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 extent 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 \(^1\)\(^2\).

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 \(\text{Pair}(x:\text{First}, y:\text{Second})\) stands for the usual binary pairing operator \(\text{Pair}(\text{true}, \text{null})\) and the two projection functions \(x.\text{First}()\) and \(x.\text{Second}()\). 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 \(\text{Pair}(x, y).\text{First}() = x\) and \(\text{Pair}(x, y).\text{Second}() = y\) to give pairings the usual semantics. At some place, the OCL Standard requires the existence of a constant symbol \(\text{invalid}\) 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 \(\text{Pair}(\text{invalid}, y) = \text{invalid}\), \(\text{Pair}(x, \text{invalid}) = \text{invalid}\), \(\text{invalid}.\text{First}() = \text{invalid}\), \(\text{invalid}.\text{Second}() = \text{invalid}\), etc. Unfortunately, this “natural” axiomatization of pairing and projection together with strictness is already inconsistent. One can derive:

\[
\text{Pair}(\text{true}, \text{invalid}).\text{First}() = \text{invalid}.\text{First}() = \text{invalid}
\]

and:

\[
\text{Pair}(\text{true}, \text{invalid}).\text{First}() = \text{true}
\]

which then results in the absurd logical consequence that \(\text{invalid} = \text{true}\). Obviously, we need to be more careful on the side-conditions of our rules\(^3\). 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\footnote{As side-effect free language, OCL has no object-constructors, but with \texttt{OclIsNew()}, the effect of object creation can be expressed in a declarative way.} 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 \texttt{oclIsTypeOf(T)} one to the other, and under particular conditions to be described in detail later, these casts are semantically *lossless*, i.e.

\[(X.oclAsType(C_j).oclAsType(C_i) = X)\]  \hspace{1cm} (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 \texttt{Boolean}, \texttt{Void}, \texttt{Integer}, \texttt{Real} and \texttt{String} as well as generic types such as \texttt{Pair(T,T’)}, \texttt{Sequence(T)} and \texttt{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 \texttt{null} and \texttt{invalid}; since this extends to \texttt{Boolean} type, this results in a four-valued logic. Consequently, Featherweight OCL contains the derivation of the *logic* of OCL.
- collection types may contain \texttt{null} (so \texttt{Set\{null\}} is a defined set) but not \texttt{invalid} (\texttt{Set\{invalid\}} is just \texttt{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., \texttt{oclAsType(Class)}).\footnote{The details of such a pre-processing are described in \cite{3}.}
- Featherweight OCL types may be arbitrarily nested. For example, the expression \texttt{Set\{Set\{1,2\}\}} = \texttt{Set\{Set\{2,1\}\}} is legal and true.
- All objects types are represented in an object universe\footnote{following the tradition of HOL-OCL \cite{5}}. The universe construction also gives semantics to type casts, dynamic type tests, as well as functions such as \texttt{allInstances()}, or \texttt{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 \cite{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.\footnote{following the tradition of HOL-OCL \cite{5}}

\[^3\]As side-effect free language, OCL has no object-constructors, but with \texttt{OclIsNew()}, the effect of object creation can be expressed in a declarative way. 
\[^4\]The details of such a pre-processing are described in \cite{3}. 
\[^5\]following the tradition of HOL-OCL \cite{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 _⇒_ allowing to form constructs like $A_1 \Rightarrow \cdots \Rightarrow A_n \Rightarrow A_{n+1}$, 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

$$[A_1; \ldots ; A_n] \Rightarrow A_{n+1}$$

or, in mathematical notation,

$$\frac{A_1 \quad \cdots \quad A_n}{A_{n+1}}.$$

(A.2)

The built-in meta-level quantification $\forall x. x$ 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:

$$\forall x_1, \ldots , x_m. [A_1; \ldots ; A_n] \Rightarrow A_{n+1}.$$

(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 $\phi$, using the Isar [24] language, will look as follows in Isabelle:

\begin{verbatim}
lemma label: \phi
  apply(case_tac)
  apply(simp_all)
  done
\end{verbatim}

This proof script instructs Isabelle to prove $\phi$ 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 $\phi$. Proof states were usually denoted by:

\begin{verbatim}
label : $\phi$
1. $\phi_1$
  ...
  $n$. $\phi_n$
\end{verbatim}

Subgoals and goals may be extracted from the proof state into theorems of the form $[\phi_1; \ldots; \phi_n] \Rightarrow \phi$ 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 $\land$, $\lor$, $\rightarrow$, $\neg$ as well as the object-logical quantifiers $\forall x. P\,x$ and $\exists x. P\,x$; in contrast to first-order logic, quantifiers may range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centered around extensional equality $=_:: \alpha \Rightarrow \alpha \Rightarrow \text{bool}$. HOL is more expressive than first-order logic, since, e.g., induction schemes can be expressed inside the logic. Being based on some polymorphically typed $\lambda$-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 $\tau_\bot := \bot | \tau_{\bot}$ that assigns to each type $\tau$ a type $\tau_\bot$ disjointly extended by the exceptional element $\bot$. The function $\tau_{\bot} : \alpha_{\bot} \rightarrow \alpha$ is the inverse of $\tau_{\bot}$ (unspecified for $\bot$). Partial functions $\alpha \rightarrow \beta$ 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:

\[
\begin{align*}
\text{types} & \quad \alpha \text{ set} = \alpha \Rightarrow \text{bool} \\
\text{definition} & \quad \text{Collect} :: (\alpha \Rightarrow \text{bool}) \Rightarrow \alpha \text{ set} \quad \text{— set comprehension} \\
\text{where} & \quad \text{Collect} S \equiv S \\
\text{definition} & \quad \text{member} :: \alpha \Rightarrow \alpha \Rightarrow \text{bool} \quad \text{— membership test} \\
\text{where} & \quad \text{member} s S \equiv S s
\end{align*}
\] (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 \(\_ \cup \_\) 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:

\[
\begin{align*}
\text{datatype} & \quad \text{option} = \text{None} \mid \text{Some} \alpha \\
\text{datatype} & \quad \alpha \text{ list} = \text{Nil} \mid \text{Cons} a l
\end{align*}
\] (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

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

respectively

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

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

\[
\begin{align*}
(\text{case } [\ ] \Rightarrow F \mid (a\#r) \Rightarrow G a r) & = F \\
(\text{case } b\#t \Rightarrow F \mid (a\#r) \Rightarrow G a r) & = G b t \\
[\ ] & \neq a\#t \\
[ a = [\ ] P ; \exists x t. a = x\#t \Rightarrow P ] & = P \\
[ P ] ; \forall a t. P t \Rightarrow P (a\#t) & \Rightarrow P x
\end{align*}
\] (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:

\[
\begin{align*}
\text{fun} & \quad \text{ins} :: [\alpha :: \text{linorder}, \alpha \text{ list }] \Rightarrow \alpha \text{ list} \\
\text{where} & \quad \text{ins } x [\ ] = [x] \\
& \quad \text{ins } x (y\#ys) = \text{if } x < y \text{ then } x\#y\#ys \text{ else } y\#(\text{ins } x\#s) \\
\text{fun} & \quad \text{sort} :: (\alpha :: \text{linorder}) \text{ list } \Rightarrow \alpha \text{ list} \\
\text{where} & \quad \text{sort } [\ ] = [\ ] \\
& \quad \text{sort } (x\#xs) = \text{ins } x (\text{sort } xs)
\end{align*}
\] (A.11, A.12)

\(^6\)To increase readability, we use a slightly simplified presentation.
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., \texttt{not (not x)} = \texttt{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 \( \sigma \) to post-state \( \sigma' \), validity statements were written \( (\sigma, \sigma') \models P \). Its major purpose is to logically establish facts (lemmas and theorems) about the denotational definitions. The third layer, called algebraic layer, also derived from the former layers, tries to establish algebraic laws of the form \( P = P' \); such laws are amenable to equational reasoning and also help for automated reasoning and code-generation. For an implementor of an OCL compiler, these consequences are of most interest.

For space reasons, we will restrict ourselves in this 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:

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

Types were directly represented in 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:

\[
\text{type synonym } \text{Boolean}_{\text{base}} := \text{bool} \downarrow \bot
\]  

(A.13)

As a consequence of this definition of the type, we have the elements \( \bot, \bot^\top, \text{true}^\top, \text{false}^\top \) 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 \( \bot^\top \) 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 \( \mathcal{A} \) 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 \((\sigma, \sigma')\) of pre-and post-state. Thus, we define valuation types by the synonym:

\[
\text{type synonym} \quad V_{\mathcal{A}}(\alpha) := \text{state}(\mathcal{A}) \times \text{state}(\mathcal{A}) \rightarrow \alpha :: \text{null} . \tag{A.14}
\]

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

\[
\begin{align*}
\text{type synonym} \quad \text{Boolean}_{\mathcal{A}} & := V_{\mathcal{A}}(\text{Boolean}_{\text{base}}) \\
\text{type synonym} \quad \text{Integer}_{\mathcal{A}} & := V_{\mathcal{A}}(\text{Integer}_{\text{base}})
\end{align*}
\]

... the other cases are analogous. In the subsequent subsections, we will drop the index \( \mathcal{A} \) 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 \( I(E) \) for the semantic interpretation function as commonly used in mathematical textbooks and the variable \( \tau \) standing for pairs of pre- and post state \((\sigma, \sigma')\). OCL provides for all OCL types the constants invalid for the exceptional computation result and null for the non-existing value. Thus we define:

\[
I[\text{invalid}]:V(\alpha) \equiv \text{bot} \quad I[\text{null}]:V(\alpha) \equiv \text{null}
\]

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):

\[
\begin{align*}
I[\text{true}]:\text{Boolean} & = \lambda \tau. \text{true} \\
I[\text{false}]:\text{Boolean} & = \lambda \tau. \text{false} \\
I[X.\text{oclIsUndefined}()] \tau & = (\text{if} I[X] \in \{\text{bot}, \text{null}\} \text{ then } I[\text{true}]) \text{ else } I[\text{false}] \tau \\
I[X.\text{oclIsInvalid}()] \tau & = (\text{if} I[X] \tau = \text{bot} \text{ then } I[\text{true}] \text{ else } I[\text{false}] \tau)
\end{align*}
\]

For reasons of conciseness, we will write \( \delta X \) for \( \text{not}(X.\text{oclIsUndefined}()) \) and \( \nu X \) for \( \text{not}(X.\text{oclIsInvalid}()) \) 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:

\[
I[\text{true}]:\text{Boolean} \tau = \omega \text{true} \]

we can therefore write:

\[
\text{true} : \text{Boolean} = \lambda \tau. \omega \text{true} \]

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

Since all operators of the assertion language depend on the context \( \tau = (\sigma, \sigma') \) and result in values that can be \( \bot \), all expressions can be viewed as evaluations from \((\sigma, \sigma')\) to a type \( \alpha \) which must posses a \( \bot \) 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

\[
V(\alpha) := \text{state} \times \text{state} \rightarrow \alpha :: \{\text{bot}, \text{null}\},
\]
where state stands for the system state and state × 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 _ pre which replaces, for example, all accessor functions _ .a by their counterparts _ .a@pre. For example, \((self.a > 5)_pre\) is just \((self@a@pre > 5)\). 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 \(\text{not}\) and \(\text{and}\) as follows:

\[
\begin{align*}
I[[\text{not } X]]^\tau &= (\text{case } I[[X]]^\tau \text{ of} \\
\bot &\Rightarrow \bot \\
|_{\bot} &\Rightarrow |_{\bot} \\
|_{\text{true}} &\Rightarrow |_{\text{false}} \\
|_{\text{false}} &\Rightarrow |_{\text{false}}) \\
I[[X \text{ and } Y]]^\tau &= (\text{case } I[[X]]^\tau \text{ of} \\
\bot &\Rightarrow (\text{case } I[[Y]]^\tau \text{ of} \\
\bot &\Rightarrow \bot \\
|_{\bot} &\Rightarrow |_{\bot} \\
|_{\text{true}} &\Rightarrow |_{\text{false}} \\
|_{\text{false}} &\Rightarrow |_{\text{false}}) \\
|_{\text{true}} &\Rightarrow (\text{case } I[[Y]]^\tau \text{ of} \\
\bot &\Rightarrow \bot \\
|_{\bot} &\Rightarrow |_{\bot} \\
|_{\text{true}} &\Rightarrow |_{\text{false}}) \\
|_{\text{false}} &\Rightarrow |_{\text{false}} \\
|_{\text{false}} &\Rightarrow |_{\text{false}})
\end{align*}
\]

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

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

\[
I[[x + y]]^\tau = \text{if } I[[\delta x]]^\tau = I[[\text{true}]]^\tau \land I[[\delta y]]^\tau = I[[\text{true}]]^\tau \text{ then } I[[\text{true}]]^\tau + I[[\text{true}]]^\tau \text{ else } \bot
\]

where the operator “+” on the left-hand side of the equation denotes the OCL addition of type \(\text{Integer } \Rightarrow \text{Integer } \Rightarrow \text{Integer}\) 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:

\[(\sigma, \sigma') \models P,\]
where $\sigma$ is the pre-state and $\sigma'$ 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 $(\sigma, \sigma')$ (i.e., $\tau$ for short) yields true. Formally this means:

$$\tau \models P \equiv (\langle [P] \rangle \tau = _{\text{true}}).$$

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

$$
\begin{align*}
\tau \models true & \quad \neg(\tau \models false) \quad \neg(\tau \models invalid) \quad \neg(\tau \models null) \\
\tau \models not P & \implies \neg(\tau \models P) \\
\tau \models P \land Q & \implies \tau \models P \quad \tau \models P \land Q \implies \tau \models Q \\
\tau \models P \land Q & \implies P \lor Q \\
\tau \models P & \implies (if \ P \ then \ B_1 \ else \ B_2 \ endif) \tau = B_1 \tau \\
\tau \models not P & \implies (if \ P \ then \ B_1 \ else \ B_2 \ endif) \tau = B_2 \tau \\
\tau \models P & \implies \delta P \\
\tau \models \delta X & \implies \tau \models \nu X
\end{align*}
$$

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 $x \neq y$ 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," and is at the heart of the OCL logic,
3. The symbol $\_ . \_ = \_ \_$ 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:

$$\langle [X \triangleq Y] \rangle \tau = _{\text{true}} \iff \langle [Y] \rangle \tau = _{\text{true}}.$$

It enjoys nearly the laws of a congruence:

$$
\begin{align*}
\tau \models (x \triangleq x) \\
\tau \models (x \triangleq y) & \implies \tau \models (y \triangleq x) \\
\tau \models (x \triangleq y) & \implies \tau \models (y \triangleq z) \implies \tau \models (x \triangleq z) \\
\text{cp} P & \implies (x \triangleq y) \implies \tau \models (P x) \implies \tau \models (P y)
\end{align*}
$$

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.

1 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:

\[
\tau \vdash \delta x \lor \tau \not\vdash x \equiv \text{invalid} \lor \tau \vdash x \equiv \text{null},
\]

\[
(\tau \vdash A \equiv \text{invalid}) = (\tau \not\vdash \(\text{not}(uA)\))
\]

\[
(\tau \vdash A \equiv \text{true}) = (\tau \vdash \text{not}\(\text{not}(\text{not}(uA))\))
\]

\[
(\tau \not\vdash (\delta x)) = (\text{not}(\tau \vdash \delta x))
\]

\[
(\tau \not\vdash (\text{not}(uA))) = (\text{not}(\tau \vdash uA))
\]

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]. \(\delta\)-closure rules for all logical connectives have the following format, e. g.:

\[
\tau \vdash \delta x \implies (\tau \vdash \text{not} x) = (\text{not}(\tau \vdash x))
\]

\[
\tau \vdash \delta x \implies \tau \vdash \delta y \implies (\tau \vdash x \land \tau \vdash y) = (\tau \vdash x \land \tau \vdash y)
\]

\[
\tau \vdash \delta x \implies \tau \vdash \delta y
\]

\[
\implies (\tau \vdash (x \implies y)) = ((\tau \vdash x) \implies (\tau \vdash y))
\]

Together with the already mentioned general case-distinction

\[
\tau \vdash \delta x \lor \tau \vdash x = \text{invalid} \lor \tau \vdash x = \text{null}
\]

which is possible for any OCL type, a case distinction on the variables in a formula can be performed; due to strictness rules, formulae containing somewhere a variable \(x\) that is known to be \text{invalid} or \text{null} reduce usually quickly to contradictions. For example, we can infer from an invariant \(\tau \vdash x \equiv y - 3\) that we have \(\tau \vdash x \equiv y - 3 \land \tau \vdash \delta x \land \tau \vdash \delta y\).

