (\%x. P)$$
Definition (futor operators) consts

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

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

OclIncludesAll :: $[(\mathcal{A}, \alpha :: \text{null}) \text{ Set}, (\mathcal{A}, \alpha) \text{ Set}] \Rightarrow \mathcal{A} \text{ Boolean}$

OclExcludesAll :: $[(\mathcal{A}, \alpha :: \text{null}) \text{ Set}, (\mathcal{A}, \alpha) \text{ Set}] \Rightarrow \mathcal{A} \text{ Boolean}$

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

notation

$OclCount \quad (\dashrightarrow count'(-))$

notation

$OclSum \quad (\dashrightarrow sum'(-))$

notation

$OclIncludesAll \quad (\dashrightarrow includesAll'(-))$

notation

$OclExcludesAll \quad (\dashrightarrow excludesAll'(-))$

notation

$OclComplement \quad (\dashrightarrow complement'(-))$

notation

$OclUnion \quad (\dashrightarrow union'(-))$

notation

$OclIntersection \quad (\dashrightarrow intersection'(-))$

Validity and Definedness Properties $OclIncluding$

lemma $OclIncluding-defined-args-valid$:

$(\tau \models \delta(X \dashrightarrow including(x))) = ((\tau \models (\delta X)) \wedge (\tau \models (v x)))$

by $(simp \text{ add: } foundation10')$

lemma $OclIncluding-valid-args-valid$:

$(\tau \models v(X \dashrightarrow including(x))) = ((\tau \models (\delta X)) \wedge (\tau \models (v x)))$

by $(metis (hide-lams, no-types) OclIncluding.def-valid-then-def OclIncluding-defined-args-valid)$

lemma $OclIncluding-defined-args-valid[simp, code-unfold]$:

$\delta(X \dashrightarrow including(x)) = ((\delta X) \text{ and } (v x))$

by $simp$

lemma $OclIncluding-valid-args-valid''[simp, code-unfold]$:

$v(X \dashrightarrow including(x)) = ((\delta X) \text{ and } (v x))$

by $(auto \text{ intro! : } transform2\text{-rev } simp : OclIncluding-valid-args-valid \text{ foundation10 defined-and-I})$

$OclExcluding$

lemma $OclExcluding-defined-args-valid$:

$(\tau \models \delta(X \dashrightarrow excluding(x))) = ((\tau \models (\delta X)) \wedge (\tau \models (v x)))$

proof –

have $A : \perp \in \{X. X = bot \vee X = null \vee (\forall x \in \llbracket X \rrbracket. x \neq bot)\}$ **by** $(simp \text{ add: } bot\text{-option-def})$

have $B : \llbracket \perp \rrbracket \in \{X. X = bot \vee X = null \vee (\forall x \in \llbracket X \rrbracket. x \neq bot)\}$

by $(simp \text{ add: } null\text{-option-def } bot\text{-option-def})$

have $C : (\tau \models (\delta X)) \implies (\tau \models (v x)) \implies$

$\llbracket \llbracket Rep\text{-Set}_{base} (X \tau) \rrbracket - \{x \tau\} \rrbracket \in \{X. X = bot \vee X = null \vee (\forall x \in \llbracket X \rrbracket. x \neq bot)\}$

```

by(frule Set-inv-lemma, simp add: foundation18 invalid-def)
have D: ( $\tau \models \delta(X \rightarrow \text{excluding}(x)) \implies ((\tau \models (\delta X)) \wedge (\tau \models (\nu x)))$ )
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: ( $\tau \models (\delta X) \implies (\tau \models (\nu x)) \implies (\tau \models \delta(X \rightarrow \text{excluding}(x)))$ )
apply(subst OclExcluding-def, subst OclValid-def, subst defined-def)
apply(auto simp: OclValid-def null-Setbase-def bot-Setbase-def null-fun-def bot-fun-def)
apply(frule Abs-Setbase-inject[OF C A, simplified OclValid-def, THEN iffD1],
  simp-all add: bot-option-def)
apply(frule Abs-Setbase-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:
( $\tau \models \nu(X \rightarrow \text{excluding}(x)) = ((\tau \models (\delta X)) \wedge (\tau \models (\nu x)))$ )
proof –
have D: ( $\tau \models \nu(X \rightarrow \text{excluding}(x)) \implies ((\tau \models (\delta X)) \wedge (\tau \models (\nu x)))$ )
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: ( $\tau \models (\delta X) \implies (\tau \models (\nu x)) \implies (\tau \models \nu(X \rightarrow \text{excluding}(x)))$ )
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]:
 $\delta(X \rightarrow \text{excluding}(x)) = ((\delta X) \text{ and } (\nu x))$ 
by(auto intro!: transform2-rev simp:OclExcluding-defined-args-valid foundation10 defined-and-I)

```

```

lemma OclExcluding-valid-args-valid''[simp,code-unfold]:
 $\nu(X \rightarrow \text{excluding}(x)) = ((\delta X) \text{ and } (\nu x))$ 
by(auto intro!: transform2-rev simp:OclExcluding-valid-args-valid foundation10 defined-and-I)

```

OclIncludes

```

lemma OclIncludes-defined-args-valid:
( $\tau \models \delta(X \rightarrow \text{includes}(x)) = ((\tau \models (\delta X)) \wedge (\tau \models (\nu x)))$ )
proof –
have A: ( $\tau \models \delta(X \rightarrow \text{includes}(x)) \implies ((\tau \models (\delta X)) \wedge (\tau \models (\nu x)))$ )
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: ( $\tau \models (\delta X) \implies (\tau \models (\nu x)) \implies (\tau \models \delta(X \rightarrow \text{includes}(x)))$ )

```

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:*

$(\tau \models v(X \rightarrow \text{includes}(x))) = ((\tau \models (\delta X)) \wedge (\tau \models (v x)))$

proof –

have A: $(\tau \models v(X \rightarrow \text{includes}(x))) \implies ((\tau \models (\delta X)) \wedge (\tau \models (v x)))$
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: $(\tau \models (\delta X)) \implies (\tau \models (v x)) \implies (\tau \models v(X \rightarrow \text{includes}(x)))$
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]:*

$\delta(X \rightarrow \text{includes}(x)) = ((\delta X) \text{ and } (v x))$

by(*auto intro!: transform2-rev simp:OclIncludes-defined-args-valid foundation10 defined-and-I*)

lemma *OclIncludes-valid-args-valid''[simp,code-unfold]:*

$v(X \rightarrow \text{includes}(x)) = ((\delta X) \text{ and } (v x))$

by(*auto intro!: transform2-rev simp:OclIncludes-valid-args-valid foundation10 defined-and-I*)

OclExcludes

lemma *OclExcludes-defined-args-valid:*

$(\tau \models \delta(X \rightarrow \text{excludes}(x))) = ((\tau \models (\delta X)) \wedge (\tau \models (v x)))$

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:*

$(\tau \models v(X \rightarrow \text{excludes}(x))) = ((\tau \models (\delta X)) \wedge (\tau \models (v x)))$

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]:*

$\delta(X \rightarrow \text{excludes}(x)) = ((\delta X) \text{ and } (v x))$

by(*auto intro!: transform2-rev simp:OclExcludes-defined-args-valid foundation10 defined-and-I*)

lemma *OclExcludes-valid-args-valid''[simp,code-unfold]:*

$v(X \rightarrow \text{excludes}(x)) = ((\delta X) \text{ and } (v x))$

by(*auto intro!: transform2-rev simp:OclExcludes-valid-args-valid foundation10 defined-and-I*)

OclSize

lemma *OclSize-defined-args-valid*: $\tau \models \delta (X \rightarrow \text{size}()) \implies \tau \models \delta X$
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*: $\tau \models \text{not}(\delta(S \rightarrow \text{size}()))$
shows $(\tau \models \text{not}(\delta(S))) \vee \neg \text{finite } [[\text{Rep-Set}_{\text{base}} (S \ \tau)]]$
apply (*insert non-finite, simp*)
apply (*rule impI*)
apply (*simp add: OclSize-def OclValid-def defined-def*)
apply (*case-tac finite [[Rep-Set_{base} (S τ)]]*,
simp-all add: null-fun-def null-option-def bot-fun-def bot-option-def)
done

lemma $\tau \models \delta X \implies \neg \text{finite } [[\text{Rep-Set}_{\text{base}} (X \ \tau)]] \implies \neg \tau \models \delta (X \rightarrow \text{size}())$
by (*simp add: OclSize-def OclValid-def defined-def bot-fun-def false-def true-def*)

lemma *size-defined*:
assumes *X-finite*: $\bigwedge \tau. \text{finite } [[\text{Rep-Set}_{\text{base}} (X \ \tau)]]$
shows $\delta (X \rightarrow \text{size}()) = \delta X$
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*: $\text{finite } [[\text{Rep-Set}_{\text{base}} (X \ \tau)]]$
shows $(\tau \models \delta (X \rightarrow \text{size}())) = (\tau \models \delta X)$
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*: $\tau \models \delta (X \rightarrow \text{isEmpty}()) \implies \tau \models \text{v } X$
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()) ≐ 0*) τ , *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 StrictRefEqInteger valid-def*)
by (*metis (hide-lams, no-types)*
bot-fun-def OclValid-def defined-def foundation2 invalid-def)

lemma $\tau \models \delta (\text{null} \rightarrow \text{isEmpty}())$
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*: $\tau \models \delta X \implies \neg \text{finite } \llbracket \text{Rep-Set}_{\text{base}}(X \ \tau) \rrbracket \implies \neg \tau \models \delta (X \rightarrow \text{isEmpty}())$
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* ($X \rightarrow \text{size}() \doteq 0$) τ , *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 StrictRefEqInteger valid-def bot-fun-def false-def true-def invalid-def*)
OclNotEmpty

lemma *OclNotEmpty-defined-args-valid*: $\tau \models \delta (X \rightarrow \text{notEmpty}()) \implies \tau \models \nu X$
by (*metis* (*hide-lams, no-types*) *OclNotEmpty-def OclNot-defargs OclNot-not foundation6 foundation9*
OclIsEmpty-defined-args-valid)

lemma $\tau \models \delta (\text{null} \rightarrow \text{notEmpty}())$
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*: $\tau \models \delta X \implies \neg \text{finite } \llbracket \text{Rep-Set}_{\text{base}}(X \ \tau) \rrbracket \implies \neg \tau \models \delta (X \rightarrow \text{notEmpty}())$
apply(*simp add*: *OclNotEmpty-def*)
apply(*drule OclIsEmpty-infinite, simp*)
by (*metis OclNot-defargs OclNot-not foundation6 foundation9*)

lemma *OclNotEmpty-has-elt* : $\tau \models \delta X \implies$
 $\tau \models X \rightarrow \text{notEmpty}() \implies$
 $\exists e. e \in \llbracket \text{Rep-Set}_{\text{base}}(X \ \tau) \rrbracket$
apply(*simp add*: *OclNotEmpty-def OclIsEmpty-def deMorgan1 deMorgan2, drule foundation5*)
apply(*subst* (*asm*) (2) *OclNot-def*,
simp add: *OclValid-def StrictRefEqInteger 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 equalsOI*)
OclANY

lemma *OclANY-defined-args-valid*: $\tau \models \delta (X \rightarrow \text{any}()) \implies \tau \models \delta X$
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 $\tau \models \delta X \implies \tau \models X \rightarrow \text{isEmpty}() \implies \neg \tau \models \delta (X \rightarrow \text{any}())$
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*:
 $(\tau \models v(X \rightarrow \text{any}())) = (\tau \models v X)$
proof –
have A: $(\tau \models v(X \rightarrow \text{any}())) \implies ((\tau \models (v X)))$
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: $(\tau \models (v X)) \implies (\tau \models v(X \rightarrow \text{any}()))$
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 (δX) $\tau = \text{true } \tau$)*

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 $Q = \lambda x. x \neq \perp$ and $P = \lambda y. y \in \llbracket \text{Rep-Set}_{\text{base}}(X \tau) \rrbracket$],
simp)
show ?thesis **by**(*auto dest:A intro:B*)
qed

lemma *OclANY-valid-args-valid''[simp,code-unfold]*:
 $v(X \rightarrow \text{any}()) = (v X)$
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]*: $(\text{invalid} \rightarrow \text{including}(x)) = \text{invalid}$
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]*: $(X \rightarrow \text{including}(\text{invalid})) = \text{invalid}$
by(*simp add: OclIncluding-def invalid-def bot-fun-def defined-def valid-def false-def true-def*)

lemma *OclIncluding-null[simp,code-unfold]*: $(\text{null} \rightarrow \text{including}(x)) = \text{invalid}$
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]*: $(\text{invalid} \rightarrow \text{excluding}(x)) = \text{invalid}$
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]*: $(X \rightarrow \text{excluding}(\text{invalid})) = \text{invalid}$
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: $\tau \models \text{not}(\delta(S \rightarrow \text{size}()))$

shows (OclIterate S A F) $\tau = \text{invalid } \tau$

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-OclIncludesI*:
(*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 *StrictRefEqInteger.cp0*)
by(simp add: *cp-defined*[symmetric] *cp-valid*[symmetric] *StrictRefEqInteger.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*: $X \rightarrow \text{any}()$ $\tau = ((\lambda -. X \tau) \rightarrow \text{any}()) \tau$
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*:
 $(S \rightarrow \text{forall}(x \mid P x)) \tau = ((\lambda -. S \tau) \rightarrow \text{forall}(x \mid P (\lambda -. x \tau))) \tau$
by(*simp add: OclForall-def cp-defined[symmetric]*)

lemma *cp-OclForallI* [*simp,intro!*]:
 $cp S \implies cp (\lambda X. ((S X) \rightarrow \text{forall}(x \mid P x)))$
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
 $cp (\lambda X St x. P (\lambda \tau. x) X St) \implies cp S \implies cp (\lambda X. (S X) \rightarrow \text{forall}(x \mid P x X))$
apply(*simp only: cp-def*)
oops

lemma
 $cp S \implies$
 $(\bigwedge x. cp(P x)) \implies$
 $cp(\lambda X. ((S X) \rightarrow \text{forall}(x \mid P x X)))$
oops

lemma *cp-OclExists*:
 $(S \rightarrow \text{exists}(x \mid P x)) \tau = ((\lambda -. S \tau) \rightarrow \text{exists}(x \mid P (\lambda -. x \tau))) \tau$
by(*simp add: OclExists-def OclNot-def, subst cp-OclForall, simp*)

lemma *cp-OclExistsI* [*simp,intro!*]:
 $cp S \implies cp (\lambda X. ((S X) \rightarrow \text{exists}(x \mid P x)))$
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*: $(X \rightarrow \text{iterate}(a; x = A \mid P a x)) \tau =$
 $((\lambda -. X \tau) \rightarrow \text{iterate}(a; x = A \mid P a x)) \tau$
by(*simp add: OclIterate-def cp-defined[symmetric]*)

lemma *cp-OclSelect*: $(X \rightarrow \text{select}(a \mid P \ a)) \ \tau =$
 $((\lambda \ -. \ X \ \tau) \rightarrow \text{select}(a \mid P \ a)) \ \tau$
by (*simp add: OclSelect-def cp-defined[symmetric]*)

lemma *cp-OclReject*: $(X \rightarrow \text{reject}(a \mid P \ a)) \ \tau =$
 $((\lambda \ -. \ X \ \tau) \rightarrow \text{reject}(a \mid P \ a)) \ \tau$
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* \rightarrow *including*(*x*))

proof –

have *A*: $\bigwedge \tau \ \tau'. \neg (\tau \models v \ x) \implies (S \rightarrow \text{including}(x) \ \tau) = (S \rightarrow \text{including}(x) \ \tau')$

apply (*simp add: foundationI8*)

apply (*erule const-subst[OF const-x const-invalid], simp-all*)

by (*rule const-charn[OF const-invalid]*)

have *B*: $\bigwedge \tau \ \tau'. \neg (\tau \models \delta \ S) \implies (S \rightarrow \text{including}(x) \ \tau) = (S \rightarrow \text{including}(x) \ \tau')$

apply (*simp add: foundationI6', 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 $\tau \ \tau'$*)

apply (*case-tac $\neg (\tau \models v \ x)$, simp add: A*)

apply (*case-tac $\neg (\tau \models \delta \ S)$, simp-all add: B*)

apply (*frule-tac $\tau'1 = \tau'$ in const-OclValid2[OF const-x, THEN iffD1]*)

apply (*frule-tac $\tau'1 = \tau'$ 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 StrictRefEqSet :
  (x::('A, 'α::null)Set) ≐ y ≡ λ τ. if (v x) τ = true ∧ (v y) τ = true
    then (x ≐ y) τ
    else invalid τ

```

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 be 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 StrictRefEqSet : profile-bin3 λ x y. (x::('A, 'α::null)Set) ≐ y
  by unfold-locales (auto simp: StrictRefEqSet)

```

Execution Rules on OclIncluding **lemma** *OclIncluding-finite-rep-set* :

```

assumes X-def : τ ⊨ δ X
and x-val : τ ⊨ v x
shows finite [[Rep-Setbase (X->including(x) τ)]] = finite [[Rep-Setbase (X τ)]]
proof –
have C : [[insert (x τ) [[Rep-Setbase (X τ)]]]] ∈ {X. X = bot ∨ X = null ∨ (∀x∈[[X]]. x ≠ 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 Abs-Setbase-inverse[OF C]
  dest: foundation13[THEN iffD2, THEN foundation22[THEN iffD1]])
qed

```

lemma *OclIncluding-rep-set*:

```

assumes S-def : τ ⊨ δ S
shows [[Rep-Setbase (S->including(λ-. [[x]]) τ)]] = insert [[x]] [[Rep-Setbase (S τ)]]
apply (simp add: OclIncluding-def S-def[simplified OclValid-def])
apply (subst Abs-Setbase-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 : τ ⊨ δ X
and a-val : τ ⊨ v a
shows [[Rep-Setbase (X->including(a) τ)]] ≠ {}
apply (simp add: OclIncluding-def X-def[simplified OclValid-def] a-val[simplified OclValid-def])
apply (subst Abs-Setbase-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 $\tau \models X \rightarrow \text{includes}(x)$

shows $X \rightarrow \text{including}(x) \ \tau = X \ \tau$

proof –

have includes-def: $\tau \models X \rightarrow \text{includes}(x) \implies \tau \models \delta X$

by (metis bot-fun-def OclIncludes-def OclValid-def defined3 foundation16)

have includes-val: $\tau \models X \rightarrow \text{includes}(x) \implies \tau \models v \ x$

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 $\tau \models X \rightarrow \text{includes}(x)$

shows $\tau \models X \rightarrow \text{including}(x) \triangleq X$

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 : $\tau \models v \ i$

and j-val : $\tau \models v \ j$

shows $\tau \models ((S :: ('a, 'a::\text{null}) \text{Set}) \rightarrow \text{including}(i) \rightarrow \text{including}(j) \triangleq (S \rightarrow \text{including}(j) \rightarrow \text{including}(i)))$

proof –

have A : $\llbracket \text{insert } (i \ \tau) \llbracket \text{Rep-Set}_{\text{base}}(S \ \tau) \rrbracket \rrbracket \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$

by(insert S-def i-val, frule Set-inv-lemma, simp add: foundation18 invalid-def)

have B : $\llbracket \text{insert } (j \ \tau) \llbracket \text{Rep-Set}_{\text{base}}(S \ \tau) \rrbracket \rrbracket \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$

by(insert S-def j-val, frule Set-inv-lemma, simp add: foundation18 invalid-def)

have G1 : $\text{Abs-Set}_{\text{base}} \llbracket \text{insert } (i \ \tau) \llbracket \text{Rep-Set}_{\text{base}}(S \ \tau) \rrbracket \rrbracket \neq \text{Abs-Set}_{\text{base}} \text{None}$

by(insert A, simp add: Abs-Set_{base}-inject bot-option-def null-option-def)

have G2 : $\text{Abs-Set}_{\text{base}} \llbracket \text{insert } (i \ \tau) \llbracket \text{Rep-Set}_{\text{base}}(S \ \tau) \rrbracket \rrbracket \neq \text{Abs-Set}_{\text{base}} [\text{None}]$

by(insert A, simp add: Abs-Set_{base}-inject bot-option-def null-option-def)

have G3 : $\text{Abs-Set}_{\text{base}} \llbracket \text{insert } (j \ \tau) \llbracket \text{Rep-Set}_{\text{base}}(S \ \tau) \rrbracket \rrbracket \neq \text{Abs-Set}_{\text{base}} \text{None}$

by(insert B, simp add: Abs-Set_{base}-inject bot-option-def null-option-def)

have G4 : $\text{Abs-Set}_{\text{base}} \llbracket \text{insert } (j \ \tau) \llbracket \text{Rep-Set}_{\text{base}}(S \ \tau) \rrbracket \rrbracket \neq \text{Abs-Set}_{\text{base}} [\text{None}]$

by(insert B, simp add: Abs-Set_{base}-inject bot-option-def null-option-def)

have * : $(\delta (\lambda \cdot. \text{Abs-Set}_{\text{base}} \llbracket \text{insert } (i \ \tau) \llbracket \text{Rep-Set}_{\text{base}}(S \ \tau) \rrbracket \rrbracket)) \ \tau = \llbracket \text{True} \rrbracket$

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 ** : (δ (λ -. *Abs-Set_{base}* \llbracket insert (j τ) \llbracket Rep-Set_{base} (S τ) $\rrbracket\rrbracket$)) τ = \llbracket True \rrbracket)
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-Set_{base}* \llbracket insert(j τ) \llbracket Rep-Set_{base}(*Abs-Set_{base}* \llbracket insert(i τ) \llbracket Rep-Set_{base}(S τ) $\rrbracket\rrbracket$) $\rrbracket\rrbracket$ =
Abs-Set_{base} \llbracket insert(i τ) \llbracket Rep-Set_{base}(*Abs-Set_{base}* \llbracket insert(j τ) \llbracket Rep-Set_{base}(S τ) $\rrbracket\rrbracket$) $\rrbracket\rrbracket$
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]*:

(($S :: ('a, 'a::null) Set$) \rightarrow including(i) \rightarrow including(j) = ($S \rightarrow$ including(j) \rightarrow including(i)))

proof –

have A: $\bigwedge \tau. \tau \models (i \triangleq \text{invalid}) \implies (S \rightarrow \text{including}(i) \rightarrow \text{including}(j)) \tau = \text{invalid } \tau$
apply(*rule foundation22[THEN iffD1]*)
by(*erule StrongEq-L-subst2-rev, simp,simp*)
have A': $\bigwedge \tau. \tau \models (i \triangleq \text{invalid}) \implies (S \rightarrow \text{including}(j) \rightarrow \text{including}(i)) \tau = \text{invalid } \tau$
apply(*rule foundation22[THEN iffD1]*)
by(*erule StrongEq-L-subst2-rev, simp,simp*)
have B: $\bigwedge \tau. \tau \models (j \triangleq \text{invalid}) \implies (S \rightarrow \text{including}(i) \rightarrow \text{including}(j)) \tau = \text{invalid } \tau$
apply(*rule foundation22[THEN iffD1]*)
by(*erule StrongEq-L-subst2-rev, simp,simp*)
have B': $\bigwedge \tau. \tau \models (j \triangleq \text{invalid}) \implies (S \rightarrow \text{including}(j) \rightarrow \text{including}(i)) \tau = \text{invalid } \tau$
apply(*rule foundation22[THEN iffD1]*)
by(*erule StrongEq-L-subst2-rev, simp,simp*)
have C: $\bigwedge \tau. \tau \models (S \triangleq \text{invalid}) \implies (S \rightarrow \text{including}(i) \rightarrow \text{including}(j)) \tau = \text{invalid } \tau$

```

    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
have C':  $\wedge \tau. \tau \models (S \triangleq \text{invalid}) \implies (S \rightarrow \text{including}(j) \rightarrow \text{including}(i)) \tau = \text{invalid} \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
have D:  $\wedge \tau. \tau \models (S \triangleq \text{null}) \implies (S \rightarrow \text{including}(i) \rightarrow \text{including}(j)) \tau = \text{invalid} \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
have D':  $\wedge \tau. \tau \models (S \triangleq \text{null}) \implies (S \rightarrow \text{including}(j) \rightarrow \text{including}(i)) \tau = \text{invalid} \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
show ?thesis
  apply(rule ext, rename-tac  $\tau$ )
  apply(case-tac  $\tau \models (v \ i)$ )
  apply(case-tac  $\tau \models (v \ j)$ )
  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 :  $\tau \models v \ x$ 
shows finite  $\llbracket \llbracket \text{Rep-Set}_{\text{base}} (X \rightarrow \text{excluding}(x) \ \tau) \rrbracket \rrbracket = \text{finite} \llbracket \llbracket \text{Rep-Set}_{\text{base}} (X \ \tau) \rrbracket \rrbracket$ 
proof -
have C :  $\llbracket \llbracket \llbracket \text{Rep-Set}_{\text{base}} (X \ \tau) \rrbracket \rrbracket - \{x \ \tau\} \rrbracket \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$ 
  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-Setbase-inverse[OF C]
  dest: foundation13[THEN iffD2, THEN foundation22[THEN iffD1]])
qed

```

lemma *OclExcluding-rep-set*:

```

assumes S-def:  $\tau \models \delta \ S$ 
shows  $\llbracket \llbracket \text{Rep-Set}_{\text{base}} (S \rightarrow \text{excluding}(\lambda -. \llbracket x \rrbracket) \ \tau) \rrbracket \rrbracket = \llbracket \llbracket \text{Rep-Set}_{\text{base}} (S \ \tau) \rrbracket \rrbracket - \{\llbracket x \rrbracket\}$ 
apply(simp add: OclExcluding-def S-def[simplified OclValid-def])
apply(subst Abs-Setbase-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 $\tau \models X \rightarrow \text{excludes}(x)$
shows $X \rightarrow \text{excluding}(x) \ \tau = X \ \tau$
proof –
have *excludes-def*: $\tau \models X \rightarrow \text{excludes}(x) \implies \tau \models \delta \ X$
by (*metis* (*hide-lams*, *no-types*) *OclExcludes-defined-args-valid foundation6*)

have *excludes-val*: $\tau \models X \rightarrow \text{excludes}(x) \implies \tau \models v \ x$
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 $\tau \models X \rightarrow \text{excludes}(x)$
shows $\tau \models X \rightarrow \text{excluding}(x) \triangleq X$
by (*simp add: StrongEq-def OclValid-def true-def OclExcluding-excludes0* [*OF assms*])

lemma *OclExcluding-cha0* [*simp*]:
assumes *val-x*: $\tau \models (v \ x)$
shows $\tau \models ((\text{Set}\{\} \rightarrow \text{excluding}(x)) \triangleq \text{Set}\{\})$
proof –
have *A* : $[None] \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in [\![X]\!]. x \neq \text{bot})\}$
by (*simp add: null-option-def bot-option-def*)
have *B* : $[\![\{\}]\!] \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in [\![X]\!]. x \neq \text{bot})\}$ **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* : $\tau \models \delta \ S$
and *i-val* : $\tau \models v \ i$
and *j-val* : $\tau \models v \ j$
shows $\tau \models ((S :: ('a, 'a::\text{null}) \text{Set}) \rightarrow \text{excluding}(i) \rightarrow \text{excluding}(j) \triangleq (S \rightarrow \text{excluding}(j) \rightarrow \text{excluding}(i)))$
proof –
have *A* : $[\![\![\text{Rep-Set}_{\text{base}}(S \ \tau)]\!] - \{i \ \tau\}]\!] \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in [\![X]\!]. x \neq \text{bot})\}$
by (*insert S-def i-val, frule Set-inv-lemma, simp add: foundation18 invalid-def*)
have *B* : $[\![\![\text{Rep-Set}_{\text{base}}(S \ \tau)]\!] - \{j \ \tau\}]\!] \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in [\![X]\!]. x \neq \text{bot})\}$
by (*insert S-def j-val, frule Set-inv-lemma, simp add: foundation18 invalid-def*)

```

have G1 : Abs-Setbase [[[[Rep-Setbase (S τ)] - {i τ}]]] ≠ Abs-Setbase None
  by(insert A, simp add: Abs-Setbase-inject bot-option-def null-option-def)
have G2 : Abs-Setbase [[[[Rep-Setbase (S τ)] - {i τ}]]] ≠ Abs-Setbase [None]
  by(insert A, simp add: Abs-Setbase-inject bot-option-def null-option-def)
have G3 : Abs-Setbase [[[[Rep-Setbase (S τ)] - {j τ}]]] ≠ Abs-Setbase None
  by(insert B, simp add: Abs-Setbase-inject bot-option-def null-option-def)
have G4 : Abs-Setbase [[[[Rep-Setbase (S τ)] - {j τ}]]] ≠ Abs-Setbase [None]
  by(insert B, simp add: Abs-Setbase-inject bot-option-def null-option-def)

have * : (δ (λ-. Abs-Setbase [[[[Rep-Setbase (S τ)] - {i τ}]]]) τ = [[True]])
  by(auto simp: OclValid-def false-def defined-def null-fun-def true-def
    bot-fun-def bot-Setbase-def null-Setbase-def S-def i-val G1 G2)

have ** : (δ (λ-. Abs-Setbase [[[[Rep-Setbase (S τ)] - {j τ}]]]) τ = [[True]])
  by(auto simp: OclValid-def false-def defined-def null-fun-def true-def
    bot-fun-def bot-Setbase-def null-Setbase-def S-def i-val G3 G4)

have *** : Abs-Setbase [[[[Rep-Setbase (Abs-Setbase [[[[Rep-Setbase (S τ)] - {i τ}]]]) - {j τ}]]] =
  Abs-Setbase [[[[Rep-Setbase (Abs-Setbase [[[[Rep-Setbase (S τ)] - {j τ}]]]) - {i τ}]]] =
  apply(simp add: Abs-Setbase-inverse[OF A] Abs-Setbase-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 *)
  done
qed

```

lemma OclExcluding-commute[simp,code-unfold]:

((S :: ('a, 'a::null) Set) -> excluding(i) -> excluding(j)) = (S -> excluding(j) -> excluding(i))

proof -

have A: $\bigwedge \tau. \tau \models i \triangleq \text{invalid} \implies (S \rightarrow \text{excluding}(i) \rightarrow \text{excluding}(j)) \tau = \text{invalid } \tau$

```

    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
have A':  $\wedge \tau. \tau \models i \triangleq \text{invalid} \implies (S \rightarrow \text{excluding}(j) \rightarrow \text{excluding}(i)) \tau = \text{invalid} \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
have B:  $\wedge \tau. \tau \models j \triangleq \text{invalid} \implies (S \rightarrow \text{excluding}(i) \rightarrow \text{excluding}(j)) \tau = \text{invalid} \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
have B':  $\wedge \tau. \tau \models j \triangleq \text{invalid} \implies (S \rightarrow \text{excluding}(j) \rightarrow \text{excluding}(i)) \tau = \text{invalid} \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
have C:  $\wedge \tau. \tau \models S \triangleq \text{invalid} \implies (S \rightarrow \text{excluding}(i) \rightarrow \text{excluding}(j)) \tau = \text{invalid} \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
have C':  $\wedge \tau. \tau \models S \triangleq \text{invalid} \implies (S \rightarrow \text{excluding}(j) \rightarrow \text{excluding}(i)) \tau = \text{invalid} \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
have D:  $\wedge \tau. \tau \models S \triangleq \text{null} \implies (S \rightarrow \text{excluding}(i) \rightarrow \text{excluding}(j)) \tau = \text{invalid} \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
have D':  $\wedge \tau. \tau \models S \triangleq \text{null} \implies (S \rightarrow \text{excluding}(j) \rightarrow \text{excluding}(i)) \tau = \text{invalid} \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
show ?thesis
    apply(rule ext, rename-tac  $\tau$ )
    apply(case-tac  $\tau \models (v i)$ )
    apply(case-tac  $\tau \models (v j)$ )
    apply(case-tac  $\tau \models (\delta S)$ )
    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-cha0-exec*[simp,code-unfold]:

$(\text{Set}\{\} \rightarrow \text{excluding}(x)) = (\text{if } (v x) \text{ then } \text{Set}\{\} \text{ else } \text{invalid } \text{endif})$

proof –

```

have A:  $\wedge \tau. (\text{Set}\{\} \rightarrow \text{excluding}(\text{invalid})) \tau = (\text{if } (v \text{ invalid}) \text{ then } \text{Set}\{\} \text{ else } \text{invalid } \text{endif}) \tau$ 
    by simp
have B:  $\wedge \tau x. \tau \models (v x) \implies$ 
     $(\text{Set}\{\} \rightarrow \text{excluding}(x)) \tau = (\text{if } (v x) \text{ then } \text{Set}\{\} \text{ else } \text{invalid } \text{endif}) \tau$ 
    by(simp add: OclExcluding-cha0[THEN foundation22[THEN iffD1]])
show ?thesis
    apply(rule ext, rename-tac  $\tau$ )

```

```

apply(case-tac  $\tau \models (v\ x)$ )
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: $\tau \models (\delta\ X)$ 
and val-x: $\tau \models (v\ x)$ 
and val-y: $\tau \models (v\ y)$ 
and neq : $\tau \models \text{not}(x \triangleq y)$ 
shows  $\tau \models ((X \rightarrow \text{including}(x)) \rightarrow \text{excluding}(y)) \triangleq ((X \rightarrow \text{excluding}(y)) \rightarrow \text{including}(x))$ 
proof –
have C :  $[[\text{insert } (x\ \tau) \text{ } \llbracket \text{Rep-Set}_{base} (X\ \tau) \rrbracket]] \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$ 
by(insert def-X val-x, frule Set-inv-lemma, simp add: foundation18 invalid-def)
have D :  $[[\llbracket \text{Rep-Set}_{base} (X\ \tau) \rrbracket] - \{y\ \tau\}] \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$ 
by(insert def-X val-x, frule Set-inv-lemma, simp add: foundation18 invalid-def)
have E :  $x\ \tau \neq y\ \tau$ 
by(insert neq,
auto simp: OclValid-def bot-fun-def OclIncluding-def OclIncludes-def
false-def true-def defined-def valid-def bot-Setbase-def
null-fun-def null-Setbase-def StrongEq-def OclNot-def)

have G1 : Abs-Setbase  $[[\text{insert } (x\ \tau) \text{ } \llbracket \text{Rep-Set}_{base} (X\ \tau) \rrbracket]] \neq \text{Abs-Set}_{base}\ \text{None}$ 
by(insert C, simp add: Abs-Setbase-inject bot-option-def null-option-def)
have G2 : Abs-Setbase  $[[\llbracket \text{Rep-Set}_{base} (X\ \tau) \rrbracket] - \{y\ \tau\}] \neq \text{Abs-Set}_{base}\ \text{None}$ 
by(insert C, simp add: Abs-Setbase-inject bot-option-def null-option-def)
have G :  $(\delta\ (\lambda \cdot \text{Abs-Set}_{base}\ [[\text{insert } (x\ \tau) \text{ } \llbracket \text{Rep-Set}_{base} (X\ \tau) \rrbracket]]))\ \tau = \text{true}\ \tau$ 
by(auto simp: OclValid-def false-def true-def defined-def
bot-fun-def bot-Setbase-def null-fun-def null-Setbase-def G1 G2)

have H1 : Abs-Setbase  $[[\llbracket \text{Rep-Set}_{base} (X\ \tau) \rrbracket] - \{y\ \tau\}] \neq \text{Abs-Set}_{base}\ \text{None}$ 
by(insert D, simp add: Abs-Setbase-inject bot-option-def null-option-def)
have H2 : Abs-Setbase  $[[\llbracket \text{Rep-Set}_{base} (X\ \tau) \rrbracket] - \{y\ \tau\}] \neq \text{Abs-Set}_{base}\ \text{None}$ 
by(insert D, simp add: Abs-Setbase-inject bot-option-def null-option-def)
have H :  $(\delta\ (\lambda \cdot \text{Abs-Set}_{base}\ [[\llbracket \text{Rep-Set}_{base} (X\ \tau) \rrbracket] - \{y\ \tau\}]))\ \tau = \text{true}\ \tau$ 
by(auto simp: OclValid-def false-def true-def defined-def
bot-fun-def bot-Setbase-def null-fun-def null-Setbase-def H1 H2)

have Z :  $\text{insert } (x\ \tau) \text{ } \llbracket \text{Rep-Set}_{base} (X\ \tau) \rrbracket - \{y\ \tau\} = \text{insert } (x\ \tau) \text{ } (\llbracket \text{Rep-Set}_{base} (X\ \tau) \rrbracket - \{y\ \tau\})$ 
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: $G\ H\ Abs\text{-}Set_{base}\text{-}inverse[OF\ C]\ Abs\text{-}Set_{base}\text{-}inverse[OF\ D]\ Z$)
done
qed

lemma *OclExcluding-charn2:*

assumes $def\text{-}X:\tau \models (\delta\ X)$

and $val\text{-}x:\tau \models (\vee\ x)$

shows $\tau \models (((X \rightarrow including(x)) \rightarrow excluding(x)) \triangleq (X \rightarrow excluding(x)))$

proof –

have $C : \llbracket insert\ (x\ \tau)\ \llbracket Rep\text{-}Set_{base}\ (X\ \tau) \rrbracket \rrbracket \in \{X.\ X = bot \vee X = null \vee (\forall x \in \llbracket X \rrbracket. x \neq bot)\}$

by(insert def-X val-x, frule Set-inv-lemma, simp add: foundation18 invalid-def)

have $G1 : Abs\text{-}Set_{base}\ \llbracket insert\ (x\ \tau)\ \llbracket Rep\text{-}Set_{base}\ (X\ \tau) \rrbracket \rrbracket \neq Abs\text{-}Set_{base}\ None$

by(insert C, simp add: Abs-Set_{base}-inject bot-option-def null-option-def)

have $G2 : Abs\text{-}Set_{base}\ \llbracket insert\ (x\ \tau)\ \llbracket Rep\text{-}Set_{base}\ (X\ \tau) \rrbracket \rrbracket \neq Abs\text{-}Set_{base}\ [None]$

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]])

$val\text{-}x[THEN foundation18[THEN iffD1,standard]]$

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:* $((X \rightarrow including(x)) \rightarrow excluding(x)) = (X \rightarrow excluding(x))$

proof –

have $A1 : \bigwedge \tau. \tau \models (X \triangleq invalid) \implies (X \rightarrow including(x)) \rightarrow excluding(x) \ \tau = invalid \ \tau$

apply(rule foundation22[THEN iffD1])

by(erule StrongEq-L-subst2-rev, simp, simp)

have $A1': \bigwedge \tau. \tau \models (X \triangleq invalid) \implies (X \rightarrow excluding(x)) \ \tau = invalid \ \tau$

apply(rule foundation22[THEN iffD1])

by(erule StrongEq-L-subst2-rev, simp, simp)

have $A2 : \bigwedge \tau. \tau \models (X \triangleq null) \implies (X \rightarrow including(x)) \rightarrow excluding(x) \ \tau = invalid \ \tau$

apply(rule foundation22[THEN iffD1])

by(erule StrongEq-L-subst2-rev, simp, simp)

have $A2': \bigwedge \tau. \tau \models (X \triangleq null) \implies (X \rightarrow excluding(x)) \ \tau = invalid \ \tau$

```

    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
have A3 :  $\bigwedge \tau. \tau \models (x \triangleq \text{invalid}) \implies (X \multimap \text{including}(x) \multimap \text{excluding}(x)) \tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
have A3':  $\bigwedge \tau. \tau \models (x \triangleq \text{invalid}) \implies (X \multimap \text{excluding}(x)) \tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)

show ?thesis
apply(rule ext, rename-tac  $\tau$ )
apply(case-tac  $\tau \models (v \ x)$ )
apply(case-tac  $\tau \models (\delta \ X)$ )
  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  $\multimap$  including(x :: (' $\mathcal{A}$ , 'a::null) val)  $\multimap$  excluding(y)) =
  (if  $\delta \ X$  then if x  $\doteq$  y
    then X  $\multimap$  excluding(y)
    else X  $\multimap$  excluding(y)  $\multimap$  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  $\doteq$  y) = invalid
and strict2: (x  $\doteq$  invalid) = invalid
and StrictRefEq-valid-args-valid:  $\bigwedge (x :: (' $\mathcal{A}$ , 'a::null) val) y \tau.
  ( $\tau \models \delta \ (x \doteq y)$ ) = (( $\tau \models (v \ x)$ )  $\wedge$  ( $\tau \models v \ y$ ))
and cp-StrictRefEq:  $\bigwedge (X :: (' $\mathcal{A}$ , 'a::null) val) Y \tau. (X  $\doteq$  Y)  $\tau$  = (( $\lambda \cdot. X \ \tau$ )  $\doteq$  ( $\lambda \cdot. Y \ \tau$ ))  $\tau$ 
and StrictRefEq-vs-StrongEq:  $\bigwedge (x :: (' $\mathcal{A}$ , 'a::null) val) y \tau.
   $\tau \models v \ x \implies \tau \models v \ y \implies (\tau \models ((x \doteq y) \triangleq (x \triangleq y)))$ 
shows (X  $\multimap$  including(x :: (' $\mathcal{A}$ , 'a::null) val)  $\multimap$  excluding(y)) =
  (if  $\delta \ X$  then if x  $\doteq$  y$$$ 
```



```

      then  $X \rightarrow \text{excluding}(y)$ 
      else  $X \rightarrow \text{excluding}(y) \rightarrow \text{including}(x)$ 
    endif
  else invalid endif)
proof –
  have A1:  $\bigwedge \tau. \tau \models (X \triangleq \text{invalid}) \implies$ 
    ( $X \rightarrow \text{including}(x) \rightarrow \text{includes}(y)$ )  $\tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp, simp)

  have B1:  $\bigwedge \tau. \tau \models (X \triangleq \text{null}) \implies$ 
    ( $X \rightarrow \text{including}(x) \rightarrow \text{includes}(y)$ )  $\tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp, simp)

  have A2:  $\bigwedge \tau. \tau \models (X \triangleq \text{invalid}) \implies X \rightarrow \text{including}(x) \rightarrow \text{excluding}(y) \tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp, simp)

  have B2:  $\bigwedge \tau. \tau \models (X \triangleq \text{null}) \implies X \rightarrow \text{including}(x) \rightarrow \text{excluding}(y) \tau = \text{invalid } \tau$ 
    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:  $\bigwedge \tau. \tau \models (x \triangleq \text{invalid}) \implies$ 
    ( $X \rightarrow \text{including}(x) \rightarrow \text{excluding}(y)$ )  $\tau =$ 
    (if  $x \doteq y$  then  $X \rightarrow \text{excluding}(y)$  else  $X \rightarrow \text{excluding}(y) \rightarrow \text{including}(x)$  endif)  $\tau$ 
    apply(rule foundation22[THEN iffD1])
    apply(erule StrongEq-L-subst2-rev, simp, simp)
    by(simp add: strict1)

  have D:  $\bigwedge \tau. \tau \models (y \triangleq \text{invalid}) \implies$ 
    ( $X \rightarrow \text{including}(x) \rightarrow \text{excluding}(y)$ )  $\tau =$ 
    (if  $x \doteq y$  then  $X \rightarrow \text{excluding}(y)$  else  $X \rightarrow \text{excluding}(y) \rightarrow \text{including}(x)$  endif)  $\tau$ 
    apply(rule foundation22[THEN iffD1])
    apply(erule StrongEq-L-subst2-rev, simp, simp)
    by (simp add: strict2)

  have E:  $\bigwedge \tau. \tau \models v x \implies \tau \models v y \implies$ 
    (if  $x \doteq y$  then  $X \rightarrow \text{excluding}(y)$  else  $X \rightarrow \text{excluding}(y) \rightarrow \text{including}(x)$  endif)  $\tau =$ 
    (if  $x \triangleq y$  then  $X \rightarrow \text{excluding}(y)$  else  $X \rightarrow \text{excluding}(y) \rightarrow \text{including}(x)$  endif)  $\tau$ 
    apply(subst cp-OclIf)
    apply(subst StrictRefEq-vs-StrongEq[THEN foundation22[THEN iffD1]])
    by(simp-all add: cp-OclIf[symmetric])

  have F:  $\bigwedge \tau. \tau \models \delta X \implies \tau \models v x \implies \tau \models (x \triangleq y) \implies$ 

