A Formal Machine-Checked Semantics for OCL

A Draft Proposal for Annex A of the OCL Standard

Achim D. Brucker[*] Frédéric Tuong[‡] Burkhart Wolff[†]

September 6, 2014

[*]SAP SE, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
achim.brucker@sap.com

[‡]Univ. Paris-Sud, IRT SystemX, 8 av. de la Vauve,
91120 Palaiseau, France
frederic.tuong@{u-psud, irt-systemx}.fr

[†]Univ. Paris-Sud, Laboratoire LRI, UMR8623, 91405 Orsay, France
CNRS, 91405 Orsay, France
burkhart.wolff@lri.fr

A. Formal Semantics of OCL

A.1. Introduction

This annex chapter formally defines the semantics of OCL. This chapter is a, to a large extend automatically generated, summary of a formal semantics of the core of OCL, called Featherweight OCL[1]. Featherweight OCL has a formal semantics in Isabelle/HOL [18]

The semantic definitions are in large parts executable, in some parts only provable, namely the essence of Set-constructions. The first goal of its construction is *consistency*, i. e., it should be possible to apply logical rules and/or evaluation rules for OCL in an arbitrary manner always yielding the same result. Moreover, except in pathological cases, this result should be unambiguously defined, i. e., represent a value.

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

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

[1] An updated, machine-checked version and formally complete version of the complete formalization 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 A.5.7.

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 *cast* via `oclIsTypeOf(T)` one to the other, and under particular conditions to be described in detail later, these casts are semantically *lossless*, i. e.

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

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

Here is a feature-list of 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 is a strongly typed language in the Hindley-Milner tradition. We assume that a pre-process for full OCL eliminates all implicit conversions due to subtyping by introducing explicit casts (e. g., `oclAsType(Class)`). [4]
- 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 `allInstances()`, or `oclIsNew()`. The object universe construction 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 [18]. 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 be represented in Featherweight OCL and a type in Featherweight OCL contains exactly the values that are possible in OCL .

[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 [3].
[5] following the tradition of HOL-OCL [5]

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 [22] and [12, 15, 17], leading to a number of formal, machine-checked versions, most notably HOL-OCL [3–5, 7] and more recent approaches [9]. All of them have in common the attempt to reconcile the conflicting demands of an industrially used specification language and its various stakeholders, the needs of OMG standardization process and the desire for sufficient logical precision for tool-implementors, in particular from the Formal Methods research community. To discuss the future directions of the standard, several OCL experts met in November 2013 in Aachen to discuss possible mid-term improvements of OCL, strategies of standardization of OCL within the OMG, and a vision for possible long-term developments of the language [8]. The participants agreed that future proposals for a formal semantics should be machine-check, to ensure the absence of syntax errors, the consistency of the formal semantics, as well as provide a a suite of corner-cases relevant for OCL tool implementors.

Organization of this document. This document is organized as follows. After a brief background section introducing a running example and basic knowledge on Isabelle/HOL and its formal notations, we present the formal semantics of 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 operation contracts.
3. Since the latter, i. e. the construction of UML class models, has to be done on the meta-level (so not *inside* HOL, rather on the level of a pre-compiler), we will describe this process with two larger examples, namely formalizations of our running example.

A.2. Background

A.2.1. Formal Foundation

Isabelle

Isabelle [18] 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

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

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

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

Isabelle supports forward- and backward reasoning on rules. For backward-reasoning, a *proof-state* can be initialized and further transformed into others. For example, a proof of ϕ, using the Isar [24] language, will look as follows in Isabelle:

$$\begin{array}{l} \text{lemma label:} \quad \phi \\ \quad \text{apply(case\_tac)} \\ \quad \text{apply(simp\_all)} \\ \text{done} \end{array} \tag{A.4}$$

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

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

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

Higher-order Logic (HOL)

Higher-order logic (HOL) [1, 10] is a classical logic based on a simple type system. It provides the usual logical connectives like $\_ \land \_$, $\_ \to \_$, $\neg \_$ 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 [21] and the SMT-solver Z3 [13].

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

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

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

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

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

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

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

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

respectively

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

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

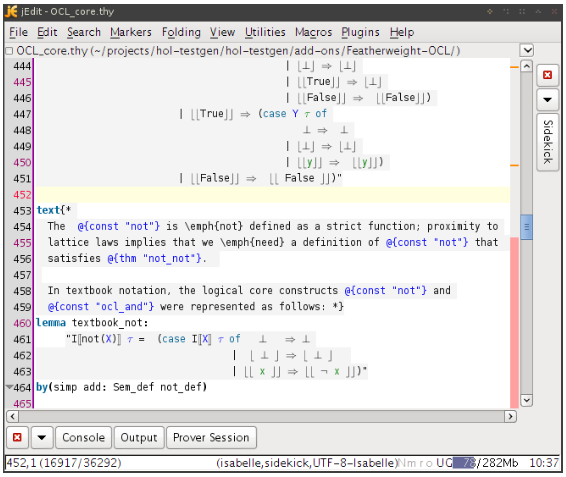
$$
\begin{array}{ll}
(\text{case}\ []\ \text{of}\ [] \Rightarrow F \mid (a\#r) \Rightarrow G\,a\,r) = F & \\
(\text{case}\ b\#t\ \text{of}\ [] \Rightarrow F \mid (a\#r) \Rightarrow G\,a\,r) = G\ b\ t & \\
[] \neq a\#t & \text{– distinctness} \\
[\![a = [] \rightarrow P; \exists\, x\ t\,.\ a = x\#t \rightarrow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

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

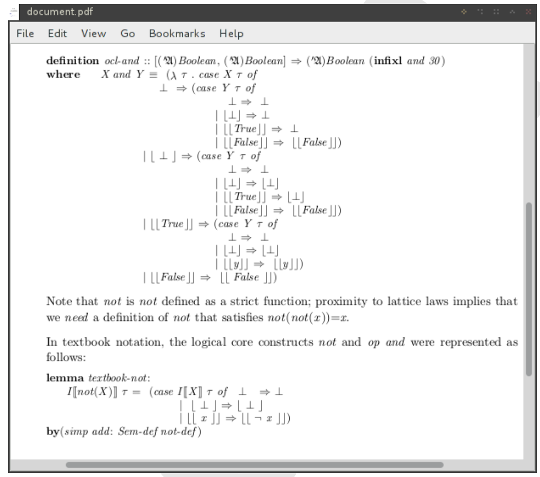
$$
\begin{array}{lll}
\text{fun} & \text{ins} & ::[\alpha :: \text{linorder}, \alpha\,\text{list}] \Rightarrow \alpha\,\text{list} \\
\text{where} & \text{ins}\,x\,[\,] & = [x] \\
 & \text{ins}\,x\,(y\#ys) & = \text{if}\,x < y\,\text{then}\,x\#y\#ys\,\text{else}\,y\#(\text{ins}\,x\,ys)
\end{array}
\tag{A.11}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

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

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



(a) The Isabelle jEdit environment.



(b) The generated formal document.

Figure A.1.: Generating documents with guaranteed syntactical and semantical consistency.

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

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

A.2.2. How this Annex A was Generated from Isabelle/HOL Theories

Isabelle, as a framework for building formal tools [23], provides the means for generating *formal documents*. With formal documents (such as the one you are currently reading) we refer to documents that are machine-generated and ensure certain formal guarantees. In particular, all formal content (e. g., definitions, formulae, types) are checked for consistency during the document generation.

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

Figure A.1 illustrates this approach: Figure A.1a shows the jEdit-based development environment of Isabelle with an excerpt of one of the core theories of Featherweight OCL . Figure A.1b 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 ensures

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

A.3. The Essence of UML-OCL Semantics

A.3.1. The Theory Organization

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

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

Denotational Semantics of Types

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

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

Types were directly represented in Featherweight OCL by types in HOL; consequently, any Featherweight OCL type must provide elements for a bottom element (also denoted $\bot$) and a null element; this is enforced in Isabelle by a type-class null that contains two distinguishable elements bot and null (see Section A.4 for the details of the construction).

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

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

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

As a consequence of this definition of the type, we have the elements $\bot, \llcorner\bot\lrcorner, \llcorner\llcorner\text{true}\lrcorner\lrcorner, \llcorner\llcorner\text{false}\lrcorner\lrcorner$ in the carrier-set of $\text{Boolean}_{\text{base}}$. We can therefore use the element $\bot$ to define the generic type class element $\bot$ and $\llcorner\bot\lrcorner$ for the generic type class null. For collection types and object types this definition is more evolved (see Section A.4).

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

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

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

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

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

the other cases are analogous. In the subsequent subsections, we will drop the index $\mathfrak{A}$ since it is constant in all formulas and expressions except for operations related to the object universe construction in Section A.6.1

The rules of the logical layer (there are no algebraic rules related to the semantics of types), and more details can be found in Section A.4.

A.3.2. Denotational Semantics of Constants and Operations

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

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

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

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

For reasons of conciseness, we will write $\delta\ X$ for $\texttt{not}(X\texttt{.oclIsUndefined())}$ and $\upsilon\ X$ for $\texttt{not}(X\texttt{.oclIsInvalid())}$ throughout this document.

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

$$I[\![\texttt{true} :: \texttt{Boolean}]\!]\tau = {}_{\sqcup}\text{true}_{\sqcup}$$

we can therefore write:

$$\texttt{true} :: \texttt{Boolean} = \lambda\, \tau. {}_{\sqcup}\text{true}_{\sqcup}$$

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

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

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 [19, 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 $\_.\mathrm{a}$ by their counterparts `_.a@pre`. For example, $(\mathit{self}.\mathrm{a} > 5)_{\text{pre}}$ is just $(\mathit{self}.\mathrm{a}\texttt{@pre} > 5)$. 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:

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

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

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

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

$$
\begin{array}{ll}
I[\![x + y]\!]\tau = & \text{if}\, I[\![\delta\ x]\!]\tau = I[\![\texttt{true}]\!]\tau \wedge I[\![\delta\ y]\!]\tau = I[\![\texttt{true}]\!]\tau \\
& \text{then}\, \llcorner\llcorner \ulcorner\ulcorner I[\![x]\!]\tau \urcorner\urcorner + \ulcorner\ulcorner I[\![y]\!]\tau \urcorner\urcorner \lrcorner\lrcorner \\
& \text{else}\, \bot
\end{array}
$$

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

A.3.3. Logical Layer

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

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

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

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

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

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

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

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

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

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

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

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

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

It enjoys nearly the laws of a congruence:

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

where the predicate cp stands for *context-passing*, a property that is true for all pure OCL expressions (but not arbitrary mixtures of OCL and HOL) in Featherweight OCL . The necessary side-calculus for establishing cp can be fully automated; the reader interested in the details is referred to Section A.5.1.

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

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

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

The logical layer of the 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 [13]. 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 already mentioned general case-distinction

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

which is possible for any OCL type, a case distinction on the variables in a formula can be performed; due to strictness rules, formulae containing somewhere a variable x that is known to be `invalid` or `null` reduce usually quickly to contradictions. For example, we can infer from an invariant $\tau \models x \doteq y - 3$ that we have $\tau \models x \doteq y - 3 \wedge \tau \models \delta\, x \wedge \tau \models \delta\, y$. We call the latter formula the δ-closure of the former. Now, we can convert a formula like $\tau \models x > 0\, \texttt{or}\, 3 * y > x * x$ into the equivalent formula $\tau \models x > 0 \vee \tau \models 3 * y > x * x$ and thus internalize the OCL-logic into a classical (and more tool-conform) logic. This works—for the price of a potential, but due to the usually "rich" δ-closures of invariants rare—exponential blow-up of the formula for all OCL formulas.

Algebraic Layer

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

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

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

$$
\begin{aligned}
(\texttt{true}\ \texttt{and}\ \texttt{true}) &= \texttt{true} & (\texttt{true}\ \texttt{and}\ \texttt{false}) &= \texttt{false}\\
(\texttt{true}\ \texttt{and}\ \texttt{null}) &= \texttt{null} & (\texttt{true}\ \texttt{and}\ \texttt{invalid}) &= \texttt{invalid}\\
(\texttt{invalid}\ \texttt{and}\ \texttt{true}) &= \texttt{invalid} & (\texttt{invalid}\ \texttt{and}\ \texttt{false}) &= \texttt{false}\\
(\texttt{invalid}\ \texttt{and}\ \texttt{null}) &= \texttt{invalid} & (\texttt{invalid}\ \texttt{and}\ \texttt{invalid}) &= \texttt{invalid}
\end{aligned}
$$

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

$$
\begin{aligned}
X\ \texttt{and}\ X &= X & X\ \texttt{and}\ Y &= Y\ \texttt{and}\ X\\
\texttt{false}\ \texttt{and}\ X &= \texttt{false} & X\ \texttt{and}\ \texttt{false} &= \texttt{false}\\
\texttt{true}\ \texttt{and}\ X &= X & X\ \texttt{and}\ \texttt{true} &= X
\end{aligned}
$$

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

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 of the standard and the major deviation point from HOL-OCL [4, 6] to Featherweight OCL as presented here. Expressed in algebraic equations, "strictness-principles" boil down to:

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

$$
\texttt{S->including(invalid)} = \texttt{invalid}
$$

$$
X \doteq X = (\texttt{if}\ \upsilon\, x\ \texttt{then}\ \texttt{true}\texttt{else}\ \texttt{invalid}\ \texttt{endif})
$$

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

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

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

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 applying these algebraic laws (which can give rise to efficient compilations). The reader interested in the list of "test-statements" like:

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

make consult Section A.5.8; these test-statements have been machine-checked and proven consistent with the denotational and logic semantics of Featherweight OCL.

A.3.4. Object-oriented Datatype Theories

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

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

1. C is a set of class names (written as $\{C_1, \ldots, C_n\}$). To each class name a type of data in OCL is associated. Moreover, class names declare two projector functions to the set of all objects in a state: C_i`.allInstances()` and C_i`.allInstances@pre()`,
2. $\_ < \_$ is an inheritance relation on classes,
3. $Attrib(C_i)$ is a collection of attributes associated to classes C_i. It declares two families of accessors; for each attribute $a \in Attrib(C_i)$ in a class definition C_i (denoted $X.a :: C_i \to A$ and $X.a$ `@pre` $:: C_i \to A$ for $A \in TYPES(C)$),
4. $Assoc(C_i, C_j)$ is a collection of associations [8]. An association $(n, rn_{from}, rn_{to}) \in Assoc(C_i, C_j)$ between to classes C_i and C_j is a triple consisting of a (unique) association name n, and the role-names rn_{to} and rn_{from}. To each role-name belong two families of accessors denoted $X.a :: C_i \to A$ and $X.a$ `@pre` $:: C_i \to A$ for $A \in TYPES(C)$),
5. for each pair $C_i < C_j$ $(C_i, C_j < C)$, there is a cast operation of type $C_j \to C_i$ that can change the static type of an object of type C_i: $obj :: C_i$`.oclAsType(`C_j`)`,
6. for each class $C_i \in C$, there are two dynamic type tests (X`.oclIsTypeOf(`C_i`)` and X`.oclIsKindOf(`C_i`)`),
7. and last but not least, for each class name $C_i \in C$ there is an instance of the overloaded referential equality (written $\_ \doteq \_$).

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

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

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

A Denotational Space for Class-Models: Object Universes

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

$$\text{typedef} \qquad \alpha \ \text{state} := \{\sigma :: \text{oid} \rightharpoonup \alpha \mid \text{inv}_\sigma(\sigma)\} \tag{A.15}$$

where inv_σ is a to be discussed invariant on states.

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

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

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

For the sake of simplicity, the present semantics chose the first option for Featherweight OCL, while HOL-OCL [5] used an involved construction allowing the latter.

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

$$\mathfrak{A} := C_1 + \cdots + C_n \,. \tag{A.16}$$

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:

$$X\texttt{.oclIsTypeOf(}C_k\texttt{)}\ \texttt{implies}\ X\texttt{.oclAsType(}C_i\texttt{).oclAsType(}C_k\texttt{)} \doteq X \tag{A.17}$$

$$\text{whenever } C_k < C_i \text{ and } X \text{ is valid.} \tag{A.18}$$

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

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

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

Example instances of this scheme—outlining a compiler—can be found in Section A.7.

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

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

- there is a generic theory of states, which must be formulated independently from a concrete object universe,
- there is a principle of translation (captured by the inductive scheme for class type extensions and class types above) that converts a given class model into an concrete object universe,
- there are fixed principles that allow to derive the semantic theory of any concrete object universe, called the *object-oriented datatype theory.*

We will work out concrete examples for the construction of the object-universes in Section A.7 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 annex which has a focus on the semantic construction and its presentation.

Denotational Semantics of Accessors on Objects and Associations

Our choice to use a shallow embedding of OCL in HOL and, thus having an injective mapping from OCL types to HOL types, results in type-safety of 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 cast to the expected format. The exceptional case of non-existence 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 non-existence in this state must be treated.

The first phase directly translates into the following formalization:

definition

$$\begin{array}{rlll} \mathrm{eval\_extract}\, X\, f = (\lambda\, \tau.\ \mathrm{case}\, X\, \tau\, \mathrm{of} & \bot & \Rightarrow \mathtt{invalid}\, \tau & \text{exception} \\ \mid & \lfloor \bot \rfloor & \Rightarrow \mathtt{invalid}\, \tau & \text{deref. null} \\ \mid & \lfloor\lfloor obj \rfloor\rfloor & \Rightarrow f\, (\mathrm{oid\_of}\ obj)\, \tau) & \end{array} \tag{A.19}$$

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

$$\text{definition deref\_oid}_C\ \mathit{fst\_snd}\ f\ \mathit{oid} = (\lambda\, \tau.\ \mathrm{case}\, (\mathrm{heap}\, (\mathit{fst\_snd}\, \tau))\ \mathit{oid}\ \mathrm{of}$$

$$\begin{array}{ll} \lfloor \mathrm{in}_C\, obj \rfloor & \Rightarrow f\, obj\, \tau \\ \mid \_ & \Rightarrow \mathtt{invalid}\, \tau) \end{array} \tag{A.20}$$

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:

$$\begin{array}{rllll} \text{definition select}_a\ f = (\lambda & \mathrm{mk}_C\ \mathit{oid} & \cdots \bot \cdots & C_{X\mathrm{ext}} & \Rightarrow \mathtt{null} \\ \mid & \mathrm{mk}_C\ \mathit{oid} & \cdots \lfloor a \rfloor \cdots & C_{X\mathrm{ext}} & \Rightarrow f\, (\lambda\, x\, \_.\ \lfloor\lfloor x \rfloor\rfloor)\, a) \end{array} \tag{A.21}$$

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:

$$\begin{array}{llll} \text{definition} & \text{in\_pre\_state} & = \text{fst} & \text{first component} \\ \text{definition} & \text{in\_post\_state} & = \text{snd} & \text{second component} \\ \text{definition} & \text{reconst\_basetype} & = \text{id} & \text{identity function} \end{array} \tag{A.22}$$

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

$$\begin{array}{lll} \text{definition} & \_\texttt{.getBase} & :: C \Rightarrow A_{base} \\ \text{where} & X\texttt{.getBase} & = \text{eval\_extract}\ X\ (\text{deref\_oid}_C\ \text{in\_post\_state} \\ & & \quad (\text{select}_{\text{getBase}}\ \text{reconst\_basetype})) \end{array} \tag{A.23}$$

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

$$\begin{array}{lll} \text{definition} & \_\texttt{.getObject} & :: C \Rightarrow A_{object} \\ \text{where} & X\texttt{.getObject} & = \text{eval\_extract}\ X\ (\text{deref\_oid}_C\ \text{in\_post\_state} \\ & & \quad (\text{select}_{\text{getObject}}\ (\text{deref\_oid}_C\ \text{in\_post\_state}))) \end{array} \tag{A.24}$$

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 A.7.8. The construction of casts and type tests `->oclIsTypeOf()` and `->oclIsKindOf()` is similarly.

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

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

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

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

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

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

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

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

If the upper bound n is infinite, the second invariant is omitted. For the definition of these invariants we are making use of the conversion of single values to sets described in Section A.3.4. If $n \leq 1$, the attribute `a` evaluates to a single value, which is then converted to a `Set` on which the `size` operation is called.

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

Logic Properties of Class-Models

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

$$\tau \vDash X\,\texttt{.oclAsType(}C_i\texttt{)} \triangleq X$$

$$\tau \vDash \texttt{invalid .oclAsType(}C_i\texttt{)} \triangleq \texttt{invalid}$$

$$\tau \vDash \texttt{null .oclAsType(}C_i\texttt{)} \triangleq \texttt{null}$$

$$\tau \vDash \texttt{((}X :: C_i\texttt{).oclAsType(C\_j) .oclAsType(}C_i\texttt{)} \triangleq X\texttt{)}$$

$$\tau \vDash X\,\texttt{.oclAsType(}C_j\texttt{) .oclAsType(}C_i\texttt{)} \triangleq X$$

$$\tau \vDash \texttt{(}X :: \mathit{OclAny}\texttt{) .oclAsType(}\mathit{OclAny}\texttt{)} \triangleq X$$

$$\tau \vDash \upsilon(X :: C_i) \Longrightarrow \tau \vDash (X\,\texttt{.oclIsTypeOf(}C_i\texttt{) implies (}X\,\texttt{.oclAsType(}C_j\texttt{).oclAsType(}C_i\texttt{))} \doteq X)$$

$$\tau \vDash \upsilon(X :: C_i) \Longrightarrow \tau \vDash X\,\texttt{.oclIsTypeOf(}C_i\texttt{) implies (}X\,\texttt{.oclAsType(}C_j\texttt{) .oclAsType(}C_i\texttt{))} \doteq X$$

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

$$\tau \vDash \delta X \Longrightarrow \tau \vDash X\texttt{.oclAsType(}C_j\texttt{)}\ \ \texttt{.oclAsType(}C_i\texttt{)} \triangleq X$$
$$\tau \vDash \upsilon X \Longrightarrow \tau \vDash X\texttt{.oclIsTypeOf(}C_i\texttt{)}\ \ \texttt{implies}X\texttt{.oclAsType(}C_j\texttt{)}\ \ \texttt{.oclAsType(}C_i\texttt{)} \doteq X$$
$$\tau \vDash X\texttt{.oclIsTypeOf(}C_j\texttt{)} \Longrightarrow \tau \vDash \delta X \Longrightarrow \tau \vDash \texttt{not}(\upsilon X\texttt{.oclAsType(}C_i\texttt{))}$$
$$\tau \vDash \texttt{invalid.oclIsTypeOf(}C_i\texttt{)} \triangleq \texttt{invalid}$$
$$\tau \vDash \texttt{null}\ \ \texttt{.oclIsTypeOf(}C_i\texttt{)} \triangleq \texttt{true}$$
$$\tau \vDash \mathit{Person}\texttt{.allInstances()->forAll(}X|X\texttt{.oclIsTypeOf(}C_z\texttt{))}$$
$$\tau \vDash \mathit{Person}\texttt{.allInstances@pre()->forAll(}X|X\texttt{.oclIsTypeOf(}C_z\texttt{))}$$
$$\tau \vDash \mathit{Person}\texttt{.allInstances()->forAll(}X|X\texttt{.oclIsKindOf(}C_i\texttt{))}$$
$$\tau \vDash \mathit{Person}\texttt{.allInstances@pre()->forAll(}X|X\texttt{.oclIsKindOf(}C_i\texttt{))}$$
$$\tau \vDash (X :: C_i)\texttt{.oclIsTypeOf(}C_j\texttt{)} \Longrightarrow \tau \vDash\ (X :: C_i)\ \texttt{.oclIsKindOf(}C_i\texttt{)}$$
$$(\tau \vDash (X :: C_j) \doteq X) = (\tau \vDash \texttt{if}\upsilon X\texttt{then true else invalid endif})$$
$$\tau \vDash (X :: C_j) \doteq Y \Longrightarrow \tau \vDash Y \doteq X$$
$$\tau \vDash (X :: C_j) \doteq Y \Longrightarrow \tau \vDash Y \doteq Z \Longrightarrow \tau \vDash X \doteq Z$$

Algebraic Properties of the Class-Models

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

$$\texttt{invalid.oclIsTypeOf(}C_i\texttt{)} = \texttt{invalid} \qquad \texttt{null.oclIsTypeOf(}C_i\texttt{)} = \texttt{true}$$
$$\texttt{invalid.oclIsKindOf(}C_i\texttt{)} = \texttt{invalid} \qquad \texttt{null.oclIsKindOf(}C_i\texttt{)} = \texttt{true}$$
$$(X :: C_i)\texttt{.oclAsType(}C_i\texttt{)} = X \qquad \texttt{invalid.oclAsType(}C_i\texttt{)} = \texttt{invalid}$$
$$\texttt{null.oclAsType(}C_i\texttt{)} = \texttt{null} \qquad (X :: C_i)\texttt{.oclAsType(}C_j\texttt{)}\texttt{.oclAsType(}C_i\texttt{)} = X$$
$$(X :: C_i) \doteq X = \texttt{if}\,\upsilon\, X\texttt{then true els invalid endif}$$

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

$$\texttt{invalid.a} = \texttt{invalid} \qquad \texttt{null.a} = \texttt{invalid}$$
$$\texttt{invalid.a@pre} = \texttt{invalid} \qquad \texttt{null.a@pre} = \texttt{invalid}$$
$$\texttt{invalid.r} = \texttt{invalid} \qquad \texttt{null.r} = \texttt{invalid}$$
$$\texttt{invalid.r@pre} = \texttt{invalid} \qquad \texttt{null.r@pre} = \texttt{invalid}$$

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. OCL semantics admits states with "dangling references,"; it is the semantics of accessors or roles which maps these references to `invalid`, which makes it possible to rule out these situations in invariants.

OCL does not guarantee that an operation only modifies the path-expressions mentioned in the postcondition, i. e., it allows arbitrary relations from pre-states to post-states. This framing problem is well-known (one of the suggested solutions is [16]). 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:

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

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 $\tau \vDash X\texttt{->forAll}(x \text{not} | (x \doteq s.a))$, we can infer that $\tau \vDash s.a \triangleq s.a\,\texttt{@pre}$.

A.3.5. Data Invariants

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

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

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

$$\begin{aligned} & I[\![\texttt{context}\ c : C_i\ \texttt{inv}\ n : \phi(c)]\!]\tau \equiv \\ & \quad \tau \vDash (C_i\,\texttt{.allInstances()->forall}(x|\phi(x))) \land \\ & \quad \tau \vDash (C_i\,\texttt{.allInstances()->forall}(x|\phi(x)))_{\text{pre}} \end{aligned} \tag{A.25}$$

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

A.3.6. Operation Contracts

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

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

result is `invalid`. This is reflected by the following definition of the contract semantics:

$$
\begin{aligned}
&I[\![\texttt{context}\; C \;::\; op(a_1,\ldots,a_n) : T\\
&\qquad \texttt{pre}\; \phi(\mathit{self}, a_1,\ldots,a_n)\\
&\qquad \texttt{post}\; \psi(\mathit{self}, a_1,\ldots,a_n, \mathit{result})]\!] \equiv\\
&\qquad\quad \lambda\, s, x_1,\ldots,x_n,\tau.\\
&\qquad\qquad \text{if}\;\; \tau \vDash \partial\, s \wedge \tau \vDash \upsilon\, x_1 \wedge \ldots \wedge \tau \vDash \upsilon\, x_n\\
&\qquad\qquad \text{then SOME}\; \mathit{result}.\quad \tau \vDash \phi(s, x_1,\ldots,x_n)_{\text{pre}}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge\, \tau \vDash \psi(s, x_1,\ldots,x_n, \mathit{result}))\\
&\qquad\qquad \text{else}\; \bot
\end{aligned}
\tag{A.26}
$$

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

$$f_{op} \equiv I[\![\texttt{context}\; C \;::\; op(a_1,\ldots,a_n) : T \ldots]\!] \tag{A.27}$$

provided that neither ϕ nor ψ contain recursive method calls of *op*. In the case of a query operation (i. e. τ must have the form: (σ,σ), which means that query operations do not change the state; c.f. `oclIsModifiedOnly()` in Section A.3.4), this constraint can be relaxed: the above equation is then stated as *axiom*. Note however, that the consistency of the overall theory is for recursive query contracts left to the user (it can be shown, for example, by a proof of termination, i. e. by showing that all recursive calls were applied to argument vectors that are smaller wrt. to a well-founded ordering). Under this condition, an f_{op} resulting from recursive query operations can be used safely inside pre- and post-conditions of other contracts.

