Part I.

Annex 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 FeatherWeight OCL — which has in itself a formally described semantics presented in Isabelle/HOL[26] [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 unambiquously defined, i. e. represent a value.

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

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

and:

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

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

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

[2] The solution to this little riddle can be found in Section 5.7.

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

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

- 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 FeatherWeight OCL contain the elements `null` and `invalid`; since this extends to `Boolean` type, this results in a four-valued logic. Consequently, FeatherWeight OCL 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, Featherweight OCL 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()`). [4]

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

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

- Featherweight OCL types may be arbitrarily nested. For example, the expression `Set{Set{1,2}} = Set{Set{2,1}}` is legal and true.

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

- As part of the OCL logic, Featherweight OCL 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, Featherweight OCL is a *semantic embedding* into a powerful semantic meta-language and environment, namely Isabelle/HOL [26]. 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 Featherweight OCL and a type in Featherweight OCL 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 [32] and [18, 21, 25], leading to a number of formal, machine-checked versions, most notably HOL-OCL [5, 6, 10] and more recent approaches [15]. All of them have in common the attempt to reconcile the conflicting demands of an industrially used specification language and its various stakeholders, the needs of OMG standardization process and the desire for sufficient logical precision for tool-implementors, in particular from the Formal Methods research community.

FiXme: *Something like this ? Shorten Paragraph !*

To discuss the future directions of the standard, several OCL experts met in November 2013 in Aachen to discuss possible mid-term improvements of OCL, strategies of standardization of OCL within the OMG, and a vision for possible long-term developments of the language [14]. During this meeting, a Request for Proposals (RFP) for OCL 2.5 was finalized and meanwhile proposed. In particular, this RFP requires that the future OCL 2.5 standard document shall be generated from a machine-checked source. This will ensure

- the absence of syntax errors,

- the consistency of the formal semantics,

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

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

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 FeatherWeight OCL 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,
    b) OCL Terms, i. e. the semantics of library operators, together with definitions, lemmas, and test cases for the implementor,
    c) UML/OCL Constructs, i. e. a core of UML class models plus user-defined constructions on them such as class-invariants and oepration constracts.
3. Since the latter, i. e. the construction of UML class models, has to be done on the meta-level (so not *inside* HOL, rather on the level of a pre-compiler), we will describe this process with two larger examples, namely formalizations of our running example.

2. Background

2.1. A Running Example for UML/OCL

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

The Unified Modeling Language (UML) [28, 29] 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 2.1 shows the class model. We model the hierarchy of roles of our system as a hierarchy of classes (e. g., `Hearer`, `Speaker`, or `Chair`) using an *inheritance* relation (also called *generalization*). In particular, *inheritance* establishes a *subtyping* relationship, i. e., every `Speaker` (*subclass*) is also a `Hearer` (*superclass*).

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

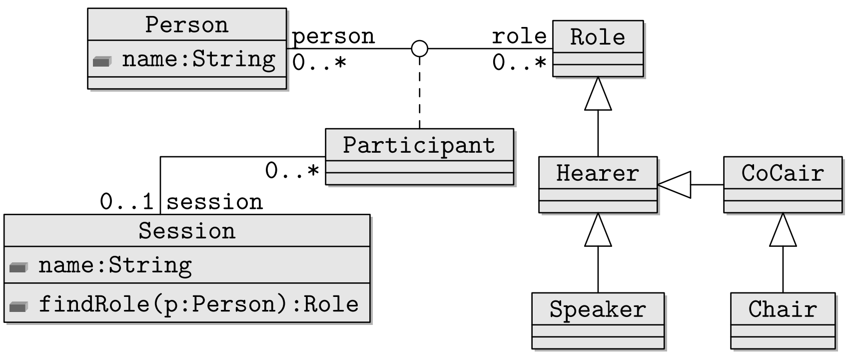


Figure 2.1.: A simple UML class model representing a conference system for organizing conference sessions: persons can participate, in different roles, in a session.

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 $\mathrm{o}_{[\mathrm{C}]}$ 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 [11] is to translate the diagrammatic UML features into a combination of more elementary features of UML and OCL expressions [20]. 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.

2.2. Formal Foundation

2.2.1. Isabelle

Isabelle [26] is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and natural deduction inference rules. Among other logics, Isabelle supports first-order logic, Zermelo-Fraenkel set theory and the instance for Church's higher-order logic (HOL).

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

$$[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1} \qquad \text{or, in mathematical notation,} \quad \frac{A_1 \qquad \cdots \qquad A_n}{A_{n+1}} \,. \tag{2.1}$$

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

$$\bigwedge x_1, \ldots, x_m.\ [\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1} \,. \tag{2.2}$$

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 [35] language, will look as follows in Isabelle:

```
lemma label:   φ
  apply(case_tac)
  apply(simp_all)
done
```
(2.3)

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

$$\begin{array}{rl} \text{label}: & \phi \\ 1. & \phi_1 \\ & \vdots \\ n. & \phi_n \end{array} \tag{2.4}$$

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

2.2.2. Higher-order Logic (HOL)

Higher-order logic (HOL) [1, 16] is a classical logic based on a simple type system. It provides the usual logical connectives like $\_ \wedge \_$, $\_ \longrightarrow \_$, $\neg \_$ as well as the object-logical quantifiers $\forall x.\ P\ x$ and $\exists x.\ P\ x$; in contrast to first-order logic, quantifiers may range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centered around extensional equality $\_ = \_ :: \alpha \Rightarrow \alpha \Rightarrow \text{bool}$. HOL is more expressive than first-order logic, since, e. g., induction schemes can be expressed inside the logic. Being based on some polymorphically typed λ-calculus, HOL can be viewed as a combination of a programming language like SML or Haskell and a specification language providing powerful logical quantifiers ranging over elementary and function types.

Isabelle/HOL is a logical embedding of HOL into Isabelle. The (original) simple-type system underlying HOL has been extended by Hindley-Milner style polymorphism with type-classes similar to Haskell. While Isabelle/HOL is usually seen as proof assistant, we use it as symbolic computation environment. Implementations on top of Isabelle/HOL can re-use existing powerful deduction mechanisms such as higher-order resolution, tableaux-based reasoners, rewriting procedures, Presburger arithmetic, and via various integration mechanisms, also external provers such as Vampire [31] and the SMT-solver Z3 [19].

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 $\tau_\perp := \perp \mid \llcorner\_\lrcorner : \alpha$ that assigns to each type τ a type $\tau_\perp$ *disjointly extended* by the exceptional element $\perp$. The function $\ulcorner\_\urcorner : \alpha_\perp \to \alpha$ is the inverse of $\llcorner\_\lrcorner$ (unspecified for $\perp$). Partial functions $\alpha \rightharpoonup \beta$ 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:[1]

$$
\begin{array}{llll}
\textsf{types} & \alpha\,\text{set} & = \alpha \Rightarrow \text{bool} & \\
\textsf{definition} & \text{Collect} & ::(\alpha \Rightarrow \text{bool}) \Rightarrow \alpha\,\text{set} & \text{— set comprehension} \\
\textsf{where} & \text{Collect}\ S & \equiv S & \\
\textsf{definition} & \text{member} & ::\alpha \Rightarrow \alpha \Rightarrow \text{bool} & \text{— membership test} \\
\textsf{where} & \text{member}\ s\ S & \equiv S s &
\end{array}
\tag{2.5}
$$

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

$$
\begin{array}{lll}
\textsf{datatype} & \text{option} & = \text{None} \mid \text{Some}\,\alpha \\
\textsf{datatype} & \alpha\,\text{list} & = \text{Nil} \mid \text{Cons}\ a\ l
\end{array}
\tag{2.6}
$$

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

$$\text{case}\ x\ \text{of}\ \ \text{None} \Rightarrow F\ \mid \text{Some}\ a \Rightarrow G\ a \tag{2.7}$$

respectively

$$\text{case}\ x\ \text{of}\ \ [] \Rightarrow F\ \mid \text{Cons}\ a\ r \Rightarrow G\ a\ r\,. \tag{2.8}$$

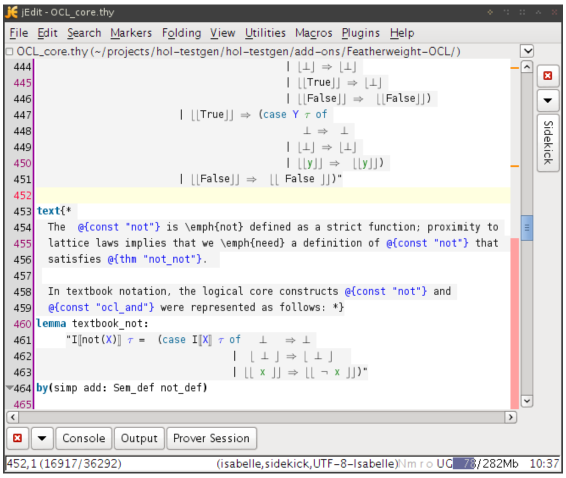
From the internal definitions (not shown here) several properties were automatically derived. We show only the case for lists:

$$
\begin{array}{ll}
(\text{case}\ []\ \text{of}\ [] \Rightarrow F \mid (a\#r) \Rightarrow G\ a\ r) = F & \\
(\text{case}\ b\#t\ \text{of}\ [] \Rightarrow F \mid (a\#r) \Rightarrow G\ a\ r) = G\ b\ t & \\
[] \neq a\#t & \text{– distinctness} \\
[\![a = [] \to P; \exists\ x\ t.\ a = x\#t \to P]\!] \Longrightarrow P & \text{– exhaust} \\
[\![P[]; \forall\ at.\ Pt \to P(a\#t)]\!] \Longrightarrow Px & \text{– induct}
\end{array}
\tag{2.9}
$$

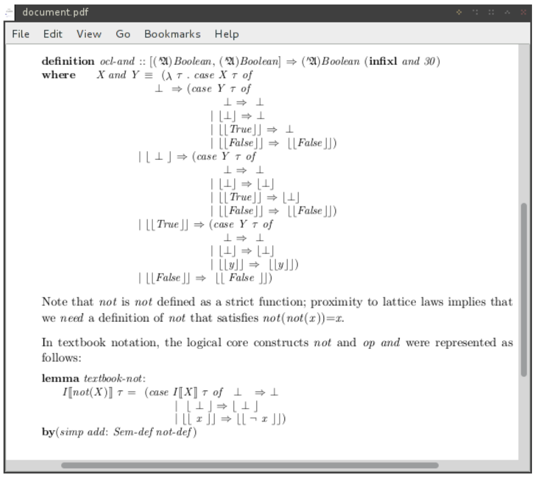
Finally, there is a compiler for primitive and wellfounded recursive function definitions. For example, we may define the sort operation of our running test example by:

$$
\begin{array}{lll}
\textsf{fun} & \text{ins} & ::[\alpha :: \text{linorder}, \alpha\,\text{list}] \Rightarrow \alpha\,\text{list} \\
\textsf{where} & \text{ins}\ x\ [\,] & = [x] \\
 & \text{ins}\ x\ (y\#ys) & = \text{if}\ x < y\ \text{then}\ x\#y\#ys\ \text{else}\ y\#(\text{ins}\ x\ ys)
\end{array}
\tag{2.10}
$$

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



(a) The Isabelle jEdit environment.



(b) The generated formal document.

Figure 2.2.: Generating documents with guaranteed syntactical and semantical consistency.

$$\begin{array}{lll} \textsf{fun} & \text{sort} & ::(\alpha :: \text{linorder})\ \text{list} \Rightarrow \alpha\ \text{list} \\ \textsf{where} & \text{sort}\,[\,] & = [\,] \\ & \text{sort}(x\#xs) & = \text{ins}\ x\ (\text{sort}\ xs) \end{array} \tag{2.11}$$

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.

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

FiXme: *Here ? Or in chap 2 ?*

Isabelle, as a framework for building formal tools [34], provides the means for generating *formal documents*. With formal documents (such as the one you are currently reading) we refer to documents that are machine-generated and ensure certain formal guarantees. In particular, all formal content (e. g., definitions, formulae, types) are checked for consistency during the document generation.

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

Figure 2.2 illustrates this approach: Figure 2.2a shows the jEdit-based development environment of Isabelle with an excerpt of one of the core theories of Featherweight OCL. Figure 2.2b 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 Featherweight OCL 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 [14].

3. Conceptual Overview

3.1. The Theory Organization

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

The semantic theory is organized in a quite conventional manner in three layers. The first layer, called the *denotational semantics* comprises a set of definitions of the operators of the language. Presented as *definitional axioms* inside Isabelle/HOL, this part assures the logically consistency of the overall construction. The 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 $(\sigma, \sigma') \models P$. 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 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.

3.1.1. Denotational Semantics

OCL is composed of

1. operators on built-in data structures such as `Boolean`, `Integer`, or `Set(A)`,
2. operators of the user-defined data-model such as accessors, type-casts and tests, and
3. user-defined, side-effect-free methods.

Conceptually, an OCL expression in general and Boolean expressions in particular (i. e., *formulae*) depends on the pair (σ, σ') of pre-and post-state. The precise form of states is irrelevant for this paper (compare [12]) and will be left abstract in this presentation. We construct in Isabelle a type-class null that contains two distinguishable elements bot and null. Any type of the form $(\alpha_\perp)_\perp$ is an instance of this type-class with bot $\equiv \perp$ and null $\equiv \lfloor\perp\rfloor$. Now, any OCL type can be represented by an HOL type of the form:

$$V(\alpha) := \mathrm{state} \times \mathrm{state} \to \alpha :: \mathrm{null} \ .$$

On this basis, we define $V((\mathrm{bool}_\perp)_\perp)$ as the HOL type for the OCL type `Boolean` and define:

$$I[\![\texttt{invalid} :: V(\alpha)]\!]\tau \equiv \mathrm{bot} \qquad I[\![\texttt{null} :: V(\alpha)]\!]\tau \equiv \mathrm{null}$$
$$I[\![\texttt{true} :: \texttt{Boolean}]\!]\tau = \lfloor\lfloor \mathrm{true} \rfloor\rfloor \qquad I[\![\texttt{false}]\!]\tau = \lfloor\lfloor \mathrm{false} \rfloor\rfloor$$

$$I[\![X\,\texttt{.oclIsUndefined()}]\!]\tau = (\text{if } I[\![X]\!]\tau \in \{\text{bot}, \text{null}\} \text{ then } I[\![\texttt{true}]\!]\tau \text{ else } I[\![\texttt{false}]\!]\tau)$$
$$I[\![X\,\texttt{.oclIsInvalid()}]\!]\tau = (\text{if } I[\![X]\!]\tau = \text{bot then } I[\![\texttt{true}]\!]\tau \text{ else } I[\![\texttt{false}]\!]\tau)$$

where $I[\![E]\!]$ is the semantic interpretation function commonly used in mathematical textbooks and τ stands for pairs of pre- and post state (σ, σ'). For reasons of conciseness, we will write $\delta\ X$ for `not` X`.oclIsUndefined()` and $\upsilon\ X$ for `not` X`.oclIsInvalid()` throughout this paper.

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

$$I[\![\texttt{true} :: \texttt{Boolean}]\!]\tau = \lfloor\lfloor \text{true} \rfloor\rfloor$$

we can therefore write:

$$\texttt{true} :: \texttt{Boolean} = \lambda\,\tau. \lfloor\lfloor \text{true} \rfloor\rfloor$$

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. Since all operators of the assertion language depend on the context $\tau = (\sigma, \sigma')$ and result in values that can be $\perp$, all expressions can be viewed as *evaluations* from (σ, σ') to a type α which must posses a $\perp$ and a null-element. Given that such constraints can be expressed in Isabelle/HOL via *type classes* (written: $\alpha :: \kappa$), all types for OCL-expressions are of a form captured by

$$V(\alpha) := \text{state} \times \text{state} \to \alpha :: \{bot, null\}\,,$$

where state stands for the system state and state $\times$ state describes the pair of pre-state and post-state and $\_ := \_$ denotes the type abbreviation.

The current OCL semantics [27, Annex A] uses different interpretation functions for invariants and pre-conditions; we achieve their semantic effect by a syntactic transformation $\_{\text{pre}}$ which replaces, for example, all accessor functions $\_.\,\text{a}$ by their counterparts $\_.\,\text{a}\,$`@pre`. For example, $(\mathit{self}.\,\text{a} > 5)_{\text{pre}}$ is just $(\mathit{self}.\,\text{a}\,\texttt{@pre} > 5)$. This way, also invariants and pre-conditions can be interpreted by the same interpretation function and have the same type of an evaluation $V(\alpha)$.

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

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

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

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

The default semantics for an OCL library operator is strict semantics; this means that the result of an operation f is invalid if one of its arguments is invalid. For a semantics comprising null, we suggest to stay conform to the standard and define the addition for integers as follows:

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

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

3.1.2. Logical Layer

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

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

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

$$\tau \vDash P \equiv \big(I[\![P]\!]\tau = \lfloor\lfloor\text{true}\rfloor\rfloor\big)\,.$$

On this basis, classical, two-valued inference rules can be established for reasoning over the logical connective, the different notions of equality, definedness and validity.

Generally speaking, rules over logical validity can relate bits and pieces in various OCL terms and allow—via strong logical equality discussed below—the replacement of semantically equivalent sub-expressions. The core inference rules are:

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

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

We propose to distinguish the *strong logical equality* (written $\_ \triangleq \_$), which follows the general principle that "equals can be replaced by equals," from the *strict referential equality* (written $\_ \doteq \_$), which is an object-oriented concept that attempts to approximate and to implement the former. Strict referential equality, which is the default in the OCL language and is written `_ = _` in the standard, is an overloaded concept and has to be defined for each OCL type individually; for objects resulting from class definitions, it is implemented by comparing the references to the objects. In contrast, strong logical equality is a polymorphic concept which is defined once and for all by:

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

It enjoys nearly the laws of a congruence:

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

where the predicate cp stands for *context-passing*, a property that is characterized by $P(X)$ equals $\lambda\,\tau.\ P(\lambda\_.\ X\tau)\tau$. It means that the state tuple $\tau = (\sigma, \sigma')$ is passed unchanged from surrounding expressions to sub-expressions. it is true for all pure OCL expressions (but not arbitrary mixtures of OCL and HOL) in Featherweight OCL. The necessary side-calculus for establishing cp can be fully automated.

The logical layer of the Featherweight OCL 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 [19]. δ-closure rules for all logical connectives have the following format, e. g.:

$$\tau \models \delta\, x \Longrightarrow (\tau \models\ \texttt{not}\ x) = (\neg(\tau \models x))$$
$$\tau \models \delta\, x \Longrightarrow \tau \models \delta\, y \Longrightarrow (\tau \models x\ \texttt{and}\ y) = (\tau \models x \wedge \tau \models y)$$

$$\tau \models \delta\, x \Longrightarrow \tau \models \delta\, y$$
$$\Longrightarrow (\tau \models (x \,\texttt{implies}\, y)) = ((\tau \models x) \longrightarrow (\tau \models y))$$

Together with the general case-distinction

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

which is possible for any OCL type, a case distinction on the variables in a formula can be performed; due to strictness rules, formulae containing somewhere a variable x that is known to be `invalid` or `null` reduce usually quickly to contradictions. For example, we can infer from an invariant $\tau \models x \doteq y \,\texttt{-}\, 3$ that we have $\tau \models x \doteq y \,\texttt{-}\, 3 \wedge \tau \models \delta\, x \wedge \tau \models \delta\, y$. We call the latter formula the δ-closure of the former. Now, we can convert a formula like $\tau \models x \,\texttt{>}\, 0 \,\texttt{or}\, 3 \,\texttt{*}\, y \,\texttt{>}\, x \,\texttt{*}\, x$ into the equivalent formula $\tau \models x \,\texttt{>}\, 0 \vee \tau \models 3 \,\texttt{*}\, y \,\texttt{>}\, x \,\texttt{*}\, x$ 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.

3.1.3. 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, where the used equality is the meta-(HOL-)equality.

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

$$\upsilon\, \texttt{invalid} = \texttt{false} \qquad \upsilon\, \texttt{null} = \texttt{true}$$
$$\upsilon\, \texttt{true} = \texttt{true} \qquad \upsilon\, \texttt{false} = \texttt{true}$$
$$\delta\, \texttt{invalid} = \texttt{false} \qquad \delta\, \texttt{null} = \texttt{false}$$
$$\delta\, \texttt{true} = \texttt{true} \qquad \delta\, \texttt{false} = \texttt{true}$$
$$\texttt{not invalid} = \texttt{invalid} \qquad \texttt{not null} = \texttt{null}$$
$$\texttt{not true} = \texttt{false} \qquad \texttt{not false} = \texttt{true}$$
$$(\texttt{null and true}) = \texttt{null} \qquad (\texttt{null and false}) = \texttt{false}$$
$$(\texttt{null and null}) = \texttt{null} \qquad (\texttt{null and invalid}) = \texttt{invalid}$$
$$(\texttt{false and true}) = \texttt{false} \qquad (\texttt{false and false}) = \texttt{false}$$
$$(\texttt{false and null}) = \texttt{false} \qquad (\texttt{false and invalid}) = \texttt{false}$$
$$(\texttt{true and true}) = \texttt{true} \qquad (\texttt{true and false}) = \texttt{false}$$
$$(\texttt{true and null}) = \texttt{null} \qquad (\texttt{true and invalid}) = \texttt{invalid}$$
$$(\texttt{invalid and true}) = \texttt{invalid}$$
$$(\texttt{invalid and false}) = \texttt{false}$$
$$(\texttt{invalid and null}) = \texttt{invalid}$$
$$(\texttt{invalid and invalid}) = \texttt{invalid}$$

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

$$X \texttt{ and } X = X \qquad X \texttt{ and } Y = Y \texttt{ and } X$$
$$\texttt{false and } X = \texttt{false} \qquad X \texttt{ and false} = \texttt{false}$$
$$\texttt{true and } X = X \qquad X \texttt{ and true} = X$$
$$X \texttt{ and } (Y \texttt{ and } Z) = X \texttt{ and } Y \texttt{ and } Z$$

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

The above equations explain the behavior for the most-important non-strict operations. The clarification of the exceptional behaviors is of key-importance for a semantic definition the standard and the major deviation point from HOL-OCL [5, 7], to Featherweight OCL as presented here. The standard expresses at many places that most operations are strict, i. e., enjoy the properties (exemplary for `_ + _`):

$$\texttt{invalid + } X = \texttt{invalid} \qquad X \texttt{ + invalid} = \texttt{invalid}$$
$$X \texttt{ + null} = \texttt{invalid} \qquad \texttt{null + } X = \texttt{invalid}$$
$$\texttt{null.oclAsType(}X\texttt{)} = \texttt{invalid}$$

besides "classical" exceptional behavior:

$$\texttt{1 / 0} = \texttt{invalid} \qquad \texttt{1 / null} = \texttt{invalid}$$
$$\texttt{null->isEmpty()} = \texttt{true}$$

Moreover, there is also the proposal to use `null` as a kind of "don't know" value for all strict operations, not only in the semantics of the logical connectives. Expressed in algebraic equations, this semantic alternative (this is *not* Featherweight OCL at present) would boil down to:

$$\texttt{invalid + } X = \texttt{invalid} \qquad X \texttt{ + invalid} = \texttt{invalid}$$
$$X \texttt{ + null} = \texttt{null} \qquad \texttt{null + } X = \texttt{null}$$
$$\texttt{null.oclAsType(}X\texttt{)} = \texttt{null}$$
$$\texttt{1 / 0} = \texttt{invalid} \qquad \texttt{1 / null} = \texttt{null}$$
$$\texttt{null->isEmpty()} = \texttt{null}$$

While this is logically perfectly possible, while it can be argued that this semantics is "intuitive", and although we do not expect a too heavy cost in deduction when computing δ-closures, we object that there are other, also "intuitive" interpretations that are even more wide-spread: In classical spreadsheet programs, for example, the semantics tends to interpret `null` (representing empty cells in a sheet) as the neutral element of the type, so `0` or the empty string, for example.[1] This semantic alternative (this is *not*

[1] In spreadsheet programs the interpretation of null varies from operation to operation; e. g., the `average` function treats `null` as non-existing value and not as `0`.

Featherweight OCL at present) would yield:

$$\texttt{invalid} + X = \texttt{invalid} \qquad X + \texttt{invalid} = \texttt{invalid}$$
$$X + \texttt{null} = X \qquad \texttt{null} + X = X$$
$$\texttt{null.oclAsType}(X) = \texttt{invalid}$$
$$\texttt{1 / 0} = \texttt{invalid} \qquad \texttt{1 / null} = \texttt{invalid}$$
$$\texttt{null->isEmpty()} = \texttt{true}$$

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

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

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

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

an expression like `Set{1,2}->includes(null)` becomes decidable in Featherweight OCL by a combination of rewriting and code-generation and execution. The generated documentation from the theory files can thus be enriched by numerous "test-statements" like:

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

which have been machine-checked and which present a high-level and in our opinion fairly readable information for OCL tool manufactures and users.

3.2. Object-oriented Datatype Theories

As mentioned earlier, the OCL is composed of

1. operators on built-in data structures such as Boolean, Integer or Set(_), and

2. operators of the user-defined data model such as accessors, type casts and tests.

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. In contrast to wide-spread opinions, UML class diagrams represent in a compact and visual manner quite complex, object-oriented data-types with a surprisingly rich theory. It is part of our endeavor here to make this theory explicit and to point out corner cases. A UML class diagram—underlying a given OCL formula—produces several implicit operations which become accessible via appropriate OCL syntax:

1. Classes and class names (written as $C_1, \ldots, C_n$), which become types of data in OCL. Class names declare two projector functions to the set of all objects in a state: C_i`.allInstances()` and C_i`.allInstances@pre()`,
2. an inheritance relation $\_ < \_$ on classes and a collection of attributes A associated to classes,
3. two families of accessors; for each attribute a in a class definition (denoted $X.a :: C_i \to A$ and $X.a$ `@pre` $:: C_i \to A$ for $A \in \{V(\ldots_\perp), C_1, \ldots, C_n\}$),
4. type casts that can change the static type of an object of a class (X`.oclAsType(`C_i`)` of type $C_j \to C_i$)
5. two dynamic type tests (X`.oclIsTypeOf(`C_i`)` and X`.oclIsKindOf(`C_i`)`),
6. and last but not least, for each class name C_i there is an instance of the overloaded referential equality (written $\_ \doteq \_$).

Assuming a strong static type discipline in the sense of Hindley-Milner types, Featherweight OCL has no "syntactic subtyping." This does not mean that subtyping cannot be expressed *semantically* in Featherweight OCL; by giving a formal semantics to type-casts, 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.

3.2.1. Object Universes

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

$$\text{typedef} \qquad \alpha \ \text{state} := \{\sigma :: \text{oid} \rightharpoonup \alpha \mid \text{inv}_\sigma(\sigma)\} \tag{3.1}$$

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 for 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, we chose the first option for Featherweight OCL, 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: $C_i := (\mathrm{oid} \times A_{i_1} \times \cdots \times A_{i_k})$ where the types $A_{i_1}, \ldots, A_{i_k}$ are the attribute types (including inherited attributes) with class types substituted by oid. The function OidOf projects the first component, the oid, out of an object representation. Then the object universe will be constructed by the type definition:

$$\mathfrak{A} := C_1 + \cdots + C_n \, . \tag{3.2}$$

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

$$X.\texttt{oclIsTypeOf(}C_k\texttt{)}\ \texttt{implies}\ X\texttt{.oclAsType(}C_i\texttt{).oclAsType(}C_k\texttt{)} \doteq X \tag{3.3}$$

$$\text{whenever } C_k < C_i \text{ and } X \text{ is valid.} \tag{3.4}$$

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

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

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

- Then the *class type* for C_i is $oid \times A_{i_1} \times \cdots \times A_{i_n} \times (C_{j_1\mathrm{ext}} + \cdots + C_{j_m\mathrm{ext}})_\perp$ where A_{i_k} ranges over the inherited *and* local attribute types of C_i and $C_{j_l\mathrm{ext}}$ 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 Chapter 7 and Chapter 8.

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 Chapter 7 and Chapter 8 and the derivation of the respective datatype theories. While an automatization is clearly possible and desirable for concrete applications of Featherweight OCL, we consider this out of the scope of this paper which has a focus on the semantic construction and its presentation.

3.2.2. 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 Featherweight OCL. 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:

$$
\begin{array}{llll}
\textcolor{teal}{\text{definition}} & & & \\
\text{eval\_extract}\, X\; f = (\lambda\, \tau.\ \text{case}\, X\, \tau\, \text{of} & \bot & \Rightarrow \texttt{invalid}\,\tau & \text{exception} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad | & \llcorner\bot\lrcorner & \Rightarrow \texttt{invalid}\,\tau & \text{deref. null} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad | & \llcorner\llcorner \mathit{obj} \lrcorner\lrcorner & \Rightarrow f\,(\text{oid\_of}\ \mathit{obj})\,\tau) &
\end{array}
\tag{3.5}
$$

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

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textcolor{teal}{\text{definition}}\ \text{deref\_oid}_C\ \mathit{fst\_snd}\ f\ \mathit{oid} = (\lambda\,\tau.\ \text{case}\,(\text{heap}\,(\mathit{fst\_snd}\ \tau))\ \mathit{oid}\ \text{of}} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \llcorner \text{in}_C\, \mathit{obj} \lrcorner & \Rightarrow f\ \mathit{obj}\ \tau \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad | \ \_ & \Rightarrow \texttt{invalid}\,\tau)
\end{array}
\tag{3.6}
$$

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:

$$
\begin{array}{llllll}
\textcolor{teal}{\text{definition}}\ \text{select}_a\ f = (\lambda & \text{mk}_C\ \mathit{oid} & \cdots\bot\cdots & C_{X\text{ext}} & \Rightarrow \texttt{null} \\
\qquad\qquad\qquad\qquad\qquad\quad | & \text{mk}_C\ \mathit{oid} & \cdots\llcorner a \lrcorner\cdots & C_{X\text{ext}} & \Rightarrow f\ (\lambda\ x\ \_.\ \llcorner\llcorner x \lrcorner\lrcorner)\ a)
\end{array}
\tag{3.7}
$$

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:

$$
\begin{array}{rrll}
\textcolor{teal}{\text{definition}} & \text{in\_pre\_state} & = \text{fst} & \text{first component} \\
\textcolor{teal}{\text{definition}} & \text{in\_post\_state} & = \text{snd} & \text{second component} \\
\textcolor{teal}{\text{definition}} & \text{reconst\_basetype} & = \text{id} & \text{identity function}
\end{array}
\tag{3.8}
$$

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

$$
\begin{array}{lll}
\textcolor{teal}{\text{definition}} & \_\texttt{.getBase} & :: C \Rightarrow A_{base} \\
\textcolor{teal}{\text{where}} & X\texttt{.getBase} & = \text{eval\_extract}\ X\ (\text{deref\_oid}_C\ \text{in\_post\_state} \\
 & & \quad (\text{select}_{\text{getBase}}\ \text{reconst\_basetype}))
\end{array}
\tag{3.9}
$$

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

$$
\begin{array}{lll}
\textcolor{teal}{\text{definition}} & \_\texttt{.getObject} & :: C \Rightarrow A_{object} \\
\textcolor{teal}{\text{where}} & X\texttt{.getObject} & = \text{eval\_extract}\ X\ (\text{deref\_oid}_C\ \text{in\_post\_state} \\
 & & \quad (\text{select}_{\text{getObject}}\ (\text{deref\_oid}_C\ \text{in\_post\_state})))
\end{array}
\tag{3.10}
$$

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 7.8, the construction of accessors via attributes in Section 8.8. 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.[2] In case a multiplicity is specified for an attribute, i. e., a

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

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 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 3.2.2. 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.

3.2.3. Other Operations on States

Defining `_.allInstances()` is straight-forward; the only difference is the property T `.allInstances()->excludes(null)` which is a consequence of the fact that `null`'s are values and do not "live" in the state. In our semantics which admits states with "dangling references," it is possible to define a counterpart to `_.oclIsNew()` called `_.oclIsDeleted()` which asks if an object id (represented by an object representation) is contained in the pre-state, but not the post-state.

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 [22]). 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:

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

where $X' = I[\![X]\!](\sigma,\sigma')$ and $M = (\text{dom}\ \sigma \cap \text{dom}\ \sigma') - \{\text{OidOf}\, x |\ x \in \ulcorner X' \urcorner\}$. Thus, if we require in a postcondition `Set{}->oclIsModifiedOnly()` and exclude via `_.oclIsNew()` and `_.oclIsDeleted()` the existence of new or deleted objects, the operation is a query in the sense of the OCL standard, i. e., the `isQuery` property is true. So, whenever we have $\tau \vDash X\text{->}\texttt{excluding}(s.a)\text{->}\texttt{oclIsModifiedOnly()}$ and $\tau \vDash X\texttt{->forAll(}x\texttt{|not(}x \doteq s.a\texttt{))}$, we can infer that $\tau \vDash s.a \triangleq s.a\,\texttt{@pre}$.

3.3. Data Invariants

(* The OCL semantics uses different interpretation functions for invariants and preconditions; instead, we achieve their semantic effect by a syntactic transformation $\_{\text{pre}}$ which replaces all accessor functions $\_.\,\text{i}$ by their counterparts $\_.\,\text{i}\,\texttt{@pre}$. For example, $(\textit{self}.\,\text{i} > 5)_{\text{pre}}$ is just $(\textit{self}.\,\text{i}\,\texttt{@pre} > 5)$. The operation `_.allInstances()` is also substituted by its `@pre` counterpart. Thus, we can re-formulate the semantics of the two OCL top-level constructs, invariant specification and method specification, as follows:

$$\begin{aligned} &I[\![\texttt{context}\ c : C\ \texttt{inv}\ n : \phi(c)]\!]\tau \equiv \\ &\quad \tau \vDash (C\ \texttt{.allInstances()->forall}(x|\phi(x))) \wedge \\ &\quad \tau \vDash (C\ \texttt{.allInstances()->forall}(x|\phi(x)))_{\text{pre}} \end{aligned} \tag{3.11}$$

The standard forbids expressions containing `@pre` constructs in invariants or preconditions syntactically; thus, mixed forms cannot arise. Since operations have strict semantics in OCL, we have to distinguish for a specification of an *op* with the arguments a_1, ..., a_n the two cases where all arguments are defined (and *self* is non-null), or not. In the former case, a method call can be replaced by a *result* that satisfies the contract, in the latter case the argument is $\bot$:

$$
\begin{aligned}
I\llbracket &\texttt{context } C \;::\; op(a_1, \dots, a_n) : T \\
&\quad \texttt{pre } \phi(\mathit{self}, a_1, \dots, a_n) \\
&\quad \texttt{post } \psi(\mathit{self}, a_1, \dots, a_n, \mathit{result})\rrbracket\tau \equiv \forall s, x_1, \dots, x_n. \\
&\qquad\qquad\qquad \Delta(s, x_1, \dots, x_n) \land \tau \vDash \phi(s, x_1, \dots, x_n)_{\text{pre}} \\
&\qquad\qquad\qquad\quad \to \tau \vDash \psi(s, x_1, \dots, x_n, s.op(x_1, \dots, x_n)) \\
&\qquad\qquad\qquad \land \neg\Delta(s, x_1, \dots, x_n) \to \tau \vDash s.op(x_1, \dots, x_n) \triangleq \bot
\end{aligned}
\tag{3.12}
$$

where $\Delta(s, x_1, \dots, x_n)$ is an abbreviation for $\tau \vDash s \not\triangleq \texttt{null} \land \tau \vDash \partial\, s \land \tau \vDash \partial\, x_1 \land \dots \tau \vDash \partial\, x_n$. This definition captures the two cases: if the arguments of an operation are defined and, moreover, *self* is not `null`, the result of a method call must satisfy the specification; otherwise the operation will be strict and return invalid $\bot$. By these definitions an OCL specification, i. e., a sequence of invariant declarations and operation contracts, can be transformed into a set of (logically conjoined) statements which is called the *context* Γ_τ. The *theory* of an OCL specification is the set of all valid transitions $\tau \vDash \phi$ that can be derived from Γ_τ. For the logical connectives of OCL, a conventional Gentzen-style calculus for pairs of the form $\Gamma_\tau \vdash \phi$ can be developed that allows for inferring valid transitions from Γ_τ by deduction (cf. [9]). Due to the inclusion of arithmetic, any calculus for OCL is necessarily incomplete. It is straight-forward to extend our notion of context to multi-transition contexts such as:

$$\Gamma \equiv \big\{(\sigma, \sigma') \vDash \phi, (\sigma', \sigma'') \vDash \psi\big\}$$

such that we can reason over systems executing several transitions. *)

3.4. Operation Contracts

Part II.

A Proposal for Formal Semantics of OCL 2.5

4. Formalization I: OCL Types and Core Definitions

theory *OCL-Types*
imports *Main*

keywords *Assert* :: *thy-decl*
and *Assert-local* :: *thy-decl*
begin

4.1. Preliminaries

4.1.1. Notations for the Option Type

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

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

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

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

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

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

4.1.2. 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 $Set\{Set\{\mathbf{2}\},null\}$, it is necessary to introduce a uniform interface for types having the *invalid* (= bottom) element. The reason is that we impose a data-invariant on raw-collection **types_code** which assures that the *invalid* element is not allowed inside the collection; all raw-collections of this form were identified with the *invalid* element itself. The construction requires that the new collection type is 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 $null\text{-}is\text{-}valid : null \neq bot$

4.1.3. 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 $bot\text{-}option\text{-}def$: $(bot::'a\ option) \equiv (None::'a\ option)$
  instance $\langle proof \rangle$
end

instantiation $option :: (bot)null$
begin
  definition $null\text{-}option\text{-}def$: $(null::'a::bot\ option) \equiv \lfloor\ bot\ \rfloor$
  instance $\langle proof \rangle$

end

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

  instance ⟨*proof*⟩
end

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

 instance ⟨*proof*⟩
end

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

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

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

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

type-synonym $oid = nat$

We refrained from the alternative:

type-synonym $oid = ind$

which is slightly more abstract but non-executable.

States in UML/OCL are a pair of

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

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

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

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

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

FiXme: *Get Appropriate Reference!*

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

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

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

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*: $oid\text{-}of\ x = oid\text{-}of\ (the\ x)$
  instance $\langle proof \rangle$
end

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

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

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

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

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

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

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

definition *invalid* :: $('\mathfrak{A},'\alpha::bot)$ *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*: $I[\![invalid]\!]\tau = bot$
⟨*proof*⟩

Note that the definition :

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

is not necessary since we defined the entire function space over null types again as null-types; the crucial definition is $null \equiv \lambda x.\ null$. 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*: $I[\![null::('\mathfrak{A},'\alpha::null)\ val]\!]\ \tau = (null::('\alpha::null))$
⟨*proof*⟩

4.2. 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 $('\mathfrak{A})Boolean = ('\mathfrak{A},Boolean_{base})\ val$

Because of the previous class definitions, Isabelle type-inference establishes that $'\mathfrak{A}$ *Boolean* lives actually both in the type class *OCL-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 $('\mathfrak{A})Integer = ('\mathfrak{A},Integer_{base})\ val$

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

type-synonym $Real_{base}$ = *nat option option*
type-synonym $('\mathfrak{A})Real = ('\mathfrak{A},Real_{base})\ val$

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 $Void_{base} = \{X::unit\ option\ option.\ X = bot \lor X = null\ \}$ $\langle proof \rangle$

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

4.3. Some OCL Collection Types

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

4.3.1. 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 $('\alpha,\ '\beta)\ Pair_{base} = \{X::('\alpha::null \times\ '\beta::null)\ option\ option.$
$X = bot \lor X = null \lor (fst\lceil\lceil X\rceil\rceil \neq bot \land snd\lceil\lceil X\rceil\rceil \neq bot)\}$
$\langle proof \rangle$

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

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

instance $\langle proof \rangle$
end

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

instance $\langle proof \rangle$
end

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

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

4.3.2. The Construction of the Set Type

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

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

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

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

instance $\langle proof \rangle$
end

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

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

instance $\langle proof \rangle$
end

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

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

4.3.3. The Construction of the Sequence Type

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

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

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

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

instance $\langle proof \rangle$
end

instantiation $Sequence_{base}$ :: $(null)null$

begin

definition *null-Sequence$_{base}$-def*: (*null*::(*'a*::*null*) *Sequence$_{base}$*) ≡ *Abs-Sequence$_{base}$* ⌊ *None* ⌋

instance ⟨*proof*⟩
end

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

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

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

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

After the previous sections, we are now able to exemplify this representation as follows:

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

Table 4.1.: Basic semantic constant definitions of the logic (except *null*)

We do not formalize the representation map here; however, its principles are quite straight-forward:

1. cartesion products of arguments were curried,
2. constants of type `T` were mapped to valuations over the HOL-type for `T`,
3. functions `T -> T'` were mapped to functions in HOL, where `T` and `T'` were mapped to the valuations for them, and
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.

⟨*ML*⟩

end

5. Formalization II: OCL Terms and Library Operations

theory *OCL-core*
imports *OCL-Types*
begin

5.1. The Operations of the Boolean Type and the OCL Logic

5.1.1. Basic Constants

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

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

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

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

lemma *bool-split-0*: $X\ \tau = invalid\ \tau \lor X\ \tau = null\ \tau \lor$
$X\ \tau = true\ \tau \quad \lor X\ \tau = false\ \tau$
⟨*proof*⟩

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

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

lemma *textbook-true*: $I[\![true]\!]\ \tau = \lfloor\lfloor True\rfloor\rfloor$
⟨*proof*⟩

lemma *textbook-false*: $I[\![false]\!]\ \tau = \lfloor\lfloor False\rfloor\rfloor$
⟨*proof*⟩

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

Table 5.1.: Basic semantic constant definitions of the logic (except *null*)

5.1.2. Validity and Definedness

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

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

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

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

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

lemma $valid4[simp]$: $\upsilon\ false = true$
⟨*proof*⟩

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

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

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

lemma $defined1[simp]$: $\delta\ invalid = false$
⟨*proof*⟩

lemma $defined2[simp]$: $\delta\ null = false$
⟨*proof*⟩

lemma $defined3[simp]$: $\delta\ true = true$
⟨*proof*⟩

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

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

lemma *defined6* [*simp*]: δ υ X $=$ *true*
⟨*proof*⟩

lemma *valid5* [*simp*]: υ υ X $=$ *true*
⟨*proof*⟩

lemma *valid6* [*simp*]: υ δ X $=$ *true*
⟨*proof*⟩

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

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

lemma *textbook-defined*: $I[\![\delta(X)]\!]\ \tau = (\mathit{if}\ I[\![X]\!]\ \tau = I[\![\mathit{bot}]\!]\ \tau\ \vee I[\![X]\!]\ \tau = I[\![\mathit{null}]\!]\ \tau$
$\mathit{then}\ I[\![\mathit{false}]\!]\ \tau$
$\mathit{else}\ I[\![\mathit{true}]\!]\ \tau)$
⟨*proof*⟩

lemma *textbook-valid*: $I[\![\upsilon(X)]\!]\ \tau = (\mathit{if}\ I[\![X]\!]\ \tau = I[\![\mathit{bot}]\!]\ \tau$
$\mathit{then}\ I[\![\mathit{false}]\!]\ \tau$
$\mathit{else}\ I[\![\mathit{true}]\!]\ \tau)$
⟨*proof*⟩

Table 5.2 and Table 5.3 summarize the results of this section.

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

Table 5.2.: Basic predicate definitions of the logic.

5.1.3. 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

Name	Theorem
defined1	δ *invalid* $=$ *false*
defined2	δ *null* $=$ *false*
defined3	δ *true* $=$ *true*
defined4	δ *false* $=$ *true*
defined5	$\delta\ \delta\ X$ $=$ *true*
defined6	$\delta\ \upsilon\ X$ $=$ *true*

Table 5.3.: Laws of the basic predicates of the logic.

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 [18] and was identified as desirable extension of OCL in the Aachen Meeting [14] in the future 2.5 OCL Standard. The purpose of strong equality is to define and reason over OCL. It is therefore a natural task in Featherweight OCL to formally investigate the somewhat quite complex relationship between these two.