```

```

 $(X \rightarrow \text{including}(x) \rightarrow \text{excluding}(y) \ \tau) = (X \rightarrow \text{excluding}(y) \ \tau)$ 
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  $\tau$ )
apply(case-tac  $\neg (\tau \models (\delta \ X))$ , simp add:defined-split,elim disjE A1 B1 A2 B2)
apply(case-tac  $\neg (\tau \models (\nu \ x))$ ,
  simp add:foundation18 foundation22[symmetric],
  drule StrongEq-L-sym)
apply(simp add: foundation22 C)
apply(case-tac  $\neg (\tau \models (\nu \ y))$ ,
  simp add:foundation18 foundation22[symmetric],
  drule StrongEq-L-sym, simp add: foundation22 D, simp)
apply(subst E,simp-all)
apply(case-tac  $\tau \models \text{not } (x \triangleq y)$ )
apply(simp add: OclExcluding-charn1[simplified foundation22]
  OclExcluding-charn2[simplified foundation22])
apply(simp add: foundation9 F)
done
qed

```

```

schematic-lemma OclExcluding-charn-execInteger[simp,code-unfold]: ?X
by(rule OclExcluding-charn-exec[OF StrictRefEqInteger.strict1 StrictRefEqInteger.strict2
  StrictRefEqInteger.defined-args-valid
  StrictRefEqInteger.cp0 StrictRefEqInteger.StrictRefEq-vs-StrongEq], simp-all)

```

```

schematic-lemma OclExcluding-charn-execBoolean[simp,code-unfold]: ?X
by(rule OclExcluding-charn-exec[OF StrictRefEqBoolean.strict1 StrictRefEqBoolean.strict2
  StrictRefEqBoolean.defined-args-valid
  StrictRefEqBoolean.cp0 StrictRefEqBoolean.StrictRefEq-vs-StrongEq], simp-all)

```

```

schematic-lemma OclExcluding-charn-execSet[simp,code-unfold]: ?X
by(rule OclExcluding-charn-exec[OF StrictRefEqSet.strict1 StrictRefEqSet.strict2
  StrictRefEqSet.defined-args-valid
  StrictRefEqSet.cp0 StrictRefEqSet.StrictRefEq-vs-StrongEq], simp-all)

```

Execution Rules on OclIncludes **lemma** *OclIncludes-charn0[simp]:*

```

assumes val-x:  $\tau \models (\nu \ x)$ 
shows  $\tau \models \text{not}(\text{Set}\{\} \rightarrow \text{includes}(x))$ 
using val-x
apply(auto simp: OclValid-def OclIncludes-def OclNot-def false-def true-def)
apply(auto simp: mtSet-def Setbase.Abs-Setbase-inverse)

```

done

lemma *OclIncludes-cha0*[simp,code-unfold]:

Set{*x*} \rightarrow includes(*x*) = (if *v* *x* then false else invalid endif)

proof –

have *A*: $\bigwedge \tau. (\text{Set}\{x\} \rightarrow \text{includes}(\text{invalid})) \tau = (\text{if } (v \text{ invalid}) \text{ then false else invalid endif}) \tau$
by *simp*

have *B*: $\bigwedge \tau x. \tau \models (v x) \implies (\text{Set}\{x\} \rightarrow \text{includes}(x)) \tau = (\text{if } v x \text{ then false else invalid endif}) \tau$

apply(*frule* *OclIncludes-cha0*, *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* $\tau \models (v x)$)

apply(*simp-all* add: *B.foundation18*)

apply(*subst* *cp-OclIncludes*, *simp* add: *cp-OclIncludes[symmetric]* *A*)

done

qed

lemma *OclIncludes-cha1*:

assumes *def-X*: $\tau \models (\delta X)$

assumes *val-x*: $\tau \models (v x)$

shows $\tau \models (X \rightarrow \text{including}(x) \rightarrow \text{includes}(x))$

proof –

have *C*: $[[\text{insert } (x \ \tau) \ [\text{Rep-Set}_{\text{base}}(X \ \tau)]]] \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in [\text{X}]. x \neq \text{bot})\}$

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-cha2*:

assumes *def-X*: $\tau \models (\delta X)$

and *val-x*: $\tau \models (v x)$

and *val-y*: $\tau \models (v y)$

and *neq* : $\tau \models \text{not}(x \triangleq y)$

shows $\tau \models (X \rightarrow \text{including}(x) \rightarrow \text{includes}(y)) \triangleq (X \rightarrow \text{includes}(y))$

proof –

have *C*: $[[\text{insert } (x \ \tau) \ [\text{Rep-Set}_{\text{base}}(X \ \tau)]]] \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in [\text{X}]. x \neq \text{bot})\}$

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 \doteq y) = invalid

and strict2: (x \doteq invalid) = invalid

and cp-StrictRefEq: $\bigwedge (X::('A, 'a::\text{null})\text{val}) Y \tau. (X \doteq Y) \tau = ((\lambda \cdot. X \tau) \doteq (\lambda \cdot. Y \tau)) \tau$

and StrictRefEq-vs-StrongEq: $\bigwedge (x::('A, 'a::\text{null})\text{val}) y \tau.$
 $\tau \models v x \implies \tau \models v y \implies (\tau \models ((x \doteq y) \triangleq (x \triangleq y)))$

shows

$(X \rightarrow \text{including}(x::('A, 'a::\text{null})\text{val}) \rightarrow \text{includes}(y)) =$
 $(\text{if } \delta X \text{ then if } x \doteq y \text{ then true else } X \rightarrow \text{includes}(y) \text{ endif else invalid endif})$

proof –

have A: $\bigwedge \tau. \tau \models (X \triangleq \text{invalid}) \implies$
 $(X \rightarrow \text{including}(x) \rightarrow \text{includes}(y)) \tau = \text{invalid } \tau$
apply(rule foundation22[THEN iffD1])
by(erule StrongEq-L-subst2-rev, simp, simp)

have B: $\bigwedge \tau. \tau \models (X \triangleq \text{null}) \implies$
 $(X \rightarrow \text{including}(x) \rightarrow \text{includes}(y)) \tau = \text{invalid } \tau$
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: $\bigwedge \tau. \tau \models (x \triangleq \text{invalid}) \implies$
 $(X \rightarrow \text{including}(x) \rightarrow \text{includes}(y)) \tau =$
 $(\text{if } x \doteq y \text{ then true else } X \rightarrow \text{includes}(y) \text{ endif}) \tau$
apply(rule foundation22[THEN iffD1])
apply(erule StrongEq-L-subst2-rev, simp, simp)
by (simp add: strict1)

have D: $\bigwedge \tau. \tau \models (y \triangleq \text{invalid}) \implies$
 $(X \rightarrow \text{including}(x) \rightarrow \text{includes}(y)) \tau =$
 $(\text{if } x \doteq y \text{ then true else } X \rightarrow \text{includes}(y) \text{ endif}) \tau$
apply(rule foundation22[THEN iffD1])
apply(erule StrongEq-L-subst2-rev, simp, simp)
by (simp add: strict2)

have E: $\bigwedge \tau. \tau \models v x \implies \tau \models v y \implies$
 $(\text{if } x \doteq y \text{ then true else } X \rightarrow \text{includes}(y) \text{ endif}) \tau =$
 $(\text{if } x \triangleq y \text{ then true else } X \rightarrow \text{includes}(y) \text{ endif}) \tau$

```

apply(subst cp-OclIf)
apply(subst StrictRefEq-vs-StrongEq[THEN foundation22[THEN iffD1]])
by(simp-all add: cp-OclIf[symmetric])
have F:  $\bigwedge \tau. \tau \models (x \triangleq y) \implies$ 
   $(X \multimap \text{including}(x) \multimap \text{includes}(y)) \ \tau = (X \multimap \text{including}(x) \multimap \text{includes}(x)) \ \tau$ 
apply(rule foundation22[THEN iffD1])
by(erule StrongEq-L-subst2-rev, simp, simp)
show ?thesis
apply(rule ext, rename-tac  $\tau$ )
apply(case-tac  $\neg (\tau \models (\delta \ X))$ , simp add: defined-split, elim disjE A B)
apply(case-tac  $\neg (\tau \models (\nu \ x))$ ,
  simp add: foundation18 foundation22[symmetric],
  drule StrongEq-L-sym)
apply(simp add: foundation22 C)
apply(case-tac  $\neg (\tau \models (\nu \ y))$ ,
  simp add: foundation18 foundation22[symmetric],
  drule StrongEq-L-sym, simp add: foundation22 D, simp)
apply(subst E, simp-all)
apply(case-tac  $\tau \models \text{not}(x \triangleq y)$ )
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-executeInteger[simp, code-unfold]: ?X
by(rule OclIncludes-execute-generic[OF StrictRefEqInteger.strict1 StrictRefEqInteger.strict2
  StrictRefEqInteger.cp0
  StrictRefEqInteger.StrictRefEq-vs-StrongEq], simp-all)

```

```

schematic-lemma OclIncludes-executeBoolean[simp, code-unfold]: ?X
by(rule OclIncludes-execute-generic[OF StrictRefEqBoolean.strict1 StrictRefEqBoolean.strict2
  StrictRefEqBoolean.cp0
  StrictRefEqBoolean.StrictRefEq-vs-StrongEq], simp-all)

```

```

schematic-lemma OclIncludes-executeSet[simp, code-unfold]: ?X
by(rule OclIncludes-execute-generic[OF StrictRefEqSet.strict1 StrictRefEqSet.strict2
  StrictRefEqSet.cp0
  StrictRefEqSet.StrictRefEq-vs-StrongEq], simp-all)

```

```

lemma OclIncludes-including-generic :
assumes OclIncludes-execute-generic [simp] :  $\bigwedge X \ x \ y.$ 
   $(X \multimap \text{including}(x :: (\mathcal{A}, 'a :: \text{null}) \text{val}) \multimap \text{includes}(y)) =$ 

```

(if δX then if $x \doteq y$ then true else $X \rightarrow \text{includes}(y)$ endif else invalid endif)
and *StrictRefEq-strict''* : $\bigwedge x y. \delta ((x::('A, 'a::\text{null})\text{val}) \doteq y) = (v(x) \text{ and } v(y))$
and *a-val* : $\tau \models v a$
and *x-val* : $\tau \models v x$
and *S-incl* : $\tau \models (S) \rightarrow \text{includes}((x::('A, 'a::\text{null})\text{val}))$
shows $\tau \models S \rightarrow \text{including}((a::('A, 'a::\text{null})\text{val})) \rightarrow \text{includes}(x)$
proof –
have *discr-eq-bot1-true* : $\bigwedge \tau. (\perp \tau = \text{true } \tau) = \text{False}$
by (*metis bot-fun-def foundation1 foundation18' valid3*)
have *discr-eq-bot2-true* : $\bigwedge \tau. (\perp = \text{true } \tau) = \text{False}$
by (*metis bot-fun-def discr-eq-bot1-true*)
have *discr-neq-invalid-true* : $\bigwedge \tau. (\text{invalid } \tau \neq \text{true } \tau) = \text{True}$
by (*metis discr-eq-bot2-true invalid-def*)
have *discr-eq-invalid-true* : $\bigwedge \tau. (\text{invalid } \tau = \text{true } \tau) = \text{False}$
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 δS] 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*: $\tau \models (v x)$
shows $\tau \models (X \rightarrow \text{excluding}(x) \rightarrow \text{excludes}(x))$
proof –
let ?OclSet = $\lambda S. \llbracket S \rrbracket \in \{X. X = \perp \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \perp)\}$
have *diff-in-Set_{base}* : ?OclSet ($\llbracket \text{Rep-Set}_{\text{base}}(X \tau) \rrbracket - \{x \tau\}$)
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]: $\text{Set}\{\} \rightarrow \text{size}() = 0$

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]:

(($X \rightarrow \text{including}(x)$) $\rightarrow \text{size}()$) = (if δX and $\forall x$ then

$X \rightarrow \text{size}() +_{\text{int}}$ if $X \rightarrow \text{includes}(x)$ then **0** else **1** endif

else

invalid

endif)

proof –

have valid-inject-true : $\bigwedge \tau P. (\forall P) \tau \neq \text{true} \tau \implies (\forall P) \tau = \text{false} \tau$

apply(simp add: valid-def true-def false-def bot-fun-def bot-option-def
null-fun-def null-option-def)

by (case-tac $P \tau = \perp$, simp-all add: true-def)

have defined-inject-true : $\bigwedge \tau P. (\delta P) \tau \neq \text{true} \tau \implies (\delta P) \tau = \text{false} \tau$

apply(simp add: defined-def true-def false-def bot-fun-def bot-option-def
null-fun-def null-option-def)

by (case-tac $P \tau = \perp \vee P \tau = \text{null}$, simp-all add: true-def)

show ?thesis

apply(rule ext, rename-tac τ)

proof –

fix τ

have includes-notin: $\neg \tau \models X \rightarrow \text{includes}(x) \implies (\delta X) \tau = \text{true} \tau \wedge (\forall x) \tau = \text{true} \tau \implies$
 $x \tau \notin \llbracket \text{Rep-Set}_{\text{base}}(X \tau) \rrbracket$

by(simp add: OclIncludes-def OclValid-def true-def)

have includes-def: $\tau \models X \rightarrow \text{includes}(x) \implies \tau \models \delta X$

by (metis bot-fun-def OclIncludes-def OclValid-def defined3 foundation16)

have includes-val: $\tau \models X \rightarrow \text{includes}(x) \implies \tau \models \forall x$

by (metis (hide-lams, no-types) foundation6

OclIncludes-valid-args-valid' OclIncluding-valid-args-valid OclIncluding-valid-args-valid')

have ins-in-Set_{base}: $\tau \models \delta X \implies \tau \models \forall x \implies$

$\llbracket \text{insert}(x \tau) \llbracket \text{Rep-Set}_{\text{base}}(X \tau) \rrbracket \rrbracket \in \{X. X = \perp \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \perp)\}$

apply(simp add: bot-option-def null-option-def)

by (metis (hide-lams, no-types) Set-inv-lemma foundation18' foundation5)

have $m : \bigwedge \tau. (\lambda \cdot. \perp) = (\lambda \cdot. \text{invalid} \tau)$ by(rule ext, simp add: invalid-def)

show $X \rightarrow \text{including}(x) \rightarrow \text{size}() \tau = (\text{if } \delta X \text{ and } \forall x$

```

      then  $X \rightarrow \text{size}()$  +int if  $X \rightarrow \text{includes}(x)$  then 0 else 1 endif
      else invalid endif)  $\tau$ 
apply(case-tac  $\tau \models \delta X$  and  $\forall x$ , simp)
apply(subst OclAddInteger.cp0)
apply(case-tac  $\tau \models X \rightarrow \text{includes}(x)$ , simp add: OclAddInteger.cp0[symmetric])
apply(case-tac  $\tau \models ((\forall (X \rightarrow \text{size}())) \text{ and not } (\delta (X \rightarrow \text{size}())))$ , simp)
apply(drule foundation5[where  $P = \forall X \rightarrow \text{size}()$ ], 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) OclAddInteger.cp0, simp)
apply(rule conjI)
apply(simp add: OclIncluding-def)
apply(subst Abs-Setbase-inverse[OF ins-in-Setbase], fast+)
apply(subst (asm) (2 3) OclValid-def, simp add: OclAddInteger-def OclInt1-def)
apply(rule impI)
apply(drule Finite-Set.card.insert[where  $x = x \ \tau$ ])
apply(rule includes-notin, simp, simp)
apply(metis Suc-eq-plus1 int-1 of-nat-add)

apply(subst (1 2) m[of  $\tau$ ], simp only: OclAddInteger.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]: $\text{Set}\{\}-\rightarrow \text{isEmpty}() = \text{true}$
by(simp add: OclIsEmpty-def)

lemma OclIsEmpty-including [simp]:
assumes $X\text{-def}: \tau \models \delta X$
and $X\text{-finite}: \text{finite } [\text{Rep-Set}_{\text{base}}(X \ \tau)]$
and $a\text{-val}: \tau \models \forall a$
shows $X \rightarrow \text{including}(a) \rightarrow \text{isEmpty}() \ \tau = \text{false} \ \tau$

proof –

have $AI : \bigwedge \tau X. X \tau = \text{true} \vee X \tau = \text{false} \tau \implies (X \text{ and not } X) \tau = \text{false} \tau$
by (metis (no-types) OclAnd-false1 OclAnd-idem OclImplies-def OclNot3 OclNot-not OclOr-false1
 cp-OclAnd cp-OclNot deMorgan1 deMorgan2)

have $\text{defined-inject-true} : \bigwedge \tau P. (\delta P) \tau \neq \text{true} \tau \implies (\delta P) \tau = \text{false} \tau$
apply (simp add: defined-def true-def false-def bot-fun-def bot-option-def
 null-fun-def null-option-def)
by (case-tac $P \tau = \perp \vee P \tau = \text{null}$, simp-all add: true-def)

have $B : \bigwedge X \tau. \tau \models v X \implies X \tau \neq \mathbf{0} \tau \implies (X \doteq \mathbf{0}) \tau = \text{false} \tau$
apply (simp add: foundation22[symmetric] foundation14 foundation9)
apply (erule StrongEq-L-subst4-rev[THEN iffD2, OF StrictRefEqInteger.StrictRefEq-vs-StrongEq])
by (simp-all)

show ?thesis

apply (simp add: OclIsEmpty-def del: OclSize-including-exec)
apply (subst cp-OclOr, subst AI)
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]: $\text{Set}\{\} \rightarrow \text{notEmpty}() = \text{false}$
by (simp add: OclNotEmpty-def)

lemma OclNotEmpty-including [simp,code-unfold]:
assumes $X\text{-def}: \tau \models \delta X$
and $X\text{-finite}: \text{finite } [\text{Rep-Set}_{\text{base}}(X \tau)]$
and $a\text{-val}: \tau \models v a$
shows $X \rightarrow \text{including}(a) \rightarrow \text{notEmpty}() \tau = \text{true} \tau$
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]: $\text{Set}\{\} \rightarrow \text{any}() = \text{null}$
by (rule ext, simp add: OclANY-def, simp add: false-def true-def)

lemma OclANY-singleton-exec [simp,code-unfold]:
 $(\text{Set}\{\} \rightarrow \text{including}(a)) \rightarrow \text{any}() = a$
apply (rule ext, rename-tac τ , simp add: mtSet-def OclANY-def)
apply (case-tac $\tau \models v a$)

```

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  $X = \text{Set}\{\}$ , 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-Setbase-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  $\tau = \perp$ )
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] : $((\text{Set}\{\}) \rightarrow \text{forAll}(z \mid P(z))) = \text{true}$

```

apply(simp add: OclForall-def)
apply(subst mtSet-def)+
apply(subst Abs-Setbase-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:

$\text{Integer}_{\text{null}} \rightarrow \text{forAll}(x \mid \text{Integer}_{\text{null}} \rightarrow \text{forAll}(y \mid x +_{\text{int}} y \triangleq y +_{\text{int}} x))$
 or even:
 $\text{Integer} \rightarrow \text{forAll}(x \mid \text{Integer} \rightarrow \text{forAll}(y \mid x +_{\text{int}} y \doteq y +_{\text{int}} x))$
 are valid OCL statements in any context τ .

theorem *OclForall-including-exec*[simp,code-unfold] :

```

assumes cp0 : cp P
shows       $((S \rightarrow \text{including}(x)) \rightarrow \text{forAll}(z \mid P(z))) = (\text{if } \delta S \text{ and } \forall x$ 
       $\text{then } P x \text{ and } (S \rightarrow \text{forAll}(z \mid P(z)))$ 
       $\text{else invalid}$ 
       $\text{endif})$ 

```

proof —

```

have cp:  $\bigwedge \tau. P x \tau = P (\lambda \cdot. x \tau) \tau$  by(insert cp0, auto simp: cp-def)

```

have $cp\text{-}eq : \wedge \tau v. (P\ x\ \tau = v) = (P\ (\lambda -. x\ \tau)\ \tau = v)$ **by**(*subst cp, simp*)

have $cp\text{-}OclNot\text{-}eq : \wedge \tau v. (P\ x\ \tau \neq v) = (P\ (\lambda -. x\ \tau)\ \tau \neq v)$ **by**(*subst cp, simp*)

have $insert\text{-}in\text{-}Set_{base} : \wedge \tau. (\tau \models (\delta\ S)) \implies (\tau \models (v\ x)) \implies$
 $\llbracket insert\ (x\ \tau)\ \llbracket Rep\text{-}Set_{base}\ (S\ \tau) \rrbracket \rrbracket \in$
 $\{X. X = bot \vee X = null \vee (\forall x \in \llbracket X \rrbracket. x \neq bot)\}$
by(*frule Set-inv-lemma, simp add: foundation18 invalid-def*)

have $forall\text{-}including\text{-}invert : \wedge \tau f. (f\ x\ \tau = f\ (\lambda -. x\ \tau)\ \tau) \implies$
 $\tau \models (\delta\ S\ and\ v\ x) \implies$
 $(\forall x \in \llbracket Rep\text{-}Set_{base}\ (S\text{-}>including(x)\ \tau) \rrbracket. f\ (\lambda -. x)\ \tau =$
 $(f\ x\ \tau \wedge (\forall x \in \llbracket Rep\text{-}Set_{base}\ (S\ \tau) \rrbracket. f\ (\lambda -. x)\ \tau))$
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 $exists\text{-}including\text{-}invert : \wedge \tau f. (f\ x\ \tau = f\ (\lambda -. x\ \tau)\ \tau) \implies$
 $\tau \models (\delta\ S\ and\ v\ x) \implies$
 $(\exists x \in \llbracket Rep\text{-}Set_{base}\ (S\text{-}>including(x)\ \tau) \rrbracket. f\ (\lambda -. x)\ \tau =$
 $(f\ x\ \tau \vee (\exists x \in \llbracket Rep\text{-}Set_{base}\ (S\ \tau) \rrbracket. f\ (\lambda -. x)\ \tau))$
apply(*subst arg-cong[where f = $\lambda x. \neg x$,*
OF forall-including-invert[where f = $\lambda x\ \tau. \neg (f\ x\ \tau)$,
simplified])
by *simp-all*

have $contradict\text{-}Rep\text{-}Set_{base} : \wedge \tau S f. \exists x \in \llbracket Rep\text{-}Set_{base}\ S \rrbracket. f\ (\lambda -. x)\ \tau \implies$
 $(\forall x \in \llbracket Rep\text{-}Set_{base}\ S \rrbracket. \neg (f\ (\lambda -. x)\ \tau)) = False$
by(*case-tac ($\forall x \in \llbracket Rep\text{-}Set_{base}\ S \rrbracket. \neg (f\ (\lambda -. x)\ \tau) = True$, simp-all)*)

have $bot\text{-}invalid : \perp = invalid$ **by**(*rule ext, simp add: invalid-def bot-fun-def*)

have $bot\text{-}invalid2 : \wedge \tau. \perp = invalid\ \tau$ **by**(*simp add: invalid-def*)

have $C1 : \wedge \tau. P\ x\ \tau = false\ \tau \vee (\exists x \in \llbracket Rep\text{-}Set_{base}\ (S\ \tau) \rrbracket. P\ (\lambda -. x)\ \tau = false\ \tau) \implies$
 $\tau \models (\delta\ S\ and\ v\ x) \implies$
 $false\ \tau = (P\ x\ and\ OclForall\ S\ P)\ \tau$
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 $\tau = false\ \tau$*)
apply(*simp only: cp-OclAnd[symmetric], simp*)
apply(*simp add: OclForall-def*)
apply(*fold OclValid-def, simp add: foundation27*)
done

have $C2 : \wedge \tau. \tau \models (\delta\ S\ and\ v\ x) \implies$

$P\ x\ \tau = \text{null}\ \tau \vee (\exists x \in [\text{Rep-Set}_{\text{base}}(S\ \tau)]]. P\ (\lambda\cdot. x)\ \tau = \text{null}\ \tau \implies$
 $P\ x\ \tau = \text{invalid}\ \tau \vee (\exists x \in [\text{Rep-Set}_{\text{base}}(S\ \tau)]]. P\ (\lambda\cdot. x)\ \tau = \text{invalid}\ \tau \implies$
 $\forall x \in [\text{Rep-Set}_{\text{base}}(S \rightarrow \text{including}(x)\ \tau)]]. P\ (\lambda\cdot. x)\ \tau \neq \text{false}\ \tau \implies$
 $\text{invalid}\ \tau = (P\ x\ \text{and}\ \text{OclForall}\ S\ P)\ \tau$
apply(subgoal-tac ($\delta\ S$) $\tau = \text{true}\ \tau$)
prefer 2 **apply**(simp add: foundation27, simp add: OclValid-def)
apply(drule forall-including-invert[of $\lambda\ x\ \tau. P\ x\ \tau \neq \text{false}\ \tau$, 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 $S\ P\ \tau = \text{invalid}\ \tau$)
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 ($P\ x\ \tau = \text{invalid}\ \tau$) \vee ($P\ x\ \tau = \text{null}\ \tau$) \vee ($P\ x\ \tau = \text{true}\ \tau$) \vee ($P\ x\ \tau = \text{false}\ \tau$))
prefer 2 **apply**(rule bool-split-0)
apply(elim disjE, simp-all)
apply(simp only: cp-OclAnd[symmetric], simp) +
done

have $A : \bigwedge \tau. \tau \models (\delta\ S\ \text{and}\ v\ x) \implies$
 $\text{OclForall}\ (S \rightarrow \text{including}(x))\ P\ \tau = (P\ x\ \text{and}\ \text{OclForall}\ S\ P)\ \tau$
proof – **fix** τ
assume $0 : \tau \models (\delta\ S\ \text{and}\ v\ x)$
let $?S = \lambda\ \text{ocl}. P\ x\ \tau \neq \text{ocl}\ \tau \wedge (\forall x \in [\text{Rep-Set}_{\text{base}}(S\ \tau)]]. P\ (\lambda\cdot. x)\ \tau \neq \text{ocl}\ \tau$
let $?S' = \lambda\ \text{ocl}. \forall x \in [\text{Rep-Set}_{\text{base}}(S \rightarrow \text{including}(x)\ \tau)]]. P\ (\lambda\cdot. x)\ \tau \neq \text{ocl}\ \tau$
let $?assms-1 = ?S'\ \text{null}$
let $?assms-2 = ?S'\ \text{invalid}$
let $?assms-3 = ?S'\ \text{false}$
have 4 : $?assms-3 \implies ?S\ \text{false}$
apply(subst forall-including-invert[of $\lambda\ x\ \tau. P\ x\ \tau \neq \text{false}\ \tau$, symmetric])
by(simp-all add: cp-OclNot-eq 0)
have 5 : $?assms-2 \implies ?S\ \text{invalid}$
apply(subst forall-including-invert[of $\lambda\ x\ \tau. P\ x\ \tau \neq \text{invalid}\ \tau$, symmetric])
by(simp-all add: cp-OclNot-eq 0)
have 6 : $?assms-1 \implies ?S\ \text{null}$
apply(subst forall-including-invert[of $\lambda\ x\ \tau. P\ x\ \tau \neq \text{null}\ \tau$, symmetric])
by(simp-all add: cp-OclNot-eq 0)
have 7 : $(\delta\ S)\ \tau = \text{true}\ \tau$
by(insert 0, simp add: foundation27, simp add: OclValid-def)
show $?thesis\ \tau$
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** $f = \lambda\ x\ \tau. P\ x\ \tau = \text{null}\ \tau$, OF cp-eq])
apply(simp-all add: exists-including-invert[**where** $f = \lambda\ x\ \tau. P\ x\ \tau = \text{invalid}\ \tau$, OF cp-eq])
apply(simp-all add: exists-including-invert[**where** $f = \lambda\ x\ \tau. P\ x\ \tau = \text{false}\ \tau$, OF cp-eq])

```

proof –
  assume 1 :  $P\ x\ \tau = \text{null}\ \tau \vee (\exists x \in [\text{Rep-Set}_{\text{base}}(S\ \tau)] . P\ (\lambda \cdot. x)\ \tau = \text{null}\ \tau)$ 
  and 2 : ?assms-2
  and 3 : ?assms-3
  show  $\text{null}\ \tau = (P\ x\ \text{and}\ \text{OclForall}\ S\ P)\ \tau$ 
  proof –
    note 4 = 4[OF 3]
    note 5 = 5[OF 2]
    have 6 :  $P\ x\ \tau = \text{null}\ \tau \vee P\ x\ \tau = \text{true}\ \tau$ 
      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 :  $P\ x\ \tau = \text{invalid}\ \tau \vee (\exists x \in [\text{Rep-Set}_{\text{base}}(S\ \tau)] . P\ (\lambda \cdot. x)\ \tau = \text{invalid}\ \tau)$ 
  and 3 : ?assms-3
  show  $\text{invalid}\ \tau = (P\ x\ \text{and}\ \text{OclForall}\ S\ P)\ \tau$ 
  proof –
    note 4 = 4[OF 3]
    note 6 = 6[OF 1]
    have 5 :  $P\ x\ \tau = \text{invalid}\ \tau \vee P\ x\ \tau = \text{true}\ \tau$ 
      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  $\text{true}\ \tau = (P\ x\ \text{and}\ \text{OclForall}\ S\ P)\ \tau$ 
  proof –
    note 4 = 4[OF 3]
    note 5 = 5[OF 2]

```



```

apply(rule ext, rename-tac  $\tau$ )
apply(case-tac  $A \ \tau = \perp \ \tau$ , simp-all, simp add:true-def false-def bot-fun-def)
apply(simp add: Abs-Setbase-inverse)
done
qed

```

In particular, this does hold for $A = \text{null}$.

lemma *OclIterate-including*:

```

assumes S-finite:  $\tau \models \delta(S \rightarrow \text{size}())$ 
and F-valid-arg:  $(\forall A) \ \tau = (\forall (F \ a \ A)) \ \tau$ 
and F-commute: comp-fun-commute F
and F-cp:  $\bigwedge x \ y \ \tau. F \ x \ y \ \tau = F \ (\lambda \ -. \ x \ \tau) \ y \ \tau$ 
shows  $((S \rightarrow \text{including}(a)) \rightarrow \text{iterate}(a; x = A \mid F \ a \ x)) \ \tau =$ 
 $((S \rightarrow \text{excluding}(a)) \rightarrow \text{iterate}(a; x = F \ a \ A \mid F \ a \ x)) \ \tau$ 
proof –
have insert-in-Setbase :  $\bigwedge \tau. (\tau \models (\delta \ S)) \implies (\tau \models (\forall a)) \implies$ 
 $\llbracket \text{insert} \ (a \ \tau) \ \llbracket \text{Rep-Set}_{\text{base}} \ (S \ \tau) \rrbracket \rrbracket \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$ 
by(frule Set-inv-lemma, simp add: foundation18 invalid-def)

have insert-defined :  $\bigwedge \tau. (\tau \models (\delta \ S)) \implies (\tau \models (\forall a)) \implies$ 
 $(\delta \ (\lambda \ -. \ \text{Abs-Set}_{\text{base}} \ \llbracket \text{insert} \ (a \ \tau) \ \llbracket \text{Rep-Set}_{\text{base}} \ (S \ \tau) \rrbracket \rrbracket)) \ \tau = \text{true} \ \tau$ 
apply(subst defined-def)
apply(simp add: bot-Setbase-def bot-fun-def null-Setbase-def null-fun-def)
by(subst Abs-Setbase-inject,
rule insert-in-Setbase, simp-all add: null-option-def bot-option-def)+

have remove-finite : finite  $\llbracket \text{Rep-Set}_{\text{base}} \ (S \ \tau) \rrbracket \implies$ 
finite  $((\lambda a \ \tau. a) \ ' (\llbracket \text{Rep-Set}_{\text{base}} \ (S \ \tau) \rrbracket - \{a \ \tau\}))$ 
by(simp)

have remove-in-Setbase :  $\bigwedge \tau. (\tau \models (\delta \ S)) \implies (\tau \models (\forall a)) \implies$ 
 $\llbracket \llbracket \text{Rep-Set}_{\text{base}} \ (S \ \tau) \rrbracket - \{a \ \tau\} \rrbracket \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$ 
by(frule Set-inv-lemma, simp add: foundation18 invalid-def)

have remove-defined :  $\bigwedge \tau. (\tau \models (\delta \ S)) \implies (\tau \models (\forall a)) \implies$ 
 $(\delta \ (\lambda \ -. \ \text{Abs-Set}_{\text{base}} \ \llbracket \llbracket \text{Rep-Set}_{\text{base}} \ (S \ \tau) \rrbracket - \{a \ \tau\} \rrbracket)) \ \tau = \text{true} \ \tau$ 
apply(subst defined-def)
apply(simp add: bot-Setbase-def bot-fun-def null-Setbase-def null-fun-def)
by(subst Abs-Setbase-inject,
rule remove-in-Setbase, simp-all add: null-option-def bot-option-def)+

have abs-rep:  $\bigwedge x. \llbracket [x] \rrbracket \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\} \implies$ 
 $\llbracket \llbracket \text{Rep-Set}_{\text{base}} \ (\text{Abs-Set}_{\text{base}} \ \llbracket [x] \rrbracket) \rrbracket \rrbracket = x$ 
by(subst Abs-Setbase-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 $\neg ((\delta S) \tau = \text{true } \tau \wedge (\vee a) \tau = \text{true } \tau)$, simp add: invalid-def)

apply(subgoal-tac OclIterate ($\lambda \cdot. \perp$) A F $\tau = \text{OclIterate } (\lambda \cdot. \perp) (F a A) F \tau$, 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 $\neg ((\vee A) \tau = \text{true } \tau)$, (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 ($\lambda \tau'. a \tau$) (($\lambda a \tau. a$) ' $\llbracket \text{Rep-Set}_{base} (S \tau) \rrbracket$))) $\tau =$
F ($\lambda \tau'. a \tau$) (Finite-Set.fold F A (($\lambda a \tau. a$) ' $\llbracket \text{Rep-Set}_{base} (S \tau) \rrbracket - \{\lambda \tau'. a \tau\}$)) 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 :: $- \Rightarrow - \Rightarrow - \Rightarrow (^{\mathcal{A}}, 'a \text{ option option}) \text{ Set}$
 $\equiv (\lambda P x \text{ acc. if } P x \doteq \text{false then acc else acc} \rightarrow \text{including}(x) \text{ endif})$

theorem OclSelect-including-exec[simp,code-unfold]:

assumes P-cp : cp P
shows OclSelect (X \rightarrow including(y)) P = OclSelect-body P y (OclSelect (X \rightarrow excluding(y)) P)
(is - = ?select)

proof –

have P-cp: $\bigwedge x \tau. P x \tau = P (\lambda \cdot. x \tau) \tau$ **by**(insert P-cp, auto simp: cp-def)

have ex-including : $\bigwedge f X y \tau. \tau \models \delta X \implies \tau \models \vee y \implies$
 $(\exists x \in \llbracket \text{Rep-Set}_{base} (X \rightarrow \text{including}(y) \tau) \rrbracket. f (P (\lambda \cdot. x)) \tau) =$
 $(f (P (\lambda \cdot. y \tau)) \tau \vee (\exists x \in \llbracket \text{Rep-Set}_{base} (X \tau) \rrbracket. f (P (\lambda \cdot. x)) \tau))$
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 *al-including* : $\bigwedge f X y \tau. \tau \models \delta X \implies \tau \models v y \implies$
 $(\forall x \in [\![\text{Rep-Set}_{base} (X \rightarrow \text{including}(y) \tau)]\!]. f (P (\lambda \cdot. x)) \tau) =$
 $(f (P (\lambda \cdot. y \tau)) \tau \wedge (\forall x \in [\![\text{Rep-Set}_{base} (X \tau)]\!]. f (P (\lambda \cdot. x)) \tau))$
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* : $\bigwedge f X y \tau. \tau \models \delta X \implies \tau \models v y \implies \neg (f (P (\lambda \cdot. y \tau)) \tau) \implies$
 $(\exists x \in [\![\text{Rep-Set}_{base} (X \tau)]\!]. f (P (\lambda \cdot. x)) \tau) =$
 $(\exists x \in [\![\text{Rep-Set}_{base} (X \rightarrow \text{excluding}(y) \tau)]\!]. f (P (\lambda \cdot. x)) \tau)$
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* : $\bigwedge f X y \tau. \tau \models \delta X \implies \tau \models v y \implies f (P (\lambda \cdot. y \tau)) \tau \implies$
 $(\forall x \in [\![\text{Rep-Set}_{base} (X \tau)]\!]. f (P (\lambda \cdot. x)) \tau) =$
 $(\forall x \in [\![\text{Rep-Set}_{base} (X \rightarrow \text{excluding}(y) \tau)]\!]. f (P (\lambda \cdot. x)) \tau)$
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* : $\bigwedge f X y \tau. \tau \models \delta X \implies \tau \models v y \implies$
 $\{x \in [\![\text{Rep-Set}_{base} (X \rightarrow \text{including}(y) \tau)]\!]. f (P (\lambda \cdot. x)) \tau\} =$
 $(\text{let } s = \{x \in [\![\text{Rep-Set}_{base} (X \tau)]\!]. f (P (\lambda \cdot. x)) \tau\} \text{ in}$
 $\text{if } f (P (\lambda \cdot. y \tau)) \tau \text{ then insert } (y \tau) s \text{ else } s)$
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 *?OclSet* = $\lambda S. [\![S]\!] \in \{X. X = \perp \vee X = \text{null} \vee (\forall x \in [\![X]\!]. x \neq \perp)\}$

have *diff-in-Set_{base}* : $\bigwedge \tau. (\delta X) \tau = \text{true} \tau \implies ?\text{OclSet} ([\![\text{Rep-Set}_{base} (X \tau)]\!] - \{y \tau\})$
apply(*simp, (rule disjI2)+*)
by (*metis (mono-tags) Diff-iff OclValid-def Set-inv-lemma*)

have *ins-in-Set_{base}* : $\bigwedge \tau. (\delta X) \tau = \text{true} \tau \implies (v y) \tau = \text{true} \tau \implies$
 $?\text{OclSet} (\text{insert } (y \tau) \{x \in [\![\text{Rep-Set}_{base} (X \tau)]\!]. P (\lambda \cdot. x) \tau \neq \text{false} \tau\})$
apply(*simp, (rule disjI2)+*)
by (*metis (hide-lams, no-types) OclValid-def Set-inv-lemma foundation18'*)

have *ins-in-Set_{base}'* : $\bigwedge \tau. (\delta X) \tau = \text{true} \tau \implies (v y) \tau = \text{true} \tau \implies$
 $?\text{OclSet} (\text{insert } (y \tau) \{x \in [\![\text{Rep-Set}_{base} (X \tau)]\!]. x \neq y \tau \wedge P (\lambda \cdot. x) \tau \neq \text{false} \tau\})$
apply(*simp, (rule disjI2)+*)
by (*metis (hide-lams, no-types) OclValid-def Set-inv-lemma foundation18'*)

have *ins-in-Set_{base}'* : $\wedge \tau. (\delta X) \tau = \text{true} \tau \implies$
 $?OclSet \{x \in [\![Rep-Set_{base} (X \tau)]\!]. P (\lambda \cdot x) \tau \neq \text{false} \tau\}$
apply(*simp*, (*rule disjI2*)+)
by (*metis* (*hide-lams*, *no-types*) *OclValid-def Set-inv-lemma*)

have *ins-in-Set_{base}'''* : $\wedge \tau. (\delta X) \tau = \text{true} \tau \implies$
 $?OclSet \{x \in [\![Rep-Set_{base} (X \tau)]\!]. x \neq y \tau \wedge P (\lambda \cdot x) \tau \neq \text{false} \tau\}$
apply(*simp*, (*rule disjI2*)+)
by(*metis* (*hide-lams*, *no-types*) *OclValid-def Set-inv-lemma*)

have *if-same* : $\wedge a b c d \tau. \tau \models \delta a \implies b \tau = d \tau \implies c \tau = d \tau \implies$
 $(\text{if } a \text{ then } b \text{ else } c \text{ endif}) \tau = d \tau$
by(*simp add: OclIf-def OclValid-def*)

have *invert-including* : $\wedge P y \tau. P \tau = \perp \implies P \neg \text{including}(y) \tau = \perp$
by (*metis* (*hide-lams*, *no-types*) *foundation16[THEN iffD1,standard]*
foundation18' OclIncluding-valid-args-valid)

have *exclude-defined* : $\wedge \tau. \tau \models \delta X \implies$
 $(\delta(\lambda \cdot. Abs-Set_{base} [\![\{x \in [\![Rep-Set_{base} (X \tau)]\!]. x \neq y \tau \wedge P (\lambda \cdot x) \tau \neq \text{false} \tau]\!]\!]) \tau = \text{true} \tau$
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* : $\wedge x A B \tau. \tau \models v x \implies \tau \models ((\text{if } x \doteq \text{false} \text{ then } A \text{ else } B \text{ endif}) \triangleq$
 $(\text{if } x \triangleq \text{false} \text{ then } A \text{ else } B \text{ endif}))$
apply(*simp add: StrictRefEqB_{boolean} OclValid-def*)
apply(*subst* (2) *StrongEq-def*)
by(*subst cp-OclIf*, *simp add: cp-OclIf[symmetric] true-def*)

have *OclSelect-body-bot* : $\wedge \tau. \tau \models \delta X \implies \tau \models v y \implies P y \tau \neq \perp \implies$
 $(\exists x \in [\![Rep-Set_{base} (X \tau)]\!]. P (\lambda \cdot x) \tau = \perp) \implies \perp = ?select \tau$
apply(*drule ex-excludingI[where X = X and y = y and f = $\lambda x \tau. x \tau = \perp$]*,
(simp add: P-cp[symmetric]))+
apply(*subgoal-tac* $\tau \models (\perp \triangleq ?select)$, *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* $\exists x \in [\![Rep-Set_{base} (X \neg \text{excluding}(y) \tau)]\!]. P (\lambda \cdot x) \tau = \perp \tau$)
prefer 2 **apply** (*metis bot-fun-def*)
apply(*subst if-same[where d = \perp]*)
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* : $\bigwedge \tau X y. (\delta X \text{ and } v y) \tau \neq \text{true } \tau \implies (\delta X \text{ and } v y) \tau = \text{false } \tau$
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'*: $\bigwedge \tau. (\delta X \text{ and } v y) \tau \neq \text{true } \tau \implies \perp = ?\text{select } \tau$
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* : $\bigwedge a b c \tau. ((a \triangleq \text{false}) \tau = \text{false } \tau \longrightarrow b) \wedge ((a \triangleq \text{false}) \tau = \text{true } \tau \longrightarrow c) \implies$
 $(a \tau \neq \text{false } \tau \longrightarrow b) \wedge (a \tau = \text{false } \tau \longrightarrow c)$
by (*metis OclValid-def defined7 foundation14 foundation22 foundation9*)