We call the latter formula the \(\delta\)-closure of the former. Now, we can convert a formula like \(\tau \vdash x > 0 \lor \tau \vdash 3 \ast y > x \ast x\) into the equivalent formula \(\tau \vdash x > 0 \lor \tau \vdash 3 \ast y > x \ast x\) 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” \(\delta\)-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 \(\text{not}\) and \(\text{and}\) can be re-formulated in the following ground equations:

\[
\begin{align*}
\upsilon \text{ invalid} &= \text{false} & \upsilon \text{ null} &= \text{true} \\
\upsilon \text{ true} &= \text{true} & \upsilon \text{ false} &= \text{true} \\
\delta \text{ invalid} &= \text{false} & \delta \text{ null} &= \text{false} \\
\delta \text{ true} &= \text{true} & \delta \text{ false} &= \text{true} \\
\text{not invalid} &= \text{invalid} & \text{not null} &= \text{null} \\
\text{not true} &= \text{false} & \text{not false} &= \text{true} \\
\text{(null and true)} &= \text{null} & \text{(null and false)} &= \text{false} \\
\text{(null and null)} &= \text{null} & \text{(null and invalid)} &= \text{invalid} \\
\text{(false and true)} &= \text{false} & \text{(false and false)} &= \text{false} \\
\text{(false and null)} &= \text{false} & \text{(false and invalid)} &= \text{false}
\end{align*}
\]
(true and true) = true     (true and false) = false
(true and null) = null      (true and invalid) = invalid
(invalid and true) = invalid (invalid and false) = false
(invalid and null) = invalid (invalid and invalid) = invalid

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

\[ X \land X = X \quad X \land Y = Y \land X \]
\[ \text{false and } X = \text{false} \quad X \land \text{false} = \text{false} \]
\[ \text{true and } X = X \quad X \land \text{true} = X \]
\[ X \land (Y \land Z) = X \land Y \land Z \]

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 \( \delta \)-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:

\[ \text{invalid} + X = \text{invalid} \quad X + \text{invalid} = \text{invalid} \]
\[ \text{null} \rightarrow \text{including}(X) = \text{invalid} \quad \text{null} \rightarrow \text{including}(X) = \text{invalid} \]
\[ \text{invalid} \equiv \text{invalid} = \text{invalid} \quad \text{invalid} \equiv X = \text{invalid} \]
\[ \text{S} \rightarrow \text{including}(\text{invalid}) = \text{invalid} \]
\[ X \equiv X = (\text{if } u x \text{ then true else invalid endif}) \]
\[ 1 \div 0 = \text{invalid} \quad 1 \div \text{null} = \text{invalid} \]
\[ \text{invalid} \rightarrow \text{isEmpty}() = \text{invalid} \quad \text{null} \rightarrow \text{isEmpty}() = \text{null} \]

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

\[ \delta \text{ Set}() = \text{true} \]
\[ \delta (X \rightarrow \text{including}(x)) = \delta X \land u x \]
\[ \text{Set}() \rightarrow \text{includes}(x) = (\text{if } u x \text{ then false else invalid endif}) \]
\[ (X \rightarrow \text{including}(x) \rightarrow \text{includes}(y)) = \]
\[ (\text{if } \delta X \]
\[ \text{then if} x \equiv y \]
\[ \text{then true else} X \rightarrow \text{includes}(y) \]
\[ \text{endif} \]
\[ \text{else invalid} \]
\[ \text{endif} \]

As Set\{1, 2\} is only syntactic sugar for
an expression like \( \text{Set}(1,2) \rightarrow \text{includes}(\text{null}) \) 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: }

\[
\text{value \( \tau \models (\text{Set}(\text{Set}(2, \text{null})) = \text{Set}(\text{Set}(\text{null}, 2))) \)}
\]

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 \( (\tau, \mathcal{C}, \mathcal{A}, \mathcal{S}) \) where:

1. \( \mathcal{C} \) 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: \( C_i.\text{allInstances}() \) and \( C_i.\text{allInstances}@\text{pre}() \).

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

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

4. \( \text{Assoc}(C_i, C_j) \) is a collection of associations. An association \( (n, \text{rn}_{\text{from}}, \text{rn}_{\text{to}}) \in \text{Assoc}(C_i, C_j) \) between to classes \( C_i \) and \( C_j \) is a triple consisting of a (unique) association name \( n \), and the role-names \( \text{rn}_{\text{from}} \) and \( \text{rn}_{\text{to}} \). To each role-name belong two families of accessors denoted \( X.a :: C_i \rightarrow A \) and \( X.a@\text{pre} :: C_i \rightarrow A \) for \( A \in \text{TYPES}(C) \)),

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

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

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

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 \( \mathcal{X} \) of all OCL operations discussed so far.

---

*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:

\[
\text{typedef } \alpha \text{ state} := \{\sigma : \text{oid} \rightarrow \alpha | \text{inv}_\sigma(\sigma)\}
\]

where \( \text{inv}_\sigma \) is a to be discussed invariant on states.

The key point is that we need a common type \( \alpha \) 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 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: \( C_i := (\text{oid} \times A_{i_1} \times \cdots \times A_{i_\ell}) \) where the types \( A_{i_1}, \ldots, A_{i_\ell} \) are the attribute types (including inherited attributes) with class types substituted by oid. The function \( \text{OidOf} \) projects the first component, the oid, out of an object representation. Then the object universe will be constructed by the type definition:

\[
\mathfrak{A} := C_1 + \cdots + C_n.
\]

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

\[
X.\text{oclIsTypeOf}(C_k) \implies X.\text{oclAsType}(C_i).\text{oclAsType}(C_k) = X
\]  
whenever \( C_k < C_i \) and \( X \) is valid.

To overcome this limitation, we introduce an auxiliary type \( C_{\text{ext}} \) 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 \( \{C_{i_1}, \ldots, C_{i_\ell}\} \).

- Then the class type extension \( C_{\text{ext}} \) associated to \( C_i \) is \( A_{i_1} \times \cdots \times A_{i_\ell} \times (C_{i_1\text{ext}} + \cdots + C_{i\text{ext}}) \) where \( A_{i_\ell} \) ranges over the local attribute types of \( C_i \) and \( C_{i\text{ext}} \) ranges over all class type extensions of the subclass \( C_j \) of \( C_i \).

- Then the class type for \( C_i \) is \( \text{oid} \times A_{i_1} \times \cdots \times A_{i_\ell} \times (C_{i\text{ext}} + \cdots + C_{i\text{ext}}) \) where \( A_{i_\ell} \) ranges over the inherited and local attribute types of \( C_i \) and \( C_{i\text{ext}} \) 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:

\[
\text{definition \ eval\_extract}_X f = (\lambda \tau. \text{case } X \tau \text{ of } \bot \Rightarrow \text{invalid } \tau \quad \text{exception} \\
& | \bot_j \Rightarrow \text{invalid } \tau \quad \text{deref. null} \\
& | \uparrow_{\text{oid}} \Rightarrow f (\text{oid\_of } \text{obj}) \tau) \quad (A.19)
\]

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

\[
\text{definition \ deref\_oid}_C f \text{st}_\text{snd} \text{f oid} = (\lambda \tau. \text{case } \text{heap } (\text{fst}_\text{snd} \tau) \text{ oid of } \\
& | \text{in}_C \text{obj}_j \Rightarrow f \text{ obj } \tau \\
& | \_ \Rightarrow \text{invalid } \tau)) \quad (A.20)
\]

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:

\[
\text{definition \ select}_a f = (\lambda \text{ mk}_C \text{ oid } \cdots \bot \cdots C_{\text{ext}} \Rightarrow \text{null} \\
& | \text{ mk}_C \text{ oid } \cdots a \cdots C_{\text{ext}} \Rightarrow f (\lambda x. \uparrow_{\text{oid}} a)) \quad (A.21)
\]

17
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:

\[
\begin{align*}
\text{definition} & \quad \text{in\_pre\_state} = \text{fst} & & \text{first component} \\
\text{definition} & \quad \text{in\_post\_state} = \text{snd} & & \text{second component} \\
\text{definition} & \quad \text{reconst\_basetype} = \text{id} & & \text{identity function}
\end{align*}
\]  

(A.22)

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

\[
\begin{align*}
\text{definition} & \quad _.\text{getBase} :: C \Rightarrow A_{\text{base}} \\
\text{where} & \quad X\cdot\text{getBase} = \text{eval\_extract}(X\text{\_deref\_oid\_C \_in\_post\_state}(\text{select}\_\text{getBase}\_\text{reconst\_basetype}))
\end{align*}
\]  

(A.23)

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

\[
\begin{align*}
\text{definition} & \quad _.\text{getObject} :: C \Rightarrow A_{\text{object}} \\
\text{where} & \quad X\cdot\text{getObject} = \text{eval\_extract}(X\text{\_deref\_oid\_C \_in\_post\_state}(\text{select}\_\text{getObject}\_\text{deref\_oid\_C \_in\_post\_state}))
\end{align*}
\]  

(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 \( _.a@\text{pre} \) were produced when \( \text{in\_post\_state} \) is replaced by \( \text{in\_pre\_state} \).

Examples for the construction of accessors via associations can be found in Section A.7.8. The construction of casts and type tests \( \text{oclIsTypeOf()} \) and \( \text{oclIsKindOf()} \) 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 \( \text{null} \) 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, \( \text{null} \) would be mapped to the singleton set containing \( \text{null} \), but the standard demands that the resulting set is empty in this case. The conversion should instead be defined as follows:

\[
\text{context} \quad \text{OclAny::asSet():T} \\
\text{post:} \quad \text{if self = \text{null} then result = Set{} else result = Set{self} endif}
\]

**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 \( \text{null} \) can belong to this collection. The OCL standard states that \( \text{null} \) can be owned by collections. However, if an attribute can evaluate to a collection containing \( \text{null} \), it is not clear how multiplicity constraints should be interpreted for this attribute. The question arises whether the \( \text{null} \) element should be counted or not when determining the cardinality of the collection. Recall that \( \text{null} \) denotes the absence of value in the case of a cardinality upper bound of one, so we would assume that \( \text{null} \) is not counted. On the other hand, the operation \( \text{size} \) defined for collections in OCL does count \( \text{null} \).
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. 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:

<table>
<thead>
<tr>
<th>context C inv lowerBound: a-&gt;size() &gt;= m</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv upperBound: a-&gt;size() &lt;= n</td>
</tr>
<tr>
<td>inv notNull: not a-&gt;includes(null)</td>
</tr>
</tbody>
</table>

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 ≤ 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:

<table>
<thead>
<tr>
<th>$\tau \models X.oclAsType(C_i) \equiv X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau \models invalid.oclAsType(C_i) \equiv invalid$</td>
</tr>
<tr>
<td>$\tau \models null.oclAsType(C_i) \equiv null$</td>
</tr>
<tr>
<td>$\tau \models ((X :: C_i).oclAsType(C_j)) .oclAsType(C_i) \equiv X$</td>
</tr>
<tr>
<td>$\tau \models X .oclAsType(C_j) .oclAsType(C_i) \equiv X$</td>
</tr>
<tr>
<td>$\tau \models (X :: OclAny).oclAsType(OclAny) \equiv X$</td>
</tr>
<tr>
<td>$\tau \models u(X :: C_i) \implies \tau \models (X .oclIsTypeOf(C_i) implies (X .oclAsType(C_j) .oclAsType(C_i)) \equiv X)$</td>
</tr>
<tr>
<td>$\tau \models u(X :: C_i) \implies \tau \models X .oclIsTypeOf(C_i) implies (X .oclAsType(C_j) .oclAsType(C_i)) \equiv X$</td>
</tr>
</tbody>
</table>

\*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.
\[ \tau \models \delta X \implies \tau \models X .oclAsType(C_j) .oclAsType(C_i) \triangleq X \]
\[ \tau \models \nu X \implies \tau \models X .oclIsTypeOf(C_i) \implies X .oclAsType(C_j) .oclAsType(C_i) \triangleq X \]
\[ \tau \models X .oclIsTypeOf(C_i) \implies \tau \models \delta X \implies \tau \models \not(\nu X .oclAsType(C_i)) \]
\[ \tau \models invalid .oclIsTypeOf(C_i) \triangleq invalid \]
\[ \tau \models null .oclIsTypeOf(C_i) \triangleq true \]

With respect to attributes \_a or \_a@pre and role-ends \_r or \_r@pre we have

\[ invalid .a = invalid \]
\[ invalid .a@pre = invalid \]
\[ invalid .r = invalid \]
\[ invalid .r@pre = invalid \]

With respect to attributes \_a or \_a@pre and role-ends \_r or \_r@pre we have

\[ invalid .a = invalid \]
\[ invalid .a@pre = invalid \]
\[ invalid .r = invalid \]
\[ invalid .r@pre = invalid \]

Other Operations on States

Defining \_allInstances() is straight-forward; the only difference is the property
\[ T .allInstances() \rightarrow excludes(null) \] 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

\[ (S : Set(OclAny)) \rightarrow oclIsModifiedOnly() : Boolean \]
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:

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

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

### 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 $\_a$ from the class model $a \in Attr(C)$ by their counterparts $\_i@\text{pre}$. For example, $(\text{self}. \text{salary} > 500)_{\text{pre}}$ is transformed to $(\text{self}. \text{salary}@\text{pre} > 500)$.

- all role accessor functions $\_\text{rn}_\text{from}$ or $\_\text{rn}_\text{to}$ within the class model (i.e. $(id, \text{rn}_\text{from}, \text{rn}_\text{to}) \in \text{Assoc}(C_i, C_j)$) were replaced by their counterparts $\_\text{rn}@\text{pre}$. For example, $(\text{self}. \text{boss} = \text{null})_{\text{pre}}$ is transformed to $\text{self}. \text{boss}@\text{pre} = \text{null}$.

- The operation $\_\text{allInstances}()$ is also substituted by its $@\text{pre}$ counterpart.

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

$$I[[\text{context } c : C_i \text{ inv } n : \phi(c)]\tau \equiv \begin{cases} \tau \models (C_i \text{.allInstances()}->\forall x(\phi(x))) & \text{if } \tau \models (C_i \text{.allInstances()}->\forall x(\phi(x)))_{\text{pre}} \end{cases}$$

Recall that expressions containing $@\text{pre}$ 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 $\text{op}$ with the arguments $a_1, \ldots, a_n$ the two cases where all arguments are valid and additionally, $\text{self}$ is non-null (i.e. it must be defined), or not. In former case, a method call can be replaced by a $\text{result}$ that satisfies the contract, in the latter case the

---

10This 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:

\[
I[\text{context } C ::= \text{op}(a_1, \ldots, a_n) : T]
\]

\[
\text{pre } \phi(\text{self}, a_1, \ldots, a_n)
\]

\[
\text{post } \psi(\text{self}, a_1, \ldots, a_n, \text{result})
\]

\[
\lambda s, x_1, \ldots, x_n. \tau.
\]

\[
\text{if } \tau \models \partial s \land \tau \models \forall x_1 \land \ldots \land \tau \models \forall x_n
\]

\[
\text{then } \text{SOME result. } \tau \models \phi(s, x_1, \ldots, x_n) \text{pre}
\]

\[
\land \tau \models \psi(s, x_1, \ldots, x_n, \text{result})
\]

\[
\text{else } \bot
\]

(A.26)

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

\[
f_{\text{op}} \equiv I[\text{context } C ::= \text{op}(a_1, \ldots, a_n) : T]...
\]

(A.27)

provided that neither \( \phi \) nor \( \psi \) contain recursive method calls of \( \text{op} \). In the case of a query operation (i.e. \( \tau \) must have the form: \((\sigma, \sigma)\), which means that query operations do not change the state; c.f. \text{oclIsModifiedOnly()} in Section A.3.4), this constraint can be relaxed: the above equation is then stated as \text{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 \( f_{\text{op}} \) 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 \( f_{\text{op}} \) to a proof of \( E(\text{res}) \) (where \( \text{res} \) must be one of the values that satisfy the post-condition \( \psi \)):

\[
\begin{array}{c}
\tau \models E(\text{res}) \\
\vdots \\
\tau \models E(f_{\text{op}} \text{self } a_1 \ldots a_n)
\end{array}
\]

(A.28)

under the conditions:

- \( E \) must be an OCL term and

- \text{self} must be defined, and the arguments valid in \( \tau \):

\[
\tau \models \partial \text{self} \land \tau \models \forall a_1 \land \ldots \land \tau \models \forall a_n
\]

- the post-condition must be satisfiable (“the operation must be implementable”): \( \exists \text{res}. \tau \models \psi \text{self } a_1 \ldots a_n \text{res} \).

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

\[
\tau \models \phi \text{self } a_1 \ldots a_n
\]

\[
(\tau \models E(f_{\text{op}} \text{self } a_1 \ldots a_n)) = e(\tau \models E(\text{BODY self } a_1 \ldots a_n))
\]

(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.