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

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

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

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

2. The fundamental reason goes as follows: whatever you do to reason consistently over a language, you need the concept of equality: you need to know what expressions can be replaced by others because they *mean the same thing.* People call this also "Leibniz Equality" because this philosopher brought this principle first explicitly to paper and shed some light over it. It is the theoretic foundation of what you do in an optimizing compiler: you replace expressions by *equal* ones,

which you hope are easier to evaluate. In a typed language, strong equality exists uniformly over all types, it is "polymorphic" $\_ = \_ :: \alpha * \alpha \to bool$—this is the way that equality is defined in HOL itself. We can express Leibniz principle as one logical rule of surprising simplicity and beauty:

$$s = t \Longrightarrow P(s) = P(t) \tag{5.1}$$

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

From this follow already elementary properties like:

lemma [*simp*,*code-unfold*]: $(true \triangleq false) = false$
⟨*proof*⟩

lemma [*simp*,*code-unfold*]: $(false \triangleq true) = false$
⟨*proof*⟩

Fundamental Predicates on Strong Equality

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

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

lemma *StrongEq-sym*: $(X \triangleq Y) = (Y \triangleq X)$
⟨*proof*⟩

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

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

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

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

lemma *valid7*[*simp*]: $\upsilon\ (X \triangleq Y) = true$
⟨*proof*⟩

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

5.1.4. Logical Connectives and their Universal Properties

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

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

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

definition *OclNot* :: $('\mathfrak{A})Boolean \Rightarrow ('\mathfrak{A})Boolean$ (*not*)

where $not\ X \equiv\ \lambda\ \tau\ .\ case\ X\ \tau\ of$
$$\begin{array}{lll} & \bot & \Rightarrow \bot \\ | & \lfloor \bot \rfloor & \Rightarrow \lfloor \bot \rfloor \\ | & \lfloor\lfloor x \rfloor\rfloor & \Rightarrow \lfloor\lfloor \neg\ x \rfloor\rfloor \end{array}$$

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

lemma *OclNot1* [*simp*]: *not invalid = invalid*
⟨*proof*⟩

lemma *OclNot2* [*simp*]: *not null = null*
⟨*proof*⟩

lemma *OclNot3* [*simp*]: *not true = false*
⟨*proof*⟩

lemma *OclNot4* [*simp*]: *not false = true*
⟨*proof*⟩

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

lemma *OclNot-inject*: $\bigwedge\ x\ y.\ not\ x = not\ y \Longrightarrow x = y$
⟨*proof*⟩

definition *OclAnd* :: $[({'}\mathfrak{A})Boolean, ({'}\mathfrak{A})Boolean] \Rightarrow ({'}\mathfrak{A})Boolean$ (**infixl** *and 30*)
where $X\ and\ Y \equiv\ (\lambda\ \tau\ .\ case\ X\ \tau\ of$
$$\begin{array}{lll} & \lfloor\lfloor False \rfloor\rfloor \Rightarrow & \lfloor\lfloor False \rfloor\rfloor \\ | & \bot & \Rightarrow (case\ Y\ \tau\ of \\ & & \quad \lfloor\lfloor False \rfloor\rfloor \Rightarrow \lfloor\lfloor False \rfloor\rfloor \\ & & \quad |\ \text{-} \quad \Rightarrow \bot) \\ | & \lfloor\bot\rfloor & \Rightarrow (case\ Y\ \tau\ of \\ & & \quad \lfloor\lfloor False \rfloor\rfloor \Rightarrow \lfloor\lfloor False \rfloor\rfloor \\ & & \quad |\ \bot \quad \Rightarrow \bot \\ & & \quad |\ \text{-} \quad \Rightarrow \lfloor\bot\rfloor) \\ | & \lfloor\lfloor True \rfloor\rfloor & \Rightarrow \quad Y\ \tau) \end{array}$$

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

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

lemma *textbook-OclNot*:
$$I[\![not(X)]\!]\ \tau = \ (case\ I[\![X]\!]\ \tau\ of\ \ \begin{array}{ll} & \bot \ \Rightarrow \bot \\ | & \lfloor \bot \rfloor \Rightarrow \lfloor \bot \rfloor \\ | & \lfloor\lfloor x \rfloor\rfloor \Rightarrow \lfloor\lfloor \neg\ x \rfloor\rfloor) \end{array}$$

$\langle proof \rangle$

lemma *textbook-OclAnd*:

$$
\begin{array}{l}
I[\![X \; and \; Y]\!] \; \tau = (case \; I[\![X]\!] \; \tau \; of \\
\quad \bot \; \Rightarrow (case \; I[\![Y]\!] \; \tau \; of \\
\qquad \bot \Rightarrow \bot \\
\qquad | \; \lfloor\bot\rfloor \Rightarrow \bot \\
\qquad | \; \lfloor\lfloor True\rfloor\rfloor \Rightarrow \bot \\
\qquad | \; \lfloor\lfloor False\rfloor\rfloor \Rightarrow \lfloor\lfloor False\rfloor\rfloor) \\
\quad | \; \lfloor \bot \rfloor \Rightarrow (case \; I[\![Y]\!] \; \tau \; of \\
\qquad \bot \Rightarrow \bot \\
\qquad | \; \lfloor\bot\rfloor \Rightarrow \lfloor\bot\rfloor \\
\qquad | \; \lfloor\lfloor True\rfloor\rfloor \Rightarrow \lfloor\bot\rfloor \\
\qquad | \; \lfloor\lfloor False\rfloor\rfloor \Rightarrow \lfloor\lfloor False\rfloor\rfloor) \\
\quad | \; \lfloor\lfloor True\rfloor\rfloor \Rightarrow (case \; I[\![Y]\!] \; \tau \; of \\
\qquad \bot \Rightarrow \bot \\
\qquad | \; \lfloor\bot\rfloor \Rightarrow \lfloor\bot\rfloor \\
\qquad | \; \lfloor\lfloor y\rfloor\rfloor \Rightarrow \lfloor\lfloor y\rfloor\rfloor) \\
\quad | \; \lfloor\lfloor False\rfloor\rfloor \Rightarrow \lfloor\lfloor False \rfloor\rfloor)
\end{array}
$$

$\langle proof \rangle$

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

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

lemma *cp-OclAnd*:$(X \; and \; Y) \; \tau = ((\lambda \text{ -. } X \; \tau) \; and \; (\lambda \text{ -. } Y \; \tau)) \; \tau$
$\langle proof \rangle$

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

lemma *cp-OclImplies*:$(X \; implies \; Y) \; \tau = ((\lambda \text{ -. } X \; \tau) \; implies \; (\lambda \text{ -. } Y \; \tau)) \; \tau$
$\langle proof \rangle$

lemma *OclAnd1* [*simp*]: $(invalid \; and \; true) = invalid$
$\langle proof \rangle$
lemma *OclAnd2* [*simp*]: $(invalid \; and \; false) = false$
$\langle proof \rangle$
lemma *OclAnd3* [*simp*]: $(invalid \; and \; null) = invalid$
$\langle proof \rangle$
lemma *OclAnd4* [*simp*]: $(invalid \; and \; invalid) = invalid$
$\langle proof \rangle$

lemma *OclAnd5* [*simp*]: $(null \; and \; true) = null$

⟨*proof*⟩
lemma *OclAnd6* [*simp*]: (*null and false*) = *false*
⟨*proof*⟩
lemma *OclAnd7* [*simp*]: (*null and null*) = *null*
⟨*proof*⟩
lemma *OclAnd8* [*simp*]: (*null and invalid*) = *invalid*
⟨*proof*⟩

lemma *OclAnd9* [*simp*]: (*false and true*) = *false*
⟨*proof*⟩
lemma *OclAnd10* [*simp*]: (*false and false*) = *false*
⟨*proof*⟩
lemma *OclAnd11* [*simp*]: (*false and null*) = *false*
⟨*proof*⟩
lemma *OclAnd12* [*simp*]: (*false and invalid*) = *false*
⟨*proof*⟩

lemma *OclAnd13* [*simp*]: (*true and true*) = *true*
⟨*proof*⟩
lemma *OclAnd14* [*simp*]: (*true and false*) = *false*
⟨*proof*⟩
lemma *OclAnd15* [*simp*]: (*true and null*) = *null*
⟨*proof*⟩
lemma *OclAnd16* [*simp*]: (*true and invalid*) = *invalid*
⟨*proof*⟩

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

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

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

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

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

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

lemma *OclAnd-bot1* [*simp*]: $\bigwedge\tau.\ X\ \tau \neq \mathit{false}\ \tau \Longrightarrow (\mathit{bot}\ \mathit{and}\ X)\ \tau = \mathit{bot}\ \tau$
⟨*proof*⟩

lemma $OclAnd\text{-}bot2[simp]$: $\bigwedge\tau.\ X\ \tau \neq false\ \tau \Longrightarrow (X\ and\ bot)\ \tau = bot\ \tau$
⟨proof⟩

lemma $OclAnd\text{-}null1[simp]$: $\bigwedge\tau.\ X\ \tau \neq false\ \tau \Longrightarrow X\ \tau \neq bot\ \tau \Longrightarrow (null\ and\ X)\ \tau = null\ \tau$
⟨proof⟩

lemma $OclAnd\text{-}null2[simp]$: $\bigwedge\tau.\ X\ \tau \neq false\ \tau \Longrightarrow X\ \tau \neq bot\ \tau \Longrightarrow (X\ and\ null)\ \tau = null\ \tau$
⟨proof⟩

lemma $OclAnd\text{-}assoc$: $(X\ and\ (Y\ and\ Z)) = (X\ and\ Y\ and\ Z)$
⟨proof⟩

lemma $OclOr1[simp]$: $(invalid\ or\ true) = true$
⟨proof⟩
lemma $OclOr2[simp]$: $(invalid\ or\ false) = invalid$
⟨proof⟩
lemma $OclOr3[simp]$: $(invalid\ or\ null) = invalid$
⟨proof⟩
lemma $OclOr4[simp]$: $(invalid\ or\ invalid) = invalid$
⟨proof⟩

lemma $OclOr5[simp]$: $(null\ or\ true) = true$
⟨proof⟩
lemma $OclOr6[simp]$: $(null\ or\ false) = null$
⟨proof⟩
lemma $OclOr7[simp]$: $(null\ or\ null) = null$
⟨proof⟩
lemma $OclOr8[simp]$: $(null\ or\ invalid) = invalid$
⟨proof⟩

lemma $OclOr\text{-}idem[simp]$: $(X\ or\ X) = X$
⟨proof⟩

lemma $OclOr\text{-}commute$: $(X\ or\ Y) = (Y\ or\ X)$
⟨proof⟩

lemma $OclOr\text{-}false1[simp]$: $(false\ or\ Y) = Y$
⟨proof⟩

lemma $OclOr\text{-}false2[simp]$: $(Y\ or\ false) = Y$
⟨proof⟩

lemma $OclOr\text{-}true1[simp]$: $(true\ or\ Y) = true$
⟨proof⟩

lemma $OclOr\text{-}true2$: $(Y\ or\ true) = true$
⟨proof⟩

lemma *OclOr-bot1*[*simp*]: $\bigwedge\tau.\ X\ \tau \neq true\ \tau \Longrightarrow (bot\ or\ X)\ \tau = bot\ \tau$
⟨*proof*⟩

lemma *OclOr-bot2*[*simp*]: $\bigwedge\tau.\ X\ \tau \neq true\ \tau \Longrightarrow (X\ or\ bot)\ \tau = bot\ \tau$
⟨*proof*⟩

lemma *OclOr-null1*[*simp*]: $\bigwedge\tau.\ X\ \tau \neq true\ \tau \Longrightarrow X\ \tau \neq bot\ \tau \Longrightarrow (null\ or\ X)\ \tau = null\ \tau$
⟨*proof*⟩

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

lemma *OclOr-assoc*: $(X\ or\ (Y\ or\ Z)) = (X\ or\ Y\ or\ Z)$
⟨*proof*⟩

lemma *OclImplies-true*: $(X\ implies\ true) = true$
⟨*proof*⟩

lemma *deMorgan1*: $not(X\ and\ Y) = ((not\ X)\ or\ (not\ Y))$
⟨*proof*⟩

lemma *deMorgan2*: $not(X\ or\ Y) = ((not\ X)\ and\ (not\ Y))$
⟨*proof*⟩

5.1.5. A Standard Logical Calculus for OCL

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

Global vs. Local Judgements

lemma *transform1*: $P = true \Longrightarrow \tau \models P$
⟨*proof*⟩

lemma *transform1-rev*: $\forall\ \tau.\ \tau \models P \Longrightarrow P = true$
⟨*proof*⟩

lemma *transform2*: $(P = Q) \Longrightarrow ((\tau \models P) = (\tau \models Q))$
⟨*proof*⟩

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

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 $H : P = true \Longrightarrow Q = true$
shows $\tau \models P \Longrightarrow \tau \models Q$
⟨*proof*⟩

Local Validity and Meta-logic

lemma *foundation1*[*simp*]: $\tau \models true$
⟨*proof*⟩

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

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

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

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

lemma *defined-split*:
$(\tau \models \delta\ x) = ((\neg(\tau \models (x \triangleq invalid))) \wedge (\neg\ (\tau \models (x \triangleq null))))$
⟨*proof*⟩

lemma *valid-bool-split*: $(\tau \models \upsilon\ A) = ((\tau \models A \triangleq null) \vee (\tau \models A) \vee\ (\tau \models not\ A))$
⟨*proof*⟩

lemma *defined-bool-split*: $(\tau \models \delta\ A) = ((\tau \models A) \vee (\tau \models not\ A))$
⟨*proof*⟩

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

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

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

lemma *foundation7'*[*simp*]:
$(\tau \models not\ (\upsilon\ x)) = (\neg\ (\tau \models \upsilon\ x))$
⟨*proof*⟩

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*:
$(\tau \models \delta\ x) \lor (\tau \models (x \triangleq invalid)) \lor (\tau \models (x \triangleq null))$
$\langle proof \rangle$

lemma *foundation9*:
$\tau \models \delta\ x \Longrightarrow (\tau \models not\ x) = (\neg\ (\tau \models x))$
$\langle proof \rangle$

lemma *foundation9'*:
$\tau \models not\ x \Longrightarrow \neg\ (\tau \models x)$
$\langle proof \rangle$

lemma *foundation9''*:
$\tau \models not\ x \Longrightarrow \tau \models \delta\ x$
$\langle proof \rangle$

lemma *foundation10*:
$\tau \models \delta\ x \Longrightarrow \tau \models \delta\ y \Longrightarrow (\tau \models (x\ and\ y)) = (\ (\tau \models x) \land (\tau \models y))$
$\langle proof \rangle$

lemma *foundation10'*: $(\tau \models (A\ and\ B)) = ((\tau \models A) \land (\tau \models B))$
$\langle proof \rangle$

lemma *foundation11*:
$\tau \models \delta\ x \Longrightarrow\ \tau \models \delta\ y \Longrightarrow (\tau \models (x\ or\ y)) = (\ (\tau \models x) \lor (\tau \models y))$
$\langle proof \rangle$

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

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

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

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

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

lemma *foundation16''*: $\neg(\tau \models (\delta\ X)) = ((\tau \models (X \triangleq invalid)) \lor (\tau \models (X \triangleq null)))$

$\langle proof \rangle$

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

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

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

lemma *foundation18''*: $(\tau \models (\upsilon\ X)\ )=\ (\neg(\tau \models (X \triangleq invalid)))$
$\langle proof \rangle$

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

lemma *foundation21*: $(not\ A \triangleq not\ B) = (A \triangleq B)$
$\langle proof \rangle$

lemma *foundation22*: $(\tau \models (X \triangleq Y)) = (X\ \tau = Y\ \tau)$
$\langle proof \rangle$

lemma *foundation23*: $(\tau \models P) = (\tau \models (\lambda\ \text{-}\ .\ P\ \tau))$
$\langle proof \rangle$

lemma *foundation24*:$(\tau \models not(X \triangleq Y)) = (X\ \tau \neq Y\ \tau)$
$\langle proof \rangle$

lemma *foundation25*: $\tau \models P \Longrightarrow \tau \models (P\ or\ Q)$
$\langle proof \rangle$

lemma *foundation25'*: $\tau \models Q \Longrightarrow \tau \models (P\ or\ Q)$
$\langle proof \rangle$

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

shows R
⟨*proof*⟩

lemma *foundation27*: $(\tau \models (A \; and \; B)) = ((\tau \models A) \wedge (\tau \models B))$
⟨*proof*⟩

lemma *defined-not-I* : $\tau \models \delta \; (x) \Longrightarrow \tau \models \delta \; (not \; x)$
⟨*proof*⟩

lemma *valid-not-I* : $\tau \models \upsilon \; (x) \Longrightarrow \tau \models \upsilon \; (not \; x)$
⟨*proof*⟩

lemma *defined-and-I* : $\tau \models \delta \; (x) \Longrightarrow \tau \models \delta \; (y) \Longrightarrow \tau \models \delta \; (x \; and \; y)$
⟨*proof*⟩

lemma *valid-and-I* : $\tau \models \upsilon \; (x) \Longrightarrow \tau \models \upsilon \; (y) \Longrightarrow \tau \models \upsilon \; (x \; and \; y)$
⟨*proof*⟩

lemma *defined-or-I* : $\tau \models \delta \; (x) \Longrightarrow \tau \models \delta \; (y) \Longrightarrow \tau \models \delta \; (x \; or \; y)$
⟨*proof*⟩

lemma *valid-or-I* : $\tau \models \upsilon \; (x) \Longrightarrow \tau \models \upsilon \; (y) \Longrightarrow \tau \models \upsilon \; (x \; or \; y)$
⟨*proof*⟩

Local Judgements and Strong Equality

lemma *StrongEq-L-refl*: $\tau \models (x \triangleq x)$
⟨*proof*⟩

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

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

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

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

The rule of substitutivity in 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*: $\bigwedge \tau. \; cp \; P \Longrightarrow \tau \models (x \triangleq y) \Longrightarrow \tau \models (P \; x \triangleq P \; y)$
⟨*proof*⟩

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

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

lemma *StrongEq-L-subst3*:
assumes *cp*: $cp\ P$
and *eq*: $\tau \models (x \triangleq y)$
shows $(\tau \models P\ x) = (\tau \models P\ y)$
⟨*proof*⟩

lemma *StrongEq-L-subst3-rev*:
assumes *eq*: $\tau \models (x \triangleq y)$
and *cp*: $cp\ P$
shows $(\tau \models P\ x) = (\tau \models P\ y)$
⟨*proof*⟩

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*]
⟨*proof*⟩

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

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

lemma *cpI3*:
$(\forall\ X\ Y\ Z\ \tau.\ f\ X\ Y\ Z\ \tau = f(\lambda\text{-}.\ X\ \tau)(\lambda\text{-}.\ Y\ \tau)(\lambda\text{-}.\ Z\ \tau)\ \tau) \Longrightarrow$
$cp\ P \Longrightarrow cp\ Q \Longrightarrow cp\ R \Longrightarrow cp(\lambda X.\ f\ (P\ X)\ (Q\ X)\ (R\ X))$
⟨*proof*⟩

lemma *cpI4*:
$(\forall\ W\ X\ Y\ Z\ \tau.\ f\ W\ X\ Y\ Z\ \tau = f(\lambda\text{-}.\ W\ \tau)(\lambda\text{-}.\ X\ \tau)(\lambda\text{-}.\ Y\ \tau)(\lambda\text{-}.\ Z\ \tau)\ \tau) \Longrightarrow$
$cp\ P \Longrightarrow cp\ Q \Longrightarrow cp\ R \Longrightarrow cp\ S \Longrightarrow cp(\lambda X.\ f\ (P\ X)\ (Q\ X)\ (R\ X)\ (S\ X))$
⟨*proof*⟩

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

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

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]]

5.1.6. OCL's if then else endif

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

lemma *cp-OclIf*: $((if\ C\ then\ B_1\ else\ B_2\ endif)\ \tau =$
$(if\ (\lambda\ \text{-}.\ C\ \tau)\ then\ (\lambda\ \text{-}.\ B_1\ \tau)\ else\ (\lambda\ \text{-}.\ B_2\ \tau)\ endif)\ \tau)$
⟨*proof*⟩

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

lemma *OclIf-invalid* [*simp*]: $(if\ invalid\ then\ B_1\ else\ B_2\ endif) = invalid$
⟨*proof*⟩

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

lemma *OclIf-true* [*simp*]: $(if\ true\ then\ B_1\ else\ B_2\ endif) = B_1$
⟨*proof*⟩

lemma *OclIf-true'* [*simp*]: $\tau \models P \Longrightarrow (if\ P\ then\ B_1\ else\ B_2\ endif)\tau = B_1\ \tau$
⟨*proof*⟩

lemma *OclIf-true''* [*simp*]: $\tau \models P \Longrightarrow \tau \models (if\ P\ then\ B_1\ else\ B_2\ endif) \triangleq B_1$
⟨*proof*⟩

lemma *OclIf-false* [*simp*]: (*if false then* B_1 *else* B_2 *endif*) = B_2
⟨*proof*⟩

lemma *OclIf-false'* [*simp*]: $\tau \models$ *not* $P \Longrightarrow$ (*if* P *then* B_1 *else* B_2 *endif*)$\tau = B_2\ \tau$
⟨*proof*⟩

lemma *OclIf-idem1*[*simp*]:(*if* δ X *then* A *else* A *endif*) = A
⟨*proof*⟩

lemma *OclIf-idem2*[*simp*]:(*if* υ X *then* A *else* A *endif*) = A
⟨*proof*⟩

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

⟨*proof*⟩

5.1.7. Fundamental Predicates on Basic Types: Strict (Referential) Equality

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

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

with term "not" we can express the notation:

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

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

5.1.8. Laws to Establish Definedness (δ-closure)

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

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

lemma *OclNot-contrapos-nn*:
assumes A: $\tau \models \delta\ A$
assumes B: $\tau \models$ *not* B

assumes C: $\tau \models A \Longrightarrow \tau \models B$
shows $\tau \models$ *not* A
⟨*proof*⟩

5.1.9. A Side-calculus for Constant Terms

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

lemma *const-charn*: *const* $X \Longrightarrow X\ \tau = X\ \tau'$
⟨*proof*⟩

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*⟩

lemma *const-imply2* :
assumes $\bigwedge\tau\ \tau'.\ P\ \tau = P\ \tau' \Longrightarrow Q\ \tau = Q\ \tau'$
shows *const* $P \Longrightarrow$ *const* Q
⟨*proof*⟩

lemma *const-imply3* :
assumes $\bigwedge\tau\ \tau'.\ P\ \tau = P\ \tau' \Longrightarrow Q\ \tau = Q\ \tau' \Longrightarrow R\ \tau = R\ \tau'$
shows *const* $P \Longrightarrow$ *const* $Q \Longrightarrow$ *const* R
⟨*proof*⟩

lemma *const-imply4* :
assumes $\bigwedge\tau\ \tau'.\ P\ \tau = P\ \tau' \Longrightarrow Q\ \tau = Q\ \tau' \Longrightarrow R\ \tau = R\ \tau' \Longrightarrow S\ \tau = S\ \tau'$
shows *const* $P \Longrightarrow$ *const* $Q \Longrightarrow$ *const* $R \Longrightarrow$ *const* S
⟨*proof*⟩

lemma *const-lam* : *const* (λ-. e)
⟨*proof*⟩

lemma *const-true*[*simp*] : *const true*
⟨*proof*⟩

lemma *const-false*[*simp*] : *const false*
⟨*proof*⟩

lemma *const-null*[*simp*] : *const null*
⟨*proof*⟩

lemma *const-invalid* [*simp*]: *const invalid*
⟨*proof*⟩

lemma *const-bot*[*simp*] : *const bot*
⟨*proof*⟩

lemma *const-defined* :
assumes *const X*
shows *const* (δ *X*)
⟨*proof*⟩

lemma *const-valid* :
assumes *const X*
shows *const* (υ *X*)
⟨*proof*⟩

lemma *const-OclAnd* :
assumes *const X*
assumes *const X′*
shows *const* (*X and X′*)
⟨*proof*⟩

lemma *const-OclNot* :
assumes *const X*
shows *const* (*not X*)
⟨*proof*⟩

lemma *const-OclOr* :
assumes *const X*
assumes *const X′*
shows *const* (*X or X′*)
⟨*proof*⟩

lemma *const-OclImplies* :
assumes *const X*
assumes *const X′*
shows *const* (*X implies X′*)
⟨*proof*⟩

lemma *const-StrongEq*:
assumes *const X*
assumes *const X′*
shows *const*($X \triangleq X'$)
⟨*proof*⟩

lemma *const-OclIf* :
assumes *const B*
and *const C1*
and *const C2*
shows *const* (*if B then C1 else C2 endif*)
⟨*proof*⟩

lemma *const-OclValid1*:
assumes *const x*
shows $(\tau \models \delta\ x) = (\tau' \models \delta\ x)$
⟨*proof*⟩

lemma *const-OclValid2*:
assumes *const x*
shows $(\tau \models \upsilon\ x) = (\tau' \models \upsilon\ x)$
⟨*proof*⟩

lemma *const-HOL-if* : *const C* $\Longrightarrow$ *const D* $\Longrightarrow$ *const F* $\Longrightarrow$ *const* ($\lambda\tau$. *if C* τ *then D* τ *else F* τ)
⟨*proof*⟩
lemma *const-HOL-and*: *const C* $\Longrightarrow$ *const D* $\Longrightarrow$ *const* ($\lambda\tau$. $C\ \tau \wedge D\ \tau$)
⟨*proof*⟩
lemma *const-HOL-eq* : *const C* $\Longrightarrow$ *const D* $\Longrightarrow$ *const* ($\lambda\tau$. $C\ \tau = D\ \tau$)
⟨*proof*⟩

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

Miscellaneous: Overloading the syntax of "bottom"

notation *bot* (⊥)

end

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

5.2. 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.

5.2.1. mono

locale *profile-mono-scheme* =
  fixes f :: $('\mathfrak{A},'\alpha::null)val \Rightarrow ('\mathfrak{A},'\beta::null)val$
  fixes g
  assumes *def-scheme*: $(f\ x) \equiv \lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau\ then\ g\ (x\ \tau)\ else\ invalid\ \tau$

locale *profile-mono2* = *profile-mono-scheme* +
  assumes $\bigwedge x.\ x \neq bot \Longrightarrow x \neq null \Longrightarrow g\ x \neq bot$
begin
  lemma *strict*[*simp*,*code-unfold*]: $f\ invalid = invalid$
  ⟨*proof*⟩

  lemma *null-strict*[*simp*,*code-unfold*]: $f\ null = invalid$
  ⟨*proof*⟩

  lemma *cp0* : $f\ X\ \tau = f\ (\lambda\ \text{-}.\ X\ \tau)\ \tau$
  ⟨*proof*⟩

  lemma *cp*[*simp*,*code-unfold*] : $cp\ P \Longrightarrow cp\ (\lambda X.\ f\ (P\ X)\ )$
  ⟨*proof*⟩

  lemma *const*[*simp*,*code-unfold*] :
    assumes *C1* :$const\ X$
    shows $const(f\ X)$
  ⟨*proof*⟩
end

locale *profile-mono0* = *profile-mono-scheme* +
  assumes *def-body*: $\bigwedge x.\ x \neq bot \Longrightarrow x \neq null \Longrightarrow g\ x \neq bot \wedge g\ x \neq null$

sublocale *profile-mono0* < *profile-mono2*
⟨*proof*⟩

context *profile-mono0*
begin
  lemma *def-homo*[*simp*,*code-unfold*]: $\delta(f\ x) = (\delta\ x)$
  ⟨*proof*⟩

  lemma *def-valid-then-def*: $\upsilon(f\ x) = (\delta(f\ x))$
  ⟨*proof*⟩

end

5.2.2. single

locale *profile-single* =
  fixes d:: $('\mathfrak{A},'a::null)val \Rightarrow {}'\mathfrak{A}\ Boolean$
  assumes *d-strict*[*simp*,*code-unfold*]: $d\ invalid = false$
  assumes *d-cp0*: $d\ X\ \tau = d\ (\lambda\ \text{-}.\ X\ \tau)\ \tau$
  assumes *d-const*[*simp*,*code-unfold*]: $const\ X \Longrightarrow const\ (d\ X)$

5.2.3. bin

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

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 d_x:: $('\mathfrak{A},'a::null)val \Rightarrow {}'\mathfrak{A}\ Boolean$
  fixes d_y:: $('\mathfrak{A},'b::null)val \Rightarrow {}'\mathfrak{A}\ Boolean$
  fixes f::$('\mathfrak{A},'a::null)val \Rightarrow ('\mathfrak{A},'b::null)val \Rightarrow ('\mathfrak{A},'c::null)val$
  fixes g
  assumes $d_x{}'$: *profile-single* d_x
  assumes $d_y{}'$: *profile-single* d_y
  assumes d_x-d_y-*homo*[*simp*,*code-unfold*]: $cp\ (f\ X) \Longrightarrow$
$cp\ (\lambda x.\ f\ x\ Y) \Longrightarrow$
$f\ X\ invalid = invalid \Longrightarrow$
$f\ invalid\ Y = invalid \Longrightarrow$
$(\neg\ (\tau \models d_x\ X) \vee \neg\ (\tau \models d_y\ Y)) \Longrightarrow$
$\tau \models (\delta\ f\ X\ Y \triangleq (d_x\ X\ and\ d_y\ Y))$
  assumes *def-scheme″*[*simplified*]: $bin\ f\ g\ d_x\ d_y\ X\ Y$
  assumes *1*: $\tau \models d_x\ X \Longrightarrow \tau \models d_y\ Y \Longrightarrow \tau \models \delta\ f\ X\ Y$
begin
    interpretation d_x : *profile-single* d_x ⟨*proof*⟩
    interpretation d_y : *profile-single* d_y ⟨*proof*⟩

    lemma *strict1*[*simp*,*code-unfold*]: $f\ invalid\ y = invalid$
    ⟨*proof*⟩

    lemma *strict2*[*simp*,*code-unfold*]: $f\ x\ invalid = invalid$
    ⟨*proof*⟩

    lemma *cp0* : $f\ X\ Y\ \tau = f\ (\lambda\ \text{-}.\ X\ \tau)\ (\lambda\ \text{-}.\ Y\ \tau)\ \tau$
    ⟨*proof*⟩

    lemma *cp*[*simp*,*code-unfold*] : $cp\ P \Longrightarrow cp\ Q \Longrightarrow cp\ (\lambda X.\ f\ (P\ X)\ (Q\ X))$

⟨*proof*⟩

lemma *def-homo*[*simp*,*code-unfold*]: $\delta(f\ x\ y) = (d_x\ x\ and\ d_y\ y)$
⟨*proof*⟩

lemma *def-valid-then-def*: $\upsilon(f\ x\ y) = (\delta(f\ x\ y))$
⟨*proof*⟩

lemma *defined-args-valid*: $(\tau \models \delta\ (f\ x\ y)) = ((\tau \models d_x\ x) \wedge (\tau \models d_y\ y))$
⟨*proof*⟩

lemma *const*[*simp*,*code-unfold*] :
assumes *C1* :*const X* **and** *C2* : *const Y*
shows *const*(*f X Y*)
⟨*proof*⟩
end

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

locale *profile-bin-scheme-defined* =
fixes d_y:: ($'\mathfrak{A}$,$'b$::*null*)*val* ⇒ $'\mathfrak{A}$ *Boolean*
fixes *f*::($'\mathfrak{A}$,$'a$::*null*)*val* ⇒ ($'\mathfrak{A}$,$'b$::*null*)*val* ⇒ ($'\mathfrak{A}$,$'c$::*null*)*val*
fixes *g*
assumes d_y : *profile-single* d_y
assumes d_y-*homo*[*simp*,*code-unfold*]: $cp\ (f\ X) \Longrightarrow$
$f\ X\ invalid = invalid \Longrightarrow$
$\neg\ \tau \models d_y\ Y \Longrightarrow$
$\tau \models \delta\ f\ X\ Y \triangleq (\delta\ X\ and\ d_y\ Y)$
assumes *def-scheme'*[*simplified*]: *bin f g defined* d_y *X Y*
assumes *def-body'*: $\bigwedge\ x\ y\ \tau.\ x{\neq}bot \Longrightarrow x{\neq}null \Longrightarrow (d_y\ y)\ \tau = true\ \tau \Longrightarrow g\ x\ (y\ \tau) \neq bot \wedge g\ x\ (y\ \tau) \neq null$
begin
lemma *strict3*[*simp*,*code-unfold*]: *f null y* = *invalid*
⟨*proof*⟩
end

sublocale *profile-bin-scheme-defined* < *profile-bin-scheme defined*
⟨*proof*⟩

locale *profile-bin1* =
fixes *f*::($'\mathfrak{A}$,$'a$::*null*)*val* ⇒ ($'\mathfrak{A}$,$'b$::*null*)*val* ⇒ ($'\mathfrak{A}$,$'c$::*null*)*val*
fixes *g*
assumes *def-scheme*[*simplified*]: *bin f g defined defined X Y*
assumes *def-body*: $\bigwedge\ x\ y.\ g\ x\ y \neq bot \wedge g\ x\ y \neq null$
begin
lemma *strict4*[*simp*,*code-unfold*]: *f x null* = *invalid*

⟨*proof*⟩
end

sublocale *profile-bin1* < *profile-bin-scheme-defined defined*
⟨*proof*⟩

locale *profile-bin2* =
fixes f::($'\mathfrak{A}$,$'a$::*null*)*val* ⇒ ($'\mathfrak{A}$,$'b$::*null*)*val* ⇒ ($'\mathfrak{A}$,$'c$::*null*)*val*
fixes g
assumes *def-scheme*[*simplified*]: *bin f g defined valid X Y*
assumes *def-body*: $\bigwedge$ x y. $x{\neq}bot \Longrightarrow x{\neq}null \Longrightarrow y{\neq}bot \Longrightarrow g\ x\ y \neq bot \wedge g\ x\ y \neq null$

sublocale *profile-bin2* < *profile-bin-scheme-defined valid*
⟨*proof*⟩

locale *profile-bin3* =
fixes f :: ($'\mathfrak{A}$,$'\alpha$::*null*)*val* ⇒ ($'\mathfrak{A}$,$'\alpha$::*null*)*val* ⇒ ($'\mathfrak{A}$) *Boolean*
assumes *def-scheme*[*simplified*]: *bin′ f StrongEq valid valid X Y*

sublocale *profile-bin3* < *profile-bin-scheme valid valid f* $\lambda x\ y.\ \lfloor\lfloor x = y\rfloor\rfloor$
⟨*proof*⟩

context *profile-bin3*
begin
lemma *idem*[*simp*,*code-unfold*]: *f null null* = *true*
⟨*proof*⟩

lemma *defargs*: $\tau \models f\ x\ y \Longrightarrow (\tau \models \upsilon\ x) \wedge (\tau \models \upsilon\ y)$
⟨*proof*⟩

lemma *defined-args-valid′* : $\delta\ (f\ x\ y) = (\upsilon\ x\ and\ \upsilon\ y)$
⟨*proof*⟩

lemma *refl-ext*[*simp*,*code-unfold*] : $(f\ x\ x) = (if\ (\upsilon\ x)\ then\ true\ else\ invalid\ endif)$
⟨*proof*⟩

lemma *sym* : $\tau \models (f\ x\ y) \Longrightarrow \tau \models (f\ y\ x)$
⟨*proof*⟩

lemma *symmetric* : $(f\ x\ y) = (f\ y\ x)$
⟨*proof*⟩

lemma *trans* : $\tau \models (f\ x\ y) \Longrightarrow \tau \models (f\ y\ z) \Longrightarrow \tau \models (f\ x\ z)$
⟨*proof*⟩

lemma *StrictRefEq-vs-StrongEq*: $\tau \models(\upsilon\ x) \Longrightarrow \tau \models(\upsilon\ y) \Longrightarrow (\tau \models ((f\ x\ y) \triangleq (x \triangleq y)))$
⟨*proof*⟩

end

locale *profile-bin4* =
  fixes f :: $('\mathfrak{A},'\alpha::null)val \Rightarrow ('\mathfrak{A},'\beta::null)val \Rightarrow ('\mathfrak{A},'\gamma::null)val$
  fixes g
  assumes *def-scheme*[*simplified*]: *bin f g valid valid X Y*
  assumes *def-body*: $\bigwedge x\ y.\ x{\neq}bot \Longrightarrow y{\neq}bot \Longrightarrow g\ x\ y \neq bot \wedge g\ x\ y \neq null$

sublocale *profile-bin4* < *profile-bin-scheme valid valid*
⟨*proof*⟩

end

theory *OCL-basic-type-Boolean*
imports *OCL-lib-common*
begin

5.2.4. Fundamental Predicates on Basic Types: Strict (Referential) Equality

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

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

which implies elementary properties like:

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

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

lemma *null-non-true* [*simp*,*code-unfold*]:$(null \doteq true) = false$
⟨*proof*⟩

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

lemma *true-non-null* [*simp*,*code-unfold*]:$(true \doteq null) = false$
⟨*proof*⟩

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

interpretation $StrictRefEq_{Boolean}$: *profile-bin3* $\lambda\ x\ y.\ (x{::}('\mathfrak{A})Boolean) \doteq y$
⟨*proof*⟩

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

lemma $(invalid \doteq false) = invalid$ ⟨*proof*⟩
lemma $(invalid \doteq true) = invalid$ ⟨*proof*⟩
lemma $(false \doteq invalid) = invalid$ ⟨*proof*⟩
lemma $(true \doteq invalid) = invalid$ ⟨*proof*⟩
lemma $((invalid{::}('\mathfrak{A})Boolean) \doteq invalid) = invalid$ ⟨*proof*⟩

Thus, the weak equality is *not* reflexive.

5.2.5. Test Statements on Boolean Operations.

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

Elementary computations on Boolean

Assert $\tau \models \upsilon(true)$
Assert $\tau \models \delta(false)$
Assert $\neg(\tau \models \delta(null))$
Assert $\neg(\tau \models \delta(invalid))$
Assert $\tau \models \upsilon((null{::}('\mathfrak{A})Boolean))$
Assert $\neg(\tau \models \upsilon(invalid))$
Assert $\tau \models (true\ and\ true)$
Assert $\tau \models (true\ and\ true \triangleq true)$
Assert $\tau \models ((null\ or\ null) \triangleq null)$
Assert $\tau \models ((null\ or\ null) \doteq null)$
Assert $\tau \models ((true \triangleq false) \triangleq false)$
Assert $\tau \models ((invalid \triangleq false) \triangleq false)$
Assert $\tau \models ((invalid \doteq false) \triangleq invalid)$
Assert $\tau \models (true <> false)$
Assert $\tau \models (false <> true)$

end

theory *OCL-basic-type-Void*
imports *OCL-basic-type-Boolean*
begin

5.3. 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.

5.3.1. Fundamental Properties on Basic Types: Strict Equality

Definition

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

  instance ⟨*proof*⟩
end

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

  instance ⟨*proof*⟩
end

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

defs $StrictRefEq_{Void}$[*code-unfold*] :
$$(x::('\mathfrak{A})\,Void) \doteq y \equiv \lambda\,\tau.\ if\ (\upsilon\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau\ then\ (x \triangleq y)\ \tau\ else\ invalid\ \tau$$

Property proof in terms of *profile-bin3*

interpretation $StrictRefEq_{Void}$: *profile-bin3* $\lambda\ x\ y.\ (x::('\mathfrak{A})\,Void) \doteq y$
  ⟨*proof*⟩

5.3.2. Test Statements

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

end

theory *OCL-basic-type-Integer*
imports *OCL-basic-type-Boolean*
begin

5.4. Basic Type Integer: Operations

5.4.1. Basic Integer Constants

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

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

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

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

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

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

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

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

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

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

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

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

5.4.2. Validity and Definedness Properties

lemma δ(*null*::($'\mathfrak{A}$)*Integer*) = *false* ⟨*proof*⟩
lemma υ(*null*::($'\mathfrak{A}$)*Integer*) = *true* ⟨*proof*⟩

lemma [*simp*,*code-unfold*]: δ (λ-. $\lfloor\lfloor n\rfloor\rfloor$) = *true*
⟨*proof*⟩

lemma [*simp*,*code-unfold*]: υ (λ-. $\lfloor\lfloor n\rfloor\rfloor$) = *true*
⟨*proof*⟩

lemma [*simp*,*code-unfold*]: δ **0** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: υ **0** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: δ **1** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: υ **1** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: δ **2** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: υ **2** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: δ **6** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: υ **6** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: δ **8** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: υ **8** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: δ **9** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: υ **9** = *true* ⟨*proof*⟩

5.4.3. 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 $OclAdd_{Integer}$ ::($'\mathfrak{A}$)*Integer* ⇒ ($'\mathfrak{A}$)*Integer* ⇒ ($'\mathfrak{A}$)*Integer* (**infix** $+_{int}$ *40*)
where $x +_{int} y \equiv \lambda\ \tau.\ \textit{if}\ (\delta\ x)\ \tau = \textit{true}\ \tau \wedge (\delta\ y)\ \tau = \textit{true}\ \tau$
$\textit{then}\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil + \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\textit{else invalid}\ \tau$
interpretation $OclAdd_{Integer}$: *profile-bin1 op* $+_{int}$ $\lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil + \lceil\lceil y\rceil\rceil\rfloor\rfloor$
⟨*proof*⟩

definition $OclMinus_{Integer}$ ::($'\mathfrak{A}$)*Integer* ⇒ ($'\mathfrak{A}$)*Integer* ⇒ ($'\mathfrak{A}$)*Integer* (**infix** $-_{int}$ *41*)
where $x -_{int} y \equiv \lambda\ \tau.\ \textit{if}\ (\delta\ x)\ \tau = \textit{true}\ \tau \wedge (\delta\ y)\ \tau = \textit{true}\ \tau$
$\textit{then}\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil - \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\textit{else invalid}\ \tau$
interpretation $OclMinus_{Integer}$: *profile-bin1 op* $-_{int}$ $\lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil - \lceil\lceil y\rceil\rceil\rfloor\rfloor$
⟨*proof*⟩

definition $OclMult_{Integer}$ ::($'\mathfrak{A}$)*Integer* ⇒ ($'\mathfrak{A}$)*Integer* ⇒ ($'\mathfrak{A}$)*Integer* (**infix** $*_{int}$ *45*)
where $x *_{int} y \equiv \lambda\ \tau.\ \textit{if}\ (\delta\ x)\ \tau = \textit{true}\ \tau \wedge (\delta\ y)\ \tau = \textit{true}\ \tau$
$\textit{then}\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil * \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\textit{else invalid}\ \tau$
interpretation $OclMult_{Integer}$: *profile-bin1 op* $*_{int}$ $\lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil * \lceil\lceil y\rceil\rceil\rfloor\rfloor$
⟨*proof*⟩

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

definition $OclDivision_{Integer}$ ::($'\mathfrak{A}$)*Integer* ⇒ ($'\mathfrak{A}$)*Integer* ⇒ ($'\mathfrak{A}$)*Integer* (**infix** div_{int} *45*)

where $x \, div_{int} \, y \equiv \lambda \, \tau. \, if \, (\delta \, x) \, \tau = true \, \tau \wedge (\delta \, y) \, \tau = true \, \tau$
$\quad then \, if \, y \, \tau \neq OclInt0 \, \tau \, then \, \lfloor\lfloor\lceil\lceil x \, \tau\rceil\rceil \, div \, \lceil\lceil y \, \tau\rceil\rceil\rfloor\rfloor \, else \, invalid \, \tau$
$\quad else \, invalid \, \tau$

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

definition $OclLess_{Integer}$ ::$('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Boolean$ (**infix** $<_{int}$ *35*)
where $x <_{int} y \equiv \lambda \, \tau. \, if \, (\delta \, x) \, \tau = true \, \tau \wedge (\delta \, y) \, \tau = true \, \tau$
$\quad then \, \lfloor\lfloor\lceil\lceil x \, \tau\rceil\rceil < \lceil\lceil y \, \tau\rceil\rceil\rfloor\rfloor$
$\quad else \, invalid \, \tau$
interpretation $OclLess_{Integer}$: *profile-bin1 op* $<_{int} \lambda \, x \, y. \, \lfloor\lfloor\lceil\lceil x\rceil\rceil < \lceil\lceil y\rceil\rceil\rfloor\rfloor$
⟨*proof*⟩

definition $OclLe_{Integer}$ ::$('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Boolean$ (**infix** $\leq_{int}$ *35*)
where $x \leq_{int} y \equiv \lambda \, \tau. \, if \, (\delta \, x) \, \tau = true \, \tau \wedge (\delta \, y) \, \tau = true \, \tau$
$\quad then \, \lfloor\lfloor\lceil\lceil x \, \tau\rceil\rceil \leq \lceil\lceil y \, \tau\rceil\rceil\rfloor\rfloor$
$\quad else \, invalid \, \tau$
interpretation $OclLe_{Integer}$: *profile-bin1 op* $\leq_{int} \lambda \, x \, y. \, \lfloor\lfloor\lceil\lceil x\rceil\rceil \leq \lceil\lceil y\rceil\rceil\rfloor\rfloor$
⟨*proof*⟩

Basic Properties

lemma $OclAdd_{Integer}$*-commute*: $(X +_{int} Y) = (Y +_{int} X)$
⟨*proof*⟩

Execution with Invalid or Null or Zero as Argument

lemma $OclAdd_{Integer}$*-zero1*[*simp*,*code-unfold*] :
$(x +_{int} \mathbf{0}) = (if \, \upsilon \, x \, and \, not \, (\delta \, x) \, then \, invalid \, else \, x \, endif)$
⟨*proof*⟩

lemma $OclAdd_{Integer}$*-zero2*[*simp*,*code-unfold*] :
$(\mathbf{0} +_{int} x) = (if \, \upsilon \, x \, and \, not \, (\delta \, x) \, then \, invalid \, else \, x \, endif)$
⟨*proof*⟩

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 $\tau \models (\mathbf{9} \leq_{int} \mathbf{10})$
Assert $\tau \models ((\mathbf{4} +_{int} \mathbf{4}) \leq_{int} \mathbf{10})$
Assert $\neg(\tau \models ((\mathbf{4} +_{int} (\mathbf{4} +_{int} \mathbf{4})) <_{int} \mathbf{10}))$
Assert $\tau \models not \, (\upsilon \, (null +_{int} \mathbf{1}))$

Assert $\tau \models (((\mathbf{9} *_{int} \mathbf{4})\ div_{int}\ \mathbf{10}) \leq_{int}\ \mathbf{4})$
Assert $\tau \models not\ (\delta\ (\mathbf{1}\ div_{int}\ \mathbf{0}))$
Assert $\tau \models not\ (\upsilon\ (\mathbf{1}\ div_{int}\ \mathbf{0}))$

5.4.4. Fundamental Predicates on Integers: Strict Equality

Definition

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

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

Property proof in terms of *profile-bin3*

interpretation $StrictRefEq_{Integer}$: *profile-bin3* $\lambda\ x\ y.\ (x::('\mathfrak{A})Integer) \doteq y$
⟨*proof*⟩

lemma *integer-non-null* [*simp*]: $((\lambda\text{-}.\ \lfloor\lfloor n\rfloor\rfloor) \doteq (null::('\mathfrak{A})Integer)) = false$
⟨*proof*⟩

lemma *null-non-integer* [*simp*]: $((null::('\mathfrak{A})Integer) \doteq (\lambda\text{-}.\ \lfloor\lfloor n\rfloor\rfloor)) = false$
⟨*proof*⟩

lemma *OclInt0-non-null* [*simp*,*code-unfold*]: $(\mathbf{0} \doteq null) = false$ ⟨*proof*⟩
lemma *null-non-OclInt0* [*simp*,*code-unfold*]: $(null \doteq \mathbf{0}) = false$ ⟨*proof*⟩
lemma *OclInt1-non-null* [*simp*,*code-unfold*]: $(\mathbf{1} \doteq null) = false$ ⟨*proof*⟩
lemma *null-non-OclInt1* [*simp*,*code-unfold*]: $(null \doteq \mathbf{1}) = false$ ⟨*proof*⟩
lemma *OclInt2-non-null* [*simp*,*code-unfold*]: $(\mathbf{2} \doteq null) = false$ ⟨*proof*⟩
lemma *null-non-OclInt2* [*simp*,*code-unfold*]: $(null \doteq \mathbf{2}) = false$ ⟨*proof*⟩
lemma *OclInt6-non-null* [*simp*,*code-unfold*]: $(\mathbf{6} \doteq null) = false$ ⟨*proof*⟩
lemma *null-non-OclInt6* [*simp*,*code-unfold*]: $(null \doteq \mathbf{6}) = false$ ⟨*proof*⟩
lemma *OclInt8-non-null* [*simp*,*code-unfold*]: $(\mathbf{8} \doteq null) = false$ ⟨*proof*⟩
lemma *null-non-OclInt8* [*simp*,*code-unfold*]: $(null \doteq \mathbf{8}) = false$ ⟨*proof*⟩
lemma *OclInt9-non-null* [*simp*,*code-unfold*]: $(\mathbf{9} \doteq null) = false$ ⟨*proof*⟩
lemma *null-non-OclInt9* [*simp*,*code-unfold*]: $(null \doteq \mathbf{9}) = false$ ⟨*proof*⟩

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

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

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

Assert $\tau \models \upsilon\ \mathbf{4}$
Assert $\tau \models \delta\ \mathbf{4}$
Assert $\tau \models \upsilon\ (null::('\mathfrak{A})Integer)$
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 $\neg(\tau \models (invalid \doteq (invalid::('\mathfrak{A})Integer)))$
Assert $\neg(\tau \models \upsilon\ (invalid \doteq (invalid::('\mathfrak{A})Integer)))$
Assert $\neg(\tau \models (invalid <> (invalid::('\mathfrak{A})Integer)))$
Assert $\neg(\tau \models \upsilon\ (invalid <> (invalid::('\mathfrak{A})Integer)))$
Assert $\tau \models (null \doteq (null::('\mathfrak{A})Integer)\ )$
Assert $\tau \models (null \doteq (null::('\mathfrak{A})Integer)\ )$
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 *OCL-basic-type-Real*
imports *OCL-basic-type-Boolean*
begin

type-synonym *real* = *nat*

5.5. Basic Type Real: Operations

5.5.1. Basic Real Constants

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

definition $OclReal0 ::('\mathfrak{A})Real\ (\mathbf{0.0})$
where $\mathbf{0.0} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 0::real \rfloor\rfloor)$

definition $OclReal1 ::('\mathfrak{A})Real\ (\mathbf{1.0})$
where $\mathbf{1.0} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 1::real \rfloor\rfloor)$

definition *OclReal2* ::('$\mathfrak{A}$)*Real* (**2.0**)
where **2.0** = (λ - . ⌊⌊*2*::*real*⌋⌋)

definition *OclReal3* ::('$\mathfrak{A}$)*Real* (**3.0**)
where **3.0** = (λ - . ⌊⌊*3*::*real*⌋⌋)

definition *OclReal4* ::('$\mathfrak{A}$)*Real* (**4.0**)
where **4.0** = (λ - . ⌊⌊*4*::*real*⌋⌋)

definition *OclReal5* ::('$\mathfrak{A}$)*Real* (**5.0**)
where **5.0** = (λ - . ⌊⌊*5*::*real*⌋⌋)

definition *OclReal6* ::('$\mathfrak{A}$)*Real* (**6.0**)
where **6.0** = (λ - . ⌊⌊*6*::*real*⌋⌋)

definition *OclReal7* ::('$\mathfrak{A}$)*Real* (**7.0**)
where **7.0** = (λ - . ⌊⌊*7*::*real*⌋⌋)

definition *OclReal8* ::('$\mathfrak{A}$)*Real* (**8.0**)
where **8.0** = (λ - . ⌊⌊*8*::*real*⌋⌋)

definition *OclReal9* ::('$\mathfrak{A}$)*Real* (**9.0**)
where **9.0** = (λ - . ⌊⌊*9*::*real*⌋⌋)

definition *OclReal10* ::('$\mathfrak{A}$)*Real* (**10.0**)
where **10.0** = (λ - . ⌊⌊*10*::*real*⌋⌋)

term *pi*

5.5.2. Validity and Definedness Properties

lemma δ(*null*::('$\mathfrak{A}$)*Real*) = *false* ⟨*proof*⟩
lemma υ(*null*::('$\mathfrak{A}$)*Real*) = *true* ⟨*proof*⟩

lemma [*simp*,*code-unfold*]: δ (λ-. ⌊⌊*n*⌋⌋) = *true*
⟨*proof*⟩

lemma [*simp*,*code-unfold*]: υ (λ-. ⌊⌊*n*⌋⌋) = *true*
⟨*proof*⟩

lemma [*simp*,*code-unfold*]: δ **0.0** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: υ **0.0** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: δ **1.0** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: υ **1.0** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: δ **2.0** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: υ **2.0** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: δ **6.0** = *true* ⟨*proof*⟩

lemma [*simp*,*code-unfold*]: υ **6.0** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: δ **8.0** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: υ **8.0** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: δ **9.0** = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: υ **9.0** = *true* ⟨*proof*⟩