have *defined-inject-true* : $\bigwedge \tau P. (\delta P) \tau \neq \text{true } \tau \implies (\delta P) \tau = \text{false } \tau$
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* $\tau = \perp \vee P \tau = \text{null}$, *simp-all add: true-def*)

have *cp-OclSelect-body* : $\bigwedge \tau. ?\text{select } \tau = \text{OclSelect-body } P y (\lambda -. (\text{OclSelect } (X \rightarrow \text{excluding}(y)) P) \tau) \tau$
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*: $\bigwedge (x::('A)\text{Boolean}) y \tau. \neg (\tau \models v x) \implies \tau \models ((x \doteq y) \triangleq \text{invalid})$
by (*simp add: StrictRefEqBoolean OclValid-def StrongEq-def true-def*)

have *conj-comm* : $\bigwedge p q r. (p \wedge q \wedge r) = ((p \wedge q) \wedge r)$ **by** *blast*

have *inv-bot* : $\bigwedge \tau. \text{invalid } \tau = \perp$ **by** (*metis bot-fun-def invalid-def*)
have *inv-bot'* : $\bigwedge \tau. \text{invalid } \tau = \perp$ **by** (*simp add: invalid-def*)

show *?thesis*
apply (*rule ext*, *rename-tac* τ)
apply (*subst OclSelect-def*)
apply (*case-tac* $(\delta (X \rightarrow \text{including}(y))) \tau = \text{true } \tau$, *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  $\neg (\tau \models (\vee P y))$ )
apply(subgoal-tac  $P y \tau \neq \text{false } \tau$ )
prefer 2
apply (metis (hide-lams, no-types) foundation1 foundation18' valid4)
apply(simp)

apply(subst conj-comm, rule conjI)
apply(drule-tac  $y = \text{false}$  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 StrictRefEqBoolean)
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 ⊨ v 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 ⊨ v i
shows τ ⊨ (S->including(i)->including(i) ≜ (S->including(i)))
by(simp add: OclIncluding-includes OclIncludes-chn1 assms)

theorem *OclIncluding-idem*[simp,code-unfold]: $((S :: ('a, 'a :: null) Set) \rightarrow \text{including}(i) \rightarrow \text{including}(i)) = (S \rightarrow \text{including}(i))$

proof –

have $A: \bigwedge \tau. \tau \models (i \triangleq \text{invalid}) \implies (S \rightarrow \text{including}(i) \rightarrow \text{including}(i)) \tau = \text{invalid } \tau$
apply(rule foundation22[THEN iffD1])
by(erule StrongEq-L-subst2-rev, simp,simp)
have $A': \bigwedge \tau. \tau \models (i \triangleq \text{invalid}) \implies (S \rightarrow \text{including}(i)) \tau = \text{invalid } \tau$
apply(rule foundation22[THEN iffD1])
by(erule StrongEq-L-subst2-rev, simp,simp)
have $C: \bigwedge \tau. \tau \models (S \triangleq \text{invalid}) \implies (S \rightarrow \text{including}(i) \rightarrow \text{including}(i)) \tau = \text{invalid } \tau$
apply(rule foundation22[THEN iffD1])
by(erule StrongEq-L-subst2-rev, simp,simp)
have $C': \bigwedge \tau. \tau \models (S \triangleq \text{invalid}) \implies (S \rightarrow \text{including}(i)) \tau = \text{invalid } \tau$
apply(rule foundation22[THEN iffD1])
by(erule StrongEq-L-subst2-rev, simp,simp)
have $D: \bigwedge \tau. \tau \models (S \triangleq \text{null}) \implies (S \rightarrow \text{including}(i) \rightarrow \text{including}(i)) \tau = \text{invalid } \tau$
apply(rule foundation22[THEN iffD1])
by(erule StrongEq-L-subst2-rev, simp,simp)
have $D': \bigwedge \tau. \tau \models (S \triangleq \text{null}) \implies (S \rightarrow \text{including}(i)) \tau = \text{invalid } \tau$
apply(rule foundation22[THEN iffD1])
by(erule StrongEq-L-subst2-rev, simp,simp)
show ?thesis
apply(rule ext, rename-tac τ)
apply(case-tac $\tau \models (v \ i)$)
apply(case-tac $\tau \models (\delta \ S)$)
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 $\tau \models \delta \ S$
and $\tau \models v \ i$
shows $\tau \models (S \rightarrow \text{excluding}(i) \rightarrow \text{excluding}(i)) \triangleq (S \rightarrow \text{excluding}(i))$
by(simp add: *OclExcluding-excludes OclExcludes-charn1* assms)

theorem *OclExcluding-idem*[simp,code-unfold]: $((S \rightarrow \text{excluding}(i)) \rightarrow \text{excluding}(i)) = (S \rightarrow \text{excluding}(i))$

proof –

have $A: \bigwedge \tau. \tau \models (i \triangleq \text{invalid}) \implies (S \rightarrow \text{excluding}(i) \rightarrow \text{excluding}(i)) \tau = \text{invalid } \tau$
apply(rule foundation22[THEN iffD1])
by(erule StrongEq-L-subst2-rev, simp,simp)
have $A': \bigwedge \tau. \tau \models (i \triangleq \text{invalid}) \implies (S \rightarrow \text{excluding}(i)) \tau = \text{invalid } \tau$
apply(rule foundation22[THEN iffD1])

```

    by(erule StrongEq-L-subst2-rev, simp,simp)
  have C:  $\bigwedge \tau. \tau \models (S \triangleq \text{invalid}) \implies (S \text{--} \text{>excluding}(i) \text{--} \text{>excluding}(i)) \tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
  have C':  $\bigwedge \tau. \tau \models (S \triangleq \text{invalid}) \implies (S \text{--} \text{>excluding}(i)) \tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
  have D:  $\bigwedge \tau. \tau \models (S \triangleq \text{null}) \implies (S \text{--} \text{>excluding}(i) \text{--} \text{>excluding}(i)) \tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
  have D':  $\bigwedge \tau. \tau \models (S \triangleq \text{null}) \implies (S \text{--} \text{>excluding}(i)) \tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
  show ?thesis
    apply(rule ext, rename-tac  $\tau$ )
    apply(case-tac  $\tau \models (v \ i)$ )
    apply(case-tac  $\tau \models (\delta \ S)$ )
    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]:

$X \text{--} \text{>includes}(X \text{--} \text{>any}()) = (\text{if } \delta \ X \text{ then}$
 $\text{if } \delta \ (X \text{--} \text{>size}()) \text{ then not}(X \text{--} \text{>isEmpty}())$
 $\text{else } X \text{--} \text{>includes}(\text{null}) \text{ endif}$
 $\text{else invalid endif})$

proof –

have *defined-inject-true* : $\bigwedge \tau \ P. (\delta \ P) \tau \neq \text{true } \tau \implies (\delta \ P) \tau = \text{false } \tau$
 apply(simp add: defined-def true-def false-def bot-fun-def bot-option-def
 null-fun-def null-option-def)
 by (case-tac $P \tau = \perp \vee P \tau = \text{null}$, simp-all add: true-def)

have *valid-inject-true* : $\bigwedge \tau \ P. (v \ P) \tau \neq \text{true } \tau \implies (v \ P) \tau = \text{false } \tau$
 apply(simp add: valid-def true-def false-def bot-fun-def bot-option-def
 null-fun-def null-option-def)
 by (case-tac $P \tau = \perp$, simp-all add: true-def)

have *notempty'*: $\bigwedge \tau \ X. \tau \models \delta \ X \implies \text{finite } \llbracket \text{Rep-Set}_{\text{base}}(X \ \tau) \rrbracket \implies \text{not}(X \text{--} \text{>isEmpty}()) \tau \neq \text{true } \tau \implies$
 $X \ \tau = \text{Set}\{\} \ \tau$

 apply(case-tac $X \ \tau$, 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-Setbase-def null-option-def foundation16[THEN iffD1, standard])
apply(case-tac X', simp, metis (hide-lams, no-types) bot-Setbase-def foundation16[THEN iffD1, standard])
apply(rename-tac X'', case-tac X'', simp)
  apply (metis (hide-lams, no-types) foundation16[THEN iffD1, standard] null-Setbase-def)
apply(simp add: OclIsEmpty-def OclSize-def)
apply(subst (asm) cp-OclNot, subst (asm) cp-OclOr, subst (asm) StrictRefEqInteger.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-Setbase-inverse split: split-if-asm)
by(simp add: true-def OclInt0-def OclNot-def StrictRefEqInteger StrongEq-def)

have B:  $\bigwedge X \tau. \neg \text{finite } [\![\text{Rep-Set}_{\text{base}}(X \tau)]\!] \implies (\delta(X \rightarrow \text{size}())) \tau = \text{false } \tau$ 
apply(subst cp-defined)
apply(simp add: OclSize-def)
by (metis bot-fun-def defined-def)

show ?thesis
apply(rule ext, rename-tac  $\tau$ , simp only: OclIncludes-def OclANY-def)
apply(subst cp-OclIf, subst (2) cp-valid)
apply(case-tac ( $\delta X$ )  $\tau = \text{true } \tau$ ,
      simp only: foundation20[simplified OclValid-def] cp-OclIf[symmetric], simp,
      subst (1 2) cp-OclAnd, simp add: cp-OclAnd[symmetric])
apply(case-tac finite  $[\![\text{Rep-Set}_{\text{base}}(X \tau)]\!]$ )
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 ( $X \rightarrow \text{notEmpty}()$ )  $\tau = \text{true } \tau$ , simp)
apply(frule OclNotEmpty-has-elt[simplified OclValid-def], simp)
apply(simp add: OclNotEmpty-def cp-OclIf[symmetric])
apply(subgoal-tac (SOME  $y. y \in [\![\text{Rep-Set}_{\text{base}}(X \tau)]\!] \in [\![\text{Rep-Set}_{\text{base}}(X \tau)]\!]$ , 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  $\neg (\text{null } \tau \in [\![\text{Rep-Set}_{\text{base}}(X \tau)]\!])$ , simp)
apply(subst OclIsEmpty-def, simp add: OclSize-def)
apply(subst cp-OclNot, subst cp-OclOr, subst StrictRefEqInteger.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-Setbase-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 ( $X \rightarrow \text{notEmpty}()$ )  $\tau = \text{true } \tau$ , simp)
apply(frule OclNotEmpty-has-elt[simplified OclValid-def], simp)
apply(simp add: OclNotEmpty-def OclIsEmpty-def)
apply(subgoal-tac  $X \rightarrow \text{size}()$   $\tau = \perp$ )
prefer 2
apply (metis (hide-lams, no-types) OclSize-def)
apply(subst (asm) cp-OclNot, subst (asm) cp-OclOr, subst (asm) StrictRefEqInteger.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 StrictRefEqInteger 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]:  $\delta (\text{Set}\{\} \rightarrow \text{size}()) = \text{true}$ 
by simp

lemma [simp,code-unfold]:  $\delta ((X \rightarrow \text{including}(x)) \rightarrow \text{size}()) = (\delta (X \rightarrow \text{size}()) \text{ and } v(x))$ 
proof –
have defined-inject-true :  $\bigwedge \tau P. (\delta P) \tau \neq \text{true } \tau \implies (\delta P) \tau = \text{false } \tau$ 
  apply(simp add: defined-def true-def false-def bot-fun-def bot-option-def
    null-fun-def null-option-def)
  by (case-tac  $P \tau = \perp \vee P \tau = \text{null}$ , simp-all add: true-def)

have valid-inject-true :  $\bigwedge \tau P. (v P) \tau \neq \text{true } \tau \implies (v P) \tau = \text{false } \tau$ 
  apply(simp add: valid-def true-def false-def bot-fun-def bot-option-def
    null-fun-def null-option-def)
  by (case-tac  $P \tau = \perp$ , simp-all add: true-def)

have OclIncluding-finite-rep-set :  $\bigwedge \tau. (\delta X \text{ and } v x) \tau = \text{true } \tau \implies$ 
  finite  $\llbracket \llbracket \text{Rep-Set}_{\text{base}} (X \rightarrow \text{including}(x) \tau) \rrbracket \rrbracket = \text{finite } \llbracket \llbracket \text{Rep-Set}_{\text{base}} (X \tau) \rrbracket \rrbracket$ 
apply(rule OclIncluding-finite-rep-set)
by(metis OclValid-def foundation5)+

have card-including-exec :  $\bigwedge \tau. (\delta (\lambda \tau. \llbracket \text{int } (\text{card } \llbracket \llbracket \text{Rep-Set}_{\text{base}} (X \rightarrow \text{including}(x) \tau) \rrbracket \rrbracket) \rrbracket) \tau =$ 
   $(\delta (\lambda \tau. \llbracket \text{int } (\text{card } \llbracket \llbracket \text{Rep-Set}_{\text{base}} (X \tau) \rrbracket \rrbracket) \rrbracket) \tau$ 
by(simp add: defined-def bot-fun-def bot-option-def null-fun-def null-option-def)

show ?thesis
apply(rule ext, rename-tac  $\tau$ )
apply(case-tac ( $\delta (X \rightarrow \text{including}(x) \rightarrow \text{size}())$ )  $\tau = \text{true } \tau$ , simp del: OclSize-including-exec)
apply(subst cp-OclAnd, subst cp-defined, simp only: cp-defined[of  $X \rightarrow \text{including}(x) \rightarrow \text{size}()$ ],

```

simp add: OclSize-def
apply(*case-tac* ((δX and νx) $\tau = \text{true } \tau \wedge \text{finite } \llbracket \text{Rep-Set}_{\text{base}} (X \rightarrow \text{including}(x) \tau) \rrbracket$), *simp*)
apply(*erule conjE*,
simp add: OclIncluding-finite-rep-set[simplified OclValid-def] card-including-exec
cp-OclAnd[of $\delta X \nu x$]
cp-OclAnd[of true, THEN sym])
apply(*subgoal-tac* (δX) $\tau = \text{true } \tau \wedge (\nu x) \tau = \text{true } \tau$, *simp*)
apply(*rule foundation5[of - $\delta X \nu x$, 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 $X \rightarrow \text{including}(x) \rightarrow \text{size}()$]*,
simp del: OclSize-including-exec,
simp only: cp-OclAnd[of $\delta (X \rightarrow \text{size}()) \nu x$],
simp add: cp-defined[of $X \rightarrow \text{including}(x) \rightarrow \text{size}()$] cp-defined[of $X \rightarrow \text{size}()$]
del: OclSize-including-exec,
simp add: OclSize-def card-including-exec
del: OclSize-including-exec)
apply(*case-tac* (δX and νx) $\tau = \text{true } \tau \wedge \text{finite } \llbracket \text{Rep-Set}_{\text{base}} (X \tau) \rrbracket$,
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* (νx) $\tau = \text{true } \tau$, *simp add: cp-OclAnd[of $\delta X \nu x$]*)
by(*drule valid-inject-true[of x]*, *simp add: cp-OclAnd[of - νx]*)
qed

lemma [*simp,code-unfold*]: $\delta ((X \rightarrow \text{excluding}(x)) \rightarrow \text{size}()) = (\delta (X \rightarrow \text{size}()) \text{ and } \nu(x))$
proof –
have *defined-inject-true* : $\bigwedge \tau P. (\delta P) \tau \neq \text{true } \tau \implies (\delta P) \tau = \text{false } \tau$
apply(*simp add: defined-def true-def false-def bot-fun-def bot-option-def*
null-fun-def null-option-def)
by (*case-tac* $P \tau = \perp \vee P \tau = \text{null}$, *simp-all add: true-def*)

have *valid-inject-true* : $\bigwedge \tau P. (\nu P) \tau \neq \text{true } \tau \implies (\nu P) \tau = \text{false } \tau$
apply(*simp add: valid-def true-def false-def bot-fun-def bot-option-def*
null-fun-def null-option-def)
by (*case-tac* $P \tau = \perp$, *simp-all add: true-def*)

have *OclExcluding-finite-rep-set* : $\bigwedge \tau. (\delta X \text{ and } \nu x) \tau = \text{true } \tau \implies$
 $\text{finite } \llbracket \text{Rep-Set}_{\text{base}} (X \rightarrow \text{excluding}(x) \tau) \rrbracket =$
 $\text{finite } \llbracket \text{Rep-Set}_{\text{base}} (X \tau) \rrbracket$
apply(*rule OclExcluding-finite-rep-set*)
by(*metis OclValid-def foundation5*) +

have *card-excluding-exec* : $\wedge \tau. (\delta (\lambda \cdot. \llbracket \text{int } (\text{card } \llbracket \llbracket \text{Rep-Set}_{\text{base}} (X \rightarrow \text{excluding}(x) \tau) \rrbracket \rrbracket) \rrbracket) \tau =$
 $(\delta (\lambda \cdot. \llbracket \text{int } (\text{card } \llbracket \llbracket \text{Rep-Set}_{\text{base}} (X \tau) \rrbracket \rrbracket) \rrbracket) \tau$
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* $(\delta (X \rightarrow \text{excluding}(x) \rightarrow \text{size}())) \tau = \text{true } \tau, \text{simp}$)
apply (*subst cp-OclAnd, subst cp-defined, simp only: cp-defined[of X \rightarrow excluding(x) \rightarrow size()],*
simp add: OclSize-def)
apply (*case-tac* $((\delta X \text{ and } \nu x) \tau = \text{true } \tau \wedge \text{finite } \llbracket \llbracket \text{Rep-Set}_{\text{base}} (X \rightarrow \text{excluding}(x) \tau) \rrbracket \rrbracket), \text{simp}$)
apply (*erule conjE,*
simp add: OclExcluding-finite-rep-set[simplified OclValid-def] card-excluding-exec
cp-OclAnd[of $\delta X \nu x$]
cp-OclAnd[of true, THEN sym])
apply (*subgoal-tac* $(\delta X) \tau = \text{true } \tau \wedge (\nu x) \tau = \text{true } \tau, \text{simp}$)
apply (*rule foundation5[of - $\delta X \nu x$, 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 \rightarrow excluding(x) \rightarrow size()],*
simp,
simp only: cp-OclAnd[of $\delta (X \rightarrow \text{size}()) \nu x$],
simp add: cp-defined[of X \rightarrow excluding(x) \rightarrow size()] cp-defined[of X \rightarrow size()],
simp add: OclSize-def card-excluding-exec)
apply (*case-tac* $(\delta X \text{ and } \nu x) \tau = \text{true } \tau \wedge \text{finite } \llbracket \llbracket \text{Rep-Set}_{\text{base}} (X \tau) \rrbracket \rrbracket,$
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* $(\nu x) \tau = \text{true } \tau, \text{simp add: cp-OclAnd[of } \delta X \nu x])$
by (*drule valid-inject-true[of x], simp add: cp-OclAnd[of - νx]*)
qed

lemma [*simp*]:
assumes *X-finite*: $\wedge \tau. \text{finite } \llbracket \llbracket \text{Rep-Set}_{\text{base}} (X \tau) \rrbracket \rrbracket$
shows $\delta ((X \rightarrow \text{including}(x)) \rightarrow \text{size}()) = (\delta(X) \text{ and } \nu(x))$
by (*simp add: size-defined[OF X-finite] del: OclSize-including-exec*)

OclForall

lemma *OclForall-rep-set-false*:
assumes $\tau \models \delta X$
shows $(\text{OclForall } X P \tau = \text{false } \tau) = (\exists x \in \llbracket \llbracket \text{Rep-Set}_{\text{base}} (X \tau) \rrbracket \rrbracket. P (\lambda \tau. x) \tau = \text{false } \tau)$
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 $(\tau \models \text{OclForall } X P) = (\forall x \in [\![\text{Rep-Set}_{\text{base}}(X \tau)]\!]. \tau \models P(\lambda \tau. x))$
proof –
have *destruct-ocl* : $\lambda x \tau. x = \text{true } \tau \vee x = \text{false } \tau \vee x = \text{null } \tau \vee x = \perp \tau$
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* : $\bigwedge P1 P2 P3 P4 R.$
 $(P1 \vee P2 \vee P3 \vee P4) \implies (P1 \implies R) \implies (P2 \implies R) \implies (P3 \implies R) \implies (P4 \implies R) \implies R$
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** $\forall x \in [\![\text{Rep-Set}_{\text{base}}(X \tau)]\!]. P(\lambda \tau. x) \tau \neq [\![\text{None}]\!]$
 $\implies \forall x \in [\![\text{Rep-Set}_{\text{base}}(X \tau)]\!]. \exists y. P(\lambda \tau. x) \tau = [y] \implies$
 $\forall x \in [\![\text{Rep-Set}_{\text{base}}(X \tau)]\!]. P(\lambda \tau. x) \tau \neq [\![\text{False}]\!]$
 $\implies x \in [\![\text{Rep-Set}_{\text{base}}(X \tau)]\!] \implies P(\lambda \tau. x) \tau = [\![\text{True}]\!]$
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 $x\text{-def} : \tau \models \delta x$
and $y\text{-def} : \tau \models \delta y$
shows $(\tau \models \text{OclForall } x (\text{OclIncludes } y)) = ([\![\text{Rep-Set}_{\text{base}}(x \tau)]\!] \subseteq [\![\text{Rep-Set}_{\text{base}}(y \tau)]\!])$
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 $x\text{-def} : \tau \models \delta x$
and $y\text{-def} : \tau \models \delta y$
shows $(\text{OclForall } x (\text{OclIncludes } y) \tau = \text{false } \tau) = (\neg [\![\text{Rep-Set}_{\text{base}}(x \tau)]\!] \subseteq [\![\text{Rep-Set}_{\text{base}}(y \tau)]\!])$
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* $\llbracket \text{Rep-Set}_{\text{base}} (S \ \tau) \rrbracket$

shows $S \rightarrow \text{forAll}(x \mid P \ x) \ \tau = (S \rightarrow \text{iterate}(x; \text{acc} = \text{true} \mid \text{acc} \text{ and } P \ x)) \ \tau$

proof –

have *and-comm* : *comp-fun-commute* $(\lambda x \text{ acc. acc and } P \ x)$

apply (*simp add*: *comp-fun-commute-def comp-def*)

by (*metis OclAnd-assoc OclAnd-commute*)

have *ex-insert* : $\bigwedge x \ F \ P. (\exists x \in \text{insert } x \ F. P \ x) = (P \ x \vee (\exists x \in F. P \ x))$

by (*metis insert-iff*)

have *destruct-ocl* : $\bigwedge x \ \tau. x = \text{true} \ \tau \vee x = \text{false} \ \tau \vee x = \text{null} \ \tau \vee x = \perp \ \tau$

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* : $\bigwedge P1 \ P2 \ P3 \ P4 \ R.$

$(P1 \vee P2 \vee P3 \vee P4) \implies (P1 \implies R) \implies (P2 \implies R) \implies (P3 \implies R) \implies (P4 \implies R) \implies R$

by *metis*

let *?P-eq* = $\lambda x \ b \ \tau. P \ (\lambda \cdot. x) \ \tau = b \ \tau$

let *?P* = $\lambda \text{set} \ b \ \tau. \exists x \in \text{set}. ?P\text{-eq } x \ b \ \tau$

let *?if* = $\lambda f \ b \ c. \text{if } f \ b \ \tau \text{ then } b \ \tau \text{ else } c$

let *?forall* = $\lambda P. ?if \ P \ \text{false} \ (?if \ P \ \text{invalid} \ (?if \ P \ \text{null} \ (\text{true } \tau)))$

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* = $\llbracket \text{Rep-Set}_{\text{base}} (S \ \tau) \rrbracket$ *in*

?forall $(?P \ \text{set}) =$

Finite-Set.fold $(\lambda x \ \text{acc. acc and } P \ x) \ \text{true} \ ((\lambda a \ \tau. a) \ ' \ \text{set}) \ \tau,$

simp only: *Let-def*, *simp add*: *S-finite*, *simp only*: *Let-def*)

apply (*case-tac* $\llbracket \text{Rep-Set}_{\text{base}} (S \ \tau) \rrbracket = \{\}$, *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* $\exists x. x \in F$) **prefer** 2
apply(*metis all-not-in-conv*)
proof – **fix** $x F$ **show** $(\delta S) \tau = \text{true} \tau \implies \exists x. x \in F \implies$
 $?forall (\lambda b \tau. ?P\text{-eq } x b \tau \vee ?P F b \tau) =$
 $((\lambda -. ?forall (?P F)) \text{ and } (\lambda -. P (\lambda \tau. x) \tau)) \tau$
apply(*rule disjE4[OF destruct-ocl[where $x = P (\lambda \tau. x) \tau$]]*)
apply(*simp-all add: true-def false-def invalid-def OclAnd-def*
 $\text{null-fun-def null-option-def bot-fun-def bot-option-def}$)
by (*metis (lifting) option.distinct(1)*) +
apply-end(*simp add: OclValid-def*) +
qed
qed

lemma *OclForall-cong*:
assumes $\bigwedge x. x \in \llbracket \text{Rep-Set}_{base} (X \tau) \rrbracket \implies \tau \models P (\lambda \tau. x) \implies \tau \models Q (\lambda \tau. x)$
assumes $P: \tau \models \text{OclForall } X P$
shows $\tau \models \text{OclForall } X Q$
proof –
have *def-X*: $\tau \models \delta X$
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 $\bigwedge x. x \in \llbracket \text{Rep-Set}_{base} (X \tau) \rrbracket \implies \tau \models P (\lambda \tau. x) \implies \tau \models Q (\lambda \tau. x) \implies \tau \models R (\lambda \tau. x)$
assumes $P: \tau \models \text{OclForall } X P$
assumes $Q: \tau \models \text{OclForall } X Q$
shows $\tau \models \text{OclForall } X R$
proof –
have *def-X*: $\tau \models \delta X$
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 *StrictRefEqSet-defined* :
assumes *x-def*: $\tau \models \delta x$
assumes *y-def*: $\tau \models \delta y$
shows $((x::(\lambda \alpha. \alpha::\text{null})\text{Set}) \doteq y) \tau =$
 $(x \rightarrow \text{forAll}(z \mid y \rightarrow \text{includes}(z)) \text{ and } (y \rightarrow \text{forAll}(z \mid x \rightarrow \text{includes}(z)))) \tau$
proof –

```

have rep-set-inj :  $\wedge \tau. (\delta x) \tau = \text{true} \tau \implies$ 
     $(\delta y) \tau = \text{true} \tau \implies$ 
     $x \tau \neq y \tau \implies$ 
     $\llbracket \text{Rep-Set}_{\text{base}}(y \tau) \rrbracket \neq \llbracket \text{Rep-Set}_{\text{base}}(x \tau) \rrbracket$ 
apply(simp add: defined-def)
apply(split split-if-asm, simp add: false-def true-def)+
apply(simp add: null-fun-def null-Setbase-def bot-fun-def bot-Setbase-def)

apply(case-tac x  $\tau$ , rename-tac x')
apply(case-tac x', simp-all, rename-tac x'')
apply(case-tac x'', simp-all)

apply(case-tac y  $\tau$ , rename-tac y')
apply(case-tac y', simp-all, rename-tac y'')
apply(case-tac y'', simp-all)

apply(simp add: Abs-Setbase-inverse)
by(blast)

show ?thesis
apply(simp add: StrictRefEqSet StrongEq-def
    foundation20[OF x-def, simplified OclValid-def]
    foundation20[OF y-def, simplified OclValid-def])
apply(subgoal-tac  $\llbracket x \tau = y \tau \rrbracket = \text{true} \tau \vee \llbracket x \tau = y \tau \rrbracket = \text{false} \tau$ )
prefer 2
apply(simp add: false-def true-def)

apply(erule disjE)
apply(simp add: true-def)

apply(subgoal-tac  $(\tau \models \text{OclForall } x (\text{OclIncludes } y)) \wedge (\tau \models \text{OclForall } y (\text{OclIncludes } x)))$ 
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  $\text{OclForall } x (\text{OclIncludes } y) \tau = \text{false} \tau \vee$ 
     $\text{OclForall } y (\text{OclIncludes } x) \tau = \text{false} \tau$ )
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 StrictRefEqSet-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 v y
      then false (* x'->includes = null *)
      else invalid
      endif
    endif)
  else if v x (* null = ??? *)
    then if v y then not(δ y) else invalid endif
    else invalid
    endif
  endif)

proof –
have defined-inject-true :  $\bigwedge \tau P. (\neg (\tau \models \delta P)) = ((\delta P) \tau = \text{false } \tau)$ 
by (metis bot-fun-def OclValid-def defined-def foundation16 null-fun-def)

have valid-inject-true :  $\bigwedge \tau P. (\neg (\tau \models v P)) = ((v P) \tau = \text{false } \tau)$ 
by (metis bot-fun-def OclIf-true' OclIncludes-chn0 OclIncludes-chn0' 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: StrictRefEqSet-defined[simplified OclValid-def])
by(simp add: StrictRefEqSet StrongEq-def false-def true-def valid-def defined-def)
qed

lemma StrictRefEqSet-L-subst1 :  $cp P \implies \tau \models v x \implies \tau \models v y \implies \tau \models v P x \implies \tau \models v P y \implies$ 
 $\tau \models (x::('A,'α::null)Set) \doteq y \implies \tau \models (P x::('A,'α::null)Set) \doteq P y$ 
apply(simp only: StrictRefEqSet 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  $\tau \models \delta s \implies \tau \models \delta t \implies \tau \models v x \implies$ 
 $\tau \models ((s::('A,'α::null)Set) \doteq t) \implies \tau \models (s->including(x) \doteq (t->including(x)))$ 
proof –
have cp:  $cp (\lambda s. (s->including(x)))$ 
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  $\tau \models \delta s \implies \tau \models \delta t \implies \tau \models v x \implies \tau \models (s \doteq t) \implies ?thesis$ 

```



```

apply(rule-tac  $P = \lambda s. (s \rightarrow \text{including}(x))$ ) in StrictRefEqSet-L-subst1)
  apply(rule cp)
  apply(simp add: foundation20) apply(simp add: foundation20)
  apply (simp add: foundation10 foundation6)+
done
qed

lemma OclIncluding-cong :  $\bigwedge (s :: ('A, 'a :: \text{null}) \text{Set}) \ t \ x \ y \ \tau. \ \tau \models \delta \ t \implies \tau \models v \ y \implies$ 
   $\tau \models s \doteq t \implies x = y \implies \tau \models s \rightarrow \text{including}(x) \doteq (t \rightarrow \text{including}(y))$ 
apply(simp only:)
apply(rule OclIncluding-cong', simp-all only:)
by(auto simp: OclValid-def OclIf-def invalid-def bot-option-def OclNot-def split : split-if-asm)

lemma const-StrictRefEqSet-empty :  $\text{const } X \implies \text{const } (X \doteq \text{Set}\{\})$ 
apply(rule StrictRefEqSet.const, assumption)
by(simp)

lemma const-StrictRefEqSet-including :
   $\text{const } a \implies \text{const } S \implies \text{const } X \implies \text{const } (X \doteq S \rightarrow \text{including}(a))$ 
apply(rule StrictRefEqSet.const, assumption)
by(rule const-OclIncluding)

```

Test Statements

```

Assert ( $\tau \models (\text{Set}\{\lambda -. \llbracket x \rrbracket\} \doteq \text{Set}\{\lambda -. \llbracket x \rrbracket\})$ )
Assert ( $\tau \models (\text{Set}\{\lambda -. \llbracket x \rrbracket\} \doteq \text{Set}\{\lambda -. \llbracket x \rrbracket\})$ )

end

```

```

theory UML-Sequence
imports ../basic-types/UML-Boolean
  ../basic-types/UML-Integer
begin

```

B.2.9. Collection Type Sequence: Operations

Constants: mtSequence

```

definition mtSequence :: ('A, 'α :: null) Sequence (Sequence{})
where  $\text{Sequence}\{\} \equiv (\lambda \tau. \text{Abs-Sequence}_{\text{base}} [\llbracket \cdot \rrbracket :: 'α \text{ list} \rrbracket])$ 

declare mtSequence-def[code-unfold]

lemma mtSequence-defined[simp,code-unfold]:  $\delta(\text{Sequence}\{\}) = \text{true}$ 
apply(rule ext, auto simp: mtSequence-def defined-def null-Sequencebase-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*]: $\mathbf{v}(\text{Sequence}\{\}) = \text{true}$
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*: $\llbracket \text{Rep-Sequence}_{\text{base}} (\text{Sequence}\{\} \tau) \rrbracket = []$
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-intro''_{Sequence}*[*intro!,simp,code-unfold*] = *cp-intro'*

Properties of Sequence Type: Every element in a defined sequence is valid.

lemma *Sequence-inv-lemma*: $\tau \models (\delta X) \implies \forall x \in \text{set } \llbracket \text{Rep-Sequence}_{\text{base}} (X \tau) \rrbracket. x \neq \text{bot}$
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 τ) = 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 τ) = 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*]:
 $((x::(\mathcal{A}, \alpha::\text{null})\text{Sequence}) \doteq y) \equiv (\lambda \tau. \text{if } (\mathbf{v} x) \tau = \text{true} \wedge (\mathbf{v} y) \tau = \text{true} \tau$
 $\text{then } (x \triangleq y) \tau$
 $\text{else invalid } \tau)$

Property proof in terms of *profile-bin3*

interpretation *StrictRefEq_{Sequence}* : *profile-bin3* $\lambda x y. (x::(\mathcal{A}, \alpha::\text{null})\text{Sequence}) \doteq y$
by *unfold-locales (auto simp: StrictRefEq_{Sequence})*

Standard Operations

Definition: including **definition** $OclIncluding :: (('A, 'a::null) Sequence, ('A, 'a) val] \Rightarrow ('A, 'a) Sequence$

where $OclIncluding\ x\ y = (\lambda\ \tau. \text{if } (\delta\ x)\ \tau = \text{true } \tau \wedge (\nu\ y)\ \tau = \text{true } \tau$
 $\text{then } Abs-Sequence_{base}\ [\llbracket Rep-Sequence_{base}\ (x\ \tau) \rrbracket] \ @\ [y\ \tau]\ \rrbracket]$
 $\text{else } invalid\ \tau)$

notation $OclIncluding\ (->including_{Seq}\ '(-'))$

interpretation $OclIncluding :$

$profile-bin2\ OclIncluding\ \lambda x\ y. Abs-Sequence_{base}\ [\llbracket Rep-Sequence_{base}\ x \rrbracket] \ @\ [y]\ \rrbracket]$

proof –

have $A : \bigwedge x\ y. x \neq bot \implies x \neq null \implies y \neq bot \implies$

$\llbracket \llbracket Rep-Sequence_{base}\ x \rrbracket \ @\ [y] \rrbracket \rrbracket \in \{X. X = bot \vee X = null \vee (\forall x \in set\ \llbracket X \rrbracket. x \neq bot)\}$

by $(auto\ intro!\ :Sequence-inv-lemma[simplified\ OclValid-def]$
 $defined-def\ false-def\ true-def\ null-fun-def\ bot-fun-def])$

show $profile-bin2\ OclIncluding\ (\lambda x\ y. Abs-Sequence_{base}\ [\llbracket Rep-Sequence_{base}\ x \rrbracket] \ @\ [y]\ \rrbracket]$

apply $unfold-locales$

apply $(auto\ simp:OclIncluding-def\ bot-option-def\ null-option-def\ null-Sequence_{base}-def\ bot-Sequence_{base}-def)$

apply $(erule-tac\ Q=Abs-Sequence_{base}\ [\llbracket Rep-Sequence_{base}\ x \rrbracket] \ @\ [y]\ \rrbracket] = Abs-Sequence_{base}\ [None]\ \text{in}\ contrapos-pp)$

apply $(subst\ Abs-Sequence_{base}-inject\ [OF\ A])$

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

apply $(erule-tac\ Q=Abs-Sequence_{base}\ [\llbracket Rep-Sequence_{base}\ x \rrbracket] \ @\ [y]\ \rrbracket] = Abs-Sequence_{base}\ [None]\ \text{in}\ contrapos-pp)$

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

$-OclFinsequence :: args \Rightarrow ('A, 'a::null) Sequence\ (Sequence\{-\})$

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 $(\tau \models (Sequence\{\} \doteq Sequence\{\}))$

Assert $\tau \models (Sequence\{\mathbf{1},invalid,\mathbf{2}\} \triangleq invalid)$

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

```
lemmas cp-intro'' [intro!,simp,code-unfold] =  
  cp-intro'  
  
  cp-intro''Set  
  cp-intro''Sequence
```

MOVE TEXT: Test Statements

```
lemma syntax-test: Set{2,1} = (Set{ } -> including(1) -> including(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{2}  $\doteq$  null) = (false::('A)Boolean)  
shows (tau::('A)st)  $\models$  (Set{Set{2},null} -> includes(null))  
by (simp add: OclIncludes-executeSet H)
```

```
lemma short-cut'[simp,code-unfold]: (8  $\doteq$  6) = false  
apply (rule ext)  
apply (simp add: StrictRefEqInteger StrongEq-def OclInt8-def OclInt6-def  
  true-def false-def invalid-def bot-option-def)  
done
```

```

lemma short-cut''[simp,code-unfold]: ( $2 \doteq 1$ ) = false
apply(rule ext)
apply(simp add: StrictRefEqInteger StrongEq-def OclInt2-def OclInt1-def
  true-def false-def invalid-def bot-option-def)
done
lemma short-cut'''[simp,code-unfold]: ( $1 \doteq 2$ ) = false
apply(rule ext)
apply(simp add: StrictRefEqInteger 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  $\neg (\tau \models v(\text{invalid}::('A, 'a::\text{null}) \text{Set}))$ 
Assert  $\tau \models v(\text{null}::('A, 'a::\text{null}) \text{Set})$ 
Assert  $\neg (\tau \models \delta(\text{null}::('A, 'a::\text{null}) \text{Set}))$ 
Assert  $\tau \models v(\text{Set}\{\})$ 
Assert  $\tau \models v(\text{Set}\{\text{Set}\{2\}, \text{null}\})$ 
Assert  $\tau \models \delta(\text{Set}\{\text{Set}\{2\}, \text{null}\})$ 
Assert  $\tau \models (\text{Set}\{2, 1\} \rightarrow \text{includes}(1))$ 
Assert  $\neg (\tau \models (\text{Set}\{2\} \rightarrow \text{includes}(1)))$ 
Assert  $\neg (\tau \models (\text{Set}\{2, 1\} \rightarrow \text{includes}(\text{null})))$ 
Assert  $\tau \models (\text{Set}\{2, \text{null}\} \rightarrow \text{includes}(\text{null}))$ 
Assert  $\tau \models (\text{Set}\{\text{null}, 2\} \rightarrow \text{includes}(\text{null}))$ 

Assert  $\tau \models ((\text{Set}\{\}) \rightarrow \text{forAll}(z \mid 0 <_{\text{int}} z))$ 

Assert  $\tau \models ((\text{Set}\{2, 1\}) \rightarrow \text{forAll}(z \mid 0 <_{\text{int}} z))$ 
Assert  $\tau \models (0 <_{\text{int}} 2) \text{ and } (0 <_{\text{int}} 1)$ 
Assert  $\neg (\tau \models ((\text{Set}\{2, 1\}) \rightarrow \text{exists}(z \mid z <_{\text{int}} 0)))$ 
Assert  $\neg (\tau \models (\delta(\text{Set}\{2, \text{null}\}) \rightarrow \text{forAll}(z \mid 0 <_{\text{int}} z)))$ 
Assert  $\neg (\tau \models ((\text{Set}\{2, \text{null}\}) \rightarrow \text{forAll}(z \mid 0 <_{\text{int}} z)))$ 
Assert  $\tau \models ((\text{Set}\{2, \text{null}\}) \rightarrow \text{exists}(z \mid 0 <_{\text{int}} z))$ 

Assert  $\neg (\tau \models (\text{Set}\{\text{null}::'a \text{ Boolean}\} \doteq \text{Set}\{\}))$ 
Assert  $\neg (\tau \models (\text{Set}\{\text{null}::'a \text{ Integer}\} \doteq \text{Set}\{\}))$ 