• \( \text{self} \) must be defined, and the arguments valid in \( \tau \):
  \[
  \tau \models \partial \text{self} \land \tau \models \nu a_1 \land \ldots \land \tau \models \nu a_n
  \]

• the postcondition \( \psi \operatorname{self} a_1 \ldots a_n \text{result} \) must be decomposable into:
  \[
  \psi' \operatorname{self} a_1 \ldots a_n \text{ and result} \triangleq \text{BODY} \operatorname{self} a_1 \ldots a_n.
  \]

We do not model \textit{overriding} of operations as in Java or C++ explicitly in Featherweight OCL. However, it is easy expressed in this core-language by adding \( \text{self}\.oclIsKindOf(C) \) in the pre-condition \( \phi \) (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:

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

- \text{notation} \( \text{Some} (\langle \cdot \rangle) \)
- \text{notation} \( \text{None} (\bot) \)

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

\[
\text{fun} \quad \text{drop} :: \alpha \text{ option} \Rightarrow \alpha (\lceil \cdot \rceil)
\]

\[
\text{where} \quad \text{drop-lift}[\text{simp}]: \lceil v \rceil = v
\]

The definitions for the constants and operations based on functions will be geared towards a format that Isabelle can check to be a “conservative” (i.e., logically safe) axiomatic definition. By introducing an explicit interpretation function (which happens to be defined just as the identity since we are using a shallow embedding of OCL into HOL), all these definitions can be rewritten into the conventional semantic textbook format. To say it in other words: The interpretation function \( \text{Sem} \) 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.

\[
\text{definition} \quad \text{Sem} :: \alpha \Rightarrow \alpha (I[-])
\]

\[
\text{where} \quad I[x] \equiv x
\]

**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 \( \text{Set}\{\text{Set}\{2\}, \text{null}\} \), it is necessary to introduce a uniform interface for types having the \textit{invalid} (= bottom) element. The reason is that we impose a data-invariant on raw-collection \textit{types code} which assures that the \textit{invalid} element is not allowed inside the collection; all raw-collections of this form were identified with the \textit{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 \( \bot \) on \( 'a \ option \ option \)) to a null element, which may have an arbitrary semantic structure, and an undefinedness element \( \bot \) to an abstract undefinedness element \( bot \) (also written \( \bot \) 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.

\[
\text{class } bot = \\
\text{fixes } \bot : 'a \\
\text{assumes nonEmpty : } \exists x. x \neq bot
\]

\[
\text{class } null = bot + \\
\text{fixes } \bot : 'a \\
\text{assumes null-is-valid : } null \neq bot
\]

**Accommodation of Basic Types to the Abstract Interface**

In the following it is shown that the “option-option” type is in fact in the \( null \) class and that function spaces over these classes again “live” in these classes. This motivates the default construction of the semantic domain for the basic types (Boolean, Integer, Real, ...).

\[
\text{instantiation option :: (type)bot} \\
\text{begin} \\
\text{definition bot-option-def: } (bot::'a option) \equiv (None::'a option) \\
\text{instance} \\
\text{end}
\]

\[
\text{instantiation option :: (bot)null} \\
\text{begin} \\
\text{definition null-option-def: } (null::'a:bot option) \equiv \bot_\bot \\
\text{instance} \\
\text{end}
\]

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

\[
\text{instantiation fun :: (type.null) null} \\
\text{begin} \\
\text{definition null-fun-def: } (null::'a \Rightarrow 'b: null) \equiv (\lambda x. null) \\
\text{instance} \\
\text{end}
\]
A trivial consequence of this adaption of the interface is that abstract and concrete versions of null are the same on base types (as could be expected).

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

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

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

**type-synonym** oid = nat

We refrained from the alternative:

**type-synonym** oid = ind

which is slightly more abstract but non-executable.

*States* in UML/OCL are a pair of

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

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

**record** (\( \mathcal{A} \)) state =  
  heap :: oid \( \rightharpoonup \) \( \mathcal{A} \)  
  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** (\( \mathcal{A} \)) st = \( \mathcal{A} \) state \( \times \) \( \mathcal{A} \) state

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

**class** object =  
  fixes oid-of :: \( \mathcal{A} \) = oid

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

**typ** \( \mathcal{A} \) :: object

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

**instantiation** option :: (object)object

begin  
  definition oid-of-option-def : oid-of x = oid-of (the x)
  instance
end
Common Infrastructure for all OCL Types (II): Valuations as OCL Types

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

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

\[
\text{type-synonym } (\mathcal{A}, \alpha) \text{ val } = \mathcal{A} \times \alpha \Rightarrow \alpha :: \text{null}
\]

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:

\[
\begin{align*}
\text{definition } & \text{invalid :: } (\mathcal{A}, \alpha :: \text{bot}) \text{ val} \\
& \quad \text{where } \quad \text{invalid } \equiv \lambda \tau. \text{bot}
\end{align*}
\]

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

\[
\text{lemma textbook-invalid: \{invalid\} } \tau = \text{bot}
\]

Note that the definition:

\[
\begin{align*}
\text{definition } & \text{null } :: " (\mathcal{A}, \alpha :: \text{null}) \text{ val}" \\
& \quad \text{where } \quad " \text{null } \equiv \lambda \tau. \text{null}"
\end{align*}
\]

is not necessary since we defined the entire function space over null types again as null-types; the crucial definition is null \( \equiv \lambda x. \text{null} \). 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:

\[
\text{lemma textbook-null-fun: \{null::(\mathcal{A}, \alpha :: \text{null}) \text{ val}\} } \tau = \text{(null::(\alpha::null))}
\]

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:

\[
\begin{align*}
\text{type-synonym } & \text{Boolean}_\text{base} = \text{bool option option} \\
\text{type-synonym } & (\mathcal{A})\text{Boolean } = (\mathcal{A}, \text{Boolean}_\text{base}) \text{ val}
\end{align*}
\]

Because of the previous class definitions, Isabelle type-inference establishes that \( \mathcal{A} \text{ Boolean} \) 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:

\[
\begin{align*}
\text{type-synonym } & \text{Integer}_\text{base} = \text{int option option} \\
\text{type-synonym } & (\mathcal{A})\text{Integer } = (\mathcal{A}, \text{Integer}_\text{base}) \text{ val}
\end{align*}
\]

\[
\text{type-synonym } \text{String}_\text{base} = \text{string option option}
\]
type-synonym (∀A)String = (∀A.String_base) val

type-synonym Real_base = real option option

type-synonym (∀A)Real = (∀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 ∨ X = null }

type-synonym (∀A)Void = (∀A.Void_base) val

A.4.3. Some OCL Collection Types

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

The Construction of the Pair Type (Tuples)
The core of an own type construction is done via a type definition which provides the base-type (′α, ′β) Pair_base. It is shown that this type “fits” indeed into the abstract type interface discussed in the previous section.

typedef (′α, ′β) Pair_base = {X::(′α::null × ′β::null) option option. X = bot ∨ X = null ∨ (fst⌜⌜X⌝⌝ ≠ bot ∧ snd⌜⌜X⌝⌝ ≠ bot) }

We “carve” out from the concrete type (′α × ′β) option option the new fully abstract type, which will not contain representations like ⊥(⊥, a) or (b, ⊥). The type constructor Pair{x,y} to be defined later will identify these with invalid.

instantiation Pair_base :: (null,null)bot
begin
  definition bot-Pair_base-def: (bot-class.bot :: (′a::null,′b::null) Pair_base) ≡ Abs-Pair_base None
  instance
end

instantiation Pair_base :: (null,null)null
begin
  definition null-Pair_base-def: (null::(′a::null,′b::null) Pair_base) ≡ Abs-Pair_base None
  instance
end

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

type-synonym (∀A,′α,′β) Pair = (∀A, (′α,′β) 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 $\alpha \text{Set}_{\text{base}}$. 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 $\alpha \text{Set}_{\text{base}} = \{X::(\alpha::\text{null}) \text{ set option option. } X = \text{bot} \lor X = \text{null} \lor (\forall x \in \set{\uparrow X}, x \neq \text{bot})\}$

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

definition bot-Set_{\text{base}}-def: (\text{bot}::(\alpha::\text{null}) \text{ Set}_{\text{base}}) \equiv \text{Abs-Set}_{\text{base}} \text{None}
   instance
end

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

definition null-Set_{\text{base}}-def: (\text{null}::(\alpha::\text{null}) \text{ Set}_{\text{base}}) \equiv \text{Abs-Set}_{\text{base}} \text{None}_\bot
   instance
end

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

type-synonym ($\forall \alpha \alpha \text{ Set} = (\forall \alpha, \alpha \text{ Set}_{\text{base}}) \text{val}$

The Construction of the Sequence Type

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


typedef $\alpha \text{Sequence}_{\text{base}} = \{X::(\alpha::\text{null}) \text{ list option option. } X = \text{bot} \lor X = \text{null} \lor (\forall x \in \set{\uparrow X}, x \neq \text{bot})\}$

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

definition bot-Sequence_{\text{base}}-def: (\text{bot}::(\alpha::\text{null}) \text{ Sequence}_{\text{base}}) \equiv \text{Abs-Sequence}_{\text{base}} \text{None}
   instance
end

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

definition null-Sequence_{\text{base}}-def: (\text{null}::(\alpha::\text{null}) \text{ Sequence}_{\text{base}}) \equiv \text{Abs-Sequence}_{\text{base}} \text{None}_\bot
   instance
end

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

type-synonym ($\forall \alpha \alpha \text{ Sequence} = (\forall \alpha, \alpha \text{ Sequence}_{\text{base}}) \text{val}$
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 presentation 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:

<table>
<thead>
<tr>
<th>OCL Type</th>
<th>HOL Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>$\forall \text{ Boolean}</td>
</tr>
<tr>
<td>Boolean -&gt; Boolean</td>
<td>$\forall \text{ Boolean} \Rightarrow \forall \text{ Boolean}</td>
</tr>
<tr>
<td>(Integer, Integer) -&gt; Boolean</td>
<td>$(\forall \text{ Integer} \Rightarrow \forall \text{ Integer}) \Rightarrow \forall \text{ Boolean}</td>
</tr>
<tr>
<td>Set (Integer)</td>
<td>$(\forall \text{ Integer}, \text{ Set})</td>
</tr>
<tr>
<td>Set (Integer) -&gt; Real</td>
<td>$(\forall \text{ Integer}, \text{ Real})</td>
</tr>
<tr>
<td>Set (Pair (Integer, Boolean))</td>
<td>$(\forall \text{ Integer}, \forall \text{ Boolean})</td>
</tr>
<tr>
<td>Set (&lt;T&gt;)</td>
<td>$(\forall \text{ T}, \text{ Set})</td>
</tr>
</tbody>
</table>

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. cartesian products of arguments were curried,
2. constants of type $T$ were mapped to valuations over the HOL-type for $T$,
3. functions $T \rightarrow T'$ were mapped to functions in HOL, where $T$ and $T'$ were mapped to the valuations for them, and
4. the arguments of type constructors $\text{Set}(T)$ 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 : (bot::(\forall)Boolean) = (\lambda \tau. \bot)

lemma null-Boolean-def : (null::(\forall)Boolean) = (\lambda \tau. \bot)

definition true ::= (\forall)Boolean
where \( \text{true} \equiv \lambda \tau. \text{True} \)

**Definition** `false :: (\forall \tau).\text{Boolean}`

where \( \text{false} \equiv \lambda \tau. \text{False} \)

**Lemma** `bool-split-0`: \( X \tau = \text{invalid} \tau \lor X \tau = \text{null} \tau \lor X \tau = \text{true} \tau \lor X \tau = \text{false} \tau \)

**Lemma** `[simp]`: \( \text{false} (a, b) = \text{False} \)

**Lemma** `[simp]`: \( \text{true} (a, b) = \text{True} \)

**Lemma** `textbook-true`: \( I[\text{true}] \tau = \text{True} \)

**Lemma** `textbook-false`: \( I[\text{false}] \tau = \text{False} \)

<table>
<thead>
<tr>
<th>Name</th>
<th>Theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>textbook-invalid</td>
<td>( I[\text{invalid}] \tau = \text{UML-Types:bot-class:bot} )</td>
</tr>
<tr>
<td>textbook-null-fun</td>
<td>( I[\text{null}] \tau = \text{null} )</td>
</tr>
<tr>
<td>textbook-true</td>
<td>( I[\text{true}] \tau = \text{True} )</td>
</tr>
<tr>
<td>textbook-false</td>
<td>( I[\text{false}] \tau = \text{False} )</td>
</tr>
</tbody>
</table>

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 :: (\forall \alpha: \text{null})\text{val} \Rightarrow (\forall \alpha)(\forall \beta: [100]100) \text{Boolean} (\nu - [100]100)`

where \( \nu X \equiv \lambda \tau. \text{if } X \tau = \text{bot } \tau \text{ then false } \tau \text{ else true } \tau \)

**Lemma** `valid1 [simp]`: \( \nu \text{invalid} = \text{false} \)

**Lemma** `valid2 [simp]`: \( \nu \text{null} = \text{true} \)

**Lemma** `valid3 [simp]`: \( \nu \text{true} = \text{true} \)

**Lemma** `valid4 [simp]`: \( \nu \text{false} = \text{true} \)

**Definition** `defined :: (\forall \alpha: \text{null})\text{val} \Rightarrow (\forall \beta)\text{Boolean} (\delta - [100]100)`

where \( \delta X \equiv \lambda \tau. \text{if } X \tau = \text{bot } \tau \lor X \tau = \text{null } \tau \text{ then false } \tau \text{ else true } \tau \)

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

**Lemma** `defined1 [simp]`: \( \delta \text{invalid} = \text{false} \)

**Lemma** `defined2 [simp]`: \( \delta \text{null} = \text{false} \)

**Lemma** `defined3 [simp]`: \( \delta \text{true} = \text{true} \)

**Lemma** `defined4 [simp]`: \( \delta \text{false} = \text{true} \)

**Lemma** `defined5 [simp]`: \( \delta \delta X = \text{true} \)
lemma defined6[simp]: \( \delta \upsilon X = \text{true} \)
lemma valid5[simp]: \( \upsilon \upsilon X = \text{true} \)
lemma valid6[simp]: \( \upsilon \delta X = \text{true} \)

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

lemma textbook-defined: \( I[\delta(X)] \tau = (\text{if } I[X] \tau = I[\text{bot}] \tau \lor I[X] \tau = I[\text{null}] \tau \text{ then } I[\text{false}] \tau \text{ else } I[\text{true}] \tau) \)
lemma textbook-valid: \( I[\upsilon(X)] \tau = (\text{if } I[X] \tau = I[\text{bot}] \tau \text{ then } I[\text{false}] \tau \text{ else } I[\text{true}] \tau) \)

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>textbook-defined</td>
<td>( I[\delta(X)] \tau = (\text{if } I[X] \tau = I[\text{bot}] \tau \lor I[X] \tau = I[\text{null}] \tau \text{ then } I[\text{false}] \tau \text{ else } I[\text{true}] \tau) )</td>
</tr>
<tr>
<td>textbook-valid</td>
<td>( I[\upsilon(X)] \tau = (\text{if } I[X] \tau = I[\text{bot}] \tau \text{ then } I[\text{false}] \tau \text{ else } I[\text{true}] \tau) )</td>
</tr>
</tbody>
</table>

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

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 \( _\triangleq _ \) 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 \( a \text{.boss} \triangleq b \text{.boss@pre} \) instead of \( a \text{.boss} = b \text{.boss@pre} \) and \((\ast \text{ just the pointers are equal! } \ast)\).\[a \text{.boss.name} = b \text{.boss@pre.name@pre} \]\[a \text{.boss.age} = b \text{.boss@pre.age@pre} \]

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

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

\[
s = t \implies P(s) = P(t) \tag{A.30}
\]

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

**definition** \( \text{StrongEq:} [\forall \alpha. \text{st} \Rightarrow \alpha, \forall \alpha. \text{st} \Rightarrow \alpha] \Rightarrow (\forall \alpha)\text{Boolean} \) \( \text{(infixl} \triangleq 30) \)

where \( X \triangleq Y \equiv \lambda \tau. \downarrow X \tau = Y \tau \downarrow \)

From this follow already elementary properties like:

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

**lemma** [simp,code-unfold]: \( (\text{false} \triangleq \text{true}) = \text{false} \)
Fundamental Predicates on Strong Equality  

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

**lemma** StrongEq-refl [simp]: \((X \eq X) = \text{true}\)

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

**lemma** StrongEq-trans-strong [simp]:

assumes \(A: (X \eq Y) = \text{true}\)

and \(B: (Y \eq Z) = \text{true}\)

shows \((X \eq Z) = \text{true}\)

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

**lemma** StrongEq-subst:

assumes \(cp: \forall X. P(X) \equiv P(\lambda - . X \ \tau) \ \tau\)

and \(eq: (X \eq Y) \ \tau = \text{true} \ \tau\)

shows \((P X \eq P Y) \tau = \text{true} \ \tau\)

**lemma** defined7[simp]: \(\delta (X \eq Y) = \text{true}\)

**lemma** valid7[simp]: \(v (X \eq Y) = \text{true}\)

**lemma** cp-StrongEq: \((X \eq Y) \ \tau = ((\lambda - . X \ \tau) \eq (\lambda - . Y \ \tau)) \ \tau\)

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 :: \((\forall )\text{Boolean} \Rightarrow (\forall )\text{Boolean} (not)\)

where \(not X = \lambda \tau. \ \text{case} X \ \tau \ \text{of}\)

\[
\begin{align*}
\bot & \Rightarrow \bot \\
\bot & \Rightarrow \bot_{\bot} \\
\bot_{\bot_{\bot}} & \Rightarrow \bot_{\bot_{\bot}} \\
\bot_{\bot_{\bot}} & \Rightarrow \bot_{\bot_{\bot}} \\
\end{align*}
\]
lemma 