5.5.3. 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 $OclAdd_{Real}$ ::$('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real$ (**infix** $+_{real}$ *40*)
where $x +_{real} y \equiv \lambda\, \tau.$ *if* $(\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
then $\lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil + \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
else invalid τ
interpretation $OclAdd_{Real}$: *profile-bin1 op* $+_{real}$ $\lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil + \lceil\lceil y\rceil\rceil\rfloor\rfloor$
⟨*proof*⟩

definition $OclMinus_{Real}$ ::$('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real$ (**infix** $-_{real}$ *41*)
where $x -_{real} y \equiv \lambda\, \tau.$ *if* $(\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
then $\lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil - \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
else invalid τ
interpretation $OclMinus_{Real}$: *profile-bin1 op* $-_{real}$ $\lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil - \lceil\lceil y\rceil\rceil\rfloor\rfloor$
⟨*proof*⟩

definition $OclMult_{Real}$ ::$('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real$ (**infix** $*_{real}$ *45*)
where $x *_{real} y \equiv \lambda\, \tau.$ *if* $(\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
then $\lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil * \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
else invalid τ
interpretation $OclMult_{Real}$: *profile-bin1 op* $*_{real}$ $\lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil * \lceil\lceil y\rceil\rceil\rfloor\rfloor$
⟨*proof*⟩

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

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

definition $OclModulus_{Real}$ ::$('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real \Rightarrow ('\mathfrak{A})Real$ (**infix** mod_{real} *45*)
where $x\ mod_{real}\ y \equiv \lambda\, \tau.$ *if* $(\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
then if $y\ \tau \neq OclReal0\ \tau$ *then* $\lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil\ mod\ \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$ *else invalid* τ

$else\ invalid\ \tau$

definition $OclLess_{Real}$ ::($'\mathfrak{A}$)$Real \Rightarrow$ ($'\mathfrak{A}$)$Real \Rightarrow$ ($'\mathfrak{A}$)$Boolean$ (**infix** $<_{real}$ *35*)
where $x <_{real} y \equiv \lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$then\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil < \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$else\ invalid\ \tau$
interpretation $OclLess_{Real}$: *profile-bin1 op* $<_{real}\ \lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil < \lceil\lceil y\rceil\rceil\rfloor\rfloor$
⟨*proof*⟩

definition $OclLe_{Real}$ ::($'\mathfrak{A}$)$Real \Rightarrow$ ($'\mathfrak{A}$)$Real \Rightarrow$ ($'\mathfrak{A}$)$Boolean$ (**infix** $\leq_{real}$ *35*)
where $x \leq_{real} y \equiv \lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$then\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil \leq \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$else\ invalid\ \tau$
interpretation $OclLe_{Real}$: *profile-bin1 op* $\leq_{real}\ \lambda\ x\ y.\ \lfloor\lfloor\lceil\lceil x\rceil\rceil \leq \lceil\lceil y\rceil\rceil\rfloor\rfloor$
⟨*proof*⟩

Basic Properties

lemma $OclAdd_{Real}$*-commute*: $(X +_{real} Y) = (Y +_{real} X)$
⟨*proof*⟩

Execution with Invalid or Null or Zero as Argument

lemma $OclAdd_{Real}$*-zero1*[*simp*,*code-unfold*] :
$(x +_{real} \mathbf{0.0}) = (if\ \upsilon\ x\ and\ not\ (\delta\ x)\ then\ invalid\ else\ x\ endif)$
⟨*proof*⟩

lemma $OclAdd_{Real}$*-zero2*[*simp*,*code-unfold*] :
$(\mathbf{0.0} +_{real} x) = (if\ \upsilon\ x\ and\ not\ (\delta\ x)\ then\ invalid\ else\ x\ endif)$
⟨*proof*⟩

Test Statements

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

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

5.5.4. Fundamental Predicates on Reals: Strict Equality

Definition

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

defs $StrictRefEq_{Real}$ [*code-unfold*] :
$$(x{::}('\mathfrak{A})Real) \doteq y \equiv \lambda\ \tau.\ if\ (\upsilon\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau\ then\ (x \triangleq y)\ \tau\ else\ invalid\ \tau$$

Property proof in terms of *profile-bin3*

interpretation $StrictRefEq_{Real}$: *profile-bin3* $\lambda\ x\ y.\ (x{::}('\mathfrak{A})Real) \doteq y$
⟨*proof*⟩

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

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

lemma *OclReal0-non-null* [*simp*,*code-unfold*]: $(\mathbf{0.0} \doteq null) = false$ ⟨*proof*⟩
lemma *null-non-OclReal0* [*simp*,*code-unfold*]: $(null \doteq \mathbf{0.0}) = false$ ⟨*proof*⟩
lemma *OclReal1-non-null* [*simp*,*code-unfold*]: $(\mathbf{1.0} \doteq null) = false$ ⟨*proof*⟩
lemma *null-non-OclReal1* [*simp*,*code-unfold*]: $(null \doteq \mathbf{1.0}) = false$ ⟨*proof*⟩
lemma *OclReal2-non-null* [*simp*,*code-unfold*]: $(\mathbf{2.0} \doteq null) = false$ ⟨*proof*⟩
lemma *null-non-OclReal2* [*simp*,*code-unfold*]: $(null \doteq \mathbf{2.0}) = false$ ⟨*proof*⟩
lemma *OclReal6-non-null* [*simp*,*code-unfold*]: $(\mathbf{6.0} \doteq null) = false$ ⟨*proof*⟩
lemma *null-non-OclReal6* [*simp*,*code-unfold*]: $(null \doteq \mathbf{6.0}) = false$ ⟨*proof*⟩
lemma *OclReal8-non-null* [*simp*,*code-unfold*]: $(\mathbf{8.0} \doteq null) = false$ ⟨*proof*⟩
lemma *null-non-OclReal8* [*simp*,*code-unfold*]: $(null \doteq \mathbf{8.0}) = false$ ⟨*proof*⟩
lemma *OclReal9-non-null* [*simp*,*code-unfold*]: $(\mathbf{9.0} \doteq null) = false$ ⟨*proof*⟩
lemma *null-non-OclReal9* [*simp*,*code-unfold*]: $(null \doteq \mathbf{9.0}) = false$ ⟨*proof*⟩

Const

lemma [*simp*,*code-unfold*]: *const*($\mathbf{0.0}$) ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: *const*($\mathbf{1.0}$) ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: *const*($\mathbf{2.0}$) ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: *const*($\mathbf{6.0}$) ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: *const*($\mathbf{8.0}$) ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: *const*($\mathbf{9.0}$) ⟨*proof*⟩

5.5.5. Test Statements on Basic Real

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

Elementary computations on Real

Assert $\tau \models \mathbf{1.0} <> \mathbf{2.0}$
Assert $\tau \models \mathbf{2.0} <> \mathbf{1.0}$
Assert $\tau \models \mathbf{2.0} \doteq \mathbf{2.0}$

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

end

theory *OCL-basic-type-String*
imports *OCL-basic-type-Boolean*
begin

5.6. Basic Type String: Operations

5.6.1. Basic String Constants

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

definition $OclStringa$ $::('\mathfrak{A})String$ (a)
where a = $(\lambda$ - . $\lfloor\lfloor''a''\rfloor\rfloor)$

definition $OclStringb$ $::('\mathfrak{A})String$ (b)
where b = $(\lambda$ - . $\lfloor\lfloor''b''\rfloor\rfloor)$

definition *OclStringc* ::($'\mathfrak{A}$)*String* (c)
where c = (λ - . $\lfloor\lfloor''c''\rfloor\rfloor$)

5.6.2. Validity and Definedness Properties

lemma $\delta(null::('\mathfrak{A})String) = false$ ⟨*proof*⟩
lemma $\upsilon(null::('\mathfrak{A})String) = true$ ⟨*proof*⟩

lemma [*simp*,*code-unfold*]: δ (λ-. $\lfloor\lfloor n\rfloor\rfloor$) = *true*
⟨*proof*⟩

lemma [*simp*,*code-unfold*]: υ (λ-. $\lfloor\lfloor n\rfloor\rfloor$) = *true*
⟨*proof*⟩

lemma [*simp*,*code-unfold*]: δ a = *true* ⟨*proof*⟩
lemma [*simp*,*code-unfold*]: υ a = *true* ⟨*proof*⟩

5.6.3. 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 $OclAdd_{String}$ ::($'\mathfrak{A}$)*String* $\Rightarrow$ ($'\mathfrak{A}$)*String* $\Rightarrow$ ($'\mathfrak{A}$)*String* (**infix** $+_{string}$ *40*)
where $x +_{string} y \equiv \lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$then\ \lfloor\lfloor concat\ [\lceil\lceil x\ \tau\rceil\rceil,\ \lceil\lceil y\ \tau\rceil\rceil]\rfloor\rfloor$
$else\ invalid\ \tau$
interpretation $OclAdd_{String}$: *profile-bin1 op* $+_{string}$ $\lambda\ x\ y.\ \lfloor\lfloor concat\ [\lceil\lceil x\rceil\rceil,\ \lceil\lceil y\rceil\rceil]\rfloor\rfloor$
⟨*proof*⟩

Basic Properties

lemma $OclAdd_{String}$-*not-commute*: $\exists\, X\ Y.\ (X +_{string} Y) \neq (Y +_{string} X)$
⟨*proof*⟩

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*.

5.6.4. 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 $'\mathfrak{A}$ *Boolean*-case as strict extension of the strong equality:

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

Property proof in terms of *profile-bin3*

interpretation $\mathit{StrictRefEq}_{String}$: $\mathit{profile\text{-}bin3}\ \lambda\ x\ y.\ (x::('\mathfrak{A})\mathit{String}) \doteq y$
⟨*proof*⟩

5.6.5. 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 $\tau \models \upsilon$ a
Assert $\tau \models \delta$ a
Assert $\tau \models \upsilon\ (\mathit{null}::('\mathfrak{A})\mathit{String})$
Assert $\tau \models (\mathit{invalid} \triangleq \mathit{invalid})$
Assert $\tau \models (\mathit{null} \triangleq \mathit{null})$
Assert $\tau \models$ (a $\triangleq$ a)
Assert $\neg(\tau \models$ (a $\triangleq$ b))
Assert $\neg(\tau \models (\mathit{invalid} \triangleq$ b))
Assert $\neg(\tau \models (\mathit{null} \triangleq$ b))
Assert $\neg(\tau \models (\mathit{invalid} \doteq (\mathit{invalid}::('\mathfrak{A})\mathit{String})))$
Assert $\neg(\tau \models \upsilon\ (\mathit{invalid} \doteq (\mathit{invalid}::('\mathfrak{A})\mathit{String})))$
Assert $\neg(\tau \models (\mathit{invalid}$ <> $(\mathit{invalid}::('\mathfrak{A})\mathit{String})))$
Assert $\neg(\tau \models \upsilon\ (\mathit{invalid}$ <> $(\mathit{invalid}::('\mathfrak{A})\mathit{String})))$
Assert $\tau \models (\mathit{null} \doteq (\mathit{null}::('\mathfrak{A})\mathit{String})\ )$
Assert $\tau \models (\mathit{null} \doteq (\mathit{null}::('\mathfrak{A})\mathit{String})\ )$
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 *OCL-collection-type-Pair*
imports *OCL-lib-common*
begin

5.7. Collection Type Pairs: Operations

The OCL standard provides the concept of *Tuples*, i. e. a family of record-types with projection functions. In FeatherWeight OCL, only the theory of a special case is developped, namely the type of Pairs, which is, however, sufficient for all applications since it can be used to mimick all tuples. In particular, it can be used to express operations with multiple arguments, roles of n-ary associations, ...

5.7.1. Semantic Properties of the Type Constructor

lemma $A[simp]{:}Rep\text{-}Pair_{base}\ x \neq None \Longrightarrow Rep\text{-}Pair_{base}\ x \neq null \Longrightarrow (fst\ \lceil\lceil Rep\text{-}Pair_{base}\ x\rceil\rceil) \neq bot$
$\langle proof \rangle$

lemma $A'[simp]{:}\ x \neq bot \Longrightarrow\ x \neq null \Longrightarrow (fst\ \lceil\lceil Rep\text{-}Pair_{base}\ x\rceil\rceil) \neq bot$
$\langle proof \rangle$

lemma $B[simp]{:}Rep\text{-}Pair_{base}\ x \neq None \Longrightarrow Rep\text{-}Pair_{base}\ x \neq null \Longrightarrow (snd\ \lceil\lceil Rep\text{-}Pair_{base}\ x\rceil\rceil) \neq bot$
$\langle proof \rangle$

lemma $B'[simp]{:}x \neq bot \Longrightarrow x \neq null \Longrightarrow (snd\ \lceil\lceil Rep\text{-}Pair_{base}\ x\rceil\rceil) \neq bot$
$\langle proof \rangle$

5.7.2. 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 $StrictRefEq_{Pair}$:
$$((x{::}('\mathfrak{A},'\alpha{::}null,'\beta{::}null)Pair) \doteq y) \equiv (\lambda\ \tau.\ if\ (\upsilon\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau\ then\ (x \triangleq y)\tau\ else\ invalid\ \tau)$$

Property proof in terms of *profile-bin3*

interpretation $StrictRefEq_{Pair}$: $profile\text{-}bin3\ \lambda\ x\ y.\ (x{::}('\mathfrak{A},'\alpha{::}null,'\beta{::}null)Pair) \doteq y$
$\langle proof \rangle$

5.7.3. Standard Operations

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

Definition: OclPair Constructor

definition $OclPair$::$('\mathfrak{A}, '\alpha)$ $val \Rightarrow$
$('\mathfrak{A}, '\beta)$ $val \Rightarrow$
$('\mathfrak{A},'\alpha::null,'\beta::null)$ $Pair$ $(Pair\{(\text{-}),(\text{-})\})$
where $Pair\{X,Y\} \equiv (\lambda\ \tau.\ if\ (\upsilon\ X)\ \tau = true\ \tau \wedge (\upsilon\ Y)\ \tau = true\ \tau$
$then\ Abs\text{-}Pair_{base}\ \lfloor\lfloor(X\ \tau,\ Y\ \tau)\rfloor\rfloor$
$else\ invalid\ \tau)$

interpretation $OclPair$: $profile\text{-}bin4$
$OclPair\ \lambda\ x\ y.\ Abs\text{-}Pair_{base}\ \lfloor\lfloor(x,\ y)\rfloor\rfloor$
$\langle proof \rangle$

Definition: OclFst

definition $OclFirst$:: $('\mathfrak{A},'\alpha::null,'\beta::null)$ $Pair \Rightarrow ('\mathfrak{A}, '\alpha)$ val $(\text{ - }.First'(')$)
where $X\ .First() \equiv (\lambda\ \tau.\ if\ (\delta\ X)\ \tau = true\ \tau$
$then\ fst\ \lceil\lceil Rep\text{-}Pair_{base}\ (X\ \tau)\rceil\rceil$
$else\ invalid\ \tau)$

interpretation $OclFirst$: $profile\text{-}mono2\ OclFirst\ \lambda x.\ \ fst\ \lceil\lceil Rep\text{-}Pair_{base}\ (x)\rceil\rceil$
$\langle proof \rangle$

Definition: OclSnd

definition $OclSecond$:: $('\mathfrak{A},'\alpha::null,'\beta::null)$ $Pair \Rightarrow ('\mathfrak{A}, '\beta)$ val $(\text{- }.Second'(')$)
where $X\ .Second() \equiv (\lambda\ \tau.\ if\ (\delta\ X)\ \tau = true\ \tau$
$then\ snd\ \lceil\lceil Rep\text{-}Pair_{base}\ (X\ \tau)\rceil\rceil$
$else\ invalid\ \tau)$

interpretation $OclSecond$: $profile\text{-}mono2\ OclSecond\ \lambda x.\ \ snd\ \lceil\lceil Rep\text{-}Pair_{base}\ (x)\rceil\rceil$
$\langle proof \rangle$

5.7.4. Logical Properties

lemma $1 : \tau \models \upsilon\ Y \Longrightarrow \tau \models Pair\{X,Y\}\ .First() \triangleq X$
$\langle proof \rangle$

lemma $2 : \tau \models \upsilon\ X \Longrightarrow \tau \models Pair\{X,Y\}\ .Second() \triangleq Y$
$\langle proof \rangle$

5.7.5. Execution Properties

lemma $proj1\text{-}exec$ $[simp,\ code\text{-}unfold]$: $Pair\{X,Y\}\ .First() = (if\ (\upsilon\ Y)\ then\ X\ else\ invalid\ endif)$

$\langle proof \rangle$

lemma *proj2-exec* [*simp*, *code-unfold*] : $Pair\{X,Y\}$ $.Second() = (if\ (\upsilon\ X)\ then\ Y\ else\ invalid$ $endif)$
$\langle proof \rangle$

5.7.6. Test Statements

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

end

theory *OCL-collection-type-Set*
imports *OCL-basic-type-Integer*
begin

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

5.8. Collection Type Set: Operations

5.8.1. As a Motivation for the (infinite) Type Construction: Type-Extensions as Sets

Our notion of typed set goes beyond the usual notion of a finite executable set and is powerful enough to capture *the extension of a type* in UML and OCL. This means we can have in Featherweight OCL Sets containing all possible elements of a type, not only those (finite) ones representable in a state. This holds for base types as well as class types, although the notion for class-types — involving object id's not occuring in a state — requires some care.

In a world with *invalid* and *null*, there are two notions extensions possible:

1. the set of all *defined* values of a type T (for which we will introduce the constant T)

2. the set of all *valid* values of a type T, so including *null* (for which we will introduce the constant T_{null}).

We define the set extensions for the base type *Integer* as follows:

definition $Integer$ :: $('\mathfrak{A},Integer_{base})$ Set
where $Integer \equiv (\lambda\ \tau.\ (Abs\text{-}Set_{base}\ o\ Some\ o\ Some)\ \ ((Some\ o\ Some)\ \text{`}\ (UNIV::int\ set)))$

definition $Integer_{null}$ :: $('\mathfrak{A},Integer_{base})$ Set
where $Integer_{null} \equiv (\lambda\ \tau.\ (Abs\text{-}Set_{base}\ o\ Some\ o\ Some)\ \ (Some\ \text{`}\ (UNIV::int\ option\ set)))$

lemma *Integer-defined* : $\delta\ Integer = true$
⟨*proof*⟩

lemma $Integer_{null}$*-defined* : $\delta\ Integer_{null} = true$
⟨*proof*⟩

This allows the theorems:
$\tau \models \delta\ x \implies \tau \models (Integer{-}{>}includes(x))$ $\tau \models \delta\ x \implies \tau \models Integer \triangleq (Integer{-}{>}including(x))$

and
$\tau \models \upsilon\ x \implies \tau \models (Integer_{null}{-}{>}includes(x))$ $\tau \models \upsilon\ x \implies \tau \models Integer_{null} \triangleq (Integer_{null}{-}{>}including(x))$

which characterize the infiniteness of these sets by a recursive property on these sets.

5.8.2. Validity and Definedness Properties

Every element in a defined set is valid.

lemma *Set-inv-lemma*: $\tau \models (\delta\ X) \implies \forall x{\in}\lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil.\ x \neq bot$
⟨*proof*⟩

lemma *Set-inv-lemma'* :
assumes *x-def* : $\tau \models \delta\ X$
and *e-mem* : $e \in \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil$
shows $\tau \models \upsilon\ (\lambda\text{-}.\ e)$
⟨*proof*⟩

lemma *abs-rep-simp'* :
assumes *S-all-def* : $\tau \models \delta\ S$
shows $Abs\text{-}Set_{base}\ \lfloor\lfloor\lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil\rfloor\rfloor = S\ \tau$
⟨*proof*⟩

lemma *S-lift'* :
assumes *S-all-def* : $(\tau :: '\mathfrak{A}\ st) \models \delta\ S$
shows $\exists S'.\ (\lambda a\ (\text{-}::'\mathfrak{A}\ st).\ a)\ \text{`}\ \lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil = (\lambda a\ (\text{-}::'\mathfrak{A}\ st).\ \lfloor a\rfloor)\ \text{`}\ S'$
⟨*proof*⟩

lemma *invalid-set-OclNot-defined* [*simp*,*code-unfold*]:$\delta(invalid::('\mathfrak{A},'\alpha::null)\ Set) = false$
⟨*proof*⟩
lemma *null-set-OclNot-defined* [*simp*,*code-unfold*]:$\delta(null::('\mathfrak{A},'\alpha::null)\ Set) = false$
⟨*proof*⟩
lemma *invalid-set-valid* [*simp*,*code-unfold*]:$\upsilon(invalid::('\mathfrak{A},'\alpha::null)\ Set) = false$
⟨*proof*⟩

lemma *null-set-valid* [*simp*,*code-unfold*]:$\upsilon(null::('\mathfrak{A},'\alpha::null)\ Set) = true$
⟨*proof*⟩

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

5.8.3. Constants on Sets

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

lemma *mtSet-defined*[*simp*,*code-unfold*]:$\delta(Set\{\}) = true$
⟨*proof*⟩

lemma *mtSet-valid*[*simp*,*code-unfold*]:$\upsilon(Set\{\}) = true$
⟨*proof*⟩

lemma *mtSet-rep-set*: $\lceil\lceil Rep\text{-}Set_{base}\ (Set\{\}\ \tau)\rceil\rceil = \{\}$
⟨*proof*⟩

lemma [*simp*,*code-unfold*]: *const* $Set\{\}$
⟨*proof*⟩

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

5.8.4. Operations

This part provides a collection of operators for the Set type.

Definition: OclIncluding

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

interpretation *OclIncluding* : *profile-bin2 OclIncluding* $\lambda x\ y.\ Abs\text{-}Set_{base}\lfloor\lfloor\lceil\lceil Rep\text{-}Set_{base}\ x\rceil\rceil \cup \{y\}\rfloor\rfloor$
⟨*proof*⟩

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

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

Definition: OclExcluding

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

Definition: OclIncludes

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

Definition: OclExcludes

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

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

Definition: OclSize

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

The following definition follows the requirement of the standard to treat null as neutral element of sets. It is a well-documented exception from the general strictness rule and the rule that the distinguished argument self should be non-null.

Definition: OclIsEmpty

definition $OclIsEmpty$ :: $('\mathfrak{A},'\alpha{::}null)\ Set \Rightarrow\ '\mathfrak{A}\ Boolean$
where $OclIsEmpty\ x = \ ((\upsilon\ x\ and\ not\ (\delta\ x))\ or\ ((OclSize\ x) \doteq \mathbf{0}))$
notation $OclIsEmpty$ $(\text{-->}isEmpty'(')\ )$

Definition: OclNotEmpty

definition $\mathit{OclNotEmpty}$ $::$ $('\mathfrak{A},'\alpha::\mathit{null})\ \mathit{Set} \Rightarrow\ '\mathfrak{A}\ \mathit{Boolean}$
where $\mathit{OclNotEmpty}\ x =\ \mathit{not}(\mathit{OclIsEmpty}\ x)$
notation $\mathit{OclNotEmpty}$ $(\text{-}{-}{>}\mathit{notEmpty}'(')\ )$

Definition: OclANY

definition OclANY $::$ $[('\mathfrak{A},'\alpha::\mathit{null})\ \mathit{Set}] \Rightarrow ('\mathfrak{A},'\alpha)\ \mathit{val}$
where $\mathit{OclANY}\ x = (\lambda\ \tau.\ \mathit{if}\ (\upsilon\ x)\ \tau = \mathit{true}\ \tau$
$\quad\quad \mathit{then}\ \mathit{if}\ (\delta\ x\ \mathit{and}\ \mathit{OclNotEmpty}\ x)\ \tau = \mathit{true}\ \tau$
$\quad\quad\quad \mathit{then}\ \mathit{SOME}\ y.\ y \in \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (x\ \tau)\rceil\rceil$
$\quad\quad\quad \mathit{else}\ \mathit{null}\ \tau$
$\quad\quad \mathit{else}\ \bot\ )$
notation OclANY $(\text{-}{-}{>}\mathit{any}'('))$

Definition: OclForall

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

definition $\mathit{OclForall}$ $::$ $[('\mathfrak{A},'\alpha::\mathit{null})\mathit{Set},('\mathfrak{A},'\alpha)\mathit{val} \Rightarrow ('\mathfrak{A})\mathit{Boolean}] \Rightarrow\ '\mathfrak{A}\ \mathit{Boolean}$
where $\mathit{OclForall}\ S\ P = (\lambda\ \tau.\ \mathit{if}\ (\delta\ S)\ \tau = \mathit{true}\ \tau$
$\quad\quad \mathit{then}\ \mathit{if}\ (\exists\, x \in \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (S\ \tau)\rceil\rceil.\ P(\lambda\ \text{-}.\ x)\ \tau = \mathit{false}\ \tau)$
$\quad\quad\quad \mathit{then}\ \mathit{false}\ \tau$
$\quad\quad\quad \mathit{else}\ \mathit{if}\ (\exists\, x \in \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (S\ \tau)\rceil\rceil.\ P(\lambda\ \text{-}.\ x)\ \tau = \mathit{invalid}\ \tau)$
$\quad\quad\quad\quad \mathit{then}\ \mathit{invalid}\ \tau$
$\quad\quad\quad\quad \mathit{else}\ \mathit{if}\ (\exists\, x \in \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (S\ \tau)\rceil\rceil.\ P(\lambda\ \text{-}.\ x)\ \tau = \mathit{null}\ \tau)$
$\quad\quad\quad\quad\quad \mathit{then}\ \mathit{null}\ \tau$
$\quad\quad\quad\quad\quad \mathit{else}\ \mathit{true}\ \tau$
$\quad\quad \mathit{else}\ \bot)$
syntax
$\text{-}\mathit{OclForall}$ $::$ $[('\mathfrak{A},'\alpha::\mathit{null})\ \mathit{Set},\mathit{id},('\mathfrak{A})\mathit{Boolean}] \Rightarrow\ '\mathfrak{A}\ \mathit{Boolean}$ $((\text{-}){-}{>}\mathit{forAll}'(\text{-}|\text{-}'))$
translations
$X{-}{>}\mathit{forAll}(x \mid P) == \mathit{CONST}\ \mathit{OclForall}\ X\ (\%x.\ P)$

Definition: OclExists

Like OclForall, OclExists is also not strict.

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

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

Definition: OclIterate

definition $\mathit{OclIterate}$ $::$ $[('\mathfrak{A},'\alpha::\mathit{null})\ \mathit{Set},('\mathfrak{A},'\beta::\mathit{null})\mathit{val},$
$\quad ('\mathfrak{A},'\alpha)\mathit{val} \Rightarrow ('\mathfrak{A},'\beta)\mathit{val} \Rightarrow ('\mathfrak{A},'\beta)\mathit{val}] \Rightarrow ('\mathfrak{A},'\beta)\mathit{val}$

where *OclIterate S A F* = (λ τ. *if* (δ *S*) τ = *true* τ $\wedge$ (υ *A*) τ = *true* τ $\wedge$ *finite*$\lceil\lceil$*Rep-Set*$_{base}$ (*S* τ)$\rceil\rceil$
then (*Finite-Set.fold* (*F*) (*A*) ((λ*a* τ. *a*) ' $\lceil\lceil$*Rep-Set*$_{base}$ (*S* τ)$\rceil\rceil$))τ
else $\perp$)

syntax

-*OclIterate* :: [('$\mathfrak{A}$,'α::*null*) *Set*, *idt*, *idt*, 'α, 'β] => ('$\mathfrak{A}$,'γ)*val*
(- −>*iterate*'(-;-=- | -'))

translations

X−>*iterate*(*a*; *x* = *A* | *P*) == *CONST OclIterate X A* (%*a*. (% *x*. *P*))

Definition: OclSelect

definition *OclSelect* :: [('$\mathfrak{A}$,'α::*null*)*Set*,('$\mathfrak{A}$,'α)*val*⇒('$\mathfrak{A}$)*Boolean*] ⇒ ('$\mathfrak{A}$,'α)*Set*

where *OclSelect S P* = ($\lambda\tau$. *if* (δ *S*) τ = *true* τ
then if ($\exists$ *x*$\in\lceil\lceil$*Rep-Set*$_{base}$ (*S* τ)$\rceil\rceil$. *P*(λ -. *x*) τ = *invalid* τ)
then invalid τ
else Abs-Set$_{base}$ $\lfloor\lfloor$\{*x*$\in\lceil\lceil$ *Rep-Set*$_{base}$ (*S* τ)$\rceil\rceil$. *P* (λ-. *x*) τ ≠ *false* τ\}$\rfloor\rfloor$
else invalid τ)

syntax

-*OclSelect* :: [('$\mathfrak{A}$,'α::*null*) *Set*,*id*,('$\mathfrak{A}$)*Boolean*] ⇒ '$\mathfrak{A}$ *Boolean* ((-)−>*select*'(-|-'))

translations

X−>*select*(*x* | *P*) == *CONST OclSelect X* (% *x*. *P*)

Definition: OclReject

definition *OclReject* :: [('$\mathfrak{A}$,'α::*null*)*Set*,('$\mathfrak{A}$,'α)*val*⇒('$\mathfrak{A}$)*Boolean*] ⇒ ('$\mathfrak{A}$,'α::*null*)*Set*

where *OclReject S P* = *OclSelect S* (*not o P*)

syntax

-*OclReject* :: [('$\mathfrak{A}$,'α::*null*) *Set*,*id*,('$\mathfrak{A}$)*Boolean*] ⇒ '$\mathfrak{A}$ *Boolean* ((-)−>*reject*'(-|-'))

translations

X−>*reject*(*x* | *P*) == *CONST OclReject X* (% *x*. *P*)

Definition (futur operators)

consts

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

notation

OclCount (-−>*count*'(-'))

notation

OclSum (-−>*sum*'('))

notation

OclIncludesAll (-−>*includesAll*'(-'))

notation
$OclExcludesAll$ (-−>*excludesAll*′(-′))
notation
$OclComplement$ (-−>*complement*′(′))
notation
$OclUnion$ (-−>*union*′(-′))
notation
$OclIntersection$(-−>*intersection*′(-′))

Validity and Definedness Properties

OclIncluding

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

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

lemma *OclIncluding-defined-args-valid*′[*simp*,*code-unfold*]:
$\delta(X{-}{>}including(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
⟨*proof*⟩

lemma *OclIncluding-valid-args-valid*′′[*simp*,*code-unfold*]:
$\upsilon(X{-}{>}including(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
⟨*proof*⟩

OclExcluding

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

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

lemma *OclExcluding-valid-args-valid*′[*simp*,*code-unfold*]:
$\delta(X{-}{>}excluding(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
⟨*proof*⟩

lemma *OclExcluding-valid-args-valid*′′[*simp*,*code-unfold*]:
$\upsilon(X{-}{>}excluding(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
⟨*proof*⟩

OclIncludes

lemma *OclIncludes-defined-args-valid*:
$(\tau \models \delta(X{-}{>}includes(x))) = ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
$\langle proof \rangle$

lemma *OclIncludes-valid-args-valid*:
$(\tau \models \upsilon(X{-}{>}includes(x))) = ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
$\langle proof \rangle$

lemma *OclIncludes-valid-args-valid'*[*simp*,*code-unfold*]:
$\delta(X{-}{>}includes(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
$\langle proof \rangle$

lemma *OclIncludes-valid-args-valid''*[*simp*,*code-unfold*]:
$\upsilon(X{-}{>}includes(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
$\langle proof \rangle$

OclExcludes

lemma *OclExcludes-defined-args-valid*:
$(\tau \models \delta(X{-}{>}excludes(x))) = ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
$\langle proof \rangle$

lemma *OclExcludes-valid-args-valid*:
$(\tau \models \upsilon(X{-}{>}excludes(x))) = ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
$\langle proof \rangle$

lemma *OclExcludes-valid-args-valid'*[*simp*,*code-unfold*]:
$\delta(X{-}{>}excludes(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
$\langle proof \rangle$

lemma *OclExcludes-valid-args-valid''*[*simp*,*code-unfold*]:
$\upsilon(X{-}{>}excludes(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
$\langle proof \rangle$

OclSize

lemma *OclSize-defined-args-valid*: $\tau \models \delta\ (X{-}{>}size()) \Longrightarrow \tau \models \delta\ X$
$\langle proof \rangle$

lemma *OclSize-infinite*:
assumes *non-finite*:$\tau \models not(\delta(S{-}{>}size()))$
shows $(\tau \models not(\delta(S))) \vee \neg\ finite\ \lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil$
$\langle proof \rangle$

lemma $\tau \models \delta\ X \Longrightarrow \neg\ finite\ \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil \Longrightarrow \neg\ \tau \models \delta\ (X{-}{>}size())$
$\langle proof \rangle$

lemma *size-defined*:
assumes *X-finite*: $\bigwedge\tau.\ finite\ \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil$
shows $\delta\ (X{-}{>}size()) = \delta\ X$

$\langle proof \rangle$

lemma *size-defined′*:
assumes *X-finite*: *finite* $\lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil$
shows $(\tau \models \delta\ (X{-}{>}size())) = (\tau \models \delta\ X)$
$\langle proof \rangle$

OclIsEmpty

lemma *OclIsEmpty-defined-args-valid*:$\tau \models \delta\ (X{-}{>}isEmpty()) \Longrightarrow \tau \models \upsilon\ X$
$\langle proof \rangle$

lemma $\tau \models \delta\ (null{-}{>}isEmpty())$
$\langle proof \rangle$

lemma *OclIsEmpty-infinite*: $\tau \models \delta\ X \Longrightarrow \neg\ finite\ \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil \Longrightarrow \neg\ \tau \models \delta\ (X{-}{>}isEmpty())$
$\langle proof \rangle$

OclNotEmpty

lemma *OclNotEmpty-defined-args-valid*:$\tau \models \delta\ (X{-}{>}notEmpty()) \Longrightarrow \tau \models \upsilon\ X$
$\langle proof \rangle$

lemma $\tau \models \delta\ (null{-}{>}notEmpty())$
$\langle proof \rangle$

lemma *OclNotEmpty-infinite*: $\tau \models \delta\ X \Longrightarrow \neg\ finite\ \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil \Longrightarrow \neg\ \tau \models \delta\ (X{-}{>}notEmpty())$
$\langle proof \rangle$

lemma *OclNotEmpty-has-elt* : $\tau \models \delta\ X \Longrightarrow$
$\tau \models X{-}{>}notEmpty() \Longrightarrow$
$\exists\, e.\ e \in \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil$
$\langle proof \rangle$

OclANY

lemma *OclANY-defined-args-valid*: $\tau \models \delta\ (X{-}{>}any()) \Longrightarrow \tau \models \delta\ X$
$\langle proof \rangle$

lemma $\tau \models \delta\ X \Longrightarrow \tau \models X{-}{>}isEmpty() \Longrightarrow \neg\ \tau \models \delta\ (X{-}{>}any())$
$\langle proof \rangle$

lemma *OclANY-valid-args-valid*:
$(\tau \models \upsilon(X{-}{>}any())) = (\tau \models \upsilon\ X)$
$\langle proof \rangle$

lemma *OclANY-valid-args-valid′′*[*simp*,*code-unfold*]:
$\upsilon(X{-}{>}any()) = (\upsilon\ X)$
$\langle proof \rangle$

Execution with Invalid or Null or Infinite Set as Argument

OclIncluding

lemma *OclIncluding-invalid*[*simp*,*code-unfold*]:(*invalid*−>*including*(*x*)) = *invalid*
⟨*proof*⟩

lemma *OclIncluding-invalid-args*[*simp*,*code-unfold*]:(*X*−>*including*(*invalid*)) = *invalid*
⟨*proof*⟩

lemma *OclIncluding-null*[*simp*,*code-unfold*]:(*null*−>*including*(*x*)) = *invalid*
⟨*proof*⟩

OclExcluding

lemma *OclExcluding-invalid*[*simp*,*code-unfold*]:(*invalid*−>*excluding*(*x*)) = *invalid*
⟨*proof*⟩

lemma *OclExcluding-invalid-args*[*simp*,*code-unfold*]:(*X*−>*excluding*(*invalid*)) = *invalid*
⟨*proof*⟩

lemma *OclExcluding-null*[*simp*,*code-unfold*]:(*null*−>*excluding*(*x*)) = *invalid*
⟨*proof*⟩

OclIncludes

lemma *OclIncludes-invalid*[*simp*,*code-unfold*]:(*invalid*−>*includes*(*x*)) = *invalid*
⟨*proof*⟩

lemma *OclIncludes-invalid-args*[*simp*,*code-unfold*]:(*X*−>*includes*(*invalid*)) = *invalid*
⟨*proof*⟩

lemma *OclIncludes-null*[*simp*,*code-unfold*]:(*null*−>*includes*(*x*)) = *invalid*
⟨*proof*⟩

OclExcludes

lemma *OclExcludes-invalid*[*simp*,*code-unfold*]:(*invalid*−>*excludes*(*x*)) = *invalid*
⟨*proof*⟩

lemma *OclExcludes-invalid-args*[*simp*,*code-unfold*]:(*X*−>*excludes*(*invalid*)) = *invalid*
⟨*proof*⟩

lemma *OclExcludes-null*[*simp*,*code-unfold*]:(*null*−>*excludes*(*x*)) = *invalid*
⟨*proof*⟩

OclSize

lemma *OclSize-invalid*[*simp*,*code-unfold*]:(*invalid*−>*size*()) = *invalid*
⟨*proof*⟩

lemma *OclSize-null*[*simp*,*code-unfold*]:(*null*−>*size*()) = *invalid*
⟨*proof*⟩

OclIsEmpty

lemma $OclIsEmpty\text{-}invalid[simp,code\text{-}unfold]:(invalid{-}{>}isEmpty()) = invalid$
⟨*proof*⟩

lemma $OclIsEmpty\text{-}null[simp,code\text{-}unfold]:(null{-}{>}isEmpty()) = true$
⟨*proof*⟩

OclNotEmpty

lemma $OclNotEmpty\text{-}invalid[simp,code\text{-}unfold]:(invalid{-}{>}notEmpty()) = invalid$
⟨*proof*⟩

lemma $OclNotEmpty\text{-}null[simp,code\text{-}unfold]:(null{-}{>}notEmpty()) = false$
⟨*proof*⟩

OclANY

lemma $OclANY\text{-}invalid[simp,code\text{-}unfold]:(invalid{-}{>}any()) = invalid$
⟨*proof*⟩

lemma $OclANY\text{-}null[simp,code\text{-}unfold]:(null{-}{>}any()) = null$
⟨*proof*⟩

OclForall

lemma $OclForall\text{-}invalid[simp,code\text{-}unfold]:invalid{-}{>}forAll(a|\ P\ a) = invalid$
⟨*proof*⟩

lemma $OclForall\text{-}null[simp,code\text{-}unfold]:null{-}{>}forAll(a\ |\ P\ a) = invalid$
⟨*proof*⟩

OclExists

lemma $OclExists\text{-}invalid[simp,code\text{-}unfold]:invalid{-}{>}exists(a|\ P\ a) = invalid$
⟨*proof*⟩

lemma $OclExists\text{-}null[simp,code\text{-}unfold]:null{-}{>}exists(a\ |\ P\ a) = invalid$
⟨*proof*⟩

OclIterate

lemma $OclIterate\text{-}invalid[simp,code\text{-}unfold]:invalid{-}{>}iterate(a;\ x = A\ |\ P\ a\ x) = invalid$
⟨*proof*⟩

lemma $OclIterate\text{-}null[simp,code\text{-}unfold]:null{-}{>}iterate(a;\ x = A\ |\ P\ a\ x) = invalid$
⟨*proof*⟩

lemma $OclIterate\text{-}invalid\text{-}args[simp,code\text{-}unfold]:S{-}{>}iterate(a;\ x = invalid\ |\ P\ a\ x) = invalid$
⟨*proof*⟩

An open question is this ...

lemma $S{-}{>}iterate(a;\ x = null\ |\ P\ a\ x) = invalid$
⟨*proof*⟩

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

OclSelect

lemma *OclSelect-invalid*[*simp*,*code-unfold*]:$invalid{-}{>}select(a \mid P\ a) = invalid$
⟨*proof*⟩

lemma *OclSelect-null*[*simp*,*code-unfold*]:$null{-}{>}select(a \mid P\ a) = invalid$
⟨*proof*⟩

OclReject

lemma *OclReject-invalid*[*simp*,*code-unfold*]:$invalid{-}{>}reject(a \mid P\ a) = invalid$
⟨*proof*⟩

lemma *OclReject-null*[*simp*,*code-unfold*]:$null{-}{>}reject(a \mid P\ a) = invalid$
⟨*proof*⟩

Context Passing

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

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

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

lemma *cp-OclIncludes1*:
$(X{-}{>}includes(x))\ \tau = (X{-}{>}includes(\lambda\ \text{-}.\ x\ \tau))\ \tau$
⟨*proof*⟩

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

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

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

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

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

lemma *cp-OclForall*:
$(S{-}{>}forAll(x \mid P\ x))\ \tau = ((\lambda\ \text{-}.\ S\ \tau){-}{>}forAll(x \mid P\ (\lambda\ \text{-}.\ x\ \tau)))\ \tau$
⟨*proof*⟩

lemma *cp-OclForall1* [*simp*,*intro*!]:
$cp\ S \Longrightarrow cp\ (\lambda X.\ ((S\ X){-}{>}forAll(x \mid P\ x)))$
⟨*proof*⟩

lemma
$cp\ (\lambda X\ St\ x.\ P\ (\lambda\tau.\ x)\ X\ St) \Longrightarrow cp\ S \Longrightarrow cp\ (\lambda X.\ (S\ X){-}{>}forAll(x|P\ x\ X))$
⟨*proof*⟩

lemma
$cp\ S \Longrightarrow$
$(\bigwedge\ x.\ cp(P\ x)) \Longrightarrow$
$cp(\lambda X.\ ((S\ X){-}{>}forAll(x \mid P\ x\ X)))$
⟨*proof*⟩

lemma *cp-OclExists*:
$(S{-}{>}exists(x \mid P\ x))\ \tau = ((\lambda\ \text{-}.\ S\ \tau){-}{>}exists(x \mid P\ (\lambda\ \text{-}.\ x\ \tau)))\ \tau$
⟨*proof*⟩

lemma *cp-OclExists1* [*simp*,*intro*!]:
$cp\ S \Longrightarrow cp\ (\lambda X.\ ((S\ X){-}{>}exists(x \mid P\ x)))$
⟨*proof*⟩

lemma *cp-OclIterate*: $(X{-}{>}iterate(a;\ x = A \mid P\ a\ x))\ \tau =$
$((\lambda\ \text{-}.\ X\ \tau){-}{>}iterate(a;\ x = A \mid P\ a\ x))\ \tau$
⟨*proof*⟩

lemma *cp-OclSelect*: $(X{-}{>}select(a \mid P\ a))\ \tau =$
$((\lambda\ \text{-}.\ X\ \tau){-}{>}select(a \mid P\ a))\ \tau$
⟨*proof*⟩

lemma *cp-OclReject*: $(X{-}{>}reject(a \mid P\ a))\ \tau =$
$((\lambda\ \text{-}.\ X\ \tau){-}{>}reject(a \mid P\ a))\ \tau$
⟨*proof*⟩

lemmas $cp\text{-}intro''_{Set}$[*intro*!,*simp*,*code-unfold*] =
cp-OclIncluding [*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of OclIncluding*]]
cp-OclExcluding [*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of OclExcluding*]]

cp-OclIncludes [*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of OclIncludes*]]
cp-OclExcludes [*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of OclExcludes*]]
cp-OclSize [*THEN allI*[*THEN allI*[*THEN cpI1*], *of OclSize*]]
cp-OclIsEmpty [*THEN allI*[*THEN allI*[*THEN cpI1*], *of OclIsEmpty*]]
cp-OclNotEmpty [*THEN allI*[*THEN allI*[*THEN cpI1*], *of OclNotEmpty*]]
cp-OclANY [*THEN allI*[*THEN allI*[*THEN cpI1*], *of OclANY*]]

Const

lemma *const-OclIncluding*[*simp*,*code-unfold*] :
assumes *const-x* : *const x*
and *const-S* : *const S*
shows *const* (*S*−>*including*(*x*))
⟨*proof*⟩

5.8.5. 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 $StrictRefEq_{Set}$:

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

One might object here that for the case of objects, this is an empty definition. The answer is no, we will restrain later on states and objects such that any object has its oid stored inside the object (so the ref, under which an object can be referenced in the store will represented in the object itself). For such well-formed stores that satisfy this invariant (the WFF-invariant), the referential equality and the strong equality—and therefore the strict equality on sets in the sense above—coincides.