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

$$
\frac{\begin{array}{c}[\tau \vDash \psi\; self\; a_1 \ldots a_n\; res]_{res}\\ \vdots\\ \tau \vDash E(res)\end{array}}{\tau \vDash E(f_{op}\; self\; a_1 \ldots a_n)}
\tag{A.28}
$$

under the conditions:

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

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

$$
\frac{\tau \vDash \phi\; self\; a_1 \ldots a_n}{(\tau \vDash E(f_{op}\; self\; a_1 \ldots a_n)) = e(\tau \vDash E(BODY\; self\; a_1 ... a_n))}
\tag{A.29}
$$

where

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

- E must be an OCL term.
- *self* must be defined, and the arguments valid in τ:
  $\tau \models \partial\ self \wedge \tau \models \upsilon\ a_1 \wedge \ldots \wedge \tau \models \upsilon\ a_n$
- the postcondition $\psi\ self\ a_1\ \ldots\ a_n\ result$ must be decomposable into:
  $\psi'\ self\ a_1\ \ldots a_n$ and $result \triangleq BODY\ self\ a_1\ \ldots\ a_n$.

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

A.4. Formalization I: OCL Types and Core Definitions

A.4.1. Preliminaries

Notations for the Option Type

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

no-notation *ceiling* ($\ulcorner$-$\urcorner$)
no-notation *floor* ($\llcorner$-$\lrcorner$)

notation *Some* ($\llcorner$(-)$\lrcorner$)
notation *None* ($\perp$)

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

fun *drop* :: $'\alpha$ *option* $\Rightarrow$ $'\alpha$ ($\ulcorner$(-)$\urcorner$)
where *drop-lift*[*simp*]: $\ulcorner\llcorner v\lrcorner\urcorner = v$

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* :: $'a \Rightarrow 'a$ ($I[\![$-$]\!]$)
where $I[\![x]\!] \equiv x$

Common Infrastructure for all OCL Types

In order to have the possibility to nest collection types, such that we can give semantics to expressions like $Set\{Set\{\mathbf{2}\},null\}$, 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 $\llcorner \perp \lrcorner$ 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 $null\text{-}is\text{-}valid : null \neq bot$

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
end

instantiation $option :: (bot)null$
begin
  definition $null\text{-}option\text{-}def: (null::'a::bot\ option) \equiv \llcorner bot \lrcorner$
  instance
end

instantiation $fun :: (type,bot)\ bot$
begin
  definition $bot\text{-}fun\text{-}def: bot \equiv (\lambda\, x.\ bot)$
  instance
end

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

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

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

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

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

type-synonym $oid = nat$

We refrained from the alternative:

type-synonym $oid = ind$

which is slightly more abstract but non-executable.

States in UML/OCL are a pair of

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

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

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

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 projects the oid of an object in the state (we will settle the question how to define this function later). We will use the Isabelle type class mechanism [14] to capture this:

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

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

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

Valuations are functions from a state pair (built upon data universe $'\mathfrak{A}$) 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:

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*: $I[\![invalid]\!]\tau = bot$

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*: $I[\![null::('\mathfrak{A},'\alpha::null)\ val]\!]\ \tau = (null::('\alpha::null))$

A.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 *UML-Types.bot-class.bot* and *null*; this type is sufficiently rich to contain at least these two elements. Analogously we build:

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

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

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

type-synonym $Real_{base} = real\ option\ option$
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 $Void_{base} = \{X{::}unit\ option\ option.\ X = bot \lor X = null\ \}$

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

A.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.

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 $({}'\alpha,\ {}'\beta)\ Pair_{base} = \{X{::}({}'\alpha{::}null \times {}'\beta{::}null)\ option\ option.$
$X = bot \lor X = null \lor (fst\ulcorner\ulcorner X\urcorner\urcorner \neq bot \land snd\ulcorner\ulcorner X\urcorner\urcorner \neq bot)\}$

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
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
end

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

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

The Construction of the Set Type

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

typedef $'\alpha\ Set_{base} = \{X{::}('\alpha{::}null)\ set\ option\ option.\ X = bot \lor X = null \lor (\forall x {\in} \ulcorner\ulcorner X \urcorner\urcorner.\ x \neq bot)\}$

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

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

instance
end

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

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

instance
end

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

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

The Construction of the Sequence Type

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

typedef $'\alpha\ Sequence_{base} = \{X{::}('\alpha{::}null)\ list\ option\ option.$
$X = bot \lor X = null \lor (\forall x {\in} set\ \ulcorner\ulcorner X \urcorner\urcorner.\ x \neq bot)\}$

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

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

instance
end

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

definition $null\text{-}Sequence_{base}\text{-}def$: $(null{::}('a{::}null)\ Sequence_{base}) \equiv Abs\text{-}Sequence_{base}\ \llcorner None \lrcorner$

instance
end

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

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

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

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

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

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

Table A.1.: Correspondance between OCL types and HOL types

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.

A.5. Formalization II: OCL Terms and Library Operations

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

Basic Constants

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

lemma *null-Boolean-def* : $(\mathit{null}{::}('\mathfrak{A})\mathit{Boolean}) = (\lambda\ \tau.\ \llcorner\bot\lrcorner)$

definition *true* :: $('\mathfrak{A})\mathit{Boolean}$

where $\quad true \equiv \lambda\ \tau.\ \llcorner\llcorner True \lrcorner\lrcorner$

definition $false :: ('\mathfrak{A})Boolean$
where $\quad false \equiv\ \lambda\ \tau.\ \llcorner\llcorner False \lrcorner\lrcorner$

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

lemma [*simp*]: $false\ (a, b) = \llcorner\llcorner False \lrcorner\lrcorner$

lemma [*simp*]: $true\ (a, b) = \llcorner\llcorner True \lrcorner\lrcorner$

lemma *textbook-true*: $I[\![true]\!]\ \tau = \llcorner\llcorner True \lrcorner\lrcorner$

lemma *textbook-false*: $I[\![false]\!]\ \tau = \llcorner\llcorner False \lrcorner\lrcorner$

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

Table A.2.: Basic semantic constant definitions of the logic

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 $\quad \upsilon\ X \equiv\ \lambda\ \tau\ .\ if\ X\ \tau = bot\ \tau\ then\ false\ \tau\ else\ true\ \tau$

lemma *valid1*[*simp*]: $\upsilon\ invalid = false$
lemma *valid2*[*simp*]: $\upsilon\ null = true$
lemma *valid3*[*simp*]: $\upsilon\ true = true$
lemma *valid4*[*simp*]: $\upsilon\ false = true$
definition $defined :: ('\mathfrak{A},'a::null)val \Rightarrow ('\mathfrak{A})Boolean\ (\delta\ \text{-}\ [100]100)$
where $\quad \delta\ X \equiv\ \lambda\ \tau\ .\ if\ X\ \tau = bot\ \tau\ \lor X\ \tau = null\ \tau\ then\ false\ \tau\ else\ true\ \tau$

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

lemma *defined1*[*simp*]: $\delta\ invalid = false$
lemma *defined2*[*simp*]: $\delta\ null = false$
lemma *defined3*[*simp*]: $\delta\ true = true$
lemma *defined4*[*simp*]: $\delta\ false = true$
lemma *defined5*[*simp*]: $\delta\ \delta\ X = true$

lemma *defined6*[*simp*]: $\delta\ \upsilon\ X = true$
lemma *valid5*[*simp*]: $\upsilon\ \upsilon\ X = true$
lemma *valid6*[*simp*]: $\upsilon\ \delta\ X = true$

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

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

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

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

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

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

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

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

The Equalities of OCL

The OCL contains a particular version of equality, written in Standard documents _ = _ and _ <> _ for its negation, which is referred as *weak referential equality* hereafter and for which we use the symbol $\_ \doteq \_$ throughout the formal part of this document. Its semantics is motivated by the desire of fast execution, and similarity to languages like Java and C, but does not satisfy the needs of logical reasoning over OCL expressions and specifications. We therefore introduce a second equality, referred as *strong equality* or *logical equality* and written $\_ \triangleq \_$ which is not present in the current standard but was discussed in prior texts on OCL like the Amsterdam Manifesto [12] and was identified as desirable extension of OCL in the Aachen Meeting [8] 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:

   $$s = t \Longrightarrow P(s) = P(t) \tag{A.30}$$

   "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 $X \triangleq Y \equiv \lambda\ \tau.\ _{\llcorner\llcorner}X\ \tau = Y\ \tau_{\lrcorner\lrcorner}$

From this follow already elementary properties like:

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

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

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

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

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

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

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: $(X \triangleq Y)\tau = \mathit{true}\ \tau$
shows $(P\ X \triangleq P\ Y)\tau = \mathit{true}\ \tau$

lemma *defined7*[*simp*]: $\delta\ (X \triangleq Y) = \mathit{true}$

lemma *valid7*[*simp*]: $\upsilon\ (X \triangleq Y) = \mathit{true}$

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

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* :: $({}'\mathfrak{A})\mathit{Boolean} \Rightarrow ({}'\mathfrak{A})\mathit{Boolean}$ (not)
where $\mathit{not}\ X \equiv\ \lambda\ \tau\ .\ \mathit{case}\ X\ \tau\ \mathit{of}$
$\quad \bot \Rightarrow \bot$
$\quad |\ \lfloor\bot\rfloor \Rightarrow \lfloor\bot\rfloor$
$\quad |\ \lfloor\lfloor x\rfloor\rfloor \Rightarrow \lfloor\lfloor\neg\, x\rfloor\rfloor$

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

lemma *OclNot1*[*simp*]: $not\ invalid = invalid$

lemma *OclNot2*[*simp*]: $not\ null = null$

lemma *OclNot3*[*simp*]: $not\ true = false$

lemma *OclNot4*[*simp*]: $not\ false = true$

lemma *OclNot-not*[*simp*]: $not\ (not\ X) = X$

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

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

$$
\begin{array}{ll}
\lfloor\lfloor False \rfloor\rfloor \Rightarrow & \lfloor\lfloor False \rfloor\rfloor \\
\mid \bot \quad \Rightarrow & (case\ Y\ \tau\ of \\
 & \quad \lfloor\lfloor False \rfloor\rfloor \Rightarrow \lfloor\lfloor False \rfloor\rfloor \\
 & \quad \mid \text{-} \quad \Rightarrow \bot) \\
\mid \lfloor\bot\rfloor \quad \Rightarrow & (case\ Y\ \tau\ of \\
 & \quad \lfloor\lfloor False \rfloor\rfloor \Rightarrow \lfloor\lfloor False \rfloor\rfloor \\
 & \quad \mid \bot \quad \Rightarrow \bot \\
 & \quad \mid \text{-} \quad \Rightarrow \lfloor\bot\rfloor) \\
\mid \lfloor\lfloor True \rfloor\rfloor \Rightarrow & Y\ \tau)
\end{array}
$$

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*:

$$
I[\![not(X)]\!]\ \tau = (case\ I[\![X]\!]\ \tau\ of\ \begin{array}{l} \bot \Rightarrow \bot \\ \mid \lfloor\bot\rfloor \Rightarrow \lfloor\bot\rfloor \\ \mid \lfloor\lfloor x \rfloor\rfloor \Rightarrow \lfloor\lfloor \neg\ x \rfloor\rfloor) \end{array}
$$

lemma *textbook-OclAnd*:

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

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

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

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

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

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

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

lemma *OclAnd5*[*simp*]: $(null\ and\ true) = null$
lemma *OclAnd6*[*simp*]: $(null\ and\ false) = false$
lemma *OclAnd7*[*simp*]: $(null\ and\ null) = null$
lemma *OclAnd8*[*simp*]: $(null\ and\ invalid) = invalid$

lemma *OclAnd9*[*simp*]: $(false\ and\ true) = false$
lemma *OclAnd10*[*simp*]: $(false\ and\ false) = false$
lemma *OclAnd11*[*simp*]: $(false\ and\ null) = false$
lemma *OclAnd12*[*simp*]: $(false\ and\ invalid) = false$

lemma *OclAnd13*[*simp*]: $(true\ and\ true) = true$
lemma *OclAnd14*[*simp*]: $(true\ and\ false) = false$
lemma *OclAnd15*[*simp*]: $(true\ and\ null) = null$
lemma *OclAnd16*[*simp*]: $(true\ and\ invalid) = invalid$

lemma *OclAnd-idem*[*simp*]: $(X\ and\ X) = X$

lemma *OclAnd-commute*: $(X\ and\ Y) = (Y\ and\ X)$

lemma *OclAnd-false1*[*simp*]: $(false\ and\ X) = false$

lemma *OclAnd-false2*[*simp*]: $(X\ and\ false) = false$

lemma *OclAnd-true1*[*simp*]: $(true\ and\ X) = X$

lemma *OclAnd-true2*[*simp*]: $(X\ and\ true) = X$

lemma *OclAnd-bot1*[*simp*]: $\bigwedge\tau.\ X\ \tau \neq false\ \tau \Longrightarrow (bot\ and\ X)\ \tau = bot\ \tau$

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

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

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

lemma *OclAnd-assoc*: $(X\ \mathit{and}\ (Y\ \mathit{and}\ Z)) = (X\ \mathit{and}\ Y\ \mathit{and}\ Z)$

lemma *OclOr1*[*simp*]: $(\mathit{invalid}\ \mathit{or}\ \mathit{true}) = \mathit{true}$
lemma *OclOr2*[*simp*]: $(\mathit{invalid}\ \mathit{or}\ \mathit{false}) = \mathit{invalid}$
lemma *OclOr3*[*simp*]: $(\mathit{invalid}\ \mathit{or}\ \mathit{null}) = \mathit{invalid}$
lemma *OclOr4*[*simp*]: $(\mathit{invalid}\ \mathit{or}\ \mathit{invalid}) = \mathit{invalid}$

lemma *OclOr5*[*simp*]: $(\mathit{null}\ \mathit{or}\ \mathit{true}) = \mathit{true}$
lemma *OclOr6*[*simp*]: $(\mathit{null}\ \mathit{or}\ \mathit{false}) = \mathit{null}$
lemma *OclOr7*[*simp*]: $(\mathit{null}\ \mathit{or}\ \mathit{null}) = \mathit{null}$
lemma *OclOr8*[*simp*]: $(\mathit{null}\ \mathit{or}\ \mathit{invalid}) = \mathit{invalid}$

lemma *OclOr-idem*[*simp*]: $(X\ \mathit{or}\ X) = X$

lemma *OclOr-commute*: $(X\ \mathit{or}\ Y) = (Y\ \mathit{or}\ X)$

lemma *OclOr-false1*[*simp*]: $(\mathit{false}\ \mathit{or}\ Y) = Y$

lemma *OclOr-false2*[*simp*]: $(Y\ \mathit{or}\ \mathit{false}) = Y$

lemma *OclOr-true1*[*simp*]: $(\mathit{true}\ \mathit{or}\ Y) = \mathit{true}$

lemma *OclOr-true2*: $(Y\ \mathit{or}\ \mathit{true}) = \mathit{true}$

lemma *OclOr-bot1*[*simp*]: $\bigwedge\tau.\ X\ \tau \neq \mathit{true}\ \tau \Longrightarrow (\mathit{bot}\ \mathit{or}\ X)\ \tau = \mathit{bot}\ \tau$

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

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

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

lemma *OclOr-assoc*: $(X\ \mathit{or}\ (Y\ \mathit{or}\ Z)) = (X\ \mathit{or}\ Y\ \mathit{or}\ Z)$

lemma *OclImplies-true*: $(X\ \mathit{implies}\ \mathit{true}) = \mathit{true}$

lemma *deMorgan1*: $\mathit{not}(X\ \mathit{and}\ Y) = ((\mathit{not}\ X)\ \mathit{or}\ (\mathit{not}\ Y))$

lemma *deMorgan2*: $\mathit{not}(X\ \mathit{or}\ Y) = ((\mathit{not}\ X)\ \mathit{and}\ (\mathit{not}\ Y))$

A Standard Logical Calculus for OCL

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

Global vs. Local Judgements **lemma** *transform1*: $P = true \Longrightarrow \tau \models P$

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

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

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

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 $\tau \models P \Longrightarrow \tau \models Q$

Local Validity and Meta-logic **lemma** *foundation1*[*simp*]: $\tau \models true$

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

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

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

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

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

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

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

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

lemma *foundation6*:
$\tau \models P \Longrightarrow \tau \models \delta\ P$

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

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

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*:
$(\tau \models \delta\ x) \lor (\tau \models (x \triangleq \mathit{invalid})) \lor (\tau \models (x \triangleq \mathit{null}))$

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

lemma *foundation9′*:
$\tau \models \mathit{not}\ x \Longrightarrow \neg\ (\tau \models x)$

lemma *foundation9″*:
$\tau \models \mathit{not}\ x \Longrightarrow \tau \models \delta\ x$

lemma *foundation10*:
$\tau \models \delta\ x \Longrightarrow \tau \models \delta\ y \Longrightarrow (\tau \models (x\ \mathit{and}\ y)) = (\ (\tau \models x) \land (\tau \models y))$

lemma *foundation10′*: $(\tau \models (A\ \mathit{and}\ B)) = ((\tau \models A) \land (\tau \models B))$

lemma *foundation11*:
$\tau \models \delta\ x \Longrightarrow\ \tau \models \delta\ y \Longrightarrow (\tau \models (x\ \mathit{or}\ y)) = (\ (\tau \models x) \lor (\tau \models y))$

lemma *foundation12*:
$\tau \models \delta\ x \Longrightarrow (\tau \models (x\ \mathit{implies}\ y)) = (\ (\tau \models x) \longrightarrow (\tau \models y))$

lemma *foundation13*:$(\tau \models A \triangleq \mathit{true}) \quad = (\tau \models A)$

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

lemma *foundation15*:$(\tau \models A \triangleq \mathit{invalid}) = (\tau \models \mathit{not}(\upsilon\ A))$

lemma *foundation16*: $\tau \models (\delta\ X) = (X\ \tau \neq \mathit{bot} \land X\ \tau \neq \mathit{null})$

lemma *foundation16″*: $\neg(\tau \models (\delta\ X)) = ((\tau \models (X \triangleq \mathit{invalid})) \lor (\tau \models (X \triangleq \mathit{null})))$

lemma *foundation16′*: $(\tau \models (\delta\ X)) = (X\ \tau \neq \mathit{invalid}\ \tau \land X\ \tau \neq \mathit{null}\ \tau)$

lemma *foundation18*: $(\tau \models (\upsilon\ X)) = (X\ \tau \neq \mathit{invalid}\ \tau)$

lemma *foundation18′*: $(\tau \models (\upsilon\ X)) = (X\ \tau \neq \mathit{bot})$

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

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

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

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

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

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

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

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

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*

lemma *foundation27*: $(\tau \models (A\ and\ B)) = ((\tau \models A) \wedge (\tau \models B))$

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

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

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

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

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

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

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

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

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

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 $\quad cp\ P \equiv (\exists\, f.\ \forall\ X\ \tau.\ P\ X\ \tau = f\ (X\ \tau)\ \tau)$

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*: $\bigwedge \tau.\ cp\ P \Longrightarrow \tau \models (x \triangleq y) \Longrightarrow \tau \models (P\ x \triangleq P\ y)$

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

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

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

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

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

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

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

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

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

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

lemma *cp-id* : $\quad cp(\lambda X.\ X)$

OCL's if then else endif

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

lemma *cp-OclIf*: $((\mathit{if}\ C\ \mathit{then}\ B_1\ \mathit{else}\ B_2\ \mathit{endif})\ \tau =$
$(\mathit{if}\ (\lambda\ \text{-}.\ C\ \tau)\ \mathit{then}\ (\lambda\ \text{-}.\ B_1\ \tau)\ \mathit{else}\ (\lambda\ \text{-}.\ B_2\ \tau)\ \mathit{endif})\ \tau)$
lemma *OclIf-invalid* [*simp*]: $(\mathit{if}\ \mathit{invalid}\ \mathit{then}\ B_1\ \mathit{else}\ B_2\ \mathit{endif}) = \mathit{invalid}$

lemma *OclIf-null* [*simp*]: $(\mathit{if}\ \mathit{null}\ \mathit{then}\ B_1\ \mathit{else}\ B_2\ \mathit{endif}) = \mathit{invalid}$

lemma *OclIf-true* [*simp*]: $(\mathit{if}\ \mathit{true}\ \mathit{then}\ B_1\ \mathit{else}\ B_2\ \mathit{endif}) = B_1$

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

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

lemma *OclIf-false* [*simp*]: $(\mathit{if}\ \mathit{false}\ \mathit{then}\ B_1\ \mathit{else}\ B_2\ \mathit{endif}) = B_2$

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

lemma *OclIf-idem1*[*simp*]: $(\mathit{if}\ \ \delta\ X\ \mathit{then}\ A\ \mathit{else}\ A\ \mathit{endif}) = A$

lemma *OclIf-idem2*[*simp*]: $(\mathit{if}\ \ \upsilon\ X\ \mathit{then}\ A\ \mathit{else}\ A\ \mathit{endif}) = A$

lemma *OclNot-if*[*simp*]:
$\mathit{not}(\mathit{if}\ P\ \mathit{then}\ C\ \mathit{else}\ E\ \mathit{endif}) = (\mathit{if}\ P\ \mathit{then}\ \mathit{not}\ C\ \mathit{else}\ \mathit{not}\ E\ \mathit{endif})$

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

with term "not" we can express the notation:

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

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

Laws to Establish Definedness (δ-closure)

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

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

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 $\tau \models not\ A$

A Side-calculus for Constant Terms

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

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

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 $P\ X\ \tau = P\ X\ \tau'$

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

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

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 $const\ P \Longrightarrow const\ Q \Longrightarrow const\ R \Longrightarrow const\ S$

lemma *const-lam* : $const\ (\lambda\text{-}.\ e)$

lemma *const-true*[*simp*] : *const true*

lemma *const-false*[*simp*] : *const false*

lemma *const-null*[*simp*] : *const null*

lemma *const-invalid* [*simp*]: *const invalid*

lemma *const-bot*[*simp*] : *const bot*

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

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

lemma *const-OclAnd* :
assumes *const X*
assumes *const* X'
shows *const* (*X and* X')

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

lemma *const-OclOr* :
assumes *const X*
assumes *const* X'
shows *const* (*X or* X')

lemma *const-OclImplies* :
assumes *const X*
assumes *const* X'
shows *const* (*X implies* X')

lemma *const-StrongEq*:
assumes *const X*
assumes *const* X'
shows *const*($X \triangleq X'$)

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

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

lemma *const-OclValid2*:

assumes *const x*
shows $(\tau \models \upsilon\ x) = (\tau' \models \upsilon\ x)$

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

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* $(\bot)$

A.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.

Property Profiles for Monadic Operators

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

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

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

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

end

locale *profile-mono-schemeV* =
fixes $f :: ('\mathfrak{A},'\alpha{::}null)val \Rightarrow ('\mathfrak{A},'\beta{::}null)val$
fixes g
assumes *def-scheme*: $(f\ x) \equiv \lambda\ \tau.\ if\ (\upsilon\ x)\ \tau = true\ \tau\ then\ g\ (x\ \tau)\ else\ invalid\ \tau$
begin
lemma *strict*[*simp*,*code-unfold*]: $f\ invalid = invalid$

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

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

end

locale *profile-mono2* = *profile-mono-schemeD* +
  assumes $\bigwedge x.\ x \neq bot \Longrightarrow x \neq null \Longrightarrow g\ x \neq bot$
begin

  lemma *const*[*simp*,*code-unfold*] :
    assumes *C1* :*const X*
    shows *const*(*f X*)
end

locale *profile-mono0* = *profile-mono-schemeD* +
  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*

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

  lemma *def-valid-then-def*: $\upsilon(f\,x) = (\delta(f\,x))$
end

Property Profiles for 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*]: $const\ X \Longrightarrow const\ (d\ X)$

Property Profiles for Binary Operators

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
interpretation d_y : *profile-single* d_y

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

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

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

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

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

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

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

lemma *const*[*simp*,*code-unfold*] :
assumes *C1* :*const X* **and** *C2* : *const Y*
shows *const*(*f X Y*)
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:: $({}'\mathfrak{A},{}'b{::}null)val \Rightarrow {}'\mathfrak{A}\ Boolean$
fixes $f{::}({}'\mathfrak{A},{}'a{::}null)val \Rightarrow ({}'\mathfrak{A},{}'b{::}null)val \Rightarrow ({}'\mathfrak{A},{}'c{::}null)val$
fixes *g*
assumes d_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*]: $f\ null\ y = invalid$
end

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

locale *profile-bin1* =
fixes $f{::}({}'\mathfrak{A},{}'a{::}null)val \Rightarrow ({}'\mathfrak{A},{}'b{::}null)val \Rightarrow ({}'\mathfrak{A},{}'c{::}null)val$

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*]: $f\ x\ null = invalid$
end

sublocale *profile-bin1* < *profile-bin-scheme-defined defined*

locale *profile-bin2* =
fixes f::$('\mathfrak{A},'a::null)val \Rightarrow ('\mathfrak{A},'b::null)val \Rightarrow ('\mathfrak{A},'c::null)val$
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*

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

sublocale *profile-bin3* < *profile-bin-scheme valid valid f* $\lambda x\ y.\ _{\sqcup\sqcup}x = y_{\sqcup\sqcup}$

context *profile-bin3*
begin
lemma *idem*[*simp*,*code-unfold*]: $f\ null\ null = true$

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

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

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

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

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

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

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

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*

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

which implies elementary properties like:

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

lemma *null-non-false* $[\mathit{simp},\mathit{code\text{-}unfold}]{:}(\mathit{null} \doteq \mathit{false}) = \mathit{false}$

lemma *null-non-true* $[\mathit{simp},\mathit{code\text{-}unfold}]{:}(\mathit{null} \doteq \mathit{true}) = \mathit{false}$

lemma *false-non-null* $[\mathit{simp},\mathit{code\text{-}unfold}]{:}(\mathit{false} \doteq \mathit{null}) = \mathit{false}$

lemma *true-non-null* $[\mathit{simp},\mathit{code\text{-}unfold}]{:}(\mathit{true} \doteq \mathit{null}) = \mathit{false}$

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

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

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

lemma $(\mathit{invalid} \doteq \mathit{false}) = \mathit{invalid}$
lemma $(\mathit{invalid} \doteq \mathit{true}) = \mathit{invalid}$
lemma $(\mathit{false} \doteq \mathit{invalid}) = \mathit{invalid}$
lemma $(\mathit{true} \doteq \mathit{invalid}) = \mathit{invalid}$
lemma $((\mathit{invalid}{::}('\mathfrak{A})\mathit{Boolean}) \doteq \mathit{invalid}) = \mathit{invalid}$

Thus, the weak equality is *not* reflexive.

Test Statements on Boolean Operations.

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

Elementary computations on Boolean

Assert $\tau \models \upsilon(\mathit{true})$
Assert $\tau \models \delta(\mathit{false})$
Assert $\neg(\tau \models \delta(\mathit{null}))$
Assert $\neg(\tau \models \delta(\mathit{invalid}))$
Assert $\tau \models \upsilon((\mathit{null}{::}('\mathfrak{A})\mathit{Boolean}))$
Assert $\neg(\tau \models \upsilon(\mathit{invalid}))$

Assert $\tau \models (\mathit{true\ and\ true})$
Assert $\tau \models (\mathit{true\ and\ true} \triangleq \mathit{true})$
Assert $\tau \models ((\mathit{null\ or\ null}) \triangleq \mathit{null})$
Assert $\tau \models ((\mathit{null\ or\ null}) \doteq \mathit{null})$
Assert $\tau \models ((\mathit{true} \triangleq \mathit{false}) \triangleq \mathit{false})$
Assert $\tau \models ((\mathit{invalid} \triangleq \mathit{false}) \triangleq \mathit{false})$
Assert $\tau \models ((\mathit{invalid} \doteq \mathit{false}) \triangleq \mathit{invalid})$
Assert $\tau \models (\mathit{true} <> \mathit{false})$
Assert $\tau \models (\mathit{false} <> \mathit{true})$

A.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.

Fundamental Properties on Basic Types: Strict Equality

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

  instance
end

instantiation Void_{base} :: *null*
begin
  definition *null-Void-def*: $(\mathit{null}{::}\mathit{Void}_{base}) \equiv \mathit{Abs\text{-}Void}_{base}\ \llcorner \mathit{None} \lrcorner$

  instance
end

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

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

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

Property proof in terms of *profile-bin3*

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

Test Statements

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

A.5.4. Basic Type Integer: Operations

Fundamental Predicates on Integers: Strict Equality

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

Property proof in terms of *profile-bin3*

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

Basic Integer Constants

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

definition $OclInt0 ::('\mathfrak{A})Integer\ (\mathbf{0})$ **where** $\mathbf{0} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 0::int \rfloor\rfloor)$
definition $OclInt1 ::('\mathfrak{A})Integer\ (\mathbf{1})$ **where** $\mathbf{1} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 1::int \rfloor\rfloor)$
definition $OclInt2 ::('\mathfrak{A})Integer\ (\mathbf{2})$ **where** $\mathbf{2} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 2::int \rfloor\rfloor)$

Etc.