\( \text{cp-OclNot}: (\neg X) \tau = (\neg (\lambda \tau. X) \tau) \)

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:

\( \neg (\neg X) = X \)

lemma OclNot-inject:

\( \land \ x \ y. \ \neg x = \neg y \implies x = y \)

definition OclAnd :: \((\forall \! A. \ Boolean), (\forall \! A. \ Boolean)\) \Rightarrow (\forall \! A. \ Boolean) (infixl and 30)
where

\[
\begin{align*}
\text{X and Y} & \equiv \langle \lambda \ \tau. \ \text{case X} \ \tau \ of \ \\
\text{False} & \Rightarrow \langle \text{False} \rangle \\
\bot & \Rightarrow (\langle \bot \rangle) \\
\text{True} & \Rightarrow \text{Y} \ \tau \\
\rangle \\
\end{align*}
\]

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

In textbook notation, the logical core constructs \( \neg \) and \( \land \) were represented as follows:

lemma textbook-OclNot:

\[
\begin{align*}
I^\llbracket\neg(X)\rrbracket \ \tau & = (\text{case I}^\llbracket X \rrbracket \ \tau \ of \ \\
\bot & \Rightarrow \bot \\
\langle \bot \rangle & \Rightarrow \langle \bot \rangle \\
\langle \bot, x \rangle & \Rightarrow \langle \bot, \neg x \rangle \\
\end{align*}
\]

lemma textbook-OclAnd:

\[
\begin{align*}
I^\llbracket X \land Y \rrbracket \ \tau & = (\text{case I}^\llbracket X \rrbracket \ \tau \ of \ \\
\bot & \Rightarrow (\text{case I}^\llbracket Y \rrbracket \ \tau \ of \ \\
\bot & \Rightarrow \bot \\
\langle \bot \rangle & \Rightarrow \langle \bot \rangle \\
\text{True} & \Rightarrow \text{Y} \ \tau \\
\text{False} & \Rightarrow \text{False} \ \tau \\
\rangle \\
\end{align*}
\]

34
definition OclOr :: \[ ([\mathcal{A}] \text{Boolean}, ([\mathcal{A}] \text{Boolean}) \Rightarrow ([\mathcal{A}] \text{Boolean}) \text{ (infixl or 25)} \]
where \( X \text{ or } Y \equiv \text{not}(\text{not } X \text{ and } \text{not } Y) \)

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

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

lemma cp-OclOr: \( ((X::([\mathcal{A}] \text{Boolean}) \text{ or } Y) \tau = ((\lambda \cdot X \tau) \text{ or } (\lambda \cdot Y \tau)) \tau \)

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

lemma OclAnd1[simp]: \( \text{invalid and true} = \text{invalid} \)
lemma OclAnd2[simp]: \( \text{invalid and false} = \text{false} \)
lemma OclAnd3[simp]: \( \text{invalid and null} = \text{invalid} \)
lemma OclAnd4[simp]: \( \text{invalid and invalid} = \text{invalid} \)

lemma OclAnd5[simp]: \( \text{null and true} = \text{null} \)
lemma OclAnd6[simp]: \( \text{null and false} = \text{false} \)
lemma OclAnd7[simp]: \( \text{null and null} = \text{null} \)
lemma OclAnd8[simp]: \( \text{null and invalid} = \text{invalid} \)

lemma OclAnd9[simp]: \( \text{false and true} = \text{false} \)
lemma OclAnd10[simp]: \( \text{false and false} = \text{false} \)
lemma OclAnd11[simp]: \( \text{false and null} = \text{false} \)
lemma OclAnd12[simp]: \( \text{false and invalid} = \text{false} \)

lemma OclAnd13[simp]: \( \text{true and true} = \text{true} \)
lemma OclAnd14[simp]: \( \text{true and false} = \text{false} \)
lemma OclAnd15[simp]: \( \text{true and null} = \text{null} \)
lemma OclAnd16[simp]: \( \text{true and invalid} = \text{invalid} \)

lemma OclAnd-idem[simp]: \( (X \text{ and } X) = X \)
lemma OclAnd-commute: \( (X \text{ and } Y) = (Y \text{ and } X) \)

lemma OclAnd-false1[simp]: \( \text{false and } X = \text{false} \)
lemma OclAnd-false2[simp]: \( \text{X and false} = \text{false} \)

lemma OclAnd-true1[simp]: \( \text{true and } X = X \)
lemma OclAnd-true2[simp]: \( \text{X and true} = X \)

lemma OclAnd-bot1[simp]: \( \land \tau. X \tau \neq \text{false} \tau \Rightarrow (\text{bot and } X) \tau = \text{bot} \tau \)
lemma OclAnd-bot2[simp]: \( \forall \tau. X \tau \neq false \Rightarrow (X \text{ and bot}) \tau = \text{bot} \)

lemma OclAnd-null1[simp]: \( \forall \tau. X \tau \neq false \Rightarrow X \tau \neq \text{bot} \Rightarrow (null \text{ and } X) \tau = \text{null} \)

lemma OclAnd-null2[simp]: \( \forall \tau. X \tau \neq false \Rightarrow X \tau \neq \text{bot} \Rightarrow (X \text{ and } null) \tau = \text{null} \)

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

lemma OclOr1[simp]: \( (\text{invalid or true}) = \text{true} \)

lemma OclOr2[simp]: \( (\text{invalid or false}) = \text{invalid} \)

lemma OclOr3[simp]: \( (\text{invalid or null}) = \text{invalid} \)

lemma OclOr4[simp]: \( (\text{invalid or invalid}) = \text{invalid} \)

lemma OclOr5[simp]: \( (null \text{ or true}) = \text{true} \)

lemma OclOr6[simp]: \( (null \text{ or false}) = \text{null} \)

lemma OclOr7[simp]: \( (null \text{ or null}) = \text{null} \)

lemma OclOr8[simp]: \( (null \text{ or invalid}) = \text{invalid} \)

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

lemma OclOr-commute: \( (X \text{ or } Y) = (Y \text{ or } X) \)

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

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

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

lemma OclOr-true2: \( (true \text{ or } X) = true \)

lemma OclOr-bot1[simp]: \( \forall \tau. X \tau \neq true \Rightarrow (\text{bot or } X) \tau = \text{bot} \)

lemma OclOr-bot2[simp]: \( \forall \tau. X \tau \neq true \Rightarrow (X \text{ or } \text{bot}) \tau = \text{bot} \)

lemma OclOr-null1[simp]: \( \forall \tau. X \tau \neq true \Rightarrow X \tau \neq \text{bot} \Rightarrow (null \text{ or } X) \tau = \text{null} \)

lemma OclOr-null2[simp]: \( \forall \tau. X \tau \neq true \Rightarrow X \tau \neq \text{bot} \Rightarrow (X \text{ or } null) \tau = \text{null} \)

lemma OclOr-assoc: \( (X \text{ or } (Y \text{ or } Z)) = (X \text{ or } Y \text{ or } Z) \)

lemma OclImplies-true: \( (X \text{ implies } true) = true \)

lemma deMorgan1: \( \text{not}(X \text{ and } Y) = ((\text{not } X) \text{ or } (\text{not } Y)) \)

lemma deMorgan2: \( \text{not}(X \text{ or } Y) = ((\text{not } X) \text{ and } (\text{not } Y)) \)

A Standard Logical Calculus for OCL

definition OclValid :: \( ((\forall \forall) \text{st, } (\forall\forall) \text{Boolean}) \Rightarrow \text{bool} \ ((1(-)/ \models (-)) \ 50) \)

where \( \models P \equiv ((P \forall \tau) = true \)
Global vs. Local Judgements  \lemmatransform1: \( P = true \implies \tau \models P \)

\lemmatransform1-rev: \( \forall \tau. \tau \models P \implies P = true \)

\lemmatransform2: \( (P = Q) \implies ((\tau \models P) = (\tau \models Q)) \)

\lemmatransform2-rev: \( \forall \tau. (\tau \models \delta P) \land (\tau \models \delta Q) \land (\tau \models P) \implies (\tau \models Q) \implies P = Q \)

However, certain properties (like transitivity) cannot be \emph{transformed} from the global level to the local one, they have to be re-proven on the local level.

\lemmashows: \( \tau \models P \implies \tau \models Q \)

Local Validity and Meta-logic  \lemmakeytheorem2: \( \tau \models true \)

\lemmakeytheorem3: \( -(\tau \models false) \)

\lemmakeytheorem4: \( -(\tau \models invalid) \)

\lemmakeytheorem5: \( -(\tau \models null) \)

\lemmakeytheorem6: \( (\tau \models (x \triangleq invalid)) \lor (\tau \models (x \triangleq null)) \lor (\tau \models (x \triangleq true)) \lor (\tau \models (x \triangleq false)) \)

\lemmakeytheorem7: \( (\tau \models \delta x) = ((\neg (\tau \models (x \triangleq invalid)))) \land (\neg (\tau \models (x \triangleq null))))\)

\lemmakeytheorem8: \( (\tau \models \upsilon A) = ((\tau \models A \triangleq null) \lor (\tau \models A) \lor (\tau \models not A)) \)

\lemmakeytheorem9: \( (\tau \models \delta A) = ((\tau \models A) \lor (\tau \models not A)) \)

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

**lemma foundation8:**
\[(\tau \models \delta x) \lor (\tau \models (x \triangleq invalid)) \lor (\tau \models (x \triangleq null))\]

**lemma foundation9:**
\[\tau \models \delta x \iff (\tau \models not x) = (\neg (\tau \models x))\]

**lemma foundation9':**
\[\tau \models not x \iff (\tau \models x)\]

**lemma foundation9'':**
\[\tau \models not x \iff \tau \models \delta x\]

**lemma foundation10:**
\[\tau \models \delta x \iff (\tau \models (x and y)) = ( (\tau \models x) \land (\tau \models y))\]

**lemma foundation10':**
\[\tau \models (A and B) = ((\tau \models A) \land (\tau \models B))\]

**lemma foundation11:**
\[\tau \models \delta x \iff (\tau \models (x or y)) = ( (\tau \models x) \lor (\tau \models y))\]

**lemma foundation12:**
\[\tau \models \delta x \iff (\tau \models (x implies y)) = ( (\tau \models x) \rightarrow (\tau \models y))\]

**lemma foundation13:**
\[(\tau \models A \triangleq true) = (\tau \models A)\]

**lemma foundation14:**
\[(\tau \models A \triangleq false) = (\tau \models not A)\]

**lemma foundation15:**
\[(\tau \models A \triangleq invalid) = (\tau \models not(\nu A))\]

**lemma foundation16:**
\[\tau \models (\delta X) = (X \tau \neq bot \land X \tau \neq null)\]

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

**lemma foundation16'':**
\[\tau \models (\delta X) = (X \tau \neq invalid \land X \tau \neq null \tau)\]

**lemma foundation18:**
\[(\tau \models (\nu X)) = (X \tau \neq invalid)\]

**lemma foundation18':**
\[(\tau \models (\nu X)) = (X \tau \neq bot)\]

**lemma foundation18'':**
\[\tau \models (\nu X) = (\neg(\tau \models (X \triangleq invalid)))\]
lemma foundation20: \( \tau \models (\delta X) \implies \tau \models \nu X \)

lemma foundation21: \((\text{not } A \triangleq \text{not } B) = (A \triangleq B)\)

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

lemma foundation23: \((\tau \models P) = (\tau \models (\lambda \cdot P \tau))\)

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

lemma foundation25: \(\tau \models P \implies \tau \models (P or Q)\)

lemma foundation25': \(\tau \models Q \implies \tau \models (P or Q)\)

lemma foundation26: assumes defP: \(\tau \models \delta P\)
assumes defQ: \(\tau \models \delta Q\)
assumes H: \(\tau \models (P or Q)\)
assumes P: \(\tau \models P \implies R\)
assumes Q: \(\tau \models Q \implies R\)
shows R

lemma foundation27: \((\tau \models (A and B)) = (((\tau \models A) \land (\tau \models B))\)

lemma defined-not-I: \(\tau \models \delta (x) \implies \tau \models \delta (\not x)\)

lemma valid-not-I: \(\tau \models \nu (x) \implies \tau \models \nu (\not x)\)

lemma defined-and-I: \(\tau \models \delta (x) \implies \tau \models \delta (y) \implies \tau \models \delta (x \text{ and } y)\)

lemma valid-and-I: \(\tau \models \nu (x) \implies \tau \models \nu (y) \implies \tau \models \nu (x \text{ and } y)\)

lemma defined-or-I: \(\tau \models \delta (x) \implies \tau \models \delta (y) \implies \tau \models \delta (x \text{ or } y)\)

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

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

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

lemma StrongEq-L-trans: \(\tau \models (x \triangleq y) \implies \tau \models (y \triangleq z) \implies \tau \models (x \triangleq z)\)
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 \( cp \) :: \((\forall x. x)\) val \(\Rightarrow\) \((\forall y. y)\) val \(\Rightarrow\) bool

where \( cp\ P \equiv (\exists f. \forall X \tau. PX\tau = f (X\tau)\tau) \)

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

lemma StrongEq-L-subst1: \(\forall \tau. cp\ P \implies \tau \models (x \triangle y) \implies \tau \models (P\ x \triangle P\ y)\)

lemma StrongEq-L-subst2: \(\forall \tau. cp\ P \implies \tau \models (x \triangle y) \implies \tau \models (P\ x) \implies \tau \models (P\ y)\)

lemma StrongEq-L-subst2-rev: \(\exists \tau. cp\ P \models y \triangle x \implies cp\ P \implies \tau \models P\ x \implies \tau \models P\ y\)

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

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

lemma StrongEq-L-subst4-rev: assumes \( eq\ :\ \tau \models (x \triangle y)\) and \( cp\ :\ cp\ P\)
shows \( (\neg(\tau \models P\ x)) = (\neg(\tau \models P\ y))\)

thm arg-cong[of - - Not]

lemma cpI1: 
\( (\forall X\ \tau. fX\tau = f(\lambda\ X\ \tau)\ \tau) \implies cp\ P \implies cp(\lambda X.f (PX))\)

lemma cpI2: 
\( (\forall X Y\ \tau. fXY\tau = f(\lambda\ X\ \tau)(\lambda\ Y\ \tau)\ \tau) \implies cp\ P \implies cp\ Q \implies cp(\lambda X.f (PX) (QX))\)

lemma cpI3: 
\( (\forall X Y Z\ \tau. fXYZ\tau = f(\lambda\ X\ \tau)(\lambda\ Y\ \tau)(\lambda\ Z\ \tau)\ \tau) \implies cp\ P \implies cp\ Q \implies cp\ R \implies cp(\lambda X.f (PX) (QX) (RX))\)

lemma cpI4: 
\( (\forall W X Y Z\ \tau. fWXYZ\tau = f(\lambda\ W\ \tau)(\lambda\ X\ \tau)(\lambda\ Y\ \tau)(\lambda\ Z\ \tau)\ \tau) \implies cp\ P \implies cp\ Q \implies cp\ R \implies cp\ S \implies cp(\lambda X.f (PX) (QX) (RX) (SX))\)

lemma cp-const: \( cp(\lambda\ -\ c)\)

lemma cp-id: \( cp(\lambda X.X)\)
OCL's if then else endif

**definition** OclIf :: [($\alpha$)Boolean , ($\alpha$::null) val, ($\alpha$) val] => ($\alpha$) val
(if (-) then (-) else (-) endif) [10,10,10,50]

**where** (if C then B1 else B2 endif) = (λ τ. if (δ C) τ = true τ then (C τ) = true τ then B1 τ else B2 τ) else invalid τ

**lemma** cp-OclIf:((if C then B1 else B2 endif) τ = (if (λ τ.-. C τ) then (λ τ.-. B1 τ) else (λ τ.-. B2 τ) endif) τ)

**lemma** OclIf-invalid [simp]: (if invalid then B1 else B2 endif) = invalid

**lemma** OclIf-null [simp]: (if null then B1 else B2 endif) = invalid

**lemma** OclIf-true [simp]: (if true then B1 else B2 endif) = B1

**lemma** OclIf-true' [simp]: τ |= P => (if P then B1 else B2 endif) τ = B1 τ

**lemma** OclIf-true'' [simp]: τ |= P => τ |= (if P then B1 else B2 endif) ≜ B1

**lemma** OclIf-true' [simp]: τ |= not P => (if P then B1 else B2 endif) τ = B2 τ

**lemma** OclIf-true'' [simp]: τ |= not P => (if P then B1 else B2 endif) τ = B2 τ

**lemma** OclIf-true'' [simp]: (if δ X then A else A endif) = A

**lemma** OclIf-true'' [simp]: (if υ X then A else A endif) = A

**lemma** OclNot-if [simp]: not(if P then C else E endif) = (if P then not C else not E 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** StrictRefEq : [(∀$\alpha$)val,(∀$\alpha$)val] => (∀$\alpha$)Boolean (infix = 30)

with term "not" we can express the notation:

**syntax** notequal :: (∀$\alpha$)Boolean => (∀$\alpha$)Boolean => (∀$\alpha$)Boolean (infix <> 40)

**translations** a <> b == CONST OclNot(a = b)

We will define instances of this equality in a case-by-case basis.
Laws to Establish Definedness ($\delta$-closure)

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

**Lemma OclNot-defargs:**
$\tau \models (\neg P) \Rightarrow \tau \models \delta P$

**Lemma OclNot-contrapos-nn:**
assumes $A$: $\tau \models \delta A$
assumes $B$: $\tau \models \neg B$
assumes $C$: $\tau \models A \Rightarrow \tau \models B$
shows $\tau \models \neg A$

A Side-calculus for Constant Terms

**Definition** $\text{const } X \equiv \forall \tau \tau' . X \tau = X \tau'$

**Lemma** $\text{const-charn: } const X \Rightarrow X \tau = X \tau'$

**Lemma** $\text{const-subst:}$
assumes $\text{const-X: } const X$
and $\text{const-Y: } const Y$
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** $\text{const-impl2:}$
assumes $\forall \tau \tau'. P \tau = P \tau' \Rightarrow Q \tau = Q \tau'$
shows $\text{const } P \Rightarrow \text{const } Q$

**Lemma** $\text{const-impl3:}$
assumes $\forall \tau \tau'. P \tau = P \tau' \Rightarrow Q \tau = Q \tau' \Rightarrow R \tau = R \tau'$
shows $\text{const } P \Rightarrow \text{const } Q \Rightarrow \text{const } R$

**Lemma** $\text{const-impl4:}$
assumes $\forall \tau \tau'. P \tau = P \tau' \Rightarrow Q \tau = Q \tau' \Rightarrow R \tau = R \tau' \Rightarrow S \tau = S \tau'$
shows $\text{const } P \Rightarrow \text{const } Q \Rightarrow \text{const } R \Rightarrow \text{const } S$

**Lemma** $\text{const-lam: } const \left( \lambda \cdot . e \right)$

**Lemma** $\text{const-true} [\text{simp}]: \text{const true}$

**Lemma** $\text{const-false} [\text{simp}]: \text{const false}$

**Lemma** $\text{const-null} [\text{simp}]: \text{const null}$

**Lemma** $\text{const-invalid} [\text{simp}]: \text{const invalid}$
lemma const-bot[simp] : const bot

lemma const-defined :
assumes const X
shows const (\delta X)

lemma const-valid :
assumes const X
shows const (\upsilon 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 \equiv 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 \( \text{const } x \)
shows \( (\tau \models \nu x) = (\tau' \models \nu x) \)

lemma \( \text{const-HOL-if} \) : \( \text{const } C \implies \text{const } D \implies \text{const } F \implies \text{const } (\lambda \tau. \text{if } C \tau \text{ then } D \tau \text{ else } F \tau) \)
lemma \( \text{const-HOL-and} \) : \( \text{const } C \implies \text{const } D \implies \text{const } (\lambda \tau. C \tau \land D \tau) \)
lemma \( \text{const-HOL-eq} \) : \( \text{const } C \implies \text{const } D \implies \text{const } (\lambda \tau. C \tau = D \tau) \)

lemmas \( \text{const-ss} = \text{const-bot} \) \( \text{const-null} \) \( \text{const-invalid} \) \( \text{const-false} \) \( \text{const-true} \) \( \text{const-lam} \)
\( \text{const-defined} \) \( \text{const-valid} \) \( \text{const-StrongEq} \) \( \text{const-OclNot} \) \( \text{const-OclAnd} \)
\( \text{const-OclOr} \) \( \text{const-OclImplies} \) \( \text{const-OclIf} \)
\( \text{const-HOL-if} \) \( \text{const-HOL-and} \) \( \text{const-HOL-eq} \)

Miscellaneous: Overloading the syntax of “bottom”

notation \( \text{bot} \) (\( \bot \))

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 \( \text{profile-mono-schemeD} \) =
fixes \( f :: (\forall \alpha::\text{null})\text{val} \Rightarrow (\forall \beta::\text{null})\text{val} \)
fixes \( g \)
assumes \( \text{def-scheme}: (f x) \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \text{ then } g (x \tau) \text{ else invalid } \tau \)
begin
lemma \( \text{strict[simp,code-unfold]}: f \text{ invalid} = \text{invalid} \)
lemma \( \text{null-strict[simp,code-unfold]}: f \text{ null} = \text{invalid} \)
lemma \( \text{cp0 } : f \ X \tau = f (\lambda - \ X \tau) \tau \)
lemma \( \text{cp[simp,code-unfold]} : \text{ cp } P \Rightarrow \text{ cp } (\lambda X. f (P X) ) \)
end

locale \( \text{profile-mono-schemeV} \) =
fixes \( f :: (\forall \alpha::\text{null})\text{val} \Rightarrow (\forall \beta::\text{null})\text{val} \)
fixes \( g \)
assumes \( \text{def-scheme}: (f x) \equiv \lambda \tau. \text{if } (\nu x) \tau = \text{true } \tau \text{ then } g (x \tau) \text{ else invalid } \tau \)
begin
lemma \( \text{strict[simp,code-unfold]}: f \text{ invalid} = \text{invalid} \)
lemma \( \text{cp0 } : f \ X \tau = f (\lambda - \ X \tau) \tau \)
lemma \( \text{cp[simp,code-unfold]} : \text{ cp } P \Rightarrow \text{ cp } (\lambda X. f (P X) ) \)


locale profile-mono2 = profile-mono-schemeD +
assumes $\land x. x \neq \text{bot} \implies x \neq \text{null} \implies g x \neq \text{bot}$
begin

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

locale profile-mono0 = profile-mono-schemeD +
assumes def-body: $\land x. x \neq \text{bot} \implies x \neq \text{null} \implies g x \neq \text{bot}$

sublocale profile-mono0 < profile-mono2

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

lemma def-valid-then-def: $\nu(f x) = (\delta(f x))$
end

Property Profiles for Single

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

Property Profiles for Binary Operators

definition bin f g d x d y X Y =
  $(f X Y = (\lambda \tau. \text{if } (d x X) \tau = \text{true } \tau \land (d y Y) \tau = \text{true } \tau \text{ then } g X Y \tau \text{ else invalid } \tau))$

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

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

locale profile-bin-scheme =
  fixes $d_x :: (\forall a::\text{null})\text{val} \Rightarrow \forall \text{Boolean}$
  fixes $d_y :: (\forall b::\text{null})\text{val} \Rightarrow \forall \text{Boolean}$
  fixes $f :: (\forall a::\text{null})\text{val} \Rightarrow (\forall b::\text{null})\text{val} \Rightarrow (\forall c::\text{null})\text{val}$
  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) \Rightarrow cp (\lambda x. f x Y) \Rightarrow f X \text{invalid} = \text{invalid} \Rightarrow
\[ f \text{ invalid } Y = \text{ invalid } \implies \]
\[ (\neg (\tau \models d_\text{x} X) \lor (\tau \models d_\text{y} Y)) \implies \]
\[ \tau \models (\delta f X Y \triangleq (d_\text{x} X \text{ and } d_\text{y} Y)) \]

assumes def-scheme'\[\text{simplified}]: \text{bin } f \text{ } g \text{ } d_\text{x} \text{ } d_\text{y} \text{ } X \text{ } Y

assumes 1: \[ \tau \models d_\text{x} X \implies \tau \models d_\text{y} Y \implies \tau \models \delta f X Y \]

begin
interpretation \[d_\text{x} : \text{profile-single } d_\text{x}\]
interpretation \[d_\text{y} : \text{profile-single } d_\text{y}\]

lemma strict1[simp,code-unfold]: \[ f \text{ invalid } y = \text{ invalid } \]

lemma strict2[simp,code-unfold]: \[ f x \text{ invalid } = \text{ invalid } \]

lemma cp0: \[ f X Y \tau = f (\lambda X. \tau) (\lambda Y. \tau) \tau \]

lemma cp[simp,code-unfold]: \[ cp P \implies cp Q \implies cp (\lambda x. f (P X) (Q X)) \]

lemma def-homo[simp,code-unfold]: \[ \delta(f x y) = (d_\text{x} x \text{ and } d_\text{y} y) \]

lemma def-valid-then-def: \[ \upsilon(f x y) = (\delta(f x y)) \]

lemma defined-args-valid: \[ (\tau \models (f x y)) = ((\tau \models d_\text{x} x) \land (\tau \models d_\text{y} y)) \]

lemma const[simp,code-unfold]:
  assumes \[ C1 : \text{const } X \text{ and } C2 : \text{const } Y \]
  shows \[ \text{const}(f X Y) \]
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 \( f \text{ } g \) we know that it satisfies a number of properties like strict1 or strict2 i.e. \( f \text{ invalid } y = \text{ invalid } \) and \( f \text{ null } y = \text{ invalid } \). Since some of the more advanced Locales come with 10 - 15 theorems, property profiles represent a major structuring mechanism for the OCL library.

locale profile-bin-scheme-defined =
  fixes \[d_\text{x} : \text{profile-single } d_\text{x}\]
  fixes \[d_\text{y} : \text{profile-single } d_\text{y}\]
  assumes d1-homo[simp,code-unfold]: \[ cp (f X) \implies \]
  \[ f X \text{ invalid } = \text{ invalid } \implies \]
  \[ \neg \tau \models d_\text{y} Y \implies \]
  \[ \tau \models \delta f X Y \triangleq (d_\text{x} X \text{ and } d_\text{y} Y) \]
  assumes def-scheme'\[\text{simplified}]: \text{bin } f \text{ } g \text{ } d_\text{x} \text{ } X \text{ } Y
  assumes def-body\[': \land x y. \tau. x \neq \text{bot} \implies x \neq \text{null} \implies (d_\text{x} y) \tau = \text{true} \tau \implies g x (y \tau) \neq \text{bot} \land g x (y \tau) \neq \text{null} \]
begin
  lemma strict3[simp,code-unfold]: \[ f \text{ null } y = \text{ invalid } \]
end sublocale profile-bin-scheme-defined < profile-bin-scheme defined

locale profile-bin1 =
  fixes \[f :: (\forall A, a :: \text{null}) \forall x \Rightarrow (\forall A, b :: \text{null}) \forall x \Rightarrow (\forall A, c :: \text{null}) \forall x \]

Draft Proposal
fixes $g$
assumes def-scheme[simplified]: bin $f$ $g$ defined defined $X$ $Y$
assumes def-body: $\land x y. g x y \neq \text{bot} \land g x y \neq \text{null}$
begin
  lemma strict4[simp,code-unfold]: $f$ $\text{null} = \text{invalid}$
end

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

locale profile-bin2 =
  fixes $f$ :: $\langle \mathbb{A}, \alpha :: \text{null} \rangle \text{val} \Rightarrow \langle \mathbb{A}, \beta :: \text{null} \rangle \text{val} \Rightarrow \langle \mathbb{A}, \gamma :: \text{null} \rangle \text{val}$
  fixes $g$
  assumes def-scheme[simplified]: bin $f$ $g$ defined valid $X$ $Y$
  assumes def-body: $\land x y. x \neq \text{bot} = \Rightarrow x \neq \text{null} = \Rightarrow y \neq \text{bot} = \Rightarrow g x y \neq \text{bot} \land g x y \neq \text{null}$

sublocale profile-bin2 < profile-bin-scheme-defined valid

locale profile-bin3 =
  fixes $f$ :: $\langle \mathbb{A}, \alpha :: \text{null} \rangle \text{val} \Rightarrow \langle \mathbb{A}, \alpha :: \text{null} \rangle \text{val} \Rightarrow \langle \mathbb{A}, \text{Boolean} \rangle$
  assumes def-scheme[simplified]: bin $f$ StrongEq valid valid $X$ $Y$
sublocale profile-bin3 < profile-bin-scheme valid valid $f$
  \begin{align*}
    \text{context profile-bin3} \begin{align*}
    \text{begin} & \\
    \text{lemma idem[simp,code-unfold]: $f$ null null = true} & \\
    \text{lemma defargs} : \tau \models f x y \Longrightarrow (\tau \models \upsilon x) \land (\tau \models \upsilon y) & \\
    \text{lemma defined-args-valid'} : \delta (f x y) = (\upsilon x \text{ and } \upsilon y) & \\
    \text{lemma refl-ext[simp,code-unfold]} : (f x x) = (\text{if } (\upsilon x) \text{ then true else invalid endif}) & \\
    \text{lemma sym} : \tau \models (f x y) \Longrightarrow \tau \models (f y x) & \\
    \text{lemma symmetric} : (f x y) = (f y x) & \\
    \text{lemma trans} : \tau \models (f x y) \Longrightarrow \tau \models (f y z) \Longrightarrow \tau \models (f x z) & \\
    \text{lemma StrictRefEq-vs-StrongEq: } \tau \models (\upsilon x) \Longrightarrow \tau \models (\upsilon y) \Longrightarrow (\tau \models ((f x y) \triangleq (x \triangle y))) & \\
    \end{align*}
\end{align*}
\end{align*}

end

locale profile-bin4 =
  fixes $f$ :: $\langle \mathbb{A}, \alpha :: \text{null} \rangle \text{val} \Rightarrow \langle \mathbb{A}, \beta :: \text{null} \rangle \text{val} \Rightarrow \langle \mathbb{A}, \gamma :: \text{null} \rangle \text{val}$
  fixes $g$
  assumes def-scheme[simplified]: bin $f$ $g$ valid valid $X$ $Y$
  assumes def-body: $\land x y. x \neq \text{bot} = \Rightarrow y \neq \text{bot} = \Rightarrow g x y \neq \text{bot} \land g x y \neq \text{null}$

Draft Proposal
**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 \( \forall \text{ Boolean} \), it is just the strict extension of the logical equality:

```plaintext
defs StrictRefEqBoolean [code-unfold] :
    (x::(\forall)Boolean) \equiv y \equiv \lambda \tau. if (\lor x) \tau = true \land (\lor y) \tau = true \tau
    then (x \equiv y) \tau
    else invalid \tau
```

which implies elementary properties like:

```plaintext
lemma [simp, code-unfold] : (true \equiv false) = false
lemma [simp, code-unfold] : (false \equiv true) = false
lemma null-non-false [simp, code-unfold] : (null \equiv false) = false
lemma null-non-true [simp, code-unfold] : (null \equiv true) = false
lemma false-non-null [simp, code-unfold] : (false \equiv null) = false
lemma true-non-null [simp, code-unfold] : (true \equiv null) = false
```

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

```plaintext
interpretation StrictRefEqBoolean : profile-bin3 \lambda x y. (x::(\forall)Boolean) \equiv y
```