Assert  $\neg (\tau \models (\text{Set}\{\text{true}\} \doteq \text{Set}\{\text{false}\}))$ 
Assert  $\neg (\tau \models (\text{Set}\{\text{true}, \text{true}\} \doteq \text{Set}\{\text{false}\}))$ 
Assert  $\neg (\tau \models (\text{Set}\{2\} \doteq \text{Set}\{1\}))$ 
Assert  $\tau \models (\text{Set}\{2, \text{null}, 2\} \doteq \text{Set}\{\text{null}, 2\})$ 
Assert  $\tau \models (\text{Set}\{1, \text{null}, 2\} <> \text{Set}\{\text{null}, 2\})$ 
Assert  $\tau \models (\text{Set}\{\text{Set}\{2, \text{null}\}\} \doteq \text{Set}\{\text{Set}\{\text{null}, 2\}\})$ 
Assert  $\tau \models (\text{Set}\{\text{Set}\{2, \text{null}\}\} <> \text{Set}\{\text{Set}\{\text{null}, 2\}, \text{null}\})$ 
Assert  $\tau \models (\text{Set}\{\text{null}\} \rightarrow \text{select}(x \mid \text{not } x) \doteq \text{Set}\{\text{null}\})$ 
Assert  $\tau \models (\text{Set}\{\text{null}\} \rightarrow \text{reject}(x \mid \text{not } x) \doteq \text{Set}\{\text{null}\})$ 

```

lemma *const* (Set{Set{2,null}, invalid}) **by**(simp add: const-ss)

end

B.3. Formalization III: UML/OCL constructs: State Operations and Objects

theory *UML-State*
imports *UML-Library*
begin

no-notation *None* (\perp)

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 *StrictRefEqObject* :: ($\mathfrak{A}, 'a::\{\text{object}, \text{null}\}$) $\text{val} \Rightarrow (\mathfrak{A}, 'a)\text{val} \Rightarrow (\mathfrak{A})\text{Boolean}$

where *StrictRefEqObject* $x\ y$
 $\equiv \lambda\ \tau.$ if ($\mathfrak{v}\ x$) $\tau = \text{true}$ $\tau \wedge (\mathfrak{v}\ y)$ $\tau = \text{true}$ τ
 then if $x\ \tau = \text{null} \vee y\ \tau = \text{null}$
 then $\llbracket x\ \tau = \text{null} \wedge y\ \tau = \text{null} \rrbracket$
 else $\llbracket (\text{oid-of } (x\ \tau)) = (\text{oid-of } (y\ \tau)) \rrbracket$
 else *invalid* τ

Strictness and context passing **lemma** *StrictRefEqObject-strict1*[simp,code-unfold] :

(*StrictRefEqObject* x *invalid*) = *invalid*

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

lemma *StrictRefEqObject-strict2*[simp,code-unfold] :

(*StrictRefEqObject* *invalid* x) = *invalid*

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

lemma *cp-StrictRefEqObject*:

(*StrictRefEqObject* $x\ y\ \tau$) = (*StrictRefEqObject* ($\lambda\cdot. x\ \tau$) ($\lambda\cdot. y\ \tau$)) τ

by(auto simp: *StrictRefEqObject-def cp-valid[symmetric]*)

lemmas $cp0\text{-}StrictRefEqObject = cp\text{-}StrictRefEqObject[THEN\ allI[THEN\ allI[THEN\ allI[THEN\ cpI2]],$
 $of\ StrictRefEqObject]]$

lemmas $cp\text{-}intro''[intro!,simp,code\text{-}unfold] =$
 $cp\text{-}intro''$
 $cp\text{-}StrictRefEqObject[THEN\ allI[THEN\ allI[THEN\ allI[THEN\ cpI2]],$
 $of\ StrictRefEqObject]]$

Logic and Algebraic Layer on Object

Validity and Definedness Properties We derive the usual laws on definedness for (generic) object equality:

lemma $StrictRefEqObject\text{-}defargs:$
 $\tau \models (StrictRefEqObject\ x\ (y::('A, 'a::\{null, object\})val)) \implies (\tau \models (v\ x)) \wedge (\tau \models (v\ y))$
by $(simp\ add: StrictRefEqObject\text{-}def\ OclValid\text{-}def\ true\text{-}def\ invalid\text{-}def\ bot\text{-}option\text{-}def$
 $split: bool.split\text{-}asm\ HOL.split\text{-}if\text{-}asm)$

lemma $defined\text{-}StrictRefEqObject\text{-}I:$
assumes $val\text{-}x : \tau \models v\ x$
assumes $val\text{-}x : \tau \models v\ y$
shows $\tau \models \delta\ (StrictRefEqObject\ x\ y)$
apply $(insert\ assms, simp\ add: StrictRefEqObject\text{-}def\ OclValid\text{-}def)$
by $(subst\ cp\text{-}defined, simp\ add: true\text{-}def)$

lemma $StrictRefEqObject\text{-}def\ homo :$
 $\delta(StrictRefEqObject\ x\ (y::('A, 'a::\{null, object\})val)) = ((v\ x)\ and\ (v\ y))$
sorry

Symmetry **lemma** $StrictRefEqObject\text{-}sym :$
assumes $x\text{-}val : \tau \models v\ x$
shows $\tau \models StrictRefEqObject\ x\ x$
by $(simp\ add: StrictRefEqObject\text{-}def\ true\text{-}def\ OclValid\text{-}def$
 $x\text{-}val[simplified\ OclValid\text{-}def])$

Behavior vs StrongEq It remains to clarify the role of the state invariant $inv_{\sigma}(\sigma)$ mentioned above that states the condition that there is a “one-to-one” correspondence between object representations and oid’s: $\forall oid \in \text{dom } \sigma. oid = \text{OidOf}^{\ulcorner \sigma(oid) \urcorner}$. 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 $WFF :: ('A::object)st \Rightarrow bool$
where $WFF\ \tau = ((\forall x \in \text{ran}(\text{heap}(\text{fst } \tau)). \lceil \text{heap}(\text{fst } \tau)\ (oid\text{-of } x) \rceil = x) \wedge$
 $(\forall x \in \text{ran}(\text{heap}(\text{snd } \tau)). \lceil \text{heap}(\text{snd } \tau)\ (oid\text{-of } x) \rceil = x))$

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 *StrictRefEqObject-vs-StrongEq:*

assumes *WFF*: *WFF* τ

and *valid-x*: $\tau \models (v\ x)$

and *valid-y*: $\tau \models (v\ y)$

and *x-present-pre*: $x \in \text{ran}(\text{heap}(\text{fst } \tau))$

and *y-present-pre*: $y \in \text{ran}(\text{heap}(\text{fst } \tau))$

and *x-present-post*: $x \in \text{ran}(\text{heap}(\text{snd } \tau))$

and *y-present-post*: $y \in \text{ran}(\text{heap}(\text{snd } \tau))$

shows $(\tau \models (\text{StrictRefEqObject } x\ y)) = (\tau \models (x \triangleq y))$

apply(*insert WFF valid-x valid-y x-present-pre y-present-pre x-present-post y-present-post*)

apply(*auto simp: StrictRefEqObject-def OclValid-def WFF-def StrongEq-def true-def Ball-def*)

apply(*erule-tac x=x τ in alle', simp-all*)

done

theorem *StrictRefEqObject-vs-StrongEq'*:

assumes *WFF*: *WFF* τ

and *valid-x*: $\tau \models (v\ (x :: ('A::\text{object}, 'a::\{\text{null}, \text{object}\})\text{val})))$

and *valid-y*: $\tau \models (v\ y)$

and *oid-preserve*: $\bigwedge x. x \in \text{ran}(\text{heap}(\text{fst } \tau)) \vee x \in \text{ran}(\text{heap}(\text{snd } \tau)) \implies$

$H\ x \neq \perp \implies \text{oid-of } (H\ x) = \text{oid-of } x$

and *xy-together*: $x\ \tau \in H\ \text{'ran}(\text{heap}(\text{fst } \tau)) \wedge y\ \tau \in H\ \text{'ran}(\text{heap}(\text{fst } \tau)) \vee$

$x\ \tau \in H\ \text{'ran}(\text{heap}(\text{snd } \tau)) \wedge y\ \tau \in H\ \text{'ran}(\text{heap}(\text{snd } \tau))$

shows $(\tau \models (\text{StrictRefEqObject } x\ y)) = (\tau \models (x \triangleq y))$

apply(*insert WFF valid-x valid-y xy-together*)

apply(*simp add: WFF-def*)

apply(*auto simp: StrictRefEqObject-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 $\tau_0 :: ('A)st$
where $\tau_0 \equiv ((heap=Map.empty, assoc = Map.empty),$
 $(heap=Map.empty, assoc = Map.empty))$

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 $OclAllInstances-generic :: (('A::object) st \Rightarrow 'A state) \Rightarrow ('A::object \rightarrow 'A) \Rightarrow$
 $(('A, 'A option option) Set$
where $OclAllInstances-generic\ fst\ snd\ H =$
 $(\lambda \tau. Abs-Set_{base} [\mid Some\ ' (H\ ' ran\ (heap\ (fst\ snd\ \tau))) - \{ None \}]])$

lemma $OclAllInstances-generic-defined: \tau \models \delta\ (OclAllInstances-generic\ pre\ post\ H)$
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 ($rule\ notI, subst\ (asm)\ Abs-Set_{base}-inject, simp,$
 $(rule\ disjI2)+,$
 $metis\ bot-option-def\ option.distinct(I),$
 $(simp\ add: bot-option-def\ null-option-def)+$)
done

lemma $OclAllInstances-generic-init-empty:$
assumes $[simp]: \bigwedge x. pre\ post\ (x, x) = x$
shows $\tau_0 \models OclAllInstances-generic\ pre\ post\ H \triangleq Set\{\}$
by ($simp\ add: StrongEq-def\ OclAllInstances-generic-def\ OclValid-def\ \tau_0-def\ mtSet-def$)

lemma $represented-generic-objects-nonnull:$
assumes $A: \tau \models ((OclAllInstances-generic\ pre\ post\ (H::('A::object \rightarrow 'A))) -> includes(x))$
shows $\tau \models not(x \triangleq null)$
proof –
have $B: \tau \models \delta\ (OclAllInstances-generic\ pre\ post\ H)$
by ($insert\ A[THEN\ foundation6,$
 $simplified\ OclIncludes-defined-args-valid], auto$)
have $C: \tau \models v\ x$
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 Setbase.Abs-Setbase-inverse,
      auto simp: null-fun-def null-option-def bot-option-def)
done
qed

```

```

lemma represented-generic-objects-defined:
assumes A:  $\tau \models ((\text{OclAllInstances-generic pre-post } H :: ('\mathcal{A} :: \text{object} \rightarrow '\alpha))) \rightarrow \text{includes}(x)$ 
shows  $\tau \models \delta (\text{OclAllInstances-generic pre-post } H) \wedge \tau \models \delta x$ 
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-nonnul])
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:  $\tau \models (\text{OclAllInstances-generic pre-post } H) \rightarrow \text{includes}(x)$ 
shows  $x \tau \in (\text{Some } o \ H) \text{ ' ran } (\text{heap}(\text{pre-post } \tau))$ 
proof –
  have B:  $(\delta (\text{OclAllInstances-generic pre-post } H)) \tau = \text{true } \tau$ 
    by(simp add: OclValid-def[symmetric] OclAllInstances-generic-defined)
  have C:  $(\vee x) \tau = \text{true } \tau$ 
    by(insert A[THEN foundation6,
      simplified OclIncludes-defined-args-valid],
      auto simp: OclValid-def)
  have F:  $\text{Rep-Set}_{\text{base}} (\text{Abs-Set}_{\text{base}} \llbracket \text{Some } ' (H \text{ ' ran } (\text{heap} (\text{pre-post } \tau)) - \{\text{None}\}) \rrbracket) =$ 
     $\llbracket \text{Some } ' (H \text{ ' ran } (\text{heap} (\text{pre-post } \tau)) - \{\text{None}\}) \rrbracket$ 
    by(subst Setbase.Abs-Setbase-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]:  $\bigwedge a. \text{pre-post } (mk \ a) = a$ 
shows  $(mk (\text{heap}=\text{empty}, \text{assocs}=A)) \models \text{OclAllInstances-generic pre-post Type} \doteq \text{Set}\{\}$ 
proof –
  have state-update-vs-allInstances-empty:
     $(\text{OclAllInstances-generic pre-post Type}) (mk (\text{heap}=\text{empty}, \text{assocs}=A)) =$ 
     $\text{Set}\{\} (mk (\text{heap}=\text{empty}, \text{assocs}=A))$ 

```

```

by(simp add: OclAllInstances-generic-def mtSet-def)
show ?thesis
apply(simp only: OclValid-def, subst StrictRefEqSet.cp0,
  simp only: state-update-vs-allInstances-empty StrictRefEqSet.refl-ext)
apply(simp add: OclIf-def valid-def mtSet-def defined-def
  bot-fun-def null-fun-def null-option-def bot-Setbase-def)
by(subst Abs-Setbase-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]:  $\wedge a. \text{pre-post } (mk \ a) = a$ 
assumes  $\wedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$ 
and  $\text{Type Object} \neq \text{None}$ 
shows ( $\text{OclAllInstances-generic pre-post Type}$ )
  ( $mk \ (\text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A)$ )
  =
  ( $(\text{OclAllInstances-generic pre-post Type}) \text{--> including}(\lambda \_. \ll \text{drop } (\text{Type Object}) \rr)$ )
  ( $mk \ (\text{heap} = \sigma', \text{assocs} = A)$ )

```

```

proof –
have  $\text{drop-none} : \wedge x. x \neq \text{None} \implies \ll x \rr = x$ 
by(case-tac x, simp+)

have  $\text{insert-diff} : \wedge x \ S. \text{insert } \ll x \rr \ (S - \{\text{None}\}) = (\text{insert } \ll x \rr \ S) - \{\text{None}\}$ 
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-Setbase-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  $\text{ran } (\sigma'(\text{oid} \mapsto \text{Object})) = \text{insert Object } (\text{ran } \sigma')$ , simp)
apply(case-tac  $\neg (\exists x. \sigma' \text{ oid} = \text{Some } x)$ )
apply(rule ran-map-upd, simp)
apply(simp, erule exE, frule assms, simp)
apply(subgoal-tac  $\text{Object} \in \text{ran } \sigma'$ ) 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]: $\bigwedge a. \text{pre-post } (mk\ a) = a$
assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and *Type Object* $\neq \text{None}$
shows (*OclAllInstances-generic pre-post Type*)
 $(mk\ (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$
 $=$
 $((\lambda -. (\text{OclAllInstances-generic pre-post Type})$
 $(mk\ (\text{heap}=\sigma', \text{assocs}=A))) \rightarrow \text{including}(\lambda -. [\text{drop } (\text{Type Object})]])$
 $(mk\ (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$
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]: $\bigwedge a. \text{pre-post } (mk\ a) = a$
assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and *Type Object* = *None*
shows (*OclAllInstances-generic pre-post Type*)
 $(mk\ (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$
 $=$
 $(\text{OclAllInstances-generic pre-post Type})$
 $(mk\ (\text{heap}=\sigma', \text{assocs}=A))$
proof –
have *drop-none* : $\bigwedge x. x \neq \text{None} \implies [\text{drop } x] = x$
by(case-tac *x*, simp +)
have *insert-diff* : $\bigwedge x\ S. \text{insert } [\text{drop } x] (S - \{\text{None}\}) = (\text{insert } [\text{drop } x] S) - \{\text{None}\}$
by (metis *insert-Diff-if option.distinct(I) singletonE*)

show ?thesis

apply(simp add: *OclIncluding-def OclAllInstances-generic-defined[simplified OclValid-def]*
OclAllInstances-generic-def)
apply(subgoal-tac $\text{ran } (\sigma'(\text{oid} \mapsto \text{Object})) = \text{insert } \text{Object } (\text{ran } \sigma')$, simp add: *assms*)
apply(case-tac $\neg (\exists x. \sigma' \text{ oid} = \text{Some } x)$)
apply(rule *ran-map-upd*, simp)
apply(simp, erule *exE*, frule *assms*, simp)

apply(*subgoal-tac* *Object* $\in \text{ran } \sigma'$) **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*]: $\bigwedge a. \text{pre-post } (\text{mk } a) = a$
assumes *oid-def*: $\text{oid} \notin \text{dom } \sigma'$
and *non-type-conform*: *Type* *Object* = *None*
and *cp-ctxt*: $\text{cp } P$
and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P X)$
shows ($\text{mk } (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A) \models P (\text{OclAllInstances-generic pre-post Type})$) =
 $(\text{mk } (\text{heap}=\sigma', \text{assocs}=A) \models P (\text{OclAllInstances-generic pre-post Type}))$
 $(\text{is } (? \tau \models P ? \varphi) = (? \tau' \models P ? \varphi))$
proof –
have *P-cp* : $\bigwedge x \tau. P x \tau = P (\lambda -. x \tau) \tau$
by (*metis (full-types) cp-ctxt cp-def*)
have *A* : $\text{const } (P (\lambda -. ? \varphi ? \tau))$
by (*simp add: const-ctxt const-ss*)
have $(? \tau \models P ? \varphi) = (? \tau \models \lambda -. P ? \varphi ? \tau)$
by (*subst foundation23, rule refl*)
also have $\dots = (? \tau \models \lambda -. P (\lambda -. ? \varphi ? \tau) ? \tau)$
by (*subst P-cp, rule refl*)
also have $\dots = (? \tau' \models \lambda -. P (\lambda -. ? \varphi ? \tau) ? \tau')$
apply (*simp add: OclValid-def*)
by (*subst A[simplified const-def], subst const-true[simplified const-def], simp*)
finally have *X*: $(? \tau \models P ? \varphi) = (? \tau' \models \lambda -. P (\lambda -. ? \varphi ? \tau) ? \tau')$
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*]: $\bigwedge a. \text{pre-post } (\text{mk } a) = a$
assumes *oid-def*: $\text{oid} \notin \text{dom } \sigma'$
and *type-conform*: *Type* *Object* \neq *None*
and *cp-ctxt*: $\text{cp } P$
and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P X)$
shows ($\text{mk } (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A) \models P (\text{OclAllInstances-generic pre-post Type})$) =
 $(\text{mk } (\text{heap}=\sigma', \text{assocs}=A) \models P ((\text{OclAllInstances-generic pre-post Type})$
 $\quad \rightarrow \text{including } (\lambda -. \lfloor (\text{Type } \text{Object}) \rfloor)))$
 $(\text{is } (? \tau \models P ? \varphi) = (? \tau' \models P ? \varphi'))$
proof –

```

have  $P\text{-}cp : \bigwedge x \tau. P\ x\ \tau = P\ (\lambda\cdot. x\ \tau)\ \tau$ 
  by (metis (full-types) cp-ctxt cp-def)
have  $A : \text{const}\ (P\ (\lambda\cdot. ?\varphi\ ?\tau))$ 
  by (simp add: const-ctxt const-ss)
have  $(?\tau \models P\ ?\varphi) = (?\tau \models \lambda\cdot. P\ ?\varphi\ ?\tau)$ 
  by (subst foundation23, rule refl)
also have  $\dots = (?\tau \models \lambda\cdot. P\ (\lambda\cdot. ?\varphi\ ?\tau)\ ?\tau)$ 
  by (subst P-cp, rule refl)
also have  $\dots = (?\tau' \models \lambda\cdot. P\ (\lambda\cdot. ?\varphi\ ?\tau)\ ?\tau')$ 
  apply (simp add: OclValid-def)
  by (subst A[simplified const-def], subst const-true[simplified const-def], simp)
finally have  $X: (?\tau \models P\ ?\varphi) = (?\tau' \models \lambda\cdot. P\ (\lambda\cdot. ?\varphi\ ?\tau)\ ?\tau')$ 
  by simp
let  $?allInstances = OclAllInstances\text{-}generic\ pre\text{-}post\ Type$ 
have  $?allInstances\ ?\tau = \lambda\cdot. ?allInstances\ ?\tau'\text{-}>including(\lambda\cdot. \llbracket Type\ Object \rrbracket)\ ?\tau$ 
  apply (rule state-update-vs-allInstances-generic-including)
  by (insert oid-def, auto simp: type-conform)
also have  $\dots = ((\lambda\cdot. ?allInstances\ ?\tau')\text{-}>including(\lambda\cdot. (\lambda\cdot. \llbracket Type\ Object \rrbracket)\ ?\tau')\ ?\tau')$ 
  by (subst const-OclIncluding[simplified const-def], simp+)
also have  $\dots = (?allInstances\text{-}>including(\lambda\cdot. \llbracket Type\ Object \rrbracket)\ ?\tau')$ 
  apply (subst cp-OclIncluding[symmetric])
  by (insert type-conform, auto)
finally have  $Y : ?allInstances\ ?\tau = (?allInstances\text{-}>including(\lambda\cdot. \llbracket Type\ Object \rrbracket)\ ?\tau')$ 
  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 ::  $(\mathcal{A} :: object \rightarrow 'a) \Rightarrow (\mathcal{A}, 'a\ option\ option)\ Set$ 
   $(-.\ allInstances'('))$ 
where OclAllInstances-at-post = OclAllInstances-generic snd

lemma OclAllInstances-at-post-defined:  $\tau \models \delta\ (H.\ allInstances())$ 
unfolding OclAllInstances-at-post-def
by (rule OclAllInstances-generic-defined)

lemma  $\tau_0 \models H.\ allInstances() \triangleq Set\{\}$ 
unfolding OclAllInstances-at-post-def
by (rule OclAllInstances-generic-init-empty, simp)

lemma represented-at-post-objects-nonnull:
assumes  $A: \tau \models (((H::(\mathcal{A}::object \rightarrow 'a)).allInstances())\text{-}>includes(x))$ 

```

shows $\tau \models \text{not}(x \triangleq \text{null})$
by(rule *represented-generic-objects-nonnul*[OF A[simplified *OclAllInstances-at-post-def*]])

lemma *represented-at-post-objects-defined*:

assumes A: $\tau \models (((H::(\mathcal{A}::\text{object} \rightarrow \alpha)).\text{allInstances}()) \rightarrow \text{includes}(x))$

shows $\tau \models \delta (H.\text{allInstances}()) \wedge \tau \models \delta x$

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: $\tau \models H.\text{allInstances}() \rightarrow \text{includes}(x)$

shows $x \tau \in (\text{Some } o \ H) \cdot \text{ran } (\text{heap}(\text{snd } \tau))$

by(rule *represented-generic-objects-in-state*[OF A[simplified *OclAllInstances-at-post-def*]])

lemma *state-update-vs-allInstances-at-post-empty*:

shows $(\sigma, (\text{heap}=\text{empty}, \text{assocs}=A)) \models \text{Type}.\text{allInstances}() \doteq \text{Set}\{\}$

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 $\bigwedge x. \sigma' \text{oid} = \text{Some } x \implies x = \text{Object}$

and $\text{Type } \text{Object} \neq \text{None}$

shows $(\text{Type}.\text{allInstances}())$

$(\sigma, (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$

$=$

$((\text{Type}.\text{allInstances}()) \rightarrow \text{including}(\lambda -. [\text{drop } (\text{Type } \text{Object})]))$

$(\sigma, (\text{heap}=\sigma', \text{assocs}=A))$

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 $\bigwedge x. \sigma' \text{oid} = \text{Some } x \implies x = \text{Object}$

and $\text{Type } \text{Object} \neq \text{None}$

shows $(\text{Type}.\text{allInstances}())$

$(\sigma, (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$

$=$

$(\lambda -. (\text{Type}.\text{allInstances}()))$

$(\sigma, (\text{heap}=\sigma', \text{assocs}=A)) \rightarrow \text{including}(\lambda -. [\text{drop } (\text{Type } \text{Object})]))$

$(\sigma, (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$

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 $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$

and *Type Object = None*

shows (*Type .allInstances()*)

($\sigma, \lfloor \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \rfloor$)

=

(*Type .allInstances()*)

($\sigma, \lfloor \text{heap} = \sigma', \text{assocs} = A \rfloor$)

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*: $\text{oid} \notin \text{dom } \sigma'$

and *non-type-conform*: *Type Object = None*

and *cp-ctxt*: *cp P*

and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P X)$

shows (($\sigma, \lfloor \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \rfloor$) $\models (P(\text{Type .allInstances()}))$) =

(($\sigma, \lfloor \text{heap} = \sigma', \text{assocs} = A \rfloor$) $\models (P(\text{Type .allInstances()}))$)

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*: $\text{oid} \notin \text{dom } \sigma'$

and *type-conform*: *Type Object \neq None*

and *cp-ctxt*: *cp P*

and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P X)$

shows (($\sigma, \lfloor \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \rfloor$) $\models (P(\text{Type .allInstances()}))$) =

(($\sigma, \lfloor \text{heap} = \sigma', \text{assocs} = A \rfloor$) $\models (P((\text{Type .allInstances()}))$
 $\rightarrow \text{including}(\lambda -. \lfloor (\text{Type Object}) \rfloor))$)

unfolding *OclAllInstances-at-post-def*

by(rule state-update-vs-allInstances-generic-tc[*OF snd-conv*], insert assms)

OclAllInstances (@pre) **definition** *OclAllInstances-at-pre* :: ($\mathfrak{A} :: \text{object} \rightarrow \alpha$) \Rightarrow ($\mathfrak{A}, \alpha \text{ option option}$) *Set*
 $(- . \text{allInstances@pre}'())$

where *OclAllInstances-at-pre* = *OclAllInstances-generic fst*

lemma *OclAllInstances-at-pre-defined*: $\tau \models \delta (H . \text{allInstances@pre}())$

unfolding *OclAllInstances-at-pre-def*

by(rule *OclAllInstances-generic-defined*)

lemma $\tau_0 \models H . \text{allInstances@pre}() \triangleq \text{Set}\{\}$

unfolding *OclAllInstances-at-pre-def*

by(rule *OclAllInstances-generic-init-empty, simp*)

lemma *represented-at-pre-objects-nonnull*:
assumes $A: \tau \models (((H::('A::object \rightarrow 'a)).allInstances@pre()) \rightarrow includes(x))$
shows $\tau \models not(x \triangleq null)$
by(rule *represented-generic-objects-nonnull*[OF $A[simplified\ OclAllInstances-at-pre-def]$])

lemma *represented-at-pre-objects-defined*:
assumes $A: \tau \models (((H::('A::object \rightarrow 'a)).allInstances@pre()) \rightarrow includes(x))$
shows $\tau \models \delta (H.allInstances@pre()) \wedge \tau \models \delta x$
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 $A: \tau \models H.allInstances@pre() \rightarrow includes(x)$
shows $x \tau \in (Some\ o\ H) \cdot ran\ (heap(fst\ \tau))$
by(rule *represented-generic-objects-in-state*[OF $A[simplified\ OclAllInstances-at-pre-def]$])

lemma *state-update-vs-allInstances-at-pre-empty*:
shows $((\emptyset heap = empty, assoc = A), \sigma) \models Type.allInstances@pre() \doteq Set\{\}$
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 $\wedge x. \sigma' oid = Some\ x \implies x = Object$
and $Type\ Object \neq None$
shows $(Type.allInstances@pre())$
 $((\emptyset heap = \sigma'(oid \mapsto Object), assoc = A), \sigma)$
 $=$
 $((Type.allInstances@pre()) \rightarrow including(\lambda -. \ll drop\ (Type\ Object) \rr))$
 $((\emptyset heap = \sigma', assoc = A), \sigma)$
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 $\wedge x. \sigma' oid = Some\ x \implies x = Object$
and $Type\ Object \neq None$
shows $(Type.allInstances@pre())$
 $((\emptyset heap = \sigma'(oid \mapsto Object), assoc = A), \sigma)$

=
 ((λ -. (Type .allInstances@pre()
 ((\downarrow heap= σ' , assocs=A), σ)) \rightarrow including(λ -. [\downarrow drop (Type Object)])))
 ((\downarrow heap= σ' (oid \rightarrow Object), assocs=A), σ)
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 $\wedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and Type Object = None
shows (Type .allInstances@pre()
 ((\downarrow heap= σ' (oid \rightarrow Object), assocs=A), σ)
 =
 (Type .allInstances@pre()
 ((\downarrow heap= σ' , assocs=A), σ)
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: oid \notin dom σ'
and non-type-conform: Type Object = None
and cp-ctxt: cp P
and const-ctxt: $\wedge X. \text{const } X \implies \text{const } (P X)$
shows (((\downarrow heap= σ' (oid \rightarrow Object), assocs=A), σ) \models (P(Type .allInstances@pre())))) =
 (((\downarrow heap= σ' , assocs=A), σ) \models (P(Type .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: oid \notin dom σ'
and type-conform: Type Object \neq None
and cp-ctxt: cp P
and const-ctxt: $\wedge X. \text{const } X \implies \text{const } (P X)$
shows (((\downarrow heap= σ' (oid \rightarrow Object), assocs=A), σ) \models (P(Type .allInstances@pre())))) =
 (((\downarrow heap= σ' , assocs=A), σ) \models (P((Type .allInstances@pre()
 \rightarrow including(λ -. [(Type Object)]))))))
unfolding OclAllInstances-at-pre-def
by(rule state-update-vs-allInstances-generic-tc[OF fst-conv], insert assms)

@post or @pre theorem StrictRefEqObject-vs-StrongEq'':
assumes WFF: WFF τ
and valid-x: $\tau \models (v (x :: ('A::\text{object}, 'A::\text{object option option})\text{val}))$
and valid-y: $\tau \models (v y)$
and oid-preserve: $\wedge x. x \in \text{ran } (\text{heap}(\text{fst } \tau)) \vee x \in \text{ran } (\text{heap}(\text{snd } \tau)) \implies$
 oid-of (H x) = oid-of x
and xy-together: $\tau \models ((H .\text{allInstances}()) \rightarrow \text{includes}(x) \text{ and } H .\text{allInstances}() \rightarrow \text{includes}(y)) \text{ or}$

$(H .allInstances@pre() \rightarrow includes(x) \text{ and } H .allInstances@pre() \rightarrow includes(y))$

shows $(\tau \models (StrictRefEqObject\ x\ y)) = (\tau \models (x \triangleq y))$

proof –

have *at-post-def* : $\bigwedge x. \tau \models v\ x \implies \tau \models \delta\ (H .allInstances() \rightarrow includes(x))$

apply(*simp add: OclIncludes-def OclValid-def*
OclAllInstances-at-post-defined[simplified OclValid-def])

by(*subst cp-defined, simp*)

have *at-pre-def* : $\bigwedge x. \tau \models v\ x \implies \tau \models \delta\ (H .allInstances@pre() \rightarrow includes(x))$

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}* $\llbracket \text{Some } (H \text{ ' ran } (heap\ (fst\ \tau)) - \{None\}) \rrbracket$) =
 $\llbracket \text{Some } (H \text{ ' ran } (heap\ (fst\ \tau)) - \{None\}) \rrbracket$

by(*subst Set_{base}.Abs-Set_{base}-inverse, simp-all add: bot-option-def*)

have *F'*: *Rep-Set_{base}* (*Abs-Set_{base}* $\llbracket \text{Some } (H \text{ ' ran } (heap\ (snd\ \tau)) - \{None\}) \rrbracket$) =
 $\llbracket \text{Some } (H \text{ ' ran } (heap\ (snd\ \tau)) - \{None\}) \rrbracket$

by(*subst Set_{base}.Abs-Set_{base}-inverse, simp-all add: bot-option-def*)

show ?thesis

apply(*rule StrictRefEqObject-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*:: ($\mathcal{A}, ' \alpha :: \{null, object\}$) *val* \Rightarrow (\mathcal{A}) *Boolean* $((-).oclIsNew'('))$

where *X .oclIsNew*() $\equiv (\lambda \tau . \text{if } (\delta\ X)\ \tau = \text{true } \tau$
 $\text{then } \llbracket \text{oid-of } (X\ \tau) \notin \text{dom}(heap(fst\ \tau)) \wedge$
 $\text{oid-of } (X\ \tau) \in \text{dom}(heap(snd\ \tau)) \rrbracket$
 $\text{else invalid } \tau$)

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*:: ($\mathcal{A}, ' \alpha :: \{null, object\}$) *val* \Rightarrow (\mathcal{A}) *Boolean* $((-).oclIsDeleted'('))$

where *X .oclIsDeleted*() $\equiv (\lambda \tau . \text{if } (\delta\ X)\ \tau = \text{true } \tau$
 $\text{then } \llbracket \text{oid-of } (X\ \tau) \in \text{dom}(heap(fst\ \tau)) \wedge$
 $\text{oid-of } (X\ \tau) \notin \text{dom}(heap(snd\ \tau)) \rrbracket$

else invalid τ)

definition $OclIsMaintained:: ('A, 'a::\{null,object\})val \Rightarrow ('A)Boolean((-).oclIsMaintained'())$

where $X.oclIsMaintained() \equiv (\lambda \tau . \text{if } (\delta X) \tau = \text{true } \tau$
 then $\llbracket \text{oid-of } (X \tau) \in \text{dom}(\text{heap}(\text{fst } \tau)) \wedge$
 $\text{oid-of } (X \tau) \in \text{dom}(\text{heap}(\text{snd } \tau)) \rrbracket$
 else invalid τ)

definition $OclIsAbsent:: ('A, 'a::\{null,object\})val \Rightarrow ('A)Boolean((-).oclIsAbsent'())$

where $X.oclIsAbsent() \equiv (\lambda \tau . \text{if } (\delta X) \tau = \text{true } \tau$
 then $\llbracket \text{oid-of } (X \tau) \notin \text{dom}(\text{heap}(\text{fst } \tau)) \wedge$
 $\text{oid-of } (X \tau) \notin \text{dom}(\text{heap}(\text{snd } \tau)) \rrbracket$
 else invalid τ)

lemma $\text{state-split} : \tau \models \delta X \implies$

$\tau \models (X.oclIsNew()) \vee \tau \models (X.oclIsDeleted()) \vee$
 $\tau \models (X.oclIsMaintained()) \vee \tau \models (X.oclIsAbsent())$

by(simp add: $OclIsDeleted\text{-}def$ $OclIsNew\text{-}def$ $OclIsMaintained\text{-}def$ $OclIsAbsent\text{-}def$
 $OclValid\text{-}def$ $\text{true}\text{-}def$, blast)

lemma $\text{notNew-vs-others} : \tau \models \delta X \implies$

$(\neg \tau \models (X.oclIsNew())) = (\tau \models (X.oclIsDeleted()) \vee$
 $\tau \models (X.oclIsMaintained()) \vee \tau \models (X.oclIsAbsent()))$

by(simp add: $OclIsDeleted\text{-}def$ $OclIsNew\text{-}def$ $OclIsMaintained\text{-}def$ $OclIsAbsent\text{-}def$
 $OclNot\text{-}def$ $OclValid\text{-}def$ $\text{true}\text{-}def$, 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 $OclIsModifiedOnly:: ('A::object, 'a::\{null,object\})Set \Rightarrow 'A Boolean$

$(-->oclIsModifiedOnly'())$

where $X->oclIsModifiedOnly() \equiv (\lambda (\sigma, \sigma').$

 let $X' = (\text{oid-of } ' \llbracket \text{Rep-Set}_{\text{base}}(X(\sigma, \sigma')) \rrbracket \rrbracket);$

$S = ((\text{dom } (\text{heap } \sigma) \cap \text{dom } (\text{heap } \sigma')) - X')$

 in if $(\delta X) (\sigma, \sigma') = \text{true } (\sigma, \sigma') \wedge (\forall x \in \llbracket \text{Rep-Set}_{\text{base}}(X(\sigma, \sigma')) \rrbracket . x \neq \text{null})$

 then $\llbracket \forall x \in S. (\text{heap } \sigma) x = (\text{heap } \sigma') x \rrbracket$

 else invalid (σ, σ'))

Execution with Invalid or Null or Null Element as Argument **lemma** $\text{invalid} -> oclIsModifiedOnly() = \text{invalid}$

by(simp add: $OclIsModifiedOnly\text{-}def$)

lemma $\text{null} -> oclIsModifiedOnly() = \text{invalid}$

by(simp add: $OclIsModifiedOnly\text{-}def$)

lemma**assumes** $X\text{-null} : \tau \models X \rightarrow \text{includes}(\text{null})$ **shows** $\tau \models X \rightarrow \text{oclIsModifiedOnly}() \triangleq \text{invalid}$ **apply**(insert $X\text{-null}$, $\text{simp add} : \text{OclIncludes-def OclIsModifiedOnly-def StrongEq-def OclValid-def true-def}$)**apply**(case-tac τ , simp)**apply**(simp split: split-if-asm)**by**(simp add: null-fun-def, blast)

Context Passing lemma $\text{cp-OclIsModifiedOnly} : X \rightarrow \text{oclIsModifiedOnly}() \tau = (\lambda \cdot. X \tau) \rightarrow \text{oclIsModifiedOnly}() \tau$
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]: $\text{OclSelf } x \text{ } H \text{ fst-snd} = (\lambda \tau . \text{if } (\delta x) \tau = \text{true } \tau$
 $\text{then if oid-of } (x \tau) \in \text{dom}(\text{heap}(\text{fst } \tau)) \wedge \text{oid-of } (x \tau) \in \text{dom}(\text{heap}(\text{snd } \tau))$
 $\text{then } H \upharpoonright (\text{heap}(\text{fst-snd } \tau))(\text{oid-of } (x \tau)) \upharpoonright$
 $\text{else invalid } \tau$
 $\text{else invalid } \tau)$

definition $\text{OclSelf-at-pre} :: ('A :: \text{object}, 'a :: \{\text{null}, \text{object}\}) \text{val} \Rightarrow$
 $(^A \Rightarrow ^a) \Rightarrow$
 $(^A :: \text{object}, 'a :: \{\text{null}, \text{object}\}) \text{val } ((-)@pre(-))$

where $x @pre H = \text{OclSelf } x \text{ } H \text{ fst}$

definition $\text{OclSelf-at-post} :: ('A :: \text{object}, 'a :: \{\text{null}, \text{object}\}) \text{val} \Rightarrow$
 $(^A \Rightarrow ^a) \Rightarrow$
 $(^A :: \text{object}, 'a :: \{\text{null}, \text{object}\}) \text{val } ((-)@post(-))$

where $x @post H = \text{OclSelf } x \text{ } H \text{ snd}$ **Framing Theorem****lemma** all-oid-diff:**assumes** $\text{def-}x : \tau \models \delta x$ **assumes** $\text{def-}X : \tau \models \delta X$ **assumes** $\text{def-}X' : \bigwedge x. x \in \llbracket \text{Rep-Set}_{\text{base}}(X \tau) \rrbracket \implies x \neq \text{null}$ **defines** $P \equiv (\lambda a. \text{not } (\text{StrictRefEqObject } x a))$ **shows** $(\tau \models X \rightarrow \text{forAll}(a \mid P a)) = (\text{oid-of } (x \tau) \notin \text{oid-of } \llbracket \text{Rep-Set}_{\text{base}}(X \tau) \rrbracket)$ **proof** –**have** $P\text{-null-bot} : \bigwedge b. b = \text{null} \vee b = \perp \implies$ $\neg (\exists x \in \llbracket \text{Rep-Set}_{\text{base}}(X \tau) \rrbracket. P (\lambda (- :: 'a \text{ state} \times 'a \text{ state}). x) \tau = b \tau)$ **apply**(erule disjE)**apply**(simp, rule ballI,

simp add: P-def StrictRefEqObject-def, rename-tac x',
subst cp-OclNot, simp,
subgoal-tac x $\tau \neq \text{null} \wedge x' \neq \text{null}$, 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 : $\wedge x y. ((\text{not } x) \tau = (\text{not } y) \tau) = (x \tau = y \tau)$
by (metis foundation21 foundation22)

have P-false : $\exists x \in [\text{Rep-Set}_{\text{base}}(X \tau)]. P(\lambda \cdot. x) \tau = \text{false} \tau \implies$
 $\text{oid-of}(x \tau) \in \text{oid-of}' [\text{Rep-Set}_{\text{base}}(X \tau)]$
apply(erule bexE, rename-tac x')
apply(simp add: P-def)
apply(simp only: OclNot3[symmetric], simp only: not-inj)
apply(simp add: StrictRefEqObject-def split: split-if-asm)
apply(subgoal-tac x $\tau \neq \text{null} \wedge x' \neq \text{null}$, 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 : $\forall x \in [\text{Rep-Set}_{\text{base}}(X \tau)]. P(\lambda \cdot. x) \tau = \text{true} \tau \implies$
 $\text{oid-of}(x \tau) \notin \text{oid-of}' [\text{Rep-Set}_{\text{base}}(X \tau)]$
apply(subgoal-tac $\forall x' \in [\text{Rep-Set}_{\text{base}}(X \tau)]. \text{oid-of } x' \neq \text{oid-of}(x \tau)$)
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: StrictRefEqObject-def false-def split: split-if-asm)
apply(subgoal-tac x $\tau \neq \text{null} \wedge x' \neq \text{null}$, simp)
apply (metis def-X' def-x foundation16[THEN iffD1])
by(simp add: invalid-def bot-option-def false-def)+

have bool-split : $\forall x \in [\text{Rep-Set}_{\text{base}}(X \tau)]. P(\lambda \cdot. x) \tau \neq \text{null} \tau \implies$
 $\forall x \in [\text{Rep-Set}_{\text{base}}(X \tau)]. P(\lambda \cdot. x) \tau \neq \perp \tau \implies$
 $\forall x \in [\text{Rep-Set}_{\text{base}}(X \tau)]. P(\lambda \cdot. x) \tau \neq \text{false} \tau \implies$
 $\forall x \in [\text{Rep-Set}_{\text{base}}(X \tau)]. P(\lambda \cdot. x) \tau = \text{true} \tau$
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: $\tau \models (X \rightarrow \text{excluding}(x)) \rightarrow \text{oclIsModifiedOnly}()$
and oid-is-typerepr : $\tau \models X \rightarrow \text{forAll}(a \mid \text{not } (\text{StrictRefEqObject } x \ a))$
shows $\tau \models (x \text{ @pre } P \triangleq (x \text{ @post } P))$
apply(case-tac $\tau \models \delta \ x$)
proof – **show** $\tau \models \delta \ x \implies ?thesis$ **proof** – **assume** def-x : $\tau \models \delta \ x$ **show** ?thesis **proof** –

have def-X : $\tau \models \delta \ X$

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' : $\bigwedge x. x \in \llbracket \text{Rep-Set}_{\text{base}}(X \ \tau) \rrbracket \implies x \neq \text{null}$

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 = $\lambda x. x \neq \text{null}$]) **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 $(x \ \tau) \notin \text{oid-of } \llbracket \text{Rep-Set}_{\text{base}}(X \ \tau) \rrbracket$

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 $f = \lambda x. P \llbracket x \rrbracket$ in arg-cong)
apply(insert modifiesclause[simplified OclIsModifiedOnly-def OclValid-def])
apply(case-tac τ , rename-tac $\sigma \ \sigma'$, 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  $x = \text{oid-of } (x (\sigma, \sigma'))$  in ballE) apply simp+
apply (metis (hide-lams, no-types)
      DiffD1 image-iff image-insert insert-Diff-single insert-absorb oid-is-typerrepr)
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  $\tau$ 
assumes modifiesclause:  $\tau \models (X \rightarrow \text{excluding}(x)) \rightarrow \text{ocIsModifiedOnly}()$ 
and oid-is-typerrepr :  $\tau \models X \rightarrow \text{forAll}(a \mid \text{not } (x \triangleq a))$ 
and oid-preserve:  $\bigwedge x. x \in \text{ran } (\text{heap}(\text{fst } \tau)) \vee x \in \text{ran } (\text{heap}(\text{snd } \tau)) \implies$ 
       $\text{oid-of } (H x) = \text{oid-of } x$ 
and xy-together:
 $\tau \models X \rightarrow \text{forAll}(y \mid (H .\text{allInstances}() \rightarrow \text{includes}(x) \text{ and } H .\text{allInstances}() \rightarrow \text{includes}(y)) \text{ or }$ 
       $(H .\text{allInstances}@pre() \rightarrow \text{includes}(x) \text{ and } H .\text{allInstances}@pre() \rightarrow \text{includes}(y)))$ 
shows  $\tau \models (x @pre P \triangleq (x @post P))$ 
proof –
have def-X :  $\tau \models \delta X$ 
apply(insert oid-is-typerrepr, simp add: OclForall-def OclValid-def split: split-if-asm)
by(simp add: bot-option-def true-def)
show ?thesis
apply(case-tac  $\tau \models \delta x$ , drule foundation20)
apply(rule framing[OF modifiesclause])
apply(rule OclForall-cong'[OF - oid-is-typerrepr 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: StrictRefEqObject-def)
apply(simp add: OclValid-def, subst cp-defined, simp,
      assumption)
apply(rule StrictRefEqObject-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:  $\tau \models (x .\text{ocIsNew}()) \implies \neg (\tau \models v(x @pre H1)) \wedge \neg (\tau \models v(x @post H2))$ 
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:  $\tau \models (x .\text{ocIsDeleted}()) \implies \neg (\tau \models v(x @pre H1)) \wedge \neg (\tau \models v(x @post H2))$ 
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*: $\tau \models (x.\text{oclIsAbsent}()) \implies \neg (\tau \models v(x @pre H1)) \wedge \neg (\tau \models v(x @post H2))$

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*: $(\tau \models v(x @pre H1) \vee \tau \models v(x @post H2)) \implies \tau \models (x.\text{oclIsMaintained}())$

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'*:

$\tau \models (x.\text{oclIsMaintained}()) \implies (\tau \models v(x @pre (\text{Some } o H1)) \wedge \tau \models v(x @post (\text{Some } o H2)))$

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-v*

fixes *f-lam*

fixes *f* :: $(\text{'}\mathfrak{A}, \text{'}\alpha 0 :: \text{null}) \text{val} \Rightarrow$

$\text{'}b \Rightarrow$

$(\text{'}\mathfrak{A}, \text{'}\text{res} :: \text{null}) \text{val}$

fixes *PRE*

fixes *POST*

assumes *def-scheme'*: $f \text{ self } x \equiv (\lambda \tau. \text{if } (\tau \models (\delta \text{ self})) \wedge f\text{-}v \ x \ \tau$

$\text{then SOME res. } (\tau \models PRE \text{ self } x) \wedge$

$(\tau \models POST \text{ self } x \ (\lambda \text{ -. res}))$

$\text{else invalid } \tau)$

assumes *all-post'*: $\forall \sigma \sigma' \sigma''. ((\sigma, \sigma') \models PRE \text{ self } x) = ((\sigma, \sigma'') \models PRE \text{ self } x)$

assumes cp_{PRE}' : $PRE (self) x \tau = PRE (\lambda -. self \tau) (f-lam x \tau) \tau$

assumes cp_{POST}' : $POST (self) x (res) \tau = POST (\lambda -. self \tau) (f-lam x \tau) (\lambda -. res \tau) \tau$

assumes $f-v-val$: $\bigwedge a1. f-v (f-lam a1 \tau) \tau = f-v a1 \tau$

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* : $f self a1 \tau = f (\lambda -. self \tau) (f-lam a1 \tau) \tau$

proof –

have A : $(\tau \models \delta (\lambda -. self \tau)) = (\tau \models \delta self)$ **by**(*simp add: OclValid-def cp-defined[symmetric]*)

have B : $f-v (f-lam a1 \tau) \tau = f-v a1 \tau$ **by** (*rule f-v-val*)

have D : $(\tau \models PRE (\lambda -. self \tau) (f-lam a1 \tau)) = (\tau \models PRE self a1)$

by(*simp add: OclValid-def cpPRE'[symmetric]*)

show ?thesis

apply(*auto simp: def-scheme' A B D*)

apply(*simp add: OclValid-def*)

by(*subst cpPOST', simp*)

qed

theorem *unfold'* :

assumes *context-ok*: $cp E$

and *args-def-or-valid*: $(\tau \models \delta self) \wedge f-v a1 \tau$

and *pre-satisfied*: $\tau \models PRE self a1$

and *post-satisfiable*: $\exists res. (\tau \models POST self a1 (\lambda -. res))$

and *sat-for-sols-post*: $(\bigwedge res. \tau \models POST self a1 (\lambda -. res) \implies \tau \models E (\lambda -. res))$

shows $\tau \models E(f self a1)$

proof –

have $cp0$: $\bigwedge X \tau. E X \tau = E (\lambda -. X \tau) \tau$ **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*: $(\tau \models \delta self) \wedge (f-v a1 \tau)$

and *pre-satisfied*: $\tau \models PRE self a1$

and *postsplit-satisfied*: $\tau \models POST' self a1$

and *post-decomposable* : $\bigwedge res. (POST self a1 res) = ((POST' self a1) \text{ and } (res \triangleq (BODY self a1)))$