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 $OclAdd_{Integer} ::('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Integer$ (**infix** $+_{int}\ 40$)
$$
\begin{aligned}
\textbf{where } x +_{int} y \equiv \lambda\ \tau.\ & if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau \\
& then\ \lfloor\lfloor \lceil\lceil x\ \tau \rceil\rceil + \lceil\lceil y\ \tau \rceil\rceil \rfloor\rfloor \\
& else\ invalid\ \tau
\end{aligned}
$$
interpretation $OclAdd_{Integer}$: $profile\text{-}bin1\ op\ +_{int}\ \lambda\ x\ y.\ \lfloor\lfloor \lceil\lceil x \rceil\rceil + \lceil\lceil y \rceil\rceil \rfloor\rfloor$

definition $OclMinus_{Integer} ::('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Integer$ (**infix** $-_{int}\ 41$)
$$
\begin{aligned}
\textbf{where } x -_{int} y \equiv \lambda\ \tau.\ & if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau \\
& then\ \lfloor\lfloor \lceil\lceil x\ \tau \rceil\rceil - \lceil\lceil y\ \tau \rceil\rceil \rfloor\rfloor \\
& else\ invalid\ \tau
\end{aligned}
$$
interpretation $OclMinus_{Integer}$: $profile\text{-}bin1\ op\ -_{int}\ \lambda\ x\ y.\ \lfloor\lfloor \lceil\lceil x \rceil\rceil - \lceil\lceil y \rceil\rceil \rfloor\rfloor$

definition $OclMult_{Integer} ::('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Integer$ (**infix** $*_{int}\ 45$)
$$
\begin{aligned}
\textbf{where } x *_{int} y \equiv \lambda\ \tau.\ & if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau \\
& then\ \lfloor\lfloor \lceil\lceil x\ \tau \rceil\rceil * \lceil\lceil y\ \tau \rceil\rceil \rfloor\rfloor \\
& else\ invalid\ \tau
\end{aligned}
$$
interpretation $OclMult_{Integer}$: $profile\text{-}bin1\ op\ *_{int}\ \lambda\ x\ y.\ \lfloor\lfloor \lceil\lceil x \rceil\rceil * \lceil\lceil y \rceil\rceil \rfloor\rfloor$

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

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

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

definition $OclLess_{Integer}$ $::('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Boolean$ (**infix** $<_{int}$ 35)
where $x <_{int} y \equiv \lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$then\ \llcorner\ulcorner\ulcorner x\ \tau\urcorner\urcorner < \ulcorner\ulcorner y\ \tau\urcorner\urcorner\lrcorner$
$else\ invalid\ \tau$
interpretation $OclLess_{Integer}$: $profile\text{-}bin1\ op <_{int} \lambda\ x\ y.\ \llcorner\ulcorner\ulcorner x\urcorner\urcorner < \ulcorner\ulcorner y\urcorner\urcorner\lrcorner$

definition $OclLe_{Integer}$ $::('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Integer \Rightarrow ('\mathfrak{A})Boolean$ (**infix** $\leq_{int}$ 35)
where $x \leq_{int} y \equiv \lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$then\ \llcorner\ulcorner\ulcorner x\ \tau\urcorner\urcorner \leq \ulcorner\ulcorner y\ \tau\urcorner\urcorner\lrcorner$
$else\ invalid\ \tau$
interpretation $OclLe_{Integer}$: $profile\text{-}bin1\ op \leq_{int} \lambda\ x\ y.\ \llcorner\ulcorner\ulcorner x\urcorner\urcorner \leq \ulcorner\ulcorner y\urcorner\urcorner\lrcorner$

Basic Properties

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

Execution with Invalid or Null or Zero as Argument

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

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

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 $\tau \models not\ (\upsilon\ (\mathbf{1}\ div_{int}\ \mathbf{0}))$

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

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

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

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

A.5.5. Basic Type Real: Operations

Fundamental Predicates on Reals: Strict Equality

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

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

Property proof in terms of *profile-bin3*

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

Basic Real Constants

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

definition $\mathit{OclReal0} :: ('\mathfrak{A})\mathit{Real}\ (\mathbf{0.0})$ **where** $\mathbf{0.0} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 0{::}\mathit{real}\rfloor\rfloor)$
definition $\mathit{OclReal1} :: ('\mathfrak{A})\mathit{Real}\ (\mathbf{1.0})$ **where** $\mathbf{1.0} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 1{::}\mathit{real}\rfloor\rfloor)$
definition $\mathit{OclReal2} :: ('\mathfrak{A})\mathit{Real}\ (\mathbf{2.0})$ **where** $\mathbf{2.0} = (\lambda\ \text{-}\ .\ \lfloor\lfloor 2{::}\mathit{real}\rfloor\rfloor)$

Etc.

Arithmetical Operations

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

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

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

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

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

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 $x\ div_{real}\ y \equiv \lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$\quad then\ if\ y\ \tau \neq OclReal0\ \tau\ then\ \lfloor\lfloor \lceil\lceil x\ \tau \rceil\rceil\ /\ \lceil\lceil y\ \tau \rceil\rceil \rfloor\rfloor\ else\ invalid\ \tau$
$\quad else\ invalid\ \tau$

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

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

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

Basic Properties

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

Execution with Invalid or Null or Zero as Argument

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

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

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 $\tau \models not\ (\upsilon\ (\mathbf{1.0}\ div_{real}\ \mathbf{0.0}))$

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

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

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

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 $\neg(\tau \models (\delta\ (\mathbf{0.0} <_{real} null)))$

A.5.6. Basic Type String: Operations

Fundamental Properties on Strings: Strict Equality

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

Basic String Constants

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

definition $\mathit{OclStringa} ::('\mathfrak{A})\mathit{String}$ (a) **where** $\mathrm{a} = (\lambda\ \text{-}\ .\ {}_{\llcorner\lrcorner}{}''a''{}_{\llcorner\lrcorner})$
definition $\mathit{OclStringb} ::('\mathfrak{A})\mathit{String}$ (b) **where** $\mathrm{b} = (\lambda\ \text{-}\ .\ {}_{\llcorner\lrcorner}{}''b''{}_{\llcorner\lrcorner})$
definition $\mathit{OclStringc} ::('\mathfrak{A})\mathit{String}$ (c) **where** $\mathrm{c} = (\lambda\ \text{-}\ .\ {}_{\llcorner\lrcorner}{}''c''{}_{\llcorner\lrcorner})$

Etc.

String Operations

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

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

definition $\mathit{OclAdd}_{String} ::('\mathfrak{A})\mathit{String} \Rightarrow ('\mathfrak{A})\mathit{String} \Rightarrow ('\mathfrak{A})\mathit{String}$ (**infix** $+_{string}$ *40*)
where $x +_{string} y \equiv \lambda\ \tau.\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau \wedge (\delta\ y)\ \tau = \mathit{true}\ \tau\ \mathit{then}\ {}_{\llcorner}\mathit{concat}\ [\ulcorner\ulcorner x\ \tau\urcorner\urcorner, \ulcorner\ulcorner y\ \tau\urcorner\urcorner]_{\lrcorner}\ \mathit{else}\ \mathit{invalid}\ \tau$
interpretation OclAdd_{String} : $\mathit{profile\text{-}bin1}\ \mathit{op}\ +_{string}\ \lambda\ x\ y.\ {}_{\llcorner}\mathit{concat}\ [\ulcorner\ulcorner x\urcorner\urcorner, \ulcorner\ulcorner y\urcorner\urcorner]_{\lrcorner}$

Basic Properties **lemma** $\mathit{OclAdd}_{String}\text{-}\mathit{not\text{-}commute}$: $\exists X\ Y.\ (X +_{string} Y) \neq (Y +_{string} X)$

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*.

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

Elementary computations on String

Assert $\tau \models \mathrm{a} <> \mathrm{b}$

Assert $\tau \models$ b $<>$ a
Assert $\tau \models$ b $\doteq$ b

Assert $\tau \models \upsilon$ a
Assert $\tau \models \delta$ a
Assert $\tau \models \upsilon$ $(null::('\mathfrak{A})String)$
Assert $\tau \models (invalid \triangleq invalid)$
Assert $\tau \models (null \triangleq null)$
Assert $\tau \models$ (a $\triangleq$ a)
Assert $\neg(\tau \models$ (a $\triangleq$ b))
Assert $\neg(\tau \models (invalid \triangleq$ b))
Assert $\neg(\tau \models (null \triangleq$ b))
Assert $\neg(\tau \models (invalid \doteq (invalid::('\mathfrak{A})String)))$
Assert $\neg(\tau \models \upsilon\ (invalid \doteq (invalid::('\mathfrak{A})String)))$
Assert $\neg(\tau \models (invalid <> (invalid::('\mathfrak{A})String)))$
Assert $\neg(\tau \models \upsilon\ (invalid <> (invalid::('\mathfrak{A})String)))$
Assert $\tau \models (null \doteq (null::('\mathfrak{A})String)\ )$
Assert $\tau \models (null \doteq (null::('\mathfrak{A})String)\ )$
Assert $\tau \models$ (b $\doteq$ b)
Assert $\neg(\tau \models$ (b $<>$ b))
Assert $\neg(\tau \models$ (b $\doteq$ c))
Assert $\tau \models$ (b $<>$ c)

A.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, ...

Semantic Properties of the Type Constructor

lemma $A[simp]{:}Rep\text{-}Pair_{base}\ x \neq None \Longrightarrow Rep\text{-}Pair_{base}\ x \neq null \Longrightarrow (fst\ ^{\ulcorner\ulcorner}Rep\text{-}Pair_{base}\ x^{\urcorner\urcorner}) \neq bot$

lemma $A'[simp]{:}\ x \neq bot \Longrightarrow\ x \neq null \Longrightarrow (fst\ ^{\ulcorner\ulcorner}Rep\text{-}Pair_{base}\ x^{\urcorner\urcorner}) \neq bot$

lemma $B[simp]{:}Rep\text{-}Pair_{base}\ x \neq None \Longrightarrow Rep\text{-}Pair_{base}\ x \neq null \Longrightarrow (snd\ ^{\ulcorner\ulcorner}Rep\text{-}Pair_{base}\ x^{\urcorner\urcorner}) \neq bot$

lemma $B'[simp]{:}x \neq bot \Longrightarrow x \neq null \Longrightarrow (snd\ ^{\ulcorner\ulcorner}Rep\text{-}Pair_{base}\ x^{\urcorner\urcorner}) \neq bot$

Fundamental Properties of Strict Equality

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 τ)

Property proof in terms of *profile-bin3*

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

Standard Operations Definitions

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

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

interpretation *OclPair* : *profile-bin4*
$OclPair\ \lambda\ x\ y.\ Abs\text{-}Pair_{base}\ \llcorner\llcorner(x,\ y)\lrcorner\lrcorner$

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

interpretation *OclFirst* : *profile-mono2* $OclFirst\ \lambda x.\ \ fst\ \ulcorner\ulcorner Rep\text{-}Pair_{base}\ (x)\urcorner\urcorner$

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

interpretation *OclSecond* : *profile-mono2* $OclSecond\ \lambda x.\ \ snd\ \ulcorner\ulcorner Rep\text{-}Pair_{base}\ (x)\urcorner\urcorner$

Logical Properties

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

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

Algebraic Execution Properties

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

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

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*

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

A.5.8. Collection Type Set: Operations

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

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

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

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

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

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

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

lemma *Integer-defined* : δ *Integer* = *true*

lemma $Integer_{null}$*-defined* : δ $Integer_{null}$ = *true*

This allows the theorems:
$\tau \models \delta\, x \implies \tau \models (Integer{-}{>}includes_{Set}(x))$ $\tau \models \delta\, x \implies \tau \models Integer\ \triangleq (Integer{-}{>}including_{Set}(x))$
and
$\tau \models \upsilon\, x \implies \tau \models (Integer_{null}{-}{>}includes_{Set}(x))$ $\tau \models \upsilon\, x \implies \tau \models Integer_{null}\ \triangleq (Integer_{null}{-}{>}including_{Set}(x))$
which characterize the infiniteness of these sets by a recursive property on these sets.

Basic Properties of the Set-Type

Every element in a defined set is valid.

lemma *Set-inv-lemma*: $\tau \models (\delta\, X) \Longrightarrow \forall x \in \ulcorner\ulcorner Rep\text{-}Set_{base}\ (X\ \tau)\urcorner\urcorner.\ x \neq bot$

lemma *Set-inv-lemma'* :

assumes *x-def* : $\tau \models \delta\, X$
and *e-mem* : $e \in \ulcorner\ulcorner Rep\text{-}Set_{base}\ (X\ \tau)\urcorner\urcorner$
shows $\tau \models \upsilon\ (\lambda\text{-}.\ e)$

lemma *abs-rep-simp'* :
assumes *S-all-def* : $\tau \models \delta\, S$
shows $Abs\text{-}Set_{base}\ \llcorner\llcorner\ulcorner\ulcorner Rep\text{-}Set_{base}\ (S\ \tau)\urcorner\urcorner\lrcorner\lrcorner = S\ \tau$

lemma *S-lift'* :
assumes *S-all-def* : $(\tau :: {'\mathfrak{A}}\ st) \models \delta\, S$
shows $\exists S'.\ (\lambda a\ (\text{-}::{'\mathfrak{A}}\ st).\ a)\ `\ \ulcorner\ulcorner Rep\text{-}Set_{base}\ (S\ \tau)\urcorner\urcorner = (\lambda a\ (\text{-}::{'\mathfrak{A}}\ st).\ \llcorner a\lrcorner)\ `\ S'$

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

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

Definition: Strict Equality

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

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* $\lambda\ x\ y.\ (x::({'\mathfrak{A}},{'\alpha}::null)Set) \doteq y$

Constants on Sets: mtSet

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

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

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

lemma *mtSet-rep-set*: $\lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (\mathit{Set}\{\}\ \tau)\rceil\rceil = \{\}$

lemma [*simp*,*code-unfold*]: *const Set*{}

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

Definition: Including

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* $(\text{-}{-}{>}including_{Set}{}'(\text{-}'))$

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

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

Definition: Excluding

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* $(\text{-}{-}{>}excluding_{Set}{}'(\text{-}'))$

Definition: Includes

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* $(\text{-}{-}{>}includes_{Set}{}'(\text{-}')\ )$

Definition: Excludes

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* $(\text{-}{-}{>}excludes_{Set}{}'(\text{-}')\ )$

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: Size

definition *OclSize* $:: ({}'\mathfrak{A},{}'\alpha{::}null)Set \Rightarrow {}'\mathfrak{A}\ Integer$

where $\mathit{OclSize}\ x = (\lambda\ \tau.\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau \wedge \mathit{finite}(\ulcorner\ulcorner \mathit{Rep\text{-}Set}_{base}\ (x\ \tau)\urcorner\urcorner)$
$\quad\quad \mathit{then}\ \llcorner\llcorner \mathit{int}(\mathit{card}\ \ulcorner\ulcorner \mathit{Rep\text{-}Set}_{base}\ (x\ \tau)\urcorner\urcorner)\lrcorner\lrcorner$
$\quad\quad \mathit{else}\ \bot\ )$
notation
$\quad \mathit{OclSize} \quad (\text{-}{-}{>}\mathit{size}_{Set}\,'('\,)\ )$

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: IsEmpty

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

Definition: NotEmpty

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 $\mathit{OclNotEmpty} \quad (\text{-}{-}{>}\mathit{notEmpty}_{Set}\,'('\,)\ )$

Definition: Any

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

Definition: Forall

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

Definition: Exists

Like OclForall, OclExists is also not strict.

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

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

Definition: Iterate

definition *OclIterate* $:: [('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\beta::null)val,$
$('\mathfrak{A},'\alpha)val \Rightarrow ('\mathfrak{A},'\beta)val \Rightarrow ('\mathfrak{A},'\beta)val] \Rightarrow ('\mathfrak{A},'\beta)val$
where $OclIterate\ S\ A\ F = (\lambda\ \tau.\ if\ (\delta\ S)\ \tau = true\ \tau \wedge (\upsilon\ A)\ \tau = true\ \tau \wedge finite^{\ulcorner\ulcorner}Rep\text{-}Set_{base}\ (S\ \tau)^{\urcorner\urcorner}$
$then\ (Finite\text{-}Set.fold\ (F)\ (A)\ ((\lambda a\ \tau.\ a)\ `\ ^{\ulcorner\ulcorner}Rep\text{-}Set_{base}\ (S\ \tau)^{\urcorner\urcorner}))\tau$
$else\ \bot)$
syntax
$\text{-}OclIterate\ :: [('\mathfrak{A},'\alpha::null)\ Set,\ idt,\ idt,\ '\alpha,\ '\beta] => ('\mathfrak{A},'\gamma)val$
$(\text{-} \ {-}{>}iterate_{Set}{'}(\text{-};\text{-}{=}\text{-} \mid \text{-}')\ )$
translations
$X{-}{>}iterate_{Set}(a;\ x = A \mid P) == CONST\ OclIterate\ X\ A\ (\%a.\ (\%\ x.\ P))$

Definition: Select

definition *OclSelect* $:: [('\mathfrak{A},'\alpha::null)Set,('\mathfrak{A},'\alpha)val \Rightarrow ('\mathfrak{A})Boolean] \Rightarrow ('\mathfrak{A},'\alpha)Set$
where $OclSelect\ S\ P = (\lambda\tau.\ if\ (\delta\ S)\ \tau = true\ \tau$
$then\ if\ (\exists x \in {}^{\ulcorner\ulcorner}Rep\text{-}Set_{base}\ (S\ \tau)^{\urcorner\urcorner}.\ P(\lambda\ \text{-}.\ x)\ \tau = invalid\ \tau)$
$then\ invalid\ \tau$
$else\ Abs\text{-}Set_{base}\ {}_{\llcorner\llcorner}\{x \in {}^{\ulcorner\ulcorner}Rep\text{-}Set_{base}\ (S\ \tau)^{\urcorner\urcorner}.\ P\ (\lambda\text{-}.\ x)\ \tau \neq false\ \tau\}_{\lrcorner\lrcorner}$
$else\ invalid\ \tau)$
syntax
$\text{-}OclSelect :: [('\mathfrak{A},'\alpha::null)\ Set,id,('\mathfrak{A})Boolean] \Rightarrow {'\mathfrak{A}}\ Boolean \quad ((\text{-}){-}{>}select_{Set}{'}(\text{-}|\text{-}'))$
translations
$X{-}{>}select_{Set}(x \mid P) == CONST\ OclSelect\ X\ (\%\ x.\ P)$

Definition: Reject

definition *OclReject* $:: [('\mathfrak{A},'\alpha::null)Set,('\mathfrak{A},'\alpha)val \Rightarrow ('\mathfrak{A})Boolean] \Rightarrow ('\mathfrak{A},'\alpha::null)Set$
where $OclReject\ S\ P = OclSelect\ S\ (not\ o\ P)$
syntax
$\text{-}OclReject :: [('\mathfrak{A},'\alpha::null)\ Set,id,('\mathfrak{A})Boolean] \Rightarrow {'\mathfrak{A}}\ Boolean \quad ((\text{-}){-}{>}reject_{Set}{'}(\text{-}|\text{-}'))$
translations
$X{-}{>}reject_{Set}(x \mid P) == CONST\ OclReject\ X\ (\%\ x.\ P)$

Definition: IncludesAll

definition *OclIncludesAll* $:: [('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow {'\mathfrak{A}}\ Boolean$
where $OclIncludesAll\ x\ y = (\lambda\ \tau.\ \ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$then\ {}_{\llcorner\llcorner}{}^{\ulcorner\ulcorner}Rep\text{-}Set_{base}\ (y\ \tau)^{\urcorner\urcorner} \subseteq {}^{\ulcorner\ulcorner}Rep\text{-}Set_{base}\ (x\ \tau)^{\urcorner\urcorner}{}_{\lrcorner\lrcorner}$
$else\ \bot\ )$
notation $OclIncludesAll\ (\text{-}{-}{>}includesAll_{Set}{'}(\text{-}')\ )$

Definition: ExcludesAll

definition $\mathit{OclExcludesAll}$ :: $[('\mathfrak{A},'\alpha{::}null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow {}'\mathfrak{A}\ Boolean$
where $\mathit{OclExcludesAll}\ x\ y = (\lambda\ \tau.\ \ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$\quad then\ \lfloor\lfloor \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (y\ \tau)\rceil\rceil \cap \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (x\ \tau)\rceil\rceil = \{\}\rfloor\rfloor$
$\quad else\ \bot\ )$
notation $\mathit{OclExcludesAll}$ $(\text{-}{-}{>}excludesAll_{Set}{}'(\text{-}'))$

Definition: Union

definition $\mathit{OclUnion}$ :: $[('\mathfrak{A},'\alpha{::}null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow ('\mathfrak{A},'\alpha)\ Set$
where $\mathit{OclUnion}\ x\ y = (\lambda\ \tau.\ \ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$\quad then\ \mathit{Abs\text{-}Set}_{base}\lfloor\lfloor \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (y\ \tau)\rceil\rceil \cup \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (x\ \tau)\rceil\rceil\rfloor\rfloor$
$\quad else\ \bot\ )$
notation $\mathit{OclUnion}$ $(\text{-}{-}{>}union_{Set}{}'(\text{-}'))$

Definition: Intersection

definition $\mathit{OclIntersection}$ :: $[('\mathfrak{A},'\alpha{::}null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow ('\mathfrak{A},'\alpha)\ Set$
where $\mathit{OclIntersection}\ x\ y = (\lambda\ \tau.\ \ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$\quad then\ \mathit{Abs\text{-}Set}_{base}\lfloor\lfloor \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (y\ \tau)\rceil\rceil$
$\qquad \cap \lceil\lceil \mathit{Rep\text{-}Set}_{base}\ (x\ \tau)\rceil\rceil\rfloor\rfloor$
$\quad else\ \bot\ )$
notation $\mathit{OclIntersection}(\text{-}{-}{>}intersection_{Set}{}'(\text{-}'))$

Definition (futur operators)

consts
$\mathit{OclCount}$ :: $[('\mathfrak{A},'\alpha{::}null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow {}'\mathfrak{A}\ Integer$
OclSum :: $('\mathfrak{A},'\alpha{::}null)\ Set \Rightarrow {}'\mathfrak{A}\ Integer$

notation $\mathit{OclCount}$ $(\text{-}{-}{>}count_{Set}{}'(\text{-}'))$
notation OclSum $(\text{-}{-}{>}sum_{Set}{}'('))$

Logical Properties

OclIncluding

lemma *OclIncluding-defined-args-valid*:
$(\tau \models \delta(X{-}{>}including_{Set}(x))) = ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$

lemma *OclIncluding-valid-args-valid*:
$(\tau \models \upsilon(X{-}{>}including_{Set}(x))) = ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$

lemma *OclIncluding-defined-args-valid'*[*simp*,*code-unfold*]:
$\delta(X{-}{>}including_{Set}(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$

lemma *OclIncluding-valid-args-valid''*[*simp*,*code-unfold*]:
$\upsilon(X{-}{>}including_{Set}(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$

etc. etc.

Execution Laws with Invalid or Null or Infinite Set as Argument

OclIncluding

lemma *OclIncluding-invalid*[*simp*,*code-unfold*]:$(invalid{-}{>}including_{Set}(x)) = invalid$

lemma *OclIncluding-invalid-args*[*simp*,*code-unfold*]:$(X{-}{>}including_{Set}(invalid)) = invalid$

lemma *OclIncluding-null*[*simp*,*code-unfold*]:$(null{-}{>}including_{Set}(x)) = invalid$

OclExcluding

lemma *OclExcluding-invalid*[*simp*,*code-unfold*]:$(invalid{-}{>}excluding_{Set}(x)) = invalid$

lemma *OclExcluding-invalid-args*[*simp*,*code-unfold*]:$(X{-}{>}excluding_{Set}(invalid)) = invalid$

lemma *OclExcluding-null*[*simp*,*code-unfold*]:$(null{-}{>}excluding_{Set}(x)) = invalid$

OclIncludes

lemma *OclIncludes-invalid*[*simp*,*code-unfold*]:$(invalid{-}{>}includes_{Set}(x)) = invalid$

lemma *OclIncludes-invalid-args*[*simp*,*code-unfold*]:$(X{-}{>}includes_{Set}(invalid)) = invalid$

lemma *OclIncludes-null*[*simp*,*code-unfold*]:$(null{-}{>}includes_{Set}(x)) = invalid$

OclExcludes

lemma *OclExcludes-invalid*[*simp*,*code-unfold*]:$(invalid{-}{>}excludes_{Set}(x)) = invalid$

lemma *OclExcludes-invalid-args*[*simp*,*code-unfold*]:$(X{-}{>}excludes_{Set}(invalid)) = invalid$

lemma *OclExcludes-null*[*simp*,*code-unfold*]:$(null{-}{>}excludes_{Set}(x)) = invalid$

OclSize

lemma *OclSize-invalid*[*simp*,*code-unfold*]:$(invalid{-}{>}size_{Set}()) = invalid$

lemma *OclSize-null*[*simp*,*code-unfold*]:$(null{-}{>}size_{Set}()) = invalid$

OclIsEmpty

lemma *OclIsEmpty-invalid*[*simp*,*code-unfold*]:$(invalid{-}{>}isEmpty_{Set}()) = invalid$

lemma *OclIsEmpty-null*[*simp*,*code-unfold*]:$(null{-}{>}isEmpty_{Set}()) = true$

OclNotEmpty

lemma *OclNotEmpty-invalid*[*simp*,*code-unfold*]:$(invalid{-}{>}notEmpty_{Set}()) = invalid$

lemma *OclNotEmpty-null*[*simp*,*code-unfold*]:$(null{-}{>}notEmpty_{Set}()) = false$

OclANY

lemma *OclANY-invalid*[*simp*,*code-unfold*]:$(invalid{-}{>}any_{Set}()) = invalid$

lemma *OclANY-null*[*simp*,*code-unfold*]:$(null{-}{>}any_{Set}()) = null$

OclForall

lemma *OclForall-invalid*[*simp*,*code-unfold*]:$invalid{-}{>}forAll_{Set}(a|\ P\ a) = invalid$

lemma *OclForall-null*[*simp*,*code-unfold*]:$null{-}{>}forAll_{Set}(a \mid P\ a) = invalid$

OclExists

lemma *OclExists-invalid*[*simp*,*code-unfold*]:$invalid{-}{>}exists_{Set}(a|\ P\ a) = invalid$

lemma *OclExists-null*[*simp*,*code-unfold*]:$null{-}{>}exists_{Set}(a \mid P\ a) = invalid$

OclIterate

lemma *OclIterate-invalid*[*simp*,*code-unfold*]:$invalid{-}{>}iterate_{Set}(a;\ x = A \mid P\ a\ x) = invalid$

lemma *OclIterate-null*[*simp*,*code-unfold*]:$null{-}{>}iterate_{Set}(a;\ x = A \mid P\ a\ x) = invalid$

lemma *OclIterate-invalid-args*[*simp*,*code-unfold*]:$S{-}{>}iterate_{Set}(a;\ x = invalid \mid P\ a\ x) = invalid$

An open question is this ...

lemma $S{-}{>}iterate_{Set}(a;\ x = null \mid P\ a\ x) = invalid$

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

OclSelect

lemma *OclSelect-invalid*[*simp*,*code-unfold*]:$invalid{-}{>}select_{Set}(a \mid P\ a) = invalid$

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

OclReject

lemma *OclReject-invalid*[*simp*,*code-unfold*]:$invalid{-}{>}reject_{Set}(a \mid P\ a) = invalid$

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

General Albegraic Execution Rules

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

assumes *X-def* : $\tau \models \delta\ X$
and *x-val* : $\tau \models \upsilon\ x$
shows $finite\ \ulcorner\ulcorner Rep\text{-}Set_{base}\ (X{-}{>}including_{Set}(x)\ \tau)\urcorner\urcorner = finite\ \ulcorner\ulcorner Rep\text{-}Set_{base}\ (X\ \tau)\urcorner\urcorner$

lemma *OclIncluding-rep-set*:
assumes *S-def*: $\tau \models \delta\ S$
shows $\ulcorner\ulcorner Rep\text{-}Set_{base}\ (S{-}{>}including_{Set}(\lambda\text{-}.\ \llcorner\llcorner x\lrcorner\lrcorner)\ \tau)\urcorner\urcorner = insert\ \llcorner\llcorner x\lrcorner\lrcorner\ \ulcorner\ulcorner Rep\text{-}Set_{base}\ (S\ \tau)\urcorner\urcorner$

lemma *OclIncluding-notempty-rep-set*:
assumes *X-def*: $\tau \models \delta\ X$
and *a-val*: $\tau \models \upsilon\ a$
shows $\ulcorner\ulcorner Rep\text{-}Set_{base}\ (X{-}{>}including_{Set}(a)\ \tau)\urcorner\urcorner \neq \{\}$

lemma *OclIncluding-includes0*:
assumes $\tau \models X{-}{>}includes_{Set}(x)$
shows $X{-}{>}including_{Set}(x)\ \tau = X\ \tau$

lemma *OclIncluding-includes*:
assumes $\tau \models X{-}{>}includes_{Set}(x)$
shows $\tau \models X{-}{>}including_{Set}(x) \triangleq X$