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

```plaintext
lemma (invalid \equiv false) = invalid
lemma (invalid \equiv true) = invalid
lemma (false \equiv invalid) = invalid
lemma (true \equiv invalid) = 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 \( \tau \models \nu(true) \)
Assert \( \tau \models \delta(false) \)
Assert \( \neg(\tau \models \delta(nullptr)) \)
Assert \( \neg(\tau \models \delta(invalid)) \)
Assert \( \tau \models \nu(nullptr::(\forall)Boolean) \)
Assert \( \neg(\tau \models \nu(invalid)) \)

48
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 Voidbase :: bot
begin
definition bot-Void-def: (bot-class.bot :: Voidbase) ≡ Abs-Voidbase None

instance
end

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

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 \( \forall \text{Void-case} \) as strict extension of the strong equality:

defs StrictRefEqVoid[code-unfold] :
\[
(x::(\forall)\text{Void}) \equiv y \equiv \lambda \tau. \text{ if } (\forall x) \tau = \text{true} \land (\forall y) \tau = \text{true} \text{ then } (x \equiv y) \tau \\
\text{else invalid } \tau
\]

Property proof in terms of profile-bin3

interpretation StrictRefEqVoid : profile-bin3 \( \lambda \ x \ y. \ (x::(\forall)\text{Void}) \equiv y \)

Test Statements

Assert \( \tau \models ((\text{null}::(\forall)\text{Void}) \equiv \text{null}) \)
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 Boolean-case as strict extension of the strong equality:

\[
\text{defs } \text{StrictRefEqInteger}[\text{code-unfold}] :
(\mathfrak{x} : (A)\text{Integer}) \doteq y \equiv \lambda \tau. \text{if } (\mathfrak{v} \mathfrak{x}) \tau = \text{true } \tau \land (\mathfrak{v} y) \tau = \text{true } \tau 
\text{then } (x \triangleq y) \tau 
\text{else invalid } \tau
\]

Property proof in terms of profile-bin3

interpretation \text{StrictRefEqInteger} : profile-bin3 \lambda \mathfrak{x} y. (\mathfrak{x} : (A)\text{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.

\[
\text{definition } \text{OclInt0} : (A)\text{Integer} (0) \text{ where } 0 = (\lambda - \text{int}_0)
\]

\[
\text{definition } \text{OclInt1} : (A)\text{Integer} (1) \text{ where } 1 = (\lambda - \text{int}_1)
\]

\[
\text{definition } \text{OclInt2} : (A)\text{Integer} (2) \text{ where } 2 = (\lambda - \text{int}_2)
\]

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.

\[
\text{definition } \text{OclAddInteger} : (A)\text{Integer} \Rightarrow (A)\text{Integer} \Rightarrow (A)\text{Integer} \text{ infix } +_{\text{int}} 40
\]

\[
\text{where } x +_{\text{int}} y \equiv \lambda \tau. \text{if } (\mathfrak{d} x) \tau = \text{true } \tau \land (\mathfrak{d} y) \tau = \text{true } \tau 
\text{then } \mathfrak{t} x \tau + \mathfrak{t} y \tau \text{ else invalid } \tau
\]

interpretation \text{OclAddInteger} : profile-bin1 op +_{\text{int}} \lambda \mathfrak{x} y. \mathfrak{t} x \tau + \mathfrak{t} y \tau

\[
\text{definition } \text{OclMinusInteger} : (A)\text{Integer} \Rightarrow (A)\text{Integer} \Rightarrow (A)\text{Integer} \text{ infix } -_{\text{int}} 41
\]

\[
\text{where } x -_{\text{int}} y \equiv \lambda \tau. \text{if } (\mathfrak{d} x) \tau = \text{true } \tau \land (\mathfrak{d} y) \tau = \text{true } \tau 
\text{then } \mathfrak{t} x \tau - \mathfrak{t} y \tau \text{ else invalid } \tau
\]

interpretation \text{OclMinusInteger} : profile-bin1 op -_{\text{int}} \lambda \mathfrak{x} y. \mathfrak{t} x \tau - \mathfrak{t} y \tau

\[
\text{definition } \text{OclMultInteger} : (A)\text{Integer} \Rightarrow (A)\text{Integer} \Rightarrow (A)\text{Integer} \text{ infix } *_{\text{int}} 45
\]

\[
\text{where } x *_{\text{int}} y \equiv \lambda \tau. \text{if } (\mathfrak{d} x) \tau = \text{true } \tau \land (\mathfrak{d} y) \tau = \text{true } \tau 
\text{then } \mathfrak{t} x \tau * \mathfrak{t} y \tau \text{ else invalid } \tau
\]

interpretation \text{OclMultInteger} : profile-bin1 op *_{\text{int}} \lambda \mathfrak{x} y. \mathfrak{t} x \tau * \mathfrak{t} y \tau

50
Here is the special case of division, which is defined as invalid for division by zero.

definition OclDivisionInt :: (\A)Integer \Rightarrow (\A)Integer \Rightarrow (\A)Integer (infix div_int 45)
where \( x \text{ div}_\text{int} y \equiv \lambda \tau. \text{ if } (\delta x) \tau = \text{ true } \text{ and } (\delta y) \tau = \text{ true } \tau \)
then if \( y \tau \neq Ocl\text{Int0} \tau \) then \( \lceil \lfloor x \tau \text{ div}_\text{int} y \tau \rfloor \rfloor \) else invalid \( \tau \)
else invalid \( \tau \)

definition OclModulusInt :: (\A)Integer \Rightarrow (\A)Integer \Rightarrow (\A)Integer (infix mod_int 45)
where \( x \text{ mod}_\text{int} y \equiv \lambda \tau. \text{ if } (\delta x) \tau = \text{ true } \text{ and } (\delta y) \tau = \text{ true } \tau \)
then if \( y \tau \neq Ocl\text{Int0} \tau \) then \( \lceil \lfloor x \tau \text{ mod}_\text{int} y \tau \rfloor \rfloor \) else invalid \( \tau \)
else invalid \( \tau \)

definition OclLessInt :: (\A)Integer \Rightarrow (\A)Integer \Rightarrow (\A)Boolean (infix <_\text{int} 35)
where \( x <_{\text{int}} y \equiv \lambda \tau. \text{ if } (\delta x) \tau = \text{ true } \text{ and } (\delta y) \tau = \text{ true } \tau \)
then \( \lceil \lfloor x \tau \text{ < } y \tau \rfloor \rfloor = \text{ true } \)
else invalid \( \tau \)
interpretation OclLessInt : profile-bin1 op <_{\text{int}} \lambda x y \lceil x \tau \text{ < } y \tau \rfloor \)

definition OclLeInt :: (\A)Integer \Rightarrow (\A)Integer \Rightarrow (\A)Boolean (infix \leq_{\text{int} 35})
where \( x \leq_{\text{int}} y \equiv \lambda \tau. \text{ if } (\delta x) \tau = \text{ true } \text{ and } (\delta y) \tau = \text{ true } \tau \)
then \( \lceil \lfloor x \tau \text{ \leq } y \tau \rfloor \rfloor = \text{ true } \)
else invalid \( \tau \)
interpretation OclLeInt : profile-bin1 op \leq_{\text{int}} \lambda x y \lceil x \tau \text{ \leq } y \tau \rfloor \)

Basic Properties

lemma OclAddInt-commute: (X + int Y) = (Y + int X)

Execution with Invalid or Null or Zero as Argument

lemma OclAddInt-zero1[simp,code-unfold]:
\((x + \text{int } 0) = (\text{if } \lor x \text{ and not } (\delta x) \text{ then invalid else } x \text{ endif})\)

lemma OclAddInt-zero2[simp,code-unfold]:
\((0 + \text{int } x) = (\text{if } \lor x \text{ and not } (\delta x) \text{ then invalid else } x \text{ endif})\)

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 \( \tau \models (9 \leq_{\text{int}} 10) \)
assert \( \tau \models ((4 +_{\text{int}} 4) \leq_{\text{int}} 10) \)
assert \( \neg(\tau \models ((4 +_{\text{int}} (4 +_{\text{int}} 4)) <_{\text{int}} 10)) \)
assert \( \tau \models \text{ not } (\lor (\text{null } +_{\text{int}} 1)) \)
assert \( \tau \models (((9 *_{\text{int}} 4) \text{ div}_\text{int} 10 \leq_{\text{int}} 4) \)
assert \( \tau \models \text{ not } (\delta (1 \text{ div}_\text{int} 0)) \)
assert \( \tau \models \text{ not } (\lor (1 \text{ div}_\text{int} 0)) \)
lemma integer-non-null [simp]: \((\lambda - \_\_ \_ n \_ ) \dashv (null::(\forall)\text{Integer}) = false\)

lemma null-non-integer [simp]: \((null::(\forall)\text{Integer}) \dashv (\lambda - \_\_ \_ n \_ )) = false\)

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

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

Elementary computations on Integer

\(\text{Assert}\quad \tau \models (0 <_{\text{int}} 2) \text{ and } (0 <_{\text{int}} 1)\)

\(\text{Assert}\quad \tau \models 1 <> 2\)
\(\text{Assert}\quad \tau \models 2 <> 1\)
\(\text{Assert}\quad \tau \models 2 = 2\)

\(\text{Assert}\quad \tau \models \_v 4\)
\(\text{Assert}\quad \tau \models \_d 4\)
\(\text{Assert}\quad \tau \models \_v (null::(\forall)\text{Integer})\)
\(\text{Assert}\quad \tau \models (invalid \_d invalid)\)
\(\text{Assert}\quad \tau \models (null \_d null)\)
\(\text{Assert}\quad \tau \models (4 \_d 4)\)
\(\text{Assert}\quad \neg (\tau \models (9 \_d 10))\)
\(\text{Assert}\quad \neg (\tau \models (invalid \_d 10))\)
\(\text{Assert}\quad \neg (\tau \models (null \_d 10))\)
\(\text{Assert}\quad \neg (\tau \models (invalid \_d (invalid::(\forall)\text{Integer})))\)
\(\text{Assert}\quad \neg (\tau \models (invalid <> (invalid::(\forall)\text{Integer})))\)
\(\text{Assert}\quad \neg (\tau \models (invalid <> (invalid::(\forall)\text{Integer})))\)
\(\text{Assert}\quad \tau \models (null \_d (null::(\forall)\text{Integer}))\)
\(\text{Assert}\quad \tau \models (null \_d (null::(\forall)\text{Integer}))\)
\(\text{Assert}\quad \tau \models (4 \_d 4)\)
\(\text{Assert}\quad \neg (\tau \models (4 <> 4))\)
\(\text{Assert}\quad \neg (\tau \models (4 \_d 10))\)
\(\text{Assert}\quad \tau \models (4 <> 10)\)
\(\text{Assert}\quad \neg (\tau \models (0 <_{\text{int}} null))\)
\(\text{Assert}\quad \neg (\tau \models (\_d (0 <_{\text{int}} null)))\)

52
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 Boolean case as strict extension of the strong equality:

\[
\text{defs } \text{StrictRefEqReal [code-unfold]} : \\
(x :: (\forall\mathcal{A}) \text{Real}) \equiv y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau \\
\text{then } (x \equiv y) \tau \\
\text{else invalid } \tau
\]

Property proof in terms of profile-bin3 interpretation

\[
\text{StrictRefEqReal : profile-bin3 } x y. (x :: (\forall\mathcal{A}) \text{Real}) \equiv y
\]

Basic Real Constants

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

\[
\text{definition OclReal0 } :: (\forall\mathcal{A}) \text{Real} (0.0) \text{ where } 0.0 = (\lambda - . \{0 \cdot \text{real}_0\})
\]

\[
\text{definition OclReal1 } :: (\forall\mathcal{A}) \text{Real} (1.0) \text{ where } 1.0 = (\lambda - . \{1 \cdot \text{real}_1\})
\]

\[
\text{definition OclReal2 } :: (\forall\mathcal{A}) \text{Real} (2.0) \text{ where } 2.0 = (\lambda - . \{2 \cdot \text{real}_2\})
\]

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.

\[
\text{definition OclAddReal } :: (\forall\mathcal{A}) \text{Real } \Rightarrow (\forall\mathcal{A}) \text{Real } \Rightarrow (\forall\mathcal{A}) \text{Real } (\text{infix } +\text{real } 40) \\
\text{where } x +\text{real } y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau \\
\text{then } (\uplus x \tau +\uplus y \tau) \tau \\
\text{else invalid } \tau
\]

interpretation OclAddReal : profile-bin1 op +\text{real } \lambda x y. (\uplus x \tau +\uplus y \tau) \tau

\[
\text{definition OclMinusReal } :: (\forall\mathcal{A}) \text{Real } \Rightarrow (\forall\mathcal{A}) \text{Real } \Rightarrow (\forall\mathcal{A}) \text{Real } (\text{infix } -\text{real } 41) \\
\text{where } x -\text{real } y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau \\
\text{then } (\uplus x \tau -\uplus y \tau) \tau \\
\text{else invalid } \tau
\]

interpretation OclMinusReal : profile-bin1 op -\text{real } \lambda x y. (\uplus x \tau -\uplus y \tau) \tau

\[
\text{definition OclMultReal } :: (\forall\mathcal{A}) \text{Real } \Rightarrow (\forall\mathcal{A}) \text{Real } \Rightarrow (\forall\mathcal{A}) \text{Real } (\text{infix } \ast\text{real } 45) \\
\text{where } x \ast\text{real } y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau \\
\text{then } (\uplus x \tau \ast\uplus y \tau) \tau \\
\text{else invalid } \tau
\]

interpretation OclMultReal : profile-bin1 op \ast\text{real } \lambda x y. (\uplus x \tau \ast\uplus y \tau) \tau

53
Here is the special case of division, which is defined as invalid for division by zero.

**definition** 
\( \text{OclDivisionReal} :: (\forall \alpha) \alpha \rightarrow (\forall \alpha) \alpha \rightarrow (\forall \alpha) \alpha \) (infix \( \div \text{real} \) 45)

where \( x \div_{\text{real}} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau \)
then if \( y \tau \neq OclReal0 \tau \text{ then } \langle x \tau, y \tau \rangle \rangle \text{ else invalid } \tau \)
else invalid \( \tau \)

**definition** 
\( \text{mod-float} a = a - \text{real} \) (floor \( (a / b) \)) \( * b \)

**definition** 
\( \text{OclModulusReal} :: (\forall \alpha) \alpha \rightarrow (\forall \alpha) \alpha \rightarrow (\forall \alpha) \alpha \) (infix \( \text{mod-float} \) 45)

where \( x \text{mod-float} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau \)
then if \( y \tau \neq OclReal0 \tau \text{ then } \langle x \tau, y \tau \rangle \rangle \text{ else invalid } \tau \)
else invalid \( \tau \)

**interpretation** 
\( \text{OclLessReal} : \text{profile-bin1} \ op \langle \text{real} \rangle \ x \ y \uparrow \langle \text{real} \rangle \ y \uparrow \rangle \)

**definition** 
\( \text{OclLessReal} :: (\forall \alpha) \alpha \rightarrow (\forall \alpha) \alpha \rightarrow (\forall \alpha) \alpha \) (infix \( \leq \text{real} \) 35)

where \( x \leq_{\text{real}} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau \)
then \( \langle x \tau, y \tau \rangle \rangle \text{ else invalid } \tau \)
else invalid \( \tau \)

**interpretation** 
\( \text{OclLessReal} : \text{profile-bin1} \ op \leq_{\text{real}} \ x \ y \uparrow \langle \text{real} \rangle \ x \uparrow \) \( \uparrow \langle \text{real} \rangle \ y \uparrow \)

**Basic Properties**

**lemma** 
\( \text{OclAddReal-compose} : (x +_{\text{real}} Y) = (Y +_{\text{real}} X) \)

**Execution with Invalid or Null or Zero as Argument**

**lemma** 
\( \text{OclAddReal-zero1\[simp,code-unfold\]} : (x +_{\text{real}} 0.0) = (\text{if } \nu x \text{ and not } (\delta x) \text{ then invalid else } x \text{ endif}) \)

**lemma** 
\( \text{OclAddReal-zero2\[simp,code-unfold\]} : (0.0 +_{\text{real}} x) = (\text{if } \nu x \text{ and not } (\delta x) \text{ then invalid else } x \text{ 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 \( \text{True} \).

\[
\begin{align*}
\text{Assert } & \tau \vdash (9.0 \leq_{\text{real}} 10.0) \\
\text{Assert } & \tau \vdash ((4.0 +_{\text{real}} 4.0) \leq_{\text{real}} 10.0) \\
\text{Assert } & \neg (\tau \vdash ((4.0 +_{\text{real}} (4.0 +_{\text{real}} 4.0)) \leq_{\text{real}} 10.0)) \\
\text{Assert } & \tau \vdash \neg (\nu (\text{null} +_{\text{real}} 1.0)) \\
\text{Assert } & \tau \vdash \neg (\nu (9.0 \text{div}_{\text{real}} 10.0) \leq_{\text{real}} 4.0) \\
\text{Assert } & \tau \vdash \neg (\nu (\delta 1.0 \text{div}_{\text{real}} 0.0)) \\
\text{Assert } & \tau \vdash \neg (\nu (1.0 \text{div}_{\text{real}} 0.0)) \\
\end{align*}
\]
lemma real-non-null [simp]: \((\lambda x . x \downarrow) \doteq (\text{null} :: (\forall x) \text{Real})) = \text{false}

lemma null-non-real [simp]: \((\text{null} :: (\forall x) \text{Real}) \doteq (\lambda x . x \downarrow)\) = \text{false}

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

Here follows a list of code-examples, that explain the meanings of the above definitions by compilation to code and execution to \(\text{True}\).

Elementary computations on Real

\text{Assert} \tau \models 1.0 <> 2.0
\text{Assert} \tau \models 2.0 <> 1.0
\text{Assert} \tau \models 2.0 \doteq 2.0

\text{Assert} \tau \models \nu 4.0
\text{Assert} \tau \models \delta 4.0
\text{Assert} \tau \models \nu (\text{null} :: (\forall x) \text{Real})
\text{Assert} \tau \models (\text{invalid} \doteq \text{invalid})
\text{Assert} \tau \models (\text{null} \doteq \text{null})
\text{Assert} \tau \models (4.0 \doteq 4.0)
\text{Assert} \neg(\tau \models (9.0 \doteq 10.0))
\text{Assert} \neg(\tau \models (\text{invalid} \doteq 10.0))
\text{Assert} \neg(\tau \models (\text{null} \doteq 10.0))
\text{Assert} \neg(\tau \models (\text{invalid} \doteq (\text{invalid} :: (\forall x) \text{Real})))
\text{ Assert} \neg(\tau \models (\text{invalid} \doteq (\text{invalid} :: (\forall x) \text{Real})))
\text{ Assert} \neg(\tau \models (\text{invalid} <> (\text{invalid} :: (\forall x) \text{Real})))
\text{ Assert} \tau \models (\text{null} \doteq (\text{null} :: (\forall x) \text{Real}))
\text{ Assert} \tau \models (\text{null} \doteq (\text{null} :: (\forall x) \text{Real}))
\text{ Assert} \tau \models (4.0 \doteq 4.0)
\text{ Assert} \neg(\tau \models (4.0 <> 4.0))
\text{ Assert} \neg(\tau \models (4.0 \doteq 10.0))
\text{ Assert} \tau \models (4.0 <> 10.0)
\text{ Assert} \neg(\tau \models (0.0 < \text{real} \text{null}))
\text{ Assert} \neg(\tau \models (\delta (0.0 < \text{real} \text{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 \(\mathbb{A} Boolean\)-case as strict extension of the strong equality:

```plaintext
defs  StrictRefEqString\[code-unfold\] :
(x::('A)String) \equiv y \equiv \lambda \tau. \text{if } (\forall x) \tau = true \land (\forall y) \tau = true \tau
then (x \equiv y) \tau
else invalid \tau
```

Property proof in terms of profile-bin3

```plaintext
interpretation StrictRefEqString : profile-bin3 \lambda. x y. (x::('A)String) \equiv y
```

Basic String Constants

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

```plaintext
definition OclStringa ::= ('A)String (a) where a = (\lambda -. "a"")
definition OclStringb ::= ('A)String (b) where b = (\lambda -. "b")
definition OclStringc ::= ('A)String (c) where c = (\lambda -. "c")

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.

```plaintext
definition OclAddString ::= ('A)String + ('A)String \Rightarrow ('A)String \(\text{infix} +_{string} 40\)
where \(x +_{string} y \equiv \lambda \tau. \text{if } (\delta x) \tau = true \land (\delta y) \tau = true \tau
then \text{concat } [\tau x \tau], [\tau y \tau]_{\text{concat}}
else invalid \tau
```

```plaintext
interpretation OclAddString : profile-bin1 op +_{string} \lambda. x y. \text{concat } [\tau x \tau], [\tau y \tau]_{\text{concat}}
```

Basic Properties lemma OclAddString-not-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

```plaintext
Assert \tau |= a <> b
```
Assert \( \tau \vdash b <> a \)
Assert \( \tau \vdash b \triangleq b \)

Assert \( \tau \vdash \nu \ a \)
Assert \( \tau \vdash \delta \ a \)
Assert \( \tau \vdash (null \triangleq \nu \ String) \)
Assert \( \tau \vdash (invalid \triangleq invalid) \)
Assert \( \tau \vdash (null \triangleq null) \)
Assert \( \tau \vdash (a \triangleq a) \)
Assert \( \neg (\tau \vdash (a \triangleq b)) \)
Assert \( \neg (\tau \vdash (invalid \triangleq b)) \)
Assert \( \neg (\tau \vdash (null \triangleq b)) \)
Assert \( \neg (\tau \vdash (invalid \triangleq (invalid :: String))) \)
Assert \( \neg (\tau \vdash (invalid \triangleq (invalid :: String))) \)
Assert \( \neg (\tau \vdash (invalid <> (invalid :: String))) \)
Assert \( \tau \vdash (null \triangleq (null :: String)) \)
Assert \( \tau \vdash (null \triangleq (null :: String)) \)
Assert \( \tau \vdash (b \triangleq b) \)
Assert \( \neg (\tau \vdash (b <> b)) \)
Assert \( \neg (\tau \vdash (b \triangleq c)) \)
Assert \( \tau \vdash (b <> c) \)

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 Feather-Weight OCL, only the theory of a special case is developed, namely the type of Pairs, which is, however, sufficient for all applications since it can be used to mimic 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]**: \( \text{Rep-Pair}_{base} x \neq \text{None} \rightarrow \text{Rep-Pair}_{base} x \neq \text{null} \rightarrow (fs t \text{Rep-Pair}_{base} x) \neq \text{bot} \)

**lemma A' [simp]**: \( x \neq \text{bot} \rightarrow x \neq \text{null} \rightarrow (fs t \text{Rep-Pair}_{base} x) \neq \text{bot} \)

**lemma B [simp]**: \( \text{Rep-Pair}_{base} x \neq \text{None} \rightarrow \text{Rep-Pair}_{base} x \neq \text{null} \rightarrow (snd \text{Rep-Pair}_{base} x) \neq \text{bot} \)

**lemma B' [simp]**: \( x \neq \text{bot} \rightarrow x \neq \text{null} \rightarrow (snd \text{Rep-Pair}_{base} x) \neq \text{bot} \)

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** \( \text{StrictRefEq}_{pair} : ((x::(\text{null}::\alpha::\text{null})::(\text{null}::\beta::\text{null})::\text{Pair}) \triangleq y) \equiv (\lambda \tau. \text{if } (\nu x) \tau = true \tau \land (\nu y) \tau = true \tau \text{ then } (x \triangleq y) \tau) \)
Property proof in terms of profile-bin3

**interpretation**  \( \text{StrictRefEq}_{\text{Pair}} : \text{profile-bin3} \lambda \ x \ y. \ (x : (\mathcal{X}, \alpha :: \text{null}, \beta :: \text{null}) \text{Pair}) \Downarrow y \)

**Standard Operations Definitions**

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

**Definition: Pair Constructor**  \( \text{definition} \ \text{OclPair} : (\mathcal{X}, \alpha) \ \text{val} \Rightarrow (\mathcal{X}, \alpha :: \text{null}, \beta :: \text{null}) \ \text{Pair} \ ((\mathcal{X}, \alpha :: \text{null}, \beta :: \text{null}) \ \text{Pair} \ ((\mathcal{X}, \alpha :: \text{null}, \beta :: \text{null}) \ \text{Pair}) \ \downarrow)

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

**interpretation**  \( \text{OclPair} : \text{profile-bin4} \ \text{OclPair} \lambda \ x \ y. \ \text{Abs-Pair}_{\text{base}}(X, Y) \Downarrow \)

**Definition: First**  \( \text{definition} \ \text{OclFirst} : (\mathcal{X}, \alpha :: \text{null}, \beta :: \text{null}) \ \text{Pair} \Rightarrow (\mathcal{X}, \alpha) \ \text{val} \ (\mathcal{X}, \alpha :: \text{null}) \ \text{First}()

\[
\begin{align*}
& \text{where} \quad X . \text{First}() \equiv (\lambda \tau. \text{if } (\delta X) \tau = \text{true } \tau \quad \\
& \quad \quad \text{then } \text{fst } \text{Rep-Pair}_{\text{base}}(X \Downarrow \tau) \\
& \quad \quad \text{else invalid } \tau)
\end{align*}
\]

**interpretation**  \( \text{OclFirst} : \text{profile-mono2} \ \text{OclFirst} \lambda \ x. \ \text{fst } \text{Rep-Pair}_{\text{base}}(X) \Downarrow \)

**Definition: Second**  \( \text{definition} \ \text{OclSecond} : (\mathcal{X}, \alpha :: \text{null}, \beta :: \text{null}) \ \text{Pair} \Rightarrow (\mathcal{X}, \beta) \ \text{val} \ (\mathcal{X}, \beta :: \text{null}) \ \text{Second}()

\[
\begin{align*}
& \text{where} \quad X . \text{Second}() \equiv (\lambda \tau. \text{if } (\delta X) \tau = \text{true } \tau \quad \\
& \quad \quad \text{then } \text{snd } \text{Rep-Pair}_{\text{base}}(X \Downarrow \tau) \\
& \quad \quad \text{else invalid } \tau)
\end{align*}
\]

**interpretation**  \( \text{OclSecond} : \text{profile-mono2} \ \text{OclSecond} \lambda \ x. \ \text{snd } \text{Rep-Pair}_{\text{base}}(X) \Downarrow \)

**Logical Properties**

**lemma**  \( \lambda \tau \vdash \nu Y \Rightarrow \tau \vdash \text{Pair}(X, Y) . \text{First}() \Downarrow X \)

**lemma**  \( \lambda \tau \vdash \nu X \Rightarrow \tau \vdash \text{Pair}(X, Y) . \text{Second}() \Downarrow Y \)

**Algebraic Execution Properties**

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

**lemma**  \( \text{proj2-exec} [\text{simp}, \text{code-unfold}] : \text{Pair}(X, Y) . \text{Second}() = (\text{if } (\nu X) \text{ then } Y \text{ else invalid endif}) \)
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 $\tau \models \nu(Pair(null, true).First()) \triangleq null$
Assert $\tau \models (Pair(null, Pair(true, invalid))).First() \triangleq invalid$

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 occurring in a state — requires some care.

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

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

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

**definition** Integer :: ($\forall$.Integer_base) Set
**where** Integer $\equiv (\lambda \tau. (Abs-Set_base o Some o Some) ((Some o Some) \ \cdot \ (UNIV::int set)))$

**definition** Integer_{null} :: ($\forall$.Integer_base) Set
**where** Integer_{null} $\equiv (\lambda \tau. (Abs-Set_base o Some o Some) \ (Some \ \cdot \ (UNIV::int option set)))$

**lemma** Integer-defined : $\delta$ Integer $=$ true

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

This allows the theorems:

$\tau \models \delta x \implies \tau \models (Integer\rightarrow includes_{Set}(x)) \tau \models \delta x \implies \tau \models Integer \triangleq (Integer\rightarrow including_{Set}(x))$

and

$\tau \models \nu x \implies \tau \models (Integer_{null}\rightarrow includes_{Set}(x)) \tau \models \nu x \implies \tau \models Integer_{null} \triangleq (Integer_{null}\rightarrow including_{Set}(x))$

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: $\tau \models (\delta X) \implies \forall x \in \gamma Rep-Set_{base} (X \tau)^{\gamma}, x \neq bot$

**lemma** Set-inv-lemma':
assumes \( x \text{-def} : \tau \models \delta \ A \)
and \( e \text{-mem} : e \in ("\text{Rep-Set}_{\text{base}} (X \ \tau)^\tau") \)
shows \( \tau \models \nu (\lambda \cdot \ e) \)

lemma \textit{abs-rep-simp}: 
assumes \( \text{S-all-def} : \tau \models \delta \ S \)
shows \( \text{Abs-Set}_{\text{base}} = ("\text{Rep-Set}_{\text{base}} (S \ \tau)^\tau") = S \ \tau \)

lemma \textit{S-lift}:
assumes \( \text{S-all-def} : (\tau :: \ ' A \ st) \models \delta \ S \)
shows \( \exists S'. (\lambda a (:: (' A \ st) \ a) \cdot "\text{Rep-Set}_{\text{base}} (S \ \tau)^\tau") = (\lambda a (:: (' A \ st) \ a) ' S' \ \tau \)

lemma \textit{invalid-set-OclNot-defined} \( \text{simp,code-unfold}; \delta (\text{invalid} :: (\ ' A, ' A :: ('null) \ Set) \models false \)
lemma \textit{null-set-OclNot-defined} \( \text{simp,code-unfold}; \delta (\text{null} :: (\ ' A, ' A :: ('null) \ Set) \models false \)
lemma \textit{invalid-set-valid} \( \text{simp,code-unfold}; \nu (\text{invalid} :: (\ ' A, ' A :: ('null) \ Set) \models false \)
lemma \textit{null-set-valid} \( \text{simp,code-unfold}; \nu (\text{null} :: (\ ' A, ' A :: ('null) \ Set) \models true \)

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

**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:

\[
\text{defs} \quad \text{StrictRefEq}_{\text{Set}} : \quad (x :: (\ ' A, ' A :: ('null) \ Set) \models y \
\begin{align*}
\quad & \quad \text{then} (x \ \& \ y) \ \tau = \ true \ \& \ (y \ \tau) = \ true \\
\quad & \quad \text{else invalid} \ \tau
\end{align*}
\]

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 represent 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 \textit{profile-bin3}

interpretation \text{StrictRefEq}_{\text{Set}} : \text{profile-bin3} \quad \lambda x. (x :: (\ ' A, ' A :: ('null) \ Set) \models y

**Constants on Sets: mtSet**

definition \text{mtSet} : (\ ' A, ' A :: ('null) \ Set (Set (\}))

where \( \text{Set (\}) \models (\lambda \ t. \text{Abs-Set}_{\text{base} (\{} :: ' \text{set}_{\text{}}) }\)

lemma \text{mtSet-defined} \( \text{simp,code-unfold}; \delta (\text{Set (\}) \models true \)
lemma \text{mtSet-valid} \( \text{simp,code-unfold}; \nu (\text{Set (\}) \models true \)

60
lemma mtSet-rep-set: ’TRep-Set_{\text{base}} (\{\} \tau)^\tau = \{\}

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 :: [([\text{\&},'\alpha::null}) Set,(\text{\&},'\alpha) \text{ val}] \Rightarrow (\text{\&},'\alpha) \text{ Set}
where OclIncluding x y = (\lambda \tau. if (\delta x) \tau = true \land (\nu y) \tau = true \tau
then Abs-Set_{\text{base}} \bigcup^{\prime} (\tau) \text{Rep-Set}_{\text{base}} (x \tau)^\tau \cup \{y \tau\}_\bot
else invalid \tau

notation OclIncluding (\lambda \tau.\Rightarrow including) \lbrack\rightarrow\lbrack\}

interpretation OclIncluding : profile-bin2 OclIncluding \lambda x y. Abs-Set_{\text{base}} \bigcup^{\prime} (\tau) \text{Rep-Set}_{\text{base}} x^\tau \cup \{y\}_\bot

syntax
-OclFinset :: args => (\text{\&},'\alpha::null) Set (Set{})
translations
Set{} x x \Rightarrow CONST OclIncluding (Set{\{\}) x
Set{} x \Rightarrow CONST OclIncluding (Set{}) x

Definition: Excluding

definition OclExcluding :: [([\text{\&},'\alpha::null}) Set,(\text{\&},'\alpha) \text{ val}] \Rightarrow (\text{\&},'\alpha) \text{ Set}
where OclExcluding x y = (\lambda \tau. if (\delta x) \tau = true \tau \land (\nu y) \tau = true \tau
then Abs-Set_{\text{base}} \bigcup^{\prime} (\tau) \text{Rep-Set}_{\text{base}} (x \tau)^\tau \cup \{y \tau\}_\bot
else invalid \tau

notation OclExcluding (\lambda \tau.\Rightarrow excluding) \lbrack\rightarrow\lbrack\}

Definition: Includes

definition OclIncludes :: [([\text{\&},'\alpha::null}) Set,(\text{\&},'\alpha) \text{ val}] \Rightarrow \text{\& Boolean}
where OclIncludes x y = (\lambda \tau. if (\delta x) \tau = true \tau \land (\nu y) \tau = true \tau
then \{y \tau\} \in \bigcup^{\prime} (\tau) \text{Rep-Set}_{\text{base}} (x \tau)^\tau \cup \{y \tau\}_\bot
else invalid \tau

notation OclIncludes (\lambda \tau.\Rightarrow includes) \lbrack\rightarrow\lbrack\}

Definition: Excludes

definition OclExcludes :: [([\text{\&},'\alpha::null}) Set,(\text{\&},'\alpha) \text{ val}] \Rightarrow \text{\& Boolean}
where OclExcludes x y = \lnot (OclIncludes x y)
notation OclExcludes (\lambda \tau.\Rightarrow excludes) \lbrack\rightarrow\lbrack\}

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 :: (\text{\&},'\alpha::null)Set \Rightarrow \text{\& Integer}
\[
OclSize x = (\lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \land \text{finite}(\triangledown \text{Rep-Set}_{\text{base}} (x \tau)) \text{ then } \int \text{int}(\text{card}(\triangledown \text{Rep-Set}_{\text{base}} (x \tau))) \text{ else } \bot)
\]

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**: IsEmpty :: (\forall 'a::null) Set \Rightarrow 'a Boolean

where IsEmpty x = ((\forall x \text{ and } (\delta x)) \text{ or } ((OclSize x) = 0))

**notation**: IsEmpty (\text{-}\text{isEmpty} Set')

**Definition: NotEmpty**

**Definition**: NotEmpty :: (\forall 'a::null) Set \Rightarrow 'a Boolean

where NotEmpty x = \text{not} (IsEmpty x)

**notation**: NotEmpty (\text{-}\text{notEmpty} Set')

**Definition: Any**

**Definition**: Any :: [(\forall 'a::null) Set, \forall 'a::val] \Rightarrow (\forall 'a, 'a) val

where Any x = (\lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \text{ then } \text{SOME } y . y \in (\triangledown \text{Rep-Set}_{\text{base}} (x \tau)) \text{ else } \text{null } \tau \text{ else } \bot)

**notation**: Any (\text{-}\text{any} Set')

**Definition: Forall**

The definition of OclForall mimics the one of \text{op and}: OclForall is not a strict operation.