Property proof in terms of *profile-bin3*

interpretation $StrictRefEq_{Set}$: *profile-bin3* $\lambda\ x\ y.\ (x{::}('\mathfrak{A},'\alpha{::}null)Set) \doteq y$
⟨*proof*⟩

Execution Rules on OclIncluding

lemma *OclIncluding-finite-rep-set* :
assumes *X-def* : $\tau \models \delta\ X$
and *x-val* : $\tau \models \upsilon\ x$
shows $finite\ \lceil\lceil Rep\text{-}Set_{base}\ (X{-}{>}including(x)\ \tau)\rceil\rceil = finite\ \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil$
⟨*proof*⟩

lemma *OclIncluding-rep-set*:
assumes *S-def*: $\tau \models \delta\ S$

shows $\lceil\lceil Rep\text{-}Set_{base}\ (S{-}{>}including(\lambda\text{-}.\ \lfloor\lfloor x\rfloor\rfloor)\ \tau)\rceil\rceil = insert\ \lfloor\lfloor x\rfloor\rfloor\ \lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil$
⟨*proof*⟩

lemma *OclIncluding-notempty-rep-set*:
assumes *X-def*: $\tau \models \delta\ X$
and *a-val*: $\tau \models \upsilon\ a$
shows $\lceil\lceil Rep\text{-}Set_{base}\ (X{-}{>}including(a)\ \tau)\rceil\rceil \neq \{\}$
⟨*proof*⟩

lemma *OclIncluding-includes0*:
assumes $\tau \models X{-}{>}includes(x)$
shows $X{-}{>}including(x)\ \tau = X\ \tau$
⟨*proof*⟩

lemma *OclIncluding-includes*:
assumes $\tau \models X{-}{>}includes(x)$
shows $\tau \models X{-}{>}including(x) \triangleq X$
⟨*proof*⟩

lemma *OclIncluding-commute0* :
assumes *S-def* : $\tau \models \delta\ S$
and *i-val* : $\tau \models \upsilon\ i$
and *j-val* : $\tau \models \upsilon\ j$
shows $\tau \models ((S :: ('\mathfrak{A}, 'a::null)\ Set){-}{>}including(i){-}{>}including(j) \triangleq (S{-}{>}including(j){-}{>}including(i)))$
⟨*proof*⟩

lemma *OclIncluding-commute*[*simp*,*code-unfold*]:
$((S :: ('\mathfrak{A}, 'a::null)\ Set){-}{>}including(i){-}{>}including(j) = (S{-}{>}including(j){-}{>}including(i)))$
⟨*proof*⟩

Execution Rules on OclExcluding

lemma *OclExcluding-finite-rep-set* :
assumes *X-def* : $\tau \models \delta\ X$
and *x-val* : $\tau \models \upsilon\ x$
shows *finite* $\lceil\lceil Rep\text{-}Set_{base}\ (X{-}{>}excluding(x)\ \tau)\rceil\rceil =$ *finite* $\lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil$
⟨*proof*⟩

lemma *OclExcluding-rep-set*:
assumes *S-def*: $\tau \models \delta\ S$
shows $\lceil\lceil Rep\text{-}Set_{base}\ (S{-}{>}excluding(\lambda\text{-}.\ \lfloor\lfloor x\rfloor\rfloor)\ \tau)\rceil\rceil = \lceil\lceil Rep\text{-}Set_{base}\ (S\ \tau)\rceil\rceil - \{\lfloor\lfloor x\rfloor\rfloor\}$
⟨*proof*⟩

lemma *OclExcluding-excludes0*:
assumes $\tau \models X{-}{>}excludes(x)$
shows $X{-}{>}excluding(x)\ \tau = X\ \tau$
⟨*proof*⟩

lemma *OclExcluding-excludes*:
assumes $\tau \models X{-}{>}excludes(x)$
shows $\tau \models X{-}{>}excluding(x) \triangleq X$
$\langle proof \rangle$

lemma *OclExcluding-charn0*[*simp*]:
assumes *val-x*:$\tau \models (\upsilon\ x)$
shows $\tau \models ((Set\{\}{-}{>}excluding(x)) \triangleq Set\{\})$
$\langle proof \rangle$

lemma *OclExcluding-commute0* :
assumes *S-def* : $\tau \models \delta\ S$
and *i-val* : $\tau \models \upsilon\ i$
and *j-val* : $\tau \models \upsilon\ j$
shows $\tau \models ((S :: ('\mathfrak{A}, 'a::null)\ Set){-}{>}excluding(i){-}{>}excluding(j) \triangleq (S{-}{>}excluding(j){-}{>}excluding(i)))$
$\langle proof \rangle$

lemma *OclExcluding-commute*[*simp*,*code-unfold*]:
$((S :: ('\mathfrak{A}, 'a::null)\ Set){-}{>}excluding(i){-}{>}excluding(j) = (S{-}{>}excluding(j){-}{>}excluding(i)))$
$\langle proof \rangle$

lemma *OclExcluding-charn0-exec*[*simp*,*code-unfold*]:
$(Set\{\}{-}{>}excluding(x)) = (if\ (\upsilon\ x)\ then\ Set\{\}\ else\ invalid\ endif)$
$\langle proof \rangle$

lemma *OclExcluding-charn1*:
assumes *def-X*:$\tau \models (\delta\ X)$
and *val-x*:$\tau \models (\upsilon\ x)$
and *val-y*:$\tau \models (\upsilon\ y)$
and *neq* :$\tau \models not(x \triangleq y)$
shows $\tau \models ((X{-}{>}including(x)){-}{>}excluding(y)) \triangleq ((X{-}{>}excluding(y)){-}{>}including(x))$
$\langle proof \rangle$

lemma *OclExcluding-charn2*:
assumes *def-X*:$\tau \models (\delta\ X)$
and *val-x*:$\tau \models (\upsilon\ x)$
shows $\tau \models (((X{-}{>}including(x)){-}{>}excluding(x)) \triangleq (X{-}{>}excluding(x)))$
$\langle proof \rangle$

theorem *OclExcluding-charn3*: $((X{-}{>}including(x)){-}{>}excluding(x)) = (X{-}{>}excluding(x))$

⟨*proof*⟩

One would like a generic theorem of the form:

lemma OclExcluding_charn_exec:
"(X−>including(x::('$\mathfrak{A}$,'a::null)val)−>excluding(y)) =
(if δ X then if x $\doteq$y
then X−>excluding(y)
else X−>excluding(y)−>including(x)
endif
else invalid endif)"

Unfortunately, this does not hold in general, since referential equality is an overloaded concept and has to be defined for each type individually. Consequently, it is only valid for concrete type instances for Boolean, Integer, and Sets thereof...

The computational law *OclExcluding-charn-exec* becomes generic since it uses strict equality which in itself is generic. It is possible to prove the following generic theorem and instantiate it later (using properties that link the polymorphic logical strong equality with the concrete instance of strict quality).

lemma *OclExcluding-charn-exec*:
assumes *strict1*: $(invalid \doteq y) = invalid$
and *strict2*: $(x \doteq invalid) = invalid$
and *StrictRefEq-valid-args-valid*: $\bigwedge (x::('\mathfrak{A},'a::null)val)\ y\ \tau.$
$(\tau \models \delta\ (x \doteq y)) = ((\tau \models (\upsilon\ x)) \wedge (\tau \models \upsilon\ y))$
and *cp-StrictRefEq*: $\bigwedge (X::('\mathfrak{A},'a::null)val)\ Y\ \tau.\ (X \doteq Y)\ \tau = ((\lambda\text{-}.\ X\ \tau) \doteq (\lambda\text{-}.\ Y\ \tau))\ \tau$
and *StrictRefEq-vs-StrongEq*: $\bigwedge (x::('\mathfrak{A},'a::null)val)\ y\ \tau.$
$\tau \models \upsilon\ x \Longrightarrow \tau \models \upsilon\ y \Longrightarrow (\tau \models ((x \doteq y) \triangleq (x \triangleq y)))$
shows $(X{-}{>}including(x::('\mathfrak{A},'a::null)val){-}{>}excluding(y)) =$
$(if\ \delta\ X\ then\ if\ x \doteq y$
$then\ X{-}{>}excluding(y)$
$else\ X{-}{>}excluding(y){-}{>}including(x)$
$endif$
$else\ invalid\ endif)$
⟨*proof*⟩

schematic-lemma *OclExcluding-charn-exec*$_{Integer}$[*simp*,*code-unfold*]: *?X*
⟨*proof*⟩

schematic-lemma *OclExcluding-charn-exec*$_{Boolean}$[*simp*,*code-unfold*]: *?X*
⟨*proof*⟩

schematic-lemma *OclExcluding-charn-exec*$_{Set}$[*simp*,*code-unfold*]: *?X*
⟨*proof*⟩

Execution Rules on OclIncludes

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

lemma *OclIncludes-charn0'*[*simp*,*code-unfold*]:
$Set\{\}{-}{>}includes(x) = (if\ \upsilon\ x\ then\ false\ else\ invalid\ endif)$
⟨*proof*⟩

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

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

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*: $\bigwedge (X::('\mathfrak{A},'a::null)val)\ Y\ \tau.\ (X \doteq Y)\ \tau = ((\lambda\text{-}.\ X\ \tau) \doteq (\lambda\text{-}.\ Y\ \tau))\ \tau$
and *StrictRefEq-vs-StrongEq*: $\bigwedge (x::('\mathfrak{A},'a::null)val)\ y\ \tau.$
$\tau \models \upsilon\ x \Longrightarrow \tau \models \upsilon\ y \Longrightarrow (\tau \models ((x \doteq y) \triangleq (x \triangleq y)))$
shows
$(X{-}{>}including(x::('\mathfrak{A},'a::null)val){-}{>}includes(y)) =$
$(if\ \delta\ X\ then\ if\ x \doteq y\ then\ true\ else\ X{-}{>}includes(y)\ endif\ else\ invalid\ endif)$
⟨*proof*⟩

schematic-lemma *OclIncludes-execute*$_{Integer}$[*simp*,*code-unfold*]: *?X*
⟨*proof*⟩

schematic-lemma *OclIncludes-execute*$_{Boolean}$[*simp*,*code-unfold*]: *?X*
⟨*proof*⟩

schematic-lemma *OclIncludes-execute*$_{Set}$[*simp*,*code-unfold*]: *?X*
⟨*proof*⟩

lemma *OclIncludes-including-generic* :
assumes *OclIncludes-execute-generic* [*simp*] : $\bigwedge X\ x\ y.$
$(X{-}{>}including(x{::}('\mathfrak{A},'a{::}null)val){-}{>}includes(y)) =$
$(if\ \delta\ X\ then\ if\ x \doteq y\ then\ true\ else\ X{-}{>}includes(y)\ endif\ else\ invalid\ endif)$
and *StrictRefEq-strict''* : $\bigwedge x\ y.\ \delta\ ((x{::}('\mathfrak{A},'a{::}null)val) \doteq y) = (\upsilon(x)\ and\ \upsilon(y))$
and *a-val* : $\tau \models \upsilon\ a$
and *x-val* : $\tau \models \upsilon\ x$
and *S-incl* : $\tau \models (S){-}{>}includes((x{::}('\mathfrak{A},'a{::}null)val))$
shows $\tau \models S{-}{>}including((a{::}('\mathfrak{A},'a{::}null)val)){-}{>}includes(x)$
⟨*proof*⟩

lemmas *OclIncludes-including*$_{Integer}$ =
OclIncludes-including-generic[*OF OclIncludes-execute*$_{Integer}$ *StrictRefEq*$_{Integer}$.*def-homo*]

Execution Rules on OclExcludes

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

Execution Rules on OclSize

lemma [*simp*,*code-unfold*]: $Set\{\}\ {-}{>}size() = \mathbf{0}$
⟨*proof*⟩

lemma *OclSize-including-exec*[*simp*,*code-unfold*]:
$((X\ {-}{>}including(x))\ {-}{>}size()) = (if\ \delta\ X\ and\ \upsilon\ x\ then$
$X\ {-}{>}size()\ +_{int}\ if\ X\ {-}{>}includes(x)\ then\ \mathbf{0}\ else\ \mathbf{1}\ endif$
$else$
$invalid$
$endif)$
⟨*proof*⟩

Execution Rules on OclIsEmpty

lemma [*simp*,*code-unfold*]: $Set\{\}{-}{>}isEmpty() = true$
⟨*proof*⟩

lemma *OclIsEmpty-including* [*simp*]:
assumes *X-def*: $\tau \models \delta\ X$
and *X-finite*: *finite* $\lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil$
and *a-val*: $\tau \models \upsilon\ a$
shows $X{-}{>}including(a){-}{>}isEmpty()\ \tau = false\ \tau$
⟨*proof*⟩

Execution Rules on OclNotEmpty

lemma [*simp*,*code-unfold*]: $Set\{\}{-}{>}notEmpty() = false$
⟨*proof*⟩

lemma *OclNotEmpty-including* [*simp*,*code-unfold*]:
assumes *X-def*: $\tau \models \delta\ X$
  and *X-finite*: $finite\ \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil$
  and *a-val*: $\tau \models \upsilon\ a$
shows $X{-}{>}including(a){-}{>}notEmpty()\ \tau = true\ \tau$
⟨*proof*⟩

Execution Rules on OclANY

lemma [*simp*,*code-unfold*]: $Set\{\}{-}{>}any() = null$
⟨*proof*⟩

lemma *OclANY-singleton-exec*[*simp*,*code-unfold*]:
  $(Set\{\}{-}{>}including(a)){-}{>}any() = a$
⟨*proof*⟩

Execution Rules on OclForall

lemma *OclForall-mtSet-exec*[*simp*,*code-unfold*] :$((Set\{\}){-}{>}forAll(z|\ P(z))) = true$
⟨*proof*⟩

The following rule is a main theorem of our approach: From a denotational definition that assures consistency, but may be — as in the case of the *OclForall X P* — dauntingly complex, we derive operational rules that can serve as a gold-standard for operational execution, since they may be evaluated in whatever situation and according to whatever strategy. In the case of *OclForall X P*, the operational rule gives immediately a way to evaluation in any finite (in terms of conventional OCL: denotable) set, although the rule also holds for the infinite case:
$Integer_{null}{-}{>}forAll(x|Integer_{null}{-}{>}forAll(y|x +_{int} y \triangleq y +_{int} x))$
or even:
$Integer{-}{>}forAll(x|Integer{-}{>}forAll(y|x +_{int} y \doteq y +_{int} x))$
are valid OCL statements in any context τ.

theorem *OclForall-including-exec*[*simp*,*code-unfold*] :
  assumes $cp0 : cp\ P$
  shows $((S{-}{>}including(x)){-}{>}forAll(z \mid P(z))) = (if\ \delta\ S\ and\ \upsilon\ x$
  $then\ P\ x\ and\ (S{-}{>}forAll(z \mid P(z)))$
  $else\ invalid$
  $endif)$
⟨*proof*⟩

Execution Rules on OclExists

lemma *OclExists-mtSet-exec*[*simp*,*code-unfold*] :
$((Set\{\}){-}{>}exists(z \mid P(z))) = false$

⟨*proof*⟩

lemma *OclExists-including-exec*[*simp*,*code-unfold*] :
assumes *cp*: *cp P*
shows $((S{-}{>}including(x)){-}{>}exists(z \mid P(z))) = (if\ \delta\ S\ and\ \upsilon\ x$
$then\ P\ x\ or\ (S{-}{>}exists(z \mid P(z)))$
$else\ invalid$
$endif)$
⟨*proof*⟩

Execution Rules on OclIterate

lemma *OclIterate-empty*[*simp*,*code-unfold*]: $((Set\{\}){-}{>}iterate(a;\ x = A \mid P\ a\ x)) = A$
⟨*proof*⟩

In particular, this does hold for A = null.

lemma *OclIterate-including*:
assumes *S-finite*: $\tau \models \delta(S{-}{>}size())$
and *F-valid-arg*: $(\upsilon\ A)\ \tau = (\upsilon\ (F\ a\ A))\ \tau$
and *F-commute*: *comp-fun-commute F*
and *F-cp*: $\bigwedge\ x\ y\ \tau.\ F\ x\ y\ \tau = F\ (\lambda\ \text{-}.\ x\ \tau)\ y\ \tau$
shows $((S{-}{>}including(a)){-}{>}iterate(a;\ x = A \mid F\ a\ x))\ \tau =$
$((S{-}{>}excluding(a)){-}{>}iterate(a;\ x = F\ a\ A \mid F\ a\ x))\ \tau$
⟨*proof*⟩

Execution Rules on OclSelect

lemma *OclSelect-mtSet-exec*[*simp*,*code-unfold*]: *OclSelect mtSet P* = *mtSet*
⟨*proof*⟩

definition *OclSelect-body* :: - ⇒ - ⇒ - ⇒ ($'\mathfrak{A}$, *'a option option*) *Set*
$\equiv (\lambda P\ x\ acc.\ if\ P\ x \doteq false\ then\ acc\ else\ acc{-}{>}including(x)\ endif)$

theorem *OclSelect-including-exec*[*simp*,*code-unfold*]:
assumes *P-cp* : *cp P*
shows $OclSelect\ (X{-}{>}including(y))\ P = OclSelect\text{-}body\ P\ y\ (OclSelect\ (X{-}{>}excluding(y))\ P)$
(**is** - = *?select*)
⟨*proof*⟩

Execution Rules on OclReject

lemma *OclReject-mtSet-exec*[*simp*,*code-unfold*]: *OclReject mtSet P* = *mtSet*
⟨*proof*⟩

lemma *OclReject-including-exec*[*simp*,*code-unfold*]:
assumes *P-cp* : *cp P*
shows $OclReject\ (X{-}{>}including(y))\ P = OclSelect\text{-}body\ (not\ o\ P)\ y\ (OclReject\ (X{-}{>}excluding(y))\ P)$
⟨*proof*⟩

Execution Rules Combining Previous Operators

OclIncluding

lemma *OclIncluding-idem0* :
assumes $\tau \models \delta\ S$
and $\tau \models \upsilon\ i$
shows $\tau \models (S{-}{>}including(i){-}{>}including(i) \triangleq (S{-}{>}including(i)))$
$\langle proof \rangle$

theorem *OclIncluding-idem*[*simp*,*code-unfold*]: $((S :: ('\mathfrak{A},'a{::}null)Set){-}{>}including(i){-}{>}including(i) = (S{-}{>}including(i)))$
$\langle proof \rangle$

OclExcluding

lemma *OclExcluding-idem0* :
assumes $\tau \models \delta\ S$
and $\tau \models \upsilon\ i$
shows $\tau \models (S{-}{>}excluding(i){-}{>}excluding(i) \triangleq (S{-}{>}excluding(i)))$
$\langle proof \rangle$

theorem *OclExcluding-idem*[*simp*,*code-unfold*]: $((S{-}{>}excluding(i)){-}{>}excluding(i)) = (S{-}{>}excluding(i))$
$\langle proof \rangle$

OclIncludes

lemma *OclIncludes-any*[*simp*,*code-unfold*]:
$X{-}{>}includes(X{-}{>}any()) = (if\ \ \delta\ X\ then$
$if\ \ \delta\ (X{-}{>}size())\ then\ not(X{-}{>}isEmpty())$
$else\ X{-}{>}includes(null)\ endif$
$else\ invalid\ endif)$
$\langle proof \rangle$

OclSize

lemma [*simp*,*code-unfold*]: $\delta\ (Set\{\}\ {-}{>}size()) = true$
$\langle proof \rangle$

lemma [*simp*,*code-unfold*]: $\delta\ ((X\ {-}{>}including(x))\ {-}{>}size()) = (\delta(X{-}{>}size())\ and\ \upsilon(x))$
$\langle proof \rangle$

lemma [*simp*,*code-unfold*]: $\delta\ ((X\ {-}{>}excluding(x))\ {-}{>}size()) = (\delta(X{-}{>}size())\ and\ \upsilon(x))$
$\langle proof \rangle$

lemma [*simp*]:
assumes *X-finite*: $\bigwedge\tau.\ finite\ \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil$
shows $\delta\ ((X\ {-}{>}including(x))\ {-}{>}size()) = (\delta(X)\ and\ \upsilon(x))$
$\langle proof \rangle$

OclForall

lemma *OclForall-rep-set-false*:
 assumes $\tau \models \delta\ X$
 shows $(\mathit{OclForall}\ X\ P\ \tau = \mathit{false}\ \tau) = (\exists\, x \in \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil.\ P\ (\lambda\tau.\ x)\ \tau = \mathit{false}\ \tau)$
$\langle \mathit{proof} \rangle$

lemma *OclForall-rep-set-true*:
 assumes $\tau \models \delta\ X$
 shows $(\tau \models \mathit{OclForall}\ X\ P) = (\forall\, x \in \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil.\ \tau \models P\ (\lambda\tau.\ x))$
$\langle \mathit{proof} \rangle$

lemma *OclForall-includes* :
 assumes *x-def* : $\tau \models \delta\ x$
   and *y-def* : $\tau \models \delta\ y$
  shows $(\tau \models \mathit{OclForall}\ x\ (\mathit{OclIncludes}\ y)) = (\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (x\ \tau)\rceil\rceil \subseteq \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (y\ \tau)\rceil\rceil)$
 $\langle \mathit{proof} \rangle$

lemma *OclForall-not-includes* :
 assumes *x-def* : $\tau \models \delta\ x$
   and *y-def* : $\tau \models \delta\ y$
  shows $(\mathit{OclForall}\ x\ (\mathit{OclIncludes}\ y)\ \tau = \mathit{false}\ \tau) = (\neg\ \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (x\ \tau)\rceil\rceil \subseteq \lceil\lceil \mathit{Rep\text{-}Set}_{base}$
$(y\ \tau)\rceil\rceil)$
 $\langle \mathit{proof} \rangle$

lemma *OclForall-iterate*:
 assumes *S-finite*: $\mathit{finite}\ \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (S\ \tau)\rceil\rceil$
  shows $S{-}{>}\mathit{forAll}(x\ |\ P\ x)\ \tau = (S{-}{>}\mathit{iterate}(x;\ \mathit{acc} = \mathit{true}\ |\ \mathit{acc}\ \mathit{and}\ P\ x))\ \tau$
$\langle \mathit{proof} \rangle$

lemma *OclForall-cong*:
 assumes $\bigwedge x.\ x \in \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil \Longrightarrow \tau \models P\ (\lambda\tau.\ x) \Longrightarrow \tau \models Q\ (\lambda\tau.\ x)$
 assumes *P*: $\tau \models \mathit{OclForall}\ X\ P$
 shows $\tau \models \mathit{OclForall}\ X\ Q$
$\langle \mathit{proof} \rangle$

lemma *OclForall-cong'*:
 assumes $\bigwedge x.\ x \in \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (X\ \tau)\rceil\rceil \Longrightarrow \tau \models P\ (\lambda\tau.\ x) \Longrightarrow \tau \models Q\ (\lambda\tau.\ x) \Longrightarrow \tau \models R$
$(\lambda\tau.\ x)$
 assumes *P*: $\tau \models \mathit{OclForall}\ X\ P$
 assumes *Q*: $\tau \models \mathit{OclForall}\ X\ Q$
 shows $\tau \models \mathit{OclForall}\ X\ R$
$\langle \mathit{proof} \rangle$

Strict Equality

lemma $\mathit{StrictRefEq}_{Set}$*-defined* :
 assumes *x-def*: $\tau \models \delta\ x$
 assumes *y-def*: $\tau \models \delta\ y$
 shows $((x{::}('\mathfrak{A},'\alpha{::}\mathit{null})\mathit{Set}) \doteq y)\ \tau =$
   $(x{-}{>}\mathit{forAll}(z|\ y{-}{>}\mathit{includes}(z))\ \mathit{and}\ (y{-}{>}\mathit{forAll}(z|\ x{-}{>}\mathit{includes}(z))))\ \tau$

$\langle proof \rangle$

lemma *StrictRefEq$_{Set}$-exec[simp,code-unfold]* :
$((x::('\mathfrak{A},'\alpha::null)Set) \doteq y) =$
$(if\ \delta\ x\ then\ (if\ \delta\ y$
$then\ ((x{-}{>}forAll(z|\ y{-}{>}includes(z))\ and\ (y{-}{>}forAll(z|\ x{-}{>}includes(z)))))$
$else\ if\ \upsilon\ y$
$then\ false\ (*\ x'{-}{>}includes = null\ *)$
$else\ invalid$
$endif$
$endif)$
$else\ if\ \upsilon\ x\ (*\ null = ???\ *)$
$then\ if\ \upsilon\ y\ then\ not(\delta\ y)\ else\ invalid\ endif$
$else\ invalid$
$endif$
$endif)$
$\langle proof \rangle$

lemma *StrictRefEq$_{Set}$-L-subst1* : $cp\ P \Longrightarrow \tau \models \upsilon\ x \Longrightarrow \tau \models \upsilon\ y \Longrightarrow \tau \models \upsilon\ P\ x \Longrightarrow \tau \models \upsilon\ P\ y \Longrightarrow$
$\tau \models (x::('\mathfrak{A},'\alpha::null)Set) \doteq y \Longrightarrow \tau \models (P\ x\ ::('\mathfrak{A},'\alpha::null)Set) \doteq P\ y$
$\langle proof \rangle$

lemma *OclIncluding-cong'* :
shows $\tau \models \delta\ s \Longrightarrow \tau \models \delta\ t \Longrightarrow \tau \models \upsilon\ x \Longrightarrow$
$\tau \models ((s::('\mathfrak{A},'a::null)Set) \doteq t) \Longrightarrow \tau \models (s{-}{>}including(x) \doteq (t{-}{>}including(x)))$
$\langle proof \rangle$

lemma *OclIncluding-cong* : $\bigwedge(s::('\mathfrak{A},'a::null)Set)\ t\ x\ y\ \tau.\ \tau \models \delta\ t \Longrightarrow \tau \models \upsilon\ y \Longrightarrow$
$\tau \models s \doteq t \Longrightarrow x = y \Longrightarrow \tau \models s{-}{>}including(x) \doteq (t{-}{>}including(y))$
$\langle proof \rangle$

lemma *const-StrictRefEq$_{Set}$-empty* : $const\ X \Longrightarrow\ const\ (X \doteq Set\{\})$
$\langle proof \rangle$

lemma *const-StrictRefEq$_{Set}$-including* :
$const\ a \Longrightarrow const\ S \Longrightarrow const\ X \Longrightarrow\ const\ (X \doteq S{-}{>}including(a))$
$\langle proof \rangle$

5.8.6. Test Statements

Assert $(\tau \models (Set\{\lambda\text{-}.\ \lfloor\lfloor x\rfloor\rfloor\} \doteq Set\{\lambda\text{-}.\ \lfloor\lfloor x\rfloor\rfloor\}))$
Assert $(\tau \models (Set\{\lambda\text{-}.\ \lfloor x\rfloor\} \doteq Set\{\lambda\text{-}.\ \lfloor x\rfloor\}))$

end

theory *OCL-collection-type-Sequence*
imports *OCL-basic-type-Integer*
begin

5.9. Collection Type Sequence: Operations

5.9.1. Constants: mtSequence

definition $mtSequence ::('\mathfrak{A},'\alpha::null)\ Sequence\ \ (Sequence\{\})$
where $\quad Sequence\{\} \equiv (\lambda\ \tau.\ \ Abs\text{-}Sequence_{base}\ \lfloor\lfloor[]::'\alpha\ list\rfloor\rfloor\ )$

declare *mtSequence-def* [*code-unfold*]

lemma *mtSequence-defined*[*simp*,*code-unfold*]:$\delta(Sequence\{\}) = true$
⟨*proof*⟩

lemma *mtSequence-valid*[*simp*,*code-unfold*]:$\upsilon(Sequence\{\}) = true$
⟨*proof*⟩

lemma *mtSequence-rep-set*: $\lceil\lceil Rep\text{-}Sequence_{base}\ (Sequence\{\}\ \tau)\rceil\rceil = []$
⟨*proof*⟩

lemma [*simp*,*code-unfold*]: *const Sequence*{}
⟨*proof*⟩

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

lemmas $cp\text{-}intro''_{Sequence}$[*intro!*,*simp*,*code-unfold*] = *cp-intro′*

Properties of Sequence Type:

Every element in a defined sequence is valid.

lemma *Sequence-inv-lemma*: $\tau \models (\delta\ X) \Longrightarrow \forall x{\in}set\ \lceil\lceil Rep\text{-}Sequence_{base}\ (X\ \tau)\rceil\rceil.\ x \neq bot$
⟨*proof*⟩

5.9.2. Strict Equality

Definition

After the part of foundational operations on sets, we detail here equality on sets. Strong equality is inherited from the OCL core, but we have to consider the case of the strict equality. We decide to overload strict equality in the same way we do for other value's in OCL:

defs $StrictRefEq_{Sequence}$ [*code-unfold*]:
$$((x::('\mathfrak{A},'\alpha::null)Sequence) \doteq y) \equiv (\lambda\ \tau.\ if\ \ (\upsilon\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau\ \ then\ (x \triangleq y)\tau\ \ else\ invalid\ \tau)$$

Property proof in terms of *profile-bin3*

interpretation $StrictRefEq_{Sequence}$: *profile-bin3* λ x y. $(x::('\mathfrak{A},'\alpha::null)Sequence) \doteq y$
⟨*proof*⟩

5.9.3. Standard Operations

Definition: including

definition *OclIncluding* :: $[('\mathfrak{A},'\alpha::null)$ *Sequence*,$('\mathfrak{A},'\alpha)$ *val*$] \Rightarrow ('\mathfrak{A},'\alpha)$ *Sequence*
where *OclIncluding* x y = $(\lambda\ \tau.$ *if* $(\delta\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau$
then $Abs\text{-}Sequence_{base}$ $\lfloor\lfloor$ $\lceil\lceil Rep\text{-}Sequence_{base}\ (x\ \tau)\rceil\rceil$ @ $[y\ \tau]$ $\rfloor\rfloor$
else invalid τ)
notation *OclIncluding* $(\text{-->}including_{Seq}'(\text{-}'))$

interpretation *OclIncluding* :
profile-bin2 OclIncluding $\lambda x\ y.\ Abs\text{-}Sequence_{base}\lfloor\lfloor\lceil\lceil Rep\text{-}Sequence_{base}\ x\rceil\rceil\ @\ [y]\rfloor\rfloor$
⟨*proof*⟩

syntax
-*OclFinsequence* :: *args* => $('\mathfrak{A},'a::null)$ *Sequence* (*Sequence*{(-)})
translations
Sequence{x, *xs*} == *CONST OclIncluding* (*Sequence*{*xs*}) x
Sequence{x} == *CONST OclIncluding* (*Sequence*{}) x

typ *int*
typ *num*

Definition: excluding

Definition: union

Definition: append

identical to including

Definition: prepend

Definition: subSequence

Definition: at

Definition: first

Definition: last

Definition: asSet

instantiation $Sequence_{base}$ :: (*equal*)*equal*
begin
definition *HOL.equal* k l $\longleftrightarrow$ $(k::('a::equal)Sequence_{base}) = l$
instance ⟨*proof*⟩
end

lemma *equal-Sequence$_{base}$-code* [*code*]:
$HOL.equal\ k\ (l{::}('a{::}\{equal,null\})Sequence_{base}) \longleftrightarrow Rep\text{-}Sequence_{base}\ k = Rep\text{-}Sequence_{base}\ l$
⟨*proof*⟩

5.9.4. Test Statements

Assert $(\tau \models (Sequence\{\} \doteq Sequence\{\}))$
Assert $\tau \models (Sequence\{\mathbf{1},invalid,\mathbf{2}\} \triangleq invalid)$

end

theory *OCL-lib*
imports
OCL-basic-type-Boolean
OCL-basic-type-Void
OCL-basic-type-Integer
OCL-basic-type-Real
OCL-basic-type-String

OCL-collection-type-Pair
OCL-collection-type-Set
OCL-collection-type-Sequence
begin

5.10. Miscellaneous Stuff

5.10.1. Properties on Collection Types: Strict Equality

The structure of this chapter roughly follows the structure of Chapter 10 of the OCL standard [30], which introduces the OCL Library.

5.10.2. MOVE TEXT : Collection Types

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

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

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

lemmas $cp\text{-}intro''$ $[intro!,simp,code\text{-}unfold]$ =
    $cp\text{-}intro'$

    $cp\text{-}intro''_{Set}$
    $cp\text{-}intro''_{Sequence}$

5.10.3. MOVE TEXT: Test Statements

lemma $syntax\text{-}test$: $Set\{\mathbf{2},\mathbf{1}\} = (Set\{\}{-}{>}including(\mathbf{1}){-}{>}including(\mathbf{2}))$
$\langle proof \rangle$

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 $semantic\text{-}test2$:
assumes $H{:}(Set\{\mathbf{2}\} \doteq null) = (false{::}('\mathfrak{A})Boolean)$
shows $(\tau{::}('\mathfrak{A})st) \models (Set\{Set\{\mathbf{2}\},null\}{-}{>}includes(null))$
$\langle proof \rangle$

lemma $short\text{-}cut'[simp,code\text{-}unfold]$: $(\mathbf{8} \doteq \mathbf{6}) = false$
$\langle proof \rangle$

lemma $short\text{-}cut''[simp,code\text{-}unfold]$: $(\mathbf{2} \doteq \mathbf{1}) = false$
$\langle proof \rangle$
lemma $short\text{-}cut'''[simp,code\text{-}unfold]$: $(\mathbf{1} \doteq \mathbf{2}) = false$
$\langle proof \rangle$

Elementary computations on Sets.

declare $OclSelect\text{-}body\text{-}def$ $[simp]$

Assert $\neg\ (\tau \models \upsilon(invalid{::}('\mathfrak{A},'\alpha{::}null)\ Set))$
Assert $\tau \models \upsilon(null{::}('\mathfrak{A},'\alpha{::}null)\ Set)$
Assert $\neg\ (\tau \models \delta(null{::}('\mathfrak{A},'\alpha{::}null)\ Set))$
Assert $\tau \models \upsilon(Set\{\})$
Assert $\tau \models \upsilon(Set\{Set\{\mathbf{2}\},null\})$
Assert $\tau \models \delta(Set\{Set\{\mathbf{2}\},null\})$
Assert $\tau \models (Set\{\mathbf{2},\mathbf{1}\}{-}{>}includes(\mathbf{1}))$
Assert $\neg\ (\tau \models (Set\{\mathbf{2}\}{-}{>}includes(\mathbf{1})))$
Assert $\neg\ (\tau \models (Set\{\mathbf{2},\mathbf{1}\}{-}{>}includes(null)))$
Assert $\tau \models (Set\{\mathbf{2},null\}{-}{>}includes(null))$

Assert $\tau \models (Set\{null,\mathbf{2}\}{-}{>}includes(null))$

Assert $\tau \models ((Set\{\}){-}{>}forAll(z \mid \mathbf{0} <_{int} z))$

Assert $\tau \models ((Set\{\mathbf{2},\mathbf{1}\}){-}{>}forAll(z \mid \mathbf{0} <_{int} z))$
Assert $\tau \models (\mathbf{0} <_{int} \mathbf{2})\ and\ (\mathbf{0} <_{int} \mathbf{1})$
Assert $\neg\ (\tau \models ((Set\{\mathbf{2},\mathbf{1}\}){-}{>}exists(z \mid z <_{int} \mathbf{0}\ )))$
Assert $\neg\ (\tau \models (\delta(Set\{\mathbf{2},null\}){-}{>}forAll(z \mid \mathbf{0} <_{int} z)))$
Assert $\neg\ (\tau \models ((Set\{\mathbf{2},null\}){-}{>}forAll(z \mid \mathbf{0} <_{int} z)))$
Assert $\tau \models ((Set\{\mathbf{2},null\}){-}{>}exists(z \mid \mathbf{0} <_{int} z))$

Assert $\neg\ (\tau \models (Set\{null{::}'a\ Boolean\} \doteq Set\{\}))$
Assert $\neg\ (\tau \models (Set\{null{::}'a\ Integer\} \doteq Set\{\}))$

Assert $\neg\ (\tau \models (Set\{true\} \doteq Set\{false\}))$
Assert $\neg\ (\tau \models (Set\{true,true\} \doteq Set\{false\}))$
Assert $\neg\ (\tau \models (Set\{\mathbf{2}\} \doteq Set\{\mathbf{1}\}))$
Assert $\tau \models (Set\{\mathbf{2},null,\mathbf{2}\} \doteq Set\{null,\mathbf{2}\})$
Assert $\tau \models (Set\{\mathbf{1},null,\mathbf{2}\} <> Set\{null,\mathbf{2}\})$
Assert $\tau \models (Set\{Set\{\mathbf{2},null\}\} \doteq Set\{Set\{null,\mathbf{2}\}\})$
Assert $\tau \models (Set\{Set\{\mathbf{2},null\}\} <> Set\{Set\{null,\mathbf{2}\},null\})$
Assert $\tau \models (Set\{null\}{-}{>}select(x \mid not\ x) \doteq Set\{null\})$
Assert $\tau \models (Set\{null\}{-}{>}reject(x \mid not\ x) \doteq Set\{null\})$

lemma $const\ (Set\{Set\{\mathbf{2},null\},\ invalid\})\ \langle proof\rangle$

end

6. Formalization III: UML/OCL constructs: State Operations and Objects

theory *OCL-state*
imports *OCL-lib*
begin

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

6.1. 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.

6.1.1. Fundamental Properties on Objects: Core Referential Equality

Definition

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

definition $StrictRefEq_{Object}$ :: $('\mathfrak{A},'a{::}\{object,null\})val \Rightarrow ('\mathfrak{A},'a)val \Rightarrow ('\mathfrak{A})Boolean$
where $StrictRefEq_{Object}\ x\ y$
$\equiv \lambda\ \tau.\ if\ (\upsilon\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau$
$then\ if\ x\ \tau = null \vee y\ \tau = null$
$then\ \lfloor\lfloor x\ \tau = null \wedge y\ \tau = null\rfloor\rfloor$
$else\ \lfloor\lfloor(oid\text{-}of\ (x\ \tau)) = (oid\text{-}of\ (y\ \tau))\ \rfloor\rfloor$
$else\ invalid\ \tau$

Strictness and context passing

lemma $StrictRefEq_{Object}$-*strict1*[*simp*,*code-unfold*] :
$(StrictRefEq_{Object}\ x\ invalid) = invalid$
⟨*proof*⟩

lemma $StrictRefEq_{Object}$-*strict2*[*simp*,*code-unfold*] :
$(StrictRefEq_{Object}\ invalid\ x) = invalid$
⟨*proof*⟩

lemma *cp*-$StrictRefEq_{Object}$:
$(StrictRefEq_{Object}\ x\ y\ \tau) = (StrictRefEq_{Object}\ (\lambda\text{-}.\ x\ \tau)\ (\lambda\text{-}.\ y\ \tau))\ \tau$

⟨*proof*⟩

lemmas *cp0-StrictRefEq*$_{Object}$= *cp-StrictRefEq*$_{Object}$[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]],
of StrictRefEq$_{Object}$]]

lemmas *cp-intro*''[*intro*!,*simp*,*code-unfold*] =
cp-intro''
cp-StrictRefEq$_{Object}$[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]],
of StrictRefEq$_{Object}$]]

6.1.2. Logic and Algebraic Layer on Object

Validity and Definedness Properties

We derive the usual laws on definedness for (generic) object equality:

lemma *StrictRefEq*$_{Object}$*-defargs*:
$\tau \models (\mathit{StrictRefEq}_{Object}\ x\ (y{::}('\mathfrak{A},'a{::}\{null,object\})val)) \Longrightarrow (\tau \models(\upsilon\ x)) \wedge (\tau \models(\upsilon\ y))$
⟨*proof*⟩

lemma *defined-StrictRefEq*$_{Object}$*-I*:
assumes *val-x* : $\tau \models \upsilon\ x$
assumes *val-x* : $\tau \models \upsilon\ y$
shows $\tau \models \delta\ (\mathit{StrictRefEq}_{Object}\ x\ y)$
⟨*proof*⟩

lemma *StrictRefEq*$_{Object}$*-def-homo* :
$\delta(\mathit{StrictRefEq}_{Object}\ x\ (y{::}('\mathfrak{A},'a{::}\{null,object\})val)) = ((\upsilon\ x)\ \mathit{and}\ (\upsilon\ y))$
⟨*proof*⟩

Symmetry

lemma *StrictRefEq*$_{Object}$*-sym* :
assumes *x-val* : $\tau \models \upsilon\ x$
shows $\tau \models \mathit{StrictRefEq}_{Object}\ x\ x$
⟨*proof*⟩

Behavior vs StrongEq

It remains to clarify the role of the state invariant $\mathrm{inv}_\sigma(\sigma)$ mentioned above that states the condition that there is a "one-to-one" correspondence between object representations and oid's: $\forall \mathit{oid} \in \mathrm{dom}\,\sigma.\ \mathit{oid} = \mathrm{OidOf}\ \ulcorner\sigma(\mathit{oid})\urcorner$. This condition is also mentioned in [30, Annex A] and goes back to Richters [32]; however, we state this condition as an invariant on states rather than a global axiom. It can, therefore, not be taken for granted that an oid makes sense both in pre- and post-states of OCL expressions.

We capture this invariant in the predicate WFF :

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

where $WFF\ \tau = ((\forall\ x \in ran(heap(fst\ \tau)).\ \lceil heap(fst\ \tau)\ (oid\text{-}of\ x)\rceil = x) \wedge$
$(\forall\ x \in ran(heap(snd\ \tau)).\ \lceil heap(snd\ \tau)\ (oid\text{-}of\ x)\rceil = x))$

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 *StrictRefEq*$_{Object}$*-vs-StrongEq*:
assumes *WFF*: $WFF\ \tau$
and *valid-x*: $\tau \models (\upsilon\ x)$
and *valid-y*: $\tau \models (\upsilon\ y)$
and *x-present-pre*: $x\ \tau \in ran\ (heap(fst\ \tau))$
and *y-present-pre*: $y\ \tau \in ran\ (heap(fst\ \tau))$
and *x-present-post*:$x\ \tau \in ran\ (heap(snd\ \tau))$
and *y-present-post*:$y\ \tau \in ran\ (heap(snd\ \tau))$

shows $(\tau \models (StrictRefEq_{Object}\ x\ y)) = (\tau \models (x \triangleq y))$
$\langle proof \rangle$

theorem *StrictRefEq*$_{Object}$*-vs-StrongEq'*:
assumes *WFF*: $WFF\ \tau$
and *valid-x*: $\tau \models (\upsilon\ (x :: ('\mathfrak{A}::object,'\alpha::\{null,object\})val))$
and *valid-y*: $\tau \models (\upsilon\ y)$
and *oid-preserve*: $\bigwedge x.\ x \in ran\ (heap(fst\ \tau)) \vee x \in ran\ (heap(snd\ \tau)) \Longrightarrow$
$H\ x \neq \bot \Longrightarrow oid\text{-}of\ (H\ x) = oid\text{-}of\ x$
and *xy-together*: $x\ \tau \in H\ `\ ran\ (heap(fst\ \tau)) \wedge y\ \tau \in H\ `\ ran\ (heap(fst\ \tau)) \vee$
$x\ \tau \in H\ `\ ran\ (heap(snd\ \tau)) \wedge y\ \tau \in H\ `\ ran\ (heap(snd\ \tau))$

shows $(\tau \models (StrictRefEq_{Object}\ x\ y)) = (\tau \models (x \triangleq y))$
$\langle proof \rangle$

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.