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 $\tau \models ((S :: ('\mathfrak{A}, 'a{::}null)\ Set){-}{>}including_{Set}(i){-}{>}including_{Set}(j) \triangleq (S{-}{>}including_{Set}(j){-}{>}including_{Set}(i)))$

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

Execution Rules on Excluding

lemma *OclExcluding-finite-rep-set* :
assumes *X-def* : $\tau \models \delta\ X$
and *x-val* : $\tau \models \upsilon\ x$
shows $finite\ \ulcorner\ulcorner Rep\text{-}Set_{base}\ (X{-}{>}excluding_{Set}(x)\ \tau)\urcorner\urcorner = finite\ \ulcorner\ulcorner Rep\text{-}Set_{base}\ (X\ \tau)\urcorner\urcorner$

lemma *OclExcluding-rep-set*:
assumes *S-def*: $\tau \models \delta\ S$
shows $\ulcorner\ulcorner Rep\text{-}Set_{base}\ (S{-}{>}excluding_{Set}(\lambda\text{-}.\ \llcorner\llcorner x\lrcorner\lrcorner)\ \tau)\urcorner\urcorner = \ulcorner\ulcorner Rep\text{-}Set_{base}\ (S\ \tau)\urcorner\urcorner - \{\llcorner\llcorner x\lrcorner\lrcorner\}$

lemma *OclExcluding-excludes0*:
assumes $\tau \models X{-}{>}excludes_{Set}(x)$
shows $X{-}{>}excluding_{Set}(x)\ \tau = X\ \tau$

lemma *OclExcluding-excludes*:
assumes $\tau \models X{-}{>}excludes_{Set}(x)$
shows $\tau \models X{-}{>}excluding_{Set}(x) \triangleq X$

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

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 $\tau \models ((S :: ('\mathfrak{A}, 'a::null)\ Set){-}{>}excluding_{Set}(i){-}{>}excluding_{Set}(j) \triangleq (S{-}{>}excluding_{Set}(j){-}{>}excluding_{Set}(i)))$

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

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

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 $\tau \models ((X{-}{>}including_{Set}(x)){-}{>}excluding_{Set}(y)) \triangleq ((X{-}{>}excluding_{Set}(y)){-}{>}including_{Set}(x))$

lemma *OclExcluding-charn2*:
assumes *def-X*:$\tau \models (\delta\ X)$
and *val-x*:$\tau \models (\upsilon\ x)$
shows $\tau \models (((X{-}{>}including_{Set}(x)){-}{>}excluding_{Set}(x)) \triangleq (X{-}{>}excluding_{Set}(x)))$

theorem *OclExcluding-charn3*: $((X{-}{>}including_{Set}(x)){-}{>}excluding_{Set}(x)) = (X{-}{>}excluding_{Set}(x))$

One would like a generic theorem of the form:

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

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

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

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

Execution Rules on Includes **lemma** *OclIncludes-charn0*[*simp*]:
assumes *val-x*: $\tau \models (\upsilon\ x)$
shows $\tau \models not(Set\{\}{-}{>}includes_{Set}(x))$

lemma *OclIncludes-charn0'*[*simp*,*code-unfold*]:
$Set\{\}{-}{>}includes_{Set}(x) =$ *(if* $\upsilon\ x$ *then false else invalid endif)*

lemma *OclIncludes-charn1*:
assumes *def-X*: $\tau \models (\delta\ X)$
assumes *val-x*: $\tau \models (\upsilon\ x)$
shows $\tau \models (X{-}{>}including_{Set}(x){-}{>}includes_{Set}(x))$

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 $\tau \models (X{-}{>}including_{Set}(x){-}{>}includes_{Set}(y)) \triangleq (X{-}{>}includes_{Set}(y))$

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
$(X{-}{>}including_{Set}(x::('\mathfrak{A},'a::null)val){-}{>}includes_{Set}(y)) =$
$(if\ \ \delta\ X\ then\ if\ x \doteq y\ then\ true\ else\ X{-}{>}includes_{Set}(y)\ endif\ else\ invalid\ endif)$

schematic-lemma *OclIncludes-execute*$_{Integer}$[*simp*,*code-unfold*]: *?X*

schematic-lemma *OclIncludes-execute*$_{Boolean}$[*simp*,*code-unfold*]: *?X*

schematic-lemma *OclIncludes-execute*$_{Set}$[*simp*,*code-unfold*]: *?X*

lemma *OclIncludes-including-generic* :
assumes *OclIncludes-execute-generic* [*simp*] : $\bigwedge X\ x\ y.$
$(X{-}{>}including_{Set}(x::('\mathfrak{A},'a::null)val){-}{>}includes_{Set}(y)) =$
$(if\ \ \delta\ X\ then\ if\ x \doteq y\ then\ true\ else\ X{-}{>}includes_{Set}(y)\ endif\ else\ invalid\ endif)$
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* : $\tau \models (S){-}{>}includes_{Set}((x::('\mathfrak{A},'a::null)val))$
shows $\tau \models S{-}{>}including_{Set}((a::('\mathfrak{A},'a::null)val)){-}{>}includes_{Set}(x)$

lemmas *OclIncludes-including*$_{Integer}$ =
OclIncludes-including-generic[*OF OclIncludes-execute*$_{Integer}$ *StrictRefEq*$_{Integer}$.*def-homo*]

Execution Rules on Excludes **lemma** *OclExcludes-charn1*:
assumes *def-X*: $\tau \models (\delta\ X)$
assumes *val-x*: $\tau \models (\upsilon\ x)$
shows $\tau \models (X{-}{>}excluding_{Set}(x){-}{>}excludes_{Set}(x))$

Execution Rules on Size **lemma** [*simp*,*code-unfold*]: $Set\{\}\ {-}{>}size_{Set}() = \mathbf{0}$

lemma *OclSize-including-exec*[*simp*,*code-unfold*]:
$((X\ {-}{>}including_{Set}(x))\ {-}{>}size_{Set}()) = (if\ \ \delta\ X\ and\ \upsilon\ x\ then$
$X\ {-}{>}size_{Set}()\ +_{int}\ if\ X\ {-}{>}includes_{Set}(x)\ then\ \mathbf{0}\ else\ \mathbf{1}\ endif$
$else$
$invalid$
$endif)$

Execution Rules on IsEmpty **lemma** [*simp*,*code-unfold*]: $Set\{\}{-}{>}isEmpty_{Set}() = true$

lemma *OclIsEmpty-including* [*simp*]:
assumes *X-def*: $\tau \models \delta\ X$
and *X-finite*: *finite* $\ulcorner\ulcorner Rep\text{-}Set_{base}\ (X\ \tau)\urcorner\urcorner$
and *a-val*: $\tau \models \upsilon\ a$
shows $X{-}{>}including_{Set}(a){-}{>}isEmpty_{Set}()\ \tau = false\ \tau$

Execution Rules on NotEmpty **lemma** [*simp*,*code-unfold*]: $Set\{\}{-}{>}notEmpty_{Set}() = false$

lemma *OclNotEmpty-including* [*simp*,*code-unfold*]:
assumes *X-def*: $\tau \models \delta\ X$
  and *X-finite*: $finite\ \ulcorner\ulcorner Rep\text{-}Set_{base}\ (X\ \tau)\urcorner\urcorner$
  and *a-val*: $\tau \models \upsilon\ a$
shows $X{-}{>}including_{Set}(a){-}{>}notEmpty_{Set}()\ \tau = true\ \tau$

Execution Rules on Any **lemma** [*simp*,*code-unfold*]: $Set\{\}{-}{>}any_{Set}() = null$

lemma *OclANY-singleton-exec*[*simp*,*code-unfold*]:
  $(Set\{\}{-}{>}including_{Set}(a)){-}{>}any_{Set}() = a$

Execution Rules on Forall **lemma** *OclForall-mtSet-exec*[*simp*,*code-unfold*] :$((Set\{\}){-}{>}forAll_{Set}(z|\ P(z))) = true$

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:
$Integer_{null}{-}{>}forAll_{Set}(x|Integer_{null}{-}{>}forAll_{Set}(y|x +_{int} y \triangleq y +_{int} x))$
or even:
$Integer{-}{>}forAll_{Set}(x|Integer{-}{>}forAll_{Set}(y|x +_{int} y \doteq y +_{int} x))$
are valid OCL statements in any context τ.

theorem *OclForall-including-exec*[*simp*,*code-unfold*] :
  assumes *cp0* : *cp P*
  shows $((S{-}{>}including_{Set}(x)){-}{>}forAll_{Set}(z \mid P(z))) = (if\ \delta\ S\ and\ \upsilon\ x$
    $then\ P\ x\ and\ (S{-}{>}forAll_{Set}(z \mid P(z)))$
    $else\ invalid$
    $endif)$

Execution Rules on Exists **lemma** *OclExists-mtSet-exec*[*simp*,*code-unfold*] :
$((Set\{\}){-}{>}exists_{Set}(z \mid P(z))) = false$

lemma *OclExists-including-exec*[*simp*,*code-unfold*] :
assumes *cp*: *cp P*
shows $((S{-}{>}including_{Set}(x)){-}{>}exists_{Set}(z \mid P(z))) = (if\ \delta\ S\ and\ \upsilon\ x$
  $then\ P\ x\ or\ (S{-}{>}exists_{Set}(z \mid P(z)))$
  $else\ invalid$
  $endif)$

Execution Rules on Iterate **lemma** *OclIterate-empty*[*simp*,*code-unfold*]: $((Set\{\}){-}{>}iterate_{Set}(a; x = A \mid P\ a\ x)) = A$

In particular, this does hold for A = null.

lemma *OclIterate-including*:
assumes *S-finite*: $\tau \models \delta(S{-}{>}size_{Set}())$
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 $((S{-}{>}including_{Set}(a)){-}{>}iterate_{Set}(a; x = \quad A \mid F\ a\ x))\ \tau =$
$((S{-}{>}excluding_{Set}(a)){-}{>}iterate_{Set}(a; x = F\ a\ A \mid F\ a\ x))\ \tau$

Execution Rules on Select

lemma *OclSelect-mtSet-exec*[*simp*,*code-unfold*]: $OclSelect\ mtSet\ P = mtSet$

definition *OclSelect-body* :: - $\Rightarrow$ - $\Rightarrow$ - $\Rightarrow$ $('\mathfrak{A}, 'a\ option\ option)\ Set$
$\equiv (\lambda P\ x\ acc.\ if\ P\ x \doteq false\ then\ acc\ else\ acc{-}{>}including_{Set}(x)\ endif)$

theorem *OclSelect-including-exec*[*simp*,*code-unfold*]:
assumes *P-cp* : *cp P*
shows $OclSelect\ (X{-}{>}including_{Set}(y))\ P = OclSelect\text{-}body\ P\ y\ (OclSelect\ (X{-}{>}excluding_{Set}(y))\ P)$
(**is** - = *?select*)

Execution Rules on Reject

lemma *OclReject-mtSet-exec*[*simp*,*code-unfold*]: $OclReject\ mtSet\ P = mtSet$

lemma *OclReject-including-exec*[*simp*,*code-unfold*]:
assumes *P-cp* : *cp P*
shows $OclReject\ (X{-}{>}including_{Set}(y))\ P = OclSelect\text{-}body\ (not\ o\ P)\ y\ (OclReject\ (X{-}{>}excluding_{Set}(y))\ P)$

Execution Rules Combining Previous Operators

OclIncluding

lemma *OclIncluding-idem0* :
assumes $\tau \models \delta\ S$
and $\tau \models \upsilon\ i$
shows $\tau \models (S{-}{>}including_{Set}(i){-}{>}including_{Set}(i) \triangleq (S{-}{>}including_{Set}(i)))$

theorem *OclIncluding-idem*[*simp*,*code-unfold*]: $((S :: ('\mathfrak{A},'a{::}null)Set){-}{>}including_{Set}(i){-}{>}including_{Set}(i) = (S{-}{>}including_{Set}(i)))$

OclExcluding

lemma *OclExcluding-idem0* :
assumes $\tau \models \delta\ S$
and $\tau \models \upsilon\ i$
shows $\tau \models (S{-}{>}excluding_{Set}(i){-}{>}excluding_{Set}(i) \triangleq (S{-}{>}excluding_{Set}(i)))$

theorem *OclExcluding-idem*[*simp*,*code-unfold*]: $((S{-}{>}excluding_{Set}(i)){-}{>}excluding_{Set}(i)) = (S{-}{>}excluding_{Set}(i))$

OclIncludes

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

$X\text{−>}includes_{Set}(X\text{−>}any_{Set}()) = (\textit{if } \delta\ X \textit{ then}$
$\quad \textit{if } \delta\ (X\text{−>}size_{Set}()) \textit{ then } not(X\text{−>}isEmpty_{Set}())$
$\quad \textit{else } X\text{−>}includes_{Set}(null) \textit{ endif}$
$\textit{else invalid endif})$

OclSize

lemma [*simp*,*code-unfold*]: $\delta\ (Set\{\}\ \text{−>}size_{Set}()) = true$

lemma [*simp*,*code-unfold*]: $\delta\ ((X\ \text{−>}including_{Set}(x))\ \text{−>}size_{Set}()) = (\delta(X\text{−>}size_{Set}())\ and\ \upsilon(x))$

lemma [*simp*,*code-unfold*]: $\delta\ ((X\ \text{−>}excluding_{Set}(x))\ \text{−>}size_{Set}()) = (\delta(X\text{−>}size_{Set}())\ and\ \upsilon(x))$

lemma [*simp*]:
assumes *X-finite*: $\bigwedge\tau.\ finite\ \ulcorner\ulcorner Rep\text{-}Set_{base}\ (X\ \tau)\urcorner\urcorner$
shows $\delta\ ((X\ \text{−>}including_{Set}(x))\ \text{−>}size_{Set}()) = (\delta(X)\ and\ \upsilon(x))$

OclForall

lemma *OclForall-rep-set-false*:
assumes $\tau \models \delta\ X$
shows $(OclForall\ X\ P\ \tau = false\ \tau) = (\exists x \in \ulcorner\ulcorner Rep\text{-}Set_{base}\ (X\ \tau)\urcorner\urcorner.\ P\ (\lambda\tau.\ x)\ \tau = false\ \tau)$

lemma *OclForall-rep-set-true*:
assumes $\tau \models \delta\ X$
shows $(\tau \models OclForall\ X\ P) = (\forall x \in \ulcorner\ulcorner Rep\text{-}Set_{base}\ (X\ \tau)\urcorner\urcorner.\ \tau \models P\ (\lambda\tau.\ x))$

lemma *OclForall-includes* :
assumes *x-def* : $\tau \models \delta\ x$
and *y-def* : $\tau \models \delta\ y$
shows $(\tau \models OclForall\ x\ (OclIncludes\ y)) = (\ulcorner\ulcorner Rep\text{-}Set_{base}\ (x\ \tau)\urcorner\urcorner \subseteq \ulcorner\ulcorner Rep\text{-}Set_{base}\ (y\ \tau)\urcorner\urcorner)$

lemma *OclForall-not-includes* :
assumes *x-def* : $\tau \models \delta\ x$
and *y-def* : $\tau \models \delta\ y$
shows $(OclForall\ x\ (OclIncludes\ y)\ \tau = false\ \tau) = (\neg\ \ulcorner\ulcorner Rep\text{-}Set_{base}\ (x\ \tau)\urcorner\urcorner \subseteq \ulcorner\ulcorner Rep\text{-}Set_{base}\ (y\ \tau)\urcorner\urcorner)$

lemma *OclForall-iterate*:
assumes *S-finite*: $finite\ \ulcorner\ulcorner Rep\text{-}Set_{base}\ (S\ \tau)\urcorner\urcorner$
shows $S\text{−>}forAll_{Set}(x \mid P\ x)\ \tau = (S\text{−>}iterate_{Set}(x; acc = true \mid acc\ and\ P\ x))\ \tau$

lemma *OclForall-cong*:
assumes $\bigwedge x.\ x \in \ulcorner\ulcorner Rep\text{-}Set_{base}\ (X\ \tau)\urcorner\urcorner \Longrightarrow \tau \models P\ (\lambda\tau.\ x) \Longrightarrow \tau \models Q\ (\lambda\tau.\ x)$
assumes *P*: $\tau \models OclForall\ X\ P$
shows $\tau \models OclForall\ X\ Q$

lemma *OclForall-cong'*:
assumes $\bigwedge x.\ x \in \ulcorner\ulcorner Rep\text{-}Set_{base}\ (X\ \tau)\urcorner\urcorner \Longrightarrow \tau \models P\ (\lambda\tau.\ x) \Longrightarrow \tau \models Q\ (\lambda\tau.\ x) \Longrightarrow \tau \models R\ (\lambda\tau.\ x)$
assumes *P*: $\tau \models OclForall\ X\ P$
assumes *Q*: $\tau \models OclForall\ X\ Q$
shows $\tau \models OclForall\ X\ R$

Strict Equality

lemma *StrictRefEq*$_{Set}$*-defined* :
assumes *x-def*: $\tau \models \delta\ x$
assumes *y-def*: $\tau \models \delta\ y$
shows $((x::('\mathfrak{A},'\alpha::null)Set) \doteq y)\ \tau =$
$(x{-}{>}forAll_{Set}(z|\ y{-}{>}includes_{Set}(z))\ and\ (y{-}{>}forAll_{Set}(z|\ x{-}{>}includes_{Set}(z))))\ \tau$

lemma *StrictRefEq*$_{Set}$*-exec*[*simp*,*code-unfold*] :
$((x::('\mathfrak{A},'\alpha::null)Set) \doteq y) =$
$(if\ \delta\ x\ then\ (if\ \delta\ y$
$then\ ((x{-}{>}forAll_{Set}(z|\ y{-}{>}includes_{Set}(z))\ and\ (y{-}{>}forAll_{Set}(z|\ x{-}{>}includes_{Set}(z)))))$
$else\ if\ \upsilon\ y$
$then\ false\ (*\ x'{-}{>}includes = null\ *)$
$else\ invalid$
$endif$
$endif)$
$else\ if\ \upsilon\ x\ (*\ null = ???\ *)$
$then\ if\ \upsilon\ y\ then\ not(\delta\ y)\ else\ invalid\ endif$
$else\ invalid$
$endif$
$endif)$

lemma *StrictRefEq*$_{Set}$*-L-subst1* : $cp\ P \Longrightarrow \tau \models \upsilon\ x \Longrightarrow \tau \models \upsilon\ y \Longrightarrow \tau \models \upsilon\ P\ x \Longrightarrow \tau \models \upsilon\ P\ y \Longrightarrow$
$\tau \models (x::('\mathfrak{A},'\alpha::null)Set) \doteq y \Longrightarrow \tau \models (P\ x ::('\mathfrak{A},'\alpha::null)Set) \doteq P\ y$

lemma *OclIncluding-cong'* :
shows $\tau \models \delta\ s \Longrightarrow \tau \models \delta\ t \Longrightarrow \tau \models \upsilon\ x \Longrightarrow$
$\tau \models ((s::('\mathfrak{A},'a::null)Set) \doteq t) \Longrightarrow \tau \models (s{-}{>}including_{Set}(x) \doteq (t{-}{>}including_{Set}(x)))$

lemma *OclIncluding-cong* : $\bigwedge(s::('\mathfrak{A},'a::null)Set)\ t\ x\ y\ \tau.\ \tau \models \delta\ t \Longrightarrow \tau \models \upsilon\ y \Longrightarrow$
$\tau \models s \doteq t \Longrightarrow x = y \Longrightarrow \tau \models s{-}{>}including_{Set}(x) \doteq (t{-}{>}including_{Set}(y))$

lemma *const-StrictRefEq*$_{Set}$*-empty* : $const\ X \Longrightarrow\ const\ (X \doteq Set\{\})$

lemma *const-StrictRefEq*$_{Set}$*-including* :
$const\ a \Longrightarrow const\ S \Longrightarrow const\ X \Longrightarrow\ const\ (X \doteq S{-}{>}including_{Set}(a))$

Test Statements

Assert $(\tau \models (Set\{\lambda\text{-}.\ \llcorner\llcorner x\lrcorner\lrcorner\} \doteq Set\{\lambda\text{-}.\ \llcorner\llcorner x\lrcorner\lrcorner\}))$
Assert $(\tau \models (Set\{\lambda\text{-}.\ \llcorner x\lrcorner\} \doteq Set\{\lambda\text{-}.\ \llcorner x\lrcorner\}))$

A.5.9. Collection Type Sequence: Operations

Properties of the Sequence Type Every element in a defined sequence is valid.

lemma *Sequence-inv-lemma*: $\tau \models (\delta\ X) \Longrightarrow \forall x{\in}set\ \ulcorner\ulcorner Rep\text{-}Sequence_{base}\ (X\ \tau)\urcorner\urcorner.\ x \neq bot$

Constants: mtSequence

definition *mtSequence* $::(\prime\mathfrak{A},\prime\alpha::null)$ *Sequence* $(Sequence\{\})$
where $Sequence\{\} \equiv (\lambda\ \tau.\ Abs\text{-}Sequence_{base}\ \lfloor\lfloor[]::\prime\alpha\ list\rfloor\rfloor)$

declare *mtSequence-def* [*code-unfold*]

lemma *mtSequence-defined*[*simp*,*code-unfold*]: $\delta(Sequence\{\}) = true$

lemma *mtSequence-valid*[*simp*,*code-unfold*]: $\upsilon(Sequence\{\}) = true$

lemma *mtSequence-rep-set*: $\ulcorner\ulcorner Rep\text{-}Sequence_{base}\ (Sequence\{\}\ \tau)\urcorner\urcorner = []$

Definition: Strict Equality

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

Property proof in terms of *profile-bin3*

interpretation $StrictRefEq_{Sequence}$: *profile-bin3* $\lambda\ x\ y.\ (x::(\prime\mathfrak{A},\prime\alpha::null)Sequence) \doteq y$

Definition: including

definition *OclIncluding* $:: [(\prime\mathfrak{A},\prime\alpha::null)\ Sequence,(\prime\mathfrak{A},\prime\alpha)\ val] \Rightarrow (\prime\mathfrak{A},\prime\alpha)\ Sequence$
where $OclIncluding\ x\ y = (\lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau$
$then\ Abs\text{-}Sequence_{base}\ \lfloor\lfloor\ulcorner\ulcorner Rep\text{-}Sequence_{base}\ (x\ \tau)\urcorner\urcorner\ @\ [y\ \tau]\rfloor\rfloor$
$else\ invalid\ \tau\ )$
notation *OclIncluding* $(\text{-}{-}{>}including_{Seq}\prime(\text{-}\prime))$

interpretation *OclIncluding* :
profile-bin2 OclIncluding $\lambda x\ y.\ Abs\text{-}Sequence_{base}\lfloor\lfloor\ulcorner\ulcorner Rep\text{-}Sequence_{base}\ x\urcorner\urcorner\ @\ [y]\rfloor\rfloor$

syntax
-OclFinsequence $::$ *args* $=>$ $(\prime\mathfrak{A},\prime a::null)$ *Sequence* $(Sequence\{(\text{-})\})$
translations
$Sequence\{x, xs\} == CONST\ OclIncluding\ (Sequence\{xs\})\ x$
$Sequence\{x\} == CONST\ OclIncluding\ (Sequence\{\})\ x$

Definition: excluding

definition *OclExcluding* $:: [(\prime\mathfrak{A},\prime\alpha::null)\ Sequence,(\prime\mathfrak{A},\prime\alpha)\ val] \Rightarrow (\prime\mathfrak{A},\prime\alpha)\ Sequence$
where $OclExcluding\ x\ y = (\lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau$
$then\ Abs\text{-}Sequence_{base}\ \lfloor\lfloor filter\ (\lambda x.\ x = y\ \tau)$

$\lceil\lceil Rep\text{-}Sequence_{base}\ (x\ \tau)\rceil\rceil\rfloor\rfloor$
$else\ invalid\ \tau\ )$
notation $OclExcluding\quad (\text{-}{-}{>}excluding_{Seq}{}'(\text{-}'))$

Definition: append

identical to including

definition $OclAppend\ :: [({}'\mathfrak{A},{}'\alpha::null)\ Sequence,({}'\mathfrak{A},{}'\alpha)\ val] \Rightarrow ({}'\mathfrak{A},{}'\alpha)\ Sequence$
where $OclAppend = OclIncluding$

Definition: union

definition $OclUnion\ :: [({}'\mathfrak{A},{}'\alpha::null)\ Sequence,({}'\mathfrak{A},{}'\alpha)\ Sequence] \Rightarrow ({}'\mathfrak{A},{}'\alpha)\ Sequence$
where $OclUnion\ x\ y = (\lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$then\ Abs\text{-}Sequence_{base}\ {}_{\lfloor\lfloor}\lceil\lceil Rep\text{-}Sequence_{base}\ (x\ \tau)\rceil\rceil\ @$
$\lceil\lceil Rep\text{-}Sequence_{base}\ (y\ \tau)\rceil\rceil\rfloor\rfloor$
$else\ invalid\ \tau\ )$
notation $OclUnion\quad (\text{-}{-}{>}union_{Seq}{}'(\text{-}'))$

Definition: prepend

definition $OclPrepend\ :: [({}'\mathfrak{A},{}'\alpha::null)\ Sequence,({}'\mathfrak{A},{}'\alpha)\ val] \Rightarrow ({}'\mathfrak{A},{}'\alpha)\ Sequence$
where $OclPrepend\ x\ y = (\lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau$
$then\ Abs\text{-}Sequence_{base}\ {}_{\lfloor\lfloor}(y\ \tau)\#\lceil\lceil Rep\text{-}Sequence_{base}\ (x\ \tau)\rceil\rceil\rfloor\rfloor$
$else\ invalid\ \tau\ )$
notation $OclPrepend\quad (\text{-}{-}{>}prepend_{Seq}{}'(\text{-}'))$

interpretation $OclPrepend{:}profile\text{-}bin2\ OclPrepend\ \lambda x\ y.\ Abs\text{-}Sequence_{base\lfloor\lfloor}y\#\lceil\lceil Rep\text{-}Sequence_{base}\ x\rceil\rceil\rfloor\rfloor$

Definition: subSequence

Definition: at

definition $OclAt\ :: [({}'\mathfrak{A},{}'\alpha::null)\ Sequence,({}'\mathfrak{A})\ Integer] \Rightarrow ({}'\mathfrak{A},{}'\alpha)\ val$
where $OclAt\ x\ y = (\lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$then\ if\ \ 1 \leq \lceil\lceil y\ \tau\rceil\rceil \wedge\ \lceil\lceil y\ \tau\rceil\rceil \leq length\lceil\lceil Rep\text{-}Sequence_{base}\ (x\ \tau)\rceil\rceil$
$then\ \lceil\lceil Rep\text{-}Sequence_{base}\ (x\ \tau)\rceil\rceil\ !\ (nat\ \lceil\lceil y\ \tau\rceil\rceil + 1)$
$else\ invalid\ \tau$
$else\ invalid\ \tau\ )$
notation $OclAt\ (\text{-}{-}{>}at_{Seq}{}'(\text{-}'))$

Definition: first

definition $OclFirst\ :: [({}'\mathfrak{A},{}'\alpha::null)\ Sequence] \Rightarrow ({}'\mathfrak{A},{}'\alpha)\ val$
where $OclFirst\ x = (\lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau$
$then\ hd\ \lceil\lceil Rep\text{-}Sequence_{base}\ (x\ \tau)\rceil\rceil$
$else\ invalid\ \tau\ )$
notation $OclFirst\quad (\text{-}{-}{>}first_{Seq}{}'(\text{-}'))$

Definition: last

definition $OclLast\ :: [({}'\mathfrak{A},{}'\alpha::null)\ Sequence] \Rightarrow ({}'\mathfrak{A},{}'\alpha)\ val$

where $OclLast\ x = (\lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau$
$then\ last\ \lceil\lceil Rep\text{-}Sequence_{base}\ (x\ \tau)\rceil\rceil$
$else\ invalid\ \tau\ )$
notation $OclLast\ \ (\text{-}{-}{>}last_{Seq}{'}(\text{-}'))$

Definition: asSet

instantiation $Sequence_{base}$ :: $(equal)equal$
begin
 definition $HOL.equal\ k\ l \longleftrightarrow\ (k{::}('a{::}equal)Sequence_{base}) =\ l$
 instance
end

lemma $equal\text{-}Sequence_{base}\text{-}code$ [$code$]:
 $HOL.equal\ k\ (l{::}('a{::}\{equal,null\})Sequence_{base}) \longleftrightarrow Rep\text{-}Sequence_{base}\ k = Rep\text{-}Sequence_{base}\ l$

Test Statements

Assert $(\tau \models (Sequence\{\} \doteq Sequence\{\}))$
Assert $\tau \models (Sequence\{\mathbf{1},invalid,\mathbf{2}\} \triangleq invalid)$

A.5.10. Miscellaneous Stuff

Properties on Collection Types: Strict Equality

The structure of this chapter roughly follows the structure of Chapter 10 of the OCL standard [20], which introduces the OCL Library.