```

shows ( $\tau \models E(f \text{ self } a1)$ ) = ( $\tau \models E(\text{BODY self } a1)$ )
proof –
  have  $cp0: \bigwedge X \tau. E X \tau = E (\lambda -. X \tau) \tau$  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  $f :: ('A, 'a0::null)val \Rightarrow$ 
     $('A, 'res::null)val$ 
  fixes  $PRE$ 
  fixes  $POST$ 
  assumes  $\text{def-scheme}: f \text{ self} \equiv (\lambda \tau. \text{if } (\tau \models (\delta \text{ self}))$ 
     $\text{then SOME } res. (\tau \models PRE \text{ self}) \wedge$ 
     $(\tau \models POST \text{ self } (\lambda -. res))$ 
     $\text{else invalid } \tau)$ 
  assumes  $\text{all-post}: \forall \sigma \sigma' \sigma''. ((\sigma, \sigma') \models PRE \text{ self}) = ((\sigma, \sigma'') \models PRE \text{ self})$ 

  assumes  $cp_{PRE}: PRE (\text{self}) \tau = PRE (\lambda -. \text{self } \tau) \tau$ 

  assumes  $cp_{POST}: POST (\text{self}) (res) \tau = POST (\lambda -. \text{self } \tau) (\lambda -. res \tau) \tau$ 

  sublocale contract0 < contract-scheme  $\lambda -. \text{True } \lambda x -. x \lambda x -. f x \lambda x -. PRE x \lambda x -. POST x$ 
  apply (unfold-locales)
  apply (simp add: def-scheme, rule all-post, rule cp_{PRE}, rule cp_{POST})
by simp

context contract0
begin
  lemma  $cp\text{-pre}: cp \text{ self}' \Longrightarrow cp (\lambda X. PRE (\text{self}' X))$ 
  by (rule-tac f=PRE in cpI1, auto intro: cp_{PRE})

  lemma  $cp\text{-post}: cp \text{ self}' \Longrightarrow cp \text{ res}' \Longrightarrow cp (\lambda X. POST (\text{self}' X) (\text{res}' X))$ 
  by (rule-tac f=POST in cpI2, auto intro: cp_{POST})

  lemma  $cp [simp]: cp \text{ self}' \Longrightarrow cp \text{ res}' \Longrightarrow cp (\lambda X. f (\text{self}' X))$ 
  by (rule-tac f=f in cpI1, auto intro: cp0)

  lemmas  $\text{unfold} = \text{unfold}'[\text{simplified}]$ 

  lemma  $\text{unfold2} :$ 

```

```

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  $f :: ('A, 'a0::null)val \Rightarrow$ 
         $('A, 'a1::null)val \Rightarrow$ 
         $('A, 'res::null)val$ 
fixes  $PRE$ 
fixes  $POST$ 
assumes def-scheme:  $f\ self\ a1 \equiv$ 
         $(\lambda\ \tau. \text{if } (\tau \models (\delta\ self)) \wedge (\tau \models v\ a1)$ 
         $\text{then } SOME\ res. (\tau \models PRE\ self\ a1) \wedge$ 
         $(\tau \models POST\ self\ a1\ (\lambda\ -. res))$ 
         $\text{else } invalid\ \tau)$ 
assumes all-post:  $\forall\ \sigma\ \sigma'\ \sigma''. ((\sigma, \sigma') \models PRE\ self\ a1) = ((\sigma, \sigma'') \models PRE\ self\ a1)$ 

assumes  $cp_{PRE}: PRE\ (self)\ (a1)\ \tau = PRE\ (\lambda\ -. self\ \tau)\ (\lambda\ -. a1\ \tau)\ \tau$ 

assumes  $cp_{POST}: POST\ (self)\ (a1)\ (res)\ \tau = POST\ (\lambda\ -. self\ \tau)\ (\lambda\ -. a1\ \tau)\ (\lambda\ -. res\ \tau)\ \tau$ 

sublocale contract1 < contract-scheme  $\lambda a1\ \tau. (\tau \models v\ a1)\ \lambda a1\ \tau. (\lambda\ -. a1\ \tau)$ 
apply(unfold-locales)
apply(rule def-scheme, rule all-post, rule cpPRE, rule cpPOST)
by(simp add: OclValid-def cp-valid[symmetric])

context contract1
begin
lemma strict1[simp]:  $f\ self\ invalid = invalid$ 
by(rule ext, rename-tac  $\tau$ , simp add: def-scheme)

lemma cp-pre:  $cp\ self' \Longrightarrow cp\ a1' \Longrightarrow cp\ (\lambda X. PRE\ (self'\ X)\ (a1'\ X))$ 
by(rule-tac  $f=PRE$  in cpI2, auto intro: cpPRE)

lemma cp-post:  $cp\ self' \Longrightarrow cp\ a1' \Longrightarrow cp\ res'$ 
         $\Longrightarrow cp\ (\lambda X. POST\ (self'\ X)\ (a1'\ X)\ (res'\ X))$ 
by(rule-tac  $f=POST$  in cpI3, auto intro: cpPOST)

lemma cp [simp]:  $cp\ self' \Longrightarrow cp\ a1' \Longrightarrow cp\ res' \Longrightarrow cp\ (\lambda X. f\ (self'\ X)\ (a1'\ X))$ 
by(rule-tac  $f=f$  in cpI2, auto intro: cp0)

```

```

lemmas unfold = unfold'
lemmas unfold2 = unfold2'
end

locale contract2 =
  fixes f :: (' $\mathcal{A}$ , ' $\alpha 0$ ::null)val  $\Rightarrow$ 
    (' $\mathcal{A}$ , ' $\alpha 1$ ::null)val  $\Rightarrow$  (' $\mathcal{A}$ , ' $\alpha 2$ ::null)val  $\Rightarrow$ 
    (' $\mathcal{A}$ , 'res::null)val
  fixes PRE
  fixes POST
  assumes def-scheme: f self a1 a2  $\equiv$ 
    ( $\lambda \tau. \text{if } (\tau \models (\delta \text{ self})) \wedge (\tau \models v \ a1) \wedge (\tau \models v \ a2)$ 
       $\text{then SOME } res. (\tau \models PRE \text{ self } a1 \ a2) \wedge$ 
       $(\tau \models POST \text{ self } a1 \ a2 \ (\lambda -. \text{res}))$ 
       $\text{else invalid } \tau$ )
  assumes all-post:  $\forall \sigma \sigma' \sigma''. ((\sigma, \sigma') \models PRE \text{ self } a1 \ a2) = ((\sigma, \sigma'') \models PRE \text{ self } a1 \ a2)$ 
  assumes cpPRE:  $PRE \ (self) \ (a1) \ (a2) \ \tau = PRE \ (\lambda -. \text{self } \tau) \ (\lambda -. \ a1 \ \tau) \ (\lambda -. \ a2 \ \tau) \ \tau$ 
  assumes cpPOST:  $\wedge res. POST \ (self) \ (a1) \ (a2) \ (res) \ \tau =$ 
     $POST \ (\lambda -. \text{self } \tau) (\lambda -. \ a1 \ \tau) (\lambda -. \ a2 \ \tau) (\lambda -. \ res \ \tau) \ \tau$ 

sublocale contract2 < contract-scheme  $\lambda (a1, a2) \tau. (\tau \models v \ a1) \wedge (\tau \models v \ a2)$ 
   $\lambda (a1, a2) \tau. (\lambda -. \ a1 \ \tau, \lambda -. \ a2 \ \tau)$ 
   $(\lambda x \ (a, b). f \ x \ a \ b)$ 
   $(\lambda x \ (a, b). PRE \ x \ a \ b)$ 
   $(\lambda x \ (a, b). POST \ x \ a \ b)$ 
apply(unfold-locales)
  apply(auto simp add: def-scheme)
  apply (metis all-post, metis all-post)
  apply(subst cpPRE, simp)
  apply(subst cpPOST, 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  $\tau$ , 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*: $cp\ self' \implies cp\ a1' \implies cp\ a2' \implies cp\ (\lambda X. PRE\ (self'\ X)\ (a1'\ X)\ (a2'\ X))$
by(*rule-tac* $f=PRE$ **in** *cpI3*, *auto intro: cpPRE*)

lemma *cp-post*: $cp\ self' \implies cp\ a1' \implies cp\ a2' \implies cp\ res'$
 $\implies cp\ (\lambda X. POST\ (self'\ X)\ (a1'\ X)\ (a2'\ X)\ (res'\ X))$
by(*rule-tac* $f=POST$ **in** *cpI4*, *auto intro: cpPOST*)

lemma *cp0* : $f\ self\ a1\ a2\ \tau = f\ (\lambda -. self\ \tau)\ (\lambda -. a1\ \tau)\ (\lambda -. a2\ \tau)\ \tau$
by (*rule cp0*[*of* - (*a1*,*a2*), *simplified*])

lemma *cp* [*simp*]: $cp\ self' \implies cp\ a1' \implies cp\ a2' \implies cp\ res'$
 $\implies cp\ (\lambda X. f\ (self'\ X)\ (a1'\ X)\ (a2'\ X))$
by(*rule-tac* $f=f$ **in** *cpI3*, *auto intro:cp0*)

theorem *unfold* :

assumes $cp\ E$
and $(\tau \models \delta\ self) \wedge (\tau \models v\ a1) \wedge (\tau \models v\ a2)$
and $\tau \models PRE\ self\ a1\ a2$
and $\exists res. (\tau \models POST\ self\ a1\ a2\ (\lambda -. res))$
and $(\wedge res. \tau \models POST\ self\ a1\ a2\ (\lambda -. res) \implies \tau \models E\ (\lambda -. res))$
shows $\tau \models E(f\ self\ a1\ a2)$
apply(*rule unfold'*[*of* - - (*a1*, *a2*), *simplified*])
by((*rule assms*)+)

lemma *unfold2* :

assumes $cp\ E$
and $(\tau \models \delta\ self) \wedge (\tau \models v\ a1) \wedge (\tau \models v\ a2)$
and $\tau \models PRE\ self\ a1\ a2$
and $\tau \models POST'\ self\ a1\ a2$
and $\wedge res. (POST\ self\ a1\ a2\ res) = ((POST'\ self\ a1\ a2)\ and\ (res \triangleq (BODY\ self\ a1\ a2)))$
shows $(\tau \models E(f\ self\ a1\ a2)) = (\tau \models E(BODY\ self\ a1\ a2))$
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 substitution step)

  ⟩⟩

lemma test1 :  $\tau \models A \implies \tau \models (A \text{ and } B \triangleq B)$ 
apply(tactic ocl-subst-asm-tac @{context} 1)
apply(simp)
done

lemma test2 :  $\tau \models A \implies \tau \models (A \text{ and } B \triangleq B)$ 
by(ocl-subst-asm, simp)

lemma test3 :  $\tau \models A \implies \tau \models (A \text{ and } A)$ 
by(ocl-subst-asm, simp)

```


lemma *test4* : $\tau \models \text{not } A \implies \tau \models (A \text{ and } B \triangleq \text{false})$
by(*ocl-subst-asm, simp*)

lemma *test5* : $\tau \models (A \triangleq \text{null}) \implies \tau \models (B \triangleq \text{null}) \implies \neg (\tau \models (A \text{ and } B))$
by(*ocl-subst-asm, ocl-subst-asm, simp*)

lemma *test6* : $\tau \models \text{not } A \implies \neg (\tau \models (A \text{ and } B))$
by(*ocl-subst-asm, simp*)

lemma *test7* : $\neg (\tau \models (\vee A)) \implies \tau \models (\text{not } B) \implies \neg (\tau \models (A \text{ and } B))$
by(*ocl-subst-asm, ocl-subst-asm, simp*)

lemma *X*: $\neg (\tau \models (\text{invalid and } B))$
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'*: $\neg (\tau \models (\text{invalid and } B))$
by(*simp add:foundation10'*)
lemma *Y*: $\neg (\tau \models (\text{null and } B))$
by(*simp add:foundation10'*)
lemma *Z*: $\neg (\tau \models (\text{false and } B))$
by(*simp add:foundation10'*)
lemma *Z'*: $(\tau \models (\text{true and } B)) = (\tau \models B)$
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.


```

type-synonym Integer    =  $\mathbb{A}$  Integer
type-synonym Void      =  $\mathbb{A}$  Void
type-synonym OclAny    = ( $\mathbb{A}$ , typeOclAny option option) val
type-synonym Person    = ( $\mathbb{A}$ , typePerson option option) val
type-synonym Set-Integer = ( $\mathbb{A}$ , int option option) Set
type-synonym Set-Person = ( $\mathbb{A}$ , typePerson 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 typePerson :: object
begin
  definition oid-of-typePerson-def: oid-of  $x = (case\ x\ of\ mk_{Person}\ oid - \Rightarrow\ oid)$ 
  instance ..
end

```

```

instantiation typeOclAny :: object
begin
  definition oid-of-typeOclAny-def: oid-of  $x = (case\ x\ of\ mk_{OclAny}\ oid - \Rightarrow\ oid)$ 
  instance ..
end

```

```

instantiation  $\mathbb{A}$  :: object
begin
  definition oid-of- $\mathbb{A}$ -def: oid-of  $x = (case\ x\ of$ 
     $in_{Person}\ person \Rightarrow\ oid-of\ person$ 
     $| in_{OclAny}\ oclany \Rightarrow\ oid-of\ oclany)$ 
  instance ..
end

```

B.4.3. Instantiation of the Generic Strict Equality

We instantiate the referential equality on *Person* and *OclAny*

```

defs(overloaded) StrictRefEqObject-Person : ( $x::Person$ )  $\doteq$   $y \equiv StrictRefEq_{Object}\ x\ y$ 
defs(overloaded) StrictRefEqObject-OclAny : ( $x::OclAny$ )  $\doteq$   $y \equiv StrictRefEq_{Object}\ x\ y$ 

```

lemmas

```

cp-StrictRefEqObject [of  $x::Person\ y::Person\ \tau$ ,
  simplified StrictRefEqObject-Person [symmetric]]
cp-intro(9) [of  $P::Person \Rightarrow PersonQ::Person \Rightarrow Person$ ,
  simplified StrictRefEqObject-Person [symmetric]]
StrictRefEqObject-def [of  $x::Person\ y::Person$ ,
  simplified StrictRefEqObject-Person [symmetric]]
StrictRefEqObject-defargs [of -  $x::Person\ y::Person$ ,

```

$$\begin{array}{l}
\text{simplified StrictRefEqObject-} \text{Person} [\text{symmetric}] \\
\text{StrictRefEqObject-strict1} \\
[\text{of } x :: \text{Person}, \\
\text{simplified StrictRefEqObject-} \text{Person} [\text{symmetric}]] \\
\text{StrictRefEqObject-strict2} \\
[\text{of } x :: \text{Person}, \\
\text{simplified StrictRefEqObject-} \text{Person} [\text{symmetric}]]
\end{array}$$

For each Class C , we will have a casting operation $\text{.oclAsType } (C)$, a test on the actual type $\text{.oclIsTypeOf } (C)$ as well as its relaxed form $\text{.oclIsKindOf } (C)$ (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 $\text{OclAsType}_{\text{OclAny}} :: 'a \Rightarrow \text{OclAny } ((-) . \text{oclAsType}'(\text{OclAny}'))$
consts $\text{OclAsType}_{\text{Person}} :: 'a \Rightarrow \text{Person } ((-) . \text{oclAsType}'(\text{Person}'))$

definition $\text{OclAsType}_{\text{OclAny-}\mathfrak{A}} = (\lambda u. \text{case } u \text{ of } \text{in}_{\text{OclAny}} a \Rightarrow a$
 $\quad \mid \text{in}_{\text{Person}} (\text{mk}_{\text{Person}} \text{oid } a) \Rightarrow \text{mk}_{\text{OclAny}} \text{oid } [a])$

lemma $\text{OclAsType}_{\text{OclAny-}\mathfrak{A}}\text{-some}: \text{OclAsType}_{\text{OclAny-}\mathfrak{A}} x \neq \text{None}$
by(simp add: $\text{OclAsType}_{\text{OclAny-}\mathfrak{A}}\text{-def}$)

defs (overloaded) $\text{OclAsType}_{\text{OclAny-OclAny}}:$
 $(X :: \text{OclAny}) . \text{oclAsType}(\text{OclAny}) \equiv X$

defs (overloaded) $\text{OclAsType}_{\text{OclAny-Person}}:$
 $(X :: \text{Person}) . \text{oclAsType}(\text{OclAny}) \equiv$
 $(\lambda \tau. \text{case } X \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad \mid \lfloor \perp \rfloor \Rightarrow \text{null } \tau$
 $\quad \mid \lfloor \text{mk}_{\text{Person}} \text{oid } a \rfloor \Rightarrow \lfloor \text{mk}_{\text{OclAny}} \text{oid } [a] \rfloor)$

definition $\text{OclAsType}_{\text{Person-}\mathfrak{A}} = (\lambda u. \text{case } u \text{ of } \text{in}_{\text{Person}} p \Rightarrow [p]$
 $\quad \mid \text{in}_{\text{OclAny}} (\text{mk}_{\text{OclAny}} \text{oid } [a]) \Rightarrow \text{mk}_{\text{Person}} \text{oid } a$
 $\quad \mid - \Rightarrow \text{None})$

defs (overloaded) $\text{OclAsType}_{\text{Person-OclAny}}:$
 $(X :: \text{OclAny}) . \text{oclAsType}(\text{Person}) \equiv$
 $(\lambda \tau. \text{case } X \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad \mid \lfloor \perp \rfloor \Rightarrow \text{null } \tau$
 $\quad \mid \lfloor \text{mk}_{\text{OclAny}} \text{oid } \perp \rfloor \Rightarrow \text{invalid } \tau \quad (* \text{down-cast exception } *)$
 $\quad \mid \lfloor \text{mk}_{\text{OclAny}} \text{oid } [a] \rfloor \Rightarrow \lfloor \text{mk}_{\text{Person}} \text{oid } a \rfloor)$

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(λX. (P (X::Person)::Person) .oclAsType(OclAny))*
by(rule *cpI1*, simp-all add: *OclAsType_{OclAny}-Person*)
lemma *cp-OclAsType_{OclAny}-OclAny-OclAny*: *cp P ⇒ cp(λX. (P (X::OclAny)::OclAny) .oclAsType(OclAny))*
by(rule *cpI1*, simp-all add: *OclAsType_{OclAny}-OclAny*)
lemma *cp-OclAsType_{Person}-Person-Person*: *cp P ⇒ cp(λX. (P (X::Person)::Person) .oclAsType(Person))*
by(rule *cpI1*, simp-all add: *OclAsType_{Person}-Person*)
lemma *cp-OclAsType_{Person}-OclAny-OclAny*: *cp P ⇒ cp(λX. (P (X::OclAny)::OclAny) .oclAsType(Person))*
by(rule *cpI1*, simp-all add: *OclAsType_{Person}-OclAny*)

lemma *cp-OclAsType_{OclAny}-Person-OclAny*: *cp P ⇒ cp(λX. (P (X::Person)::OclAny) .oclAsType(OclAny))*
by(rule *cpI1*, simp-all add: *OclAsType_{OclAny}-OclAny*)
lemma *cp-OclAsType_{OclAny}-OclAny-Person*: *cp P ⇒ cp(λX. (P (X::OclAny)::Person) .oclAsType(OclAny))*
by(rule *cpI1*, simp-all add: *OclAsType_{OclAny}-Person*)
lemma *cp-OclAsType_{Person}-Person-OclAny*: *cp P ⇒ cp(λX. (P (X::Person)::OclAny) .oclAsType(Person))*
by(rule *cpI1*, simp-all add: *OclAsType_{Person}-OclAny*)
lemma *cp-OclAsType_{Person}-OclAny-Person*: *cp P ⇒ cp(λX. (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 $OclIsTypeOf_{OclAny} :: 'α \Rightarrow Boolean ((-) .oclIsTypeOf'(OclAny'))$
consts $OclIsTypeOf_{Person} :: 'α \Rightarrow Boolean ((-) .oclIsTypeOf'(Person'))$

defs (overloaded) $OclIsTypeOf_{OclAny-OclAny}$:
 $(X::OclAny) .oclIsTypeOf(OclAny) \equiv$
 $(\lambda \tau. \text{case } X \text{ of}$
 $\quad \perp \Rightarrow invalid \ \tau$
 $\quad | \lfloor \perp \rfloor \Rightarrow true \ \tau \ (* \ invalid \ ?? \ *)$
 $\quad | \lfloor mk_{OclAny} \ oid \ \perp \rfloor \rfloor \Rightarrow true \ \tau$
 $\quad | \lfloor \lfloor mk_{OclAny} \ oid \ [-] \rfloor \rfloor \Rightarrow false \ \tau)$

defs (overloaded) $OclIsTypeOf_{OclAny-Person}$:
 $(X::Person) .oclIsTypeOf(OclAny) \equiv$
 $(\lambda \tau. \text{case } X \text{ of}$
 $\quad \perp \Rightarrow invalid \ \tau$
 $\quad | \lfloor \perp \rfloor \Rightarrow true \ \tau \ (* \ invalid \ ?? \ *)$
 $\quad | \lfloor \lfloor - \rfloor \rfloor \Rightarrow false \ \tau)$

defs (overloaded) $OclIsTypeOf_{Person-OclAny}$:
 $(X::OclAny) .oclIsTypeOf(Person) \equiv$
 $(\lambda \tau. \text{case } X \text{ of}$
 $\quad \perp \Rightarrow invalid \ \tau$

$$\begin{aligned}
&| \lfloor \perp \rfloor \Rightarrow \text{true } \tau \\
&| \lfloor \text{mk}_{OclAny} \text{ oid } \perp \rfloor \Rightarrow \text{false } \tau \\
&| \lfloor \text{mk}_{OclAny} \text{ oid } [-] \rfloor \Rightarrow \text{true } \tau
\end{aligned}$$

defs (overloaded) $OclIsTypeOf_{Person-Person}$:
 $(X::Person) .oclIsTypeOf(Person) \equiv$
 $(\lambda \tau. \text{case } X \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | - \Rightarrow \text{true } \tau)$

Context Passing

lemma $cp\text{-}OclIsTypeOf_{OclAny-Person-Person}$: $cp \ P \Rightarrow cp(\lambda X. (P(X::Person)::Person).oclIsTypeOf(OclAny))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny-Person}$)
lemma $cp\text{-}OclIsTypeOf_{OclAny-OclAny-OclAny}$: $cp \ P \Rightarrow cp(\lambda X. (P(X::OclAny)::OclAny).oclIsTypeOf(OclAny))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny-OclAny}$)
lemma $cp\text{-}OclIsTypeOf_{Person-Person-Person}$: $cp \ P \Rightarrow cp(\lambda X. (P(X::Person)::Person).oclIsTypeOf(Person))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{Person-Person}$)
lemma $cp\text{-}OclIsTypeOf_{Person-OclAny-OclAny}$: $cp \ P \Rightarrow cp(\lambda X. (P(X::OclAny)::OclAny).oclIsTypeOf(Person))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{Person-OclAny}$)

lemma $cp\text{-}OclIsTypeOf_{OclAny-Person-OclAny}$: $cp \ P \Rightarrow cp(\lambda X. (P(X::Person)::OclAny).oclIsTypeOf(OclAny))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny-OclAny}$)
lemma $cp\text{-}OclIsTypeOf_{OclAny-OclAny-Person}$: $cp \ P \Rightarrow cp(\lambda X. (P(X::OclAny)::Person).oclIsTypeOf(OclAny))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny-Person}$)
lemma $cp\text{-}OclIsTypeOf_{Person-Person-OclAny}$: $cp \ P \Rightarrow cp(\lambda X. (P(X::Person)::OclAny).oclIsTypeOf(Person))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{Person-OclAny}$)
lemma $cp\text{-}OclIsTypeOf_{Person-OclAny-Person}$: $cp \ P \Rightarrow cp(\lambda X. (P(X::OclAny)::Person).oclIsTypeOf(Person))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{Person-Person}$)

lemmas [simp] =
 $cp\text{-}OclIsTypeOf_{OclAny-Person-Person}$
 $cp\text{-}OclIsTypeOf_{OclAny-OclAny-OclAny}$
 $cp\text{-}OclIsTypeOf_{Person-Person-Person}$
 $cp\text{-}OclIsTypeOf_{Person-OclAny-OclAny}$

 $cp\text{-}OclIsTypeOf_{OclAny-Person-OclAny}$
 $cp\text{-}OclIsTypeOf_{OclAny-OclAny-Person}$
 $cp\text{-}OclIsTypeOf_{Person-Person-OclAny}$
 $cp\text{-}OclIsTypeOf_{Person-OclAny-Person}$

Execution with Invalid or Null as Argument

lemma $OclIsTypeOf_{OclAny-OclAny-strictI}$ [simp]:
 $(\text{invalid}::OclAny) .oclIsTypeOf(OclAny) = \text{invalid}$
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
 $OclIsTypeOf_{OclAny-OclAny}$)


```

lemma OclIsTypeOfOclAny-OclAny-strict2[simp]:
  (null::OclAny) .ocIsTypeOf(OclAny) = true
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OclIsTypeOfOclAny-OclAny)
lemma OclIsTypeOfOclAny-Person-strict1[simp]:
  (invalid::Person) .ocIsTypeOf(OclAny) = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OclIsTypeOfOclAny-Person)
lemma OclIsTypeOfOclAny-Person-strict2[simp]:
  (null::Person) .ocIsTypeOf(OclAny) = true
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OclIsTypeOfOclAny-Person)
lemma OclIsTypeOfPerson-OclAny-strict1[simp]:
  (invalid::OclAny) .ocIsTypeOf(Person) = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OclIsTypeOfPerson-OclAny)
lemma OclIsTypeOfPerson-OclAny-strict2[simp]:
  (null::OclAny) .ocIsTypeOf(Person) = true
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OclIsTypeOfPerson-OclAny)
lemma OclIsTypeOfPerson-Person-strict1[simp]:
  (invalid::Person) .ocIsTypeOf(Person) = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OclIsTypeOfPerson-Person)
lemma OclIsTypeOfPerson-Person-strict2[simp]:
  (null::Person) .ocIsTypeOf(Person) = true
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OclIsTypeOfPerson-Person)

```

Up Down Casting

```

lemma actualType-larger-staticType:
assumes isdef:  $\tau \models (\delta X)$ 
shows  $\tau \models (X::Person) .ocIsTypeOf(OclAny) \triangleq \text{false}$ 
using isdef
by(auto simp : null-option-def bot-option-def
  OclIsTypeOfOclAny-Person foundation22 foundation16)

lemma down-cast-type:
assumes isOclAny:  $\tau \models (X::OclAny) .ocIsTypeOf(OclAny)$ 
and non-null:  $\tau \models (\delta X)$ 
shows  $\tau \models (X .oclAsType(Person)) \triangleq \text{invalid}$ 
using isOclAny non-null
apply(auto simp : bot-fun-def null-fun-def null-option-def bot-option-def null-def invalid-def
  OclAsTypeOclAny-Person OclAsTypePerson-OclAny foundation22 foundation16
  split: option.split typeOclAny.split typePerson.split)
by(simp add: OclIsTypeOfOclAny-OclAny OclValid-def false-def true-def)

```

```

lemma down-cast-type':
assumes isOclAny:  $\tau \models (X :: \text{OclAny}) . \text{oclIsTypeOf}(\text{OclAny})$ 
and non-null:  $\tau \models (\delta X)$ 
shows  $\tau \models \text{not } (v (X . \text{oclAsType}(\text{Person})))$ 
by(rule foundation15[THEN iffD1], simp add: down-cast-type[OF assms])

lemma up-down-cast :
assumes isdef:  $\tau \models (\delta X)$ 
shows  $\tau \models ((X :: \text{Person}) . \text{oclAsType}(\text{OclAny}) . \text{oclAsType}(\text{Person}) \triangleq X)$ 
using isdef
by(auto simp : null-fun-def null-option-def bot-option-def null-def invalid-def
    OclAsTypeOclAny-Person OclAsTypePerson-OclAny foundation22 foundation16
    split: option.split typePerson.split)

```

```

lemma up-down-cast-Person-OclAny-Person [simp]:
shows  $((X :: \text{Person}) . \text{oclAsType}(\text{OclAny}) . \text{oclAsType}(\text{Person}) = X)$ 
apply(rule ext, rename-tac  $\tau$ )
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  $\tau \models v X$ 
shows  $\tau \models (((X :: \text{Person}) . \text{oclAsType}(\text{OclAny}) . \text{oclAsType}(\text{Person})) \doteq X)$ 
apply(simp only: up-down-cast-Person-OclAny-Person StrictRefEqObject-Person)
by(rule StrictRefEqObject-sym, simp add: assms)

```

```

lemma up-down-cast-Person-OclAny-Person'':
assumes  $\tau \models v (X :: \text{Person})$ 
shows  $\tau \models (X . \text{oclIsTypeOf}(\text{Person}) \text{ implies } (X . \text{oclAsType}(\text{OclAny}) . \text{oclAsType}(\text{Person})) \doteq X)$ 
apply(simp add: OclValid-def)
apply(subst cp-OclImplies)
apply(simp add: StrictRefEqObject-Person StrictRefEqObject-sym[OF assms, simplified OclValid-def])
apply(subst cp-OclImplies[symmetric])
by (simp add: OclImplies-true)

```

B.4.6. OclIsKindOf

Definition

```

consts OclIsKindOfOclAny :: ' $\alpha \Rightarrow \text{Boolean } ((-) . \text{oclIsKindOf}'(\text{OclAny}'))$ '
consts OclIsKindOfPerson :: ' $\alpha \Rightarrow \text{Boolean } ((-) . \text{oclIsKindOf}'(\text{Person}'))$ '

```

```

defs (overloaded) OclIsKindOfOclAny-OclAny:
   $(X :: \text{OclAny}) . \text{oclIsKindOf}(\text{OclAny}) \equiv$ 

```

$$\begin{aligned}
&(\lambda \tau. \text{case } X \ \tau \text{ of} \\
&\quad \perp \Rightarrow \text{invalid } \tau \\
&\quad | \ - \Rightarrow \text{true } \tau)
\end{aligned}$$

defs (overloaded) *OclIsKindOf_{OclAny}-Person*:
 $(X::\text{Person}) .\text{oclIsKindOf}(\text{OclAny}) \equiv$
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \ - \Rightarrow \text{true } \tau)$

defs (overloaded) *OclIsKindOf_{Person}-OclAny*:
 $(X::\text{OclAny}) .\text{oclIsKindOf}(\text{Person}) \equiv$
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \ [\perp] \Rightarrow \text{true } \tau$
 $\quad | \ [\text{mk}_{\text{OclAny}} \text{oid } \perp] \Rightarrow \text{false } \tau$
 $\quad | \ [\text{mk}_{\text{OclAny}} \text{oid } _] \Rightarrow \text{true } \tau)$

defs (overloaded) *OclIsKindOf_{Person}-Person*:
 $(X::\text{Person}) .\text{oclIsKindOf}(\text{Person}) \equiv$
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \ - \Rightarrow \text{true } \tau)$

Context Passing

lemma *cp-OclIsKindOf_{OclAny}-Person-Person*: $cp \ P \implies cp(\lambda X. (P(X::\text{Person})::\text{Person}).\text{oclIsKindOf}(\text{OclAny}))$
by(rule *cpI1*, simp-all add: *OclIsKindOf_{OclAny}-Person*)
lemma *cp-OclIsKindOf_{OclAny}-OclAny-OclAny*: $cp \ P \implies cp(\lambda X. (P(X::\text{OclAny})::\text{OclAny}).\text{oclIsKindOf}(\text{OclAny}))$
by(rule *cpI1*, simp-all add: *OclIsKindOf_{OclAny}-OclAny*)
lemma *cp-OclIsKindOf_{Person}-Person-Person*: $cp \ P \implies cp(\lambda X. (P(X::\text{Person})::\text{Person}).\text{oclIsKindOf}(\text{Person}))$
by(rule *cpI1*, simp-all add: *OclIsKindOf_{Person}-Person*)
lemma *cp-OclIsKindOf_{Person}-OclAny-OclAny*: $cp \ P \implies cp(\lambda X. (P(X::\text{OclAny})::\text{OclAny}).\text{oclIsKindOf}(\text{Person}))$
by(rule *cpI1*, simp-all add: *OclIsKindOf_{Person}-OclAny*)

lemma *cp-OclIsKindOf_{OclAny}-Person-OclAny*: $cp \ P \implies cp(\lambda X. (P(X::\text{Person})::\text{OclAny}).\text{oclIsKindOf}(\text{OclAny}))$
by(rule *cpI1*, simp-all add: *OclIsKindOf_{OclAny}-OclAny*)
lemma *cp-OclIsKindOf_{OclAny}-OclAny-Person*: $cp \ P \implies cp(\lambda X. (P(X::\text{OclAny})::\text{Person}).\text{oclIsKindOf}(\text{OclAny}))$
by(rule *cpI1*, simp-all add: *OclIsKindOf_{OclAny}-Person*)
lemma *cp-OclIsKindOf_{Person}-Person-OclAny*: $cp \ P \implies cp(\lambda X. (P(X::\text{Person})::\text{OclAny}).\text{oclIsKindOf}(\text{Person}))$
by(rule *cpI1*, simp-all add: *OclIsKindOf_{Person}-OclAny*)
lemma *cp-OclIsKindOf_{Person}-OclAny-Person*: $cp \ P \implies cp(\lambda X. (P(X::\text{OclAny})::\text{Person}).\text{oclIsKindOf}(\text{Person}))$
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
      OclIsKindOfOclAny-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
      OclAsTypeOclAny-Person OclAsTypePerson-OclAny foundation22 foundation16
      split: option.split typeOclAny.split typePerson.split)
by(simp add: OclIsKindOfPerson-OclAny OclValid-def false-def true-def)

```

B.4.7. 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 sufficient “characterization.”

definition $Person \equiv OclAsType_{Person}\mathcal{A}$

definition $OclAny \equiv OclAsType_{OclAny}\mathcal{A}$

lemmas [simp] = Person-def OclAny-def

lemma $OclAllInstances-generic_{OclAny-exec}: OclAllInstances-generic\ pre-post\ OclAny =$
 $(\lambda \tau. Abs-Set_{base} \llbracket Some\ 'OclAny'\ ran\ (heap\ (pre-post\ \tau)) \rrbracket)$

proof –

```

let ?S1 =  $\lambda \tau. OclAny\ 'ran\ (heap\ (pre-post\ \tau))$ 
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: OclAsTypeOclAny- $\mathcal{A}$ -some)

```

show ?thesis **by**(insert equalityI[OF B C], simp)

qed

lemma $OclAllInstances-at-post_{OclAny-exec}: OclAny.allInstances() =$
 $(\lambda \tau. Abs-Set_{base} \llbracket Some\ 'OclAny'\ ran\ (heap\ (snd\ \tau)) \rrbracket)$

unfolding OclAllInstances-at-post-def

by(rule OclAllInstances-generic_{OclAny-exec})

lemma $OclAllInstances-at-pre_{OclAny-exec}: OclAny.allInstances@pre() =$
 $(\lambda \tau. Abs-Set_{base} \llbracket Some\ 'OclAny'\ ran\ (heap\ (fst\ \tau)) \rrbracket)$

unfolding OclAllInstances-at-pre-def

by(rule OclAllInstances-generic_{OclAny-exec})

OclIsTypeOf

lemma $OclAny.allInstances-generic-oclIsTypeOf_{OclAny}I:$

assumes $[simp]: \bigwedge x. pre\text{-}post(x, x) = x$
shows $\exists \tau. (\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ OclAny) \rightarrow_{forall} (X|X. oclIsTypeOf(OclAny))))$
apply(*rule-tac* $x = \tau_0$ **in** *exI*, *simp add*: $\tau_0\text{-}def\ OclValid\text{-}def\ del: OclAllInstances\text{-}generic\text{-}def$)
apply(*simp only*: *assms* $OclForall\text{-}def\ refl\ if\ True$
 $OclAllInstances\text{-}generic\text{-}defined[simplified\ OclValid\text{-}def]$)
apply(*simp only*: $OclAllInstances\text{-}generic\text{-}def$)
apply(*subst* $(1\ 2\ 3)\ Abs\text{-}Set_{base}\text{-}inverse$, *simp add*: *bot-option-def*)
by(*simp add*: $OclIsTypeOf_{OclAny}\text{-}OclAny$)

lemma $OclAny\text{-}allInstances\text{-}at\text{-}post\text{-}oclIsTypeOf_{OclAny}1$:
 $\exists \tau. (\tau \models (OclAny. allInstances() \rightarrow_{forall} (X|X. oclIsTypeOf(OclAny))))$
unfolding $OclAllInstances\text{-}at\text{-}post\text{-}def$
by(*rule* $OclAny\text{-}allInstances\text{-}generic\text{-}oclIsTypeOf_{OclAny}1$, *simp*)

lemma $OclAny\text{-}allInstances\text{-}at\text{-}pre\text{-}oclIsTypeOf_{OclAny}1$:
 $\exists \tau. (\tau \models (OclAny. allInstances@pre() \rightarrow_{forall} (X|X. oclIsTypeOf(OclAny))))$
unfolding $OclAllInstances\text{-}at\text{-}pre\text{-}def$
by(*rule* $OclAny\text{-}allInstances\text{-}generic\text{-}oclIsTypeOf_{OclAny}1$, *simp*)

lemma $OclAny\text{-}allInstances\text{-}generic\text{-}oclIsTypeOf_{OclAny}2$:
assumes $[simp]: \bigwedge x. pre\text{-}post(x, x) = x$
shows $\exists \tau. (\tau \models not((OclAllInstances\text{-}generic\ pre\text{-}post\ OclAny) \rightarrow_{forall} (X|X. oclIsTypeOf(OclAny))))$
proof – **fix** *oid a* **let** $?i0 = (\text{heap} = \text{empty}(oid \mapsto in_{OclAny}(mk_{OclAny}\ oid\ [a])),$
 $assocs = \text{empty})$ **show** *?thesis*
apply(*rule-tac* $x = (?i0, ?i0)$ **in** *exI*, *simp add*: $OclValid\text{-}def\ del: OclAllInstances\text{-}generic\text{-}def$)
apply(*simp only*: $OclForall\text{-}def\ refl\ if\ True$
 $OclAllInstances\text{-}generic\text{-}defined[simplified\ OclValid\text{-}def]$)
apply(*simp only*: $OclAllInstances\text{-}generic\text{-}def\ OclAsType_{OclAny}\text{-}\mathcal{A}\text{-}def$)
apply(*subst* $(1\ 2\ 3)\ Abs\text{-}Set_{base}\text{-}inverse$, *simp add*: *bot-option-def*)
by(*simp add*: $OclIsTypeOf_{OclAny}\text{-}OclAny\ OclNot\text{-}def\ OclAny\text{-}def$)
qed

lemma $OclAny\text{-}allInstances\text{-}at\text{-}post\text{-}oclIsTypeOf_{OclAny}2$:
 $\exists \tau. (\tau \models not(OclAny. allInstances() \rightarrow_{forall} (X|X. oclIsTypeOf(OclAny))))$
unfolding $OclAllInstances\text{-}at\text{-}post\text{-}def$
by(*rule* $OclAny\text{-}allInstances\text{-}generic\text{-}oclIsTypeOf_{OclAny}2$, *simp*)

lemma $OclAny\text{-}allInstances\text{-}at\text{-}pre\text{-}oclIsTypeOf_{OclAny}2$:
 $\exists \tau. (\tau \models not(OclAny. allInstances@pre() \rightarrow_{forall} (X|X. oclIsTypeOf(OclAny))))$
unfolding $OclAllInstances\text{-}at\text{-}pre\text{-}def$
by(*rule* $OclAny\text{-}allInstances\text{-}generic\text{-}oclIsTypeOf_{OclAny}2$, *simp*)

lemma $Person\text{-}allInstances\text{-}generic\text{-}oclIsTypeOf_{Person}$:
 $\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ Person) \rightarrow_{forall} (X|X. oclIsTypeOf(Person)))$
apply(*simp add*: $OclValid\text{-}def\ del: OclAllInstances\text{-}generic\text{-}def$)
apply(*simp only*: $OclForall\text{-}def\ refl\ if\ True$
 $OclAllInstances\text{-}generic\text{-}defined[simplified\ OclValid\text{-}def]$)
apply(*simp only*: $OclAllInstances\text{-}generic\text{-}def$)