6.2. Operations on Object

6.2.1. Initial States (for testing and code generation)

definition $\tau_0 :: ('\mathfrak{A})st$
where $\tau_0 \equiv (\llparenthesis heap{=}Map.empty,\ assocs\ =\ Map.empty\rrparenthesis,$
$\llparenthesis heap{=}Map.empty,\ assocs\ =\ Map.empty\rrparenthesis)$

6.2.2. 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 $OclAllInstances\text{-}generic :: (('\mathfrak{A}::object)\ st \Rightarrow '\mathfrak{A}\ state) \Rightarrow ('\mathfrak{A}::object \rightharpoonup '\alpha) \Rightarrow$
$('\mathfrak{A},\ '\alpha\ option\ option)\ Set$
where $OclAllInstances\text{-}generic\ fst\text{-}snd\ H =$
$(\lambda\tau.\ Abs\text{-}Set_{base}\ \lfloor\lfloor\ Some\ `\ ((H\ `\ ran\ (heap\ (fst\text{-}snd\ \tau))) - \{\ None\ \})\ \rfloor\rfloor)$

lemma *OclAllInstances-generic-defined*: $\tau \models \delta\ (OclAllInstances\text{-}generic\ pre\text{-}post\ H)$
$\langle proof \rangle$

lemma *OclAllInstances-generic-init-empty*:
assumes $[simp]$: $\bigwedge x.\ pre\text{-}post\ (x,\ x) = x$
shows $\tau_0 \models OclAllInstances\text{-}generic\ pre\text{-}post\ H \triangleq Set\{\}$
$\langle proof \rangle$

lemma *represented-generic-objects-nonnull*:
assumes A: $\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ (H::('\mathfrak{A}::object \rightharpoonup '\alpha)))\ {-}{>}includes(x))$
shows $\tau \models not(x \triangleq null)$
$\langle proof \rangle$

lemma *represented-generic-objects-defined*:
assumes A: $\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ (H::('\mathfrak{A}::object \rightharpoonup '\alpha)))\ {-}{>}includes(x))$
shows $\tau \models \delta\ (OclAllInstances\text{-}generic\ pre\text{-}post\ H) \wedge \tau \models \delta\ x$
$\langle proof \rangle$

One way to establish the actual presence of an object representation in a state is:

lemma *represented-generic-objects-in-state*:
assumes A: $\tau \models (OclAllInstances\text{-}generic\ pre\text{-}post\ H){-}{>}includes(x)$
shows $x\ \tau \in (Some\ o\ H)\ `\ ran\ (heap(pre\text{-}post\ \tau))$
$\langle proof \rangle$

lemma *state-update-vs-allInstances-generic-empty*:
assumes $[simp]$: $\bigwedge a.\ pre\text{-}post\ (mk\ a) = a$
shows $(mk\ \llparenthesis heap{=}empty,\ assocs{=}A\rrparenthesis) \models OclAllInstances\text{-}generic\ pre\text{-}post\ Type \doteq Set\{\}$
$\langle proof \rangle$

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*]: $\bigwedge a.\ pre\text{-}post\ (mk\ a) = a$
assumes $\bigwedge x.\ \sigma'\ oid = Some\ x \Longrightarrow x = Object$
and $Type\ Object \neq None$
shows $(OclAllInstances\text{-}generic\ pre\text{-}post\ Type)$
$(mk\ (\!|heap{=}\sigma'(oid{\mapsto}Object),\ assocs{=}A|\!))$
$=$
$((OclAllInstances\text{-}generic\ pre\text{-}post\ Type){-}{>}including(\lambda\ \text{-}.\ \lfloor\lfloor\ drop\ (Type\ Object)\ \rfloor\rfloor))$
$(mk\ (\!|heap{=}\sigma',assocs{=}A|\!))$
⟨*proof*⟩

lemma *state-update-vs-allInstances-generic-including*:
assumes [*simp*]: $\bigwedge a.\ pre\text{-}post\ (mk\ a) = a$
assumes $\bigwedge x.\ \sigma'\ oid = Some\ x \Longrightarrow x = Object$
and $Type\ Object \neq None$
shows $(OclAllInstances\text{-}generic\ pre\text{-}post\ Type)$
$(mk\ (\!|heap{=}\sigma'(oid{\mapsto}Object),\ assocs{=}A|\!))$
$=$
$((\lambda\text{-}.\ (OclAllInstances\text{-}generic\ pre\text{-}post\ Type)$
$(mk\ (\!|heap{=}\sigma',\ assocs{=}A|\!))){-}{>}including(\lambda\ \text{-}.\ \lfloor\lfloor\ drop\ (Type\ Object)\ \rfloor\rfloor))$
$(mk\ (\!|heap{=}\sigma'(oid{\mapsto}Object),\ assocs{=}A|\!))$
⟨*proof*⟩

lemma *state-update-vs-allInstances-generic-noincluding'*:
assumes [*simp*]: $\bigwedge a.\ pre\text{-}post\ (mk\ a) = a$
assumes $\bigwedge x.\ \sigma'\ oid = Some\ x \Longrightarrow x = Object$
and $Type\ Object = None$
shows $(OclAllInstances\text{-}generic\ pre\text{-}post\ Type)$
$(mk\ (\!|heap{=}\sigma'(oid{\mapsto}Object),\ assocs{=}A|\!))$
$=$
$(OclAllInstances\text{-}generic\ pre\text{-}post\ Type)$
$(mk\ (\!|heap{=}\sigma',\ assocs{=}A|\!))$
⟨*proof*⟩

theorem *state-update-vs-allInstances-generic-ntc*:
assumes [*simp*]: $\bigwedge a.\ pre\text{-}post\ (mk\ a) = a$
assumes *oid-def*: $oid {\notin} dom\ \sigma'$
and *non-type-conform*: $Type\ Object = None$
and *cp-ctxt*: $cp\ P$
and *const-ctxt*: $\bigwedge X.\ const\ X \Longrightarrow const\ (P\ X)$

shows $(mk\ (\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!) \models P\ (OclAllInstances\text{-}generic\ pre\text{-}post\ Type)) =$
$(mk\ (\!|heap{=}\sigma',\ assocs{=}A|\!) \qquad\qquad \models P\ (OclAllInstances\text{-}generic\ pre\text{-}post\ Type))$
(**is** $(?\tau \models P\ ?\varphi) = (?\tau' \models P\ ?\varphi)$)
$\langle proof \rangle$

theorem *state-update-vs-allInstances-generic-tc*:
assumes [*simp*]: $\bigwedge a.\ pre\text{-}post\ (mk\ a) = a$
assumes *oid-def*: $oid{\notin}dom\ \sigma'$
and *type-conform*: $Type\ Object \neq None$
and *cp-ctxt*: $cp\ P$
and *const-ctxt*: $\bigwedge X.\ const\ X \Longrightarrow const\ (P\ X)$
shows $(mk\ (\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!) \models P\ (OclAllInstances\text{-}generic\ pre\text{-}post\ Type)) =$
$(mk\ (\!|heap{=}\sigma',\ assocs{=}A|\!) \qquad\qquad \models P\ ((OclAllInstances\text{-}generic\ pre\text{-}post\ Type)$
$\qquad\qquad\qquad\qquad {-}{>}including(\lambda\ \text{-}.\ \lfloor(Type\ Object)\rfloor)))$
(**is** $(?\tau \models P\ ?\varphi) = (?\tau' \models P\ ?\varphi')$)
$\langle proof \rangle$

declare *OclAllInstances-generic-def* [*simp*]

OclAllInstances (@post)

definition $OclAllInstances\text{-}at\text{-}post :: ('\mathfrak{A} :: object \rightharpoonup '\alpha) \Rightarrow ('\mathfrak{A}, '\alpha\ option\ option)\ Set$
$(\text{-}\ .allInstances'('))$
where $OclAllInstances\text{-}at\text{-}post = \ OclAllInstances\text{-}generic\ snd$

lemma *OclAllInstances-at-post-defined*: $\tau \models \delta\ (H\ .allInstances())$
$\langle proof \rangle$

lemma $\tau_0 \models H\ .allInstances() \triangleq Set\{\}$
$\langle proof \rangle$

lemma *represented-at-post-objects-nonnull*:
assumes A: $\tau \models (((H{::}('\mathfrak{A}{::}object \rightharpoonup '\alpha)).allInstances())\ {-}{>}includes(x))$
shows $\tau \models not(x \triangleq null)$
$\langle proof \rangle$

lemma *represented-at-post-objects-defined*:
assumes A: $\tau \models (((H{::}('\mathfrak{A}{::}object \rightharpoonup '\alpha)).allInstances())\ {-}{>}includes(x))$
shows $\tau \models \delta\ (H\ .allInstances()) \land \tau \models \delta\ x$
$\langle proof \rangle$

One way to establish the actual presence of an object representation in a state is:

lemma
assumes A: $\tau \models H\ .allInstances(){-}{>}includes(x)$
shows $x\ \tau \in (Some\ o\ H)\ `\ ran\ (heap(snd\ \tau))$
$\langle proof \rangle$

lemma *state-update-vs-allInstances-at-post-empty*:

shows $(\sigma, (\!|heap{=}empty,\ assocs{=}A|\!)) \models Type\ .allInstances() \doteq Set\{\}$
$\langle proof \rangle$

Here comes a couple of operational rules that allow to infer the value of oclAllInstances from the context τ. These rules are a special-case in the sense that they are the only rules that relate statements with *different* τ's. For that reason, new concepts like "constant contexts P" are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

lemma *state-update-vs-allInstances-at-post-including'*:
assumes $\bigwedge x.\ \sigma'\ oid = Some\ x \Longrightarrow x = Object$
and $Type\ Object \neq None$
shows $(Type\ .allInstances())$
$(\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object),\ assocs{=}A|\!))$
$=$
$((Type\ .allInstances()){-}{>}including(\lambda\ \text{-}.\ \lfloor\lfloor\ drop\ (Type\ Object)\ \rfloor\rfloor))$
$(\sigma, (\!|heap{=}\sigma',assocs{=}A|\!))$
$\langle proof \rangle$

lemma *state-update-vs-allInstances-at-post-including*:
assumes $\bigwedge x.\ \sigma'\ oid = Some\ x \Longrightarrow x = Object$
and $Type\ Object \neq None$
shows $(Type\ .allInstances())$
$(\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object),\ assocs{=}A|\!))$
$=$
$((\lambda\text{-}.\ (Type\ .allInstances())$
$(\sigma, (\!|heap{=}\sigma',\ assocs{=}A|\!))){-}{>}including(\lambda\ \text{-}.\ \lfloor\lfloor\ drop\ (Type\ Object)\ \rfloor\rfloor))$
$(\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object),\ assocs{=}A|\!))$
$\langle proof \rangle$

lemma *state-update-vs-allInstances-at-post-noincluding'*:
assumes $\bigwedge x.\ \sigma'\ oid = Some\ x \Longrightarrow x = Object$
and $Type\ Object = None$
shows $(Type\ .allInstances())$
$(\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object),\ assocs{=}A|\!))$
$=$
$(Type\ .allInstances())$
$(\sigma, (\!|heap{=}\sigma',\ assocs{=}A|\!))$
$\langle proof \rangle$

theorem *state-update-vs-allInstances-at-post-ntc*:
assumes *oid-def*: $oid{\notin}dom\ \sigma'$
and *non-type-conform*: $Type\ Object = None$
and *cp-ctxt*: $cp\ P$
and *const-ctxt*: $\bigwedge X.\ const\ X \Longrightarrow const\ (P\ X)$
shows $((\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object),assocs{=}A|\!)) \models (P(Type\ .allInstances()))) =$

$((\sigma, (\!|heap{=}\sigma', assocs{=}A|\!)) \qquad \models (P(\mathit{Type} .allInstances())))$
⟨proof⟩

theorem *state-update-vs-allInstances-at-post-tc*:
assumes *oid-def*: $oid \notin dom\ \sigma'$
and *type-conform*: $\mathit{Type\ Object} \neq \mathit{None}$
and *cp-ctxt*: $cp\ P$
and *const-ctxt*: $\bigwedge X.\ const\ X \Longrightarrow const\ (P\ X)$
shows $((\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!)) \models (P(\mathit{Type} .allInstances()))) =$
$((\sigma, (\!|heap{=}\sigma', assocs{=}A|\!)) \qquad \models (P((\mathit{Type} .allInstances())$
$\quad {-}{>}including(\lambda\ \text{-}.\ \lfloor(\mathit{Type\ Object})\rfloor))))$
⟨proof⟩

OclAllInstances (@pre)

definition *OclAllInstances-at-pre* $:: ('\mathfrak{A} :: object \rightharpoonup '\alpha) \Rightarrow ('\mathfrak{A}, '\alpha\ option\ option)\ Set$
$(\text{-}\ .allInstances@pre'(')')$
where *OclAllInstances-at-pre* = *OclAllInstances-generic fst*

lemma *OclAllInstances-at-pre-defined*: $\tau \models \delta\ (H\ .allInstances@pre())$
⟨proof⟩

lemma $\tau_0 \models H\ .allInstances@pre() \triangleq Set\{\}$
⟨proof⟩

lemma *represented-at-pre-objects-nonnull*:
assumes A: $\tau \models (((H::('\mathfrak{A}::object \rightharpoonup '\alpha)).allInstances@pre())\ {-}{>}includes(x))$
shows $\tau \models not(x \triangleq null)$
⟨proof⟩

lemma *represented-at-pre-objects-defined*:
assumes A: $\tau \models (((H::('\mathfrak{A}::object \rightharpoonup '\alpha)).allInstances@pre())\ {-}{>}includes(x))$
shows $\tau \models \delta\ (H\ .allInstances@pre()) \wedge \tau \models \delta\ x$
⟨proof⟩

One way to establish the actual presence of an object representation in a state is:

lemma
assumes A: $\tau \models H\ .allInstances@pre(){-}{>}includes(x)$
shows $x\ \tau \in (Some\ o\ H)\ `\ ran\ (heap(fst\ \tau))$
⟨proof⟩

lemma *state-update-vs-allInstances-at-pre-empty*:
shows $((\!|heap{=}empty, assocs{=}A|\!), \sigma) \models \mathit{Type}\ .allInstances@pre() \doteq Set\{\}$
⟨proof⟩

Here comes a couple of operational rules that allow to infer the value of oclAllInstances@pre from the context τ. These rules are a special-case in the sense that they are the only rules that relate statements with *different* τ's. For that reason, new concepts like "constant contexts P" are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

lemma *state-update-vs-allInstances-at-pre-including'*:
assumes $\bigwedge x.\ \sigma'\ oid = Some\ x \Longrightarrow x = Object$
  and $Type\ Object \neq None$
 shows $(Type\ .allInstances@pre())$
 $(\lparr heap{=}\sigma'(oid{\mapsto}Object),\ assocs{=}A\rparr,\ \sigma)$
 $=$
 $((Type\ .allInstances@pre()){-}{>}including(\lambda\ \text{-}.\ \lfloor\lfloor\ drop\ (Type\ Object)\ \rfloor\rfloor))$
 $(\lparr heap{=}\sigma',assocs{=}A\rparr,\ \sigma)$
$\langle proof \rangle$

lemma *state-update-vs-allInstances-at-pre-including*:
assumes $\bigwedge x.\ \sigma'\ oid = Some\ x \Longrightarrow x = Object$
  and $Type\ Object \neq None$
shows $(Type\ .allInstances@pre())$
 $(\lparr heap{=}\sigma'(oid{\mapsto}Object),\ assocs{=}A\rparr,\ \sigma)$
 $=$
 $((\lambda\text{-}.\ (Type\ .allInstances@pre())$
 $(\lparr heap{=}\sigma',\ assocs{=}A\rparr,\ \sigma)){-}{>}including(\lambda\ \text{-}.\ \lfloor\lfloor\ drop\ (Type\ Object)\ \rfloor\rfloor))$
 $(\lparr heap{=}\sigma'(oid{\mapsto}Object),\ assocs{=}A\rparr,\ \sigma)$
$\langle proof \rangle$

lemma *state-update-vs-allInstances-at-pre-noincluding'*:
assumes $\bigwedge x.\ \sigma'\ oid = Some\ x \Longrightarrow x = Object$
  and $Type\ Object = None$
 shows $(Type\ .allInstances@pre())$
 $(\lparr heap{=}\sigma'(oid{\mapsto}Object),\ assocs{=}A\rparr,\ \sigma)$
 $=$
 $(Type\ .allInstances@pre())$
 $(\lparr heap{=}\sigma',\ assocs{=}A\rparr,\ \sigma)$
$\langle proof \rangle$

theorem *state-update-vs-allInstances-at-pre-ntc*:
assumes *oid-def*: $oid{\notin}dom\ \sigma'$
and *non-type-conform*: $Type\ Object = None$
and *cp-ctxt*: $cp\ P$
and *const-ctxt*: $\bigwedge X.\ const\ X \Longrightarrow const\ (P\ X)$
shows $(((\lparr heap{=}\sigma'(oid{\mapsto}Object),assocs{=}A\rparr,\ \sigma) \models (P(Type\ .allInstances@pre()))) =$
 $(((\lparr heap{=}\sigma',\ assocs{=}A\rparr,\ \sigma) \models (P(Type\ .allInstances@pre())))$
$\langle proof \rangle$

theorem *state-update-vs-allInstances-at-pre-tc*:
assumes *oid-def*: $oid{\notin}dom\ \sigma'$
and *type-conform*: $Type\ Object \neq None$

and *cp-ctxt*: $cp\ P$
and *const-ctxt*: $\bigwedge X.\ const\ X \Longrightarrow const\ (P\ X)$
shows $(((\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!), \sigma) \models (P(Type\ .allInstances@pre()))) =$
$(((\!|heap{=}\sigma',\ assocs{=}A|\!), \sigma) \models (P((Type\ .allInstances@pre())$
$->including(\lambda\ \text{-}.\ \lfloor(Type\ Object)\rfloor))))$
$\langle proof \rangle$

@post or @pre

theorem *StrictRefEq$_{Object}$-vs-StrongEq''*:
assumes *WFF*: $WFF\ \tau$
and *valid-x*: $\tau \models(\upsilon\ (x :: ('\mathfrak{A}::object,'\alpha::object\ option\ option)val))$
and *valid-y*: $\tau \models(\upsilon\ y)$
and *oid-preserve*: $\bigwedge x.\ x \in ran\ (heap(fst\ \tau)) \lor x \in ran\ (heap(snd\ \tau)) \Longrightarrow$
$oid\text{-}of\ (H\ x) = oid\text{-}of\ x$
and *xy-together*: $\tau \models ((H\ .allInstances()->includes(x)\ and\ H\ .allInstances()->includes(y))$
or
$(H\ .allInstances@pre()->includes(x)\ and\ H\ .allInstances@pre()->includes(y)))$

shows $(\tau \models (StrictRefEq_{Object}\ x\ y)) = (\tau \models (x \triangleq y))$
$\langle proof \rangle$

6.2.3. OclIsNew, OclIsDeleted, OclIsMaintained, OclIsAbsent

definition *OclIsNew*:: $('\mathfrak{A},\ '\alpha::\{null,object\})val \Rightarrow ('\mathfrak{A})Boolean \quad ((\text{-}).oclIsNew'('))$
where $X\ .oclIsNew() \equiv (\lambda\tau\ .\ if\ (\delta\ X)\ \tau = true\ \tau$
$then\ \lfloor\lfloor oid\text{-}of\ (X\ \tau) \notin dom(heap(fst\ \tau)) \land$
$oid\text{-}of\ (X\ \tau) \in dom(heap(snd\ \tau))\rfloor\rfloor$
$else\ invalid\ \tau)$

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*:: $('\mathfrak{A},\ '\alpha::\{null,object\})val \Rightarrow ('\mathfrak{A})Boolean \quad ((\text{-}).oclIsDeleted'('))$
where $X\ .oclIsDeleted() \equiv (\lambda\tau\ .\ if\ (\delta\ X)\ \tau = true\ \tau$
$then\ \lfloor\lfloor oid\text{-}of\ (X\ \tau) \in dom(heap(fst\ \tau)) \land$
$oid\text{-}of\ (X\ \tau) \notin dom(heap(snd\ \tau))\rfloor\rfloor$
$else\ invalid\ \tau)$

definition *OclIsMaintained*:: $('\mathfrak{A},\ '\alpha::\{null,object\})val \Rightarrow ('\mathfrak{A})Boolean((\text{-}).oclIsMaintained'('))$
where $X\ .oclIsMaintained() \equiv (\lambda\tau\ .\ if\ (\delta\ X)\ \tau = true\ \tau$
$then\ \lfloor\lfloor oid\text{-}of\ (X\ \tau) \in dom(heap(fst\ \tau)) \land$
$oid\text{-}of\ (X\ \tau) \in dom(heap(snd\ \tau))\rfloor\rfloor$
$else\ invalid\ \tau)$

definition *OclIsAbsent*:: $('\mathfrak{A},\ '\alpha::\{null,object\})val \Rightarrow ('\mathfrak{A})Boolean \quad ((\text{-}).oclIsAbsent'('))$
where $X\ .oclIsAbsent() \equiv (\lambda\tau\ .\ if\ (\delta\ X)\ \tau = true\ \tau$
$then\ \lfloor\lfloor oid\text{-}of\ (X\ \tau) \notin dom(heap(fst\ \tau)) \land$
$oid\text{-}of\ (X\ \tau) \notin dom(heap(snd\ \tau))\rfloor\rfloor$
$else\ invalid\ \tau)$

lemma *state-split* : $\tau \models \delta\ X \Longrightarrow$
$\tau \models (X\ .oclIsNew()) \lor \tau \models (X\ .oclIsDeleted()) \lor$
$\tau \models (X\ .oclIsMaintained()) \lor \tau \models (X\ .oclIsAbsent())$
⟨*proof*⟩

lemma *notNew-vs-others* : $\tau \models \delta\ X \Longrightarrow$
$(\neg\ \tau \models (X\ .oclIsNew())) = (\tau \models (X\ .oclIsDeleted()) \lor$
$\tau \models (X\ .oclIsMaintained()) \lor \tau \models (X\ .oclIsAbsent()))$
⟨*proof*⟩

6.2.4. OclIsModifiedOnly

Definition

The following predicate—which is not part of the OCL standard—provides a simple, but powerful means to describe framing conditions. For any formal approach, be it animation of OCL contracts, test-case generation or die-hard theorem proving, the specification of the part of a system transition that *does not change* is of primordial importance. The following operator establishes the equality between old and new objects in the state (provided that they exist in both states), with the exception of those objects.

definition *OclIsModifiedOnly* $::('\mathfrak{A}::object,'\alpha::\{null,object\})Set \Rightarrow {}'\mathfrak{A}\ Boolean$
$(\text{-->}oclIsModifiedOnly'(')$)
where $X\text{->}oclIsModifiedOnly() \equiv (\lambda(\sigma,\sigma').$
$let\ X' = (oid\text{-}of\ `\ \lceil\lceil Rep\text{-}Set_{base}(X(\sigma,\sigma'))\rceil\rceil);$
$S = ((dom\ (heap\ \sigma) \cap dom\ (heap\ \sigma')) - X')$
$in\ if\ (\delta\ X)\ (\sigma,\sigma') = true\ (\sigma,\sigma') \land (\forall x \in \lceil\lceil Rep\text{-}Set_{base}(X(\sigma,\sigma'))\rceil\rceil.\ x \neq null)$
$then\ \lfloor\lfloor\forall\ x \in S.\ (heap\ \sigma)\ x = (heap\ \sigma')\ x\rfloor\rfloor$
$else\ invalid\ (\sigma,\sigma'))$

Execution with Invalid or Null or Null Element as Argument

lemma $invalid\text{->}oclIsModifiedOnly() = invalid$
⟨*proof*⟩

lemma $null\text{->}oclIsModifiedOnly() = invalid$
⟨*proof*⟩

lemma
assumes *X-null* : $\tau \models X\text{->}includes(null)$
shows $\tau \models X\text{->}oclIsModifiedOnly() \triangleq invalid$
⟨*proof*⟩

Context Passing

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

6.2.5. OclSelf

The following predicate—which is not part of the OCL standard—explicitly retrieves in the pre or post state the original OCL expression given as argument.

definition [*simp*]: $OclSelf\ x\ H\ fst\text{-}snd = (\lambda\tau\ .\ if\ (\delta\ x)\ \tau = true\ \tau$
$\quad then\ if\ oid\text{-}of\ (x\ \tau) \in dom(heap(fst\ \tau)) \wedge oid\text{-}of\ (x\ \tau) \in dom(heap\ (snd\ \tau))$
$\quad\quad then\ \ H\ \lceil(heap(fst\text{-}snd\ \tau))(oid\text{-}of\ (x\ \tau))\rceil$
$\quad\quad else\ invalid\ \tau$
$\quad else\ invalid\ \tau)$

definition $OclSelf\text{-}at\text{-}pre :: ('\mathfrak{A}::object,'\alpha::\{null,object\})val \Rightarrow$
$\quad ('\mathfrak{A} \Rightarrow '\alpha) \Rightarrow$
$\quad ('\mathfrak{A}::object,'\alpha::\{null,object\})val\ \ ((\text{-})@pre(\text{-}))$
where $x\ @pre\ H = OclSelf\ x\ H\ fst$

definition $OclSelf\text{-}at\text{-}post :: ('\mathfrak{A}::object,'\alpha::\{null,object\})val \Rightarrow$
$\quad ('\mathfrak{A} \Rightarrow '\alpha) \Rightarrow$
$\quad ('\mathfrak{A}::object,'\alpha::\{null,object\})val\ \ ((\text{-})@post(\text{-}))$
where $x\ @post\ H = OclSelf\ x\ H\ snd$

6.2.6. Framing Theorem

lemma *all-oid-diff*:
assumes $def\text{-}x : \tau \models \delta\ x$
assumes $def\text{-}X : \tau \models \delta\ X$
assumes $def\text{-}X' : \bigwedge x.\ x \in \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil \Longrightarrow x \neq null$

defines $P \equiv (\lambda a.\ not\ (StrictRefEq_{Object}\ x\ a))$
shows $(\tau \models X{-}{>}forAll(a|\ P\ a)) = (oid\text{-}of\ (x\ \tau) \notin oid\text{-}of\ `\ \lceil\lceil Rep\text{-}Set_{base}\ (X\ \tau)\rceil\rceil)$
$\langle proof \rangle$

theorem *framing*:
assumes $modifiesclause{:}\tau \models (X{-}{>}excluding(x)){-}{>}oclIsModifiedOnly()$
and $oid\text{-}is\text{-}typerepr : \tau \models X{-}{>}forAll(a|\ not\ (StrictRefEq_{Object}\ x\ a))$
shows $\tau \models (x\ @pre\ P\ \triangleq\ (x\ @post\ P))$
$\langle proof \rangle$

As corollary, the framing property can be expressed with only the strong equality as comparison operator.

theorem *framing'*:
assumes $wff : WFF\ \tau$
assumes $modifiesclause{:}\tau \models (X{-}{>}excluding(x)){-}{>}oclIsModifiedOnly()$
and $oid\text{-}is\text{-}typerepr : \tau \models X{-}{>}forAll(a|\ not\ (x \triangleq a))$
and $oid\text{-}preserve{:}\ \bigwedge x.\ x \in ran\ (heap(fst\ \tau)) \vee x \in ran\ (heap(snd\ \tau)) \Longrightarrow$
$\quad oid\text{-}of\ (H\ x) = oid\text{-}of\ x$
and *xy-together*:
$\tau \models X{-}{>}forAll(y\ |\ (H\ .allInstances(){-}{>}includes(x)\ and\ H\ .allInstances(){-}{>}includes(y))\ or$
$\quad (H\ .allInstances@pre(){-}{>}includes(x)\ and\ H\ .allInstances@pre(){-}{>}includes(y)))$
shows $\tau \models (x\ @pre\ P\ \triangleq\ (x\ @post\ P))$

$\langle proof \rangle$

6.2.7. Miscellaneous

lemma *pre-post-new*: $\tau \models (x\ .oclIsNew()) \Longrightarrow \neg\ (\tau \models \upsilon(x\ @pre\ H1)) \wedge \neg\ (\tau \models \upsilon(x\ @post\ H2))$
$\langle proof \rangle$

lemma *pre-post-old*: $\tau \models (x\ .oclIsDeleted()) \Longrightarrow \neg\ (\tau \models \upsilon(x\ @pre\ H1)) \wedge \neg\ (\tau \models \upsilon(x\ @post\ H2))$
$\langle proof \rangle$

lemma *pre-post-absent*: $\tau \models (x\ .oclIsAbsent()) \Longrightarrow \neg\ (\tau \models \upsilon(x\ @pre\ H1)) \wedge \neg\ (\tau \models \upsilon(x\ @post\ H2))$
$\langle proof \rangle$

lemma *pre-post-maintained*: $(\tau \models \upsilon(x\ @pre\ H1) \vee \tau \models \upsilon(x\ @post\ H2)) \Longrightarrow \tau \models (x\ .oclIsMaintained())$
$\langle proof \rangle$

lemma *pre-post-maintained'*:
$\tau \models (x\ .oclIsMaintained()) \Longrightarrow (\tau \models \upsilon(x\ @pre\ (Some\ o\ H1)) \wedge \tau \models \upsilon(x\ @post\ (Some\ o\ H2)))$
$\langle proof \rangle$

lemma *framing-same-state*: $(\sigma, \sigma) \models (x\ @pre\ H \triangleq (x\ @post\ H))$
$\langle proof \rangle$

end

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

lemmas *substs1* = *OCL-core.StrongEq-L-subst2-rev*
OCL-core.foundation15[*THEN iffD2*, *THEN OCL-core.StrongEq-L-subst2-rev*]
OCL-core.foundation7'[*THEN iffD2*, *THEN OCL-core.foundation15*[*THEN iffD2*,
THEN OCL-core.StrongEq-L-subst2-rev]]
OCL-core.foundation14[*THEN iffD2*, *THEN OCL-core.StrongEq-L-subst2-rev*]
OCL-core.foundation13[*THEN iffD2*, *THEN OCL-core.StrongEq-L-subst2-rev*]

lemmas *substs2* = *OCL-core.StrongEq-L-subst3-rev*
OCL-core.foundation15[*THEN iffD2*, *THEN OCL-core.StrongEq-L-subst3-rev*]

OCL-core.foundation7'[*THEN iffD2*, *THEN OCL-core.foundation15*[*THEN iffD2*,

THEN OCL-core.StrongEq-L-subst3-rev]]
OCL-core.foundation14[*THEN iffD2*, *THEN OCL-core.StrongEq-L-subst3-rev*]
OCL-core.foundation13[*THEN iffD2*, *THEN OCL-core.StrongEq-L-subst3-rev*]

lemmas *substs4* = *OCL-core.StrongEq-L-subst4-rev*
OCL-core.foundation15[*THEN iffD2*, *THEN OCL-core.StrongEq-L-subst4-rev*]
OCL-core.foundation7'[*THEN iffD2*, *THEN OCL-core.foundation15*[*THEN iffD2*,

THEN OCL-core.StrongEq-L-subst4-rev]]
OCL-core.foundation14[*THEN iffD2*, *THEN OCL-core.StrongEq-L-subst4-rev*]
OCL-core.foundation13[*THEN iffD2*, *THEN OCL-core.StrongEq-L-subst4-rev*]

lemmas *substs* = *substs1 substs2 substs4* [*THEN iffD2*] *substs4*
thm *substs*
⟨*ML*⟩

lemma *test1* : $\tau \models A \Longrightarrow \tau \models (A \textit{ and } B \triangleq B)$
⟨*proof*⟩

lemma *test2* : $\tau \models A \Longrightarrow \tau \models (A \textit{ and } B \triangleq B)$
⟨*proof*⟩

lemma *test3* : $\tau \models A \Longrightarrow \tau \models (A \textit{ and } A)$
⟨*proof*⟩

lemma *test4* : $\tau \models \textit{not } A \Longrightarrow \tau \models (A \textit{ and } B \triangleq \textit{false})$
⟨*proof*⟩

lemma *test5* : $\tau \models (A \triangleq \textit{null}) \Longrightarrow \tau \models (B \triangleq \textit{null}) \Longrightarrow \neg (\tau \models (A \textit{ and } B))$
⟨*proof*⟩

lemma *test6* : $\tau \models \textit{not } A \Longrightarrow \neg (\tau \models (A \textit{ and } B))$
⟨*proof*⟩

lemma *test7* : $\neg (\tau \models (\upsilon\ A)) \Longrightarrow \tau \models (\textit{not } B) \Longrightarrow \neg (\tau \models (A \textit{ and } B))$
⟨*proof*⟩

lemma *X*: $\neg (\tau \models (\textit{invalid and } B))$
⟨*proof*⟩

lemma X': $\neg$ ($\tau \models$ (*invalid and B*))
⟨*proof*⟩
lemma Y: $\neg$ ($\tau \models$ (*null and B*))
⟨*proof*⟩
lemma Z: $\neg$ ($\tau \models$ (*false and B*))
⟨*proof*⟩
lemma Z': ($\tau \models$ (*true and B*)) = ($\tau \models$ *B*)
⟨*proof*⟩

end

theory *Contracts*
imports *OCL-state OCL-lib*
begin

Modeling of an operation contract for an operation with 2 arguments, (so depending on three parameters if one takes "self" into account).

locale *contract0* =
  fixes *f* :: $('\mathfrak{A},'\alpha 0::null)val \Rightarrow$
    $('\mathfrak{A},'res::null)val$
  fixes *PRE* :: $('\mathfrak{A},'\alpha 0::null)val \Rightarrow$
    $('\mathfrak{A}, Boolean_{base})val$
  fixes *POST* :: $('\mathfrak{A},'\alpha 0::null)val \Rightarrow$
    $('\mathfrak{A},'res::null)val \Rightarrow$
    $('\mathfrak{A}, Boolean_{base})val$
  assumes *def-scheme*: *f self* $\equiv$ (λ τ. *if* ($\tau \models$ (δ *self*))
    then SOME res. ($\tau \models$ *PRE self*) $\wedge$
      ($\tau \models$ *POST self* (λ -. *res*))
    else invalid τ)
  assumes $\forall$ σ σ' σ''. ((σ,σ') $\models$ *PRE self*) = ((σ,σ'') $\models$ *PRE self*)

  assumes cp_{PRE}: *PRE* (*self*) τ = *PRE* (λ -. *self* τ) τ

  assumes cp_{POST}:*POST* (*self*) (*res*) τ = *POST* (λ -. *self* τ) (λ -. *res* τ) τ

begin
  lemma *strict0* [*simp*]: *f invalid* = *invalid*

$\langle proof \rangle$

lemma *nullstrict0*[*simp*]: $f\ null = invalid$
$\langle proof \rangle$

lemma *cp-pre*: $cp\ self' \Longrightarrow\ cp\ (\lambda X.\ PRE\ (self'\ X)\ )$
$\langle proof \rangle$

lemma *cp-post*: $cp\ self' \Longrightarrow cp\ res' \Longrightarrow cp\ (\lambda X.\ POST\ (self'\ X)\ (res'\ X))$
$\langle proof \rangle$

lemma *cp0* : $f\ self\ \tau = f\ (\lambda\ \text{-}.\ self\ \tau)\ \tau$
$\langle proof \rangle$

lemma *cp* [*simp*]: $cp\ self' \Longrightarrow\ cp\ res' \Longrightarrow cp\ (\lambda X.\ f\ (self'\ X)\ )$
$\langle proof \rangle$

theorem *unfold* :
assumes *context-ok*: $cp\ E$
and *args-def-or-valid*: $(\tau \models \delta\ self)$
and *pre-satisfied*: $\tau \models PRE\ self$
and *post-satisfiable*: $\exists res.\ (\tau \models POST\ self\ (\lambda\ \text{-}.\ res))$
and *sat-for-sols-post*: $(\bigwedge res.\ \tau \models POST\ self\ (\lambda\ \text{-}.\ res) \Longrightarrow \tau \models E\ (\lambda\ \text{-}.\ res))$
shows $\tau \models E(f\ self)$
$\langle proof \rangle$

lemma *unfold2* :
assumes *context-ok*: $cp\ E$
and *args-def-or-valid*: $(\tau \models \delta\ self)$
and *pre-satisfied*: $\tau \models PRE\ self$
and *postsplit-satisfied*: $\tau \models POST'\ self$
and *post-decomposable* : $\bigwedge res.\ (POST\ self\ res) =$
$((POST'\ self)\ \ and\ (res \triangleq (BODY\ self)))$
shows $(\tau \models E(f\ self)) = (\tau \models E(BODY\ self))$
$\langle proof \rangle$
end

locale *contract1* =
fixes f :: $('\mathfrak{A},'\alpha 0::null)val \Rightarrow$
$('\mathfrak{A},'\alpha 1::null)val \Rightarrow$
$('\mathfrak{A},'res::null)val$
fixes PRE :: $('\mathfrak{A},'\alpha 0::null)val \Rightarrow$
$('\mathfrak{A},'\alpha 1::null)val \Rightarrow$
$('\mathfrak{A},\ Boolean_{base})val$
fixes $POST$ :: $('\mathfrak{A},'\alpha 0::null)val \Rightarrow$
$('\mathfrak{A},'\alpha 1::null)val \Rightarrow$
$('\mathfrak{A},'res::null)val \Rightarrow$

$('\mathfrak{A}, Boolean_{base})val$

assumes *def-scheme*: $f\ self\ a1 \equiv$
$(\lambda\ \tau.\ if\ (\tau \models (\delta\ self)) \wedge\ (\tau \models \upsilon\ a1)$
$then\ SOME\ res.\ (\tau \models PRE\ self\ a1) \wedge$
$(\tau \models POST\ self\ a1\ (\lambda\ \text{-}.\ res))$
$else\ invalid\ \tau)$

assumes $\forall\ \sigma\ \sigma'\ \sigma''.\ ((\sigma,\sigma') \models PRE\ self\ a1) = ((\sigma,\sigma'') \models PRE\ self\ a1)$

assumes cp_{PRE}: $PRE\ (self)\ (a1)\ \ \tau = PRE\ (\lambda\ \text{-}.\ self\ \tau)\ (\lambda\ \text{-}.\ a1\ \tau)\ \tau$

assumes cp_{POST}:$POST\ (self)\ (a1)\ (res)\ \tau = POST\ (\lambda\ \text{-}.\ self\ \tau)(\lambda\ \text{-}.\ a1\ \tau)\ (\lambda\ \text{-}.\ res\ \tau)\ \tau$

begin

lemma *strict0* [*simp*]: $f\ invalid\ X = invalid$
⟨*proof*⟩

lemma *nullstrict0*[*simp*]: $f\ null\ X = invalid$
⟨*proof*⟩

lemma *strict1*[*simp*]: $f\ self\ invalid = invalid$
⟨*proof*⟩

lemma *cp-pre*: $cp\ self' \Longrightarrow cp\ a1' \Longrightarrow\ cp\ (\lambda X.\ PRE\ (self'\ X)\ (a1'\ X)\ )$
⟨*proof*⟩

lemma *cp-post*: $cp\ self' \Longrightarrow cp\ a1' \Longrightarrow cp\ res'$
$\Longrightarrow cp\ (\lambda X.\ POST\ (self'\ X)\ (a1'\ X)\ (res'\ X))$
⟨*proof*⟩

lemma *cp0* : $f\ self\ a1\ \tau = f\ (\lambda\ \text{-}.\ self\ \tau)\ (\lambda\ \text{-}.\ a1\ \tau)\ \tau$
⟨*proof*⟩

lemma *cp* [*simp*]: $cp\ self' \Longrightarrow cp\ a1' \Longrightarrow\ cp\ res' \Longrightarrow cp\ (\lambda X.\ f\ (self'\ X)\ (a1'\ X))$
⟨*proof*⟩

theorem *unfold* :
assumes *context-ok*: $cp\ E$
and *args-def-or-valid*: $(\tau \models \delta\ self) \wedge (\tau \models \upsilon\ a1)$
and *pre-satisfied*: $\tau \models PRE\ self\ a1$
and *post-satisfiable*: $\exists res.\ (\tau \models POST\ self\ a1\ (\lambda\ \text{-}.\ res))$
and *sat-for-sols-post*: $(\bigwedge res.\ \tau \models POST\ self\ a1\ (\lambda\ \text{-}.\ res)\ \Longrightarrow \tau \models E\ (\lambda\ \text{-}.\ res))$
shows $\tau \models E(f\ self\ a1)$
⟨*proof*⟩

lemma *unfold2* :
assumes *context-ok*: $cp\ E$
and *args-def-or-valid*: $(\tau \models \delta\ self) \wedge (\tau \models \upsilon\ a1)$
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*) *and* (*res* $\triangleq$ (*BODY self a1*)))
shows $(\tau \models E(f\ self\ a1)) = (\tau \models E(BODY\ self\ a1))$
⟨*proof*⟩
end

locale *contract2* =
fixes *f* :: $('\mathfrak{A},'\alpha 0::null)val \Rightarrow$
$('\mathfrak{A},'\alpha 1::null)val \Rightarrow ('\mathfrak{A},'\alpha 2::null)val \Rightarrow$
$('\mathfrak{A},'res::null)val$
fixes *PRE* :: $('\mathfrak{A},'\alpha 0::null)val \Rightarrow$
$('\mathfrak{A},'\alpha 1::null)val \Rightarrow ('\mathfrak{A},'\alpha 2::null)val \Rightarrow$
$('\mathfrak{A}, Boolean_{base})val$
fixes *POST* :: $('\mathfrak{A},'\alpha 0::null)val \Rightarrow$
$('\mathfrak{A},'\alpha 1::null)val \Rightarrow ('\mathfrak{A},'\alpha 2::null)val \Rightarrow$
$('\mathfrak{A},'res::null)val \Rightarrow$
$('\mathfrak{A}, Boolean_{base})val$
assumes *def-scheme*: *f self a1 a2* $\equiv$
$(\lambda\ \tau.\ if\ (\tau \models (\delta\ self)) \wedge\ (\tau \models \upsilon\ a1) \wedge\ (\tau \models \upsilon\ a2)$
$then\ SOME\ res.\ (\tau \models PRE\ self\ a1\ a2) \wedge$
$(\tau \models POST\ self\ a1\ a2\ (\lambda\ \text{-}.\ res))$
$else\ invalid\ \tau)$
assumes $\forall\ \sigma\ \sigma'\ \sigma''.\ ((\sigma,\sigma') \models PRE\ self\ a1\ a2) = ((\sigma,\sigma'') \models PRE\ self\ a1\ a2)$

assumes cp_{PRE}: $PRE\ (self)\ (a1)\ (a2)\ \tau = PRE\ (\lambda\ \text{-}.\ self\ \tau)\ (\lambda\ \text{-}.\ a1\ \tau)\ (\lambda\ \text{-}.\ a2\ \tau)\ \tau$

assumes cp_{POST}:$\bigwedge res.\ POST\ (self)\ (a1)\ (a2)\ (res)\ \tau =$
$POST\ (\lambda\ \text{-}.\ self\ \tau)(\lambda\ \text{-}.\ a1\ \tau)(\lambda\ \text{-}.\ a2\ \tau)\ (\lambda\ \text{-}.\ res\ \tau)\ \tau$

begin
lemma *strict0* [*simp*]: *f invalid X Y* = *invalid*
⟨*proof*⟩

lemma *nullstrict0*[*simp*]: *f null X Y* = *invalid*
⟨*proof*⟩

lemma *strict1*[*simp*]: *f self invalid Y* = *invalid*
⟨*proof*⟩

lemma *strict2*[*simp*]: *f self X invalid* = *invalid*
⟨*proof*⟩

lemma *cp-pre*: $cp\ self' \Longrightarrow cp\ a1' \Longrightarrow cp\ a2' \Longrightarrow cp\ (\lambda X.\ PRE\ (self'\ X)\ (a1'\ X)\ (a2'\ X)$
)
⟨*proof*⟩

lemma *cp-post*: $cp\ self' \Longrightarrow cp\ a1' \Longrightarrow cp\ a2' \Longrightarrow cp\ res'$
$\Longrightarrow cp\ (\lambda X.\ POST\ (self'\ X)\ (a1'\ X)\ (a2'\ X)\ (res'\ X))$
⟨*proof*⟩

lemma *cp0* : $f\ self\ a1\ a2\ \tau = f\ (\lambda\ \text{-}.\ self\ \tau)\ (\lambda\ \text{-}.\ a1\ \tau)\ (\lambda\ \text{-}.\ a2\ \tau)\ \tau$
⟨*proof*⟩

lemma *cp* [*simp*]: $cp\ self' \Longrightarrow cp\ a1' \Longrightarrow cp\ a2' \Longrightarrow cp\ res'$
$\Longrightarrow cp\ (\lambda X.\ f\ (self'\ X)\ (a1'\ X)\ (a2'\ X))$
⟨*proof*⟩

theorem *unfold* :
assumes *context-ok*: $cp\ E$
and *args-def-or-valid*: $(\tau \models \delta\ self) \wedge (\tau \models \upsilon\ a1) \wedge (\tau \models \upsilon\ a2)$
and *pre-satisfied*: $\tau \models PRE\ self\ a1\ a2$
and *post-satisfiable*: $\exists res.\ (\tau \models POST\ self\ a1\ a2\ (\lambda\ \text{-}.\ res))$
and *sat-for-sols-post*: $(\bigwedge res.\ \tau \models POST\ self\ a1\ a2\ (\lambda\ \text{-}.\ res) \Longrightarrow \tau \models E\ (\lambda\ \text{-}.\ res))$
shows $\tau \models E(f\ self\ a1\ a2)$
⟨*proof*⟩

lemma *unfold2* :
assumes *context-ok*: $cp\ E$
and *args-def-or-valid*: $(\tau \models \delta\ self) \wedge (\tau \models \upsilon\ a1) \wedge (\tau \models \upsilon\ a2)$
and *pre-satisfied*: $\tau \models PRE\ self\ a1\ a2$
and *postsplit-satisfied*: $\tau \models POST'\ self\ a1\ a2$
and *post-decomposable* : $\bigwedge res.\ (POST\ self\ a1\ a2\ res) =$
$((POST'\ self\ a1\ a2)\ \ and\ (res \triangleq (BODY\ self\ a1\ a2)))$
shows $(\tau \models E(f\ self\ a1\ a2)) = (\tau \models E(BODY\ self\ a1\ a2))$
⟨*proof*⟩
end

end

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

end

7. Example I : The Employee Analysis Model (UML)

theory
  Employee-AnalysisModel-UMLPart
imports
  ../src/OCL-main
begin

7.1. 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.

7.1.1. Outlining the Example

We are presenting here an "analysis-model" of the (slightly modified) example Figure 7.3, page 20 of the OCL standard [30]. Here, analysis model means that associations were really represented as relation on objects on the state—as is intended by the standard—rather by pointers between objects as is done in our "design model" (see Chapter 8). To be precise, this theory contains the formalization of the data-part covered by the UML class model (see Figure 7.1):

This means that the association (attached to the association class `EmployeeRanking`) with the association ends `boss` and `employees` is implemented by the attribute `boss` and the operation `employees` (to be discussed in the OCL part captured by the subsequent theory).

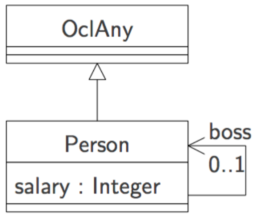


Figure 7.1.: A simple UML class model drawn from Figure 7.3, page 20 of [30].

7.2. 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 $\mathit{type}_{Person} = \mathit{mk}_{Person}\ \mathit{oid}$
$\quad\quad\quad\quad\quad\quad\quad \mathit{int\ option}$

datatype $\mathit{type}_{OclAny} = \mathit{mk}_{OclAny}\ \mathit{oid}$
$\quad\quad\quad\quad\quad\quad\quad (\mathit{int\ option})\ \mathit{option}$

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 $\mathfrak{A} = \mathit{in}_{Person}\ \mathit{type}_{Person} \mid \mathit{in}_{OclAny}\ \mathit{type}_{OclAny}$

Having fixed the object universe, we can introduce type synonyms that exactly correspond to OCL types. Again, we exploit that our representation of OCL is a "shallow embedding" with a one-to-one correspondance of OCL-types to types of the meta-language HOL.

type-synonym $\mathit{Boolean} = \mathfrak{A}\ \mathit{Boolean}$
type-synonym $\mathit{Integer} = \mathfrak{A}\ \mathit{Integer}$
type-synonym $\mathit{Void} = \mathfrak{A}\ \mathit{Void}$
type-synonym $\mathit{OclAny} = (\mathfrak{A},\ \mathit{type}_{OclAny}\ \mathit{option\ option})\ \mathit{val}$
type-synonym $\mathit{Person} = (\mathfrak{A},\ \mathit{type}_{Person}\ \mathit{option\ option})\ \mathit{val}$
type-synonym $\mathit{Set\text{-}Integer} = (\mathfrak{A},\ \mathit{int\ option\ option})\ \mathit{Set}$
type-synonym $\mathit{Set\text{-}Person} = (\mathfrak{A},\ \mathit{type}_{Person}\ \mathit{option\ option})\ \mathit{Set}$

Just a little check:

typ *Boolean*

To reuse key-elements of the library like referential equality, we have to show that the object universe belongs to the type class "oclany," i. e., each class type has to provide a function *oid-of* yielding the object id (oid) of the object.

instantiation $type_{Person}$ :: *object*
begin
  definition *oid-of-*$type_{Person}$*-def*: *oid-of* x = (*case* x *of* mk_{Person} *oid* - ⇒ *oid*)
  instance ⟨*proof*⟩
end

instantiation $type_{OclAny}$ :: *object*
begin
  definition *oid-of-*$type_{OclAny}$*-def*: *oid-of* x = (*case* x *of* mk_{OclAny} *oid* - ⇒ *oid*)
  instance ⟨*proof*⟩
end

instantiation $\mathfrak{A}$ :: *object*
begin
  definition *oid-of-*$\mathfrak{A}$*-def*: *oid-of* x = (*case* x *of*
    in_{Person} *person* ⇒ *oid-of person*
    | in_{OclAny} *oclany* ⇒ *oid-of oclany*)
  instance ⟨*proof*⟩
end

7.3. Instantiation of the Generic Strict Equality

We instantiate the referential equality on *Person* and *OclAny*

defs(**overloaded**) $StrictRefEq_{Object}$*-*$_{Person}$: $(x::Person) \doteq y \ \equiv \ StrictRefEq_{Object}\ x\ y$
defs(**overloaded**) $StrictRefEq_{Object}$*-*$_{OclAny}$: $(x::OclAny) \doteq y \ \equiv \ StrictRefEq_{Object}\ x\ y$

lemmas
  cp-$StrictRefEq_{Object}$[*of* x::*Person* y::*Person* τ,
    simplified $StrictRefEq_{Object}$*-*$_{Person}$[*symmetric*]]
  cp-intro(*9*) [*of* P::*Person* ⇒*Person*Q::*Person* ⇒*Person*,
    simplified $StrictRefEq_{Object}$*-*$_{Person}$[*symmetric*]]
  $StrictRefEq_{Object}$*-def* [*of* x::*Person* y::*Person*,
    simplified $StrictRefEq_{Object}$*-*$_{Person}$[*symmetric*]]
  $StrictRefEq_{Object}$*-defargs* [*of* - x::*Person* y::*Person*,
    simplified $StrictRefEq_{Object}$*-*$_{Person}$[*symmetric*]]
  $StrictRefEq_{Object}$*-strict1*
    [*of* x::*Person*,
    simplified $StrictRefEq_{Object}$*-*$_{Person}$[*symmetric*]]
  $StrictRefEq_{Object}$*-strict2*
    [*of* x::*Person*,
    simplified $StrictRefEq_{Object}$*-*$_{Person}$[*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.