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.

Test Statements

lemma *syntax-test*: $Set\{\mathbf{2},\mathbf{1}\} = (Set\{\}{-}{>}including_{Set}(\mathbf{1}){-}{>}including_{Set}(\mathbf{2}))$

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

lemma *semantic-test2*:
assumes $H{:}(Set\{\mathbf{2}\} \doteq null) = (false{::}({}'\mathfrak{A})Boolean)$
shows $(\tau{::}({}'\mathfrak{A})st) \models (Set\{Set\{\mathbf{2}\},null\}{-}{>}includes_{Set}(null))$

lemma *short-cut*$'$[*simp*,*code-unfold*]: $(\mathbf{8} \doteq \mathbf{6}) = false$

lemma *short-cut*$''$[*simp*,*code-unfold*]: $(\mathbf{2} \doteq \mathbf{1}) = false$
lemma *short-cut*$'''$[*simp*,*code-unfold*]: $(\mathbf{1} \doteq \mathbf{2}) = false$

Elementary computations on Sets.

declare *OclSelect-body-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 $\tau \models (Set\{\mathbf{2},\mathbf{1}\}{-}{>}includes_{Set}(\mathbf{1}))$
Assert $\neg\ (\tau \models (Set\{\mathbf{2}\}{-}{>}includes_{Set}(\mathbf{1})))$
Assert $\neg\ (\tau \models (Set\{\mathbf{2},\mathbf{1}\}{-}{>}includes_{Set}(null)))$
Assert $\tau \models (Set\{\mathbf{2},null\}{-}{>}includes_{Set}(null))$
Assert $\tau \models (Set\{null,\mathbf{2}\}{-}{>}includes_{Set}(null))$

Assert $\tau \models ((Set\{\}){-}{>}forAll_{Set}(z \mid \mathbf{0} <_{int} z))$

Assert $\tau \models ((Set\{\mathbf{2},\mathbf{1}\}){-}{>}forAll_{Set}(z \mid \mathbf{0} <_{int} z))$
Assert $\tau \models (\mathbf{0} <_{int} \mathbf{2})\ and\ (\mathbf{0} <_{int} \mathbf{1})$
Assert $\neg\ (\tau \models ((Set\{\mathbf{2},\mathbf{1}\}){-}{>}exists_{Set}(z \mid z <_{int} \mathbf{0}\ )))$
Assert $\neg\ (\tau \models (\delta(Set\{\mathbf{2},null\}){-}{>}forAll_{Set}(z \mid \mathbf{0} <_{int} z)))$
Assert $\neg\ (\tau \models ((Set\{\mathbf{2},null\}){-}{>}forAll_{Set}(z \mid \mathbf{0} <_{int} z)))$
Assert $\tau \models ((Set\{\mathbf{2},null\}){-}{>}exists_{Set}(z \mid \mathbf{0} <_{int} z))$

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 $\tau \models (Set\{null\}{-}{>}select_{Set}(x \mid not\ x) \doteq Set\{null\})$
Assert $\tau \models (Set\{null\}{-}{>}reject_{Set}(x \mid not\ x) \doteq Set\{null\})$

lemma $const\ (Set\{Set\{\mathbf{2},null\},\ invalid\})$

A.6. Formalization III: UML/OCL constructs: State Operations and Objects

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

A.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.

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 $StrictRefEq_{Object}\ x\ y$
$\equiv \lambda\ \tau.\ if\ (\upsilon\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau$
$then\ if\ x\ \tau = null \vee y\ \tau = null$
$then\ \llcorner\llcorner x\ \tau = null \wedge y\ \tau = null\lrcorner\lrcorner$
$else\ \llcorner\llcorner(oid\text{-}of\ (x\ \tau)) = (oid\text{-}of\ (y\ \tau))\lrcorner\lrcorner$
$else\ invalid\ \tau$

Strictness and context passing **lemma** $StrictRefEq_{Object}\text{-}strict1[simp,code\text{-}unfold]$:
$(StrictRefEq_{Object}\ x\ invalid) = invalid$

lemma $StrictRefEq_{Object}\text{-}strict2[simp,code\text{-}unfold]$:
$(StrictRefEq_{Object}\ invalid\ x) = invalid$

lemma $cp\text{-}StrictRefEq_{Object}$:
$(StrictRefEq_{Object}\ x\ y\ \tau) = (StrictRefEq_{Object}\ (\lambda\text{-}.\ x\ \tau)\ (\lambda\text{-}.\ y\ \tau))\ \tau$

Logic and Algebraic Layer on Object

Validity and Definedness Properties We derive the usual laws on definedness for (generic) object equality:

lemma $StrictRefEq_{Object}\text{-}defargs$:
$\tau \models (StrictRefEq_{Object}\ x\ (y{::}({'}\mathfrak{A},{'}a{::}\{null,object\})val)) \Longrightarrow (\tau \models (\upsilon\ x)) \wedge (\tau \models (\upsilon\ y))$

lemma *defined-StrictRefEq*$_{Object}$*-I*:
assumes *val-x* : $\tau \models \upsilon\ x$
assumes *val-x* : $\tau \models \upsilon\ y$
shows $\tau \models \delta\ (StrictRefEq_{Object}\ x\ y)$

lemma *StrictRefEq*$_{Object}$*-def-homo* :
$\delta(StrictRefEq_{Object}\ x\ (y{::}('\mathfrak{A},'a{::}\{null,object\})val)) = ((\upsilon\ x)\ and\ (\upsilon\ y))$

Symmetry **lemma** *StrictRefEq*$_{Object}$*-sym* :
assumes *x-val* : $\tau \models \upsilon\ x$
shows $\tau \models StrictRefEq_{Object}\ x\ x$

Behavior vs StrongEq It remains to clarify the role of the state invariant $\mathrm{inv}_\sigma(\sigma)$ mentioned above that states the condition that there is a "one-to-one" correspondence between object representations and oid's: $\forall oid \in \mathrm{dom}\,\sigma.\ oid = \mathrm{OidOf}\ulcorner\sigma(oid)\urcorner$. This condition is also mentioned in [20, Annex A] and goes back to Richters [22]; 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 $WFF\ \tau = ((\forall\ x \in ran(heap(fst\ \tau)).\ \ulcorner heap(fst\ \tau)\ (oid\text{-}of\ x)\urcorner = x) \wedge$
$(\forall\ x \in ran(heap(snd\ \tau)).\ \ulcorner heap(snd\ \tau)\ (oid\text{-}of\ x)\urcorner = x))$

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 [4, 6], 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 $(\tau \models (StrictRefEq_{Object}\ x\ y)) = (\tau \models (x \triangleq y))$

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 $(\tau \models (StrictRefEq_{Object}\ x\ y)) = (\tau \models (x \triangleq y))$

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.

A.6.2. Operations on Object

Initial States (for testing and code generation)

definition $\tau_0 :: ('\mathfrak{A})st$
where $\tau_0 \equiv ((\!|heap=Map.empty, assocs = Map.empty|\!),$
$(\!|heap=Map.empty, assocs = Map.empty|\!))$

OclAllInstances

To denote OCL types occurring in OCL expressions syntactically—as, for example, as "argument" of `oclAllInstances()`—we use the inverses of the injection functions into the object universes; we show that this is a sufficient "characterization."

definition *OclAllInstances-generic* :: $(('\mathfrak{A}::object)\ st \Rightarrow '\mathfrak{A}\ state) \Rightarrow ('\mathfrak{A}::object \rightharpoonup '\alpha) \Rightarrow$
$('\mathfrak{A}, '\alpha\ option\ option)\ Set$
where *OclAllInstances-generic fst-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*: $\tau \models \delta\ (OclAllInstances\text{-}generic\ pre\text{-}post\ H)$

lemma *OclAllInstances-generic-init-empty*:
assumes [*simp*]: $\bigwedge x.\ pre\text{-}post\ (x, x) = x$
shows $\tau_0 \models OclAllInstances\text{-}generic\ pre\text{-}post\ H \triangleq Set\{\}$

lemma *represented-generic-objects-nonnull*:
assumes *A*: $\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ (H::('\mathfrak{A}::object \rightharpoonup '\alpha))) -{>}includes_{Set}(x))$
shows $\tau \models not(x \triangleq null)$

lemma *represented-generic-objects-defined*:
assumes *A*: $\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ (H::('\mathfrak{A}::object \rightharpoonup '\alpha))) -{>}includes_{Set}(x))$
shows $\tau \models \delta\ (OclAllInstances\text{-}generic\ pre\text{-}post\ H) \wedge \tau \models \delta\ x$

One way to establish the actual presence of an object representation in a state is:

lemma *represented-generic-objects-in-state*:
assumes *A*: $\tau \models (OclAllInstances\text{-}generic\ pre\text{-}post\ H){-}{>}includes_{Set}(x)$
shows $x\ \tau \in (Some\ o\ H)\ `\ ran\ (heap(pre\text{-}post\ \tau))$

lemma *state-update-vs-allInstances-generic-empty*:
assumes [*simp*]: $\bigwedge a.\ \mathit{pre\text{-}post}\ (\mathit{mk}\ a) = a$
shows $(\mathit{mk}\ (\!|\mathit{heap}{=}\mathit{empty},\ \mathit{assocs}{=}A|\!)) \models \mathit{OclAllInstances\text{-}generic}\ \mathit{pre\text{-}post}\ \mathit{Type} \doteq \mathit{Set}\{\}$

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*]: $\bigwedge a.\ \mathit{pre\text{-}post}\ (\mathit{mk}\ a) = a$
assumes $\bigwedge x.\ \sigma'\ \mathit{oid} = \mathit{Some}\ x \Longrightarrow x = \mathit{Object}$
and $\mathit{Type}\ \mathit{Object} \neq \mathit{None}$
shows $(\mathit{OclAllInstances\text{-}generic}\ \mathit{pre\text{-}post}\ \mathit{Type})$
$(\mathit{mk}\ (\!|\mathit{heap}{=}\sigma'(\mathit{oid}{\mapsto}\mathit{Object}),\ \mathit{assocs}{=}A|\!))$
$=$
$((\mathit{OclAllInstances\text{-}generic}\ \mathit{pre\text{-}post}\ \mathit{Type}){-}{>}\mathit{including}_{\mathit{Set}}(\lambda\ \text{-}.\ \lfloor\lfloor \mathit{drop}\ (\mathit{Type}\ \mathit{Object})\rfloor\rfloor))$
$(\mathit{mk}\ (\!|\mathit{heap}{=}\sigma',\mathit{assocs}{=}A|\!))$

lemma *state-update-vs-allInstances-generic-including*:
assumes [*simp*]: $\bigwedge a.\ \mathit{pre\text{-}post}\ (\mathit{mk}\ a) = a$
assumes $\bigwedge x.\ \sigma'\ \mathit{oid} = \mathit{Some}\ x \Longrightarrow x = \mathit{Object}$
and $\mathit{Type}\ \mathit{Object} \neq \mathit{None}$
shows $(\mathit{OclAllInstances\text{-}generic}\ \mathit{pre\text{-}post}\ \mathit{Type})$
$(\mathit{mk}\ (\!|\mathit{heap}{=}\sigma'(\mathit{oid}{\mapsto}\mathit{Object}),\ \mathit{assocs}{=}A|\!))$
$=$
$((\lambda\text{-}.\ (\mathit{OclAllInstances\text{-}generic}\ \mathit{pre\text{-}post}\ \mathit{Type})$
$(\mathit{mk}\ (\!|\mathit{heap}{=}\sigma',\ \mathit{assocs}{=}A|\!))){-}{>}\mathit{including}_{\mathit{Set}}(\lambda\ \text{-}.\ \lfloor\lfloor \mathit{drop}\ (\mathit{Type}\ \mathit{Object})\rfloor\rfloor))$
$(\mathit{mk}\ (\!|\mathit{heap}{=}\sigma'(\mathit{oid}{\mapsto}\mathit{Object}),\ \mathit{assocs}{=}A|\!))$

lemma *state-update-vs-allInstances-generic-noincluding'*:
assumes [*simp*]: $\bigwedge a.\ \mathit{pre\text{-}post}\ (\mathit{mk}\ a) = a$
assumes $\bigwedge x.\ \sigma'\ \mathit{oid} = \mathit{Some}\ x \Longrightarrow x = \mathit{Object}$
and $\mathit{Type}\ \mathit{Object} = \mathit{None}$
shows $(\mathit{OclAllInstances\text{-}generic}\ \mathit{pre\text{-}post}\ \mathit{Type})$
$(\mathit{mk}\ (\!|\mathit{heap}{=}\sigma'(\mathit{oid}{\mapsto}\mathit{Object}),\ \mathit{assocs}{=}A|\!))$
$=$
$(\mathit{OclAllInstances\text{-}generic}\ \mathit{pre\text{-}post}\ \mathit{Type})$
$(\mathit{mk}\ (\!|\mathit{heap}{=}\sigma',\ \mathit{assocs}{=}A|\!))$

theorem *state-update-vs-allInstances-generic-ntc*:
assumes [*simp*]: $\bigwedge a.\ \mathit{pre\text{-}post}\ (\mathit{mk}\ a) = a$
assumes *oid-def*: $\mathit{oid}{\notin}\mathit{dom}\ \sigma'$
and *non-type-conform*: $\mathit{Type}\ \mathit{Object} = \mathit{None}$
and *cp-ctxt*: $\mathit{cp}\ P$
and *const-ctxt*: $\bigwedge X.\ \mathit{const}\ X \Longrightarrow \mathit{const}\ (P\ X)$
shows $(\mathit{mk}\ (\!|\mathit{heap}{=}\sigma'(\mathit{oid}{\mapsto}\mathit{Object}),\mathit{assocs}{=}A|\!) \models P\ (\mathit{OclAllInstances\text{-}generic}\ \mathit{pre\text{-}post}\ \mathit{Type})) =$
$(\mathit{mk}\ (\!|\mathit{heap}{=}\sigma',\ \mathit{assocs}{=}A|\!) \qquad \models P\ (\mathit{OclAllInstances\text{-}generic}\ \mathit{pre\text{-}post}\ \mathit{Type}))$
(**is** $(?\tau \models P\ ?\varphi) = (?\tau' \models P\ ?\varphi)$)

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 $(mk\ (\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!) \models P\ (OclAllInstances\text{-}generic\ pre\text{-}post\ Type)) =$
$(mk\ (\!|heap{=}\sigma', assocs{=}A|\!) \models P\ ((OclAllInstances\text{-}generic\ pre\text{-}post\ Type)$
$\quad {-}{>}including_{Set}(\lambda\ \text{-}.\ \llcorner(Type\ Object)\lrcorner)))$
(**is** $(?\tau \models P\ ?\varphi) = (?\tau' \models P\ ?\varphi')$)

declare *OclAllInstances-generic-def* [*simp*]

OclAllInstances (@post) **definition** *OclAllInstances-at-post* :: $('\mathfrak{A} :: object \rightharpoonup '\alpha) \Rightarrow ('\mathfrak{A}, '\alpha\ option\ option)\ Set$
(- *.allInstances*$'$($'$))
where *OclAllInstances-at-post* = *OclAllInstances-generic snd*

lemma *OclAllInstances-at-post-defined*: $\tau \models \delta\ (H\ .allInstances())$

lemma $\tau_0 \models H\ .allInstances() \triangleq Set\{\}$

lemma *represented-at-post-objects-nonnull*:
assumes A: $\tau \models (((H{::}('\mathfrak{A}{::}object \rightharpoonup '\alpha)).allInstances())\ {-}{>}includes_{Set}(x))$
shows $\tau \models not(x \triangleq null)$

lemma *represented-at-post-objects-defined*:
assumes A: $\tau \models (((H{::}('\mathfrak{A}{::}object \rightharpoonup '\alpha)).allInstances())\ {-}{>}includes_{Set}(x))$
shows $\tau \models \delta\ (H\ .allInstances()) \wedge \tau \models \delta\ x$

One way to establish the actual presence of an object representation in a state is:

lemma
assumes A: $\tau \models H\ .allInstances(){-}{>}includes_{Set}(x)$
shows $x\ \tau \in (Some\ o\ H)\ `\ ran\ (heap(snd\ \tau))$

lemma *state-update-vs-allInstances-at-post-empty*:
shows $(\sigma, (\!|heap{=}empty, assocs{=}A|\!)) \models Type\ .allInstances() \doteq Set\{\}$

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 $(Type\ .allInstances())$
$(\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!))$
$=$

$((Type\ .allInstances()) {-}{>} including_{Set}(\lambda\ \text{-}.\ \lfloor\lfloor drop\ (Type\ Object)\ \rfloor\rfloor))$
$(\sigma, (\!|heap{=}\sigma',assocs{=}A|\!))$

lemma *state-update-vs-allInstances-at-post-including*:
assumes $\bigwedge x.\ \sigma'\ oid = Some\ x \Longrightarrow x = Object$
and $Type\ Object \neq None$
shows $(Type\ .allInstances())$
$(\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!))$
$=$
$((\lambda\text{-}.\ (Type\ .allInstances())$
$(\sigma, (\!|heap{=}\sigma', assocs{=}A|\!))) {-}{>} including_{Set}(\lambda\ \text{-}.\ \lfloor\lfloor drop\ (Type\ Object)\ \rfloor\rfloor))$
$(\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!))$

lemma *state-update-vs-allInstances-at-post-noincluding′*:
assumes $\bigwedge x.\ \sigma'\ oid = Some\ x \Longrightarrow x = Object$
and $Type\ Object = None$
shows $(Type\ .allInstances())$
$(\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!))$
$=$
$(Type\ .allInstances())$
$(\sigma, (\!|heap{=}\sigma', assocs{=}A|\!))$

theorem *state-update-vs-allInstances-at-post-ntc*:
assumes *oid-def*: $oid{\notin}dom\ \sigma'$
and *non-type-conform*: $Type\ Object = None$
and *cp-ctxt*: $cp\ P$
and *const-ctxt*: $\bigwedge X.\ const\ X \Longrightarrow const\ (P\ X)$
shows $((\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object),assocs{=}A|\!)) \models (P(Type\ .allInstances()))) =$
$((\sigma, (\!|heap{=}\sigma', assocs{=}A|\!)) \qquad \models (P(Type\ .allInstances())))$

theorem *state-update-vs-allInstances-at-post-tc*:
assumes *oid-def*: $oid{\notin}dom\ \sigma'$
and *type-conform*: $Type\ Object \neq None$
and *cp-ctxt*: $cp\ P$
and *const-ctxt*: $\bigwedge X.\ const\ X \Longrightarrow const\ (P\ X)$
shows $((\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object),assocs{=}A|\!)) \models (P(Type\ .allInstances()))) =$
$((\sigma, (\!|heap{=}\sigma', assocs{=}A|\!)) \qquad \models (P((Type\ .allInstances())$
${-}{>}including_{Set}(\lambda\ \text{-}.\ \lfloor(Type\ Object)\rfloor))))$

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*: $\tau \models \delta\ (H\ .allInstances@pre())$

lemma $\tau_0 \models H\ .allInstances@pre() \triangleq Set\{\}$

lemma *represented-at-pre-objects-nonnull*:
assumes A: $\tau \models (((H::('\mathfrak{A}::object \rightharpoonup '\alpha)).allInstances@pre()) \; {-}{>}includes_{Set}(x))$
shows $\tau \models not(x \triangleq null)$

lemma *represented-at-pre-objects-defined*:
assumes A: $\tau \models (((H::('\mathfrak{A}::object \rightharpoonup '\alpha)).allInstances@pre()) \; {-}{>}includes_{Set}(x))$
shows $\tau \models \delta \; (H \; .allInstances@pre()) \wedge \tau \models \delta \; x$

One way to establish the actual presence of an object representation in a state is:

lemma
assumes A: $\tau \models H \; .allInstances@pre(){-}{>}includes_{Set}(x)$
shows $x \; \tau \in (Some \; o \; H) \; ` \; ran \; (heap(fst \; \tau))$

lemma *state-update-vs-allInstances-at-pre-empty*:
shows $((\!|heap{=}empty, assocs{=}A|\!), \sigma) \models Type \; .allInstances@pre() \doteq Set\{\}$

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 $(Type \; .allInstances@pre())$
$((\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!), \sigma)$
$=$
$((Type \; .allInstances@pre()){-}{>}including_{Set}(\lambda \; \text{-}. \; \lfloor\lfloor drop \; (Type \; Object) \rfloor\rfloor))$
$((\!|heap{=}\sigma',assocs{=}A|\!), \sigma)$

lemma *state-update-vs-allInstances-at-pre-including*:
assumes $\bigwedge x. \; \sigma' \; oid = Some \; x \Longrightarrow x = Object$
and $Type \; Object \neq None$
shows $(Type \; .allInstances@pre())$
$((\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!), \sigma)$
$=$
$((\lambda\text{-}. \; (Type \; .allInstances@pre())$
$((\!|heap{=}\sigma', assocs{=}A|\!), \sigma)){-}{>}including_{Set}(\lambda \; \text{-}. \; \lfloor\lfloor drop \; (Type \; Object) \rfloor\rfloor))$
$((\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!), \sigma)$

lemma *state-update-vs-allInstances-at-pre-noincluding'*:
assumes $\bigwedge x. \; \sigma' \; oid = Some \; x \Longrightarrow x = Object$
and $Type \; Object = None$
shows $(Type \; .allInstances@pre())$
$((\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!), \sigma)$
$=$
$(Type \; .allInstances@pre())$

$((\!|heap{=}\sigma', assocs{=}A|\!), \sigma)$

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 $(((\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!), \sigma) \models (P(Type\ .allInstances@pre()))) =$
$(((\!|heap{=}\sigma', assocs{=}A|\!), \sigma) \qquad \models (P(Type\ .allInstances@pre())))$

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 $(((\!|heap{=}\sigma'(oid{\mapsto}Object), assocs{=}A|\!), \sigma) \models (P(Type\ .allInstances@pre()))) =$
$(((\!|heap{=}\sigma', assocs{=}A|\!), \sigma) \qquad \models (P((Type\ .allInstances@pre())$
$\quad -{>}including_{Set}(\lambda\ \text{-}.\ \llcorner(Type\ Object)\lrcorner))))$

@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*: $\bigwedge x.\ x \in ran\ (heap(fst\ \tau)) \vee x \in ran\ (heap(snd\ \tau)) \Longrightarrow$
$oid\text{-}of\ (H\ x) = oid\text{-}of\ x$
and *xy-together*: $\tau \models ((H\ .allInstances()-{>}includes_{Set}(x)\ and\ H\ .allInstances()-{>}includes_{Set}(y))\ or$
$(H\ .allInstances@pre()-{>}includes_{Set}(x)\ and\ H\ .allInstances@pre()-{>}includes_{Set}(y)))$

shows $(\tau \models (StrictRefEq_{Object}\ x\ y)) = (\tau \models (x \triangleq y))$

OclIsNew, OclIsDeleted, OclIsMaintained, OclIsAbsent

definition *OclIsNew*:: $('\mathfrak{A}, '\alpha::\{null,object\})val \Rightarrow ('\mathfrak{A})Boolean \quad ((\text{-}).oclIsNew'('))$
where $X\ .oclIsNew() \equiv (\lambda\tau\ .\ if\ (\delta\ X)\ \tau = true\ \tau$
$then\ \llcorner\llcorner oid\text{-}of\ (X\ \tau) \notin dom(heap(fst\ \tau)) \wedge$
$oid\text{-}of\ (X\ \tau) \in dom(heap(snd\ \tau))\lrcorner\lrcorner$
$else\ invalid\ \tau)$

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 $X\ .oclIsDeleted() \equiv (\lambda\tau\ .\ if\ (\delta\ X)\ \tau = true\ \tau$
$then\ \llcorner\llcorner oid\text{-}of\ (X\ \tau) \in dom(heap(fst\ \tau)) \wedge$
$oid\text{-}of\ (X\ \tau) \notin dom(heap(snd\ \tau))\lrcorner\lrcorner$
$else\ invalid\ \tau)$

definition *OclIsMaintained*:: $('\mathfrak{A}, '\alpha::\{null,object\})val \Rightarrow ('\mathfrak{A})Boolean((\text{-}).oclIsMaintained'('))$
where $X\ .oclIsMaintained() \equiv (\lambda\tau\ .\ if\ (\delta\ X)\ \tau = true\ \tau$
$then\ \llcorner\llcorner oid\text{-}of\ (X\ \tau) \in dom(heap(fst\ \tau)) \wedge$
$oid\text{-}of\ (X\ \tau) \in dom(heap(snd\ \tau))\lrcorner\lrcorner$

$\quad\quad$ *else invalid* τ)

definition $\mathit{OclIsAbsent}$:: $('\mathfrak{A}, '\alpha::\{null,object\})val \Rightarrow ('\mathfrak{A})Boolean \quad ((\text{-}).oclIsAbsent'('))$
where $X\ .oclIsAbsent() \equiv (\lambda\tau\ .\ if\ (\delta\ X)\ \tau = true\ \tau$
$\quad then\ {}_{\sqcup\!\sqcup}oid\text{-}of\ (X\ \tau) \notin dom(heap(fst\ \tau)) \wedge$
$\quad\quad oid\text{-}of\ (X\ \tau) \notin dom(heap(snd\ \tau))_{\sqcup\!\sqcup}$
$\quad else\ invalid\ \tau)$

lemma $\mathit{state\text{-}split}$: $\tau \models \delta\ X \Longrightarrow$
$\quad \tau \models (X\ .oclIsNew()) \vee \tau \models (X\ .oclIsDeleted()) \vee$
$\quad \tau \models (X\ .oclIsMaintained()) \vee \tau \models (X\ .oclIsAbsent())$

lemma $\mathit{notNew\text{-}vs\text{-}others}$: $\tau \models \delta\ X \Longrightarrow$
$\quad (\neg\ \tau \models (X\ .oclIsNew())) = (\tau \models (X\ .oclIsDeleted()) \vee$
$\quad \tau \models (X\ .oclIsMaintained()) \vee \tau \models (X\ .oclIsAbsent()))$

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 $\mathit{OclIsModifiedOnly}$ $::('\mathfrak{A}::object,'\alpha::\{null,object\})Set \Rightarrow '\mathfrak{A}\ Boolean$
$\quad (\text{-}{-}{>}oclIsModifiedOnly'('))$
where $X{-}{>}oclIsModifiedOnly() \equiv (\lambda(\sigma,\sigma').$
$\quad let\ X' = (oid\text{-}of\ `\ {}^{\ulcorner\ulcorner}Rep\text{-}Set_{base}(X(\sigma,\sigma'))^{\urcorner\urcorner});$
$\quad\quad S = ((dom\ (heap\ \sigma) \cap dom\ (heap\ \sigma')) - X')$
$\quad in\ if\ (\delta\ X)\ (\sigma,\sigma') = true\ (\sigma,\sigma') \wedge (\forall x \in {}^{\ulcorner\ulcorner}Rep\text{-}Set_{base}(X(\sigma,\sigma'))^{\urcorner\urcorner}.\ x \neq null)$
$\quad\quad then\ {}_{\sqcup\!\sqcup}\forall\ x \in S.\ (heap\ \sigma)\ x = (heap\ \sigma')\ x_{\sqcup\!\sqcup}$
$\quad\quad else\ invalid\ (\sigma,\sigma'))$

Execution with Invalid or Null or Null Element as Argument **lemma** $invalid{-}{>}oclIsModifiedOnly() = invalid$

lemma $null{-}{>}oclIsModifiedOnly() = invalid$

lemma
assumes $X\text{-}null$: $\tau \models X{-}{>}includes_{Set}(null)$
shows $\tau \models X{-}{>}oclIsModifiedOnly() \triangleq invalid$

Context Passing **lemma** $\mathit{cp\text{-}OclIsModifiedOnly}$: $X{-}{>}oclIsModifiedOnly()\ \tau = (\lambda\text{-}.\ X\ \tau){-}{>}oclIsModifiedOnly()\ \tau$

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*]: $OclSelf\ x\ H\ fst\text{-}snd = (\lambda\tau\ .\ if\ (\delta\ x)\ \tau = true\ \tau$
$\quad then\ if\ oid\text{-}of\ (x\ \tau) \in dom(heap(fst\ \tau)) \wedge oid\text{-}of\ (x\ \tau) \in dom(heap\ (snd\ \tau))$