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}:
 $\tau \models (Person \text{ .allInstances}() \rightarrow \text{forAll}(X|X \text{ .oclIsTypeOf}(Person)))$
unfolding OclAllInstances-at-post-def
by(rule Person-allInstances-generic-oclIsTypeOf_{Person})

lemma Person-allInstances-at-pre-oclIsTypeOf_{Person}:
 $\tau \models (Person \text{ .allInstances@pre}() \rightarrow \text{forAll}(X|X \text{ .oclIsTypeOf}(Person)))$
unfolding OclAllInstances-at-pre-def
by(rule Person-allInstances-generic-oclIsTypeOf_{Person})

OclIsKindOf

lemma OclAny-allInstances-generic-oclIsKindOf_{OclAny}:
 $\tau \models ((OclAllInstances-generic \text{ pre-post } OclAny) \rightarrow \text{forAll}(X|X \text{ .oclIsKindOf}(OclAny)))$
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}:
 $\tau \models (OclAny \text{ .allInstances}() \rightarrow \text{forAll}(X|X \text{ .oclIsKindOf}(OclAny)))$
unfolding OclAllInstances-at-post-def
by(rule OclAny-allInstances-generic-oclIsKindOf_{OclAny})

lemma OclAny-allInstances-at-pre-oclIsKindOf_{OclAny}:
 $\tau \models (OclAny \text{ .allInstances@pre}() \rightarrow \text{forAll}(X|X \text{ .oclIsKindOf}(OclAny)))$
unfolding OclAllInstances-at-pre-def
by(rule OclAny-allInstances-generic-oclIsKindOf_{OclAny})

lemma Person-allInstances-generic-oclIsKindOf_{OclAny}:
 $\tau \models ((OclAllInstances-generic \text{ pre-post } Person) \rightarrow \text{forAll}(X|X \text{ .oclIsKindOf}(OclAny)))$
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}:
 $\tau \models (Person \text{ .allInstances}() \rightarrow \text{forAll}(X|X \text{ .oclIsKindOf}(OclAny)))$
unfolding OclAllInstances-at-post-def
by(rule Person-allInstances-generic-oclIsKindOf_{OclAny})

```

lemma Person-allInstances-at-pre-oclIsKindOfOclAny:
 $\tau \models (Person.allInstances@pre() \multimap \text{forAll}(X|X.oclIsKindOf(OclAny)))$ 
unfolding OclAllInstances-at-pre-def
by(rule Person-allInstances-generic-oclIsKindOfOclAny)

lemma Person-allInstances-generic-oclIsKindOfPerson:
 $\tau \models ((OclAllInstances-generic\ pre\ post\ Person) \multimap \text{forAll}(X|X.oclIsKindOf(Person)))$ 
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-Setbase-inverse, simp add: bot-option-def)
by(simp add: OclIsKindOfPerson-Person)

lemma Person-allInstances-at-post-oclIsKindOfPerson:
 $\tau \models (Person.allInstances() \multimap \text{forAll}(X|X.oclIsKindOf(Person)))$ 
unfolding OclAllInstances-at-post-def
by(rule Person-allInstances-generic-oclIsKindOfPerson)

lemma Person-allInstances-at-pre-oclIsKindOfPerson:
 $\tau \models (Person.allInstances@pre() \multimap \text{forAll}(X|X.oclIsKindOf(Person)))$ 
unfolding OclAllInstances-at-pre-def
by(rule Person-allInstances-generic-oclIsKindOfPerson)

```

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 $oid_{Person\mathcal{BOSS}} :: oid$ **where** $oid_{Person\mathcal{BOSS}} = 10$

From there on, we can already define an empty state which must contain for $oid_{Person\mathcal{BOSS}}$ the empty relation (encoded as association list, since there are associations with a Sequence-like structure).

definition $eval_extract :: ('A, ('a::object) option option) val$
 $\Rightarrow (oid \Rightarrow ('A, 'c::null) val)$
 $\Rightarrow ('A, 'c::null) val$
where $eval_extract\ Xf = (\lambda\ \tau. case\ X\ \tau\ of$
 $\quad \perp \Rightarrow invalid\ \tau\ (*\ exception\ propagation\ *)$
 $\quad | \perp \Rightarrow invalid\ \tau\ (*\ dereferencing\ null\ pointer\ *)$
 $\quad | [obj] \Rightarrow f\ (oid_of\ obj)\ \tau)$

definition *choose₂-1* = *fst*

definition *choose₂-2* = *snd*

definition *List-flatten* = ($\lambda l. (\text{foldl } ((\lambda acc. (\lambda l. (\text{foldl } ((\lambda acc. (\lambda l. (\text{Cons } l) (acc)))) (acc) ((\text{rev } l)))))) (Nil) ((\text{rev } l))))$)

definition *deref-assocs₂* :: ($\mathcal{A} \text{ state} \times \mathcal{A} \text{ state} \Rightarrow \mathcal{A} \text{ state}$)
 $\Rightarrow (oid \text{ list list} \Rightarrow oid \text{ list} \times oid \text{ list})$
 $\Rightarrow oid$
 $\Rightarrow (oid \text{ list} \Rightarrow (\mathcal{A}, f)val)$
 $\Rightarrow oid$
 $\Rightarrow (\mathcal{A}, f::null)val$

where *deref-assocs₂ pre-post to-from assoc-oid f oid* =
 $(\lambda \tau. \text{case } (assoc (pre-post \tau)) \text{ assoc-oid of}$
 $\quad [S] \Rightarrow f (List-flatten (map (choose_2-2 \circ to-from)$
 $\quad \quad (filter (\lambda p. List.member (choose_2-1 (to-from p)) oid) S)))$
 $\quad \tau$
 $\quad | - \Rightarrow invalid \tau)$

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 *switch₂-1* = ($\lambda [x,y] \Rightarrow (x,y)$)

definition *switch₂-2* = ($\lambda [x,y] \Rightarrow (y,x)$)

definition *switch₃-1* = ($\lambda [x,y,z] \Rightarrow (x,y)$)

definition *switch₃-2* = ($\lambda [x,y,z] \Rightarrow (x,z)$)

definition *switch₃-3* = ($\lambda [x,y,z] \Rightarrow (y,x)$)

definition *switch₃-4* = ($\lambda [x,y,z] \Rightarrow (y,z)$)

definition *switch₃-5* = ($\lambda [x,y,z] \Rightarrow (z,x)$)

definition *switch₃-6* = ($\lambda [x,y,z] \Rightarrow (z,y)$)

definition *select-object* :: ($(\mathcal{A}, 'b::null)val$)
 $\Rightarrow ((\mathcal{A}, 'b)val \Rightarrow (\mathcal{A}, 'c)val \Rightarrow (\mathcal{A}, 'b)val)$
 $\Rightarrow ((\mathcal{A}, 'b)val \Rightarrow (\mathcal{A}, 'd)val)$
 $\Rightarrow (oid \Rightarrow (\mathcal{A}, 'c::null)val)$
 $\Rightarrow oid \text{ list}$
 $\Rightarrow (\mathcal{A}, 'd)val$

where *select-object mt incl smash derefl* = *smash(foldl incl mt (map derefl))*
 (* *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*) :: ($\mathcal{A}, 'a::null)val$

definition *deref-oid_{Person}* :: ($\mathcal{A} \text{ state} \times \mathcal{A} \text{ state} \Rightarrow \mathcal{A} \text{ state}$)
 $\Rightarrow (type_{Person} \Rightarrow (\mathcal{A}, 'c::null)val)$
 $\Rightarrow oid$
 $\Rightarrow (\mathcal{A}, 'c::null)val$

where *deref-oid_{Person} fst-snd f oid* = ($\lambda \tau. \text{case } (heap (fst-snd \tau)) \text{ oid of}$

$$\begin{array}{l} \lfloor in_{person} \text{ obj} \rfloor \Rightarrow f \text{ obj } \tau \\ | - \quad \Rightarrow \text{invalid } \tau \end{array}$$

definition $deref-oid_{OclAny} :: (\mathfrak{A} \text{ state} \times \mathfrak{A} \text{ state} \Rightarrow \mathfrak{A} \text{ state})$
 $\Rightarrow (type_{OclAny} \Rightarrow (\mathfrak{A}, 'c::null)val)$
 $\Rightarrow oid$
 $\Rightarrow (\mathfrak{A}, 'c::null)val$

where $deref-oid_{OclAny} \text{ fst-snd } f \text{ oid} = (\lambda \tau. \text{ case } (heap \text{ (fst-snd } \tau)) \text{ oid of}$
 $\lfloor in_{OclAny} \text{ obj} \rfloor \Rightarrow f \text{ obj } \tau$
 $| - \quad \Rightarrow \text{invalid } \tau)$

pointer undefined in state or not referencing a type conform object representation

definition $select_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y} f = (\lambda X. \text{ case } X \text{ of}$
 $(mk_{OclAny} - \perp) \Rightarrow null$
 $| (mk_{OclAny} - \lfloor any \rfloor) \Rightarrow f (\lambda x -. \lfloor \lfloor x \rfloor \rfloor) any)$

definition $select_{person} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} f = \text{select-object } mtSet \text{ UML-Set.OclIncluding } OclANY (f (\lambda x -. \lfloor \lfloor x \rfloor \rfloor))$

definition $select_{person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y} f = (\lambda X. \text{ case } X \text{ of}$
 $(mk_{person} - \perp) \Rightarrow null$
 $| (mk_{person} - \lfloor salary \rfloor) \Rightarrow f (\lambda x -. \lfloor \lfloor x \rfloor \rfloor) salary)$

definition $deref-assocs_2 \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} \text{ fst-snd } f = (\lambda mk_{person} \text{ oid} -. \Rightarrow$
 $deref-assocs_2 \text{ fst-snd } switch_{2-1} \text{ oid}_{person} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} f \text{ oid})$

definition $in\text{-}pre\text{-}state = fst$

definition $in\text{-}post\text{-}state = snd$

definition $reconst\text{-}basetype = (\lambda \text{ convert } x. \text{ convert } x)$

definition $dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y} :: OclAny \Rightarrow - \ ((I(-).any) 50)$

where $(X).any = \text{eval-extract } X$
 $(deref-oid_{OclAny} \text{ in-post-state}$
 $(select_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y}$
 $reconst\text{-}basetype))$

definition $dot_{person} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} :: Person \Rightarrow Person \ ((I(-).boss) 50)$

where $(X).boss = \text{eval-extract } X$
 $(deref-oid_{person} \text{ in-post-state}$
 $(deref-assocs_2 \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} \text{ in-post-state}$
 $(select_{person} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S}$
 $(deref-oid_{person} \text{ in-post-state}))))$

definition $\text{dot}_{\text{Person}} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y} :: \text{Person} \Rightarrow \text{Integer} \ ((1(-).\text{salary}) \ 50)$
where $(X).\text{salary} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{Person}} \text{ in-post-state}$
 $(\text{select}_{\text{Person}} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}$
 $\text{reconst-basetype}))$

definition $\text{dot}_{\text{OclAny}} \mathcal{A} \mathcal{N} \mathcal{Y} \text{-at-pre} :: \text{OclAny} \Rightarrow - \ ((1(-).\text{any@pre}) \ 50)$
where $(X).\text{any@pre} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{OclAny}} \text{ in-pre-state}$
 $(\text{select}_{\text{OclAny}} \mathcal{A} \mathcal{N} \mathcal{Y}$
 $\text{reconst-basetype}))$

definition $\text{dot}_{\text{Person}} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} \text{-at-pre} :: \text{Person} \Rightarrow \text{Person} \ ((1(-).\text{boss@pre}) \ 50)$
where $(X).\text{boss@pre} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{Person}} \text{ in-pre-state}$
 $(\text{deref-assocs}_2 \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} \text{ in-pre-state}$
 $(\text{select}_{\text{Person}} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S}$
 $(\text{deref-oid}_{\text{Person}} \text{ in-pre-state}))))$

definition $\text{dot}_{\text{Person}} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y} \text{-at-pre} :: \text{Person} \Rightarrow \text{Integer} \ ((1(-).\text{salary@pre}) \ 50)$
where $(X).\text{salary@pre} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{Person}} \text{ in-pre-state}$
 $(\text{select}_{\text{Person}} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}$
 $\text{reconst-basetype}))$

lemmas $\text{dot-accessor} =$
 $\text{dot}_{\text{OclAny}} \mathcal{A} \mathcal{N} \mathcal{Y} \text{-def}$
 $\text{dot}_{\text{Person}} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} \text{-def}$
 $\text{dot}_{\text{Person}} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y} \text{-def}$
 $\text{dot}_{\text{OclAny}} \mathcal{A} \mathcal{N} \mathcal{Y} \text{-at-pre-def}$
 $\text{dot}_{\text{Person}} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} \text{-at-pre-def}$
 $\text{dot}_{\text{Person}} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y} \text{-at-pre-def}$

Context Passing

lemmas $[\text{simp}] = \text{eval-extract-def}$

lemma $\text{cp-dot}_{\text{OclAny}} \mathcal{A} \mathcal{N} \mathcal{Y} :: ((X).\text{any}) \ \tau = ((\lambda -. X \ \tau).\text{any}) \ \tau \text{ by } (\text{simp add: dot-accessor})$

lemma $\text{cp-dot}_{\text{Person}} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} :: ((X).\text{boss}) \ \tau = ((\lambda -. X \ \tau).\text{boss}) \ \tau \text{ by } (\text{simp add: dot-accessor})$

lemma $\text{cp-dot}_{\text{Person}} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y} :: ((X).\text{salary}) \ \tau = ((\lambda -. X \ \tau).\text{salary}) \ \tau \text{ by } (\text{simp add: dot-accessor})$

lemma $\text{cp-dot}_{\text{OclAny}} \mathcal{A} \mathcal{N} \mathcal{Y} \text{-at-pre} :: ((X).\text{any@pre}) \ \tau = ((\lambda -. X \ \tau).\text{any@pre}) \ \tau \text{ by } (\text{simp add: dot-accessor})$

lemma $\text{cp-dot}_{\text{Person}} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} \text{-at-pre} :: ((X).\text{boss@pre}) \ \tau = ((\lambda -. X \ \tau).\text{boss@pre}) \ \tau \text{ by } (\text{simp add: dot-accessor})$

lemma $\text{cp-dot}_{\text{Person}} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y} \text{-at-pre} :: ((X).\text{salary@pre}) \ \tau = ((\lambda -. X \ \tau).\text{salary@pre}) \ \tau \text{ by } (\text{simp add: dot-accessor})$

lemmas $\text{cp-dot}_{\text{OclAny}} \mathcal{A} \mathcal{N} \mathcal{Y} \text{-I } [\text{simp}, \text{intro!}] =$
 $\text{cp-dot}_{\text{OclAny}} \mathcal{A} \mathcal{N} \mathcal{Y} [\text{THEN allI} [\text{THEN allI}],$
 $\text{of } \lambda X -. X \ \lambda -. \tau, \text{ THEN cpII}]$

lemmas $cp\text{-}dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y}\text{-}at\text{-}pre\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y}\text{-}at\text{-}pre$ [THEN allI [THEN allI],
of $\lambda X \cdot X \lambda \cdot \tau. \tau$, THEN cpII]

lemmas $cp\text{-}dot_{Person} \mathcal{BO} \mathcal{S} \mathcal{S}\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{Person} \mathcal{BO} \mathcal{S} \mathcal{S}$ [THEN allI [THEN allI],
of $\lambda X \cdot X \lambda \cdot \tau. \tau$, THEN cpII]

lemmas $cp\text{-}dot_{Person} \mathcal{BO} \mathcal{S} \mathcal{S}\text{-}at\text{-}pre\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{Person} \mathcal{BO} \mathcal{S} \mathcal{S}\text{-}at\text{-}pre$ [THEN allI [THEN allI],
of $\lambda X \cdot X \lambda \cdot \tau. \tau$, THEN cpII]

lemmas $cp\text{-}dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}$ [THEN allI [THEN allI],
of $\lambda X \cdot X \lambda \cdot \tau. \tau$, THEN cpII]

lemmas $cp\text{-}dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}\text{-}at\text{-}pre\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}\text{-}at\text{-}pre$ [THEN allI [THEN allI],
of $\lambda X \cdot X \lambda \cdot \tau. \tau$, THEN cpII]

Execution with Invalid or Null as Argument

lemma $dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y}\text{-}nullstrict$ [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 $dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y}\text{-}at\text{-}pre\text{-}nullstrict$ [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 $dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y}\text{-}strict$ [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 $dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y}\text{-}at\text{-}pre\text{-}strict$ [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 $dot_{Person} \mathcal{BO} \mathcal{S} \mathcal{S}\text{-}nullstrict$ [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 $dot_{Person} \mathcal{BO} \mathcal{S} \mathcal{S}\text{-}at\text{-}pre\text{-}nullstrict$ [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 $dot_{Person} \mathcal{BO} \mathcal{S} \mathcal{S}\text{-}strict$ [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 $dot_{Person} \mathcal{BO} \mathcal{S} \mathcal{S}\text{-}at\text{-}pre\text{-}strict$ [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 $dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}\text{-}nullstrict$ [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 $dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}\text{-}at\text{-}pre\text{-}nullstrict$ [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 $dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}\text{-}strict$ [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 $dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}\text{-}at\text{-}pre\text{-}strict$ [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)

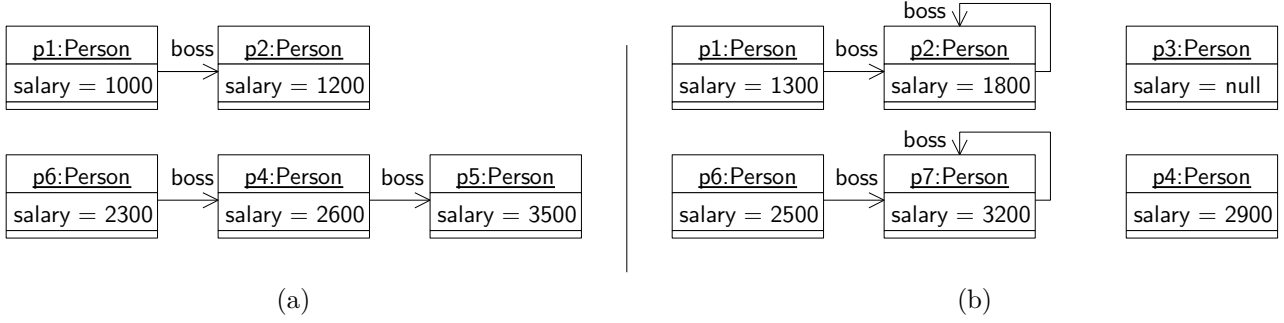


Figure B.2.: (a) pre-state σ_1 and (b) post-state σ'_1 .

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 = ( $\lambda$  . . [1000])
definition OclInt1200 (1200) where OclInt1200 = ( $\lambda$  . . [1200])
definition OclInt1300 (1300) where OclInt1300 = ( $\lambda$  . . [1300])
definition OclInt1800 (1800) where OclInt1800 = ( $\lambda$  . . [1800])
definition OclInt2600 (2600) where OclInt2600 = ( $\lambda$  . . [2600])
definition OclInt2900 (2900) where OclInt2900 = ( $\lambda$  . . [2900])
definition OclInt3200 (3200) where OclInt3200 = ( $\lambda$  . . [3200])
definition OclInt3500 (3500) where OclInt3500 = ( $\lambda$  . . [3500])

```

```

definition oid0  $\equiv$  0
definition oid1  $\equiv$  1
definition oid2  $\equiv$  2
definition oid3  $\equiv$  3
definition oid4  $\equiv$  4
definition oid5  $\equiv$  5
definition oid6  $\equiv$  6
definition oid7  $\equiv$  7
definition oid8  $\equiv$  8

```

```

definition person1  $\equiv$  mkPerson oid0 [1300]
definition person2  $\equiv$  mkPerson oid1 [1800]
definition person3  $\equiv$  mkPerson oid2 None
definition person4  $\equiv$  mkPerson oid3 [2900]
definition person5  $\equiv$  mkPerson oid4 [3500]
definition person6  $\equiv$  mkPerson oid5 [2500]
definition person7  $\equiv$  mkOclAny oid6 [3200]
definition person8  $\equiv$  mkOclAny oid7 None
definition person9  $\equiv$  mkPerson oid8 [0]

```

```

definition
   $\sigma_1 \equiv (\text{heap} = \text{empty}(\text{oid0} \mapsto \text{inPerson} (\text{mkPerson } \text{oid0} \text{ } [1000])))$ 

```

$(oid1 \mapsto in_{Person} (mk_{Person} oid1 \ [1200]))$
 $(*oid2*)$
 $(oid3 \mapsto in_{Person} (mk_{Person} oid3 \ [2600]))$
 $(oid4 \mapsto in_{Person} person5)$
 $(oid5 \mapsto in_{Person} (mk_{Person} oid5 \ [2300]))$
 $(*oid6*)$
 $(*oid7*)$
 $(oid8 \mapsto in_{Person} person9),$
 $assocs = empty(oid_{Person} \mathcal{B}\mathcal{O}\mathcal{S}\mathcal{S} \mapsto [[oid0],[oid1]], [[oid3],[oid4]], [[oid5],[oid3]]])$

definition

$\sigma_1' \equiv () \text{ heap} = empty(oid0 \mapsto in_{Person} person1)$
 $(oid1 \mapsto in_{Person} person2)$
 $(oid2 \mapsto in_{Person} person3)$
 $(oid3 \mapsto in_{Person} person4)$
 $(*oid4*)$
 $(oid5 \mapsto in_{Person} person6)$
 $(oid6 \mapsto in_{OclAny} person7)$
 $(oid7 \mapsto in_{OclAny} person8)$
 $(oid8 \mapsto in_{Person} person9),$
 $assocs = empty(oid_{Person} \mathcal{B}\mathcal{O}\mathcal{S}\mathcal{S} \mapsto [[oid0],[oid1]], [[oid1],[oid1]], [[oid5],[oid6]], [[oid6],[oid6]]))$

definition $\sigma_0 \equiv () \text{ heap} = empty, assocs = empty$

lemma *basic- τ -wff*: $WFF(\sigma_1, \sigma_1')$

by(*auto simp*: $WFF\text{-}def \ \sigma_1\text{-}def \ \sigma_1'\text{-}def$
 $oid0\text{-}def \ oid1\text{-}def \ oid2\text{-}def \ oid3\text{-}def \ oid4\text{-}def \ oid5\text{-}def \ oid6\text{-}def \ oid7\text{-}def \ oid8\text{-}def$
 $oid\text{-}of\text{-}\mathcal{A}\text{-}def \ oid\text{-}of\text{-}type_{Person}\text{-}def \ oid\text{-}of\text{-}type_{OclAny}\text{-}def$
 $person1\text{-}def \ person2\text{-}def \ person3\text{-}def \ person4\text{-}def$
 $person5\text{-}def \ person6\text{-}def \ person7\text{-}def \ person8\text{-}def \ person9\text{-}def$)

lemma [*simp,code-unfold*]: $dom \ (heap \ \sigma_1) = \{oid0,oid1,(*,oid2*)oid3,oid4,oid5(*,oid6,oid7*),oid8\}$

by(*auto simp*: $\sigma_1\text{-}def$)

lemma [*simp,code-unfold*]: $dom \ (heap \ \sigma_1') = \{oid0,oid1,oid2,oid3,(*,oid4*)oid5,oid6,oid7,oid8\}$

by(*auto simp*: $\sigma_1'\text{-}def$)

definition $X_{Person1} :: Person \equiv \lambda \cdot . \llbracket person1 \rrbracket$

definition $X_{Person2} :: Person \equiv \lambda \cdot . \llbracket person2 \rrbracket$

definition $X_{Person3} :: Person \equiv \lambda \cdot . \llbracket person3 \rrbracket$

definition $X_{Person4} :: Person \equiv \lambda \cdot . \llbracket person4 \rrbracket$

definition $X_{Person5} :: Person \equiv \lambda \cdot . \llbracket person5 \rrbracket$

definition $X_{Person6} :: Person \equiv \lambda \cdot . \llbracket person6 \rrbracket$

definition $X_{Person7} :: OclAny \equiv \lambda \cdot . \llbracket person7 \rrbracket$

definition $X_{Person8} :: OclAny \equiv \lambda \cdot . \llbracket person8 \rrbracket$

definition $X_{Person9} :: Person \equiv \lambda \cdot . \llbracket person9 \rrbracket$

lemma [code-unfold]: $((x::Person) \doteq y) = StrictRefEqObject\ x\ y$ **by** (simp only: StrictRefEqObject-Person)
lemma [code-unfold]: $((x::OclAny) \doteq y) = StrictRefEqObject\ x\ y$ **by** (simp only: StrictRefEqObject-OclAny)

lemmas [simp,code-unfold] =
OclAsTypeOclAny-OclAny
OclAsTypeOclAny-Person
OclAsTypePerson-OclAny
OclAsTypePerson-Person

OclIsTypeOfOclAny-OclAny
OclIsTypeOfOclAny-Person
OclIsTypeOfPerson-OclAny
OclIsTypeOfPerson-Person

OclIsKindOfOclAny-OclAny
OclIsKindOfOclAny-Person
OclIsKindOfPerson-OclAny
OclIsKindOfPerson-Person

Assert $\wedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1}.salary <> \mathbf{1000})$
Assert $\wedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1}.salary \doteq \mathbf{1300})$
Assert $\wedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1}.salary@pre \doteq \mathbf{1000})$
Assert $\wedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1}.salary@pre <> \mathbf{1300})$

lemma $(\sigma_1, \sigma_1') \models (X_{Person1}.ocIsMaintained())$
by (simp add: OclValid-def OclIsMaintained-def
 σ_1 -def σ_1' -def
XPerson1-def person1-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def
oid-of-option-def oid-of-typePerson-def)

lemma $\wedge_{s_{pre}\ s_{post}} \cdot (s_{pre}, s_{post}) \models ((X_{Person1}.oclAsType(OclAny).oclAsType(Person)) \doteq X_{Person1})$
by (rule up-down-cast-Person-OclAny-Person', simp add: *XPerson1-def*)
Assert $\wedge_{s_{pre}\ s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person1}.ocIsTypeOf(Person))$
Assert $\wedge_{s_{pre}\ s_{post}} \cdot (s_{pre}, s_{post}) \models \text{not}(X_{Person1}.ocIsTypeOf(OclAny))$
Assert $\wedge_{s_{pre}\ s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person1}.ocIsKindOf(Person))$
Assert $\wedge_{s_{pre}\ s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person1}.ocIsKindOf(OclAny))$
Assert $\wedge_{s_{pre}\ s_{post}} \cdot (s_{pre}, s_{post}) \models \text{not}(X_{Person1}.oclAsType(OclAny).ocIsTypeOf(OclAny))$

Assert $\wedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person2}.salary \doteq \mathbf{1800})$
Assert $\wedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person2}.salary@pre \doteq \mathbf{1200})$

lemma $(\sigma_1, \sigma_1') \models (X_{Person2}.ocIsMaintained())$
by (simp add: OclValid-def OclIsMaintained-def
 σ_1 -def σ_1' -def
XPerson2-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 $\wedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person3}.salary \doteq null)$
Assert $\wedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models not(v(X_{Person3}.salary@pre))$
lemma $(\sigma_1, \sigma_1') \models (X_{Person3}.oclIsNew())$
by(simp add: OclValid-def OclIsNew-def
 σ_1 -def σ_1' -def
 $X_{Person3}$ -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 $(\sigma_1, \sigma_1') \models (X_{Person4}.oclIsMaintained())$
by(simp add: OclValid-def OclIsMaintained-def
 σ_1 -def σ_1' -def
 $X_{Person4}$ -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 $\wedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models not(v(X_{Person5}.salary))$
Assert $\wedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person5}.salary@pre \doteq 3500)$

lemma $(\sigma_1, \sigma_1') \models (X_{Person5}.oclIsDeleted())$
by(simp add: OclNot-def OclValid-def OclIsDeleted-def
 σ_1 -def σ_1' -def
 $X_{Person5}$ -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 $(\sigma_1, \sigma_1') \models (X_{Person6}.oclIsMaintained())$
by(simp add: OclValid-def OclIsMaintained-def
 σ_1 -def σ_1' -def
 $X_{Person6}$ -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 $\wedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models v(X_{Person7}.oclAsType(Person))$

lemma $\wedge_{s_{pre} s_{post}} (s_{pre}, s_{post}) \models ((X_{Person7} .oclAsType(Person) .oclAsType(OclAny) .oclAsType(Person)) \doteq (X_{Person7} .oclAsType(Person)))$
by(rule up-down-cast-Person-OclAny-Person', simp add: X_{Person7}-def OclValid-def valid-def person7-def)
lemma $(\sigma_1, \sigma_1') \models (X_{Person7} .oclIsNew())$
by(simp add: OclValid-def OclIsNew-def
 σ_1 -def σ_1' -def
X_{Person7}-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 $\wedge_{s_{pre} s_{post}} (s_{pre}, s_{post}) \models (X_{Person8} <> X_{Person7})$
Assert $\wedge_{s_{pre} s_{post}} (s_{pre}, s_{post}) \models \text{not}(\vee(X_{Person8} .oclAsType(Person)))$
Assert $\wedge_{s_{pre} s_{post}} (s_{pre}, s_{post}) \models (X_{Person8} .oclIsTypeOf(OclAny))$
Assert $\wedge_{s_{pre} s_{post}} (s_{pre}, s_{post}) \models \text{not}(X_{Person8} .oclIsTypeOf(Person))$
Assert $\wedge_{s_{pre} s_{post}} (s_{pre}, s_{post}) \models \text{not}(X_{Person8} .oclIsKindOf(Person))$
Assert $\wedge_{s_{pre} s_{post}} (s_{pre}, s_{post}) \models (X_{Person8} .oclIsKindOf(OclAny))$

lemma σ -modifiedonly: $(\sigma_1, \sigma_1') \models (\text{Set}\{ X_{Person1} .oclAsType(OclAny) , X_{Person2} .oclAsType(OclAny) (*, X_{Person3} .oclAsType(OclAny)*) , X_{Person4} .oclAsType(OclAny) (*, X_{Person5} .oclAsType(OclAny)*) , X_{Person6} .oclAsType(OclAny) (*, X_{Person7} .oclAsType(OclAny)*) (*, X_{Person8} .oclAsType(OclAny)*) (*, X_{Person9} .oclAsType(OclAny)*) \} - > oclIsModifiedOnly())$
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_{Person1}-def X_{Person2}-def X_{Person3}-def X_{Person4}-def
X_{Person5}-def X_{Person6}-def X_{Person7}-def X_{Person8}-def X_{Person9}-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 σ_1' -def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)
done

lemma $(\sigma_1, \sigma_1') \models ((X_{Person9} @pre (\lambda x. \lfloor OclAsType_{Person} \mathfrak{A} x \rfloor)) \triangleq X_{Person9})$
by(simp add: OclSelf-at-pre-def σ_1 -def oid-of-option-def oid-of-type_{Person}-def
X_{Person9}-def person9-def oid8-def OclValid-def StrongEq-def OclAsType_{Person}- \mathfrak{A} -def)

lemma $(\sigma_1, \sigma_1') \models ((X_{Person9} @post (\lambda x. \lfloor OclAsType_{Person} \mathfrak{A} x \rfloor)) \triangleq X_{Person9})$

by(simp add: OclSelf-at-post-def σ_1 '-def oid-of-option-def oid-of-type_{Person}-def
 $X_{Person9}$ -def person9-def oid8-def OclValid-def StrongEq-def OclAsType_{Person}- \mathfrak{A} -def)

lemma (σ_1, σ_1') $\models (((X_{Person9} .oclAsType(OclAny)) @pre (\lambda x. [OclAsType_{OclAny} \mathfrak{A} x])) \triangleq$
 $((X_{Person9} .oclAsType(OclAny)) @post (\lambda x. [OclAsType_{OclAny} \mathfrak{A} x])))$

proof –

have including4 : $\bigwedge a b c d \tau$.

$Set\{\lambda \tau. [[a]], \lambda \tau. [[b]], \lambda \tau. [[c]], \lambda \tau. [[d]]\} \tau = Abs-Set_{base} [[\{[[a]], [[b]], [[c]], [[d]]\}]]$

apply(subst abs-rep-simp'[symmetric], simp)

apply(simp add: OclIncluding-rep-set mtSet-rep-set)

by(rule arg-cong[of - $\lambda x. (Abs-Set_{base}([x]))$], auto)

have excluding1 : $\bigwedge s a b c d e \tau$.

$(\lambda \tau. Abs-Set_{base} [[\{[[a]], [[b]], [[c]], [[d]]\}]] -> excluding(\lambda \tau. [[e]]) \tau =$
 $Abs-Set_{base} [[\{[[a]], [[b]], [[c]], [[d]]\} - \{[[e]]\}]]$

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** $X = Set\{ X_{Person1} .oclAsType(OclAny)$

, $X_{Person2} .oclAsType(OclAny)$

(*, $X_{Person3} .oclAsType(OclAny)$ *)

, $X_{Person4} .oclAsType(OclAny)$

(*, $X_{Person5} .oclAsType(OclAny)$ *)

, $X_{Person6} .oclAsType(OclAny)$

(*, $X_{Person7} .oclAsType(OclAny)$ *)

(*, $X_{Person8} .oclAsType(OclAny)$ *)

(*, $X_{Person9} .oclAsType(OclAny)$ *)}])

apply(cut-tac σ -modifiedonly)

apply(simp only: OclValid-def

$X_{Person1}$ -def $X_{Person2}$ -def $X_{Person3}$ -def $X_{Person4}$ -def

$X_{Person5}$ -def $X_{Person6}$ -def $X_{Person7}$ -def $X_{Person8}$ -def $X_{Person9}$ -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: $X_{Person1}$ -def $X_{Person2}$ -def $X_{Person3}$ -def $X_{Person4}$ -def

$X_{Person5}$ -def $X_{Person6}$ -def $X_{Person7}$ -def $X_{Person8}$ -def $X_{Person9}$ -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: StrictRefEqObject-def oid-of-option-def oid-of-typeOclAny-def OclNot-def OclValid-def
    null-option-def bot-option-def)
done
qed

lemma perm- $\sigma_1'$  :  $\sigma_1' = () \text{ heap} = \text{empty}$ 
    (oid8  $\mapsto$  inPerson person9)
    (oid7  $\mapsto$  inOclAny person8)
    (oid6  $\mapsto$  inOclAny person7)
    (oid5  $\mapsto$  inPerson person6)
    (*oid4*)
    (oid3  $\mapsto$  inPerson person4)
    (oid2  $\mapsto$  inPerson person3)
    (oid1  $\mapsto$  inPerson person2)
    (oid0  $\mapsto$  inPerson 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  $\wedge \sigma_1$ .
    ( $\sigma_1, \sigma_1'$ )  $\models$  (Person .allInstances()  $\doteq$  Set{ XPerson1, XPerson2, XPerson3, XPerson4(*, XPerson5*), XPerson6,
    XPerson7 .oclAsType(Person)(* , XPerson8*), XPerson9 })
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
    XPerson1-def XPerson2-def XPerson3-def XPerson4-def
    XPerson5-def XPerson6-def XPerson7-def XPerson8-def XPerson9-def
    person7-def)

```

```

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- $\mathcal{A}$ -def, simp, rule const-StrictRefEqSet-including,
simp, simp, simp, rule OclIncluding-cong, simp, simp)
apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- $\mathcal{A}$ -def, simp, rule const-StrictRefEqSet-including,
simp, simp, simp, rule OclIncluding-cong, simp, simp)
apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- $\mathcal{A}$ -def, simp, rule const-StrictRefEqSet-including,
simp, simp, simp, rule OclIncluding-cong, simp, simp)
apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- $\mathcal{A}$ -def, simp, rule const-StrictRefEqSet-including,
simp, simp, simp, rule OclIncluding-cong, simp, simp)
apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- $\mathcal{A}$ -def, simp, rule const-StrictRefEqSet-including,
simp, simp, simp, rule OclIncluding-cong, simp, simp)
apply(subst state-update-vs-allInstances-at-post-ntc, simp, simp add: OclAsTypePerson- $\mathcal{A}$ -def
person8-def, simp, rule const-StrictRefEqSet-including, simp, simp, simp)
apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- $\mathcal{A}$ -def, simp, rule const-StrictRefEqSet-including,
simp, simp, simp, rule OclIncluding-cong, simp, simp)
apply(rule state-update-vs-allInstances-at-post-empty)
by(simp-all add: OclAsTypePerson- $\mathcal{A}$ -def)

lemma  $\wedge \sigma_1$ .
 $(\sigma_1, \sigma_1') \models (\text{OclAny} . \text{allInstances}() \doteq \text{Set}\{ X_{\text{Person}1} . \text{oclAsType}(\text{OclAny}), X_{\text{Person}2} . \text{oclAsType}(\text{OclAny}),$ 
 $X_{\text{Person}3} . \text{oclAsType}(\text{OclAny}), X_{\text{Person}4} . \text{oclAsType}(\text{OclAny})$ 
 $(*, X_{\text{Person}5*}), X_{\text{Person}6} . \text{oclAsType}(\text{OclAny}),$ 
 $X_{\text{Person}7}, X_{\text{Person}8}, X_{\text{Person}9} . \text{oclAsType}(\text{OclAny}) \})$ 

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
XPerson1-def XPerson2-def XPerson3-def XPerson4-def XPerson5-def XPerson6-def XPerson7-def XPerson8-def
XPerson9-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: OclAsTypeOclAny- $\mathcal{A}$ -def, simp, rule const-StrictRefEqSet-including,
simp, simp, simp, rule OclIncluding-cong, simp, simp)+
apply(rule state-update-vs-allInstances-at-post-empty)
by(simp-all add: OclAsTypeOclAny- $\mathcal{A}$ -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 $Person\text{-}label_{inv} :: Person \Rightarrow Boolean$

where $Person\text{-}label_{inv}(self) \equiv$
 $(self .boss <> null \text{ implies } (self .salary \leq_{int} ((self .boss) .salary)))$

definition $Person\text{-}label_{invAT\ pre} :: Person \Rightarrow Boolean$

where $Person\text{-}label_{invAT\ pre}(self) \equiv$
 $(self .boss@pre <> null \text{ implies } (self .salary@pre \leq_{int} ((self .boss@pre) .salary@pre)))$

definition $Person\text{-}label_{global\ inv} :: Boolean$

where $Person\text{-}label_{global\ inv} \equiv (Person .allInstances() \text{--}> \text{forAll}(x \mid Person\text{-}label_{inv}(x)) \text{ and }$
 $(Person .allInstances@pre() \text{--}> \text{forAll}(x \mid Person\text{-}label_{invAT\ pre}(x))))$

lemma $\tau \models \delta(X .boss) \implies \tau \models Person .allInstances() \text{--}> \text{includes}(X .boss) \wedge$
 $\tau \models Person .allInstances() \text{--}> \text{includes}(X)$

sorry

lemma $REC\text{-}pre : \tau \models Person\text{-}label_{global\ inv}$

$\implies \tau \models Person .allInstances() \text{--}> \text{includes}(X) \text{ (* } X \text{ represented object in state *)}$
 $\implies \exists REC. \tau \models REC(X) \triangleq (Person\text{-}label_{inv}(X) \text{ and } (X .boss <> null \text{ implies } REC(X .boss)))$

sorry

This allows to state a predicate:

axiomatization $inv_{Person\text{-}label} :: Person \Rightarrow Boolean$

where $inv_{Person\text{-}label}\text{-}def:$

$(\tau \models Person .allInstances() \text{--}> \text{includes}(self)) \implies$
 $(\tau \models (inv_{Person\text{-}label}(self) \triangleq (self .boss <> null \text{ implies }$
 $(self .salary \leq_{int} ((self .boss) .salary)) \text{ and }$
 $inv_{Person\text{-}label}(self .boss))))$

axiomatization $inv_{Person\text{-}labelAT\ pre} :: Person \Rightarrow Boolean$

where $inv_{Person\text{-}labelAT\ pre}\text{-}def:$

$(\tau \models Person .allInstances@pre() \text{--}> \text{includes}(self)) \implies$
 $(\tau \models (inv_{Person\text{-}labelAT\ pre}(self) \triangleq (self .boss@pre <> null \text{ implies }$

$$(self.salary@pre \leq_{int} ((self.boss@pre).salary@pre)) \text{ and } inv_{Person-labelAT\ pre}(self.boss@pre)))$$

lemma *inv-1* :

$$\begin{aligned} (\tau \models Person.allInstances() \rightarrow includes(self)) \implies \\ (\tau \models inv_{Person-label}(self) = ((\tau \models (self.boss \doteq null)) \vee \\ (\tau \models (self.boss <> null) \wedge \\ \tau \models ((self.salary) \leq_{int} (self.boss.salary)) \wedge \\ \tau \models (inv_{Person-label}(self.boss)))))) \end{aligned}$$

sorry

lemma *inv-2* :

$$\begin{aligned} (\tau \models Person.allInstances@pre() \rightarrow includes(self)) \implies \\ (\tau \models inv_{Person-labelAT\ pre}(self) = ((\tau \models (self.boss@pre \doteq null)) \vee \\ (\tau \models (self.boss@pre <> null) \wedge \\ (\tau \models (self.boss@pre.salary@pre \leq_{int} self.salary@pre)) \wedge \\ (\tau \models (inv_{Person-labelAT\ pre}(self.boss@pre)))))) \end{aligned}$$

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 (\mathbb{A})*st* \Rightarrow *bool* **where**

$$\begin{aligned} (\tau \models (\delta\ self)) \implies ((\tau \models (self.boss \doteq null)) \vee \\ (\tau \models (self.boss <> null) \wedge (\tau \models (self.boss.salary \leq_{int} self.salary)) \wedge \\ ((inv(self.boss))\tau))) \\ \implies (inv\ self\ \tau) \end{aligned}$$

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* \Rightarrow *Set-Integer* ((*I*(-).*contents'*(')) 50)

where *contents-def*:

$$\begin{aligned} (self.contents()) = (\lambda\ \tau. (if\ \tau \models (\delta\ self) \\ then\ SOME\ res. ((\tau \models true) \wedge \\ (\tau \models (\lambda\ - . res) \triangleq if\ (self.boss \doteq null) \\ then\ (Set\{self.salary\}) \end{aligned}$$

```

else (self .boss .contents()
      ->including(self .salary))
endif))
else invalid  $\tau$ )

```

interpretation *contents* : contract0 *contents* λ self. true

```

 $\lambda$  self res. res  $\triangleq$  if (self .boss  $\doteq$  null)
  then (Set{self .salary})
  else (self .boss .contents()
        ->including(self .salary))
endif

```

proof (*unfold-locales*)

show \wedge self τ . true $\tau =$ true τ **by** auto

next

show \wedge self. $\forall \sigma \sigma' \sigma''$. $((\sigma, \sigma') \models \text{true}) = ((\sigma, \sigma'') \models \text{true})$ **by** auto

next

```

show  $\wedge$ self. self .contents()  $\equiv$ 
 $\lambda \tau$ . if  $\tau \models \delta$  self
  then SOME res.
     $\tau \models \text{true} \wedge$ 
     $\tau \models (\lambda -. res) \triangleq$  (if self .boss  $\doteq$  null then Set{self .salary}
      else self .boss .contents() ->including(self .salary)
    endif)
  else invalid  $\tau$ 

```

by(auto simp: contents-def)

next

```

have A:  $\wedge$ self  $\tau$ .  $((\lambda -. self \tau) .boss \doteq \text{null}) \tau = (\lambda -. (self .boss \doteq \text{null}) \tau) \tau$  sorry
have B:  $\wedge$ self  $\tau$ .  $(\lambda -. \text{Set}\{(\lambda -. self \tau) .salary\} \tau) = (\lambda -. \text{Set}\{self .salary\} \tau)$  sorry
have C:  $\wedge$ self  $\tau$ .  $((\lambda -. self \tau) .boss .contents() ->including((\lambda -. self \tau) .salary) \tau) =$ 
   $(self .boss .contents() ->including(self .salary) \tau)$  sorry

```

show \wedge self res τ .