7.4. OclAsType

7.4.1. Definition

consts $OclAsType_{OclAny}$ :: $'\alpha \Rightarrow OclAny$ ((-) $.oclAsType'(OclAny')$)
consts $OclAsType_{Person}$ :: $'\alpha \Rightarrow Person$ ((-) $.oclAsType'(Person')$)

definition $OclAsType_{OclAny}\text{-}\mathfrak{A} = (\lambda u.\ \lfloor case\ u\ of\ in_{OclAny}\ a \Rightarrow a$
$\quad | \ in_{Person}\ (mk_{Person}\ oid\ a) \Rightarrow mk_{OclAny}\ oid\ \lfloor a \rfloor \rfloor)$

lemma $OclAsType_{OclAny}\text{-}\mathfrak{A}\text{-}some$: $OclAsType_{OclAny}\text{-}\mathfrak{A}\ x \neq None$
$\langle proof \rangle$

defs (**overloaded**) $OclAsType_{OclAny}\text{-}OclAny$:
$\quad (X::OclAny)\ .oclAsType(OclAny) \equiv X$

defs (**overloaded**) $OclAsType_{OclAny}\text{-}Person$:
$\quad (X::Person)\ .oclAsType(OclAny) \equiv$
$\quad\quad (\lambda\tau.\ case\ X\ \tau\ of$
$\quad\quad\quad \bot \ \Rightarrow invalid\ \tau$
$\quad\quad\quad | \ \lfloor \bot \rfloor \Rightarrow null\ \tau$
$\quad\quad\quad | \ \lfloor \lfloor mk_{Person}\ oid\ a\ \rfloor \rfloor \Rightarrow\ \lfloor \lfloor\ (mk_{OclAny}\ oid\ \lfloor a \rfloor)\ \rfloor \rfloor)$

definition $OclAsType_{Person}\text{-}\mathfrak{A} = (\lambda u.\ case\ u\ of\ in_{Person}\ p \Rightarrow \lfloor p \rfloor$
$\quad | \ in_{OclAny}\ (mk_{OclAny}\ oid\ \lfloor a \rfloor) \Rightarrow \lfloor mk_{Person}\ oid\ a \rfloor$
$\quad | \ \text{-} \Rightarrow None)$

defs (**overloaded**) $OclAsType_{Person}\text{-}OclAny$:
$\quad (X::OclAny)\ .oclAsType(Person) \equiv$
$\quad\quad (\lambda\tau.\ case\ X\ \tau\ of$
$\quad\quad\quad \bot \ \Rightarrow invalid\ \tau$
$\quad\quad\quad | \ \lfloor \bot \rfloor \Rightarrow null\ \tau$
$\quad\quad\quad | \ \lfloor \lfloor mk_{OclAny}\ oid\ \bot\ \rfloor \rfloor \Rightarrow\ invalid\ \tau \quad (*\ down{-}cast\ exception\ *)$
$\quad\quad\quad | \ \lfloor \lfloor mk_{OclAny}\ oid\ \lfloor a \rfloor\ \rfloor \rfloor \Rightarrow\ \lfloor \lfloor mk_{Person}\ oid\ a \rfloor \rfloor)$

defs (**overloaded**) $OclAsType_{Person}\text{-}Person$:
$\quad (X::Person)\ .oclAsType(Person) \equiv X$

lemmas [$simp$] =
$OclAsType_{OclAny}\text{-}OclAny$
$OclAsType_{Person}\text{-}Person$

7.4.2. Context Passing

lemma $cp\text{-}OclAsType_{OclAny}\text{-}Person\text{-}Person$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X::Person)::Person)$ $.oclAsType(OclAny))$
⟨proof⟩
lemma $cp\text{-}OclAsType_{OclAny}\text{-}OclAny\text{-}OclAny$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X::OclAny)::OclAny)$ $.oclAsType(OclAny))$
⟨proof⟩
lemma $cp\text{-}OclAsType_{Person}\text{-}Person\text{-}Person$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X::Person)::Person)$ $.oclAsType(Person))$
⟨proof⟩
lemma $cp\text{-}OclAsType_{Person}\text{-}OclAny\text{-}OclAny$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X::OclAny)::OclAny)$ $.oclAsType(Person))$
⟨proof⟩

lemma $cp\text{-}OclAsType_{OclAny}\text{-}Person\text{-}OclAny$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X::Person)::OclAny)$ $.oclAsType(OclAny))$
⟨proof⟩
lemma $cp\text{-}OclAsType_{OclAny}\text{-}OclAny\text{-}Person$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X::OclAny)::Person)$ $.oclAsType(OclAny))$
⟨proof⟩
lemma $cp\text{-}OclAsType_{Person}\text{-}Person\text{-}OclAny$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X::Person)::OclAny)$ $.oclAsType(Person))$
⟨proof⟩
lemma $cp\text{-}OclAsType_{Person}\text{-}OclAny\text{-}Person$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X::OclAny)::Person)$ $.oclAsType(Person))$
⟨proof⟩

lemmas [*simp*] =
$cp\text{-}OclAsType_{OclAny}\text{-}Person\text{-}Person$
$cp\text{-}OclAsType_{OclAny}\text{-}OclAny\text{-}OclAny$
$cp\text{-}OclAsType_{Person}\text{-}Person\text{-}Person$
$cp\text{-}OclAsType_{Person}\text{-}OclAny\text{-}OclAny$

$cp\text{-}OclAsType_{OclAny}\text{-}Person\text{-}OclAny$
$cp\text{-}OclAsType_{OclAny}\text{-}OclAny\text{-}Person$
$cp\text{-}OclAsType_{Person}\text{-}Person\text{-}OclAny$
$cp\text{-}OclAsType_{Person}\text{-}OclAny\text{-}Person$

7.4.3. Execution with Invalid or Null as Argument

lemma $OclAsType_{OclAny}\text{-}OclAny\text{-}strict$: $(invalid::OclAny)\ .oclAsType(OclAny) = invalid$
⟨proof⟩

lemma $OclAsType_{OclAny}\text{-}OclAny\text{-}nullstrict$: $(null::OclAny)\ .oclAsType(OclAny) = null$
⟨proof⟩

lemma $OclAsType_{OclAny}\text{-}Person\text{-}strict[simp]$: $(invalid::Person)\ .oclAsType(OclAny) = invalid$
⟨proof⟩

lemma $OclAsType_{OclAny}$-Person-nullstrict[simp] : $(null::Person)$ $.oclAsType(OclAny) = null$
⟨proof⟩

lemma $OclAsType_{Person}$-OclAny-strict[simp] : $(invalid::OclAny)$ $.oclAsType(Person) = invalid$
⟨proof⟩

lemma $OclAsType_{Person}$-OclAny-nullstrict[simp] : $(null::OclAny)$ $.oclAsType(Person) = null$
⟨proof⟩

lemma $OclAsType_{Person}$-Person-strict : $(invalid::Person)$ $.oclAsType(Person) = invalid$
⟨proof⟩
lemma $OclAsType_{Person}$-Person-nullstrict : $(null::Person)$ $.oclAsType(Person) = null$
⟨proof⟩

7.5. OclIsTypeOf

7.5.1. Definition

consts $OclIsTypeOf_{OclAny}$ :: $'\alpha \Rightarrow Boolean$ $((\text{-}).oclIsTypeOf'(OclAny'))$
consts $OclIsTypeOf_{Person}$ :: $'\alpha \Rightarrow Boolean$ $((\text{-}).oclIsTypeOf'(Person'))$

defs (**overloaded**) $OclIsTypeOf_{OclAny}$-OclAny:
$(X::OclAny)$ $.oclIsTypeOf(OclAny) \equiv$
$(\lambda\tau.$ case X τ of
$\bot \Rightarrow invalid\ \tau$
$\mid \lfloor\bot\rfloor \Rightarrow true\ \tau$ (* invalid ?? *)
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \bot\ \rfloor\rfloor \Rightarrow true\ \tau$
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \lfloor\text{-}\rfloor\ \rfloor\rfloor \Rightarrow false\ \tau)$

defs (**overloaded**) $OclIsTypeOf_{OclAny}$-Person:
$(X::Person)$ $.oclIsTypeOf(OclAny) \equiv$
$(\lambda\tau.$ case X τ of
$\bot \Rightarrow invalid\ \tau$
$\mid \lfloor\bot\rfloor \Rightarrow true\ \tau$ (* invalid ?? *)
$\mid \lfloor\lfloor\ \text{-}\ \rfloor\rfloor \Rightarrow false\ \tau)$

defs (**overloaded**) $OclIsTypeOf_{Person}$-OclAny:
$(X::OclAny)$ $.oclIsTypeOf(Person) \equiv$
$(\lambda\tau.$ case X τ of
$\bot \Rightarrow invalid\ \tau$
$\mid \lfloor\bot\rfloor \Rightarrow true\ \tau$
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \bot\ \rfloor\rfloor \Rightarrow false\ \tau$
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \lfloor\text{-}\rfloor\ \rfloor\rfloor \Rightarrow true\ \tau)$

defs (**overloaded**) $OclIsTypeOf_{Person}$-Person:
$(X::Person)$ $.oclIsTypeOf(Person) \equiv$
$(\lambda\tau.$ case X τ of
$\bot \Rightarrow invalid\ \tau$

| - ⇒ $true\ \tau$)

7.5.2. Context Passing

lemma *cp-OclIsTypeOf*$_{OclAny}$*-Person-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X{::}Person){::}Person).oclIsTypeOf(OclAny))$
⟨*proof*⟩
lemma *cp-OclIsTypeOf*$_{OclAny}$*-OclAny-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X{::}OclAny){::}OclAny).oclIsTypeOf(OclAny))$
⟨*proof*⟩
lemma *cp-OclIsTypeOf*$_{Person}$*-Person-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X{::}Person){::}Person).oclIsTypeOf(Person))$
⟨*proof*⟩
lemma *cp-OclIsTypeOf*$_{Person}$*-OclAny-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X{::}OclAny){::}OclAny).oclIsTypeOf(Person))$
⟨*proof*⟩

lemma *cp-OclIsTypeOf*$_{OclAny}$*-Person-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X{::}Person){::}OclAny).oclIsTypeOf(OclAny))$
⟨*proof*⟩
lemma *cp-OclIsTypeOf*$_{OclAny}$*-OclAny-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X{::}OclAny){::}Person).oclIsTypeOf(OclAny))$
⟨*proof*⟩
lemma *cp-OclIsTypeOf*$_{Person}$*-Person-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X{::}Person){::}OclAny).oclIsTypeOf(Person))$
⟨*proof*⟩
lemma *cp-OclIsTypeOf*$_{Person}$*-OclAny-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X{::}OclAny){::}Person).oclIsTypeOf(Person))$
⟨*proof*⟩

lemmas [*simp*] =
cp-OclIsTypeOf$_{OclAny}$*-Person-Person*
cp-OclIsTypeOf$_{OclAny}$*-OclAny-OclAny*
cp-OclIsTypeOf$_{Person}$*-Person-Person*
cp-OclIsTypeOf$_{Person}$*-OclAny-OclAny*

cp-OclIsTypeOf$_{OclAny}$*-Person-OclAny*
cp-OclIsTypeOf$_{OclAny}$*-OclAny-Person*
cp-OclIsTypeOf$_{Person}$*-Person-OclAny*
cp-OclIsTypeOf$_{Person}$*-OclAny-Person*

7.5.3. Execution with Invalid or Null as Argument

lemma *OclIsTypeOf*$_{OclAny}$*-OclAny-strict1*[*simp*]:
$(invalid{::}OclAny)\ .oclIsTypeOf(OclAny) = invalid$
⟨*proof*⟩
lemma *OclIsTypeOf*$_{OclAny}$*-OclAny-strict2*[*simp*]:
$(null{::}OclAny)\ .oclIsTypeOf(OclAny) = true$
⟨*proof*⟩

lemma $OclIsTypeOf_{OclAny}$*-Person-strict1* [*simp*]:
$(invalid::Person)$ $.oclIsTypeOf(OclAny) = invalid$
⟨*proof*⟩
lemma $OclIsTypeOf_{OclAny}$*-Person-strict2* [*simp*]:
$(null::Person)$ $.oclIsTypeOf(OclAny) = true$
⟨*proof*⟩
lemma $OclIsTypeOf_{Person}$*-OclAny-strict1* [*simp*]:
$(invalid::OclAny)$ $.oclIsTypeOf(Person) = invalid$
⟨*proof*⟩
lemma $OclIsTypeOf_{Person}$*-OclAny-strict2* [*simp*]:
$(null::OclAny)$ $.oclIsTypeOf(Person) = true$
⟨*proof*⟩
lemma $OclIsTypeOf_{Person}$*-Person-strict1* [*simp*]:
$(invalid::Person)$ $.oclIsTypeOf(Person) = invalid$
⟨*proof*⟩
lemma $OclIsTypeOf_{Person}$*-Person-strict2* [*simp*]:
$(null::Person)$ $.oclIsTypeOf(Person) = true$
⟨*proof*⟩

7.5.4. Up Down Casting

lemma *actualType-larger-staticType*:
assumes *isdef*: $\tau \models (\delta\ X)$
shows $\tau \models (X::Person)$ $.oclIsTypeOf(OclAny) \triangleq false$
⟨*proof*⟩

lemma *down-cast-type*:
assumes *isOclAny*: $\tau \models (X::OclAny)$ $.oclIsTypeOf(OclAny)$
and *non-null*: $\tau \models (\delta\ X)$
shows $\tau \models (X\ .oclAsType(Person)) \triangleq invalid$
⟨*proof*⟩

lemma *down-cast-type'*:
assumes *isOclAny*: $\tau \models (X::OclAny)$ $.oclIsTypeOf(OclAny)$
and *non-null*: $\tau \models (\delta\ X)$
shows $\tau \models not\ (\upsilon\ (X\ .oclAsType(Person)))$
⟨*proof*⟩

lemma *up-down-cast* :
assumes *isdef*: $\tau \models (\delta\ X)$
shows $\tau \models ((X::Person)\ .oclAsType(OclAny)\ .oclAsType(Person) \triangleq X)$
⟨*proof*⟩

lemma *up-down-cast-Person-OclAny-Person* [*simp*]:
shows $((X::Person)\ .oclAsType(OclAny)\ .oclAsType(Person) = X)$
⟨*proof*⟩

lemma *up-down-cast-Person-OclAny-Person'*:

assumes $\tau \models \upsilon\ X$
shows $\tau \models (((X :: Person)\ .oclAsType(OclAny)\ .oclAsType(Person)) \doteq X)$
$\langle proof \rangle$

lemma *up-down-cast-Person-OclAny-Person''*:
assumes $\tau \models \upsilon\ (X :: Person)$
shows $\tau \models (X\ .oclIsTypeOf(Person)\ implies\ (X\ .oclAsType(OclAny)\ .oclAsType(Person)) \doteq X)$
$\langle proof \rangle$

7.6. OclIsKindOf

7.6.1. Definition

consts $OclIsKindOf_{OclAny} :: {}'\alpha \Rightarrow Boolean\ ((\text{-}).oclIsKindOf'(OclAny'))$
consts $OclIsKindOf_{Person} :: {}'\alpha \Rightarrow Boolean\ ((\text{-}).oclIsKindOf'(Person'))$

defs (**overloaded**) $OclIsKindOf_{OclAny}$*-OclAny*:
$(X{::}OclAny)\ .oclIsKindOf(OclAny) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\bot \Rightarrow invalid\ \tau$
$|\ \text{-} \Rightarrow true\ \tau)$

defs (**overloaded**) $OclIsKindOf_{OclAny}$*-Person*:
$(X{::}Person)\ .oclIsKindOf(OclAny) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\bot \Rightarrow invalid\ \tau$
$|\ \text{-}\Rightarrow true\ \tau)$

defs (**overloaded**) $OclIsKindOf_{Person}$*-OclAny*:
$(X{::}OclAny)\ .oclIsKindOf(Person) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\bot \Rightarrow invalid\ \tau$
$|\ \lfloor\bot\rfloor \Rightarrow true\ \tau$
$|\ \lfloor\lfloor mk_{OclAny}\ oid\ \bot\ \rfloor\rfloor \Rightarrow false\ \tau$
$|\ \lfloor\lfloor mk_{OclAny}\ oid\ \lfloor\text{-}\rfloor\ \rfloor\rfloor \Rightarrow true\ \tau)$

defs (**overloaded**) $OclIsKindOf_{Person}$*-Person*:
$(X{::}Person)\ .oclIsKindOf(Person) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\bot \Rightarrow invalid\ \tau$
$|\ \text{-} \Rightarrow true\ \tau)$

7.6.2. Context Passing

lemma $cp\text{-}OclIsKindOf_{OclAny}$*-Person-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X{::}Person){::}Person).oclIsKindOf(OclAny))$

$\langle proof \rangle$

lemma *cp-OclIsKindOf*$_{OclAny}$*-OclAny-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::OclAny).oclIsKindOf(OclAny))$

$\langle proof \rangle$

lemma *cp-OclIsKindOf*$_{Person}$*-Person-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::Person)::Person).oclIsKindOf(Person))$

$\langle proof \rangle$

lemma *cp-OclIsKindOf*$_{Person}$*-OclAny-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::OclAny).oclIsKindOf(Person))$

$\langle proof \rangle$

lemma *cp-OclIsKindOf*$_{OclAny}$*-Person-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(OclAny))$

$\langle proof \rangle$

lemma *cp-OclIsKindOf*$_{OclAny}$*-OclAny-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(OclAny))$

$\langle proof \rangle$

lemma *cp-OclIsKindOf*$_{Person}$*-Person-OclAny*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(Person))$

$\langle proof \rangle$

lemma *cp-OclIsKindOf*$_{Person}$*-OclAny-Person*: $cp\ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(Person))$

$\langle proof \rangle$

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*

7.6.3. Execution with Invalid or Null as Argument

lemma *OclIsKindOf*$_{OclAny}$*-OclAny-strict1*[*simp*] : $(invalid::OclAny)\ .oclIsKindOf(OclAny) = invalid$

$\langle proof \rangle$

lemma *OclIsKindOf*$_{OclAny}$*-OclAny-strict2*[*simp*] : $(null::OclAny)\ .oclIsKindOf(OclAny) = true$

$\langle proof \rangle$

lemma *OclIsKindOf*$_{OclAny}$*-Person-strict1*[*simp*] : $(invalid::Person)\ .oclIsKindOf(OclAny) = invalid$

$\langle proof \rangle$

lemma $OclIsKindOf_{OclAny}$-*Person-strict2*[*simp*] : (*null*::*Person*) .*oclIsKindOf*(*OclAny*) = *true*
⟨*proof*⟩

lemma $OclIsKindOf_{Person}$-*OclAny-strict1*[*simp*]: (*invalid*::*OclAny*) .*oclIsKindOf*(*Person*) = *invalid*
⟨*proof*⟩

lemma $OclIsKindOf_{Person}$-*OclAny-strict2*[*simp*]: (*null*::*OclAny*) .*oclIsKindOf*(*Person*) = *true*
⟨*proof*⟩

lemma $OclIsKindOf_{Person}$-*Person-strict1*[*simp*]: (*invalid*::*Person*) .*oclIsKindOf*(*Person*) = *invalid*
⟨*proof*⟩

lemma $OclIsKindOf_{Person}$-*Person-strict2*[*simp*]: (*null*::*Person*) .*oclIsKindOf*(*Person*) = *true*
⟨*proof*⟩

7.6.4. Up Down Casting

lemma *actualKind-larger-staticKind*:
assumes *isdef*: $\tau \models (\delta\ X)$
shows $\tau \models ((X{::}Person)\ .oclIsKindOf(OclAny) \triangleq true)$
⟨*proof*⟩

lemma *down-cast-kind*:
assumes *isOclAny*: $\neg\ (\tau \models ((X{::}OclAny).oclIsKindOf(Person)))$
and *non-null*: $\tau \models (\delta\ X)$
shows $\tau \models ((X\ .oclAsType(Person)) \triangleq invalid)$
⟨*proof*⟩

7.7. OclAllInstances

To denote OCL-types occuring in OCL expressions syntactically—as, for example, as "argument" of oclAllInstances()—we use the inverses of the injection functions into the object universes; we show that this is sufficient "characterization."

definition *Person* ≡ $OclAsType_{Person}$-$\mathfrak{A}$
definition *OclAny* ≡ $OclAsType_{OclAny}$-$\mathfrak{A}$
lemmas [*simp*] = *Person-def OclAny-def*

lemma $OclAllInstances\text{-}generic_{OclAny}$-*exec*: *OclAllInstances-generic pre-post OclAny* =
$(\lambda\tau.\ Abs\text{-}Set_{base}\ \lfloor\lfloor\ Some\ `\ OclAny\ `\ ran\ (heap\ (pre\text{-}post\ \tau))\ \rfloor\rfloor)$
⟨*proof*⟩

lemma $OclAllInstances\text{-}at\text{-}post_{OclAny}$-*exec*: *OclAny* .*allInstances*() =
$(\lambda\tau.\ Abs\text{-}Set_{base}\ \lfloor\lfloor\ Some\ `\ OclAny\ `\ ran\ (heap\ (snd\ \tau))\ \rfloor\rfloor)$
⟨*proof*⟩

lemma $OclAllInstances\text{-}at\text{-}pre_{OclAny}$-*exec*: *OclAny* .*allInstances@pre*() =

$(\lambda\tau.\ \ Abs\text{-}Set_{base}\ \ \lfloor\lfloor\ Some\ \text{`}\ OclAny\ \text{`}\ ran\ (heap\ (fst\ \tau))\ \rfloor\rfloor)$
$\langle proof\rangle$

7.7.1. OclIsTypeOf

lemma $OclAny\text{-}allInstances\text{-}generic\text{-}oclIsTypeOf_{OclAny}1$:
assumes $[simp]$: $\bigwedge x.\ pre\text{-}post\ (x, x) = x$
shows $\exists\tau.\ \ (\tau\ \ \models\ \ ((OclAllInstances\text{-}generic\ \ pre\text{-}post\ \ OclAny){-}{>}forAll(X|X\ .oclIsTypeOf(OclAny))))$
$\langle proof\rangle$

lemma $OclAny\text{-}allInstances\text{-}at\text{-}post\text{-}oclIsTypeOf_{OclAny}1$:
$\exists\tau.\ (\tau \models\ \ (OclAny\ .allInstances(){-}{>}forAll(X|X\ .oclIsTypeOf(OclAny))))$
$\langle proof\rangle$

lemma $OclAny\text{-}allInstances\text{-}at\text{-}pre\text{-}oclIsTypeOf_{OclAny}1$:
$\exists\tau.\ (\tau \models\ \ (OclAny\ .allInstances@pre(){-}{>}forAll(X|X\ .oclIsTypeOf(OclAny))))$
$\langle proof\rangle$

lemma $OclAny\text{-}allInstances\text{-}generic\text{-}oclIsTypeOf_{OclAny}2$:
assumes $[simp]$: $\bigwedge x.\ pre\text{-}post\ (x, x) = x$
shows $\exists\tau.\ \ (\tau\ \ \models\ \ not\ \ ((OclAllInstances\text{-}generic\ \ pre\text{-}post\ \ OclAny){-}{>}forAll(X|X\ .oclIsTypeOf(OclAny))))$
$\langle proof\rangle$

lemma $OclAny\text{-}allInstances\text{-}at\text{-}post\text{-}oclIsTypeOf_{OclAny}2$:
$\exists\tau.\ (\tau \models not\ (OclAny\ .allInstances(){-}{>}forAll(X|X\ .oclIsTypeOf(OclAny))))$
$\langle proof\rangle$

lemma $OclAny\text{-}allInstances\text{-}at\text{-}pre\text{-}oclIsTypeOf_{OclAny}2$:
$\exists\tau.\ (\tau \models not\ (OclAny\ .allInstances@pre(){-}{>}forAll(X|X\ .oclIsTypeOf(OclAny))))$
$\langle proof\rangle$

lemma $Person\text{-}allInstances\text{-}generic\text{-}oclIsTypeOf_{Person}$:
$\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ Person){-}{>}forAll(X|X\ .oclIsTypeOf(Person)))$
$\langle proof\rangle$

lemma $Person\text{-}allInstances\text{-}at\text{-}post\text{-}oclIsTypeOf_{Person}$:
$\tau \models (Person\ .allInstances(){-}{>}forAll(X|X\ .oclIsTypeOf(Person)))$
$\langle proof\rangle$

lemma $Person\text{-}allInstances\text{-}at\text{-}pre\text{-}oclIsTypeOf_{Person}$:
$\tau \models (Person\ .allInstances@pre(){-}{>}forAll(X|X\ .oclIsTypeOf(Person)))$
$\langle proof\rangle$

7.7.2. OclIsKindOf

lemma $OclAny\text{-}allInstances\text{-}generic\text{-}oclIsKindOf_{OclAny}$:
$\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ OclAny){-}{>}forAll(X|X\ .oclIsKindOf(OclAny)))$
$\langle proof\rangle$

lemma *OclAny-allInstances-at-post-oclIsKindOf*$_{OclAny}$:
$\tau \models (OclAny\ .allInstances() -{>} forAll(X|X\ .oclIsKindOf(OclAny)))$
$\langle proof \rangle$

lemma *OclAny-allInstances-at-pre-oclIsKindOf*$_{OclAny}$:
$\tau \models (OclAny\ .allInstances@pre() -{>} forAll(X|X\ .oclIsKindOf(OclAny)))$
$\langle proof \rangle$

lemma *Person-allInstances-generic-oclIsKindOf*$_{OclAny}$:
$\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ Person) -{>} forAll(X|X\ .oclIsKindOf(OclAny)))$
$\langle proof \rangle$

lemma *Person-allInstances-at-post-oclIsKindOf*$_{OclAny}$:
$\tau \models (Person\ .allInstances() -{>} forAll(X|X\ .oclIsKindOf(OclAny)))$
$\langle proof \rangle$

lemma *Person-allInstances-at-pre-oclIsKindOf*$_{OclAny}$:
$\tau \models (Person\ .allInstances@pre() -{>} forAll(X|X\ .oclIsKindOf(OclAny)))$
$\langle proof \rangle$

lemma *Person-allInstances-generic-oclIsKindOf*$_{Person}$:
$\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ Person) -{>} forAll(X|X\ .oclIsKindOf(Person)))$
$\langle proof \rangle$

lemma *Person-allInstances-at-post-oclIsKindOf*$_{Person}$:
$\tau \models (Person\ .allInstances() -{>} forAll(X|X\ .oclIsKindOf(Person)))$
$\langle proof \rangle$

lemma *Person-allInstances-at-pre-oclIsKindOf*$_{Person}$:
$\tau \models (Person\ .allInstances@pre() -{>} forAll(X|X\ .oclIsKindOf(Person)))$
$\langle proof \rangle$

7.8. The Accessors (any, boss, salary)

Should be generated entirely from a class-diagram.

7.8.1. 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 Employee_DesignModel_UMLPart, 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 $\mathit{eval\text{-}extract} :: ('\mathfrak{A},('a::\mathit{object})\ \mathit{option}\ \mathit{option})\ \mathit{val}$
$\Rightarrow (\mathit{oid} \Rightarrow ('\mathfrak{A},'c::\mathit{null})\ \mathit{val})$
$\Rightarrow ('\mathfrak{A},'c::\mathit{null})\ \mathit{val}$
where $\mathit{eval\text{-}extract}\ X\ f = (\lambda\ \tau.\ \mathit{case}\ X\ \tau\ \mathit{of}$
$\bot \Rightarrow \mathit{invalid}\ \tau \quad (*\ \mathit{exception\ propagation}\ *)$
$|\ \lfloor\ \bot\ \rfloor \Rightarrow \mathit{invalid}\ \tau\ (*\ \mathit{dereferencing\ null\ pointer}\ *)$
$|\ \lfloor\lfloor\ \mathit{obj}\ \rfloor\rfloor \Rightarrow f\ (\mathit{oid\text{-}of}\ \mathit{obj})\ \tau)$

definition $\mathit{choose}_2\text{-}1 = \mathit{fst}$
definition $\mathit{choose}_2\text{-}2 = \mathit{snd}$

definition $\mathit{List\text{-}flatten} = (\lambda l.\ (\mathit{foldl}\ ((\lambda \mathit{acc}.\ (\lambda l.\ (\mathit{foldl}\ ((\lambda \mathit{acc}.\ (\lambda l.\ (\mathit{Cons}\ (l)\ (\mathit{acc})))))\ (\mathit{acc})\ ((\mathit{rev}\ (l)))))))\ (\mathit{Nil})\ ((\mathit{rev}\ (l)))))$
definition $\mathit{deref\text{-}assocs}_2 :: ('\mathfrak{A}\ \mathit{state} \times '\mathfrak{A}\ \mathit{state} \Rightarrow '\mathfrak{A}\ \mathit{state})$
$\Rightarrow (\mathit{oid\ list\ list} \Rightarrow \mathit{oid\ list} \times \mathit{oid\ list})$
$\Rightarrow \mathit{oid}$
$\Rightarrow (\mathit{oid\ list} \Rightarrow ('\mathfrak{A},'f)\mathit{val})$
$\Rightarrow \mathit{oid}$
$\Rightarrow ('\mathfrak{A},\ 'f::\mathit{null})\mathit{val}$
where $\mathit{deref\text{-}assocs}_2\ \mathit{pre\text{-}post}\ \mathit{to\text{-}from}\ \mathit{assoc\text{-}oid}\ f\ \mathit{oid} =$
$(\lambda\tau.\ \mathit{case}\ (\mathit{assocs}\ (\mathit{pre\text{-}post}\ \tau))\ \mathit{assoc\text{-}oid}\ \mathit{of}$
$\lfloor\ S\ \rfloor \Rightarrow f\ (\mathit{List\text{-}flatten}\ (\mathit{map}\ (\mathit{choose}_2\text{-}2 \circ \mathit{to\text{-}from})$
$(\mathit{filter}\ (\lambda\ p.\ \mathit{List.member}\ (\mathit{choose}_2\text{-}1\ (\mathit{to\text{-}from}\ p))\ \mathit{oid})\ S)))$
τ
$|\ \text{-} \quad \Rightarrow \mathit{invalid}\ \tau)$

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 $\mathit{switch}_2\text{-}1 = (\lambda[x,y]\Rightarrow (x,y))$
definition $\mathit{switch}_2\text{-}2 = (\lambda[x,y]\Rightarrow (y,x))$
definition $\mathit{switch}_3\text{-}1 = (\lambda[x,y,z]\Rightarrow (x,y))$
definition $\mathit{switch}_3\text{-}2 = (\lambda[x,y,z]\Rightarrow (x,z))$
definition $\mathit{switch}_3\text{-}3 = (\lambda[x,y,z]\Rightarrow (y,x))$
definition $\mathit{switch}_3\text{-}4 = (\lambda[x,y,z]\Rightarrow (y,z))$
definition $\mathit{switch}_3\text{-}5 = (\lambda[x,y,z]\Rightarrow (z,x))$
definition $\mathit{switch}_3\text{-}6 = (\lambda[x,y,z]\Rightarrow (z,y))$

definition $\mathit{select\text{-}object} :: (('\mathfrak{A},\ 'b::\mathit{null})\mathit{val})$
$\Rightarrow (('\mathfrak{A},'b)\mathit{val} \Rightarrow ('\mathfrak{A},'c)\mathit{val} \Rightarrow ('\mathfrak{A},\ 'b)\mathit{val})$
$\Rightarrow (('\mathfrak{A},\ 'b)\mathit{val} \Rightarrow ('\mathfrak{A},\ 'd)\mathit{val})$
$\Rightarrow (\mathit{oid} \Rightarrow ('\mathfrak{A},'c::\mathit{null})\mathit{val})$
$\Rightarrow \mathit{oid\ list}$
$\Rightarrow ('\mathfrak{A},\ 'd)\mathit{val}$
where $\mathit{select\text{-}object}\ \mathit{mt}\ \mathit{incl}\ \mathit{smash}\ \mathit{deref}\ l = \mathit{smash}(\mathit{foldl}\ \mathit{incl}\ \mathit{mt}\ (\mathit{map}\ \mathit{deref}\ l))$
$(*\ \mathit{smash\ returns\ null\ with\ mt\ in\ input}\ (\mathit{in\ this\ case},\ \mathit{object\ contains\ null\ pointer})\ *)$

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 $(\textit{select-object mtSet OCL-collection-type-Set.OclIncluding OclANY } f \quad l \ \textit{oid} \ )::(\text{'}\mathfrak{A}, \text{'}a::\textit{null})\textit{val}$

definition $\textit{deref-oid}_{Person} :: (\mathfrak{A}\ \textit{state} \times \mathfrak{A}\ \textit{state} \Rightarrow \mathfrak{A}\ \textit{state})$
$\Rightarrow (\textit{type}_{Person} \Rightarrow (\mathfrak{A}, \text{'}c::\textit{null})\textit{val})$
$\Rightarrow \textit{oid}$
$\Rightarrow (\mathfrak{A}, \text{'}c::\textit{null})\textit{val}$
where $\textit{deref-oid}_{Person}\ \textit{fst-snd}\ f\ \textit{oid} = (\lambda\tau.\ \textit{case}\ (\textit{heap}\ (\textit{fst-snd}\ \tau))\ \textit{oid of}$
$\lfloor \textit{in}_{Person}\ \textit{obj}\ \rfloor \Rightarrow f\ \textit{obj}\ \tau$
$\mid \text{-} \quad \Rightarrow \textit{invalid}\ \tau)$

definition $\textit{deref-oid}_{OclAny} :: (\mathfrak{A}\ \textit{state} \times \mathfrak{A}\ \textit{state} \Rightarrow \mathfrak{A}\ \textit{state})$
$\Rightarrow (\textit{type}_{OclAny} \Rightarrow (\mathfrak{A}, \text{'}c::\textit{null})\textit{val})$
$\Rightarrow \textit{oid}$
$\Rightarrow (\mathfrak{A}, \text{'}c::\textit{null})\textit{val}$
where $\textit{deref-oid}_{OclAny}\ \textit{fst-snd}\ f\ \textit{oid} = (\lambda\tau.\ \textit{case}\ (\textit{heap}\ (\textit{fst-snd}\ \tau))\ \textit{oid of}$
$\lfloor \textit{in}_{OclAny}\ \textit{obj}\ \rfloor \Rightarrow f\ \textit{obj}\ \tau$
$\mid \text{-} \quad \Rightarrow \textit{invalid}\ \tau)$

pointer undefined in state or not referencing a type conform object representation

definition $\textit{select}_{OclAny}\mathcal{ANY}\ f = (\lambda\ X.\ \textit{case}\ X\ \textit{of}$
$(\textit{mk}_{OclAny}\ \text{-}\ \bot) \Rightarrow \textit{null}$
$\mid (\textit{mk}_{OclAny}\ \text{-}\ \lfloor \textit{any} \rfloor) \Rightarrow f\ (\lambda x\ \text{-}.\ \lfloor\lfloor x \rfloor\rfloor)\ \textit{any})$

definition $\textit{select}_{Person}\mathcal{BOSS}\ f = \textit{select-object mtSet OCL-collection-type-Set.OclIncluding OclANY}\ (f\ (\lambda x\ \text{-}.\ \lfloor\lfloor x \rfloor\rfloor))$

definition $\textit{select}_{Person}\mathcal{SALARY}\ f = (\lambda\ X.\ \textit{case}\ X\ \textit{of}$
$(\textit{mk}_{Person}\ \text{-}\ \bot) \Rightarrow \textit{null}$
$\mid (\textit{mk}_{Person}\ \text{-}\ \lfloor \textit{salary} \rfloor) \Rightarrow f\ (\lambda x\ \text{-}.\ \lfloor\lfloor x \rfloor\rfloor)\ \textit{salary})$

definition $\textit{deref-assocs}_2\mathcal{BOSS}\ \textit{fst-snd}\ f = (\lambda\ \textit{mk}_{Person}\ \textit{oid}\ \text{-} \Rightarrow$
$\textit{deref-assocs}_2\ \textit{fst-snd}\ \textit{switch}_2\textit{-1}\ \textit{oid}_{Person}\mathcal{BOSS}\ f\ \textit{oid})$

definition $\textit{in-pre-state} = \textit{fst}$
definition $\textit{in-post-state} = \textit{snd}$

definition $\textit{reconst-basetype} = (\lambda\ \textit{convert}\ x.\ \textit{convert}\ x)$

definition $\textit{dot}_{OclAny}\mathcal{ANY} :: \textit{OclAny} \Rightarrow \text{-} \quad ((\textit{1}(\text{-}).\textit{any})\ \textit{50})$
where $(X).\textit{any} = \textit{eval-extract}\ X$
$(\textit{deref-oid}_{OclAny}\ \textit{in-post-state}$
$(\textit{select}_{OclAny}\mathcal{ANY}$

reconst-basetype))

definition $dot_{Person}\mathcal{BOSS}$ :: *Person* ⇒ *Person* ((*1*(-).*boss*) *50*)
where (*X*).*boss* = *eval-extract X*
(*deref-oid*$_{Person}$ *in-post-state*
(*deref-assocs*$_2\mathcal{BOSS}$ *in-post-state*
(*select*$_{Person}\mathcal{BOSS}$
(*deref-oid*$_{Person}$ *in-post-state*))))

definition $dot_{Person}\mathcal{SALARY}$ :: *Person* ⇒ *Integer* ((*1*(-).*salary*) *50*)
where (*X*).*salary* = *eval-extract X*
(*deref-oid*$_{Person}$ *in-post-state*
(*select*$_{Person}\mathcal{SALARY}$
reconst-basetype))

definition $dot_{OclAny}\mathcal{ANY}$*-at-pre* :: *OclAny* ⇒ *-* ((*1*(-).*any@pre*) *50*)
where (*X*).*any@pre* = *eval-extract X*
(*deref-oid*$_{OclAny}$ *in-pre-state*
(*select*$_{OclAny}\mathcal{ANY}$
reconst-basetype))

definition $dot_{Person}\mathcal{BOSS}$*-at-pre*:: *Person* ⇒ *Person* ((*1*(-).*boss@pre*) *50*)
where (*X*).*boss@pre* = *eval-extract X*
(*deref-oid*$_{Person}$ *in-pre-state*
(*deref-assocs*$_2\mathcal{BOSS}$ *in-pre-state*
(*select*$_{Person}\mathcal{BOSS}$
(*deref-oid*$_{Person}$ *in-pre-state*))))

definition $dot_{Person}\mathcal{SALARY}$*-at-pre*:: *Person* ⇒ *Integer* ((*1*(-).*salary@pre*) *50*)
where (*X*).*salary@pre* = *eval-extract X*
(*deref-oid*$_{Person}$ *in-pre-state*
(*select*$_{Person}\mathcal{SALARY}$
reconst-basetype))

lemmas [*simp*] =
$dot_{OclAny}\mathcal{ANY}$*-def*
$dot_{Person}\mathcal{BOSS}$*-def*
$dot_{Person}\mathcal{SALARY}$*-def*
$dot_{OclAny}\mathcal{ANY}$*-at-pre-def*
$dot_{Person}\mathcal{BOSS}$*-at-pre-def*
$dot_{Person}\mathcal{SALARY}$*-at-pre-def*

7.8.2. Context Passing

lemmas [*simp*] = *eval-extract-def*

lemma *cp-dot*$_{OclAny}\mathcal{ANY}$: ((*X*).*any*) τ = ((λ-. *X* τ).*any*) τ ⟨*proof*⟩
lemma *cp-dot*$_{Person}\mathcal{BOSS}$: ((*X*).*boss*) τ = ((λ-. *X* τ).*boss*) τ ⟨*proof*⟩
lemma *cp-dot*$_{Person}\mathcal{SALARY}$: ((*X*).*salary*) τ = ((λ-. *X* τ).*salary*) τ ⟨*proof*⟩

lemma *cp-dot*$_{OclAny}$$\mathcal{ANY}$*-at-pre*: $((X).any@pre)\ \tau = ((\lambda\text{-}.\ X\ \tau).any@pre)\ \tau$ ⟨*proof*⟩
lemma *cp-dot*$_{Person}$$\mathcal{BOSS}$*-at-pre*: $((X).boss@pre)\ \tau = ((\lambda\text{-}.\ X\ \tau).boss@pre)\ \tau$ ⟨*proof*⟩
lemma *cp-dot*$_{Person}$$\mathcal{SALARY}$*-at-pre*: $((X).salary@pre)\ \tau = ((\lambda\text{-}.\ X\ \tau).salary@pre)\ \tau$ ⟨*proof*⟩

lemmas *cp-dot*$_{OclAny}$$\mathcal{ANY}$*-I* [*simp*, *intro*!]=
cp-dot$_{OclAny}$$\mathcal{ANY}$[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]
lemmas *cp-dot*$_{OclAny}$$\mathcal{ANY}$*-at-pre-I* [*simp*, *intro*!]=
cp-dot$_{OclAny}$$\mathcal{ANY}$*-at-pre*[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

lemmas *cp-dot*$_{Person}$$\mathcal{BOSS}$*-I* [*simp*, *intro*!]=
cp-dot$_{Person}$$\mathcal{BOSS}$[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]
lemmas *cp-dot*$_{Person}$$\mathcal{BOSS}$*-at-pre-I* [*simp*, *intro*!]=
cp-dot$_{Person}$$\mathcal{BOSS}$*-at-pre*[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

lemmas *cp-dot*$_{Person}$$\mathcal{SALARY}$*-I* [*simp*, *intro*!]=
cp-dot$_{Person}$$\mathcal{SALARY}$[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]
lemmas *cp-dot*$_{Person}$$\mathcal{SALARY}$*-at-pre-I* [*simp*, *intro*!]=
cp-dot$_{Person}$$\mathcal{SALARY}$*-at-pre*[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

7.8.3. Execution with Invalid or Null as Argument

lemma *dot*$_{OclAny}$$\mathcal{ANY}$*-nullstrict* [*simp*]: $(null).any = invalid$
⟨*proof*⟩
lemma *dot*$_{OclAny}$$\mathcal{ANY}$*-at-pre-nullstrict* [*simp*] : $(null).any@pre = invalid$
⟨*proof*⟩
lemma *dot*$_{OclAny}$$\mathcal{ANY}$*-strict* [*simp*] : $(invalid).any = invalid$
⟨*proof*⟩
lemma *dot*$_{OclAny}$$\mathcal{ANY}$*-at-pre-strict* [*simp*] : $(invalid).any@pre = invalid$
⟨*proof*⟩

lemma *dot*$_{Person}$$\mathcal{BOSS}$*-nullstrict* [*simp*]: $(null).boss = invalid$
⟨*proof*⟩
lemma *dot*$_{Person}$$\mathcal{BOSS}$*-at-pre-nullstrict* [*simp*] : $(null).boss@pre = invalid$
⟨*proof*⟩
lemma *dot*$_{Person}$$\mathcal{BOSS}$*-strict* [*simp*] : $(invalid).boss = invalid$
⟨*proof*⟩
lemma *dot*$_{Person}$$\mathcal{BOSS}$*-at-pre-strict* [*simp*] : $(invalid).boss@pre = invalid$
⟨*proof*⟩

lemma *dot*$_{Person}$$\mathcal{SALARY}$*-nullstrict* [*simp*]: $(null).salary = invalid$

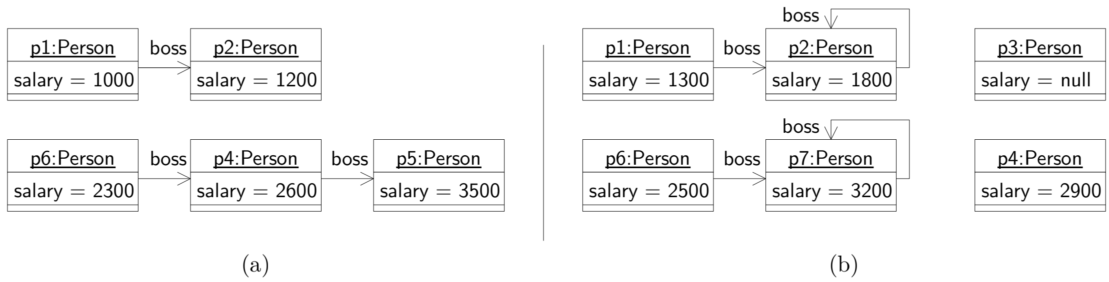


Figure 7.2.: (a) pre-state σ_1 and (b) post-state σ_1'.