$\quad\quad then\ \ H\ \ulcorner(heap(fst\text{-}snd\ \tau))(oid\text{-}of\ (x\ \tau))\urcorner$
$\quad\quad else\ invalid\ \tau$
$\quad else\ invalid\ \tau)$

definition $OclSelf\text{-}at\text{-}pre$ :: $('\mathfrak{A}::object,'\alpha::\{null,object\})val \Rightarrow$
$('\mathfrak{A} \Rightarrow '\alpha) \Rightarrow$
$('\mathfrak{A}::object,'\alpha::\{null,object\})val\ ((\text{-})@pre(\text{-}))$
where $x\ @pre\ H = OclSelf\ x\ H\ fst$

definition $OclSelf\text{-}at\text{-}post$ :: $('\mathfrak{A}::object,'\alpha::\{null,object\})val \Rightarrow$
$('\mathfrak{A} \Rightarrow '\alpha) \Rightarrow$
$('\mathfrak{A}::object,'\alpha::\{null,object\})val\ ((\text{-})@post(\text{-}))$
where $x\ @post\ H = OclSelf\ x\ H\ snd$

Framing Theorem

lemma *all-oid-diff*:
assumes *def-x* : $\tau \models \delta\ x$
assumes *def-X* : $\tau \models \delta\ X$
assumes *def-X'* : $\bigwedge x.\ x \in \ulcorner\ulcorner Rep\text{-}Set_{base}\ (X\ \tau)\urcorner\urcorner \Longrightarrow x \neq null$

defines $P \equiv (\lambda a.\ not\ (StrictRefEq_{Object}\ x\ a))$
shows $(\tau \models X\text{->}forAll_{Set}(a|\ P\ a)) = (oid\text{-}of\ (x\ \tau) \notin oid\text{-}of\ `\ \ulcorner\ulcorner Rep\text{-}Set_{base}\ (X\ \tau)\urcorner\urcorner)$

theorem *framing*:
assumes *modifiesclause*:$\tau \models (X\text{->}excluding_{Set}(x))\text{->}oclIsModifiedOnly()$
and *oid-is-typerepr* : $\tau \models X\text{->}forAll_{Set}(a|\ not\ (StrictRefEq_{Object}\ x\ a))$
shows $\tau \models (x\ @pre\ P\ \triangleq\ (x\ @post\ P))$

As corollary, the framing property can be expressed with only the strong equality as comparison operator.

theorem *framing'*:
assumes *wff* : $WFF\ \tau$
assumes *modifiesclause*:$\tau \models (X\text{->}excluding_{Set}(x))\text{->}oclIsModifiedOnly()$
and *oid-is-typerepr* : $\tau \models X\text{->}forAll_{Set}(a|\ not\ (x \triangleq a))$
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*:
$\tau \models X\text{->}forAll_{Set}(y \mid (H\ .allInstances()\text{->}includes_{Set}(x)\ and\ H\ .allInstances()\text{->}includes_{Set}(y))\ or$
$(H\ .allInstances@pre()\text{->}includes_{Set}(x)\ and\ H\ .allInstances@pre()\text{->}includes_{Set}(y)))$
shows $\tau \models (x\ @pre\ P\ \triangleq\ (x\ @post\ P))$

Miscellaneous

lemma *pre-post-new*: $\tau \models (x\ .oclIsNew()) \Longrightarrow \neg\ (\tau \models \upsilon(x\ @pre\ H1)) \land \neg\ (\tau \models \upsilon(x\ @post\ H2))$

lemma *pre-post-old*: $\tau \models (x\ .oclIsDeleted()) \Longrightarrow \neg\ (\tau \models \upsilon(x\ @pre\ H1)) \land \neg\ (\tau \models \upsilon(x\ @post\ H2))$

lemma *pre-post-absent*: $\tau \models (x\ .oclIsAbsent()) \Longrightarrow \neg\ (\tau \models \upsilon(x\ @pre\ H1)) \land \neg\ (\tau \models \upsilon(x\ @post\ H2))$

lemma *pre-post-maintained*: $(\tau \models \upsilon(x\ @pre\ H1) \lor \tau \models \upsilon(x\ @post\ H2)) \Longrightarrow \tau \models (x\ .oclIsMaintained())$

lemma *pre-post-maintained'*:
$\tau \models (x\ .oclIsMaintained()) \Longrightarrow (\tau \models \upsilon(x\ @pre\ (Some\ o\ H1)) \wedge \tau \models \upsilon(x\ @post\ (Some\ o\ H2)))$

lemma *framing-same-state*: $(\sigma, \sigma) \models (x\ @pre\ H \triangleq (x\ @post\ H))$

Modeling of an operation contract for an operation with 2 arguments, (so depending on three parameters if one takes "self" into account).

locale *contract-scheme* $=$
  fixes $f\text{-}\upsilon$
  fixes $f\text{-}lam$
  fixes $f\ ::\ ('\mathfrak{A},'\alpha 0::null)val \Rightarrow$
    $'b \Rightarrow$
    $('\mathfrak{A},'res::null)val$
  fixes PRE
  fixes $POST$
  assumes *def-scheme'*: $f\ self\ x \equiv (\lambda\ \tau.\ if\ (\tau \models (\delta\ self)) \wedge f\text{-}\upsilon\ x\ \tau$
    $then\ SOME\ res.\ (\tau \models PRE\ self\ x) \wedge$
      $(\tau \models POST\ self\ x\ (\lambda\ \text{-}.\ res))$
    $else\ invalid\ \tau)$
  assumes *all-post'*: $\forall\ \sigma\ \sigma'\ \sigma''.\ ((\sigma,\sigma') \models PRE\ self\ x) = ((\sigma,\sigma'') \models PRE\ self\ x)$

  assumes $cp_{PRE}{}'$: $PRE\ (self)\ x\ \tau = PRE\ (\lambda\ \text{-}.\ self\ \tau)\ (f\text{-}lam\ x\ \tau)\ \tau$

  assumes $cp_{POST}{}'$:$POST\ (self)\ x\ (res)\ \tau = POST\ (\lambda\ \text{-}.\ self\ \tau)\ (f\text{-}lam\ x\ \tau)\ (\lambda\ \text{-}.\ res\ \tau)\ \tau$
  assumes *f-υ-val*: $\bigwedge a1.\ f\text{-}\upsilon\ (f\text{-}lam\ a1\ \tau)\ \tau = f\text{-}\upsilon\ a1\ \tau$
begin
  lemma *strict0* [*simp*]: $f\ invalid\ X = invalid$

  lemma *nullstrict0*[*simp*]: $f\ null\ X = invalid$

  lemma *cp0* : $f\ self\ a1\ \tau = f\ (\lambda\ \text{-}.\ self\ \tau)\ (f\text{-}lam\ a1\ \tau)\ \tau$

  theorem *unfold'* :
    assumes *context-ok*: $cp\ E$
    and *args-def-or-valid*: $(\tau \models \delta\ self) \wedge f\text{-}\upsilon\ a1\ \tau$
    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 $\tau \models E(f\ self\ a1)$

  lemma *unfold2'* :
    assumes *context-ok*: $cp\ E$
    and *args-def-or-valid*: $(\tau \models \delta\ self) \wedge (f\text{-}\upsilon\ a1\ \tau)$
    and *pre-satisfied*: $\tau \models PRE\ self\ a1$
    and *postsplit-satisfied*: $\tau \models POST'\ self\ a1$
    and *post-decomposable* : $\bigwedge res.\ (POST\ self\ a1\ res) =$
      $((POST'\ self\ a1)\ \ and\ (res \triangleq (BODY\ self\ a1)))$
    shows $(\tau \models E(f\ self\ a1)) = (\tau \models E(BODY\ self\ a1))$
end

locale *contract0* $=$

fixes $f\ \ :: (\prime\mathfrak{A},\prime\alpha 0::null)val \Rightarrow$
$(\prime\mathfrak{A},\prime res::null)val$
fixes *PRE*
fixes *POST*
assumes *def-scheme*: $f\ self \equiv\ (\lambda\ \tau.\ if\ (\tau \models (\delta\ self))$
$then\ SOME\ res.\ (\tau \models PRE\ self) \wedge$
$(\tau \models POST\ self\ (\lambda\ \text{-}.\ res))$
$else\ invalid\ \tau)$
assumes *all-post*: $\forall\ \sigma\ \sigma'\ \sigma''.\ ((\sigma,\sigma') \models PRE\ self) = ((\sigma,\sigma'') \models PRE\ self)$

assumes cp_{PRE}: $PRE\ (self)\ \ \tau = PRE\ (\lambda\ \text{-}.\ self\ \tau)\ \tau$

assumes cp_{POST}:$POST\ (self)\ (res)\ \tau = POST\ (\lambda\ \text{-}.\ self\ \tau)\ (\lambda\ \text{-}.\ res\ \tau)\ \tau$

sublocale *contract0* $<$ *contract-scheme* $\lambda\text{-}\ \text{-}.\ True\ \lambda x\ \text{-}.\ x\ \lambda x\ \text{-}.\ f\ x\ \lambda x\ \text{-}.\ PRE\ x\ \lambda x\ \text{-}.\ POST\ x$

context *contract0*
begin
lemma *cp-pre*: $cp\ self' \Longrightarrow\ cp\ (\lambda X.\ PRE\ (self'\ X)\ \ )$

lemma *cp-post*: $cp\ self' \Longrightarrow cp\ res'\ \Longrightarrow cp\ (\lambda X.\ POST\ (self'\ X)\ (res'\ X))$

lemma *cp* [*simp*]: $cp\ self' \Longrightarrow\ cp\ res' \Longrightarrow cp\ (\lambda X.\ f\ (self'\ X)\ )$

lemmas *unfold* = *unfold*′[*simplified*]

lemma *unfold2* :
assumes $cp\ E$
and $(\tau \models \delta\ self)$
and $\tau \models PRE\ self$
and $\tau \models POST'\ self$
and $\bigwedge res.\ (POST\ self\ res) =$
$((POST'\ self)\ \ and\ (res \triangleq (BODY\ self)))$
shows $(\tau \models E(f\ self)) = (\tau \models E(BODY\ self))$

end

locale *contract1* =
fixes $f\ \ :: (\prime\mathfrak{A},\prime\alpha 0::null)val \Rightarrow$
$(\prime\mathfrak{A},\prime\alpha 1::null)val \Rightarrow$
$(\prime\mathfrak{A},\prime res::null)val$
fixes *PRE*
fixes *POST*
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 *all-post*: $\forall\ \sigma\ \sigma'\ \sigma''.\ \ ((\sigma,\sigma') \models PRE\ self\ a1) = ((\sigma,\sigma'') \models PRE\ self\ a1)$

assumes cp_{PRE}: $PRE\ (self)\ (a1)\ \ \tau = PRE\ (\lambda\ \text{-}.\ self\ \tau)\ (\lambda\ \text{-}.\ a1\ \tau)\ \tau$

assumes cp_{POST}:$POST\ (self)\ (a1)\ (res)\ \tau = POST\ (\lambda\ \text{-}.\ self\ \tau)(\lambda\ \text{-}.\ a1\ \tau)\ (\lambda\ \text{-}.\ res\ \tau)\ \tau$

sublocale *contract1* $<$ *contract-scheme* $\lambda a1\ \tau.\ (\tau \models \upsilon\ a1)\ \lambda a1\ \tau.\ (\lambda\ \text{-}.\ a1\ \tau)$

context *contract1*
begin
lemma *strict1*[*simp*]: $f\ self\ invalid = invalid$

lemma *cp-pre*: $cp\ self' \Longrightarrow cp\ a1' \Longrightarrow\ cp\ (\lambda X.\ PRE\ (self'\ X)\ (a1'\ X)\ )$

lemma *cp-post*: $cp\ self' \Longrightarrow cp\ a1' \Longrightarrow cp\ res'$
$\Longrightarrow cp\ (\lambda X.\ POST\ (self'\ X)\ (a1'\ X)\ (res'\ X))$

lemma *cp* [*simp*]: $cp\ self' \Longrightarrow cp\ a1' \Longrightarrow\ cp\ res' \Longrightarrow cp\ (\lambda X.\ f\ (self'\ X)\ (a1'\ X))$

lemmas $unfold = unfold'$
lemmas $unfold2 = unfold2'$
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*
fixes *POST*
assumes *def-scheme*: $f\ self\ a1\ a2 \equiv$
$(\lambda\ \tau.\ if\ (\tau \models (\delta\ self)) \land\ (\tau \models \upsilon\ a1) \land\ (\tau \models \upsilon\ a2)$
$then\ SOME\ res.\ (\tau \models PRE\ self\ a1\ a2) \land$
$(\tau \models POST\ self\ a1\ a2\ (\lambda\ \text{-}.\ res))$
$else\ invalid\ \tau)$
assumes *all-post*: $\forall\ \sigma\ \sigma'\ \sigma''.\ \ ((\sigma,\sigma') \models PRE\ self\ a1\ a2) = ((\sigma,\sigma'') \models PRE\ self\ a1\ a2)$

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}:$\bigwedge res.\ POST\ (self)\ (a1)\ (a2)\ (res)\ \tau =$
$POST\ (\lambda\ \text{-}.\ self\ \tau)(\lambda\ \text{-}.\ a1\ \tau)(\lambda\ \text{-}.\ a2\ \tau)\ (\lambda\ \text{-}.\ res\ \tau)\ \tau$

sublocale *contract2* $<$ *contract-scheme* $\lambda(a1,a2)\ \tau.\ (\tau \models \upsilon\ a1) \land (\tau \models \upsilon\ a2)$
$\lambda(a1,a2)\ \tau.\ (\lambda\ \text{-}.a1\ \tau,\ \lambda\ \text{-}.a2\ \tau)$
$(\lambda x\ (a,b).\ f\ x\ a\ b)$
$(\lambda x\ (a,b).\ PRE\ x\ a\ b)$
$(\lambda x\ (a,b).\ POST\ x\ a\ b)$

context *contract2*
begin
lemma *strict0*[*simp*] : $f\ invalid\ X\ Y = invalid$

lemma *nullstrict0*[*simp*]: $f\ null\ X\ Y = invalid$

lemma *strict1*[*simp*]: $f\ self\ invalid\ Y = invalid$

lemma *strict2*[*simp*]: $f\ self\ X\ invalid = invalid$

lemma *cp-pre*: $cp\ self' \Longrightarrow cp\ a1' \Longrightarrow cp\ a2' \Longrightarrow cp\ (\lambda X.\ PRE\ (self'\ X)\ (a1'\ X)\ (a2'\ X)\ )$

lemma *cp-post*: $cp\ self' \Longrightarrow cp\ a1' \Longrightarrow cp\ a2' \Longrightarrow cp\ res'$
$\Longrightarrow cp\ (\lambda X.\ POST\ (self'\ X)\ (a1'\ X)\ (a2'\ X)\ (res'\ X))$

lemma *cp0* : $f\ self\ a1\ a2\ \tau = f\ (\lambda\ \text{-}.\ self\ \tau)\ (\lambda\ \text{-}.\ a1\ \tau)\ (\lambda\ \text{-}.\ a2\ \tau)\ \tau$

lemma *cp* [*simp*]: $cp\ self' \Longrightarrow cp\ a1' \Longrightarrow cp\ a2' \Longrightarrow cp\ res'$
$\Longrightarrow cp\ (\lambda X.\ f\ (self'\ X)\ (a1'\ X)\ (a2'\ X))$

theorem *unfold* :
assumes $cp\ E$
and $(\tau \models \delta\ self) \wedge (\tau \models \upsilon\ a1) \wedge\ (\tau \models \upsilon\ a2)$
and $\tau \models PRE\ self\ a1\ a2$
and $\exists res.\ (\tau \models POST\ self\ a1\ a2\ (\lambda\ \text{-}.\ res))$
and $(\bigwedge res.\ \tau \models POST\ self\ a1\ a2\ (\lambda\ \text{-}.\ res)\ \Longrightarrow \tau \models E\ (\lambda\ \text{-}.\ res))$
shows $\tau \models E(f\ self\ a1\ a2)$

lemma *unfold2* :
assumes $cp\ E$
and $(\tau \models \delta\ self) \wedge (\tau \models \upsilon\ a1) \wedge\ (\tau \models \upsilon\ a2)$
and $\tau \models PRE\ self\ a1\ a2$
and $\tau \models POST'\ self\ a1\ a2$
and $\bigwedge res.\ (POST\ self\ a1\ a2\ res) =$
$((POST'\ self\ a1\ a2)\ \ and\ (res \triangleq (BODY\ self\ a1\ a2)))$
shows $(\tau \models E(f\ self\ a1\ a2)) = (\tau \models E(BODY\ self\ a1\ a2))$
end

A.7. Example I : The Employee Analysis Model (UML)

A.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 [3, 5]. In this paper, we follow another approach for UML 2.4: we define the concepts of the compilation informally, and present a concrete example which is verified in Isabelle/HOL.

Outlining the Example

We are presenting here an "analysis-model" of the (slightly modified) example Figure 7.3, page 20 of the OCL standard [20]. 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

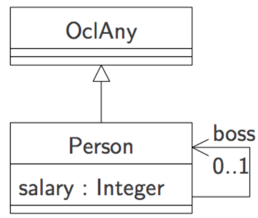


Figure A.2.: A simple UML class model drawn from Figure 7.3, page 20 of [20].

http://afp.sourceforge.net/entries/Featherweight_OCL.shtml). To be precise, this theory contains the formalization of the data-part covered by the UML class model (see Figure A.2):

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).

A.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 $\mathit{type}_{Person} = \mathit{mk}_{Person}\ \mathit{oid}$
$\quad\quad\mathit{int\ option}$

datatype $\mathit{type}_{OclAny} = \mathit{mk}_{OclAny}\ \mathit{oid}$
$\quad\quad(\mathit{int\ option})\ \mathit{option}$

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 $oid\text{-}of\text{-}type_{Person}\text{-}def$: $oid\text{-}of\ x = (case\ x\ of\ mk_{Person}\ oid\ \text{-} \Rightarrow oid)$
  instance
end

instantiation $type_{OclAny}$ :: *object*
begin
  definition $oid\text{-}of\text{-}type_{OclAny}\text{-}def$: $oid\text{-}of\ x = (case\ x\ of\ mk_{OclAny}\ oid\ \text{-} \Rightarrow oid)$
  instance
end

instantiation $\mathfrak{A}$ :: *object*
begin
  definition $oid\text{-}of\text{-}\mathfrak{A}\text{-}def$: $oid\text{-}of\ x = (case\ x\ of$
      $in_{Person}\ person \Rightarrow oid\text{-}of\ person$
    $|\ in_{OclAny}\ oclany \Rightarrow oid\text{-}of\ oclany)$
  instance
end

A.7.3. 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 *cps23*[*standard*] =
  $cp\text{-}StrictRefEq_{Object}$[*of x::Person y::Person* τ,
      simplified $StrictRefEq_{Object}\text{-}_{Person}$[*symmetric*]]
  cp-intro(*9*) [*of P::Person* $\Rightarrow$*PersonQ::Person* $\Rightarrow$*Person*,
      simplified $StrictRefEq_{Object}\text{-}_{Person}$[*symmetric*]]
  $StrictRefEq_{Object}$*-def* [*of x::Person y::Person*,
      simplified $StrictRefEq_{Object}\text{-}_{Person}$[*symmetric*]]
  $StrictRefEq_{Object}$*-defargs* [*of - x::Person y::Person*,
      simplified $StrictRefEq_{Object}\text{-}_{Person}$[*symmetric*]]
  $StrictRefEq_{Object}$*-strict1*
      [*of x::Person*,
      simplified $StrictRefEq_{Object}\text{-}_{Person}$[*symmetric*]]
  $StrictRefEq_{Object}$*-strict2*
      [*of x::Person*,
      simplified $StrictRefEq_{Object}\text{-}_{Person}$[*symmetric*]]

For each Class C, we will have a casting operation `.oclAsType(`C`)`, a test on the actual type `.oclIsTypeOf(`C`)` as well as its relaxed form `.oclIsKindOf(`C`)` (corresponding exactly to Java’s `instanceof`-operator.

Thus, since we have two class-types in our concrete class hierarchy, we have two operations to declare and to provide two overloading definitions for the two static types.

A.7.4. OclAsType

Definition

consts $OclAsType_{OclAny}$ :: $'\alpha \Rightarrow OclAny$ $((\text{-})\ .oclAsType'(OclAny'))$
consts $OclAsType_{Person}$ :: $'\alpha \Rightarrow Person$ $((\text{-})\ .oclAsType'(Person'))$

definition $OclAsType_{OclAny}\text{-}\mathfrak{A} = (\lambda u.\ \lfloor case\ u\ of\ in_{OclAny}\ a \Rightarrow a$
$\quad\quad | \ in_{Person}\ (mk_{Person}\ oid\ a) \Rightarrow mk_{OclAny}\ oid\ \lfloor a \rfloor \rfloor)$

lemma $OclAsType_{OclAny}\text{-}\mathfrak{A}\text{-}some$: $OclAsType_{OclAny}\text{-}\mathfrak{A}\ x \neq None$

defs (**overloaded**) $OclAsType_{OclAny}\text{-}OclAny$:
$(X{::}OclAny)\ .oclAsType(OclAny) \equiv X$

defs (**overloaded**) $OclAsType_{OclAny}\text{-}Person$:
$(X{::}Person)\ .oclAsType(OclAny) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\quad \bot \ \Rightarrow invalid\ \tau$
$\quad | \lfloor\bot\rfloor \Rightarrow null\ \tau$
$\quad | \lfloor\lfloor mk_{Person}\ oid\ a\rfloor\rfloor \Rightarrow \lfloor\lfloor (mk_{OclAny}\ oid\ \lfloor a\rfloor)\rfloor\rfloor)$

definition $OclAsType_{Person}\text{-}\mathfrak{A} = (\lambda u.\ case\ u\ of\ in_{Person}\ p \Rightarrow \lfloor p\rfloor$
$\quad\quad | \ in_{OclAny}\ (mk_{OclAny}\ oid\ \lfloor a\rfloor) \Rightarrow \lfloor mk_{Person}\ oid\ a\rfloor$
$\quad\quad | \ \text{-} \Rightarrow None)$

defs (**overloaded**) $OclAsType_{Person}\text{-}OclAny$:
$(X{::}OclAny)\ .oclAsType(Person) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\quad \bot \ \Rightarrow invalid\ \tau$
$\quad | \lfloor\bot\rfloor \Rightarrow null\ \tau$
$\quad | \lfloor\lfloor mk_{OclAny}\ oid\ \bot\rfloor\rfloor \Rightarrow \ invalid\ \tau \quad (*\ down{-}cast\ exception\ *)$
$\quad | \lfloor\lfloor mk_{OclAny}\ oid\ \lfloor a\rfloor\rfloor\rfloor \Rightarrow \ \lfloor\lfloor mk_{Person}\ oid\ a\rfloor\rfloor)$

defs (**overloaded**) $OclAsType_{Person}\text{-}Person$:
$(X{::}Person)\ .oclAsType(Person) \equiv X$

Execution with Invalid or Null as Argument

lemma $OclAsType_{OclAny}\text{-}OclAny\text{-}strict$: $(invalid{::}OclAny)\ .oclAsType(OclAny) = invalid$
lemma $OclAsType_{OclAny}\text{-}OclAny\text{-}nullstrict$: $(null{::}OclAny)\ .oclAsType(OclAny) = null$
lemma $OclAsType_{OclAny}\text{-}Person\text{-}strict[simp]$: $(invalid{::}Person)\ .oclAsType(OclAny) = invalid$
lemma $OclAsType_{OclAny}\text{-}Person\text{-}nullstrict[simp]$: $(null{::}Person)\ .oclAsType(OclAny) = null$
lemma $OclAsType_{Person}\text{-}OclAny\text{-}strict[simp]$: $(invalid{::}OclAny)\ .oclAsType(Person) = invalid$
lemma $OclAsType_{Person}\text{-}OclAny\text{-}nullstrict[simp]$: $(null{::}OclAny)\ .oclAsType(Person) = null$
lemma $OclAsType_{Person}\text{-}Person\text{-}strict$: $(invalid{::}Person)\ .oclAsType(Person) = invalid$
lemma $OclAsType_{Person}\text{-}Person\text{-}nullstrict$: $(null{::}Person)\ .oclAsType(Person) = null$

A.7.5. OclIsTypeOf

Definition

consts $OclIsTypeOf_{OclAny}$ :: $'\alpha \Rightarrow Boolean$ $((\text{-}).oclIsTypeOf'(OclAny'))$
consts $OclIsTypeOf_{Person}$ :: $'\alpha \Rightarrow Boolean$ $((\text{-}).oclIsTypeOf'(Person'))$

defs (**overloaded**) *OclIsTypeOf*$_{OclAny}$-*OclAny*:
$(X{::}OclAny)\ .oclIsTypeOf(OclAny) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\perp\ \Rightarrow invalid\ \tau$
$\mid \lfloor\perp\rfloor \Rightarrow true\ \tau\ \ (*\ invalid\ ??\ *)$
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \perp\rfloor\rfloor \Rightarrow true\ \tau$
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \lfloor\_\rfloor\rfloor\rfloor \Rightarrow false\ \tau)$

lemma *OclIsTypeOf*$_{OclAny}$-*OclAny'*:
$(X{::}OclAny)\ .oclIsTypeOf(OclAny) =$
$(\lambda\ \tau.\ if\ \tau \models \upsilon\ X\ then\ (case\ X\ \tau\ of$
$\lfloor\perp\rfloor \Rightarrow true\ \tau\ \ (*\ invalid\ ??\ *)$
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \perp\rfloor\rfloor \Rightarrow true\ \tau$
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \lfloor\_\rfloor\rfloor\rfloor \Rightarrow false\ \tau)$
$else\ invalid\ \tau)$

interpretation *OclIsTypeOf*$_{OclAny}$-*OclAny* :
profile-mono-schemeV
$OclIsTypeOf_{OclAny}{::}OclAny \Rightarrow Boolean$
$\lambda\ X.\ (case\ X\ of$
$\lfloor None\rfloor \Rightarrow \lfloor\lfloor True\rfloor\rfloor\ \ (*\ invalid\ ??\ *)$
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ None\rfloor\rfloor \Rightarrow \lfloor\lfloor True\rfloor\rfloor$
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \lfloor\_\rfloor\rfloor\rfloor \Rightarrow \lfloor\lfloor False\rfloor\rfloor)$

defs (**overloaded**) *OclIsTypeOf*$_{OclAny}$-*Person*:
$(X{::}Person)\ .oclIsTypeOf(OclAny) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\perp\ \Rightarrow invalid\ \tau$
$\mid \lfloor\perp\rfloor \Rightarrow true\ \tau\ \ \ (*\ invalid\ ??\ *)$
$\mid \lfloor\lfloor\_\rfloor\rfloor \Rightarrow false\ \tau)$

defs (**overloaded**) *OclIsTypeOf*$_{Person}$-*OclAny*:
$(X{::}OclAny)\ .oclIsTypeOf(Person) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\perp\ \Rightarrow invalid\ \tau$
$\mid \lfloor\perp\rfloor \Rightarrow true\ \tau$
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \perp\rfloor\rfloor \Rightarrow false\ \tau$
$\mid \lfloor\lfloor mk_{OclAny}\ oid\ \lfloor\_\rfloor\rfloor\rfloor \Rightarrow true\ \tau)$

defs (**overloaded**) *OclIsTypeOf*$_{Person}$-*Person*:
$(X{::}Person)\ .oclIsTypeOf(Person) \equiv$
$(\lambda\tau.\ case\ X\ \tau\ of$
$\perp \Rightarrow invalid\ \tau$
$\mid \_ \Rightarrow true\ \tau)$

Execution with Invalid or Null as Argument

lemma *OclIsTypeOf*$_{OclAny}$-*OclAny-strict1*[*simp*]:
$(invalid{::}OclAny)\ .oclIsTypeOf(OclAny) = invalid$
lemma *OclIsTypeOf*$_{OclAny}$-*OclAny-strict2*[*simp*]:
$(null{::}OclAny)\ .oclIsTypeOf(OclAny) = true$

lemma $\mathit{OclIsTypeOf}_{OclAny}$*-Person-strict1*[*simp*]:
$(\mathit{invalid}::\mathit{Person})\ .\mathit{oclIsTypeOf}(\mathit{OclAny}) = \mathit{invalid}$
lemma $\mathit{OclIsTypeOf}_{OclAny}$*-Person-strict2*[*simp*]:
$(\mathit{null}::\mathit{Person})\ .\mathit{oclIsTypeOf}(\mathit{OclAny}) = \mathit{true}$
lemma $\mathit{OclIsTypeOf}_{Person}$*-OclAny-strict1*[*simp*]:
$(\mathit{invalid}::\mathit{OclAny})\ .\mathit{oclIsTypeOf}(\mathit{Person}) = \mathit{invalid}$
lemma $\mathit{OclIsTypeOf}_{Person}$*-OclAny-strict2*[*simp*]:
$(\mathit{null}::\mathit{OclAny})\ .\mathit{oclIsTypeOf}(\mathit{Person}) = \mathit{true}$
lemma $\mathit{OclIsTypeOf}_{Person}$*-Person-strict1*[*simp*]:
$(\mathit{invalid}::\mathit{Person})\ .\mathit{oclIsTypeOf}(\mathit{Person}) = \mathit{invalid}$
lemma $\mathit{OclIsTypeOf}_{Person}$*-Person-strict2*[*simp*]:
$(\mathit{null}::\mathit{Person})\ .\mathit{oclIsTypeOf}(\mathit{Person}) = \mathit{true}$