```

(res  $\triangleq$  if (self .boss)  $\doteq$  null then Set{self .salary}
  else self .boss .contents() ->including(self .salary) endif)  $\tau =$ 
 $((\lambda -. res \tau) \triangleq$  if  $(\lambda -. self \tau) .boss \doteq$  null then Set{ $(\lambda -. self \tau) .salary$ }
  else  $(\lambda -. self \tau) .boss .contents() ->including((\lambda -. self \tau) .salary)$  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

Specializing $\llbracket \text{cp } E; \tau \models \delta \text{ self}; \tau \models \text{true}; \tau \models \text{POST}' \text{ self}; \wedge \text{res. } (res \triangleq \text{if self.boss} \doteq \text{null then Set}\{self.salary\} \text{ else self.boss.contents() ->including(self.salary) endif}) = (\text{POST}' \text{ self and } (res \triangleq \text{BODY self})) \rrbracket \implies (\tau \models E (self.contents())) = (\tau \models E (\text{BODY self}))$, one gets the following more practical rewrite rule that is amenable to symbolic evaluation:

theorem *unfold-contents* :

assumes cp E

```

and    $\tau \models \delta \text{ self}$ 
shows ( $\tau \models E (\text{self} .\text{contents}()) =$ 
  ( $\tau \models E (\text{if self} .\text{boss} \doteq \text{null}$ 
     $\text{then Set}\{\text{self} .\text{salary}\}$ 
     $\text{else self} .\text{boss} .\text{contents}() \text{-->including}(\text{self} .\text{salary}) \text{ endif})$ )
by(rule contents.unfold2[of - -  $\lambda X. \text{true}$ ], simp-all add: assms)

```

Since we have only one interpretation function, we need the corresponding operation on the pre-state:

```

consts contentsATpre :: Person  $\Rightarrow$  Set-Integer ((1(-).contents@pre'()) 50)

```

axiomatization where contentsATpre-def:

```

( $\text{self} .\text{contents}@pre()$ ) = ( $\lambda \tau.$ 
  ( $\text{if } \tau \models (\delta \text{ self})$ 
     $\text{then SOME res.} ((\tau \models \text{true}) \wedge$ 
      ( $\tau \models ((\lambda -. \text{res}) \triangleq \text{if } (\text{self} .\text{boss}@pre \doteq \text{null}) (* \text{pre} *)$ 
         $\text{then Set}\{(\text{self} .\text{salary}@pre)\}$ 
         $\text{else } (\text{self} .\text{boss}@pre .\text{contents}@pre()$ 
           $\text{-->including}(\text{self} .\text{salary}@pre)$ 
           $\text{endif}))$ )
     $\text{else invalid } \tau))$ 

```

```

interpretation contentsATpre : contract0 contentsATpre  $\lambda \text{ self} . \text{true}$ 
   $\lambda \text{ self res. res} \triangleq \text{if } (\text{self} .\text{boss}@pre \doteq \text{null})$ 
     $\text{then } (\text{Set}\{\text{self} .\text{salary}@pre\})$ 
     $\text{else } (\text{self} .\text{boss}@pre .\text{contents}@pre()$ 
       $\text{-->including}(\text{self} .\text{salary}@pre))$ 
     $\text{endif}$ 

```

proof (unfold-locales)

show $\bigwedge \text{self } \tau. \text{true } \tau = \text{true } \tau$ **by** auto

next

show $\bigwedge \text{self} . \forall \sigma \sigma' \sigma''. ((\sigma, \sigma') \models \text{true}) = ((\sigma, \sigma'') \models \text{true})$ **by** auto

next

```

show  $\bigwedge \text{self} . \text{self} .\text{contents}@pre() \equiv$ 
   $\lambda \tau. \text{if } \tau \models \delta \text{ self}$ 
     $\text{then SOME res.}$ 
       $\tau \models \text{true} \wedge$ 
       $\tau \models ((\lambda -. \text{res}) \triangleq (\text{if self} .\text{boss}@pre \doteq \text{null} \text{ then Set}\{\text{self} .\text{salary}@pre\}$ 
         $\text{else self} .\text{boss}@pre .\text{contents}@pre() \text{-->including}(\text{self} .\text{salary}@pre)$ 
         $\text{endif}))$ 
     $\text{else invalid } \tau$ 

```

by(auto simp: contentsATpre-def)

next

```

have A:  $\bigwedge \text{self } \tau. ((\lambda -. \text{self } \tau) .\text{boss}@pre \doteq \text{null}) \tau = (\lambda -. (\text{self} .\text{boss}@pre \doteq \text{null}) \tau) \tau$  sorry
have B:  $\bigwedge \text{self } \tau. (\lambda -. \text{Set}\{(\lambda -. \text{self } \tau) .\text{salary}@pre\} \tau) = (\lambda -. \text{Set}\{\text{self} .\text{salary}@pre\} \tau) \tau$  sorry
have C:  $\bigwedge \text{self } \tau. ((\lambda -. \text{self } \tau) .\text{boss}@pre .\text{contents}@pre() \text{-->including}((\lambda -. \text{self } \tau) .\text{salary}@pre) \tau) =$ 
   $(\text{self} .\text{boss}@pre .\text{contents}@pre() \text{-->including}(\text{self} .\text{salary}@pre) \tau) \tau$  sorry
show  $\bigwedge \text{self res } \tau.$ 
   $(\text{res} \triangleq \text{if } (\text{self} .\text{boss}@pre) \doteq \text{null} \text{ then Set}\{\text{self} .\text{salary}@pre\}$ 

```



```

    else self .boss@pre .contents@pre()->including(self .salary@pre) endif)  $\tau$  =
    (( $\lambda$ -. res  $\tau$ )  $\triangleq$  if ( $\lambda$ -. self  $\tau$ ) .boss@pre  $\doteq$  null then Set{( $\lambda$ -. self  $\tau$ ) .salary@pre}
      else( $\lambda$ -. self  $\tau$ ) .boss@pre .contents@pre()->including(( $\lambda$ -. self  $\tau$ ) .salary@pre) 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  $\tau \models \delta$  self
shows ( $\tau \models E$  (self .contents@pre())) =
  ( $\tau \models E$  (if self .boss@pre  $\doteq$  null
    then Set{self .salary@pre}
    else self .boss@pre .contents@pre()->including(self .salary@pre) endif))
by(rule contentsATpre.unfold2[of - -  $\lambda$  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  $\Rightarrow$  Integer  $\Rightarrow$  Void ((I(-).insert'(-)) 50)
where self .insert(x)  $\equiv$ 
  ( $\lambda$   $\tau$ . if ( $\tau \models (\delta$  self))  $\wedge$  ( $\tau \models v$  x)
    then SOME res. ( $\tau \models$  true  $\wedge$ 
      ( $\tau \models ((self).contents() \triangleq (self).contents@pre()->including(x)))$ )
    else invalid  $\tau$ )

```

The semantic consequences of this definition were computed inside this locale interpretation:

```

interpretation insert : contract1 insert  $\lambda$  self x. true
   $\lambda$  self x res. ((self .contents())  $\triangleq$ 
    (self .contents@pre()->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
<i>insert.strict0</i>	$(invalid.insert(X)) = invalid$
<i>insert.nullstrict0</i>	$(null.insert(X)) = invalid$
<i>insert.strict1</i>	$(self.insert(invalid)) = invalid$
<i>insert.cpPRE</i>	$true \tau = true \tau$
<i>insert.cpPOST</i>	$(self.contents() \triangleq self.contents@pre() \rightarrow including(a1.0)) \tau = (\lambda -. self \tau.contents() \triangleq \lambda -. self \tau.contents@pre() \rightarrow including(\lambda -. a1.0 \tau)) \tau$
<i>insert.cp-pre</i>	$\llbracket cp \ self'; cp \ a1' \rrbracket \implies cp \ (\lambda X. self' \tau.contents() \triangleq self' X.contents@pre() \rightarrow including(a1' X))$
<i>insert.cp-post</i>	$\llbracket cp \ self'; cp \ a1'; cp \ res \rrbracket \implies cp \ (\lambda X. self' X.contents() \triangleq self' X.contents@pre() \rightarrow including(a1' X))$
<i>insert.cp</i>	$\llbracket cp \ self'; cp \ a1'; cp \ res \rrbracket \implies cp \ (\lambda X. self' X.insert(a1' X))$
<i>insert.cp0</i>	$(self.insert(a1.0)) \tau = (\lambda -. self \tau.insert(\lambda -. a1.0 \tau)) \tau$
<i>insert.def-scheme</i>	$self.insert(a1.0) \equiv \lambda \tau. \text{if } \tau \models \delta \ self \wedge \tau \models v \ a1.0 \text{ then } SOME \ res. \tau \models true \wedge \tau \models self.contents() \triangleq self.contents@pre() \rightarrow including(a1.0) \text{ else } invalid \ \tau$
<i>insert.unfold</i>	$\llbracket cp \ E; \tau \models \delta \ self \wedge \tau \models v \ a1.0; \tau \models true; \exists res. \tau \models self.contents() \triangleq self.contents@pre() \rightarrow including(a1.0); \wedge res. \tau \models self.contents() \triangleq self.contents@pre() \rightarrow including(a1.0) \implies \tau \models E \ (\lambda -. res) \rrbracket \implies \tau \models E (self.insert(a1.0))$
<i>insert.unfold2</i>	$\llbracket cp \ E; \tau \models \delta \ self \wedge \tau \models v \ a1.0; \tau \models true; \tau \models POST' \ self \ a1.0; \wedge res. (self.contents() \triangleq self.contents@pre() \rightarrow including(a1.0)) = (POST' \ self \ a1.0 \text{ and } (res \triangleq BODY \ self \ a1.0)) \rrbracket \implies (\tau \models E (self.insert(a1.0))) = (\tau \models E (BODY \ self \ a1.0))$

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

```

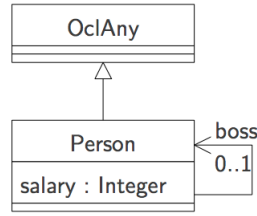


Figure B.3.: A simple UML class model drawn from Figure 7.3, page 20 of [28].

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 typePerson = mkPerson oid
                    int option
                    oid option
  
```

```
datatype typeOclAny = mkOclAny oid
  (int option × 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  $\mathfrak{A}$  = inPerson typePerson | inOclAny typeOclAny
```

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     = ( $\mathfrak{A}$ , typeOclAny option option) val
type-synonym Person     = ( $\mathfrak{A}$ , typePerson option option) val
type-synonym Set-Integer = ( $\mathfrak{A}$ , int option option) Set
type-synonym Set-Person  = ( $\mathfrak{A}$ , typePerson 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 “oclaney,” i. e., each class type has to provide a function *oid-of* yielding the object id (oid) of the object.

```
instantiation typePerson :: object
```

```
begin
```

```
  definition oid-of-typePerson-def: oid-of x = (case x of mkPerson oid - - ⇒ oid)
```

```
  instance ..
```

```
end
```

```
instantiation typeOclAny :: object
```

```
begin
```

```
  definition oid-of-typeOclAny-def: oid-of x = (case x of mkOclAny oid - - ⇒ oid)
```

```
  instance ..
```

```
end
```

```
instantiation  $\mathfrak{A}$  :: object
```

```
begin
```

```
  definition oid-of- $\mathfrak{A}$ -def: oid-of x = (case x of
    inPerson person ⇒ oid-of person
    | inOclAny oclany ⇒ oid-of oclany)
```

```
  instance ..
```

```
end
```

B.5.3. Instantiation of the Generic Strict Equality

We instantiate the referential equality on *Person* and *OclAny*

```
defs(overloaded) StrictRefEqObject-Person : (x::Person)  $\doteq$  y  $\equiv$  StrictRefEqObject x y
defs(overloaded) StrictRefEqObject-OclAny : (x::OclAny)  $\doteq$  y  $\equiv$  StrictRefEqObject x y
```

lemmas

```
cp-StrictRefEqObject [of x::Person y::Person  $\tau$ ,
  simplified StrictRefEqObject-Person [symmetric]]
cp-intro (9) [of P::Person  $\Rightarrow$  Person Q::Person  $\Rightarrow$  Person,
  simplified StrictRefEqObject-Person [symmetric]]
StrictRefEqObject-def [of x::Person y::Person,
  simplified StrictRefEqObject-Person [symmetric]]
StrictRefEqObject-defargs [of - x::Person y::Person,
  simplified StrictRefEqObject-Person [symmetric]]
StrictRefEqObject-strict1
  [of x::Person,
  simplified StrictRefEqObject-Person [symmetric]]
StrictRefEqObject-strict2
  [of x::Person,
  simplified StrictRefEqObject-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 OclAsTypeOclAny :: ' $\alpha \Rightarrow$  OclAny ((-) .oclAsType'(OclAny'))
consts OclAsTypePerson :: ' $\alpha \Rightarrow$  Person ((-) .oclAsType'(Person'))
```

```
definition OclAsTypeOclAny- $\mathcal{A}$  = ( $\lambda u$ . [case u of inOclAny a  $\Rightarrow$  a
  | inPerson (mkPerson oid a b)  $\Rightarrow$  mkOclAny oid [(a,b)]])
```

```
lemma OclAsTypeOclAny- $\mathcal{A}$ -some: OclAsTypeOclAny- $\mathcal{A}$  x  $\neq$  None
by (simp add: OclAsTypeOclAny- $\mathcal{A}$ -def)
```

```
defs (overloaded) OclAsTypeOclAny-OclAny:
  (X::OclAny) .oclAsType(OclAny)  $\equiv$  X
```

```
defs (overloaded) OclAsTypeOclAny-Person:
  (X::Person) .oclAsType(OclAny)  $\equiv$ 
  ( $\lambda \tau$ . case X  $\tau$  of
     $\perp \Rightarrow$  invalid  $\tau$ 
```

$$\begin{aligned} &| \lfloor \perp \rfloor \Rightarrow \text{null } \tau \\ &| \lfloor \text{mk}_{\text{Person}} \text{ oid } a \ b \rfloor \rfloor \Rightarrow \lfloor \lfloor \text{mk}_{\text{OclAny}} \text{ oid } \lfloor (a,b) \rfloor \rfloor \rfloor \end{aligned}$$

definition $\text{OclAsType}_{\text{Person-OclAny}} = (\lambda u. \text{case } u \text{ of } \text{in}_{\text{Person}} p \Rightarrow \lfloor p \rfloor$
 $\quad \quad \quad | \text{in}_{\text{OclAny}} (\text{mk}_{\text{OclAny}} \text{ oid } \lfloor (a,b) \rfloor) \Rightarrow \lfloor \text{mk}_{\text{Person}} \text{ oid } a \ b \rfloor$
 $\quad \quad \quad | - \Rightarrow \text{None})$

defs (overloaded) $\text{OclAsType}_{\text{Person-OclAny}}$:
 $(X::\text{OclAny}) .\text{oclAsType}(\text{Person}) \equiv$
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \lfloor \perp \rfloor \Rightarrow \text{null } \tau$
 $\quad | \lfloor \text{mk}_{\text{OclAny}} \text{ oid } \perp \rfloor \rfloor \Rightarrow \text{invalid } \tau \quad (* \text{down-cast exception} *)$
 $\quad | \lfloor \text{mk}_{\text{OclAny}} \text{ oid } \lfloor (a,b) \rfloor \rfloor \rfloor \Rightarrow \lfloor \lfloor \text{mk}_{\text{Person}} \text{ oid } a \ b \rfloor \rfloor)$

defs (overloaded) $\text{OclAsType}_{\text{Person-Person}}$:
 $(X::\text{Person}) .\text{oclAsType}(\text{Person}) \equiv X$

lemmas [simp] =
 $\text{OclAsType}_{\text{OclAny-OclAny}}$
 $\text{OclAsType}_{\text{Person-Person}}$

Context Passing

lemma $\text{cp-OclAsType}_{\text{OclAny-Person-Person}}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P (X::\text{Person})::\text{Person}) .\text{oclAsType}(\text{OclAny}))$
by(rule cpI1 , $\text{simp-all add: OclAsType}_{\text{OclAny-Person}}$)
lemma $\text{cp-OclAsType}_{\text{OclAny-OclAny-OclAny}}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P (X::\text{OclAny})::\text{OclAny}) .\text{oclAsType}(\text{OclAny}))$
by(rule cpI1 , $\text{simp-all add: OclAsType}_{\text{OclAny-OclAny}}$)
lemma $\text{cp-OclAsType}_{\text{Person-Person-Person}}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P (X::\text{Person})::\text{Person}) .\text{oclAsType}(\text{Person}))$
by(rule cpI1 , $\text{simp-all add: OclAsType}_{\text{Person-Person}}$)
lemma $\text{cp-OclAsType}_{\text{Person-OclAny-OclAny}}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P (X::\text{OclAny})::\text{OclAny}) .\text{oclAsType}(\text{Person}))$
by(rule cpI1 , $\text{simp-all add: OclAsType}_{\text{Person-OclAny}}$)

lemma $\text{cp-OclAsType}_{\text{OclAny-Person-OclAny}}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P (X::\text{Person})::\text{OclAny}) .\text{oclAsType}(\text{OclAny}))$
by(rule cpI1 , $\text{simp-all add: OclAsType}_{\text{OclAny-OclAny}}$)
lemma $\text{cp-OclAsType}_{\text{OclAny-OclAny-Person}}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P (X::\text{OclAny})::\text{Person}) .\text{oclAsType}(\text{OclAny}))$
by(rule cpI1 , $\text{simp-all add: OclAsType}_{\text{OclAny-Person}}$)
lemma $\text{cp-OclAsType}_{\text{Person-Person-OclAny}}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P (X::\text{Person})::\text{OclAny}) .\text{oclAsType}(\text{Person}))$
by(rule cpI1 , $\text{simp-all add: OclAsType}_{\text{Person-OclAny}}$)
lemma $\text{cp-OclAsType}_{\text{Person-OclAny-Person}}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P (X::\text{OclAny})::\text{Person}) .\text{oclAsType}(\text{Person}))$
by(rule cpI1 , $\text{simp-all add: OclAsType}_{\text{Person-Person}}$)

lemmas [simp] =
 $\text{cp-OclAsType}_{\text{OclAny-Person-Person}}$
 $\text{cp-OclAsType}_{\text{OclAny-OclAny-OclAny}}$
 $\text{cp-OclAsType}_{\text{Person-Person-Person}}$
 $\text{cp-OclAsType}_{\text{Person-OclAny-OclAny}}$

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.5.5. OclIsTypeOf

Definition

consts *OclIsTypeOf_{OclAny}* :: ' α \Rightarrow Boolean ((-).*oclIsTypeOf*'(*OclAny*'))
consts *OclIsTypeOf_{Person}* :: ' α \Rightarrow Boolean ((-).*oclIsTypeOf*'(*Person*'))

defs (**overloaded**) *OclIsTypeOf_{OclAny}-OclAny*:
 (*X::OclAny*) .*oclIsTypeOf*(*OclAny*) \equiv
 ($\lambda \tau$. *case X* τ *of*
 $\perp \Rightarrow$ *invalid* τ
 | [\perp] \Rightarrow *true* τ (* *invalid* ?? *)
 | [*mk_{OclAny} oid* \perp] \Rightarrow *true* τ)

| [$\llbracket mk_{OclAny} \text{ oid } _ \rrbracket \rrbracket \Rightarrow \text{false } \tau$)

defs (overloaded) $OclIsTypeOf_{OclAny-Person}$:
 $(X::Person) .oclIsTypeOf(OclAny) \equiv$
 $(\lambda \tau. \text{case } X \text{ } \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \llbracket \perp \rrbracket \Rightarrow \text{true } \tau \quad (* \text{ invalid } ?? *)$
 $\quad | \llbracket _ \rrbracket \Rightarrow \text{false } \tau)$

defs (overloaded) $OclIsTypeOf_{Person-OclAny}$:
 $(X::OclAny) .oclIsTypeOf(Person) \equiv$
 $(\lambda \tau. \text{case } X \text{ } \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \llbracket \perp \rrbracket \Rightarrow \text{true } \tau$
 $\quad | \llbracket mk_{OclAny} \text{ oid } \perp \rrbracket \Rightarrow \text{false } \tau$
 $\quad | \llbracket mk_{OclAny} \text{ oid } _ \rrbracket \Rightarrow \text{true } \tau)$

defs (overloaded) $OclIsTypeOf_{Person-Person}$:
 $(X::Person) .oclIsTypeOf(Person) \equiv$
 $(\lambda \tau. \text{case } X \text{ } \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | _ \Rightarrow \text{true } \tau)$

Context Passing

lemma $cp\text{-}OclIsTypeOf_{OclAny-Person-Person}$: $cp \ P \Rightarrow cp(\lambda X. (P(X::Person)::Person) .oclIsTypeOf(OclAny))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny-Person}$)
lemma $cp\text{-}OclIsTypeOf_{OclAny-OclAny-OclAny}$: $cp \ P \Rightarrow cp(\lambda X. (P(X::OclAny)::OclAny) .oclIsTypeOf(OclAny))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny-OclAny}$)
lemma $cp\text{-}OclIsTypeOf_{Person-Person-Person}$: $cp \ P \Rightarrow cp(\lambda X. (P(X::Person)::Person) .oclIsTypeOf(Person))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{Person-Person}$)
lemma $cp\text{-}OclIsTypeOf_{Person-OclAny-OclAny}$: $cp \ P \Rightarrow cp(\lambda X. (P(X::OclAny)::OclAny) .oclIsTypeOf(Person))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{Person-OclAny}$)

lemma $cp\text{-}OclIsTypeOf_{OclAny-Person-OclAny}$: $cp \ P \Rightarrow cp(\lambda X. (P(X::Person)::OclAny) .oclIsTypeOf(OclAny))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny-OclAny}$)
lemma $cp\text{-}OclIsTypeOf_{OclAny-OclAny-Person}$: $cp \ P \Rightarrow cp(\lambda X. (P(X::OclAny)::Person) .oclIsTypeOf(OclAny))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny-Person}$)
lemma $cp\text{-}OclIsTypeOf_{Person-Person-OclAny}$: $cp \ P \Rightarrow cp(\lambda X. (P(X::Person)::OclAny) .oclIsTypeOf(Person))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{Person-OclAny}$)
lemma $cp\text{-}OclIsTypeOf_{Person-OclAny-Person}$: $cp \ P \Rightarrow cp(\lambda X. (P(X::OclAny)::Person) .oclIsTypeOf(Person))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{Person-Person}$)

lemmas [simp] =
 $cp\text{-}OclIsTypeOf_{OclAny-Person-Person}$
 $cp\text{-}OclIsTypeOf_{OclAny-OclAny-OclAny}$

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
    OclIsTypeOfOclAny-Person foundation22 foundation16)

lemma down-cast-type:
assumes isOclAny:  $\tau \models (X :: \text{OclAny}) . \text{oclIsTypeOf}(\text{OclAny})$ 
and non-null:  $\tau \models (\delta X)$ 
shows  $\tau \models (X . \text{oclAsType}(\text{Person})) \triangleq \text{invalid}$ 
using isOclAny non-null
apply(auto simp : bot-fun-def null-fun-def null-option-def bot-option-def null-def invalid-def
    OclAsTypeOclAny-Person OclAsTypePerson-OclAny foundation22 foundation16
    split: option.split typeOclAny.split typePerson.split)
by(simp add: OclIsTypeOfOclAny-OclAny OclValid-def false-def true-def)

lemma down-cast-type':
assumes isOclAny:  $\tau \models (X :: \text{OclAny}) . \text{oclIsTypeOf}(\text{OclAny})$ 
and non-null:  $\tau \models (\delta X)$ 
shows  $\tau \models \text{not } (v (X . \text{oclAsType}(\text{Person})))$ 
by(rule foundation15[THEN iffD1], simp add: down-cast-type[OF assms])

lemma up-down-cast :
assumes isdef:  $\tau \models (\delta X)$ 
shows  $\tau \models ((X :: \text{Person}) . \text{oclAsType}(\text{OclAny}) . \text{oclAsType}(\text{Person})) \triangleq X$ 
using isdef
by(auto simp : null-fun-def null-option-def bot-option-def null-def invalid-def
    OclAsTypeOclAny-Person OclAsTypePerson-OclAny foundation22 foundation16
    split: option.split typePerson.split)

lemma up-down-cast-Person-OclAny-Person [simp]:
shows  $((X :: \text{Person}) . \text{oclAsType}(\text{OclAny}) . \text{oclAsType}(\text{Person})) = X$ 
apply(rule ext, rename-tac  $\tau$ )
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  $\tau \models v X$ 
shows  $\tau \models (((X :: \text{Person}) . \text{oclAsType}(\text{OclAny}) . \text{oclAsType}(\text{Person})) \doteq X)$ 
apply(simp only: up-down-cast-Person-OclAny-Person StrictRefEqObject-Person)
by(rule StrictRefEqObject-sym, simp add: assms)

lemma up-down-cast-Person-OclAny-Person'': assumes  $\tau \models v (X :: \text{Person})$ 
shows  $\tau \models (X . \text{oclIsTypeOf}(\text{Person}) \text{ implies } (X . \text{oclAsType}(\text{OclAny}) . \text{oclAsType}(\text{Person})) \doteq X)$ 
apply(simp add: OclValid-def)
apply(subst cp-OclImplies)
apply(simp add: StrictRefEqObject-Person StrictRefEqObject-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 \Rightarrow Boolean ((-).oclIsKindOf'(*OclAny*'))
consts *OclIsKindOf*_{Person} :: ' α \Rightarrow Boolean ((-).oclIsKindOf'(*Person*'))

defs (overloaded) *OclIsKindOf*_{OclAny-OclAny}:

(*X*::*OclAny*) .oclIsKindOf(*OclAny*) \equiv
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \ - \Rightarrow \text{true } \tau)$

defs (overloaded) *OclIsKindOf*_{OclAny-Person}:

(*X*::*Person*) .oclIsKindOf(*OclAny*) \equiv
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \ - \Rightarrow \text{true } \tau)$

defs (overloaded) *OclIsKindOf*_{Person-OclAny}:

(*X*::*OclAny*) .oclIsKindOf(*Person*) \equiv
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \ [\perp] \Rightarrow \text{true } \tau$
 $\quad | \ [mk_{OclAny} \text{ oid } \perp] \Rightarrow \text{false } \tau$
 $\quad | \ [mk_{OclAny} \text{ oid } [-]] \Rightarrow \text{true } \tau)$

defs (overloaded) *OclIsKindOf*_{Person-Person}:

(*X*::*Person*) .oclIsKindOf(*Person*) \equiv
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \ - \Rightarrow \text{true } \tau)$

Context Passing

lemma *cp-OclIsKindOf*_{OclAny-Person-Person}: *cp P* \Rightarrow *cp*($\lambda X. (P(X::\text{Person})::\text{Person}).oclIsKindOf(\text{OclAny}))$)
by(rule *cpI1*, simp-all add: *OclIsKindOf*_{OclAny-Person})
lemma *cp-OclIsKindOf*_{OclAny-OclAny-OclAny}: *cp P* \Rightarrow *cp*($\lambda X. (P(X::\text{OclAny})::\text{OclAny}).oclIsKindOf(\text{OclAny}))$)
by(rule *cpI1*, simp-all add: *OclIsKindOf*_{OclAny-OclAny})
lemma *cp-OclIsKindOf*_{Person-Person-Person}: *cp P* \Rightarrow *cp*($\lambda X. (P(X::\text{Person})::\text{Person}).oclIsKindOf(\text{Person}))$)
by(rule *cpI1*, simp-all add: *OclIsKindOf*_{Person-Person})
lemma *cp-OclIsKindOf*_{Person-OclAny-OclAny}: *cp P* \Rightarrow *cp*($\lambda X. (P(X::\text{OclAny})::\text{OclAny}).oclIsKindOf(\text{Person}))$)
by(rule *cpI1*, simp-all add: *OclIsKindOf*_{Person-OclAny})

lemma $cp\text{-}OclIsKindOf_{OclAny}\text{-}Person\text{-}OclAny$: $cp\ P \implies cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(OclAny))$
by(rule $cpI1$, simp-all add: $OclIsKindOf_{OclAny}\text{-}OclAny$)
lemma $cp\text{-}OclIsKindOf_{OclAny}\text{-}OclAny\text{-}Person$: $cp\ P \implies cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(OclAny))$
by(rule $cpI1$, simp-all add: $OclIsKindOf_{OclAny}\text{-}Person$)
lemma $cp\text{-}OclIsKindOf_{Person}\text{-}Person\text{-}OclAny$: $cp\ P \implies cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(Person))$
by(rule $cpI1$, simp-all add: $OclIsKindOf_{Person}\text{-}OclAny$)
lemma $cp\text{-}OclIsKindOf_{Person}\text{-}OclAny\text{-}Person$: $cp\ P \implies cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(Person))$
by(rule $cpI1$, simp-all add: $OclIsKindOf_{Person}\text{-}Person$)

lemmas [simp] =
 $cp\text{-}OclIsKindOf_{OclAny}\text{-}Person\text{-}Person$
 $cp\text{-}OclIsKindOf_{OclAny}\text{-}OclAny\text{-}OclAny$
 $cp\text{-}OclIsKindOf_{Person}\text{-}Person\text{-}Person$
 $cp\text{-}OclIsKindOf_{Person}\text{-}OclAny\text{-}OclAny$

 $cp\text{-}OclIsKindOf_{OclAny}\text{-}Person\text{-}OclAny$
 $cp\text{-}OclIsKindOf_{OclAny}\text{-}OclAny\text{-}Person$
 $cp\text{-}OclIsKindOf_{Person}\text{-}Person\text{-}OclAny$
 $cp\text{-}OclIsKindOf_{Person}\text{-}OclAny\text{-}Person$

Execution with Invalid or Null as Argument

lemma $OclIsKindOf_{OclAny}\text{-}OclAny\text{-}strict1$ [simp] : $(invalid::OclAny).oclIsKindOf(OclAny) = invalid$
by(rule ext, simp add: invalid-def bot-option-def
 $OclIsKindOf_{OclAny}\text{-}OclAny$)

lemma $OclIsKindOf_{OclAny}\text{-}OclAny\text{-}strict2$ [simp] : $(null::OclAny).oclIsKindOf(OclAny) = true$
by(rule ext, simp add: null-fun-def null-option-def
 $OclIsKindOf_{OclAny}\text{-}OclAny$)

lemma $OclIsKindOf_{OclAny}\text{-}Person\text{-}strict1$ [simp] : $(invalid::Person).oclIsKindOf(OclAny) = invalid$
by(rule ext, simp add: bot-option-def invalid-def
 $OclIsKindOf_{OclAny}\text{-}Person$)

lemma $OclIsKindOf_{OclAny}\text{-}Person\text{-}strict2$ [simp] : $(null::Person).oclIsKindOf(OclAny) = true$
by(rule ext, simp add: null-fun-def null-option-def bot-option-def
 $OclIsKindOf_{OclAny}\text{-}Person$)

lemma $OclIsKindOf_{Person}\text{-}OclAny\text{-}strict1$ [simp] : $(invalid::OclAny).oclIsKindOf(Person) = invalid$
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
 $OclIsKindOf_{Person}\text{-}OclAny$)

lemma $OclIsKindOf_{Person}\text{-}OclAny\text{-}strict2$ [simp] : $(null::OclAny).oclIsKindOf(Person) = true$
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
 $OclIsKindOf_{Person}\text{-}OclAny$)

lemma $OclIsKindOf_{Person}\text{-}Person\text{-}strict1$ [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 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 sufficient “characterization.”

definition $Person \equiv OclAsType_{Person}\mathcal{A}$
definition $OclAny \equiv OclAsType_{OclAny}\mathcal{A}$
lemmas [simp] = Person-def OclAny-def

lemma OclAllInstances-generic_{OclAny-exec}: OclAllInstances-generic pre-post OclAny =
($\lambda \tau. Abs_Set_{base} \llbracket Some \text{ ‘ } OclAny \text{ ‘ } ran (heap (pre-post \tau)) \rrbracket$)

proof –
let ?S1 = $\lambda \tau. OclAny \text{ ‘ } ran (heap (pre-post \tau))$
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}- \mathcal{A} -some)

show ?thesis **by**(insert equalityI[OF B C], simp)
qed

lemma OclAllInstances-at-post_{OclAny-exec}: OclAny .allInstances() =

$(\lambda \tau. \text{Abs-Set}_{base} \llbracket \text{Some } 'OclAny' \text{ ran } (\text{heap } (\text{snd } \tau)) \rrbracket)$
unfolding *OclAllInstances-at-post-def*
by(rule *OclAllInstances-generic_{OclAny-exec}*)

lemma *OclAllInstances-at-pre_{OclAny-exec}: OclAny .allInstances@pre()* =
 $(\lambda \tau. \text{Abs-Set}_{base} \llbracket \text{Some } 'OclAny' \text{ ran } (\text{heap } (\text{fst } \tau)) \rrbracket)$
unfolding *OclAllInstances-at-pre-def*
by(rule *OclAllInstances-generic_{OclAny-exec}*)

OclIsTypeOf

lemma *OclAny-allInstances-generic-oclIsTypeOf_{OclAny1}*:
assumes [*simp*]: $\bigwedge x. \text{pre-post } (x, x) = x$
shows $\exists \tau. (\tau \models ((\text{OclAllInstances-generic pre-post OclAny}) \rightarrow \text{forAll}(X|X. \text{oclIsTypeOf}(\text{OclAny}))))$
apply(rule-tac $x = \tau_0$ in *ex1*, *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_{OclAny1}*:
 $\exists \tau. (\tau \models (\text{OclAny .allInstances}() \rightarrow \text{forAll}(X|X. \text{oclIsTypeOf}(\text{OclAny}))))$
unfolding *OclAllInstances-at-post-def*
by(rule *OclAny-allInstances-generic-oclIsTypeOf_{OclAny1}*, *simp*)

lemma *OclAny-allInstances-at-pre-oclIsTypeOf_{OclAny1}*:
 $\exists \tau. (\tau \models (\text{OclAny .allInstances@pre}() \rightarrow \text{forAll}(X|X. \text{oclIsTypeOf}(\text{OclAny}))))$
unfolding *OclAllInstances-at-pre-def*
by(rule *OclAny-allInstances-generic-oclIsTypeOf_{OclAny1}*, *simp*)

lemma *OclAny-allInstances-generic-oclIsTypeOf_{OclAny2}*:
assumes [*simp*]: $\bigwedge x. \text{pre-post } (x, x) = x$
shows $\exists \tau. (\tau \models \text{not } ((\text{OclAllInstances-generic pre-post OclAny}) \rightarrow \text{forAll}(X|X. \text{oclIsTypeOf}(\text{OclAny}))))$
proof – **fix** *oid* **let** $?t0 = (\text{heap} = \text{empty}(\text{oid} \mapsto \text{in}_{OclAny} (\text{mk}_{OclAny} \text{oid } \llbracket a \rrbracket)),$
 $\text{assocs} = \text{empty})$ **show** *?thesis*
apply(rule-tac $x = (?t0, ?t0)$ in *ex1*, *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}-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_{OclAny2}*:
 $\exists \tau. (\tau \models \text{not } (\text{OclAny .allInstances}() \rightarrow \text{forAll}(X|X. \text{oclIsTypeOf}(\text{OclAny}))))$
unfolding *OclAllInstances-at-post-def*
by(rule *OclAny-allInstances-generic-oclIsTypeOf_{OclAny2}*, *simp*)

lemma *OclAny-allInstances-at-pre-oclIsTypeOf_{OclAny2}*:
 $\exists \tau. (\tau \models \text{not } (\text{OclAny} .\text{allInstances}@pre() \rightarrow \text{forAll}(X|X .\text{oclIsTypeOf}(\text{OclAny}))))$
unfolding *OclAllInstances-at-pre-def*
by(rule *OclAny-allInstances-generic-oclIsTypeOf_{OclAny2}*, *simp*)

lemma *Person-allInstances-generic-oclIsTypeOf_{Person}*:
 $\tau \models ((\text{OclAllInstances-generic pre-post Person}) \rightarrow \text{forAll}(X|X .\text{oclIsTypeOf}(\text{Person})))$
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}*:
 $\tau \models (\text{Person} .\text{allInstances}() \rightarrow \text{forAll}(X|X .\text{oclIsTypeOf}(\text{Person})))$
unfolding *OclAllInstances-at-post-def*
by(rule *Person-allInstances-generic-oclIsTypeOf_{Person}*)

lemma *Person-allInstances-at-pre-oclIsTypeOf_{Person}*:
 $\tau \models (\text{Person} .\text{allInstances}@pre() \rightarrow \text{forAll}(X|X .\text{oclIsTypeOf}(\text{Person})))$
unfolding *OclAllInstances-at-pre-def*
by(rule *Person-allInstances-generic-oclIsTypeOf_{Person}*)

OclIsKindOf

lemma *OclAny-allInstances-generic-oclIsKindOf_{OclAny}*:
 $\tau \models ((\text{OclAllInstances-generic pre-post OclAny}) \rightarrow \text{forAll}(X|X .\text{oclIsKindOf}(\text{OclAny})))$
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}*:
 $\tau \models (\text{OclAny} .\text{allInstances}() \rightarrow \text{forAll}(X|X .\text{oclIsKindOf}(\text{OclAny})))$
unfolding *OclAllInstances-at-post-def*
by(rule *OclAny-allInstances-generic-oclIsKindOf_{OclAny}*)

lemma *OclAny-allInstances-at-pre-oclIsKindOf_{OclAny}*:
 $\tau \models (\text{OclAny} .\text{allInstances}@pre() \rightarrow \text{forAll}(X|X .\text{oclIsKindOf}(\text{OclAny})))$
unfolding *OclAllInstances-at-pre-def*
by(rule *OclAny-allInstances-generic-oclIsKindOf_{OclAny}*)

lemma *Person-allInstances-generic-oclIsKindOf_{OclAny}*:
 $\tau \models ((\text{OclAllInstances-generic pre-post Person}) \rightarrow \text{forAll}(X|X .\text{oclIsKindOf}(\text{OclAny})))$

```

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-Setbase-inverse, simp add: bot-option-def)
by(simp add: OclIsKindOfOclAny-Person)

lemma Person-allInstances-at-post-oclIsKindOfOclAny:
 $\tau \models (Person.allInstances() \rightarrow \text{forAll}(X|X.oclIsKindOf(OclAny)))$ 
unfolding OclAllInstances-at-post-def
by(rule Person-allInstances-generic-oclIsKindOfOclAny)

lemma Person-allInstances-at-pre-oclIsKindOfOclAny:
 $\tau \models (Person.allInstances@pre() \rightarrow \text{forAll}(X|X.oclIsKindOf(OclAny)))$ 
unfolding OclAllInstances-at-pre-def
by(rule Person-allInstances-generic-oclIsKindOfOclAny)

lemma Person-allInstances-generic-oclIsKindOfPerson:
 $\tau \models ((OclAllInstances-generic\ pre\ post\ Person) \rightarrow \text{forAll}(X|X.oclIsKindOf(Person)))$ 
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-Setbase-inverse, simp add: bot-option-def)
by(simp add: OclIsKindOfPerson-Person)

lemma Person-allInstances-at-post-oclIsKindOfPerson:
 $\tau \models (Person.allInstances() \rightarrow \text{forAll}(X|X.oclIsKindOf(Person)))$ 
unfolding OclAllInstances-at-post-def
by(rule Person-allInstances-generic-oclIsKindOfPerson)

lemma Person-allInstances-at-pre-oclIsKindOfPerson:
 $\tau \models (Person.allInstances@pre() \rightarrow \text{forAll}(X|X.oclIsKindOf(Person)))$ 
unfolding OclAllInstances-at-pre-def
by(rule Person-allInstances-generic-oclIsKindOfPerson)

```

B.5.8. The Accessors (any, boss, salary)

Should be generated entirely from a class-diagram.

Definition