$\langle proof \rangle$
lemma $dot_{Person}\mathcal{SALARY}$*-at-pre-nullstrict* [*simp*] : $(null).salary@pre = invalid$
$\langle proof \rangle$
lemma $dot_{Person}\mathcal{SALARY}$*-strict* [*simp*] : $(invalid).salary = invalid$
$\langle proof \rangle$
lemma $dot_{Person}\mathcal{SALARY}$*-at-pre-strict* [*simp*] : $(invalid).salary@pre = invalid$
$\langle proof \rangle$

7.9. A Little Infra-structure on Example States

The example we are defining in this section comes from the figure 7.2.

definition *OclInt1000* (**1000**) **where** $OclInt1000 = (\lambda\ \text{-}\ .\ \lfloor\lfloor 1000 \rfloor\rfloor)$
definition *OclInt1200* (**1200**) **where** $OclInt1200 = (\lambda\ \text{-}\ .\ \lfloor\lfloor 1200 \rfloor\rfloor)$
definition *OclInt1300* (**1300**) **where** $OclInt1300 = (\lambda\ \text{-}\ .\ \lfloor\lfloor 1300 \rfloor\rfloor)$
definition *OclInt1800* (**1800**) **where** $OclInt1800 = (\lambda\ \text{-}\ .\ \lfloor\lfloor 1800 \rfloor\rfloor)$
definition *OclInt2600* (**2600**) **where** $OclInt2600 = (\lambda\ \text{-}\ .\ \lfloor\lfloor 2600 \rfloor\rfloor)$
definition *OclInt2900* (**2900**) **where** $OclInt2900 = (\lambda\ \text{-}\ .\ \lfloor\lfloor 2900 \rfloor\rfloor)$
definition *OclInt3200* (**3200**) **where** $OclInt3200 = (\lambda\ \text{-}\ .\ \lfloor\lfloor 3200 \rfloor\rfloor)$
definition *OclInt3500* (**3500**) **where** $OclInt3500 = (\lambda\ \text{-}\ .\ \lfloor\lfloor 3500 \rfloor\rfloor)$

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 $person1 \equiv mk_{Person}\ oid0\ \lfloor 1300 \rfloor$
definition $person2 \equiv mk_{Person}\ oid1\ \lfloor 1800 \rfloor$
definition $person3 \equiv mk_{Person}\ oid2\ None$
definition $person4 \equiv mk_{Person}\ oid3\ \lfloor 2900 \rfloor$
definition $person5 \equiv mk_{Person}\ oid4\ \lfloor 3500 \rfloor$

definition $person6 \equiv mk_{Person}\ oid5\ \lfloor 2500 \rfloor$
definition $person7 \equiv mk_{OclAny}\ oid6\ \lfloor\lfloor 3200 \rfloor\rfloor$
definition $person8 \equiv mk_{OclAny}\ oid7\ None$
definition $person9 \equiv mk_{Person}\ oid8\ \lfloor 0 \rfloor$

definition
$\sigma_1 \equiv (\!|\ heap = empty(oid0 \mapsto in_{Person}\ (mk_{Person}\ oid0\ \lfloor 1000 \rfloor))$
$(oid1 \mapsto in_{Person}\ (mk_{Person}\ oid1\ \lfloor 1200 \rfloor))$
$(*oid2*)$
$(oid3 \mapsto in_{Person}\ (mk_{Person}\ oid3\ \lfloor 2600 \rfloor))$
$(oid4 \mapsto in_{Person}\ person5)$
$(oid5 \mapsto in_{Person}\ (mk_{Person}\ oid5\ \lfloor 2300 \rfloor))$
$(*oid6*)$
$(*oid7*)$
$(oid8 \mapsto in_{Person}\ person9),$
$assocs = empty(oid_{Person}\mathcal{BOSS} \mapsto [[[oid0],[oid1]],[[oid3],[oid4]],[[oid5],[oid3]]])\ |\!)$

definition
$\sigma_1{}' \equiv (\!|\ heap = empty(oid0 \mapsto in_{Person}\ person1)$
$(oid1 \mapsto in_{Person}\ person2)$
$(oid2 \mapsto in_{Person}\ person3)$
$(oid3 \mapsto in_{Person}\ person4)$
$(*oid4*)$
$(oid5 \mapsto in_{Person}\ person6)$
$(oid6 \mapsto in_{OclAny}\ person7)$
$(oid7 \mapsto in_{OclAny}\ person8)$
$(oid8 \mapsto in_{Person}\ person9),$
$assocs = empty(oid_{Person}\mathcal{BOSS} \mapsto$
$[[[oid0],[oid1]],[[oid1],[oid1]],[[oid5],[oid6]],[[oid6],[oid6]]])\ |\!)$

definition $\sigma_0 \equiv (\!|\ heap = empty,\ assocs = empty\ |\!)$

lemma *basic-τ-wff*: $WFF(\sigma_1,\sigma_1{}')$
⟨*proof*⟩

lemma [*simp*,*code-unfold*]: $dom\ (heap\ \sigma_1) = \{oid0,oid1,(*,oid2*)oid3,oid4,oid5(*,oid6,oid7*),oid8\}$
⟨*proof*⟩

lemma [*simp*,*code-unfold*]: $dom\ (heap\ \sigma_1{}') = \{oid0,oid1,oid2,oid3,(*,oid4*)oid5,oid6,oid7,oid8\}$
⟨*proof*⟩

definition $X_{Person}1 :: Person \equiv \lambda\ \text{-}\ .\lfloor\lfloor\ person1\ \rfloor\rfloor$
definition $X_{Person}2 :: Person \equiv \lambda\ \text{-}\ .\lfloor\lfloor\ person2\ \rfloor\rfloor$
definition $X_{Person}3 :: Person \equiv \lambda\ \text{-}\ .\lfloor\lfloor\ person3\ \rfloor\rfloor$
definition $X_{Person}4 :: Person \equiv \lambda\ \text{-}\ .\lfloor\lfloor\ person4\ \rfloor\rfloor$
definition $X_{Person}5 :: Person \equiv \lambda\ \text{-}\ .\lfloor\lfloor\ person5\ \rfloor\rfloor$
definition $X_{Person}6 :: Person \equiv \lambda\ \text{-}\ .\lfloor\lfloor\ person6\ \rfloor\rfloor$
definition $X_{Person}7 :: OclAny \equiv \lambda\ \text{-}\ .\lfloor\lfloor\ person7\ \rfloor\rfloor$

definition $X_{Person}8$ :: $OclAny \equiv \lambda$ - .$\lfloor\lfloor$ $person8$ $\rfloor\rfloor$
definition $X_{Person}9$ :: $Person \equiv \lambda$ - .$\lfloor\lfloor$ $person9$ $\rfloor\rfloor$

lemma [*code-unfold*]: $((x::Person) \doteq y) = StrictRefEq_{Object}\ x\ y$ $\langle proof \rangle$
lemma [*code-unfold*]: $((x::OclAny) \doteq y) = StrictRefEq_{Object}\ x\ y$ $\langle proof \rangle$

lemmas [*simp*,*code-unfold*] =
$OclAsType_{OclAny}$-$OclAny$
$OclAsType_{OclAny}$-$Person$
$OclAsType_{Person}$-$OclAny$
$OclAsType_{Person}$-$Person$

$OclIsTypeOf_{OclAny}$-$OclAny$
$OclIsTypeOf_{OclAny}$-$Person$
$OclIsTypeOf_{Person}$-$OclAny$
$OclIsTypeOf_{Person}$-$Person$

$OclIsKindOf_{OclAny}$-$OclAny$
$OclIsKindOf_{OclAny}$-$Person$
$OclIsKindOf_{Person}$-$OclAny$
$OclIsKindOf_{Person}$-$Person$

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models (X_{Person}1\ .salary <> \mathbf{1000})$
Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models (X_{Person}1\ .salary \doteq \mathbf{1300})$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}1\ .salary@pre \doteq \mathbf{1000})$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}1\ .salary@pre <> \mathbf{1300})$

lemma $(\sigma_1,\sigma_1{}') \models (X_{Person}1\ .oclIsMaintained())$
$\langle proof \rangle$

lemma $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models ((X_{Person}1\ .oclAsType(OclAny)\ .oclAsType(Person)) \doteq X_{Person}1)$
$\langle proof \rangle$
Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models (X_{Person}1\ .oclIsTypeOf(Person))$
Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models not(X_{Person}1\ .oclIsTypeOf(OclAny))$
Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models (X_{Person}1\ .oclIsKindOf(Person))$
Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models (X_{Person}1\ .oclIsKindOf(OclAny))$
Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models not(X_{Person}1\ .oclAsType(OclAny)\ .oclIsTypeOf(OclAny))$

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models (X_{Person}2\ .salary \doteq \mathbf{1800})$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}2\ .salary@pre \doteq \mathbf{1200})$

lemma $(\sigma_1,\sigma_1{}') \models (X_{Person}2\ .oclIsMaintained())$
$\langle proof \rangle$

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1') \models (X_{Person}3\ .salary \doteq null)$
Assert $\bigwedge\ s_{post}.\ (\sigma_1,s_{post}) \models not(\upsilon(X_{Person}3\ .salary@pre))$
lemma $(\sigma_1,\sigma_1') \models (X_{Person}3\ .oclIsNew())$
$\langle proof \rangle$

lemma $(\sigma_1,\sigma_1') \models (X_{Person}4\ .oclIsMaintained())$
$\langle proof \rangle$

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1') \models not(\upsilon(X_{Person}5\ .salary))$
Assert $\bigwedge\ s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}5\ .salary@pre \doteq \mathbf{3500})$

lemma $(\sigma_1,\sigma_1') \models (X_{Person}5\ .oclIsDeleted())$
$\langle proof \rangle$

lemma $(\sigma_1,\sigma_1') \models (X_{Person}6\ .oclIsMaintained())$
$\langle proof \rangle$

Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models \upsilon(X_{Person}7\ .oclAsType(Person))$

lemma $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models ((X_{Person}7\ .oclAsType(Person)\ .oclAsType(OclAny)$
$.oclAsType(Person))$
$\doteq (X_{Person}7\ .oclAsType(Person)))$
$\langle proof \rangle$
lemma $(\sigma_1,\sigma_1') \models (X_{Person}7\ .oclIsNew())$
$\langle proof \rangle$

Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models (X_{Person}8\ <>\ X_{Person}7)$
Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models not(\upsilon(X_{Person}8\ .oclAsType(Person)))$
Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models (X_{Person}8\ .oclIsTypeOf(OclAny))$
Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models not(X_{Person}8\ .oclIsTypeOf(Person))$
Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models not(X_{Person}8\ .oclIsKindOf(Person))$
Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models (X_{Person}8\ .oclIsKindOf(OclAny))$

lemma σ*-modifiedonly*: $(\sigma_1,\sigma_1') \models (Set\{\ X_{Person}1\ .oclAsType(OclAny)$
$,\ X_{Person}2\ .oclAsType(OclAny)$
$(*,\ X_{Person}3\ .oclAsType(OclAny)*)$
$,\ X_{Person}4\ .oclAsType(OclAny)$

$(*, X_{Person}5 \ .oclAsType(OclAny)*)$
$, X_{Person}6 \ .oclAsType(OclAny)$
$(*, X_{Person}7 \ .oclAsType(OclAny)*)$
$(*, X_{Person}8 \ .oclAsType(OclAny)*)$
$(*, X_{Person}9 \ .oclAsType(OclAny)*)\}{-}{>}oclIsModifiedOnly())$
⟨*proof*⟩

lemma $(\sigma_1,\sigma_1') \models ((X_{Person}9 \ @pre \ (\lambda x. \ \lfloor OclAsType_{Person}\text{-}\mathfrak{A} \ x\rfloor)) \ \triangleq X_{Person}9)$
⟨*proof*⟩

lemma $(\sigma_1,\sigma_1') \models ((X_{Person}9 \ @post \ (\lambda x. \ \lfloor OclAsType_{Person}\text{-}\mathfrak{A} \ x\rfloor)) \ \triangleq X_{Person}9)$
⟨*proof*⟩

lemma $(\sigma_1,\sigma_1') \models (((X_{Person}9 \ .oclAsType(OclAny)) \ @pre \ (\lambda x. \ \lfloor OclAsType_{OclAny}\text{-}\mathfrak{A} \ x\rfloor)) \triangleq$
$((X_{Person}9 \ .oclAsType(OclAny)) \ @post \ (\lambda x. \ \lfloor OclAsType_{OclAny}\text{-}\mathfrak{A} \ x\rfloor)))$
⟨*proof*⟩

lemma *perm-*$\sigma_1{}'$: $\sigma_1{}' = (\!| \ heap = empty$
$(oid8 \mapsto in_{Person} \ person9)$
$(oid7 \mapsto in_{OclAny} \ person8)$
$(oid6 \mapsto in_{OclAny} \ person7)$
$(oid5 \mapsto in_{Person} \ person6)$
$(*oid4*)$
$(oid3 \mapsto in_{Person} \ person4)$
$(oid2 \mapsto in_{Person} \ person3)$
$(oid1 \mapsto in_{Person} \ person2)$
$(oid0 \mapsto in_{Person} \ person1)$
$, \ assocs = assocs \ \sigma_1{}' \ |\!)$
⟨*proof*⟩

declare *const-ss* [*simp*]

lemma $\bigwedge\sigma_1.$
$(\sigma_1,\sigma_1') \models (Person \ .allInstances() \doteq Set\{ \ X_{Person}1, \ X_{Person}2, \ X_{Person}3, \ X_{Person}4(*,$
$X_{Person}5*), \ X_{Person}6,$
$X_{Person}7 \ .oclAsType(Person)(*, \ X_{Person}8*), \ X_{Person}9 \ \})$
⟨*proof*⟩

lemma $\bigwedge\sigma_1.$
$(\sigma_1,\sigma_1') \models (OclAny \ .allInstances() \doteq Set\{ \ X_{Person}1 \ .oclAsType(OclAny), \ X_{Person}2$
$.oclAsType(OclAny),$
$X_{Person}3 \ .oclAsType(OclAny), \ X_{Person}4 \ .oclAsType(OclAny)$
$(*, \ X_{Person}5*), \ X_{Person}6 \ .oclAsType(OclAny),$
$X_{Person}7, \ X_{Person}8, \ X_{Person}9 \ .oclAsType(OclAny) \ \})$
⟨*proof*⟩

end

theory
Employee-AnalysisModel-OCLPart
imports
Employee-AnalysisModel-UMLPart
begin

7.10. OCL Part: Standard State Infrastructure

Ideally, these definitions are automatically generated from the class model.

7.11. 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 $\textit{Person-label}_{inv} :: \textit{Person} \Rightarrow \textit{Boolean}$
where $\textit{Person-label}_{inv}\ (\textit{self}) \equiv$
$(\textit{self}\ .\textit{boss} <> \textit{null implies}\ (\textit{self}\ .\textit{salary}\ \leq_{int}\ ((\textit{self}\ .\textit{boss})\ .\textit{salary})))$

definition $\textit{Person-label}_{invATpre} :: \textit{Person} \Rightarrow \textit{Boolean}$
where $\textit{Person-label}_{invATpre}\ (\textit{self}) \equiv$
$(\textit{self}\ .\textit{boss@pre} <> \textit{null implies}\ (\textit{self}\ .\textit{salary@pre} \leq_{int} ((\textit{self}\ .\textit{boss@pre})\ .\textit{salary@pre})))$

definition $\textit{Person-label}_{globalinv} :: \textit{Boolean}$
where $\textit{Person-label}_{globalinv} \equiv (\textit{Person}\ .\textit{allInstances}()\text{−>}\textit{forAll}(x \mid \textit{Person-label}_{inv}\ (x))\ \textit{and}$
$(\textit{Person}\ .\textit{allInstances@pre}()\text{−>}\textit{forAll}(x \mid \textit{Person-label}_{invATpre}\ (x))))$

lemma $\tau \models \delta\ (X\ .\textit{boss}) \Longrightarrow \tau \models \textit{Person}\ .\textit{allInstances}()\text{−>}\textit{includes}(X\ .\textit{boss}) \wedge$
$\tau \models \textit{Person}\ .\textit{allInstances}()\text{−>}\textit{includes}(X)$
⟨*proof*⟩

lemma $\textit{REC-pre} : \tau \models \textit{Person-label}_{globalinv}$
$\Longrightarrow \tau \models \textit{Person}\ .\textit{allInstances}()\text{−>}\textit{includes}(X)$ (∗ *X represented object in state* ∗)
$\Longrightarrow \exists\ \textit{REC}.\ \tau \models \textit{REC}(X) \triangleq (\textit{Person-label}_{inv}\ (X)\ \textit{and}\ (X\ .\textit{boss} <> \textit{null implies REC}(X\ .\textit{boss})))$

⟨*proof*⟩

This allows to state a predicate:

axiomatization $inv_{Person\text{-}label}$ :: $Person \Rightarrow Boolean$
where $inv_{Person\text{-}label}\text{-}def$:
$(\tau \models Person\ .allInstances() -> includes(self)) \Longrightarrow$
$(\tau \models (inv_{Person\text{-}label}(self) \triangleq (self\ .boss <> null\ implies$
$(self\ .salary \leq_{int} ((self\ .boss)\ .salary))\ and$
$inv_{Person\text{-}label}(self\ .boss))))$

axiomatization $inv_{Person\text{-}labelATpre}$ :: $Person \Rightarrow Boolean$
where $inv_{Person\text{-}labelATpre}\text{-}def$:
$(\tau \models Person\ .allInstances@pre() -> includes(self)) \Longrightarrow$
$(\tau \models (inv_{Person\text{-}labelATpre}(self) \triangleq (self\ .boss@pre <> null\ implies$
$(self\ .salary@pre \leq_{int} ((self\ .boss@pre)\ .salary@pre))\ and$
$inv_{Person\text{-}labelATpre}(self\ .boss@pre))))$

lemma *inv-1* :
$(\tau \models Person\ .allInstances() -> includes(self)) \Longrightarrow$
$(\tau \models inv_{Person\text{-}label}(self) = ((\tau \models (self\ .boss \doteq null)) \vee$
$(\tau \models (self\ .boss <> null) \wedge$
$\tau \models ((self\ .salary) \leq_{int} (self\ .boss\ .salary)) \wedge$
$\tau \models (inv_{Person\text{-}label}(self\ .boss)))))$
⟨*proof*⟩

lemma *inv-2* :
$(\tau \models Person\ .allInstances@pre() -> includes(self)) \Longrightarrow$
$(\tau \models inv_{Person\text{-}labelATpre}(self)) = ((\tau \models (self\ .boss@pre \doteq null)) \vee$
$(\tau \models (self\ .boss@pre <> null) \wedge$
$(\tau \models (self\ .boss@pre\ .salary@pre \leq_{int} self\ .salary@pre)) \wedge$
$(\tau \models (inv_{Person\text{-}labelATpre}(self\ .boss@pre)))))$
⟨*proof*⟩

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

coinductive inv :: $Person \Rightarrow (\mathfrak{A})st \Rightarrow bool$ **where**
$(\tau \models (\delta\ self)) \Longrightarrow ((\tau \models (self\ .boss \doteq null)) \vee$
$(\tau \models (self\ .boss <> null) \wedge (\tau \models (self\ .boss\ .salary \leq_{int} self\ .salary)) \wedge$
$((inv(self\ .boss))\tau)))$
$\Longrightarrow (inv\ self\ \tau)$

7.12. The Contract of a Recursive Query

The original specification of a recursive query :

```
context Person::contents():Set(Integer)
pre:   true
```

```
post:  result = if self.boss = null
                then Set{i}
                else self.boss.contents()->including(i)
                endif
```

For the case of recursive queries, we use at present just axiomatizations:

axiomatization *contents* :: *Person* ⇒ *Set-Integer* ((*1*(-).*contents*′(′)) *50*)
where *contents-def*:
$(self\ .contents()) = (\lambda\ \tau.\ (if\ \tau \models (\delta\ self)$
$then\ SOME\ res.((\tau \models true) \wedge$
$(\tau \models (\lambda\text{-}\ .\ res) \triangleq if\ (self\ .boss \doteq null)$
$then\ (Set\{self\ .salary\})$
$else\ (self\ .boss\ .contents()$
$->including(self\ .salary))$
$endif))$
$else\ invalid\ \tau))$

declare *Employee-AnalysisModel-UMLPart.dot*$_{Person}\mathcal{SALARY}$*-def* [*simp del*]
declare *Employee-AnalysisModel-UMLPart.dot*$_{Person}\mathcal{BOSS}$*-def* [*simp del*]

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*⟩

Specializing $[\![cp\ E;\ \tau \models \delta\ self;\ \tau \models true;\ \tau \models POST'\ self;\ \bigwedge res.\ (res \triangleq if\ self.boss \doteq null\ then\ Set\{self.salary\}\ else\ self.boss.contents()->including(self.salary)\ endif) = (POST'\ self\ and\ (res \triangleq BODY\ self))]\!] \Longrightarrow (\tau \models E\ (self.contents())) = (\tau \models E\ (BODY\ self))$, one gets the following more practical rewrite rule that is amenable to symbolic evaluation:

theorem *unfold-contents* :
assumes $cp\ E$
and $\tau \models \delta\ self$
shows $(\tau \models E\ (self\ .contents())) =$
$(\tau \models E\ (if\ self\ .boss \doteq null$
$then\ Set\{self\ .salary\}$
$else\ self\ .boss\ .contents()->including(self\ .salary)\ endif))$
⟨*proof*⟩

Since we have only one interpretation function, we need the corresponding operation on the pre-state:

consts *contentsATpre* :: *Person* ⇒ *Set-Integer* ((*1*(-).*contents*@*pre*′(′)) *50*)

axiomatization where *contentsATpre-def*:
$(self).contents@pre() = (\lambda\ \tau.$

$(if\ \tau \models (\delta\ self)$
$then\ SOME\ res.((\tau \models true) \land$ $(*\ pre\ *)$
$(\tau \models ((\lambda\text{-}.\ res) \triangleq if\ (self).boss@pre \doteq null$ $(*\ post\ *)$
$then\ Set\{(self).salary@pre\}$
$else\ (self).boss@pre\ .contents@pre()$
$->including(self\ .salary@pre)$
$endif)))$
$else\ invalid\ \tau))$

declare *Employee-AnalysisModel-UMLPart.dot*$_{Person}$$\mathcal{SALARY}$*-at-pre-def* [*simp del*]
declare *Employee-AnalysisModel-UMLPart.dot*$_{Person}$$\mathcal{BOSS}$*-at-pre-def* [*simp del*]

interpretation *contentsATpre* : *contract0 contentsATpre* λ *self. true*
$\lambda\ self\ res.\ \ res \triangleq if\ (self\ .boss@pre \doteq null)$
$then\ (Set\{self\ .salary@pre\})$
$else\ (self\ .boss@pre\ .contents@pre()$
$->including(self\ .salary@pre))$
$endif$
$\langle proof \rangle$

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 $(\tau \models E\ (self\ .contents@pre())) =$
$(\tau \models E\ (if\ self\ .boss@pre \doteq null$
$then\ Set\{self\ .salary@pre\}$
$else\ self\ .boss@pre\ .contents@pre()->including(self\ .salary@pre)\ endif))$
$\langle proof \rangle$

Note that these `@pre` variants on methods are only available on queries, i. e., operations without side-effect.

7.13. 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* $((1(\text{-}).insert'(\text{-}'))\ 50)$
where $self\ .insert(x) \equiv$
$(\lambda\ \tau.\ if\ (\tau \models (\delta\ self)) \land\ (\tau \models \upsilon\ x)$
$then\ SOME\ res.\ (\tau \models true \land$

$(\tau \models ((self).contents() \triangleq (self).contents@pre()\text{−>}including(x))))$
$else\ invalid\ \tau)$

The semantic consequences of this definition were computed inside this locale interpretation:

interpretation $insert : contract1\ insert\ \lambda\ self\ x.\ true$
$\lambda\ self\ x\ res.\ ((self\ .contents()) \triangleq$
$(self\ .contents@pre()\text{−>}including(x)))$
$\langle proof \rangle$

The result of this locale interpretation for our *Employee-AnalysisModel-OCLPart.insert* contract is the following set of properties, which serves as basis for automated deduction on them:

Name	Theorem
$insert.strict0$	$(invalid.insert(X)) = invalid$
$insert.nullstrict0$	$(null.insert(X)) = invalid$
$insert.strict1$	$(self.insert(invalid)) = invalid$
$insert.cp_{PRE}$	$true\ \tau = true\ \tau$
$insert.cp_{POST}$	$(self.contents() \triangleq self.contents@pre()\text{−>}including(a1.0))\ \tau = (\lambda\text{-}.\ self\ \tau.contents() \triangleq \lambda\text{-}.\ self\ \tau.contents@pre()\text{−>}including(\lambda\text{-}.\ a1.0\ \tau))\ \tau$
$insert.cp\text{-}pre$	$[\![cp\ self'; cp\ a1']\!] \Longrightarrow cp\ (\lambda X.\ true)$
$insert.cp\text{-}post$	$[\![cp\ self'; cp\ a1'; cp\ res']\!] \Longrightarrow cp\ (\lambda X.\ self'\ X.contents() \triangleq self'\ X.contents@pre()\text{−>}including(a1'\ X))$
$insert.cp$	$[\![cp\ self'; cp\ a1'; cp\ res']\!] \Longrightarrow cp\ (\lambda X.\ self'\ X.insert(a1'\ X))$
$insert.cp0$	$(self.insert(a1.0))\ \tau = (\lambda\text{-}.\ self\ \tau.insert(\lambda\text{-}.\ a1.0\ \tau))\ \tau$
$insert.def\text{-}scheme$	$self.insert(a1.0) \equiv \lambda\tau.\ if\ \tau \models \delta\ self \wedge \tau \models \upsilon\ a1.0\ then\ SOME\ res.\ \tau \models true \wedge \tau \models self.contents() \triangleq self.contents@pre()\text{−>}including(a1.0)\ else\ invalid\ \tau$
$insert.unfold$	$[\![cp\ E; \tau \models \delta\ self \wedge \tau \models \upsilon\ a1.0; \tau \models true; \exists res.\ \tau \models self.contents() \triangleq self.contents@pre()\text{−>}including(a1.0); \bigwedge res.\ \tau \models self.contents() \triangleq self.contents@pre()\text{−>}including(a1.0) \Longrightarrow \tau \models E\ (\lambda\text{-}.\ res)]\!] \Longrightarrow \tau \models E\ (self.insert(a1.0))$
$insert.unfold2$	$[\![cp\ E; \tau \models \delta\ self \wedge \tau \models \upsilon\ a1.0; \tau \models true; \tau \models POST'\ self\ a1.0; \bigwedge res.\ (self.contents() \triangleq self.contents@pre()\text{−>}including(a1.0)) = (POST'\ self\ a1.0\ and\ (res \triangleq BODY\ self\ a1.0))]\!] \Longrightarrow (\tau \models E\ (self.insert(a1.0))) = (\tau \models E\ (BODY\ self\ a1.0))$

Table 7.1.: Basic semantic constant definitions of the logic (except *null*)

end

8. Example II: The Employee Design Model (UML)

theory
Employee-DesignModel-UMLPart
imports
../src/OCL-main
begin

8.1. 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.

8.1.1. Outlining the Example

We are presenting here a "design-model" of the (slightly modified) example Figure 7.3, page 20 of the OCL standard [30]. To be precise, this theory contains the formalization of the data-part covered by the UML class model (see Figure 8.1):

This means that the association (attached to the association class `EmployeeRanking`) with the association ends `boss` and `employees` is implemented by the attribute `boss` and the operation `employees` (to be discussed in the OCL part captured by the subsequent theory).

8.2. Example Data-Universe and its Infrastructure

Ideally, the following is generated automatically from a UML class model.

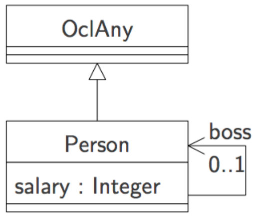


Figure 8.1.: A simple UML class model drawn from Figure 7.3, page 20 of [30].

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 $\mathit{type}_{Person} = \mathit{mk}_{Person}\ \mathit{oid}$
$\quad\mathit{int\ option}$
$\quad\mathit{oid\ option}$

datatype $\mathit{type}_{OclAny} = \mathit{mk}_{OclAny}\ \mathit{oid}$
$\quad(\mathit{int\ option} \times \mathit{oid\ option})\ \mathit{option}$

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 $\mathfrak{A} = \mathit{in}_{Person}\ \mathit{type}_{Person} \mid \mathit{in}_{OclAny}\ \mathit{type}_{OclAny}$

Having fixed the object universe, we can introduce type synonyms that exactly correspond to OCL types. Again, we exploit that our representation of OCL is a "shallow embedding" with a one-to-one correspondance of OCL-types to types of the meta-language HOL.

type-synonym *Boolean* $= \mathfrak{A}\ \mathit{Boolean}$
type-synonym *Integer* $= \mathfrak{A}\ \mathit{Integer}$
type-synonym *Void* $= \mathfrak{A}\ \mathit{Void}$
type-synonym *OclAny* $= (\mathfrak{A}, \mathit{type}_{OclAny}\ \mathit{option\ option})\ \mathit{val}$
type-synonym *Person* $= (\mathfrak{A}, \mathit{type}_{Person}\ \mathit{option\ option})\ \mathit{val}$
type-synonym *Set-Integer* $= (\mathfrak{A}, \mathit{int\ option\ option})\ \mathit{Set}$
type-synonym *Set-Person* $= (\mathfrak{A}, \mathit{type}_{Person}\ \mathit{option\ option})\ \mathit{Set}$

Just a little check:

typ *Boolean*

To reuse key-elements of the library like referential equality, we have to show that the object universe belongs to the type class "oclany," i. e., each class type has to provide a function *oid-of* yielding the object id (oid) of the object.

instantiation $type_{Person}$ :: *object*
begin
  definition *oid-of-type*$_{Person}$*-def*: *oid-of* x = (*case* x *of* mk_{Person} *oid* - - ⇒ *oid*)
  instance ⟨*proof*⟩
end

instantiation $type_{OclAny}$ :: *object*
begin
  definition *oid-of-type*$_{OclAny}$*-def*: *oid-of* x = (*case* x *of* mk_{OclAny} *oid* - ⇒ *oid*)
  instance ⟨*proof*⟩
end

instantiation $\mathfrak{A}$ :: *object*
begin
  definition *oid-of-*$\mathfrak{A}$*-def*: *oid-of* x = (*case* x *of*
    in_{Person} *person* ⇒ *oid-of person*
    | in_{OclAny} *oclany* ⇒ *oid-of oclany*)
  instance ⟨*proof*⟩
end

8.3. Instantiation of the Generic Strict Equality

We instantiate the referential equality on *Person* and *OclAny*

defs(**overloaded**) $StrictRefEq_{Object}\text{-}_{Person}$: $(x::Person) \doteq y \equiv StrictRefEq_{Object}\ x\ y$
defs(**overloaded**) $StrictRefEq_{Object}\text{-}_{OclAny}$: $(x::OclAny) \doteq y \equiv StrictRefEq_{Object}\ x\ y$

lemmas
  cp-StrictRefEq$_{Object}$[*of* x::*Person* y::*Person* τ,
    simplified StrictRefEq$_{Object}$*-*$_{Person}$[*symmetric*]]
  cp-intro(*9*) [*of* P::*Person* ⇒*PersonQ*::*Person* ⇒*Person*,
    simplified StrictRefEq$_{Object}$*-*$_{Person}$[*symmetric*]]
  StrictRefEq$_{Object}$*-def* [*of* x::*Person* y::*Person*,
    simplified StrictRefEq$_{Object}$*-*$_{Person}$[*symmetric*]]
  StrictRefEq$_{Object}$*-defargs* [*of* - x::*Person* y::*Person*,
    simplified StrictRefEq$_{Object}$*-*$_{Person}$[*symmetric*]]
  StrictRefEq$_{Object}$*-strict1*
    [*of* x::*Person*,
    simplified StrictRefEq$_{Object}$*-*$_{Person}$[*symmetric*]]
  StrictRefEq$_{Object}$*-strict2*
    [*of* x::*Person*,
    simplified StrictRefEq$_{Object}$*-*$_{Person}$[*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.

8.4. OclAsType

8.4.1. Definition

consts $OclAsType_{OclAny}$ :: $'\alpha \Rightarrow OclAny$ ((-) $.oclAsType'(OclAny')$)
consts $OclAsType_{Person}$ :: $'\alpha \Rightarrow Person$ ((-) $.oclAsType'(Person')$)

definition $OclAsType_{OclAny}\text{-}\mathfrak{A} = (\lambda u.\ \lfloor case\ u\ of\ in_{OclAny}\ a \Rightarrow a$
$\quad | \ in_{Person}\ (mk_{Person}\ oid\ a\ b) \Rightarrow mk_{OclAny}\ oid\ \lfloor(a,b)\rfloor\rfloor)$

lemma $OclAsType_{OclAny}\text{-}\mathfrak{A}\text{-}some$: $OclAsType_{OclAny}\text{-}\mathfrak{A}\ x \neq None$
⟨*proof*⟩

defs (**overloaded**) $OclAsType_{OclAny}\text{-}OclAny$:
$(X::OclAny)\ .oclAsType(OclAny) \equiv X$

defs (**overloaded**) $OclAsType_{OclAny}\text{-}Person$:
$(X::Person)\ .oclAsType(OclAny) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\quad \bot \ \Rightarrow invalid\ \tau$
$\quad | \lfloor\bot\rfloor \Rightarrow null\ \tau$
$\quad | \lfloor\lfloor mk_{Person}\ oid\ a\ b\ \rfloor\rfloor \Rightarrow\ \lfloor\lfloor\ (mk_{OclAny}\ oid\ \lfloor(a,b)\rfloor)\ \rfloor\rfloor)$

definition $OclAsType_{Person}\text{-}\mathfrak{A} = (\lambda u.\ case\ u\ of\ in_{Person}\ p \Rightarrow \lfloor p\rfloor$
$\quad | \ in_{OclAny}\ (mk_{OclAny}\ oid\ \lfloor(a,b)\rfloor) \Rightarrow \lfloor mk_{Person}\ oid\ a\ b\rfloor$
$\quad | \ \text{-} \Rightarrow None)$

defs (**overloaded**) $OclAsType_{Person}\text{-}OclAny$:
$(X::OclAny)\ .oclAsType(Person) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\quad \bot \ \Rightarrow invalid\ \tau$
$\quad | \lfloor\bot\rfloor \Rightarrow null\ \tau$
$\quad | \lfloor\lfloor mk_{OclAny}\ oid\ \bot\ \rfloor\rfloor \Rightarrow\ invalid\ \tau\quad (*\ down{-}cast\ exception\ *)$
$\quad | \lfloor\lfloor mk_{OclAny}\ oid\ \lfloor(a,b)\rfloor\ \rfloor\rfloor \Rightarrow\ \lfloor\lfloor mk_{Person}\ oid\ a\ b\ \rfloor\rfloor)$

defs (**overloaded**) $OclAsType_{Person}\text{-}Person$:
$(X::Person)\ .oclAsType(Person) \equiv X$

lemmas [*simp*] =
$OclAsType_{OclAny}\text{-}OclAny$
$OclAsType_{Person}\text{-}Person$

8.4.2. Context Passing

lemma $cp\text{-}OclAsType_{OclAny}\text{-}Person\text{-}Person$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X::Person)::Person)\ .oclAsType(OclAny))$
⟨*proof*⟩
lemma $cp\text{-}OclAsType_{OclAny}\text{-}OclAny\text{-}OclAny$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X::OclAny)::OclAny)\ .oclAsType(OclAny))$

⟨*proof*⟩
lemma $cp\text{-}OclAsType_{Person}\text{-}Person\text{-}Person$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X::Person)::Person)$ $.oclAsType(Person))$
⟨*proof*⟩
lemma $cp\text{-}OclAsType_{Person}\text{-}OclAny\text{-}OclAny$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X::OclAny)::OclAny)$ $.oclAsType(Person))$
⟨*proof*⟩

lemma $cp\text{-}OclAsType_{OclAny}\text{-}Person\text{-}OclAny$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X::Person)::OclAny)$ $.oclAsType(OclAny))$
⟨*proof*⟩
lemma $cp\text{-}OclAsType_{OclAny}\text{-}OclAny\text{-}Person$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X::OclAny)::Person)$ $.oclAsType(OclAny))$
⟨*proof*⟩
lemma $cp\text{-}OclAsType_{Person}\text{-}Person\text{-}OclAny$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X::Person)::OclAny)$ $.oclAsType(Person))$
⟨*proof*⟩
lemma $cp\text{-}OclAsType_{Person}\text{-}OclAny\text{-}Person$: $cp\ P \Longrightarrow cp(\lambda X.\ (P\ (X::OclAny)::Person)$ $.oclAsType(Person))$
⟨*proof*⟩

lemmas [*simp*] =
$cp\text{-}OclAsType_{OclAny}\text{-}Person\text{-}Person$
$cp\text{-}OclAsType_{OclAny}\text{-}OclAny\text{-}OclAny$
$cp\text{-}OclAsType_{Person}\text{-}Person\text{-}Person$
$cp\text{-}OclAsType_{Person}\text{-}OclAny\text{-}OclAny$

$cp\text{-}OclAsType_{OclAny}\text{-}Person\text{-}OclAny$
$cp\text{-}OclAsType_{OclAny}\text{-}OclAny\text{-}Person$
$cp\text{-}OclAsType_{Person}\text{-}Person\text{-}OclAny$
$cp\text{-}OclAsType_{Person}\text{-}OclAny\text{-}Person$

8.4.3. Execution with Invalid or Null as Argument

lemma $OclAsType_{OclAny}\text{-}OclAny\text{-}strict$: $(invalid::OclAny)\ .oclAsType(OclAny) = invalid$
⟨*proof*⟩

lemma $OclAsType_{OclAny}\text{-}OclAny\text{-}nullstrict$: $(null::OclAny)\ .oclAsType(OclAny) = null$
⟨*proof*⟩

lemma $OclAsType_{OclAny}\text{-}Person\text{-}strict[simp]$: $(invalid::Person)\ .oclAsType(OclAny) = invalid$
⟨*proof*⟩

lemma $OclAsType_{OclAny}\text{-}Person\text{-}nullstrict[simp]$: $(null::Person)\ .oclAsType(OclAny) = null$
⟨*proof*⟩

lemma $OclAsType_{Person}\text{-}OclAny\text{-}strict[simp]$: $(invalid::OclAny)\ .oclAsType(Person) = invalid$
⟨*proof*⟩

lemma $OclAsType_{Person}$-$OclAny$-$nullstrict[simp]$: $(null::OclAny)$ $.oclAsType(Person) = null$
⟨proof⟩

lemma $OclAsType_{Person}$-$Person$-$strict$: $(invalid::Person)$ $.oclAsType(Person) = invalid$
⟨proof⟩
lemma $OclAsType_{Person}$-$Person$-$nullstrict$: $(null::Person)$ $.oclAsType(Person) = null$
⟨proof⟩

8.5. OclIsTypeOf

8.5.1. Definition

consts $OclIsTypeOf_{OclAny} :: {'\alpha} \Rightarrow Boolean$ $((\text{-}).oclIsTypeOf'(OclAny'))$
consts $OclIsTypeOf_{Person} :: {'\alpha} \Rightarrow Boolean$ $((\text{-}).oclIsTypeOf'(Person'))$

defs (**overloaded**) $OclIsTypeOf_{OclAny}$-$OclAny$:
$(X::OclAny)$ $.oclIsTypeOf(OclAny) \equiv$
$(\lambda\tau.$ $case$ X τ of
$\bot \Rightarrow invalid$ τ
$|$ $\lfloor\bot\rfloor \Rightarrow true$ τ $(*$ $invalid$ $??$ $*)$
$|$ $\lfloor\lfloor mk_{OclAny}$ oid $\bot$ $\rfloor\rfloor \Rightarrow true$ τ
$|$ $\lfloor\lfloor mk_{OclAny}$ oid $\lfloor\text{-}\rfloor$ $\rfloor\rfloor \Rightarrow false$ $\tau)$

defs (**overloaded**) $OclIsTypeOf_{OclAny}$-$Person$:
$(X::Person)$ $.oclIsTypeOf(OclAny) \equiv$
$(\lambda\tau.$ $case$ X τ of
$\bot \Rightarrow invalid$ τ
$|$ $\lfloor\bot\rfloor \Rightarrow true$ τ $(*$ $invalid$ $??$ $*)$
$|$ $\lfloor\lfloor$ - $\rfloor\rfloor \Rightarrow false$ $\tau)$

defs (**overloaded**) $OclIsTypeOf_{Person}$-$OclAny$:
$(X::OclAny)$ $.oclIsTypeOf(Person) \equiv$
$(\lambda\tau.$ $case$ X τ of
$\bot \Rightarrow invalid$ τ
$|$ $\lfloor\bot\rfloor \Rightarrow true$ τ
$|$ $\lfloor\lfloor mk_{OclAny}$ oid $\bot$ $\rfloor\rfloor \Rightarrow false$ τ
$|$ $\lfloor\lfloor mk_{OclAny}$ oid $\lfloor\text{-}\rfloor$ $\rfloor\rfloor \Rightarrow true$ $\tau)$

defs (**overloaded**) $OclIsTypeOf_{Person}$-$Person$:
$(X::Person)$ $.oclIsTypeOf(Person) \equiv$
$(\lambda\tau.$ $case$ X τ of
$\bot \Rightarrow invalid$ τ
$|$ - $\Rightarrow true$ $\tau)$

8.5.2. Context Passing

lemma cp-$OclIsTypeOf_{OclAny}$-$Person$-$Person$: cp $P \Longrightarrow cp(\lambda X.(P(X::Person)::Person).oclIsTypeOf(OclAny))$

$\langle proof \rangle$
lemma *cp-OclIsTypeOf*$_{OclAny}$*-OclAny-OclAny*: $cp \quad P \Longrightarrow cp(\lambda X.(P(X::OclAny)::OclAny).oclIsTypeOf(OclAny))$
$\langle proof \rangle$
lemma *cp-OclIsTypeOf*$_{Person}$*-Person-Person*: $cp \quad P \Longrightarrow cp(\lambda X.(P(X::Person)::Person).oclIsTypeOf(Person))$
$\langle proof \rangle$
lemma *cp-OclIsTypeOf*$_{Person}$*-OclAny-OclAny*: $cp \quad P \Longrightarrow cp(\lambda X.(P(X::OclAny)::OclAny).oclIsTypeOf(Person))$
$\langle proof \rangle$

lemma *cp-OclIsTypeOf*$_{OclAny}$*-Person-OclAny*: $cp \quad P \Longrightarrow cp(\lambda X.(P(X::Person)::OclAny).oclIsTypeOf(OclAny))$
$\langle proof \rangle$
lemma *cp-OclIsTypeOf*$_{OclAny}$*-OclAny-Person*: $cp \quad P \Longrightarrow cp(\lambda X.(P(X::OclAny)::Person).oclIsTypeOf(OclAny))$
$\langle proof \rangle$
lemma *cp-OclIsTypeOf*$_{Person}$*-Person-OclAny*: $cp \quad P \Longrightarrow cp(\lambda X.(P(X::Person)::OclAny).oclIsTypeOf(Person))$
$\langle proof \rangle$
lemma *cp-OclIsTypeOf*$_{Person}$*-OclAny-Person*: $cp \quad P \Longrightarrow cp(\lambda X.(P(X::OclAny)::Person).oclIsTypeOf(Person))$
$\langle proof \rangle$

lemmas [*simp*] =
cp-OclIsTypeOf$_{OclAny}$*-Person-Person*
cp-OclIsTypeOf$_{OclAny}$*-OclAny-OclAny*
cp-OclIsTypeOf$_{Person}$*-Person-Person*
cp-OclIsTypeOf$_{Person}$*-OclAny-OclAny*

cp-OclIsTypeOf$_{OclAny}$*-Person-OclAny*
cp-OclIsTypeOf$_{OclAny}$*-OclAny-Person*
cp-OclIsTypeOf$_{Person}$*-Person-OclAny*
cp-OclIsTypeOf$_{Person}$*-OclAny-Person*

8.5.3. Execution with Invalid or Null as Argument

lemma *OclIsTypeOf*$_{OclAny}$*-OclAny-strict1*[*simp*]:
$(invalid::OclAny)$ $.oclIsTypeOf(OclAny) = invalid$
$\langle proof \rangle$
lemma *OclIsTypeOf*$_{OclAny}$*-OclAny-strict2*[*simp*]:
$(null::OclAny)$ $.oclIsTypeOf(OclAny) = true$
$\langle proof \rangle$
lemma *OclIsTypeOf*$_{OclAny}$*-Person-strict1*[*simp*]:
$(invalid::Person)$ $.oclIsTypeOf(OclAny) = invalid$
$\langle proof \rangle$
lemma *OclIsTypeOf*$_{OclAny}$*-Person-strict2*[*simp*]:
$(null::Person)$ $.oclIsTypeOf(OclAny) = true$

$\langle proof \rangle$

lemma $OclIsTypeOf_{Person}$-$OclAny$-$strict1$[$simp$]:
$(invalid::OclAny)$ $.oclIsTypeOf(Person) = invalid$

$\langle proof \rangle$

lemma $OclIsTypeOf_{Person}$-$OclAny$-$strict2$[$simp$]:
$(null::OclAny)$ $.oclIsTypeOf(Person) = true$

$\langle proof \rangle$

lemma $OclIsTypeOf_{Person}$-$Person$-$strict1$[$simp$]:
$(invalid::Person)$ $.oclIsTypeOf(Person) = invalid$

$\langle proof \rangle$

lemma $OclIsTypeOf_{Person}$-$Person$-$strict2$[$simp$]:
$(null::Person)$ $.oclIsTypeOf(Person) = true$

$\langle proof \rangle$

8.5.4. Up Down Casting

lemma *actualType-larger-staticType*:
assumes *isdef*: $\tau \models (\delta\ X)$
shows $\tau \models (X::Person)\ .oclIsTypeOf(OclAny) \triangleq false$
$\langle proof \rangle$

lemma *down-cast-type*:
assumes *isOclAny*: $\tau \models (X::OclAny)\ .oclIsTypeOf(OclAny)$
and *non-null*: $\tau \models (\delta\ X)$
shows $\tau \models (X\ .oclAsType(Person)) \triangleq invalid$
$\langle proof \rangle$

lemma *down-cast-type'*:
assumes *isOclAny*: $\tau \models (X::OclAny)\ .oclIsTypeOf(OclAny)$
and *non-null*: $\tau \models (\delta\ X)$
shows $\tau \models not\ (\upsilon\ (X\ .oclAsType(Person)))$
$\langle proof \rangle$

lemma *up-down-cast* :
assumes *isdef*: $\tau \models (\delta\ X)$
shows $\tau \models ((X::Person)\ .oclAsType(OclAny)\ .oclAsType(Person) \triangleq X)$
$\langle proof \rangle$

lemma *up-down-cast-Person-OclAny-Person* [*simp*]:
shows $((X::Person)\ .oclAsType(OclAny)\ .oclAsType(Person) = X)$
$\langle proof \rangle$

lemma *up-down-cast-Person-OclAny-Person'*: **assumes** $\tau \models \upsilon\ X$
shows $\tau \models (((X :: Person)\ .oclAsType(OclAny)\ .oclAsType(Person)) \doteq X)$
$\langle proof \rangle$

lemma *up-down-cast-Person-OclAny-Person''*: **assumes** $\tau \models \upsilon\ (X :: Person)$
shows $\tau \models (X\ .oclIsTypeOf(Person)\ implies\ (X\ .oclAsType(OclAny)\ .oclAsType(Person)) \doteq$

X)
⟨*proof*⟩

8.6. OclIsKindOf

8.6.1. Definition

consts $OclIsKindOf_{OclAny}$ :: $'\alpha \Rightarrow Boolean$ ((-).$oclIsKindOf'(OclAny')$)
consts $OclIsKindOf_{Person}$ :: $'\alpha \Rightarrow Boolean$ ((-).$oclIsKindOf'(Person')$)

defs (**overloaded**) $OclIsKindOf_{OclAny}$-$OclAny$:
$(X::OclAny)$.$oclIsKindOf(OclAny) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\bot \Rightarrow invalid\ \tau$
$|\ \text{-} \Rightarrow true\ \tau)$

defs (**overloaded**) $OclIsKindOf_{OclAny}$-$Person$:
$(X::Person)$.$oclIsKindOf(OclAny) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\bot \Rightarrow invalid\ \tau$
$|\ \text{-}\Rightarrow true\ \tau)$

defs (**overloaded**) $OclIsKindOf_{Person}$-$OclAny$:
$(X::OclAny)$.$oclIsKindOf(Person) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\bot \Rightarrow invalid\ \tau$
$|\ \lfloor\bot\rfloor \Rightarrow true\ \tau$
$|\ \lfloor\lfloor mk_{OclAny}\ oid\ \bot\ \rfloor\rfloor \Rightarrow false\ \tau$
$|\ \lfloor\lfloor mk_{OclAny}\ oid\ \lfloor\text{-}\rfloor\ \rfloor\rfloor \Rightarrow true\ \tau)$

defs (**overloaded**) $OclIsKindOf_{Person}$-$Person$:
$(X::Person)$.$oclIsKindOf(Person) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\bot \Rightarrow invalid\ \tau$
$|\ \text{-} \Rightarrow true\ \tau)$

8.6.2. Context Passing

lemma cp-$OclIsKindOf_{OclAny}$-$Person$-$Person$: $cp\ P \Longrightarrow cp(\lambda X.(P(X::Person)::Person).oclIsKindOf(OclAny))$
⟨*proof*⟩
lemma cp-$OclIsKindOf_{OclAny}$-$OclAny$-$OclAny$: $cp\ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::OclAny).oclIsKindOf(OclAny))$
⟨*proof*⟩
lemma cp-$OclIsKindOf_{Person}$-$Person$-$Person$: $cp\ P \Longrightarrow cp(\lambda X.(P(X::Person)::Person).oclIsKindOf(Person))$
⟨*proof*⟩

lemma *cp-OclIsKindOf*$_{Person}$*-OclAny-OclAny*: *cp* $P \Longrightarrow$ $cp(\lambda X.(P(X::OclAny)::OclAny).oclIsKindOf(Person))$
⟨*proof*⟩

lemma *cp-OclIsKindOf*$_{OclAny}$*-Person-OclAny*: *cp* $P \Longrightarrow$ $cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(OclAny))$
⟨*proof*⟩
lemma *cp-OclIsKindOf*$_{OclAny}$*-OclAny-Person*: *cp* $P \Longrightarrow$ $cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(OclAny))$
⟨*proof*⟩
lemma *cp-OclIsKindOf*$_{Person}$*-Person-OclAny*: *cp* $P \Longrightarrow$ $cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(Person))$
⟨*proof*⟩
lemma *cp-OclIsKindOf*$_{Person}$*-OclAny-Person*: *cp* $P \Longrightarrow$ $cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(Person))$
⟨*proof*⟩

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*

8.6.3. Execution with Invalid or Null as Argument

lemma *OclIsKindOf*$_{OclAny}$*-OclAny-strict1*[*simp*] : $(invalid::OclAny)$ $.oclIsKindOf(OclAny) = invalid$
⟨*proof*⟩

lemma *OclIsKindOf*$_{OclAny}$*-OclAny-strict2*[*simp*] : $(null::OclAny)$ $.oclIsKindOf(OclAny) = true$
⟨*proof*⟩

lemma *OclIsKindOf*$_{OclAny}$*-Person-strict1*[*simp*] : $(invalid::Person)$ $.oclIsKindOf(OclAny) = invalid$
⟨*proof*⟩

lemma *OclIsKindOf*$_{OclAny}$*-Person-strict2*[*simp*] : $(null::Person)$ $.oclIsKindOf(OclAny) = true$
⟨*proof*⟩

lemma *OclIsKindOf*$_{Person}$*-OclAny-strict1*[*simp*]: $(invalid::OclAny)$ $.oclIsKindOf(Person) = invalid$
⟨*proof*⟩

lemma $OclIsKindOf_{Person}$*-OclAny-strict2*[*simp*]: (*null*::*OclAny*) .*oclIsKindOf*(*Person*) = *true*
⟨*proof*⟩

lemma $OclIsKindOf_{Person}$*-Person-strict1*[*simp*]: (*invalid*::*Person*) .*oclIsKindOf*(*Person*) = *invalid*
⟨*proof*⟩

lemma $OclIsKindOf_{Person}$*-Person-strict2*[*simp*]: (*null*::*Person*) .*oclIsKindOf*(*Person*) = *true*
⟨*proof*⟩

8.6.4. Up Down Casting

lemma *actualKind-larger-staticKind*:
assumes *isdef*: $\tau \models (\delta\ X)$
shows $\tau \models ((X::Person)\ .oclIsKindOf(OclAny) \triangleq true)$
⟨*proof*⟩

lemma *down-cast-kind*:
assumes *isOclAny*: $\neg\ (\tau \models ((X::OclAny).oclIsKindOf(Person)))$
and *non-null*: $\tau \models (\delta\ X)$
shows $\tau \models ((X\ .oclAsType(Person)) \triangleq invalid)$
⟨*proof*⟩

8.7. OclAllInstances

To denote OCL-types occuring in OCL expressions syntactically—as, for example, as "argument" of oclAllInstances()—we use the inverses of the injection functions into the object universes; we show that this is sufficient "characterization."

definition $Person \equiv OclAsType_{Person}\text{-}\mathfrak{A}$
definition $OclAny \equiv OclAsType_{OclAny}\text{-}\mathfrak{A}$
lemmas [*simp*] = *Person-def OclAny-def*

lemma $OclAllInstances\text{-}generic_{OclAny}$*-exec*: *OclAllInstances-generic pre-post OclAny* =
$(\lambda\tau.\ \ Abs\text{-}Set_{base}\ \ \lfloor\lfloor\ Some\ `\ OclAny\ `\ ran\ (heap\ (pre\text{-}post\ \tau))\ \rfloor\rfloor)$
⟨*proof*⟩

lemma $OclAllInstances\text{-}at\text{-}post_{OclAny}$*-exec*: *OclAny .allInstances*() =
$(\lambda\tau.\ \ Abs\text{-}Set_{base}\ \ \lfloor\lfloor\ Some\ `\ OclAny\ `\ ran\ (heap\ (snd\ \tau))\ \rfloor\rfloor)$
⟨*proof*⟩

lemma $OclAllInstances\text{-}at\text{-}pre_{OclAny}$*-exec*: *OclAny .allInstances@pre*() =
$(\lambda\tau.\ \ Abs\text{-}Set_{base}\ \ \lfloor\lfloor\ Some\ `\ OclAny\ `\ ran\ (heap\ (fst\ \tau))\ \rfloor\rfloor)$
⟨*proof*⟩

8.7.1. OclIsTypeOf

lemma $OclAny\text{-}allInstances\text{-}generic\text{-}oclIsTypeOf_{OclAny}1$:
assumes [*simp*]: $\bigwedge x.\ pre\text{-}post\ (x,\ x) = x$

shows $\exists \tau.\ (\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ OclAny) {-}{>} forAll(X|X\ .oclIsTypeOf(OclAny))))$
$\langle proof \rangle$

lemma *OclAny-allInstances-at-post-oclIsTypeOf*$_{OclAny}$*1*:
$\exists \tau.\ (\tau \models (OclAny\ .allInstances() {-}{>} forAll(X|X\ .oclIsTypeOf(OclAny))))$
$\langle proof \rangle$

lemma *OclAny-allInstances-at-pre-oclIsTypeOf*$_{OclAny}$*1*:
$\exists \tau.\ (\tau \models (OclAny\ .allInstances@pre() {-}{>} forAll(X|X\ .oclIsTypeOf(OclAny))))$
$\langle proof \rangle$

lemma *OclAny-allInstances-generic-oclIsTypeOf*$_{OclAny}$*2*:
assumes [*simp*]: $\bigwedge x.\ pre\text{-}post\ (x, x) = x$
shows $\exists \tau.\ (\tau \models not\ ((OclAllInstances\text{-}generic\ pre\text{-}post\ OclAny) {-}{>} forAll(X|X\ .oclIsTypeOf(OclAny))))$
$\langle proof \rangle$

lemma *OclAny-allInstances-at-post-oclIsTypeOf*$_{OclAny}$*2*:
$\exists \tau.\ (\tau \models not\ (OclAny\ .allInstances() {-}{>} forAll(X|X\ .oclIsTypeOf(OclAny))))$
$\langle proof \rangle$

lemma *OclAny-allInstances-at-pre-oclIsTypeOf*$_{OclAny}$*2*:
$\exists \tau.\ (\tau \models not\ (OclAny\ .allInstances@pre() {-}{>} forAll(X|X\ .oclIsTypeOf(OclAny))))$
$\langle proof \rangle$

lemma *Person-allInstances-generic-oclIsTypeOf*$_{Person}$:
$\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ Person) {-}{>} forAll(X|X\ .oclIsTypeOf(Person)))$
$\langle proof \rangle$

lemma *Person-allInstances-at-post-oclIsTypeOf*$_{Person}$:
$\tau \models (Person\ .allInstances() {-}{>} forAll(X|X\ .oclIsTypeOf(Person)))$
$\langle proof \rangle$

lemma *Person-allInstances-at-pre-oclIsTypeOf*$_{Person}$:
$\tau \models (Person\ .allInstances@pre() {-}{>} forAll(X|X\ .oclIsTypeOf(Person)))$
$\langle proof \rangle$

8.7.2. OclIsKindOf

lemma *OclAny-allInstances-generic-oclIsKindOf*$_{OclAny}$:
$\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ OclAny) {-}{>} forAll(X|X\ .oclIsKindOf(OclAny)))$
$\langle proof \rangle$

lemma *OclAny-allInstances-at-post-oclIsKindOf*$_{OclAny}$:
$\tau \models (OclAny\ .allInstances() {-}{>} forAll(X|X\ .oclIsKindOf(OclAny)))$
$\langle proof \rangle$

lemma *OclAny-allInstances-at-pre-oclIsKindOf*$_{OclAny}$:

$\tau \models (OclAny\ .allInstances@pre()\text{->}forAll(X|X\ .oclIsKindOf(OclAny)))$
⟨*proof*⟩

lemma $Person\text{-}allInstances\text{-}generic\text{-}oclIsKindOf_{OclAny}$:
$\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ Person)\text{->}forAll(X|X\ .oclIsKindOf(OclAny)))$
⟨*proof*⟩

lemma $Person\text{-}allInstances\text{-}at\text{-}post\text{-}oclIsKindOf_{OclAny}$:
$\tau \models (Person\ .allInstances()\text{->}forAll(X|X\ .oclIsKindOf(OclAny)))$
⟨*proof*⟩

lemma $Person\text{-}allInstances\text{-}at\text{-}pre\text{-}oclIsKindOf_{OclAny}$:
$\tau \models (Person\ .allInstances@pre()\text{->}forAll(X|X\ .oclIsKindOf(OclAny)))$
⟨*proof*⟩

lemma $Person\text{-}allInstances\text{-}generic\text{-}oclIsKindOf_{Person}$:
$\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ Person)\text{->}forAll(X|X\ .oclIsKindOf(Person)))$
⟨*proof*⟩

lemma $Person\text{-}allInstances\text{-}at\text{-}post\text{-}oclIsKindOf_{Person}$:
$\tau \models (Person\ .allInstances()\text{->}forAll(X|X\ .oclIsKindOf(Person)))$
⟨*proof*⟩

lemma $Person\text{-}allInstances\text{-}at\text{-}pre\text{-}oclIsKindOf_{Person}$:
$\tau \models (Person\ .allInstances@pre()\text{->}forAll(X|X\ .oclIsKindOf(Person)))$
⟨*proof*⟩

8.8. The Accessors (any, boss, salary)

Should be generated entirely from a class-diagram.