Up Down Casting

lemma *actualType-larger-staticType*:
assumes *isdef*: $\tau \models (\delta\ X)$
shows $\tau \models (X::\mathit{Person})\ .\mathit{oclIsTypeOf}(\mathit{OclAny}) \triangleq \mathit{false}$

lemma *down-cast-type*:
assumes *isOclAny*: $\tau \models (X::\mathit{OclAny})\ .\mathit{oclIsTypeOf}(\mathit{OclAny})$
and *non-null*: $\tau \models (\delta\ X)$
shows $\tau \models (X\ .\mathit{oclAsType}(\mathit{Person})) \triangleq \mathit{invalid}$

lemma *down-cast-type'*:
assumes *isOclAny*: $\tau \models (X::\mathit{OclAny})\ .\mathit{oclIsTypeOf}(\mathit{OclAny})$
and *non-null*: $\tau \models (\delta\ X)$
shows $\tau \models \mathit{not}\ (\upsilon\ (X\ .\mathit{oclAsType}(\mathit{Person})))$

lemma *up-down-cast* :
assumes *isdef*: $\tau \models (\delta\ X)$
shows $\tau \models ((X::\mathit{Person})\ .\mathit{oclAsType}(\mathit{OclAny})\ .\mathit{oclAsType}(\mathit{Person}) \triangleq X)$

lemma *up-down-cast-Person-OclAny-Person* [*simp*]:
shows $((X::\mathit{Person})\ .\mathit{oclAsType}(\mathit{OclAny})\ .\mathit{oclAsType}(\mathit{Person}) = X)$

lemma *up-down-cast-Person-OclAny-Person'*:
assumes $\tau \models \upsilon\ X$
shows $\tau \models (((X :: \mathit{Person})\ .\mathit{oclAsType}(\mathit{OclAny})\ .\mathit{oclAsType}(\mathit{Person})) \doteq X)$

lemma *up-down-cast-Person-OclAny-Person''*:
assumes $\tau \models \upsilon\ (X :: \mathit{Person})$
shows $\tau \models (X\ .\mathit{oclIsTypeOf}(\mathit{Person})\ \mathit{implies}\ (X\ .\mathit{oclAsType}(\mathit{OclAny})\ .\mathit{oclAsType}(\mathit{Person})) \doteq X)$

A.7.6. OclIsKindOf

Definition

consts $\mathit{OclIsKindOf}_{OclAny} :: {}'\alpha \Rightarrow \mathit{Boolean}\ ((\text{-}).\mathit{oclIsKindOf}'(\mathit{OclAny}'))$
consts $\mathit{OclIsKindOf}_{Person} :: {}'\alpha \Rightarrow \mathit{Boolean}\ ((\text{-}).\mathit{oclIsKindOf}'(\mathit{Person}'))$

defs (**overloaded**) $\mathit{OclIsKindOf}_{\mathit{OclAny}}\text{-}\mathit{OclAny}$:
$(X{::}\mathit{OclAny})\ .\mathit{oclIsKindOf}(\mathit{OclAny}) \equiv$
$(\lambda\tau.\ \mathit{case}\ X\ \tau\ \mathit{of}$
$\bot \Rightarrow \mathit{invalid}\ \tau$
$\mid \text{-} \Rightarrow \mathit{true}\ \tau)$

defs (**overloaded**) $\mathit{OclIsKindOf}_{\mathit{OclAny}}\text{-}\mathit{Person}$:
$(X{::}\mathit{Person})\ .\mathit{oclIsKindOf}(\mathit{OclAny}) \equiv$
$(\lambda\tau.\ \mathit{case}\ X\ \tau\ \mathit{of}$
$\bot \Rightarrow \mathit{invalid}\ \tau$
$\mid \text{-} \Rightarrow \mathit{true}\ \tau)$

defs (**overloaded**) $\mathit{OclIsKindOf}_{\mathit{Person}}\text{-}\mathit{OclAny}$:
$(X{::}\mathit{OclAny})\ .\mathit{oclIsKindOf}(\mathit{Person}) \equiv$
$(\lambda\tau.\ \mathit{case}\ X\ \tau\ \mathit{of}$
$\bot \ \Rightarrow \mathit{invalid}\ \tau$
$\mid \llcorner\bot\lrcorner \Rightarrow \mathit{true}\ \tau$
$\mid \llcorner\llcorner \mathit{mk}_{\mathit{OclAny}}\ \mathit{oid}\ \bot\ \lrcorner\lrcorner \Rightarrow \mathit{false}\ \tau$
$\mid \llcorner\llcorner \mathit{mk}_{\mathit{OclAny}}\ \mathit{oid}\ \llcorner\text{-}\lrcorner\ \lrcorner\lrcorner \Rightarrow \mathit{true}\ \tau)$

defs (**overloaded**) $\mathit{OclIsKindOf}_{\mathit{Person}}\text{-}\mathit{Person}$:
$(X{::}\mathit{Person})\ .\mathit{oclIsKindOf}(\mathit{Person}) \equiv$
$(\lambda\tau.\ \mathit{case}\ X\ \tau\ \mathit{of}$
$\bot \Rightarrow \mathit{invalid}\ \tau$
$\mid \text{-} \Rightarrow \mathit{true}\ \tau)$

Execution with Invalid or Null as Argument

lemma $\mathit{OclIsKindOf}_{\mathit{OclAny}}\text{-}\mathit{OclAny}\text{-}\mathit{strict1}[\mathit{simp}]$: $(\mathit{invalid}{::}\mathit{OclAny})\ .\mathit{oclIsKindOf}(\mathit{OclAny}) = \mathit{invalid}$
lemma $\mathit{OclIsKindOf}_{\mathit{OclAny}}\text{-}\mathit{OclAny}\text{-}\mathit{strict2}[\mathit{simp}]$: $(\mathit{null}{::}\mathit{OclAny})\ .\mathit{oclIsKindOf}(\mathit{OclAny}) = \mathit{true}$
lemma $\mathit{OclIsKindOf}_{\mathit{OclAny}}\text{-}\mathit{Person}\text{-}\mathit{strict1}[\mathit{simp}]$: $(\mathit{invalid}{::}\mathit{Person})\ .\mathit{oclIsKindOf}(\mathit{OclAny}) = \mathit{invalid}$
lemma $\mathit{OclIsKindOf}_{\mathit{OclAny}}\text{-}\mathit{Person}\text{-}\mathit{strict2}[\mathit{simp}]$: $(\mathit{null}{::}\mathit{Person})\ .\mathit{oclIsKindOf}(\mathit{OclAny}) = \mathit{true}$
lemma $\mathit{OclIsKindOf}_{\mathit{Person}}\text{-}\mathit{OclAny}\text{-}\mathit{strict1}[\mathit{simp}]$: $(\mathit{invalid}{::}\mathit{OclAny})\ .\mathit{oclIsKindOf}(\mathit{Person}) = \mathit{invalid}$
lemma $\mathit{OclIsKindOf}_{\mathit{Person}}\text{-}\mathit{OclAny}\text{-}\mathit{strict2}[\mathit{simp}]$: $(\mathit{null}{::}\mathit{OclAny})\ .\mathit{oclIsKindOf}(\mathit{Person}) = \mathit{true}$
lemma $\mathit{OclIsKindOf}_{\mathit{Person}}\text{-}\mathit{Person}\text{-}\mathit{strict1}[\mathit{simp}]$: $(\mathit{invalid}{::}\mathit{Person})\ .\mathit{oclIsKindOf}(\mathit{Person}) = \mathit{invalid}$
lemma $\mathit{OclIsKindOf}_{\mathit{Person}}\text{-}\mathit{Person}\text{-}\mathit{strict2}[\mathit{simp}]$: $(\mathit{null}{::}\mathit{Person})\ .\mathit{oclIsKindOf}(\mathit{Person}) = \mathit{true}$

Up Down Casting

lemma *actualKind-larger-staticKind*:
assumes *isdef*: $\tau \models (\delta\ X)$
shows $\tau \models\ ((X{::}\mathit{Person})\ .\mathit{oclIsKindOf}(\mathit{OclAny}) \triangleq \mathit{true})$

lemma *down-cast-kind*:
assumes *isOclAny*: $\neg\ (\tau \models ((X{::}\mathit{OclAny}).\mathit{oclIsKindOf}(\mathit{Person})))$
and *non-null*: $\tau \models (\delta\ X)$
shows $\tau \models ((X\ .\mathit{oclAsType}(\mathit{Person})) \triangleq \mathit{invalid})$

A.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 $\mathit{Person} \equiv \mathit{OclAsType}_{\mathit{Person}}\text{-}\mathfrak{A}$
definition $\mathit{OclAny} \equiv \mathit{OclAsType}_{\mathit{OclAny}}\text{-}\mathfrak{A}$
lemmas $[\mathit{simp}] = \mathit{Person\text{-}def}\ \mathit{OclAny\text{-}def}$

lemma $\mathit{OclAllInstances\text{-}generic}_{\mathit{OclAny}}\mathit{\text{-}exec}$: $\mathit{OclAllInstances\text{-}generic}\ \mathit{pre\text{-}post}\ \mathit{OclAny} =$
$(\lambda\tau.\ \mathit{Abs\text{-}Set}_{base}\ \lfloor\lfloor \mathit{Some}\ \text{‘}\ \mathit{OclAny}\ \text{‘}\ \mathit{ran}\ (\mathit{heap}\ (\mathit{pre\text{-}post}\ \tau))\ \rfloor\rfloor)$

lemma $\mathit{OclAllInstances\text{-}at\text{-}post}_{\mathit{OclAny}}\mathit{\text{-}exec}$: $\mathit{OclAny}\ .\mathit{allInstances}() =$
$(\lambda\tau.\ \mathit{Abs\text{-}Set}_{base}\ \lfloor\lfloor \mathit{Some}\ \text{‘}\ \mathit{OclAny}\ \text{‘}\ \mathit{ran}\ (\mathit{heap}\ (\mathit{snd}\ \tau))\ \rfloor\rfloor)$

lemma $\mathit{OclAllInstances\text{-}at\text{-}pre}_{\mathit{OclAny}}\mathit{\text{-}exec}$: $\mathit{OclAny}\ .\mathit{allInstances@pre}() =$
$(\lambda\tau.\ \mathit{Abs\text{-}Set}_{base}\ \lfloor\lfloor \mathit{Some}\ \text{‘}\ \mathit{OclAny}\ \text{‘}\ \mathit{ran}\ (\mathit{heap}\ (\mathit{fst}\ \tau))\ \rfloor\rfloor)$

OclIsTypeOf

lemma $\mathit{OclAny\text{-}allInstances\text{-}generic\text{-}oclIsTypeOf}_{\mathit{OclAny}}\mathit{1}$:
assumes $[\mathit{simp}]$: $\bigwedge x.\ \mathit{pre\text{-}post}\ (x, x) = x$
shows $\exists\tau.\ (\tau \models\ \ ((\mathit{OclAllInstances\text{-}generic}\ \mathit{pre\text{-}post}\ \mathit{OclAny}){-}{>}\mathit{forAll}_{\mathit{Set}}(X|X\ .\mathit{oclIsTypeOf}(\mathit{OclAny}))))$

lemma $\mathit{OclAny\text{-}allInstances\text{-}at\text{-}post\text{-}oclIsTypeOf}_{\mathit{OclAny}}\mathit{1}$:
$\exists\tau.\ (\tau \models\ \ (\mathit{OclAny}\ .\mathit{allInstances}(){-}{>}\mathit{forAll}_{\mathit{Set}}(X|X\ .\mathit{oclIsTypeOf}(\mathit{OclAny}))))$

lemma $\mathit{OclAny\text{-}allInstances\text{-}at\text{-}pre\text{-}oclIsTypeOf}_{\mathit{OclAny}}\mathit{1}$:
$\exists\tau.\ (\tau \models\ \ (\mathit{OclAny}\ .\mathit{allInstances@pre}(){-}{>}\mathit{forAll}_{\mathit{Set}}(X|X\ .\mathit{oclIsTypeOf}(\mathit{OclAny}))))$

lemma $\mathit{OclAny\text{-}allInstances\text{-}generic\text{-}oclIsTypeOf}_{\mathit{OclAny}}\mathit{2}$:
assumes $[\mathit{simp}]$: $\bigwedge x.\ \mathit{pre\text{-}post}\ (x, x) = x$
shows $\exists\tau.\ (\tau \models \mathit{not}\ ((\mathit{OclAllInstances\text{-}generic}\ \mathit{pre\text{-}post}\ \mathit{OclAny}){-}{>}\mathit{forAll}_{\mathit{Set}}(X|X\ .\mathit{oclIsTypeOf}(\mathit{OclAny}))))$

lemma $\mathit{OclAny\text{-}allInstances\text{-}at\text{-}post\text{-}oclIsTypeOf}_{\mathit{OclAny}}\mathit{2}$:
$\exists\tau.\ (\tau \models \mathit{not}\ (\mathit{OclAny}\ .\mathit{allInstances}(){-}{>}\mathit{forAll}_{\mathit{Set}}(X|X\ .\mathit{oclIsTypeOf}(\mathit{OclAny}))))$

lemma $\mathit{OclAny\text{-}allInstances\text{-}at\text{-}pre\text{-}oclIsTypeOf}_{\mathit{OclAny}}\mathit{2}$:
$\exists\tau.\ (\tau \models \mathit{not}\ (\mathit{OclAny}\ .\mathit{allInstances@pre}(){-}{>}\mathit{forAll}_{\mathit{Set}}(X|X\ .\mathit{oclIsTypeOf}(\mathit{OclAny}))))$

lemma $\mathit{Person\text{-}allInstances\text{-}generic\text{-}oclIsTypeOf}_{\mathit{Person}}$:
$\tau \models ((\mathit{OclAllInstances\text{-}generic}\ \mathit{pre\text{-}post}\ \mathit{Person}){-}{>}\mathit{forAll}_{\mathit{Set}}(X|X\ .\mathit{oclIsTypeOf}(\mathit{Person})))$

lemma $\mathit{Person\text{-}allInstances\text{-}at\text{-}post\text{-}oclIsTypeOf}_{\mathit{Person}}$:
$\tau \models (\mathit{Person}\ .\mathit{allInstances}(){-}{>}\mathit{forAll}_{\mathit{Set}}(X|X\ .\mathit{oclIsTypeOf}(\mathit{Person})))$

lemma $\mathit{Person\text{-}allInstances\text{-}at\text{-}pre\text{-}oclIsTypeOf}_{\mathit{Person}}$:
$\tau \models (\mathit{Person}\ .\mathit{allInstances@pre}(){-}{>}\mathit{forAll}_{\mathit{Set}}(X|X\ .\mathit{oclIsTypeOf}(\mathit{Person})))$

OclIsKindOf

lemma *OclAny-allInstances-generic-oclIsKindOf*$_{OclAny}$:
$\tau \models ((\mathit{OclAllInstances\text{-}generic\ pre\text{-}post\ OclAny}){-}{>}\mathit{forAll}_{Set}(X|X\ .\mathit{oclIsKindOf}(\mathit{OclAny})))$

lemma *OclAny-allInstances-at-post-oclIsKindOf*$_{OclAny}$:
$\tau \models (\mathit{OclAny}\ .\mathit{allInstances}(){-}{>}\mathit{forAll}_{Set}(X|X\ .\mathit{oclIsKindOf}(\mathit{OclAny})))$

lemma *OclAny-allInstances-at-pre-oclIsKindOf*$_{OclAny}$:
$\tau \models (\mathit{OclAny}\ .\mathit{allInstances@pre}(){-}{>}\mathit{forAll}_{Set}(X|X\ .\mathit{oclIsKindOf}(\mathit{OclAny})))$

lemma *Person-allInstances-generic-oclIsKindOf*$_{OclAny}$:
$\tau \models ((\mathit{OclAllInstances\text{-}generic\ pre\text{-}post\ Person}){-}{>}\mathit{forAll}_{Set}(X|X\ .\mathit{oclIsKindOf}(\mathit{OclAny})))$

lemma *Person-allInstances-at-post-oclIsKindOf*$_{OclAny}$:
$\tau \models (\mathit{Person}\ .\mathit{allInstances}(){-}{>}\mathit{forAll}_{Set}(X|X\ .\mathit{oclIsKindOf}(\mathit{OclAny})))$

lemma *Person-allInstances-at-pre-oclIsKindOf*$_{OclAny}$:
$\tau \models (\mathit{Person}\ .\mathit{allInstances@pre}(){-}{>}\mathit{forAll}_{Set}(X|X\ .\mathit{oclIsKindOf}(\mathit{OclAny})))$

lemma *Person-allInstances-generic-oclIsKindOf*$_{Person}$:
$\tau \models ((\mathit{OclAllInstances\text{-}generic\ pre\text{-}post\ Person}){-}{>}\mathit{forAll}_{Set}(X|X\ .\mathit{oclIsKindOf}(\mathit{Person})))$

lemma *Person-allInstances-at-post-oclIsKindOf*$_{Person}$:
$\tau \models (\mathit{Person}\ .\mathit{allInstances}(){-}{>}\mathit{forAll}_{Set}(X|X\ .\mathit{oclIsKindOf}(\mathit{Person})))$

lemma *Person-allInstances-at-pre-oclIsKindOf*$_{Person}$:
$\tau \models (\mathit{Person}\ .\mathit{allInstances@pre}(){-}{>}\mathit{forAll}_{Set}(X|X\ .\mathit{oclIsKindOf}(\mathit{Person})))$

A.7.8. The Accessors (any, boss, salary)

Should be generated entirely from a class-diagram.

Definition (of the association Employee-Boss)

We start with a oid for the association; this oid can be used in presence of association classes to represent the association inside an object, pretty much similar to the Design_UML, where we stored an `oid` inside the class as "pointer."

definition $\mathit{oid}_{Person}\mathscr{BOSS}$ ::*oid* **where** $\mathit{oid}_{Person}\mathscr{BOSS} = 10$

From there on, we can already define an empty state which must contain for $\mathit{oid}_{Person}\mathscr{BOSS}$ the empty relation (encoded as association list, since there are associations with a Sequence-like structure).

definition *eval-extract* :: $('\mathfrak{A},('a::\mathit{object})\ \mathit{option\ option})\ \mathit{val}$
$\Rightarrow (\mathit{oid} \Rightarrow ('\mathfrak{A},'c::\mathit{null})\ \mathit{val})$
$\Rightarrow ('\mathfrak{A},'c::\mathit{null})\ \mathit{val}$
where $\mathit{eval\text{-}extract}\ X\ f = (\lambda\ \tau.\ \mathit{case}\ X\ \tau\ \mathit{of}$
$\bot \Rightarrow \mathit{invalid}\ \tau \quad (*\ \mathit{exception\ propagation}\ *)$
$|\ \lfloor \bot \rfloor \Rightarrow \mathit{invalid}\ \tau\ (*\ \mathit{dereferencing\ null\ pointer}\ *)$
$|\ \lfloor\lfloor \mathit{obj} \rfloor\rfloor \Rightarrow f\ (\mathit{oid\text{-}of}\ \mathit{obj})\ \tau)$

definition $choose_2\text{-}1 = fst$
definition $choose_2\text{-}2 = snd$

definition $List\text{-}flatten = (\lambda l.\ (foldl\ ((\lambda acc.\ (\lambda l.\ (foldl\ ((\lambda acc.\ (\lambda l.\ (Cons\ (l)\ (acc)))))\ (acc)\ ((rev\ (l)))))))\ (Nil)\ ((rev\ (l)))))$
definition $deref\text{-}assocs_2 :: ({}'\mathfrak{A}\ state \times {}'\mathfrak{A}\ state \Rightarrow {}'\mathfrak{A}\ state)$
$\Rightarrow (oid\ list\ list \Rightarrow oid\ list \times oid\ list)$
$\Rightarrow oid$
$\Rightarrow (oid\ list \Rightarrow ({}'\mathfrak{A},{}'f)val)$
$\Rightarrow oid$
$\Rightarrow ({}'\mathfrak{A},\ {}'f{::}null)val$
where $deref\text{-}assocs_2\ pre\text{-}post\ to\text{-}from\ assoc\text{-}oid\ f\ oid =$
$(\lambda\tau.\ case\ (assocs\ (pre\text{-}post\ \tau))\ assoc\text{-}oid\ of$
$\llcorner S \lrcorner \Rightarrow f\ (List\text{-}flatten\ (map\ (choose_2\text{-}2 \circ to\text{-}from)$
$(filter\ (\lambda\ p.\ List.member\ (choose_2\text{-}1\ (to\text{-}from\ p))\ oid)\ S)))$
τ
$\mid \text{-} \quad \Rightarrow invalid\ \tau)$

The *pre-post*-parameter is configured with *fst* or *snd*, the *to-from*-parameter either with the identity *id* or the following combinator *switch*:

definition $switch_2\text{-}1 = (\lambda[x,y]\Rightarrow (x,y))$
definition $switch_2\text{-}2 = (\lambda[x,y]\Rightarrow (y,x))$
definition $switch_3\text{-}1 = (\lambda[x,y,z]\Rightarrow (x,y))$
definition $switch_3\text{-}2 = (\lambda[x,y,z]\Rightarrow (x,z))$
definition $switch_3\text{-}3 = (\lambda[x,y,z]\Rightarrow (y,x))$
definition $switch_3\text{-}4 = (\lambda[x,y,z]\Rightarrow (y,z))$
definition $switch_3\text{-}5 = (\lambda[x,y,z]\Rightarrow (z,x))$
definition $switch_3\text{-}6 = (\lambda[x,y,z]\Rightarrow (z,y))$

definition $select\text{-}object :: (({}'\mathfrak{A},\ {}'b{::}null)val)$
$\Rightarrow (({}'\mathfrak{A},{}'b)val \Rightarrow ({}'\mathfrak{A},{}'c)val \Rightarrow ({}'\mathfrak{A},\ {}'b)val)$
$\Rightarrow (({}'\mathfrak{A},\ {}'b)val \Rightarrow ({}'\mathfrak{A},\ {}'d)val)$
$\Rightarrow (oid \Rightarrow ({}'\mathfrak{A},{}'c{::}null)val)$
$\Rightarrow oid\ list$
$\Rightarrow ({}'\mathfrak{A},\ {}'d)val$
where $select\text{-}object\ \ mt\ incl\ smash\ deref\ l\ = smash(foldl\ incl\ mt\ (map\ deref\ l))$
$(*\ smash\ returns\ null\ with\ mt\ in\ input\ (in\ this\ case,\ object\ contains\ null\ pointer)\ *)$

The continuation f is usually instantiated with a smashing function which is either the identity *id* or, for `0..1` cardinalities of associations, the *OclANY*-selector which also handles the *null*-cases appropriately. A standard use-case for this combinator is for example:

term $(select\text{-}object\ mtSet\ UML\text{-}Set.OclIncluding\ OclANY\ f\ \ l\ oid\ ){::}({}'\mathfrak{A},\ {}'a{::}null)val$

definition $deref\text{-}oid_{Person} :: (\mathfrak{A}\ state \times \mathfrak{A}\ state \Rightarrow \mathfrak{A}\ state)$
$\Rightarrow (type_{Person} \Rightarrow (\mathfrak{A},\ {}'c{::}null)val)$
$\Rightarrow oid$
$\Rightarrow (\mathfrak{A},\ {}'c{::}null)val$
where $deref\text{-}oid_{Person}\ fst\text{-}snd\ f\ oid = (\lambda\tau.\ case\ (heap\ (fst\text{-}snd\ \tau))\ oid\ of$
$\llcorner in_{Person}\ obj \lrcorner \Rightarrow f\ obj\ \tau$
$\mid \text{-} \quad\quad \Rightarrow invalid\ \tau)$

definition $\mathit{deref\text{-}oid}_{OclAny}$ :: $(\mathfrak{A}\ \mathit{state} \times \mathfrak{A}\ \mathit{state} \Rightarrow \mathfrak{A}\ \mathit{state})$
$\Rightarrow (\mathit{type}_{OclAny} \Rightarrow (\mathfrak{A}, {}'c{::}\mathit{null})\mathit{val})$
$\Rightarrow \mathit{oid}$
$\Rightarrow (\mathfrak{A}, {}'c{::}\mathit{null})\mathit{val}$
where $\mathit{deref\text{-}oid}_{OclAny}\ \mathit{fst\text{-}snd}\ f\ \mathit{oid} = (\lambda\tau.\ \mathit{case}\ (\mathit{heap}\ (\mathit{fst\text{-}snd}\ \tau))\ \mathit{oid}\ \mathit{of}$
$\lfloor \mathit{in}_{OclAny}\ \mathit{obj} \rfloor \Rightarrow f\ \mathit{obj}\ \tau$
$\mid \text{-} \quad \Rightarrow \mathit{invalid}\ \tau)$

pointer undefined in state or not referencing a type conform object representation

definition $\mathit{select}_{OclAny}\mathcal{ANY}\ f = (\lambda\ X.\ \mathit{case}\ X\ \mathit{of}$
$(\mathit{mk}_{OclAny}\ \text{-}\ \bot) \Rightarrow \mathit{null}$
$\mid (\mathit{mk}_{OclAny}\ \text{-}\ \lfloor \mathit{any} \rfloor) \Rightarrow f\ (\lambda x\ \text{-}.\ \lfloor\lfloor x \rfloor\rfloor)\ \mathit{any})$

definition $\mathit{select}_{Person}\mathcal{BOSS}\ f = \mathit{select\text{-}object}\ \mathit{mtSet}\ \mathit{UML\text{-}Set.OclIncluding}\ \mathit{OclANY}\ (f\ (\lambda x\ \text{-}.\ \lfloor\lfloor x \rfloor\rfloor))$

definition $\mathit{select}_{Person}\mathcal{SALARY}\ f = (\lambda\ X.\ \mathit{case}\ X\ \mathit{of}$
$(\mathit{mk}_{Person}\ \text{-}\ \bot) \Rightarrow \mathit{null}$
$\mid (\mathit{mk}_{Person}\ \text{-}\ \lfloor \mathit{salary} \rfloor) \Rightarrow f\ (\lambda x\ \text{-}.\ \lfloor\lfloor x \rfloor\rfloor)\ \mathit{salary})$

definition $\mathit{deref\text{-}assocs}_2\mathcal{BOSS}\ \mathit{fst\text{-}snd}\ f = (\lambda\ \mathit{mk}_{Person}\ \mathit{oid}\ \text{-} \Rightarrow$
$\mathit{deref\text{-}assocs}_2\ \mathit{fst\text{-}snd}\ \mathit{switch}_2\text{-}1\ \mathit{oid}_{Person}\mathcal{BOSS}\ f\ \mathit{oid})$

definition $\mathit{in\text{-}pre\text{-}state} = \mathit{fst}$
definition $\mathit{in\text{-}post\text{-}state} = \mathit{snd}$

definition $\mathit{reconst\text{-}basetype} = (\lambda\ \mathit{convert}\ x.\ \mathit{convert}\ x)$

definition $\mathit{dot}_{OclAny}\mathcal{ANY}$ :: $\mathit{OclAny} \Rightarrow \text{-}\ \ ((1(\text{-}).\mathit{any})\ 50)$
where $(X).\mathit{any} = \mathit{eval\text{-}extract}\ X$
$(\mathit{deref\text{-}oid}_{OclAny}\ \mathit{in\text{-}post\text{-}state}$
$(\mathit{select}_{OclAny}\mathcal{ANY}$
$\mathit{reconst\text{-}basetype}))$

definition $\mathit{dot}_{Person}\mathcal{BOSS}$ :: $\mathit{Person} \Rightarrow \mathit{Person}\ \ ((1(\text{-}).\mathit{boss})\ 50)$
where $(X).\mathit{boss} = \mathit{eval\text{-}extract}\ X$
$(\mathit{deref\text{-}oid}_{Person}\ \mathit{in\text{-}post\text{-}state}$
$(\mathit{deref\text{-}assocs}_2\mathcal{BOSS}\ \mathit{in\text{-}post\text{-}state}$
$(\mathit{select}_{Person}\mathcal{BOSS}$
$(\mathit{deref\text{-}oid}_{Person}\ \mathit{in\text{-}post\text{-}state}))))$

definition $\mathit{dot}_{Person}\mathcal{SALARY}$ :: $\mathit{Person} \Rightarrow \mathit{Integer}\ \ ((1(\text{-}).\mathit{salary})\ 50)$
where $(X).\mathit{salary} = \mathit{eval\text{-}extract}\ X$
$(\mathit{deref\text{-}oid}_{Person}\ \mathit{in\text{-}post\text{-}state}$
$(\mathit{select}_{Person}\mathcal{SALARY}$
$\mathit{reconst\text{-}basetype}))$

definition $\mathit{dot}_{OclAny}\mathcal{ANY}\text{-}\mathit{at\text{-}pre}$ :: $\mathit{OclAny} \Rightarrow \text{-}\ \ ((1(\text{-}).\mathit{any@pre})\ 50)$
where $(X).\mathit{any@pre} = \mathit{eval\text{-}extract}\ X$
$(\mathit{deref\text{-}oid}_{OclAny}\ \mathit{in\text{-}pre\text{-}state}$