```

definition eval-extract :: ('A, ('a::object) option option) val
    ⇒ (oid ⇒ ('A, 'c::null) val)
    ⇒ ('A, 'c::null) val
where eval-extract X f = (λ τ. case X τ of
    ⊥ ⇒ invalid τ (* exception propagation *))

```


$$\begin{aligned} &| \lfloor \perp \rfloor \Rightarrow \text{invalid } \tau \text{ (* dereferencing null pointer *)} \\ &| \lfloor \lfloor \text{obj} \rfloor \rfloor \Rightarrow f \text{ (oid-of obj) } \tau \end{aligned}$$

definition $\text{deref-oid}_{\text{Person}} :: (\mathfrak{A} \text{ state} \times \mathfrak{A} \text{ state} \Rightarrow \mathfrak{A} \text{ state})$
 $\Rightarrow (\text{type}_{\text{Person}} \Rightarrow (\mathfrak{A}, 'c::\text{null})\text{val})$
 $\Rightarrow \text{oid}$
 $\Rightarrow (\mathfrak{A}, 'c::\text{null})\text{val}$

where $\text{deref-oid}_{\text{Person}} \text{fst-snd } f \text{oid} = (\lambda \tau. \text{case } (\text{heap } (\text{fst-snd } \tau)) \text{ oid of}$
 $\lfloor \text{in}_{\text{Person}} \text{obj} \rfloor \Rightarrow f \text{obj } \tau$
 $| - \Rightarrow \text{invalid } \tau)$

definition $\text{deref-oid}_{\text{OclAny}} :: (\mathfrak{A} \text{ state} \times \mathfrak{A} \text{ state} \Rightarrow \mathfrak{A} \text{ state})$
 $\Rightarrow (\text{type}_{\text{OclAny}} \Rightarrow (\mathfrak{A}, 'c::\text{null})\text{val})$
 $\Rightarrow \text{oid}$
 $\Rightarrow (\mathfrak{A}, 'c::\text{null})\text{val}$

where $\text{deref-oid}_{\text{OclAny}} \text{fst-snd } f \text{oid} = (\lambda \tau. \text{case } (\text{heap } (\text{fst-snd } \tau)) \text{ oid of}$
 $\lfloor \text{in}_{\text{OclAny}} \text{obj} \rfloor \Rightarrow f \text{obj } \tau$
 $| - \Rightarrow \text{invalid } \tau)$

pointer undefined in state or not referencing a type conform object representation

definition $\text{select}_{\text{OclAny}} \mathcal{A} \mathcal{N} \mathcal{Y} f = (\lambda X. \text{case } X \text{ of}$
 $(\text{mk}_{\text{OclAny}} - \perp) \Rightarrow \text{null}$
 $| (\text{mk}_{\text{OclAny}} - \lfloor \text{any} \rfloor) \Rightarrow f (\lambda x -. \lfloor \lfloor x \rfloor \rfloor) \text{any})$

definition $\text{select}_{\text{Person}} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} f = (\lambda X. \text{case } X \text{ of}$
 $(\text{mk}_{\text{Person}} - \perp) \Rightarrow \text{null} \text{ (* object contains null pointer *)}$
 $| (\text{mk}_{\text{Person}} - \lfloor \text{boss} \rfloor) \Rightarrow f (\lambda x -. \lfloor \lfloor x \rfloor \rfloor) \text{boss})$

definition $\text{select}_{\text{Person}} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y} f = (\lambda X. \text{case } X \text{ of}$
 $(\text{mk}_{\text{Person}} - \perp -) \Rightarrow \text{null}$
 $| (\text{mk}_{\text{Person}} - \lfloor \text{salary} \rfloor -) \Rightarrow f (\lambda x -. \lfloor \lfloor x \rfloor \rfloor) \text{salary})$

definition $\text{in-pre-state} = \text{fst}$

definition $\text{in-post-state} = \text{snd}$

definition $\text{reconst-basetype} = (\lambda \text{convert } x. \text{convert } x)$

definition $\text{dot}_{\text{OclAny}} \mathcal{A} \mathcal{N} \mathcal{Y} :: \text{OclAny} \Rightarrow - \ ((I(-).\text{any}) \ 50)$
where $(X).\text{any} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{OclAny}} \text{in-post-state}$
 $(\text{select}_{\text{OclAny}} \mathcal{A} \mathcal{N} \mathcal{Y}$

reconst-basetype))

definition *dot_{Person}BOSS :: Person ⇒ Person* ((*I*(-).boss) 50)
where (*X*).boss = eval-extract *X*
 (deref-oid_{Person} in-post-state
 (select_{Person}BOSS
 (deref-oid_{Person} in-post-state)))

definition *dot_{Person}SALAR :: Person ⇒ Integer* ((*I*(-).salary) 50)
where (*X*).salary = eval-extract *X*
 (deref-oid_{Person} in-post-state
 (select_{Person}SALAR
 reconst-basetype))

definition *dot_{OclAny}ANY-at-pre :: OclAny ⇒ -* ((*I*(-).any@pre) 50)
where (*X*).any@pre = eval-extract *X*
 (deref-oid_{OclAny} in-pre-state
 (select_{OclAny}ANY
 reconst-basetype))

definition *dot_{Person}BOSS-at-pre :: Person ⇒ Person* ((*I*(-).boss@pre) 50)
where (*X*).boss@pre = eval-extract *X*
 (deref-oid_{Person} in-pre-state
 (select_{Person}BOSS
 (deref-oid_{Person} in-pre-state)))

definition *dot_{Person}SALAR-at-pre :: Person ⇒ Integer* ((*I*(-).salary@pre) 50)
where (*X*).salary@pre = eval-extract *X*
 (deref-oid_{Person} in-pre-state
 (select_{Person}SALAR
 reconst-basetype))

lemmas *dot-accessor =*
dot_{OclAny}ANY-def
dot_{Person}BOSS-def
dot_{Person}SALAR-def
dot_{OclAny}ANY-at-pre-def
dot_{Person}BOSS-at-pre-def
dot_{Person}SALAR-at-pre-def

Context Passing

lemmas [*simp*] = eval-extract-def

lemma *cp-dot_{OclAny}ANY*: ((*X*).any) τ = ((λ -. *X* τ).any) τ **by** (*simp add: dot-accessor*)
lemma *cp-dot_{Person}BOSS*: ((*X*).boss) τ = ((λ -. *X* τ).boss) τ **by** (*simp add: dot-accessor*)
lemma *cp-dot_{Person}SALAR*: ((*X*).salary) τ = ((λ -. *X* τ).salary) τ **by** (*simp add: dot-accessor*)

lemma $cp\text{-}dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y}\text{-}at\text{-}pre$: $((X).any@pre) \tau = ((\lambda -. X \tau).any@pre) \tau$ **by** (simp add: dot-accessor)
lemma $cp\text{-}dot_{Person} \mathcal{BO} \mathcal{S} \mathcal{S}\text{-}at\text{-}pre$: $((X).boss@pre) \tau = ((\lambda -. X \tau).boss@pre) \tau$ **by** (simp add: dot-accessor)
lemma $cp\text{-}dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}\text{-}at\text{-}pre$: $((X).salary@pre) \tau = ((\lambda -. X \tau).salary@pre) \tau$ **by** (simp add: dot-accessor)

lemmas $cp\text{-}dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y}\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y}$ [THEN allI [THEN allI],
of $\lambda X -. X \lambda - \tau. \tau$, THEN cpII]
lemmas $cp\text{-}dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y}\text{-}at\text{-}pre\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y}\text{-}at\text{-}pre$ [THEN allI [THEN allI],
of $\lambda X -. X \lambda - \tau. \tau$, THEN cpII]

lemmas $cp\text{-}dot_{Person} \mathcal{BO} \mathcal{S} \mathcal{S}\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{Person} \mathcal{BO} \mathcal{S} \mathcal{S}$ [THEN allI [THEN allI],
of $\lambda X -. X \lambda - \tau. \tau$, THEN cpII]
lemmas $cp\text{-}dot_{Person} \mathcal{BO} \mathcal{S} \mathcal{S}\text{-}at\text{-}pre\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{Person} \mathcal{BO} \mathcal{S} \mathcal{S}\text{-}at\text{-}pre$ [THEN allI [THEN allI],
of $\lambda X -. X \lambda - \tau. \tau$, THEN cpII]

lemmas $cp\text{-}dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}$ [THEN allI [THEN allI],
of $\lambda X -. X \lambda - \tau. \tau$, THEN cpII]
lemmas $cp\text{-}dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}\text{-}at\text{-}pre\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}\text{-}at\text{-}pre$ [THEN allI [THEN allI],
of $\lambda X -. X \lambda - \tau. \tau$, THEN cpII]

Execution with Invalid or Null as Argument

lemma $dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y}\text{-}nullstrict$ [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 $dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y}\text{-}at\text{-}pre\text{-}nullstrict$ [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 $dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y}\text{-}strict$ [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 $dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y}\text{-}at\text{-}pre\text{-}strict$ [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 $dot_{Person} \mathcal{BO} \mathcal{S} \mathcal{S}\text{-}nullstrict$ [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 $dot_{Person} \mathcal{BO} \mathcal{S} \mathcal{S}\text{-}at\text{-}pre\text{-}nullstrict$ [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 $dot_{Person} \mathcal{BO} \mathcal{S} \mathcal{S}\text{-}strict$ [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 $dot_{Person} \mathcal{BO} \mathcal{S} \mathcal{S}\text{-}at\text{-}pre\text{-}strict$ [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 $dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}\text{-}nullstrict$ [simp]: $(null).salary = invalid$

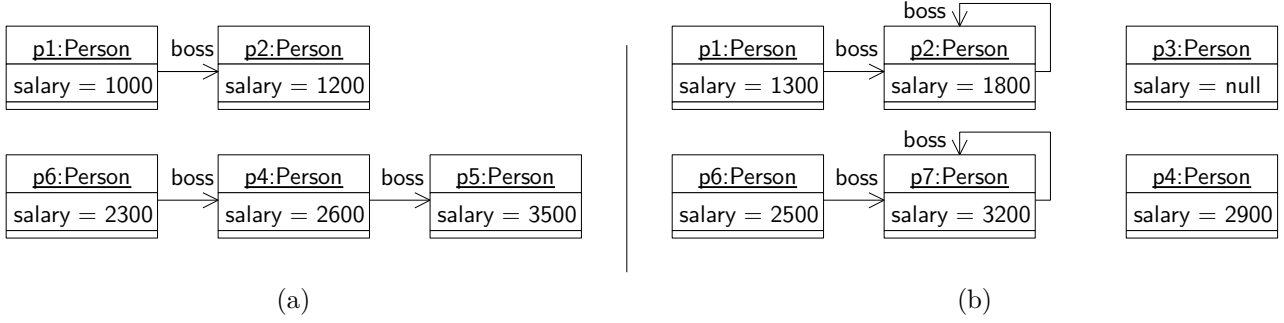


Figure B.4.: (a) pre-state σ_1 and (b) post-state σ'_1 .

```

by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def null-def invalid-def)
lemma dotPerson  $\mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}$ -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 dotPerson  $\mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}$ -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 dotPerson  $\mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}$ -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 = ( $\lambda$  . . [[1000]] )
definition OclInt1200 (1200) where OclInt1200 = ( $\lambda$  . . [[1200]] )
definition OclInt1300 (1300) where OclInt1300 = ( $\lambda$  . . [[1300]] )
definition OclInt1800 (1800) where OclInt1800 = ( $\lambda$  . . [[1800]] )
definition OclInt2600 (2600) where OclInt2600 = ( $\lambda$  . . [[2600]] )
definition OclInt2900 (2900) where OclInt2900 = ( $\lambda$  . . [[2900]] )
definition OclInt3200 (3200) where OclInt3200 = ( $\lambda$  . . [[3200]] )
definition OclInt3500 (3500) where OclInt3500 = ( $\lambda$  . . [[3500]] )

```

```

definition oid0  $\equiv$  0
definition oid1  $\equiv$  1
definition oid2  $\equiv$  2
definition oid3  $\equiv$  3
definition oid4  $\equiv$  4
definition oid5  $\equiv$  5
definition oid6  $\equiv$  6
definition oid7  $\equiv$  7
definition oid8  $\equiv$  8

```

```

definition person1  $\equiv$  mkPerson oid0 [1300] [oid1]
definition person2  $\equiv$  mkPerson oid1 [1800] [oid1]
definition person3  $\equiv$  mkPerson oid2 None None

```

definition $person4 \equiv mk_{Person} \text{ oid3 } [2900] \text{ None}$
definition $person5 \equiv mk_{Person} \text{ oid4 } [3500] \text{ None}$
definition $person6 \equiv mk_{Person} \text{ oid5 } [2500] [oid6]$
definition $person7 \equiv mk_{OclAny} \text{ oid6 } [([3200], [oid6])]$
definition $person8 \equiv mk_{OclAny} \text{ oid7 } \text{ None}$
definition $person9 \equiv mk_{Person} \text{ oid8 } [0] \text{ None}$

definition

$\sigma_1 \equiv () \text{ heap} = \text{empty}(\text{oid0} \mapsto in_{Person} (mk_{Person} \text{ oid0 } [1000] [oid1]))$
 $(oid1 \mapsto in_{Person} (mk_{Person} \text{ oid1 } [1200] \text{ None}))$
 $(*oid2*)$
 $(oid3 \mapsto in_{Person} (mk_{Person} \text{ oid3 } [2600] [oid4]))$
 $(oid4 \mapsto in_{Person} person5)$
 $(oid5 \mapsto in_{Person} (mk_{Person} \text{ oid5 } [2300] [oid3]))$
 $(*oid6*)$
 $(*oid7*)$
 $(oid8 \mapsto in_{Person} person9),$
 $assocs = \text{empty } ()$

definition

$\sigma_1' \equiv () \text{ heap} = \text{empty}(\text{oid0} \mapsto in_{Person} person1)$
 $(oid1 \mapsto in_{Person} person2)$
 $(oid2 \mapsto in_{Person} person3)$
 $(oid3 \mapsto in_{Person} person4)$
 $(*oid4*)$
 $(oid5 \mapsto in_{Person} person6)$
 $(oid6 \mapsto in_{OclAny} person7)$
 $(oid7 \mapsto in_{OclAny} person8)$
 $(oid8 \mapsto in_{Person} person9),$
 $assocs = \text{empty } ()$

definition $\sigma_0 \equiv () \text{ heap} = \text{empty}, \text{ assocs} = \text{empty } ()$

lemma $basic\text{-}\tau\text{-}wff: WFF(\sigma_1, \sigma_1')$

by($auto \text{ simp: } WFF\text{-}def \sigma_1\text{-}def \sigma_1'\text{-}def$
 $oid0\text{-}def \text{ oid1}\text{-}def \text{ oid2}\text{-}def \text{ oid3}\text{-}def \text{ oid4}\text{-}def \text{ oid5}\text{-}def \text{ oid6}\text{-}def \text{ oid7}\text{-}def \text{ oid8}\text{-}def$
 $oid\text{-}of\mathcal{A}\text{-}def \text{ oid}\text{-}of\text{-}type_{Person}\text{-}def \text{ oid}\text{-}of\text{-}type_{OclAny}\text{-}def$
 $person1\text{-}def \text{ person2}\text{-}def \text{ person3}\text{-}def \text{ person4}\text{-}def$
 $person5\text{-}def \text{ person6}\text{-}def \text{ person7}\text{-}def \text{ person8}\text{-}def \text{ person9}\text{-}def$)

lemma $[simp, code\text{-}unfold]: dom (\text{heap } \sigma_1) = \{\text{oid0}, \text{oid1}, (*, \text{oid2}*)\text{oid3}, \text{oid4}, \text{oid5}(*, \text{oid6}, \text{oid7}*), \text{oid8}\}$

by($auto \text{ simp: } \sigma_1\text{-}def$)

lemma $[simp, code\text{-}unfold]: dom (\text{heap } \sigma_1') = \{\text{oid0}, \text{oid1}, \text{oid2}, \text{oid3}, (*, \text{oid4}*)\text{oid5}, \text{oid6}, \text{oid7}, \text{oid8}\}$

by($auto \text{ simp: } \sigma_1'\text{-}def$)

definition $X_{Person} I :: Person \equiv \lambda . . [\text{ personI }]]$

```

definition  $X_{Person2} :: Person \equiv \lambda - . \llbracket person2 \rrbracket$ 
definition  $X_{Person3} :: Person \equiv \lambda - . \llbracket person3 \rrbracket$ 
definition  $X_{Person4} :: Person \equiv \lambda - . \llbracket person4 \rrbracket$ 
definition  $X_{Person5} :: Person \equiv \lambda - . \llbracket person5 \rrbracket$ 
definition  $X_{Person6} :: Person \equiv \lambda - . \llbracket person6 \rrbracket$ 
definition  $X_{Person7} :: OclAny \equiv \lambda - . \llbracket person7 \rrbracket$ 
definition  $X_{Person8} :: OclAny \equiv \lambda - . \llbracket person8 \rrbracket$ 
definition  $X_{Person9} :: Person \equiv \lambda - . \llbracket person9 \rrbracket$ 

```

```

lemma [code-unfold]:  $((x::Person) \doteq y) = StrictRefEq_{Object} \ x \ y \text{ by } (simp \ only: \ StrictRefEq_{Object-}Person)$ 
lemma [code-unfold]:  $((x::OclAny) \doteq y) = StrictRefEq_{Object} \ x \ y \text{ by } (simp \ only: \ StrictRefEq_{Object-OclAny})$ 

```

```

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  $\wedge s_{pre} . (s_{pre}, \sigma_1') \models (X_{Person1} . salary <> 1000)$ 
Assert  $\wedge s_{pre} . (s_{pre}, \sigma_1') \models (X_{Person1} . salary \doteq 1300)$ 
Assert  $\wedge s_{post} . (\sigma_1, s_{post}) \models (X_{Person1} . salary@pre \doteq 1000)$ 
Assert  $\wedge s_{post} . (\sigma_1, s_{post}) \models (X_{Person1} . salary@pre <> 1300)$ 
Assert  $\wedge s_{pre} . (s_{pre}, \sigma_1') \models (X_{Person1} . boss <> X_{Person1})$ 
Assert  $\wedge s_{pre} . (s_{pre}, \sigma_1') \models (X_{Person1} . boss . salary \doteq 1800)$ 
Assert  $\wedge s_{pre} . (s_{pre}, \sigma_1') \models (X_{Person1} . boss . boss <> X_{Person1})$ 
Assert  $\wedge s_{pre} . (s_{pre}, \sigma_1') \models (X_{Person1} . boss . boss \doteq X_{Person2})$ 
Assert  $(\sigma_1, \sigma_1') \models (X_{Person1} . boss@pre . salary \doteq 1800)$ 
Assert  $\wedge s_{post} . (\sigma_1, s_{post}) \models (X_{Person1} . boss@pre . salary@pre \doteq 1200)$ 
Assert  $\wedge s_{post} . (\sigma_1, s_{post}) \models (X_{Person1} . boss@pre . salary@pre <> 1800)$ 
Assert  $\wedge s_{post} . (\sigma_1, s_{post}) \models (X_{Person1} . boss@pre \doteq X_{Person2})$ 
Assert  $(\sigma_1, \sigma_1') \models (X_{Person1} . boss@pre . boss \doteq X_{Person2})$ 
Assert  $\wedge s_{post} . (\sigma_1, s_{post}) \models (X_{Person1} . boss@pre . boss@pre \doteq null)$ 
Assert  $\wedge s_{post} . (\sigma_1, s_{post}) \models not(v(X_{Person1} . boss@pre . boss@pre . boss@pre))$ 

```

```

lemma  $(\sigma_1, \sigma_1') \models (X_{Person1} . oclIsMaintained())$ 
by (simp add: OclValid-def OclIsMaintained-def
   $\sigma_1\text{-def } \sigma_1'\text{-def}$ )

```

X_{Person1}-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 $\wedge s_{pre} s_{post} \cdot (s_{pre}, s_{post}) \models ((X_{Person1} .oclAsType(OclAny) .oclAsType(Person)) \doteq X_{Person1})$
by(rule up-down-cast-Person-OclAny-Person', simp add: *X_{Person1}*-def)
Assert $\wedge s_{pre} s_{post} \cdot (s_{pre}, s_{post}) \models (X_{Person1} .oclIsTypeOf(Person))$
Assert $\wedge s_{pre} s_{post} \cdot (s_{pre}, s_{post}) \models \text{not}(X_{Person1} .oclIsTypeOf(OclAny))$
Assert $\wedge s_{pre} s_{post} \cdot (s_{pre}, s_{post}) \models (X_{Person1} .oclIsKindOf(Person))$
Assert $\wedge s_{pre} s_{post} \cdot (s_{pre}, s_{post}) \models (X_{Person1} .oclIsKindOf(OclAny))$
Assert $\wedge s_{pre} s_{post} \cdot (s_{pre}, s_{post}) \models \text{not}(X_{Person1} .oclAsType(OclAny) .oclIsTypeOf(OclAny))$

Assert $\wedge s_{pre} \cdot (s_{pre}, \sigma_1') \models (X_{Person2} .salary \doteq 1800)$
Assert $\wedge s_{post} \cdot (\sigma_1, s_{post}) \models (X_{Person2} .salary@pre \doteq 1200)$
Assert $\wedge s_{pre} \cdot (s_{pre}, \sigma_1') \models (X_{Person2} .boss \doteq X_{Person2})$
Assert $(\sigma_1, \sigma_1') \models (X_{Person2} .boss .salary@pre \doteq 1200)$
Assert $(\sigma_1, \sigma_1') \models (X_{Person2} .boss .boss@pre \doteq \text{null})$
Assert $\wedge s_{post} \cdot (\sigma_1, s_{post}) \models (X_{Person2} .boss@pre \doteq \text{null})$
Assert $\wedge s_{post} \cdot (\sigma_1, s_{post}) \models (X_{Person2} .boss@pre <> X_{Person2})$
Assert $(\sigma_1, \sigma_1') \models (X_{Person2} .boss@pre <> (X_{Person2} .boss))$
Assert $\wedge s_{post} \cdot (\sigma_1, s_{post}) \models \text{not}(\text{v}(X_{Person2} .boss@pre .boss))$
Assert $\wedge s_{post} \cdot (\sigma_1, s_{post}) \models \text{not}(\text{v}(X_{Person2} .boss@pre .salary@pre))$
lemma $(\sigma_1, \sigma_1') \models (X_{Person2} .oclIsMaintained())$
by(simp add: OclValid-def OclIsMaintained-def
 σ_1 -def σ_1' -def
X_{Person2}-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 $\wedge s_{pre} \cdot (s_{pre}, \sigma_1') \models (X_{Person3} .salary \doteq \text{null})$
Assert $\wedge s_{post} \cdot (\sigma_1, s_{post}) \models \text{not}(\text{v}(X_{Person3} .salary@pre))$
Assert $\wedge s_{pre} \cdot (s_{pre}, \sigma_1') \models (X_{Person3} .boss \doteq \text{null})$
Assert $\wedge s_{pre} \cdot (s_{pre}, \sigma_1') \models \text{not}(\text{v}(X_{Person3} .boss .salary))$
Assert $\wedge s_{post} \cdot (\sigma_1, s_{post}) \models \text{not}(\text{v}(X_{Person3} .boss@pre))$
lemma $(\sigma_1, \sigma_1') \models (X_{Person3} .oclIsNew())$
by(simp add: OclValid-def OclIsNew-def
 σ_1 -def σ_1' -def
X_{Person3}-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 $\wedge s_{post} \cdot (\sigma_1, s_{post}) \models (X_{Person4} .boss@pre \doteq X_{Person5})$
Assert $(\sigma_1, \sigma_1') \models \text{not}(\text{v}(X_{Person4} .boss@pre .salary))$
Assert $\wedge s_{post} \cdot (\sigma_1, s_{post}) \models (X_{Person4} .boss@pre .salary@pre \doteq 3500)$
lemma $(\sigma_1, \sigma_1') \models (X_{Person4} .oclIsMaintained())$

by(simp add: OclValid-def OclIsMaintained-def
 σ_1 -def σ_1' -def
 $X_{Person4}$ -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 $\wedge s_{pre} \cdot (s_{pre}, \sigma_1') \models \text{not}(\text{v}(X_{Person5} . \text{salary}))$
Assert $\wedge s_{post} \cdot (\sigma_1, s_{post}) \models (X_{Person5} . \text{salary}@pre \doteq 3500)$
Assert $\wedge s_{pre} \cdot (s_{pre}, \sigma_1') \models \text{not}(\text{v}(X_{Person5} . \text{boss}))$
lemma $(\sigma_1, \sigma_1') \models (X_{Person5} . \text{ocIsDeleted}())$
by(simp add: OclNot-def OclValid-def OclIsDeleted-def
 σ_1 -def σ_1' -def
 $X_{Person5}$ -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 $\wedge s_{pre} \cdot (s_{pre}, \sigma_1') \models \text{not}(\text{v}(X_{Person6} . \text{boss} . \text{salary}@pre))$
Assert $\wedge s_{post} \cdot (\sigma_1, s_{post}) \models (X_{Person6} . \text{boss}@pre \doteq X_{Person4})$
Assert $(\sigma_1, \sigma_1') \models (X_{Person6} . \text{boss}@pre . \text{salary} \doteq 2900)$
Assert $\wedge s_{post} \cdot (\sigma_1, s_{post}) \models (X_{Person6} . \text{boss}@pre . \text{salary}@pre \doteq 2600)$
Assert $\wedge s_{post} \cdot (\sigma_1, s_{post}) \models (X_{Person6} . \text{boss}@pre . \text{boss}@pre \doteq X_{Person5})$
lemma $(\sigma_1, \sigma_1') \models (X_{Person6} . \text{ocIsMaintained}())$
by(simp add: OclValid-def OclIsMaintained-def
 σ_1 -def σ_1' -def
 $X_{Person6}$ -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 $\wedge s_{pre} s_{post} \cdot (s_{pre}, s_{post}) \models \text{v}(X_{Person7} . \text{oclAsType}(\text{Person}))$
Assert $\wedge s_{post} \cdot (\sigma_1, s_{post}) \models \text{not}(\text{v}(X_{Person7} . \text{oclAsType}(\text{Person}) . \text{boss}@pre))$
lemma $\wedge s_{pre} s_{post} \cdot (s_{pre}, s_{post}) \models ((X_{Person7} . \text{oclAsType}(\text{Person}) . \text{oclAsType}(\text{OclAny}) . \text{oclAsType}(\text{Person})) \doteq (X_{Person7} . \text{oclAsType}(\text{Person})))$
by(rule up-down-cast-Person-OclAny-Person', simp add: $X_{Person7}$ -def OclValid-def valid-def person7-def)
lemma $(\sigma_1, \sigma_1') \models (X_{Person7} . \text{ocIsNew}())$
by(simp add: OclValid-def OclIsNew-def
 σ_1 -def σ_1' -def
 $X_{Person7}$ -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 $\wedge_{s_{pre} s_{post}} (s_{pre}, s_{post}) \models (X_{Person8} <> X_{Person7})$
Assert $\wedge_{s_{pre} s_{post}} (s_{pre}, s_{post}) \models \text{not}(\vee(X_{Person8}.\text{oclAsType}(\text{Person})))$
Assert $\wedge_{s_{pre} s_{post}} (s_{pre}, s_{post}) \models (X_{Person8}.\text{oclIsTypeOf}(\text{OclAny}))$
Assert $\wedge_{s_{pre} s_{post}} (s_{pre}, s_{post}) \models \text{not}(X_{Person8}.\text{oclIsTypeOf}(\text{Person}))$
Assert $\wedge_{s_{pre} s_{post}} (s_{pre}, s_{post}) \models \text{not}(X_{Person8}.\text{oclIsKindOf}(\text{Person}))$
Assert $\wedge_{s_{pre} s_{post}} (s_{pre}, s_{post}) \models (X_{Person8}.\text{oclIsKindOf}(\text{OclAny}))$

lemma $\sigma\text{-modifiedonly: } (\sigma_1, \sigma_1') \models (\text{Set}\{ X_{Person1}.\text{oclAsType}(\text{OclAny})$
 $, X_{Person2}.\text{oclAsType}(\text{OclAny})$
 $(*, X_{Person3}.\text{oclAsType}(\text{OclAny})*)$
 $, X_{Person4}.\text{oclAsType}(\text{OclAny})$
 $(*, X_{Person5}.\text{oclAsType}(\text{OclAny})*)$
 $, X_{Person6}.\text{oclAsType}(\text{OclAny})$
 $(*, X_{Person7}.\text{oclAsType}(\text{OclAny})*)$
 $(*, X_{Person8}.\text{oclAsType}(\text{OclAny})*)$
 $(*, X_{Person9}.\text{oclAsType}(\text{OclAny})*)\} \rightarrow \text{oclIsModifiedOnly}())$

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
XPerson1-def XPerson2-def XPerson3-def XPerson4-def
XPerson5-def XPerson6-def XPerson7-def XPerson8-def XPerson9-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-typeOclAny-def, clarsimp*)

apply (simp add: $\sigma_1\text{-def } \sigma_1'\text{-def}$
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)

done

lemma $(\sigma_1, \sigma_1') \models ((X_{Person9} @_{pre} (\lambda x. \lfloor \text{OclAsType}_{Person}\text{-}\mathfrak{A} x \rfloor)) \triangleq X_{Person9})$
by (simp add: *OclSelf-at-pre-def* $\sigma_1\text{-def oid-of-option-def oid-of-type}_{Person}\text{-def}$
XPerson9-def person9-def oid8-def OclValid-def StrongEq-def OclAsType}_{Person}\text{-}\mathfrak{A}\text{-def})

lemma $(\sigma_1, \sigma_1') \models ((X_{Person9} @_{post} (\lambda x. \lfloor \text{OclAsType}_{Person}\text{-}\mathfrak{A} x \rfloor)) \triangleq X_{Person9})$
by (simp add: *OclSelf-at-post-def* $\sigma_1'\text{-def oid-of-option-def oid-of-type}_{Person}\text{-def}$
XPerson9-def person9-def oid8-def OclValid-def StrongEq-def OclAsType}_{Person}\text{-}\mathfrak{A}\text{-def})

lemma $(\sigma_1, \sigma_1') \models (((X_{Person9}.\text{oclAsType}(\text{OclAny})) @_{pre} (\lambda x. \lfloor \text{OclAsType}_{OclAny}\text{-}\mathfrak{A} x \rfloor)) \triangleq$
 $((X_{Person9}.\text{oclAsType}(\text{OclAny})) @_{post} (\lambda x. \lfloor \text{OclAsType}_{OclAny}\text{-}\mathfrak{A} x \rfloor)))$

proof –

have *including4* : $\wedge a b c d \tau.$

$\text{Set}\{\lambda \tau. \lfloor \lfloor a \rfloor \rfloor, \lambda \tau. \lfloor \lfloor b \rfloor \rfloor, \lambda \tau. \lfloor \lfloor c \rfloor \rfloor, \lambda \tau. \lfloor \lfloor d \rfloor \rfloor\} \tau = \text{Abs-Set}_{base} \lfloor \lfloor \{\lfloor \lfloor a \rfloor \rfloor, \lfloor \lfloor b \rfloor \rfloor, \lfloor \lfloor c \rfloor \rfloor, \lfloor \lfloor d \rfloor \rfloor\} \rfloor \rfloor$

apply (subst *abs-rep-simp'* [symmetric], simp)

apply (simp add: *OclIncluding-rep-set mtSet-rep-set*)

by (rule *arg-cong* [of - - $\lambda x. (\text{Abs-Set}_{base}(\lfloor \lfloor x \rfloor \rfloor))$], auto)

```

have excludingI:  $\bigwedge S a b c d e \tau.$ 
  ( $\lambda -. Abs\_Set_{base} \llbracket \{ \llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket, \llbracket d \rrbracket \} \rrbracket$ )  $\rightarrow$  excluding( $\lambda \tau. \llbracket e \rrbracket$ )  $\tau =$ 
   $Abs\_Set_{base} \llbracket \{ \llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket, \llbracket d \rrbracket \} - \{ \llbracket e \rrbracket \} \rrbracket$ 
apply(simp add: OclExcluding-def)
apply(simp add: defined-def OclValid-def false-def true-def
  bot-fun-def bot-Setbase-def null-fun-def null-Setbase-def)
apply(rule conjI)
apply(rule impI, subst (asm) Abs-Setbase-inject) apply( simp add: bot-option-def)+
apply(rule conjI)
apply(rule impI, subst (asm) Abs-Setbase-inject) apply( simp add: bot-option-def null-option-def)+
apply(subst Abs-Setbase-inverse, simp add: bot-option-def, simp)
done

show ?thesis
apply(rule framing[where X = Set{ XPerson1.oclAsType(OclAny)
  , XPerson2.oclAsType(OclAny)
  (*, XPerson3.oclAsType(OclAny)*)
  , XPerson4.oclAsType(OclAny)
  (*, XPerson5.oclAsType(OclAny)*)
  , XPerson6.oclAsType(OclAny)
  (*, XPerson7.oclAsType(OclAny)*)
  (*, XPerson8.oclAsType(OclAny)*)
  (*, XPerson9.oclAsType(OclAny)*)}])
apply(cut-tac  $\sigma$ -modifiedonly)
apply(simp only: OclValid-def
  XPerson1-def XPerson2-def XPerson3-def XPerson4-def
  XPerson5-def XPerson6-def XPerson7-def XPerson8-def XPerson9-def
  person1-def person2-def person3-def person4-def
  person5-def person6-def person7-def person8-def person9-def
  OclAsTypeOclAny-Person)
apply(subst cp-OclIsModifiedOnly, subst cp-OclExcluding,
  subst (asm) cp-OclIsModifiedOnly, simp add: including4 excludingI)

apply(simp only: XPerson1-def XPerson2-def XPerson3-def XPerson4-def
  XPerson5-def XPerson6-def XPerson7-def XPerson8-def XPerson9-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: StrictRefEqObject-def oid-of-option-def oid-of-typeOclAny-def OclNot-def OclValid-def
  null-option-def bot-option-def)
done
qed

lemma perm- $\sigma_1'$ :  $\sigma_1' = () \text{ heap} = \text{empty}$ 
  (oid8  $\mapsto$  inPerson person9)
  (oid7  $\mapsto$  inOclAny person8)

```

$(oid6 \mapsto in_{OclAny} person7)$
 $(oid5 \mapsto in_{Person} person6)$
 $(*oid4*)$
 $(oid3 \mapsto in_{Person} person4)$
 $(oid2 \mapsto in_{Person} person3)$
 $(oid1 \mapsto in_{Person} person2)$
 $(oid0 \mapsto in_{Person} person1)$
 $, assocS = assocS \sigma_1' \rangle$

proof –

note $P = fun\text{-}upd\text{-}twist$

show $?thesis$

apply(simp add: σ_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 $\wedge \sigma_1.$

$(\sigma_1, \sigma_1') \models (Person \ .allInstances() \doteq Set\{ X_{Person1}, X_{Person2}, X_{Person3}, X_{Person4}(*, X_{Person5*}), X_{Person6},$
 $X_{Person7} .oclAsType(Person)(* , X_{Person8*}), X_{Person9} \})$

apply(subst perm- σ_1')

apply(simp only: oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def

$X_{Person1}$ -def $X_{Person2}$ -def $X_{Person3}$ -def $X_{Person4}$ -def

$X_{Person5}$ -def $X_{Person6}$ -def $X_{Person7}$ -def $X_{Person8}$ -def $X_{Person9}$ -def

person7-def)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: $OclAsType_{Person}\mathcal{A}$ -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule $OclIncluding$ -cong, simp, simp)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: $OclAsType_{Person}\mathcal{A}$ -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule $OclIncluding$ -cong, simp, simp)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: $OclAsType_{Person}\mathcal{A}$ -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule $OclIncluding$ -cong, simp, simp)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: $OclAsType_{Person}\mathcal{A}$ -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule $OclIncluding$ -cong, simp, simp)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: $OclAsType_{Person}\mathcal{A}$ -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule $OclIncluding$ -cong, simp, simp)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: $OclAsType_{Person}\mathcal{A}$ -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule $OclIncluding$ -cong, simp, simp)

```

apply(subst state-update-vs-allInstances-at-post-ntc, simp, simp add: OclAsTypePerson- $\mathcal{A}$ -def
      person8-def, simp, rule const-StrictRefEqSet-including, simp, simp, simp)
apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- $\mathcal{A}$ -def, simp, rule const-StrictRefEqSet-including
simp, simp, simp, rule OclIncluding-cong, simp, simp)
apply(rule state-update-vs-allInstances-at-post-empty)
by(simp-all add: OclAsTypePerson- $\mathcal{A}$ -def)

lemma  $\wedge \sigma_1$ .
( $\sigma_1, \sigma_1'$ )  $\models$  (OclAny.allInstances()  $\doteq$  Set{ XPerson1.oclAsType(OclAny), XPerson2.oclAsType(OclAny),
      XPerson3.oclAsType(OclAny), XPerson4.oclAsType(OclAny)
      (*, XPerson5*), XPerson6.oclAsType(OclAny),
      XPerson7, XPerson8, XPerson9.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
      XPerson1-def XPerson2-def XPerson3-def XPerson4-def XPerson5-def XPerson6-def XPerson7-def XPerson8-def
      XPerson9-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: OclAsTypeOclAny- $\mathcal{A}$ -def, simp, rule const-StrictRefEqSet-including
simp, simp, simp, rule OclIncluding-cong, simp, simp)+
apply(rule state-update-vs-allInstances-at-post-empty)
by(simp-all add: OclAsTypeOclAny- $\mathcal{A}$ -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 $Person\text{-}label_{inv} :: Person \Rightarrow Boolean$
where $Person\text{-}label_{inv}(self) \equiv$

$(self.boss <> null \text{ implies } (self.salary \leq_{int} ((self.boss).salary)))$

definition $Person\text{-}label_{invAT\ pre} :: Person \Rightarrow Boolean$

where $Person\text{-}label_{invAT\ pre}(self) \equiv$
 $(self.boss@pre <> null \text{ implies } (self.salary@pre \leq_{int} ((self.boss@pre).salary@pre)))$

definition $Person\text{-}label_{global\ inv} :: Boolean$

where $Person\text{-}label_{global\ inv} \equiv (Person.allInstances() \rightarrow includes(X \mid Person\text{-}label_{inv}(x)) \text{ and }$
 $(Person.allInstances@pre() \rightarrow includes(X \mid Person\text{-}label_{invAT\ pre}(x))))$

lemma $\tau \models \delta(X.boss) \implies \tau \models Person.allInstances() \rightarrow includes(X.boss) \wedge$
 $\tau \models Person.allInstances() \rightarrow includes(X)$

sorry

lemma $REC\text{-}pre : \tau \models Person\text{-}label_{global\ inv}$

$\implies \tau \models Person.allInstances() \rightarrow includes(X) \text{ (* } X \text{ represented object in state *)}$
 $\implies \exists REC. \tau \models REC(X) \triangleq (Person\text{-}label_{inv}(X) \text{ and } (X.boss <> null \text{ implies } REC(X.boss)))$

sorry

This allows to state a predicate:

axiomatization $inv_{Person\text{-}label} :: Person \Rightarrow Boolean$

where $inv_{Person\text{-}label}\text{-}def:$

$(\tau \models Person.allInstances() \rightarrow includes(self)) \implies$
 $(\tau \models (inv_{Person\text{-}label}(self) \triangleq (self.boss <> null \text{ implies }$
 $(self.salary \leq_{int} ((self.boss).salary)) \text{ and }$
 $inv_{Person\text{-}label}(self.boss))))$

axiomatization $inv_{Person\text{-}labelAT\ pre} :: Person \Rightarrow Boolean$

where $inv_{Person\text{-}labelAT\ pre}\text{-}def:$

$(\tau \models Person.allInstances@pre() \rightarrow includes(self)) \implies$
 $(\tau \models (inv_{Person\text{-}labelAT\ pre}(self) \triangleq (self.boss@pre <> null \text{ implies }$
 $(self.salary@pre \leq_{int} ((self.boss@pre).salary@pre)) \text{ and }$
 $inv_{Person\text{-}labelAT\ pre}(self.boss@pre))))$

lemma $inv\text{-}1 :$

$(\tau \models Person.allInstances() \rightarrow includes(self)) \implies$
 $(\tau \models inv_{Person\text{-}label}(self) = ((\tau \models (self.boss \doteq null)) \vee$
 $(\tau \models (self.boss <> null) \wedge$
 $\tau \models ((self.salary \leq_{int} (self.boss.salary)) \wedge$
 $\tau \models (inv_{Person\text{-}label}(self.boss))))$

sorry

lemma $inv\text{-}2 :$

$$\begin{aligned}
& (\tau \models \text{Person.allInstances@pre}() \rightarrow \text{includes}(\text{self})) \implies \\
& (\tau \models \text{inv}_{\text{Person-labelATpre}}(\text{self})) = ((\tau \models (\text{self}.\text{boss@pre} \doteq \text{null})) \vee \\
& \quad (\tau \models (\text{self}.\text{boss@pre} <> \text{null}) \wedge \\
& \quad (\tau \models (\text{self}.\text{boss@pre}.\text{salary@pre} \leq_{\text{int}} \text{self}.\text{salary@pre})) \wedge \\
& \quad (\tau \models (\text{inv}_{\text{Person-labelATpre}}(\text{self}.\text{boss@pre}))))))
\end{aligned}$$

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 $\text{inv} :: \text{Person} \Rightarrow (\mathcal{A})\text{st} \Rightarrow \text{bool}$ **where**

$$\begin{aligned}
& (\tau \models (\delta \text{ self})) \implies ((\tau \models (\text{self}.\text{boss} \doteq \text{null})) \vee \\
& \quad (\tau \models (\text{self}.\text{boss} <> \text{null}) \wedge (\tau \models (\text{self}.\text{boss}.\text{salary} \leq_{\text{int}} \text{self}.\text{salary})) \wedge \\
& \quad (\text{inv}(\text{self}.\text{boss})\tau))) \\
& \implies (\text{inv self } \tau)
\end{aligned}$$

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.

¹Our 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]), then 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

A. Overview of the OCL Semantics	1
A.1. Introduction	1
A.2. Background	4
A.2.1. A Running Example for UML/OCL	4
A.2.2. Formal Foundation	6
A.2.3. How this Annex A was Generated from Isabelle/HOL Theories	9
A.3. The Essence of UML-OCL Semantics	10
A.3.1. The Theory Organization	10
A.3.2. Denotational Semantics of Constants and Operations	11
A.3.3. Object-oriented Datatype Theories	17
A.3.4. Data Invariants	24
A.3.5. Operation Contracts	24
B. Formal Semantics of OCL	27
B.1. Formalization I: OCL Types and Core Definitions	27
B.1.1. Preliminaries	27
B.1.2. Basic OCL Value Types	31
B.1.3. Some OCL Collection Types	32
B.2. Formalization II: OCL Terms and Library Operations	35
B.2.1. The Operations of the Boolean Type and the OCL Logic	35
B.2.2. Property Profiles for OCL Operators via Isabelle Locales	60
B.2.3. Basic Type Void	68
B.2.4. Basic Type Integer: Operations	70
B.2.5. Basic Type Real: Operations	75
B.2.6. Basic Type String: Operations	80
B.2.7. Collection Type Pairs: Operations	83
B.2.8. Collection Type Set: Operations	86
B.2.9. Collection Type Sequence: Operations	145
B.2.10. Miscellaneous Stuff	149
B.3. Formalization III: UML/OCL constructs: State Operations and Objects	151
B.3.1. Introduction: States over Typed Object Universes	151
B.3.2. Operations on Object	154
B.4. Example I : The Employee Analysis Model (UML)	178
B.4.1. Introduction	178
B.4.2. Example Data-Universe and its Infrastructure	179

B.4.3.	Instantiation of the Generic Strict Equality	180
B.4.4.	OclAsType	181
B.4.5.	OclIsTypeOf	183
B.4.6.	OclIsKindOf	186
B.4.7.	OclAllInstances	189
B.4.8.	The Accessors (any, boss, salary)	192
B.4.9.	A Little Infra-structure on Example States	197
B.4.10.	OCL Part: Standard State Infrastructure	205
B.4.11.	Invariant	205
B.4.12.	The Contract of a Recursive Query	206
B.4.13.	The Contract of a User-defined Method	209
B.5.	Example II: The Employee Design Model (UML)	210
B.5.1.	Introduction	211
B.5.2.	Example Data-Universe and its Infrastructure	211
B.5.3.	Instantiation of the Generic Strict Equality	213
B.5.4.	OclAsType	213
B.5.5.	OclIsTypeOf	215
B.5.6.	OclIsKindOf	219
B.5.7.	OclAllInstances	221
B.5.8.	The Accessors (any, boss, salary)	224
B.5.9.	A Little Infra-structure on Example States	228
B.5.10.	OCL Part: Standard State Infrastructure	236
B.5.11.	Invariant	236
B.5.12.	The Contract of a Recursive Query	238
C.	Conclusion	239
C.1.	Lessons Learned and Contributions	239
C.2.	Lessons Learned	240
C.3.	Conclusion and Future Work	240