8.8.1. Definition

definition $eval\text{-}extract :: ('\mathfrak{A},('a::object)\ option\ option)\ val$
$\quad \Rightarrow (oid \Rightarrow ('\mathfrak{A},'c::null)\ val)$
$\quad \Rightarrow ('\mathfrak{A},'c::null)\ val$
where $eval\text{-}extract\ X\ f = (\lambda\ \tau.\ case\ X\ \tau\ of$
$\quad \bot \Rightarrow invalid\ \tau \quad (*\ exception\ propagation\ *)$
$\quad |\ \lfloor\ \bot\ \rfloor \Rightarrow invalid\ \tau\ (*\ dereferencing\ null\ pointer\ *)$
$\quad |\ \lfloor\lfloor\ obj\ \rfloor\rfloor \Rightarrow f\ (oid\text{-}of\ obj)\ \tau)$

definition $deref\text{-}oid_{Person} :: (\mathfrak{A}\ state \times \mathfrak{A}\ state \Rightarrow \mathfrak{A}\ state)$
$\quad \Rightarrow (type_{Person} \Rightarrow (\mathfrak{A},\ 'c::null)val)$
$\quad \Rightarrow oid$
$\quad \Rightarrow (\mathfrak{A},\ 'c::null)val$
where $deref\text{-}oid_{Person}\ fst\text{-}snd\ f\ oid = (\lambda\tau.\ case\ (heap\ (fst\text{-}snd\ \tau))\ oid\ of$

$\lfloor$ in_{Person} *obj* $\rfloor$ $\Rightarrow$ *f obj* τ
| - $\Rightarrow$ *invalid* τ)

definition $\textit{deref-oid}_{OclAny}$ :: ($\mathfrak{A}$ *state* $\times$ $\mathfrak{A}$ *state* $\Rightarrow$ $\mathfrak{A}$ *state*)
$\Rightarrow$ ($\textit{type}_{OclAny}$ $\Rightarrow$ ($\mathfrak{A}$, $'c$::*null*)*val*)
$\Rightarrow$ *oid*
$\Rightarrow$ ($\mathfrak{A}$, $'c$::*null*)*val*
where $\textit{deref-oid}_{OclAny}$ *fst-snd f oid* = ($\lambda\tau$. *case* (*heap* (*fst-snd* τ)) *oid of*
$\lfloor$ in_{OclAny} *obj* $\rfloor$ $\Rightarrow$ *f obj* τ
| - $\Rightarrow$ *invalid* τ)

pointer undefined in state or not referencing a type conform object representation

definition $\textit{select}_{OclAny}\mathcal{ANY}$ *f* = (λ *X*. *case X of*
(mk_{OclAny} - $\bot$) $\Rightarrow$ *null*
| (mk_{OclAny} - $\lfloor$*any*$\rfloor$) $\Rightarrow$ *f* (λx -. $\lfloor\lfloor x\rfloor\rfloor$) *any*)

definition $\textit{select}_{Person}\mathcal{BOSS}$ *f* = (λ *X*. *case X of*
(mk_{Person} - - $\bot$) $\Rightarrow$ *null* (* *object contains null pointer* *)
| (mk_{Person} - - $\lfloor$*boss*$\rfloor$) $\Rightarrow$ *f* (λx -. $\lfloor\lfloor x\rfloor\rfloor$) *boss*)

definition $\textit{select}_{Person}\mathcal{SALARY}$ *f* = (λ *X*. *case X of*
(mk_{Person} - $\bot$ -) $\Rightarrow$ *null*
| (mk_{Person} - $\lfloor$*salary*$\rfloor$ -) $\Rightarrow$ *f* (λx -. $\lfloor\lfloor x\rfloor\rfloor$) *salary*)

definition *in-pre-state* = *fst*
definition *in-post-state* = *snd*

definition *reconst-basetype* = (λ *convert x*. *convert x*)

definition $\textit{dot}_{OclAny}\mathcal{ANY}$ :: *OclAny* $\Rightarrow$ - ((*1*(-).*any*) *50*)
where (*X*).*any* = *eval-extract X*
($\textit{deref-oid}_{OclAny}$ *in-post-state*
($\textit{select}_{OclAny}\mathcal{ANY}$
reconst-basetype))

definition $\textit{dot}_{Person}\mathcal{BOSS}$ :: *Person* $\Rightarrow$ *Person* ((*1*(-).*boss*) *50*)
where (*X*).*boss* = *eval-extract X*
($\textit{deref-oid}_{Person}$ *in-post-state*
($\textit{select}_{Person}\mathcal{BOSS}$
($\textit{deref-oid}_{Person}$ *in-post-state*)))

definition $\textit{dot}_{Person}\mathcal{SALARY}$ :: *Person* $\Rightarrow$ *Integer* ((*1*(-).*salary*) *50*)
where (*X*).*salary* = *eval-extract X*
($\textit{deref-oid}_{Person}$ *in-post-state*

$(select_{Person}\mathcal{SALARY}$
$reconst\text{-}basetype))$

definition $dot_{OclAny}\mathcal{ANY}\text{-}at\text{-}pre :: OclAny \Rightarrow \text{-} \;\; ((1(\text{-}).any@pre)\; 50)$
where $(X).any@pre = eval\text{-}extract\; X$
$(deref\text{-}oid_{OclAny}\; in\text{-}pre\text{-}state$
$(select_{OclAny}\mathcal{ANY}$
$reconst\text{-}basetype))$

definition $dot_{Person}\mathcal{BOSS}\text{-}at\text{-}pre:: Person \Rightarrow Person \;\; ((1(\text{-}).boss@pre)\; 50)$
where $(X).boss@pre = eval\text{-}extract\; X$
$(deref\text{-}oid_{Person}\; in\text{-}pre\text{-}state$
$(select_{Person}\mathcal{BOSS}$
$(deref\text{-}oid_{Person}\; in\text{-}pre\text{-}state)))$

definition $dot_{Person}\mathcal{SALARY}\text{-}at\text{-}pre:: Person \Rightarrow Integer \;\; ((1(\text{-}).salary@pre)\; 50)$
where $(X).salary@pre = eval\text{-}extract\; X$
$(deref\text{-}oid_{Person}\; in\text{-}pre\text{-}state$
$(select_{Person}\mathcal{SALARY}$
$reconst\text{-}basetype))$

lemmas $[simp] =$
$dot_{OclAny}\mathcal{ANY}\text{-}def$
$dot_{Person}\mathcal{BOSS}\text{-}def$
$dot_{Person}\mathcal{SALARY}\text{-}def$
$dot_{OclAny}\mathcal{ANY}\text{-}at\text{-}pre\text{-}def$
$dot_{Person}\mathcal{BOSS}\text{-}at\text{-}pre\text{-}def$
$dot_{Person}\mathcal{SALARY}\text{-}at\text{-}pre\text{-}def$

8.8.2. Context Passing

lemmas $[simp] = eval\text{-}extract\text{-}def$

lemma $cp\text{-}dot_{OclAny}\mathcal{ANY}$: $((X).any)\; \tau = ((\lambda\text{-}.\; X\; \tau).any)\; \tau$ ⟨*proof*⟩
lemma $cp\text{-}dot_{Person}\mathcal{BOSS}$: $((X).boss)\; \tau = ((\lambda\text{-}.\; X\; \tau).boss)\; \tau$ ⟨*proof*⟩
lemma $cp\text{-}dot_{Person}\mathcal{SALARY}$: $((X).salary)\; \tau = ((\lambda\text{-}.\; X\; \tau).salary)\; \tau$ ⟨*proof*⟩

lemma $cp\text{-}dot_{OclAny}\mathcal{ANY}\text{-}at\text{-}pre$: $((X).any@pre)\; \tau = ((\lambda\text{-}.\; X\; \tau).any@pre)\; \tau$ ⟨*proof*⟩
lemma $cp\text{-}dot_{Person}\mathcal{BOSS}\text{-}at\text{-}pre$: $((X).boss@pre)\; \tau = ((\lambda\text{-}.\; X\; \tau).boss@pre)\; \tau$ ⟨*proof*⟩
lemma $cp\text{-}dot_{Person}\mathcal{SALARY}\text{-}at\text{-}pre$: $((X).salary@pre)\; \tau = ((\lambda\text{-}.\; X\; \tau).salary@pre)\; \tau$ ⟨*proof*⟩

lemmas $cp\text{-}dot_{OclAny}\mathcal{ANY}\text{-}I$ $[simp,\; intro!]=$
$cp\text{-}dot_{OclAny}\mathcal{ANY}[THEN\; allI[THEN\; allI],$
$of\; \lambda\; X\; \text{-}.\; X\; \lambda\; \text{-}\; \tau.\; \tau,\; THEN\; cpI1]$
lemmas $cp\text{-}dot_{OclAny}\mathcal{ANY}\text{-}at\text{-}pre\text{-}I$ $[simp,\; intro!]=$
$cp\text{-}dot_{OclAny}\mathcal{ANY}\text{-}at\text{-}pre[THEN\; allI[THEN\; allI],$
$of\; \lambda\; X\; \text{-}.\; X\; \lambda\; \text{-}\; \tau.\; \tau,\; THEN\; cpI1]$

lemmas $cp\text{-}dot_{Person}\mathcal{BOSS}\text{-}I$ $[simp,\; intro!]=$

$cp\text{-}dot_{Person}\mathcal{BOSS}$[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]
lemmas $cp\text{-}dot_{Person}\mathcal{BOSS}$*-at-pre-I* [*simp*, *intro*!]=
$cp\text{-}dot_{Person}\mathcal{BOSS}$*-at-pre*[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

lemmas $cp\text{-}dot_{Person}\mathcal{SALARY}$*-I* [*simp*, *intro*!]=
$cp\text{-}dot_{Person}\mathcal{SALARY}$[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]
lemmas $cp\text{-}dot_{Person}\mathcal{SALARY}$*-at-pre-I* [*simp*, *intro*!]=
$cp\text{-}dot_{Person}\mathcal{SALARY}$*-at-pre*[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

8.8.3. Execution with Invalid or Null as Argument

lemma $dot_{OclAny}\mathcal{ANY}$*-nullstrict* [*simp*]: (*null*).*any* = *invalid*
⟨*proof*⟩
lemma $dot_{OclAny}\mathcal{ANY}$*-at-pre-nullstrict* [*simp*] : (*null*).*any*@*pre* = *invalid*
⟨*proof*⟩
lemma $dot_{OclAny}\mathcal{ANY}$*-strict* [*simp*] : (*invalid*).*any* = *invalid*
⟨*proof*⟩
lemma $dot_{OclAny}\mathcal{ANY}$*-at-pre-strict* [*simp*] : (*invalid*).*any*@*pre* = *invalid*
⟨*proof*⟩

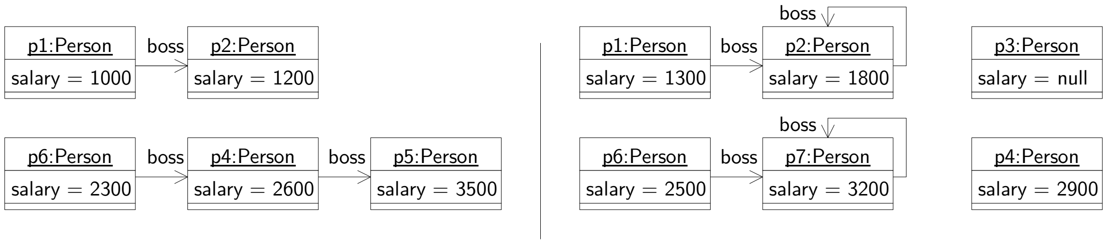
lemma $dot_{Person}\mathcal{BOSS}$*-nullstrict* [*simp*]: (*null*).*boss* = *invalid*
⟨*proof*⟩
lemma $dot_{Person}\mathcal{BOSS}$*-at-pre-nullstrict* [*simp*] : (*null*).*boss*@*pre* = *invalid*
⟨*proof*⟩
lemma $dot_{Person}\mathcal{BOSS}$*-strict* [*simp*] : (*invalid*).*boss* = *invalid*
⟨*proof*⟩
lemma $dot_{Person}\mathcal{BOSS}$*-at-pre-strict* [*simp*] : (*invalid*).*boss*@*pre* = *invalid*
⟨*proof*⟩

lemma $dot_{Person}\mathcal{SALARY}$*-nullstrict* [*simp*]: (*null*).*salary* = *invalid*
⟨*proof*⟩
lemma $dot_{Person}\mathcal{SALARY}$*-at-pre-nullstrict* [*simp*] : (*null*).*salary*@*pre* = *invalid*
⟨*proof*⟩
lemma $dot_{Person}\mathcal{SALARY}$*-strict* [*simp*] : (*invalid*).*salary* = *invalid*
⟨*proof*⟩
lemma $dot_{Person}\mathcal{SALARY}$*-at-pre-strict* [*simp*] : (*invalid*).*salary*@*pre* = *invalid*
⟨*proof*⟩

8.9. A Little Infra-structure on Example States

The example we are defining in this section comes from the figure 8.2.

definition *OclInt1000* (**1000**) **where** *OclInt1000* = (λ - . $\lfloor\lfloor$*1000*$\rfloor\rfloor$)



(a) (b)

Figure 8.2.: (a) pre-state σ_1 and (b) post-state σ_1'.

definition $\mathit{OclInt1200}$ (**1200**) **where** $\mathit{OclInt1200} = (\lambda\ \text{-}\ .\ \lfloor\lfloor \mathit{1200} \rfloor\rfloor)$
definition $\mathit{OclInt1300}$ (**1300**) **where** $\mathit{OclInt1300} = (\lambda\ \text{-}\ .\ \lfloor\lfloor \mathit{1300} \rfloor\rfloor)$
definition $\mathit{OclInt1800}$ (**1800**) **where** $\mathit{OclInt1800} = (\lambda\ \text{-}\ .\ \lfloor\lfloor \mathit{1800} \rfloor\rfloor)$
definition $\mathit{OclInt2600}$ (**2600**) **where** $\mathit{OclInt2600} = (\lambda\ \text{-}\ .\ \lfloor\lfloor \mathit{2600} \rfloor\rfloor)$
definition $\mathit{OclInt2900}$ (**2900**) **where** $\mathit{OclInt2900} = (\lambda\ \text{-}\ .\ \lfloor\lfloor \mathit{2900} \rfloor\rfloor)$
definition $\mathit{OclInt3200}$ (**3200**) **where** $\mathit{OclInt3200} = (\lambda\ \text{-}\ .\ \lfloor\lfloor \mathit{3200} \rfloor\rfloor)$
definition $\mathit{OclInt3500}$ (**3500**) **where** $\mathit{OclInt3500} = (\lambda\ \text{-}\ .\ \lfloor\lfloor \mathit{3500} \rfloor\rfloor)$

definition $\mathit{oid0} \equiv \mathit{0}$
definition $\mathit{oid1} \equiv \mathit{1}$
definition $\mathit{oid2} \equiv \mathit{2}$
definition $\mathit{oid3} \equiv \mathit{3}$
definition $\mathit{oid4} \equiv \mathit{4}$
definition $\mathit{oid5} \equiv \mathit{5}$
definition $\mathit{oid6} \equiv \mathit{6}$
definition $\mathit{oid7} \equiv \mathit{7}$
definition $\mathit{oid8} \equiv \mathit{8}$

definition $\mathit{person1} \equiv \mathit{mk}_{\mathit{Person}}\ \mathit{oid0}\ \lfloor \mathit{1300} \rfloor\ \lfloor \mathit{oid1} \rfloor$
definition $\mathit{person2} \equiv \mathit{mk}_{\mathit{Person}}\ \mathit{oid1}\ \lfloor \mathit{1800} \rfloor\ \lfloor \mathit{oid1} \rfloor$
definition $\mathit{person3} \equiv \mathit{mk}_{\mathit{Person}}\ \mathit{oid2}\ \mathit{None}\ \mathit{None}$
definition $\mathit{person4} \equiv \mathit{mk}_{\mathit{Person}}\ \mathit{oid3}\ \lfloor \mathit{2900} \rfloor\ \mathit{None}$
definition $\mathit{person5} \equiv \mathit{mk}_{\mathit{Person}}\ \mathit{oid4}\ \lfloor \mathit{3500} \rfloor\ \mathit{None}$
definition $\mathit{person6} \equiv \mathit{mk}_{\mathit{Person}}\ \mathit{oid5}\ \lfloor \mathit{2500} \rfloor\ \lfloor \mathit{oid6} \rfloor$
definition $\mathit{person7} \equiv \mathit{mk}_{\mathit{OclAny}}\ \mathit{oid6}\ \lfloor (\lfloor \mathit{3200} \rfloor, \lfloor \mathit{oid6} \rfloor) \rfloor$
definition $\mathit{person8} \equiv \mathit{mk}_{\mathit{OclAny}}\ \mathit{oid7}\ \mathit{None}$
definition $\mathit{person9} \equiv \mathit{mk}_{\mathit{Person}}\ \mathit{oid8}\ \lfloor \mathit{0} \rfloor\ \mathit{None}$

definition

$$
\begin{aligned}
\sigma_1 \ \equiv \ (\!|\ \mathit{heap} = \mathit{empty}(&\mathit{oid0} \mapsto \mathit{in}_{\mathit{Person}}\ (\mathit{mk}_{\mathit{Person}}\ \mathit{oid0}\ \lfloor \mathit{1000} \rfloor\ \lfloor \mathit{oid1} \rfloor))\\
&(\mathit{oid1} \mapsto \mathit{in}_{\mathit{Person}}\ (\mathit{mk}_{\mathit{Person}}\ \mathit{oid1}\ \lfloor \mathit{1200} \rfloor\ \ \mathit{None}))\\
&(*\mathit{oid2}*)\\
&(\mathit{oid3} \mapsto \mathit{in}_{\mathit{Person}}\ (\mathit{mk}_{\mathit{Person}}\ \mathit{oid3}\ \lfloor \mathit{2600} \rfloor\ \lfloor \mathit{oid4} \rfloor))\\
&(\mathit{oid4} \mapsto \mathit{in}_{\mathit{Person}}\ \mathit{person5})\\
&(\mathit{oid5} \mapsto \mathit{in}_{\mathit{Person}}\ (\mathit{mk}_{\mathit{Person}}\ \mathit{oid5}\ \lfloor \mathit{2300} \rfloor\ \lfloor \mathit{oid3} \rfloor))\\
&(*\mathit{oid6}*)
\end{aligned}
$$

$(*oid7*)$
$(oid8 \mapsto in_{Person}\ person9),$
$assocs = empty\ \|)$

definition
$\sigma_1{}' \equiv (\|\ heap = empty(oid0 \mapsto in_{Person}\ person1)$
$(oid1 \mapsto in_{Person}\ person2)$
$(oid2 \mapsto in_{Person}\ person3)$
$(oid3 \mapsto in_{Person}\ person4)$
$(*oid4*)$
$(oid5 \mapsto in_{Person}\ person6)$
$(oid6 \mapsto in_{OclAny}\ person7)$
$(oid7 \mapsto in_{OclAny}\ person8)$
$(oid8 \mapsto in_{Person}\ person9),$
$assocs = empty\ \|)$

definition $\sigma_0 \equiv (\|\ heap = empty,\ assocs = empty\ \|)$

lemma *basic-τ-wff*: $WFF(\sigma_1,\sigma_1{}')$
⟨*proof*⟩

lemma [*simp,code-unfold*]: $dom\ (heap\ \sigma_1) = \{oid0,oid1,(*,oid2*)oid3,oid4,oid5(*,oid6,oid7*),oid8\}$
⟨*proof*⟩

lemma [*simp,code-unfold*]: $dom\ (heap\ \sigma_1{}') = \{oid0,oid1,oid2,oid3,(*,oid4*)oid5,oid6,oid7,oid8\}$
⟨*proof*⟩

definition $X_{Person}1$ :: *Person* $\equiv \lambda$ - .⌊⌊ *person1* ⌋⌋
definition $X_{Person}2$ :: *Person* $\equiv \lambda$ - .⌊⌊ *person2* ⌋⌋
definition $X_{Person}3$ :: *Person* $\equiv \lambda$ - .⌊⌊ *person3* ⌋⌋
definition $X_{Person}4$ :: *Person* $\equiv \lambda$ - .⌊⌊ *person4* ⌋⌋
definition $X_{Person}5$ :: *Person* $\equiv \lambda$ - .⌊⌊ *person5* ⌋⌋
definition $X_{Person}6$ :: *Person* $\equiv \lambda$ - .⌊⌊ *person6* ⌋⌋
definition $X_{Person}7$ :: *OclAny* $\equiv \lambda$ - .⌊⌊ *person7* ⌋⌋
definition $X_{Person}8$ :: *OclAny* $\equiv \lambda$ - .⌊⌊ *person8* ⌋⌋
definition $X_{Person}9$ :: *Person* $\equiv \lambda$ - .⌊⌊ *person9* ⌋⌋

lemma [*code-unfold*]: $((x::Person) \doteq y) = StrictRefEq_{Object}\ x\ y$ ⟨*proof*⟩
lemma [*code-unfold*]: $((x::OclAny) \doteq y) = StrictRefEq_{Object}\ x\ y$ ⟨*proof*⟩

lemmas [*simp,code-unfold*] =
$OclAsType_{OclAny}$-*OclAny*
$OclAsType_{OclAny}$-*Person*
$OclAsType_{Person}$-*OclAny*
$OclAsType_{Person}$-*Person*

$OclIsTypeOf_{OclAny}$-*OclAny*
$OclIsTypeOf_{OclAny}$-*Person*

$OclIsTypeOf_{Person}$-$OclAny$
$OclIsTypeOf_{Person}$-$Person$

$OclIsKindOf_{OclAny}$-$OclAny$
$OclIsKindOf_{OclAny}$-$Person$
$OclIsKindOf_{Person}$-$OclAny$
$OclIsKindOf_{Person}$-$Person$

Assert $\bigwedge s_{pre} \ . \ (s_{pre}, \sigma_1{}') \models (X_{Person}1 \ .salary \ <> \mathbf{1000})$
Assert $\bigwedge s_{pre} \ . \ (s_{pre}, \sigma_1{}') \models (X_{Person}1 \ .salary \ \doteq \mathbf{1300})$
Assert $\bigwedge \ s_{post}. \ (\sigma_1, s_{post}) \models (X_{Person}1 \ .salary@pre \ \doteq \mathbf{1000})$
Assert $\bigwedge \ s_{post}. \ (\sigma_1, s_{post}) \models (X_{Person}1 \ .salary@pre \ <> \mathbf{1300})$
Assert $\bigwedge s_{pre} \ . \ (s_{pre}, \sigma_1{}') \models (X_{Person}1 \ .boss \ <> X_{Person}1)$
Assert $\bigwedge s_{pre} \ . \ (s_{pre}, \sigma_1{}') \models (X_{Person}1 \ .boss \ .salary \ \doteq \mathbf{1800})$
Assert $\bigwedge s_{pre} \ . \ (s_{pre}, \sigma_1{}') \models (X_{Person}1 \ .boss \ .boss \ <> X_{Person}1)$
Assert $\bigwedge s_{pre} \ . \ (s_{pre}, \sigma_1{}') \models (X_{Person}1 \ .boss \ .boss \ \doteq X_{Person}2)$
Assert $(\sigma_1, \sigma_1{}') \models (X_{Person}1 \ .boss@pre \ .salary \ \doteq \mathbf{1800})$
Assert $\bigwedge \ s_{post}. \ (\sigma_1, s_{post}) \models (X_{Person}1 \ .boss@pre \ .salary@pre \ \doteq \mathbf{1200})$
Assert $\bigwedge \ s_{post}. \ (\sigma_1, s_{post}) \models (X_{Person}1 \ .boss@pre \ .salary@pre \ <> \mathbf{1800})$
Assert $\bigwedge \ s_{post}. \ (\sigma_1, s_{post}) \models (X_{Person}1 \ .boss@pre \ \doteq X_{Person}2)$
Assert $(\sigma_1, \sigma_1{}') \models (X_{Person}1 \ .boss@pre \ .boss \ \doteq X_{Person}2)$
Assert $\bigwedge \ s_{post}. \ (\sigma_1, s_{post}) \models (X_{Person}1 \ .boss@pre \ .boss@pre \ \doteq null)$
Assert $\bigwedge \ s_{post}. \ (\sigma_1, s_{post}) \models not(\upsilon(X_{Person}1 \ .boss@pre \ .boss@pre \ .boss@pre))$

lemma $(\sigma_1, \sigma_1{}') \models (X_{Person}1 \ .oclIsMaintained())$
$\langle proof \rangle$

lemma $\bigwedge s_{pre} \ s_{post}. \ (s_{pre}, s_{post}) \models ((X_{Person}1 \ .oclAsType(OclAny) \ .oclAsType(Person)) \doteq X_{Person}1)$
$\langle proof \rangle$
Assert $\bigwedge s_{pre} \ s_{post}. \ (s_{pre}, s_{post}) \models (X_{Person}1 \ .oclIsTypeOf(Person))$
Assert $\bigwedge s_{pre} \ s_{post}. \ (s_{pre}, s_{post}) \models not(X_{Person}1 \ .oclIsTypeOf(OclAny))$
Assert $\bigwedge s_{pre} \ s_{post}. \ (s_{pre}, s_{post}) \models (X_{Person}1 \ .oclIsKindOf(Person))$
Assert $\bigwedge s_{pre} \ s_{post}. \ (s_{pre}, s_{post}) \models (X_{Person}1 \ .oclIsKindOf(OclAny))$
Assert $\bigwedge s_{pre} \ s_{post}. \ (s_{pre}, s_{post}) \models not(X_{Person}1 \ .oclAsType(OclAny) \ .oclIsTypeOf(OclAny))$

Assert $\bigwedge s_{pre} \ . \ (s_{pre}, \sigma_1{}') \models (X_{Person}2 \ .salary \ \doteq \mathbf{1800})$
Assert $\bigwedge \ s_{post}. \ (\sigma_1, s_{post}) \models (X_{Person}2 \ .salary@pre \ \doteq \mathbf{1200})$
Assert $\bigwedge s_{pre} \ . \ (s_{pre}, \sigma_1{}') \models (X_{Person}2 \ .boss \ \doteq X_{Person}2)$
Assert $(\sigma_1, \sigma_1{}') \models (X_{Person}2 \ .boss \ .salary@pre \ \doteq \mathbf{1200})$
Assert $(\sigma_1, \sigma_1{}') \models (X_{Person}2 \ .boss \ .boss@pre \ \doteq null)$
Assert $\bigwedge \ s_{post}. \ (\sigma_1, s_{post}) \models (X_{Person}2 \ .boss@pre \ \doteq null)$
Assert $\bigwedge \ s_{post}. \ (\sigma_1, s_{post}) \models (X_{Person}2 \ .boss@pre \ <> X_{Person}2)$
Assert $(\sigma_1, \sigma_1{}') \models (X_{Person}2 \ .boss@pre \ <> (X_{Person}2 \ .boss))$
Assert $\bigwedge \ s_{post}. \ (\sigma_1, s_{post}) \models not(\upsilon(X_{Person}2 \ .boss@pre \ .boss))$
Assert $\bigwedge \ s_{post}. \ (\sigma_1, s_{post}) \models not(\upsilon(X_{Person}2 \ .boss@pre \ .salary@pre))$

lemma $(\sigma_1,\sigma_1{}') \models (X_{Person}2\ .oclIsMaintained())$
⟨*proof*⟩

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models (X_{Person}3\ .salary \doteq null)$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models not(\upsilon(X_{Person}3\ .salary@pre))$
Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models (X_{Person}3\ .boss \doteq null)$
Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models not(\upsilon(X_{Person}3\ .boss\ .salary))$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models not(\upsilon(X_{Person}3\ .boss@pre))$
lemma $(\sigma_1,\sigma_1{}') \models (X_{Person}3\ .oclIsNew())$
⟨*proof*⟩

Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}4\ .boss@pre \doteq X_{Person}5)$
Assert $(\sigma_1,\sigma_1{}') \models not(\upsilon(X_{Person}4\ .boss@pre\ .salary))$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}4\ .boss@pre\ .salary@pre \doteq \mathbf{3500})$
lemma $(\sigma_1,\sigma_1{}') \models (X_{Person}4\ .oclIsMaintained())$
⟨*proof*⟩

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models not(\upsilon(X_{Person}5\ .salary))$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}5\ .salary@pre \doteq \mathbf{3500})$
Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models not(\upsilon(X_{Person}5\ .boss))$
lemma $(\sigma_1,\sigma_1{}') \models (X_{Person}5\ .oclIsDeleted())$
⟨*proof*⟩

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models not(\upsilon(X_{Person}6\ .boss\ .salary@pre))$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}6\ .boss@pre \doteq X_{Person}4)$
Assert $(\sigma_1,\sigma_1{}') \models (X_{Person}6\ .boss@pre\ .salary \doteq \mathbf{2900})$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}6\ .boss@pre\ .salary@pre \doteq \mathbf{2600})$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models (X_{Person}6\ .boss@pre\ .boss@pre \doteq X_{Person}5)$
lemma $(\sigma_1,\sigma_1{}') \models (X_{Person}6\ .oclIsMaintained())$
⟨*proof*⟩

Assert $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models \upsilon(X_{Person}7\ .oclAsType(Person))$
Assert $\bigwedge s_{post}.\ (\sigma_1,s_{post}) \models not(\upsilon(X_{Person}7\ .oclAsType(Person)\ .boss@pre))$
lemma $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models ((X_{Person}7\ .oclAsType(Person)\ .oclAsType(OclAny)\ .oclAsType(Person)) \doteq (X_{Person}7\ .oclAsType(Person)))$
⟨*proof*⟩
lemma $(\sigma_1,\sigma_1{}') \models (X_{Person}7\ .oclIsNew())$
⟨*proof*⟩

Assert $\bigwedge s_{pre}\ s_{post}.\quad (s_{pre},s_{post}) \models \quad (X_{Person}8\ \ <>\ X_{Person}7)$
Assert $\bigwedge s_{pre}\ s_{post}.\quad (s_{pre},s_{post}) \models not(\upsilon(X_{Person}8\ .oclAsType(Person)))$
Assert $\bigwedge s_{pre}\ s_{post}.\quad (s_{pre},s_{post}) \models \quad (X_{Person}8\ .oclIsTypeOf(OclAny))$
Assert $\bigwedge s_{pre}\ s_{post}.\quad (s_{pre},s_{post}) \models \ not(X_{Person}8\ .oclIsTypeOf(Person))$
Assert $\bigwedge s_{pre}\ s_{post}.\quad (s_{pre},s_{post}) \models \ not(X_{Person}8\ .oclIsKindOf(Person))$
Assert $\bigwedge s_{pre}\ s_{post}.\quad (s_{pre},s_{post}) \models \quad (X_{Person}8\ .oclIsKindOf(OclAny))$

lemma σ-*modifiedonly*: $(\sigma_1,\sigma_1{}') \models (Set\{\ X_{Person}1\ .oclAsType(OclAny)$
$,\ X_{Person}2\ .oclAsType(OclAny)$
$(*,\ X_{Person}3\ .oclAsType(OclAny)*)$
$,\ X_{Person}4\ .oclAsType(OclAny)$
$(*,\ X_{Person}5\ .oclAsType(OclAny)*)$
$,\ X_{Person}6\ .oclAsType(OclAny)$
$(*,\ X_{Person}7\ .oclAsType(OclAny)*)$
$(*,\ X_{Person}8\ .oclAsType(OclAny)*)$
$(*,\ X_{Person}9\ .oclAsType(OclAny)*)\}{-}{>}oclIsModifiedOnly())$
⟨*proof*⟩

lemma $(\sigma_1,\sigma_1{}') \models ((X_{Person}9\ @pre\ (\lambda x.\ \lfloor OclAsType_{Person}\text{-}\mathfrak{A}\ x\rfloor))\ \triangleq X_{Person}9)$
⟨*proof*⟩

lemma $(\sigma_1,\sigma_1{}') \models ((X_{Person}9\ @post\ (\lambda x.\ \lfloor OclAsType_{Person}\text{-}\mathfrak{A}\ x\rfloor))\ \triangleq X_{Person}9)$
⟨*proof*⟩

lemma $(\sigma_1,\sigma_1{}') \models (((X_{Person}9\ .oclAsType(OclAny))\ @pre\ (\lambda x.\ \lfloor OclAsType_{OclAny}\text{-}\mathfrak{A}\ x\rfloor)) \triangleq$
$((X_{Person}9\ .oclAsType(OclAny))\ @post\ (\lambda x.\ \lfloor OclAsType_{OclAny}\text{-}\mathfrak{A}\ x\rfloor)))$
⟨*proof*⟩

lemma *perm*-$\sigma_1{}'$: $\sigma_1{}' = (\!|\ heap = empty$
$(oid8 \mapsto in_{Person}\ person9)$
$(oid7 \mapsto in_{OclAny}\ person8)$
$(oid6 \mapsto in_{OclAny}\ person7)$
$(oid5 \mapsto in_{Person}\ person6)$
$(*oid4*)$
$(oid3 \mapsto in_{Person}\ person4)$
$(oid2 \mapsto in_{Person}\ person3)$
$(oid1 \mapsto in_{Person}\ person2)$
$(oid0 \mapsto in_{Person}\ person1)$
$,\ assocs = assocs\ \sigma_1{}'\ |\!)$
⟨*proof*⟩

declare *const-ss* [*simp*]

lemma $\bigwedge\sigma_1.$
$(\sigma_1,\sigma_1{}') \models (Person\ .allInstances() \doteq Set\{\ X_{Person}1,\ X_{Person}2,\ X_{Person}3,\ X_{Person}4(*,\ X_{Person}5*),\ X_{Person}6,$
$X_{Person}7\ .oclAsType(Person)(*,\ X_{Person}8*),\ X_{Person}9\ \})$

$\langle proof \rangle$

lemma $\bigwedge \sigma_1.$
$(\sigma_1,\sigma_1{}') \models (OclAny\ .allInstances() \doteq Set\{\ X_{Person}1\ .oclAsType(OclAny),\ X_{Person}2\ .oclAsType(OclAny),$
$X_{Person}3\ .oclAsType(OclAny),\ X_{Person}4\ .oclAsType(OclAny)$
$(*,\ X_{Person}5*),\ X_{Person}6\ .oclAsType(OclAny),$
$X_{Person}7,\ X_{Person}8,\ X_{Person}9\ .oclAsType(OclAny)\ \})$

$\langle proof \rangle$

end

theory
Employee-DesignModel-OCLPart
imports
Employee-DesignModel-UMLPart
begin

8.10. OCL Part: Standard State Infrastructure

Ideally, these definitions are automatically generated from the class model.

8.11. 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 $Person\text{-}label_{inv}\ (self) \equiv$
$(self\ .boss <> null\ implies\ (self\ .salary\ \leq_{int}\ ((self\ .boss)\ .salary)))$

definition $Person\text{-}label_{invATpre} :: Person \Rightarrow Boolean$
where $Person\text{-}label_{invATpre}\ (self) \equiv$
$(self\ .boss@pre <> null\ implies\ (self\ .salary@pre \leq_{int} ((self\ .boss@pre)\ .salary@pre)))$

definition $Person\text{-}label_{globalinv} :: Boolean$

where $Person\text{-}label_{globalinv} \equiv (Person\ .allInstances()\text{-}\!\!>\!forAll(x \mid Person\text{-}label_{inv}\ (x))\ and$

$(Person\ .allInstances@pre()\text{-}\!\!>\!forAll(x \mid Person\text{-}label_{invATpre}\ (x))))$

lemma $\tau \models \delta\ (X\ .boss) \Longrightarrow \tau \models Person\ .allInstances()\text{-}\!\!>\!includes(X\ .boss) \wedge$
$\tau \models Person\ .allInstances()\text{-}\!\!>\!includes(X)$
⟨*proof*⟩

lemma $REC\text{-}pre : \tau \models Person\text{-}label_{globalinv}$
$\Longrightarrow \tau \models Person\ .allInstances()\text{-}\!\!>\!includes(X)$ (* *X represented object in state* *)
$\Longrightarrow \exists\ REC.\ \ \tau \models REC(X) \triangleq (Person\text{-}label_{inv}\ (X)\ and\ (X\ .boss <> null\ implies\ REC(X\ .boss)))$
⟨*proof*⟩

This allows to state a predicate:

axiomatization $inv_{Person\text{-}label} :: Person \Rightarrow Boolean$
where $inv_{Person\text{-}label}\text{-}def$:
$(\tau \models Person\ .allInstances()\text{-}\!\!>\!includes(self)) \Longrightarrow$
$(\tau \models (inv_{Person\text{-}label}(self) \triangleq\ (self\ .boss <> null\ implies$
$(self\ .salary\ \leq_{int}\ ((self\ .boss)\ .salary))\ and$
$inv_{Person\text{-}label}(self\ .boss))))$

axiomatization $inv_{Person\text{-}labelATpre} :: Person \Rightarrow Boolean$
where $inv_{Person\text{-}labelATpre}\text{-}def$:
$(\tau \models Person\ .allInstances@pre()\text{-}\!\!>\!includes(self)) \Longrightarrow$
$(\tau \models (inv_{Person\text{-}labelATpre}(self) \triangleq (self\ .boss@pre <> null\ implies$
$(self\ .salary@pre\ \leq_{int}\ ((self\ .boss@pre)\ .salary@pre))\ and$
$inv_{Person\text{-}labelATpre}(self\ .boss@pre))))$

lemma $inv\text{-}1$:
$(\tau \models Person\ .allInstances()\text{-}\!\!>\!includes(self)) \Longrightarrow$
$(\tau \models inv_{Person\text{-}label}(self) = ((\tau \models (self\ .boss \doteq null)) \vee$
$(\ \tau \models (self\ .boss <> null) \wedge$
$\tau \models ((self\ .salary)\ \leq_{int}\ (self\ .boss\ .salary))\ \wedge$
$\tau \models (inv_{Person\text{-}label}(self\ .boss)))))$
⟨*proof*⟩

lemma $inv\text{-}2$:
$(\tau \models Person\ .allInstances@pre()\text{-}\!\!>\!includes(self)) \Longrightarrow$
$(\tau \models inv_{Person\text{-}labelATpre}(self)) =\ ((\tau \models (self\ .boss@pre \doteq null)) \vee$
$(\tau \models (self\ .boss@pre <> null) \wedge$
$(\tau \models (self\ .boss@pre\ .salary@pre \leq_{int} self\ .salary@pre))\ \wedge$
$(\tau \models (inv_{Person\text{-}labelATpre}(self\ .boss@pre)))))$
⟨*proof*⟩

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* ⇒ $(\mathfrak{A})st$ ⇒ *bool* **where**

$$
\begin{aligned}
(\tau \models (\delta\ \mathit{self})) \Longrightarrow ((\tau \models (\mathit{self}\ .\mathit{boss} \doteq \mathit{null})) \vee \\
\quad (\tau \models (\mathit{self}\ .\mathit{boss} <> \mathit{null}) \wedge (\tau \models (\mathit{self}\ .\mathit{boss}\ .\mathit{salary} \leq_{int} \mathit{self}\ .\mathit{salary})) \wedge \\
\quad (\ (\mathit{inv}(\mathit{self}\ .\mathit{boss}))\tau\ ))) \\
\quad \Longrightarrow (\ \mathit{inv}\ \mathit{self}\ \tau)
\end{aligned}
$$

8.12. The Contract of a Recursive Query

This part is analogous to the Analysis Model and skipped here.

end

Part III.

Conclusion

9. Conclusion

9.1. Lessons Learned and Contributions

We provided a typed and type-safe shallow embedding of the core of UML [28, 29] and OCL [30]. Shallow embedding means that types of OCL were injectively, i. e., mapped by the embedding one-to-one to types in Isabelle/HOL [26]. 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 5.1.5) 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 [18].

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.

[1] Our two examples of Employee_AnalysisModel and Employee_DesignModel (see Chapter 7 and Figure II as well as Chapter 8 and Figure II) sketch how this construction can be captured by an automated process.

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.

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 [24] for semi-automated code verification, and has been considered desirable in the Aachen Meeting [14]).

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()`.

9.2. Lessons Learned

While our paper and pencil arguments, given in [12], 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 [33] or SMT-solvers like Z3 [19] completely impractical. Concretely, if the expression `not(null)` is defined `invalid` (as is the case in the present standard [30]), than standard involution does not hold, i. e., `not(not(A))` = `A` does not hold universally. Similarly, if `null and null` is `invalid`, then not even idempotence `X and X` = `X` holds. We strongly argue in favor of a lattice-like organization, where `null` represents "more information" than `invalid` and the logical operators are monotone with respect to this semantical "information ordering."

A similar experience with prior paper and pencil arguments was our investigation of the object-oriented data-models, in particular path-expressions [15]. 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 [13] for details)) are valid in Featherweight OCL.

9.3. 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 [32]) 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 [13]. 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 [33] based animator in Isabelle can give a similar quality of animation as the OCLexec Tool [23]
- 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.

Contents

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 and B. Wolff. Semantics, calculi, and analysis for object-oriented specifications. *Acta Informatica*, 46(4):255–284, July 2009. ISSN 0001-5903. doi: 10.1007/s00236-009-0093-8. URL http://www.brucker.ch/bibliography/abstract/brucker.ea-semantics-2009.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[17] 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.

[18] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The amsterdam manifesto on OCL. In Clark and Warmer [17], pages 115–149. ISBN 3-540-43169-1.

[19] 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.

[20] M. Gogolla and M. Richters. Expressing UML class diagrams properties with OCL. In Clark and Warmer [17], pages 85–114. ISBN 3-540-43169-1.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Object Management Group. UML 2.0 OCL specification, Oct. 2003. Available as OMG document ptc/03-10-14.

[28] Object Management Group. UML 2.4.1: Infrastructure specification, Aug. 2011. Available as OMG document formal/2011-08-05.

[29] Object Management Group. UML 2.4.1: Superstructure specification, Aug. 2011. Available as OMG document formal/2011-08-06.

[30] Object Management Group. UML 2.3.1 OCL specification, Feb. 2012. Available as OMG document formal/2012-01-01.

[31] 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.

[32] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[33] 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.

[34] 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.

[35] 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.