(*select*$_{OclAny}\mathscr{ANY}$
reconst-basetype))

definition *dot*$_{Person}\mathscr{BOSS}$*-at-pre*:: *Person* $\Rightarrow$ *Person* ((*1*(-).*boss@pre*) *50*)
where (X).*boss@pre* = *eval-extract X*
(*deref-oid*$_{Person}$ *in-pre-state*
(*deref-assocs*$_2\mathscr{BOSS}$ *in-pre-state*
(*select*$_{Person}\mathscr{BOSS}$
(*deref-oid*$_{Person}$ *in-pre-state*))))

definition *dot*$_{Person}\mathscr{SALARY}$*-at-pre*:: *Person* $\Rightarrow$ *Integer* ((*1*(-).*salary@pre*) *50*)
where (X).*salary@pre* = *eval-extract X*
(*deref-oid*$_{Person}$ *in-pre-state*
(*select*$_{Person}\mathscr{SALARY}$
reconst-basetype))

lemmas *dot-accessor* =
dot$_{OclAny}\mathscr{ANY}$*-def*
dot$_{Person}\mathscr{BOSS}$*-def*
dot$_{Person}\mathscr{SALARY}$*-def*
dot$_{OclAny}\mathscr{ANY}$*-at-pre-def*
dot$_{Person}\mathscr{BOSS}$*-at-pre-def*
dot$_{Person}\mathscr{SALARY}$*-at-pre-def*

Context Passing

lemmas [*simp*] = *eval-extract-def*

lemma *cp-dot*$_{OclAny}\mathscr{ANY}$: $((X).\mathit{any})\ \tau = ((\lambda\text{-}.\ X\ \tau).\mathit{any})\ \tau$
lemma *cp-dot*$_{Person}\mathscr{BOSS}$: $((X).\mathit{boss})\ \tau = ((\lambda\text{-}.\ X\ \tau).\mathit{boss})\ \tau$
lemma *cp-dot*$_{Person}\mathscr{SALARY}$: $((X).\mathit{salary})\ \tau = ((\lambda\text{-}.\ X\ \tau).\mathit{salary})\ \tau$

lemma *cp-dot*$_{OclAny}\mathscr{ANY}$*-at-pre*: $((X).\mathit{any@pre})\ \tau = ((\lambda\text{-}.\ X\ \tau).\mathit{any@pre})\ \tau$
lemma *cp-dot*$_{Person}\mathscr{BOSS}$*-at-pre*: $((X).\mathit{boss@pre})\ \tau = ((\lambda\text{-}.\ X\ \tau).\mathit{boss@pre})\ \tau$
lemma *cp-dot*$_{Person}\mathscr{SALARY}$*-at-pre*: $((X).\mathit{salary@pre})\ \tau = ((\lambda\text{-}.\ X\ \tau).\mathit{salary@pre})\ \tau$

lemmas *cp-dot*$_{OclAny}\mathscr{ANY}$*-I* [*simp*, *intro*!]=
cp-dot$_{OclAny}\mathscr{ANY}$[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]
lemmas *cp-dot*$_{OclAny}\mathscr{ANY}$*-at-pre-I* [*simp*, *intro*!]=
cp-dot$_{OclAny}\mathscr{ANY}$*-at-pre*[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

lemmas *cp-dot*$_{Person}\mathscr{BOSS}$*-I* [*simp*, *intro*!]=
cp-dot$_{Person}\mathscr{BOSS}$[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]
lemmas *cp-dot*$_{Person}\mathscr{BOSS}$*-at-pre-I* [*simp*, *intro*!]=
cp-dot$_{Person}\mathscr{BOSS}$*-at-pre*[*THEN allI*[*THEN allI*],
of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

lemmas *cp-dot*$_{Person}\mathscr{SALARY}$*-I* [*simp*, *intro*!]=
cp-dot$_{Person}\mathscr{SALARY}$[*THEN allI*[*THEN allI*],

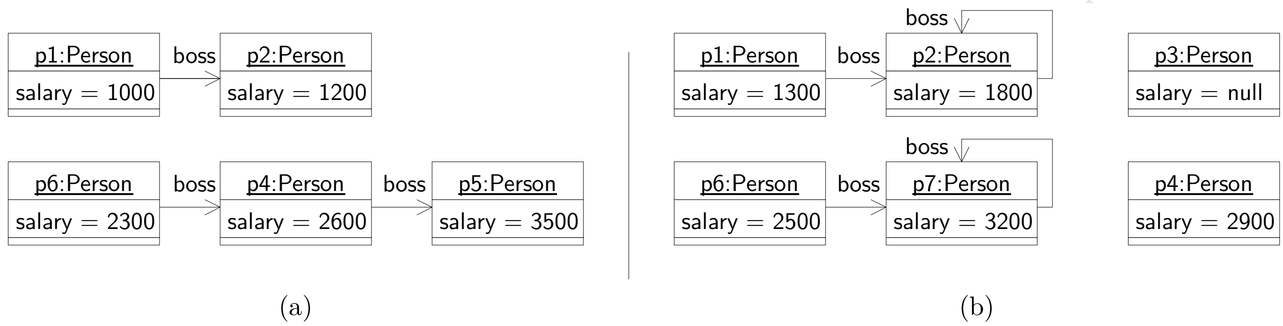


Figure A.3.: (a) pre-state σ_1 and (b) post-state σ_1'.

of λ *X -. X* λ *-* τ*.* τ, *THEN cpI1*]
lemmas *cp-dot*$_{Person}\mathscr{SALARY}$*-at-pre-I* [*simp, intro*!]=
cp-dot$_{Person}\mathscr{SALARY}$*-at-pre*[*THEN allI*[*THEN allI*],
of λ *X -. X* λ *-* τ*.* τ, *THEN cpI1*]

Execution with Invalid or Null as Argument

lemma $dot_{OclAny}\mathscr{ANY}$*-nullstrict* [*simp*]: (*null*).*any* = *invalid*
lemma $dot_{OclAny}\mathscr{ANY}$*-at-pre-nullstrict* [*simp*] : (*null*).*any*@*pre* = *invalid*
lemma $dot_{OclAny}\mathscr{ANY}$*-strict* [*simp*] : (*invalid*).*any* = *invalid*
lemma $dot_{OclAny}\mathscr{ANY}$*-at-pre-strict* [*simp*] : (*invalid*).*any*@*pre* = *invalid*

lemma $dot_{Person}\mathscr{BOSS}$*-nullstrict* [*simp*]: (*null*).*boss* = *invalid*
lemma $dot_{Person}\mathscr{BOSS}$*-at-pre-nullstrict* [*simp*] : (*null*).*boss*@*pre* = *invalid*
lemma $dot_{Person}\mathscr{BOSS}$*-strict* [*simp*] : (*invalid*).*boss* = *invalid*
lemma $dot_{Person}\mathscr{BOSS}$*-at-pre-strict* [*simp*] : (*invalid*).*boss*@*pre* = *invalid*

lemma $dot_{Person}\mathscr{SALARY}$*-nullstrict* [*simp*]: (*null*).*salary* = *invalid*
lemma $dot_{Person}\mathscr{SALARY}$*-at-pre-nullstrict* [*simp*] : (*null*).*salary*@*pre* = *invalid*
lemma $dot_{Person}\mathscr{SALARY}$*-strict* [*simp*] : (*invalid*).*salary* = *invalid*
lemma $dot_{Person}\mathscr{SALARY}$*-at-pre-strict* [*simp*] : (*invalid*).*salary*@*pre* = *invalid*

A.7.9. A Little Infra-structure on Example States

The example we are defining in this section comes from the figure A.3.

definition
$\sigma_1 \equiv (\!|\ heap = empty(oid0 \mapsto in_{Person}\ (mk_{Person}\ oid0\ \llcorner 1000 \lrcorner))$
$(oid1 \mapsto in_{Person}\ (mk_{Person}\ oid1\ \llcorner 1200 \lrcorner))$
$(*oid2*)$
$(oid3 \mapsto in_{Person}\ (mk_{Person}\ oid3\ \llcorner 2600 \lrcorner))$
$(oid4 \mapsto in_{Person}\ person5)$
$(oid5 \mapsto in_{Person}\ (mk_{Person}\ oid5\ \llcorner 2300 \lrcorner))$
$(*oid6*)$
$(*oid7*)$

$(oid8 \mapsto in_{Person}\ person9)$,
$assocs = empty(oid_{Person}\mathscr{BOSS} \mapsto [[[oid0],[oid1]],[[oid3],[oid4]],[[oid5],[oid3]]])\ \rrparenthesis$

definition
$\sigma_1{}' \equiv \llparenthesis\ heap = empty(oid0 \mapsto in_{Person}\ person1)$
$(oid1 \mapsto in_{Person}\ person2)$
$(oid2 \mapsto in_{Person}\ person3)$
$(oid3 \mapsto in_{Person}\ person4)$
$(*oid4*)$
$(oid5 \mapsto in_{Person}\ person6)$
$(oid6 \mapsto in_{OclAny}\ person7)$
$(oid7 \mapsto in_{OclAny}\ person8)$
$(oid8 \mapsto in_{Person}\ person9)$,
$assocs = empty(oid_{Person}\mathscr{BOSS} \mapsto [[[oid0],[oid1]],[[oid1],[oid1]],[[oid5],[oid6]],[[oid6],[oid6]]])\ \rrparenthesis$

definition $\sigma_0 \equiv \llparenthesis\ heap = empty, assocs = empty\ \rrparenthesis$

lemma *basic-τ-wff*: $WFF(\sigma_1,\sigma_1{}')$

lemma [*simp,code-unfold*]: $dom\ (heap\ \sigma_1) = \{oid0,oid1,(*,oid2*)oid3,oid4,oid5(*,oid6,oid7*),oid8\}$

lemma [*simp,code-unfold*]: $dom\ (heap\ \sigma_1{}') = \{oid0,oid1,oid2,oid3,(*,oid4*)oid5,oid6,oid7,oid8\}$
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 $(\sigma_1,\sigma_1{}') \models (X_{Person}1\ .oclIsMaintained())$

lemma $\bigwedge s_{pre}\ s_{post}.\ (s_{pre},s_{post}) \models ((X_{Person}1\ .oclAsType(OclAny)\ .oclAsType(Person)) \doteq X_{Person}1)$
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 $(\sigma_1,\sigma_1{}') \models (X_{Person}2\ .oclIsMaintained())$

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 $(\sigma_1,\sigma_1{}') \models (X_{Person}3\ .oclIsNew())$

lemma $(\sigma_1,\sigma_1{}') \models (X_{Person}4\ .oclIsMaintained())$

Assert $\bigwedge s_{pre}\ .\ (s_{pre},\sigma_1{}') \models not(\upsilon(X_{Person}5\ .salary))$

Assert $\bigwedge \quad s_{post}. \quad (\sigma_1, s_{post}) \models \quad (X_{Person}5 \ .salary@pre \ \doteq \mathbf{3500})$

lemma $\quad (\sigma_1, \sigma_1{}') \models \quad (X_{Person}5 \ .oclIsDeleted())$

lemma $\quad (\sigma_1, \sigma_1{}') \models \quad (X_{Person}6 \ .oclIsMaintained())$

Assert $\bigwedge s_{pre} \ s_{post}. \quad (s_{pre}, s_{post}) \models \quad \upsilon(X_{Person}7 \ .oclAsType(Person))$

lemma $\bigwedge s_{pre} \ s_{post}. \quad (s_{pre}, s_{post}) \models \quad ((X_{Person}7 \ .oclAsType(Person) \ .oclAsType(OclAny)$
$.oclAsType(Person))$
$\doteq (X_{Person}7 \ .oclAsType(Person)))$
lemma $\quad (\sigma_1, \sigma_1{}') \models \quad (X_{Person}7 \ .oclIsNew())$

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*: $(\sigma_1, \sigma_1{}') \models (Set\{ \ X_{Person}1 \ .oclAsType(OclAny)$
$, X_{Person}2 \ .oclAsType(OclAny)$
$(*, X_{Person}3 \ .oclAsType(OclAny)*)$
$, X_{Person}4 \ .oclAsType(OclAny)$
$(*, X_{Person}5 \ .oclAsType(OclAny)*)$
$, X_{Person}6 \ .oclAsType(OclAny)$
$(*, X_{Person}7 \ .oclAsType(OclAny)*)$
$(*, X_{Person}8 \ .oclAsType(OclAny)*)$
$(*, X_{Person}9 \ .oclAsType(OclAny)*)\} {-}{>} oclIsModifiedOnly())$

lemma $(\sigma_1, \sigma_1{}') \models ((X_{Person}9 \ @pre \ (\lambda x. \ \llcorner OclAsType_{Person}\text{-}\mathfrak{A} \ x \lrcorner)) \ \triangleq X_{Person}9)$

lemma $(\sigma_1, \sigma_1{}') \models ((X_{Person}9 \ @post \ (\lambda x. \ \llcorner OclAsType_{Person}\text{-}\mathfrak{A} \ x \lrcorner)) \ \triangleq X_{Person}9)$

lemma $(\sigma_1, \sigma_1{}') \models (((X_{Person}9 \ .oclAsType(OclAny)) \ @pre \ (\lambda x. \ \llcorner OclAsType_{OclAny}\text{-}\mathfrak{A} \ x \lrcorner)) \triangleq$
$((X_{Person}9 \ .oclAsType(OclAny)) \ @post \ (\lambda x. \ \llcorner OclAsType_{OclAny}\text{-}\mathfrak{A} \ x \lrcorner)))$

lemma *perm*-$\sigma_1{}'$: $\sigma_1{}' = (\!| \ heap = empty$
$(oid8 \mapsto in_{Person} \ person9)$
$(oid7 \mapsto in_{OclAny} \ person8)$
$(oid6 \mapsto in_{OclAny} \ person7)$
$(oid5 \mapsto in_{Person} \ person6)$
$(*oid4*)$
$(oid3 \mapsto in_{Person} \ person4)$
$(oid2 \mapsto in_{Person} \ person3)$
$(oid1 \mapsto in_{Person} \ person2)$
$(oid0 \mapsto in_{Person} \ person1)$

$, assocs = assocs\ \sigma_1{}' \rparr$

declare *const-ss* [*simp*]

lemma $\bigwedge\sigma_1.$
$(\sigma_1,\sigma_1{}') \models (Person\ .allInstances() \doteq Set\{\ X_{Person}1, X_{Person}2, X_{Person}3, X_{Person}4(*, X_{Person}5*), X_{Person}6,$
$X_{Person}7\ .oclAsType(Person)(*, X_{Person}8*), X_{Person}9\ \})$

lemma $\bigwedge\sigma_1.$
$(\sigma_1,\sigma_1{}') \models (OclAny\ .allInstances() \doteq Set\{\ X_{Person}1\ .oclAsType(OclAny), X_{Person}2\ .oclAsType(OclAny),$
$X_{Person}3\ .oclAsType(OclAny), X_{Person}4\ .oclAsType(OclAny)$
$(*, X_{Person}5*), X_{Person}6\ .oclAsType(OclAny),$
$X_{Person}7, X_{Person}8, X_{Person}9\ .oclAsType(OclAny)\ \})$

A.7.10. OCL Part: Standard State Infrastructure

Ideally, these definitions are automatically generated from the class model.

A.7.11. Invariant

These recursive predicates can be defined conservatively by greatest fix-point constructions—automatically. See [3, 4] 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 $Person\text{-}label_{globalinv} \equiv (Person\ .allInstances()->forAll_{Set}(x \mid Person\text{-}label_{inv}\ (x))\ and$
$(Person\ .allInstances@pre()->forAll_{Set}(x \mid Person\text{-}label_{invATpre}\ (x))))$

lemma $\tau \models \delta\ (X\ .boss) \Longrightarrow \tau \models Person\ .allInstances()->includes_{Set}(X\ .boss) \land$
$\tau \models Person\ .allInstances()->includes_{Set}(X)$

lemma $REC\text{-}pre : \tau \models Person\text{-}label_{globalinv}$
$\Longrightarrow \tau \models Person\ .allInstances()->includes_{Set}(X)$ $(*$ *X represented object in state* $*)$
$\Longrightarrow \exists\ REC.\ \ \tau \models REC(X)\ \triangleq (Person\text{-}label_{inv}\ (X)\ and\ (X\ .boss <> null\ implies\ REC(X\ .boss)))$

This allows to state a predicate:

axiomatization $inv_{Person\text{-}label}$ :: *Person* $\Rightarrow$ *Boolean*
where $inv_{Person\text{-}label}$-*def*:

$$(\tau \models Person\ .allInstances() -{>}includes_{Set}(self)) \Longrightarrow$$
$$(\tau \models (inv_{Person\text{-}label}(self) \triangleq\ (self\ .boss <> null\ implies$$
$$(self\ .salary\ \leq_{int}\ ((self\ .boss)\ .salary))\ and$$
$$inv_{Person\text{-}label}(self\ .boss))))$$

axiomatization $inv_{Person\text{-}labelAT\,pre}$ :: *Person* $\Rightarrow$ *Boolean*
where $inv_{Person\text{-}labelAT\,pre}$-*def*:

$$(\tau \models Person\ .allInstances@pre() -{>}includes_{Set}(self)) \Longrightarrow$$
$$(\tau \models (inv_{Person\text{-}labelAT\,pre}(self) \triangleq (self\ .boss@pre <> null\ implies$$
$$(self\ .salary@pre\ \leq_{int}\ ((self\ .boss@pre)\ .salary@pre))\ and$$
$$inv_{Person\text{-}labelAT\,pre}(self\ .boss@pre))))$$

lemma *inv-1* :

$$(\tau \models Person\ .allInstances() -{>}includes_{Set}(self)) \Longrightarrow$$
$$(\tau \models inv_{Person\text{-}label}(self) = ((\tau \models (self\ .boss \doteq null)) \vee$$
$$(\ \tau \models (self\ .boss <> null) \wedge$$
$$\tau \models ((self\ .salary)\ \leq_{int}\ (self\ .boss\ .salary))\ \wedge$$
$$\tau \models (inv_{Person\text{-}label}(self\ .boss)))))$$

lemma *inv-2* :

$$(\tau \models Person\ .allInstances@pre() -{>}includes_{Set}(self)) \Longrightarrow$$
$$(\tau \models inv_{Person\text{-}labelAT\,pre}(self)) =\ ((\tau \models (self\ .boss@pre \doteq null)) \vee$$
$$(\tau \models (self\ .boss@pre <> null) \wedge$$
$$(\tau \models (self\ .boss@pre\ .salary@pre \leq_{int} self\ .salary@pre))\ \wedge$$
$$(\tau \models (inv_{Person\text{-}labelAT\,pre}(self\ .boss@pre)))))$$

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

coinductive *inv* :: *Person* $\Rightarrow$ $(\mathfrak{A})st \Rightarrow bool$ **where**

$$(\tau \models (\delta\ self)) \Longrightarrow ((\tau \models (self\ .boss \doteq null)) \vee$$
$$(\tau \models (self\ .boss <> null) \wedge (\tau \models (self\ .boss\ .salary \leq_{int} self\ .salary))\ \wedge$$
$$(\ (inv(self\ .boss))\tau\ )))$$
$$\Longrightarrow (\ inv\ self\ \tau)$$

A.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* $\Rightarrow$ *Set-Integer* $((1(\text{-}).contents'(')) \; 50)$
where *contents-def*:

$$
\begin{array}{l}
(self\ .contents()) = (\lambda\ \tau.\ (if\ \tau \models (\delta\ self) \\
\quad then\ SOME\ res.((\tau \models true) \wedge \\
\qquad (\tau \models (\lambda\text{-}\ .\ res) \triangleq if\ (self\ .boss \doteq null) \\
\qquad\qquad then\ (Set\{self\ .salary\}) \\
\qquad\qquad else\ (self\ .boss\ .contents() \\
\qquad\qquad\quad -{>}including_{Set}(self\ .salary)) \\
\qquad\qquad endif)) \\
\quad else\ invalid\ \tau))
\end{array}
$$

and *cp0-contents*: $(X\ .contents())\ \tau = ((\lambda\text{-}.\ X\ \tau)\ .contents())\ \tau$

interpretation *contents* : *contract0 contents* λ *self. true*

$$
\begin{array}{l}
\lambda\ self\ res.\ \ res \triangleq if\ (self\ .boss \doteq null) \\
\qquad then\ (Set\{self\ .salary\}) \\
\qquad else\ (self\ .boss\ .contents() \\
\qquad\quad -{>}including_{Set}(self\ .salary)) \\
\qquad endif
\end{array}
$$

Specializing $[\![cp\ E;\ \tau \models \delta\ self;\ \tau \models true;\ \tau \models POST'\ self;\ \bigwedge res.\ (res \triangleq if\ self.boss \doteq null\ then\ Set\{self.salary\}\ else\ self.boss.contents() -{>}including_{Set}(self.salary)\ endif) = (POST'\ self\ and\ (res \triangleq BODY\ self))]\!] \Longrightarrow (\tau \models E\ (self.contents())) = (\tau \models E\ (BODY\ self))$, 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 $(\tau \models E\ (self\ .contents())) =$

$$
\begin{array}{l}
(\tau \models E\ (if\ self\ .boss \doteq null \\
\quad then\ Set\{self\ .salary\} \\
\quad else\ self\ .boss\ .contents() -{>}including_{Set}(self\ .salary)\ endif))
\end{array}
$$

Since we have only one interpretation function, we need the corresponding operation on the pre-state:

consts *contentsATpre* :: *Person* $\Rightarrow$ *Set-Integer* $((1(\text{-}).contents@pre'(')) \; 50)$

axiomatization where *contentsATpre-def*:

$$
\begin{array}{l}
(self).contents@pre() = (\lambda\ \tau. \\
\quad (if\ \tau \models (\delta\ self) \\
\quad\ then\ SOME\ res.((\tau \models true) \wedge \qquad\qquad (*\ pre\ *) \\
\qquad (\tau \models ((\lambda\text{-}.\ res) \triangleq if\ (self).boss@pre \doteq null\ \ (*\ post\ *) \\
\qquad\qquad then\ Set\{(self).salary@pre\} \\
\qquad\qquad else\ (self).boss@pre\ .contents@pre() \\
\qquad\qquad\quad -{>}including_{Set}(self\ .salary@pre) \\
\qquad\qquad endif))) \\
\quad\ else\ invalid\ \tau))
\end{array}
$$

and *cp0-contents-at-pre*: $(X\ .contents@pre())\ \tau = ((\lambda\text{-}.\ X\ \tau)\ .contents@pre())\ \tau$

interpretation *contentsATpre* : *contract0 contentsATpre* λ *self. true*

$$
\begin{array}{l}
\lambda\ self\ res.\ \ res \triangleq if\ (self\ .boss@pre \doteq null) \\
\qquad then\ (Set\{self\ .salary@pre\}) \\
\qquad else\ (self\ .boss@pre\ .contents@pre()
\end{array}
$$

$\qquad\qquad -\!\!>\!including_{Set}(self\ .salary@pre))$
$\qquad endif$

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 $(\tau \models E\ (self\ .contents@pre())) =$
$\quad(\tau \models E\ (if\ self\ .boss@pre \doteq null$
$\qquad then\ Set\{self\ .salary@pre\}$
$\qquad else\ self\ .boss@pre\ .contents@pre() -\!\!>\!including_{Set}(self\ .salary@pre)\ endif))$

Note that these `@pre` variants on methods are only available on queries, i. e., operations without side-effect.

A.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*(-).*insert*′(-′)) *50*)
where $self\ .insert(x) \equiv$
$\quad(\lambda\ \tau.\ if\ (\tau \models (\delta\ self)) \wedge (\tau \models \upsilon\ x)$
$\qquad then\ SOME\ res.\ (\tau \models true\ \wedge$
$\qquad\qquad (\tau \models ((self).contents() \triangleq (self).contents@pre() -\!\!>\!including_{Set}(x))))$
$\qquad else\ invalid\ \tau)$

The semantic consequences of this definition were computed inside this locale interpretation:

interpretation *insert* : *contract1 insert* $\lambda\ self\ x.\ true$
$\qquad\lambda\ self\ x\ res.\ ((self\ .contents()) \triangleq$
$\qquad\qquad (self\ .contents@pre() -\!\!>\!including_{Set}(x)))$

The result of this locale interpretation for our *Analysis-OCL.insert* contract is the following set of properties, which serves as basis for automated deduction on them:

Name	Theorem
insert.strict0	$(invalid.insert(X)) = invalid$
insert.nullstrict0	$(null.insert(X)) = invalid$
insert.strict1	$(self.insert(invalid)) = invalid$
insert.cp$_{PRE}$	$true\ \tau = true\ \tau$
insert.cp$_{POST}$	$(self.contents() \triangleq self.contents@pre()->including_{Set}(a1.0))\ \tau = (\lambda\text{-}.\ self\ \tau.contents() \triangleq \lambda\text{-}.\ self\ \tau.contents@pre()->including_{Set}(\lambda\text{-}.\ a1.0\ \tau))\ \tau$
insert.cp-pre	$[\![cp\ self'; cp\ a1']\!] \Longrightarrow cp\ (\lambda X.\ true)$
insert.cp-post	$[\![cp\ self'; cp\ a1'; cp\ res']\!] \Longrightarrow cp\ (\lambda X.\ self'\ X.contents() \triangleq self'\ X.contents@pre()->including_{Set}(a1'\ X))$
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-scheme	$self.insert(a1.0) \equiv \lambda\tau.\ if\ \tau \models \delta\ self \wedge \tau \models \upsilon\ a1.0\ then\ SOME\ res.\ \tau \models true \wedge \tau \models self.contents() \triangleq self.contents@pre()->including_{Set}(a1.0)\ else\ invalid\ \tau$
insert.unfold	$[\![cp\ E; \tau \models \delta\ self \wedge \tau \models \upsilon\ a1.0; \tau \models true; \exists res.\ \tau \models self.contents() \triangleq self.contents@pre()->including_{Set}(a1.0); \bigwedge res.\ \tau \models self.contents() \triangleq self.contents@pre()->including_{Set}(a1.0) \Longrightarrow \tau \models E\ (\lambda\text{-}.\ res)]\!] \Longrightarrow \tau \models E\ (self.insert(a1.0))$
insert.unfold2	$[\![cp\ E; \tau \models \delta\ self \wedge \tau \models \upsilon\ a1.0; \tau \models true; \tau \models POST'\ self\ a1.0; \bigwedge res.\ (self.contents() \triangleq self.contents@pre()->including_{Set}(a1.0)) = (POST'\ self\ a1.0\ and\ (res \triangleq BODY\ self\ a1.0))]\!] \Longrightarrow (\tau \models E\ (self.insert(a1.0))) = (\tau \models E\ (BODY\ self\ a1.0))$

Table A.5.: Semantic properties resulting from a user-defined operation contract.

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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[11] 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.

[12] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The amsterdam manifesto on OCL. In Clark and Warmer [11], pages 115–149. ISBN 3-540-43169-1.

[13] 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.

[14] F. Haftmann and M. Wenzel. Constructive type classes in isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 160–174. Springer, 2006. ISBN 978-3-540-74463-4. doi: 10.1007/978-3-540-74464-1_11. URL http://dx.doi.org/10.1007/978-3-540-74464-1_11.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Object Management Group. UML 2.0 OCL specification, Oct. 2003. Available as OMG document ptc/03-10-14.

[20] Object Management Group. UML 2.3.1 OCL specification, Feb. 2012. Available as OMG document formal/2012-01-01.

[21] 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.

[22] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[23] 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.

[24] M. M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, TU München, München, Feb. 2002. URL http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html.

Contents