**Definition**: Forall :: [(\forall 'a::null) Set, (\forall 'a::val)] \Rightarrow (\forall 'a) Boolean

where Forall S P = (\lambda \tau. \text{if } (\delta S) \tau = \text{true } \tau \text{ then } \text{false } \tau \text{ else if } (\exists x \in (\triangledown \text{Rep-Set}_{\text{base}} (S \tau)). P(\lambda \cdot. x) \tau = \text{false } \tau) \text{ then } \text{false } \tau \text{ else if } (\exists x \in (\triangledown \text{Rep-Set}_{\text{base}} (S \tau)). P(\lambda \cdot. x) \tau = \text{invalid } \tau) \text{ then } \text{invalid } \tau \text{ else if } (\exists x \in (\triangledown \text{Rep-Set}_{\text{base}} (S \tau)). P(\lambda \cdot. x) \tau = \text{null } \tau) \text{ then } \text{null } \tau \text{ else true } \tau \text{ else } \bot)

**syntax**: -Forall :: [(\forall 'a::null) Set.id, (\forall 'a) Boolean] \Rightarrow (\forall 'a) Boolean \quad ((\text{-}\text{forAllSet}'\cdot\cdot'))

**translations**: X\text{-}\text{forAllSet}(x | P) \Rightarrow \text{CONST OclForall X}(\%x. P)

**Definition: Exists**

Like OclForall, OclExists is also not strict.
definition OclExists :: (%\alpha::null) Set, (%\alpha)val => (%\alpha)Boolean \Rightarrow \alpha Boolean
where OclExists S P = not(OclForall S (\lambda X. not (P X)))

syntax
-OclExist :: (%\alpha::null) Set.id, (%\alpha)val => (%\alpha)Boolean \Rightarrow \alpha Boolean ((\cdot) => existsS S (\cdot))
translations
X => existsS S (x \mid P) == CONST OclExists X (%x. P)

Definition: Iterate

definition OclIterate :: (%\alpha::null) Set, (%\alpha)val => (%\alpha)Boolean
where OclIterate S A F = \lambda S. \tau. if (\delta S) \tau = true \tau \land (\forall A) \tau = true \tau \land finiteRepSetBase (S \tau)\top
then (Finite-Set.fold (F) (A ((\lambda a \cdot a) :: finiteRepSetBase (S \tau)\top)) \tau)
else ⊥

syntax
-OclIterate :: (%\alpha::null) Set, idt, idt, '%\beta' => (%\alpha)val
(- - - => iterateS S (\cdot) \mid \cdot)
translations
X => iterateS S (a; x = A \mid P) == CONST OclIterate X A (%a. (%x. P))

Definition: Select

definition OclSelect :: (%\alpha::null) Set, (%\alpha)val => (%\alpha)Boolean \Rightarrow (%\alpha)Set
where OclSelect S P = \lambda S. \tau. if (\delta S) \tau = true \tau
then if (\exists x \in\top finiteRepSetBase (S \tau)\top. P(\lambda -. x) \tau = invalid \tau)
then invalid \tau
else Abs-SetBase \lambda x \in\top finiteRepSetBase (S \tau)\top. P (\lambda -. x) \tau \neq false \tau \top
else invalid \tau

syntax
-OclSelect :: (%\alpha::null) Set, id, id, '%\alpha' => (%\alpha)val
(- - - => selectS S (\cdot) \mid \cdot)
translations
X => selectS S (x \mid P) == CONST OclSelect X (%x. P)

Definition: Reject

definition OclReject :: (%\alpha::null) Set, (%\alpha)val => (%\alpha)Boolean \Rightarrow (%\alpha)Set
where OclReject S P = OclSelect S (not o P)
syntax
-OclReject :: (%\alpha::null) Set, id, id, '%\alpha' => (%\alpha)val
(- - - => rejectS S (\cdot) \mid \cdot)
translations
X => rejectS S (x \mid P) == CONST OclReject X (%x. P)

Definition: IncludesAll

definition OclIncludesAll :: (%\alpha::null) Set, (%\alpha) Set \Rightarrow \alpha Boolean
where OclIncludesAll x y = \lambda S. \tau. if (\delta x) \tau = true \tau \land (\delta y) \tau = true \tau
then \top finiteRepSetBase (y \tau)\top \subseteq \top finiteRepSetBase (x \tau)\top
else ⊥
notation OclIncludesAll (\cdot \multimap includesAllS S (\cdot))
Definition: ExcludesAll

\[
\text{OclExcludesAll}:: \left\{ (\lambda x, y. (\delta \tau) x = true \land (\delta \tau) y = true \rightarrow \text{Rep}_{\text{base}}(y \tau)^{\mathbb{N}} \cap \text{Rep}_{\text{base}}(x \tau)^{\mathbb{N}} = \{ \}, \downarrow \}
\right. \\
\text{not} \quad \text{notation} \quad \text{OclExcludesAll} \quad (-\rightarrow \text{excludesAll}_{\text{Set}}(\cdot))
\]

Definition: Union

\[
\text{OclUnion}:: \left\{ (\lambda x, y. (\delta \tau) x = true \land (\delta \tau) y = true \rightarrow \text{Abs}_{\text{base}}_{\text{Set}}(\text{Rep}_{\text{base}}(y \tau)^{\mathbb{N}} \cup \text{Rep}_{\text{base}}(x \tau)^{\mathbb{N}})
\right. \\
\text{not} \quad \text{notation} \quad \text{OclUnion} \quad (-\rightarrow \text{union}_{\text{Set}}(\cdot))
\]

Definition: Intersection

\[
\text{OclIntersection}:: \left\{ (\lambda x, y. (\delta \tau) x = true \land (\delta \tau) y = true \rightarrow \text{Abs}_{\text{base}}_{\text{Set}}(\text{Rep}_{\text{base}}(y \tau)^{\mathbb{N}} \cap \text{Rep}_{\text{base}}(x \tau)^{\mathbb{N}})
\right. \\
\text{not} \quad \text{notation} \quad \text{OclIntersection} \quad (-\rightarrow \text{intersection}_{\text{Set}}(\cdot))
\]

Definition (futur operators)

\[
\text{consts}
\]

\[
\text{OclCount}:: \left\{ (\lambda x, y. (\delta \tau) x = true \land (\delta \tau) y = true \rightarrow \text{Integer}
\right. \\
\text{not} \quad \text{notation} \quad \text{OclCount} \quad (-\rightarrow \text{count}_{\text{Set}}(\cdot))
\]

\[
\text{OclSum}:: \left\{ (\lambda x, y. (\delta \tau) x = true \land (\delta \tau) y = true \rightarrow \text{Integer}
\right. \\
\text{not} \quad \text{notation} \quad \text{OclSum} \quad (-\rightarrow \text{sum}_{\text{Set}}(\cdot))
\]

Logical Properties

OclIncluding

\[
\text{lemma} \quad \text{OclIncluding-defined-args-valid}:
(\tau |\delta(\text{X} \rightarrow \text{including}_{\text{Set}}(x))) = ((\tau |(\delta X)) \land (\tau |(\nu x)))
\]

\[
\text{lemma} \quad \text{OclIncluding-valid-args-valid}:
(\tau |\nu(\text{X} \rightarrow \text{including}_{\text{Set}}(x))) = ((\tau |(\delta X)) \land (\tau |(\nu x)))
\]

\[
\text{lemma} \quad \text{OclIncluding-defined-args-valid}[\text{simp,code-unfold}]\ :
\delta(\text{X} \rightarrow \text{including}_{\text{Set}}(x)) = ((\delta X) \land (\nu x))
\]

\[
\text{lemma} \quad \text{OclIncluding-valid-args-valid}[\text{simp,code-unfold}]\ :
\nu(\text{X} \rightarrow \text{including}_{\text{Set}}(x)) = ((\delta X) \land (\nu x))
\]

etc. etc.
Execution Laws with Invalid or Null or Infinite Set as Argument

OclIncluding

lemma OclIncluding-invalid-simp-code-unfold; (invalid \rightarrow including_{Set}(x)) = invalid

lemma OclIncluding-invalid-args-simp-code-unfold; (X \rightarrow including_{Set}(invalid)) = invalid

lemma OclIncluding-null-simp-code-unfold; (null \rightarrow including_{Set}(x)) = invalid

OclExcluding

lemma OclExcluding-invalid-simp-code-unfold; (invalid \rightarrow excluding_{Set}(x)) = invalid

lemma OclExcluding-invalid-args-simp-code-unfold; (X \rightarrow excluding_{Set}(invalid)) = invalid

lemma OclExcluding-null-simp-code-unfold; (null \rightarrow excluding_{Set}(x)) = invalid

OclIncludes

lemma OclIncludes-invalid-simp-code-unfold; (invalid \rightarrow includes_{Set}(x)) = invalid

lemma OclIncludes-invalid-args-simp-code-unfold; (X \rightarrow includes_{Set}(invalid)) = invalid

lemma OclIncludes-null-simp-code-unfold; (null \rightarrow includes_{Set}(x)) = invalid

OclExcludes

lemma OclExcludes-invalid-simp-code-unfold; (invalid \rightarrow excludes_{Set}(x)) = invalid

lemma OclExcludes-invalid-args-simp-code-unfold; (X \rightarrow excludes_{Set}(invalid)) = invalid

lemma OclExcludes-null-simp-code-unfold; (null \rightarrow excludes_{Set}(x)) = invalid

OclSize

lemma OclSize-invalid-simp-code-unfold; (invalid \rightarrow size_{Set}()) = invalid

lemma OclSize-null-simp-code-unfold; (null \rightarrow size_{Set}()) = invalid

OclIsEmpty

lemma OclIsEmpty-invalid-simp-code-unfold; (invalid \rightarrow isEmpty_{Set}()) = invalid

lemma OclIsEmpty-null-simp-code-unfold; (null \rightarrow isEmpty_{Set}()) = true

OclNotEmpty

lemma OclNotEmpty-invalid-simp-code-unfold; (invalid \rightarrow notEmpty_{Set}()) = invalid

lemma OclNotEmpty-null-simp-code-unfold; (null \rightarrow notEmpty_{Set}()) = false
OclANY

\[\text{lemma OclANY-invalid}[\text{simp,code-unfold}]: (\text{invalid} \rightarrow \text{any}_{\text{Set}}()) = \text{invalid}\]

\[\text{lemma OclANY-null}[\text{simp,code-unfold}]: (\text{null} \rightarrow \text{any}_{\text{Set}}()) = \text{null}\]

OclForall

\[\text{lemma OclForall-invalid}[\text{simp,code-unfold}]: \text{invalid} \rightarrow \text{forall}_{\text{Set}}(a \mid P a) = \text{invalid}\]

\[\text{lemma OclForall-null}[\text{simp,code-unfold}]: \text{null} \rightarrow \text{forall}_{\text{Set}}(a \mid P a) = \text{invalid}\]

OclExists

\[\text{lemma OclExists-invalid}[\text{simp,code-unfold}]: \text{invalid} \rightarrow \text{exists}_{\text{Set}}(a \mid P a) = \text{invalid}\]

\[\text{lemma OclExists-null}[\text{simp,code-unfold}]: \text{null} \rightarrow \text{exists}_{\text{Set}}(a \mid P a) = \text{invalid}\]

OclIterate

\[\text{lemma OclIterate-invalid}[\text{simp,code-unfold}]: \text{invalid} \rightarrow \text{iterate}_{\text{Set}}(a; x = A \mid P a x) = \text{invalid}\]

\[\text{lemma OclIterate-null}[\text{simp,code-unfold}]: \text{null} \rightarrow \text{iterate}_{\text{Set}}(a; x = A \mid P a x) = \text{invalid}\]

\[\text{lemma OclIterate-invalid-args}[\text{simp,code-unfold}]: S \rightarrow \text{iterate}_{\text{Set}}(a; x = \text{invalid} \mid P a x) = \text{invalid}\]

An open question is this ... \[\text{lemma S} \rightarrow \text{iterate}_{\text{Set}}(a; x = \text{null} \mid P a x) = \text{invalid}\]

\[\text{lemma OclIterate-infinite}:\]
\[\text{assumes non-finite: } \tau \models \text{not}(\delta(S \rightarrow \text{size}_{\text{Set}}()))\]
\[\text{shows (OclIterate S A F)} \tau = \text{invalid} \tau\]

OclSelect

\[\text{lemma OclSelect-invalid}[\text{simp,code-unfold}]: \text{invalid} \rightarrow \text{select}_{\text{Set}}(a \mid P a) = \text{invalid}\]

\[\text{lemma OclSelect-null}[\text{simp,code-unfold}]: \text{null} \rightarrow \text{select}_{\text{Set}}(a \mid P a) = \text{invalid}\]

OclReject

\[\text{lemma OclReject-invalid}[\text{simp,code-unfold}]: \text{invalid} \rightarrow \text{reject}_{\text{Set}}(a \mid P a) = \text{invalid}\]

\[\text{lemma OclReject-null}[\text{simp,code-unfold}]: \text{null} \rightarrow \text{reject}_{\text{Set}}(a \mid P a) = \text{invalid}\]

General Algebraic Execution Rules

Execution Rules on Including \[\text{lemma OclIncluding-finite-rep-set} :\]
assumes $X$-def : $\tau \models \delta X$
and $x$-val : $\tau \models v x$
shows finite $\forall \tau . (X \rightarrow \text{includesSet}(x) \tau) \Rightarrow \text{finite } \forall \tau . (X \rightarrow \text{Set} \tau)\n
\text{lemma OclIncluding-rep-set:}
\text{assumes } S$-def : $\tau \models \delta S$
\text{shows } $\forall \tau . (S \rightarrow \text{includesSet}(\lambda \cdot x. x \cdot x) \tau) \Rightarrow \text{insert}_{x. x} \forall \tau . (S \rightarrow \text{Set} \tau)\n
\text{lemma OclIncluding-notempty-rep-set:}
\text{assumes } X$-def : $\tau \models \delta X$
\text{and } $a$-val : $\tau \models v a$
\text{shows } $\forall \tau . (X \rightarrow \text{includesSet}(a) \tau) \neq \emptyset\n
\text{lemma OclIncluding-includes0:}
\text{assumes } \tau \models X \rightarrow \text{includesSet}(x)$
\text{shows } $X \rightarrow \text{includesSet}(x) \tau \Rightarrow X \tau\n
\text{lemma OclIncluding-commute0:}
\text{assumes } S$-def : $\tau \models \delta S$
\text{and } $i$-val : $\tau \models v i$
\text{and } $j$-val : $\tau \models v j$
\text{shows } $\tau \models ((S :: (A, a::null) \text{Set}) \rightarrow \text{includesSet}(i) \rightarrow \text{includesSet}(j) \Rightarrow (S \rightarrow \text{includesSet}(j) \rightarrow \text{includesSet}(i)))\n
\text{lemma OclIncluding-commute[simp, code-unfold]:}
((S :: (A, a::null) \text{Set}) \rightarrow \text{includesSet}(i) \rightarrow \text{includesSet}(j) = (S \rightarrow \text{includesSet}(j) \rightarrow \text{includesSet}(i)))\n
\text{Execution Rules on Excluding }
\text{lemma OclExcluding-finite-rep-set:}
\text{assumes } X$-def : $\tau \models \delta X$
\text{and } $x$-val : $\tau \models v x$
\text{shows finite } $\forall \tau . (X \rightarrow \text{excludingSet}(x) \tau) \Rightarrow \text{finite } \forall \tau . (X \rightarrow \text{Set} \tau)\n
\text{lemma OclExcluding-rep-set:}
\text{assumes } S$-def : $\tau \models \delta S$
\text{shows } $\forall \tau . (S \rightarrow \text{excludingSet}(\lambda \cdot x. x \cdot x) \tau) \Rightarrow \text{Set} \forall \tau . (S \rightarrow \text{Set} \tau)\n
\text{lemma OclExcluding-excludes0:}
\text{assumes } \tau \models X \rightarrow \text{excludingSet}(x)$
\text{shows } $X \rightarrow \text{excludingSet}(x) \tau \Rightarrow X \tau\n
\text{lemma OclExcluding-excludes:}
\text{assumes } \tau \models X \rightarrow \text{excludingSet}(x)$
\text{shows } $\tau \models X \rightarrow \text{excludingSet}(x) \tau \Rightarrow X \tau\n
\text{lemma OclExcluding-charn0[simp]:}
\text{assumes } val-x : \tau \models (v x)$
\text{shows } $\tau \models ((\text{Set}\{\}) \rightarrow \text{excludingSet}(x) \tau \Rightarrow \text{Set}\{\})$
lemma OclExcluding-commute0:
assumes S-def : \( \tau \vdash \delta S \)
and i-val : \( \tau \vdash \nu i \)
and j-val : \( \tau \vdash \nu j \)
shows \( \tau \vdash ((S :: (\forall A, 'a::null) Set) \to excluding_{Set}(i) \to excluding_{Set}(j)) \triangleq (S \to excluding_{Set}(j) \to excluding_{Set}(i)) \)

lemma OclExcluding-commute[simp.code-unfold]:
\((S :: (\forall A, 'a::null) Set) \to excluding_{Set}(i) \to excluding_{Set}(j) = (S \to excluding_{Set}(j) \to excluding_{Set}(i)) \)

lemma OclExcluding-commute1-exec[simp.code-unfold]:
\((\text{Set}\{\} \to excluding_{Set}(x)) = (\text{if } \nu x \text{ then } \text{Set}\{\} \text{ else invalid endif})\)

lemma OclExcluding-charn1:
assumes def-X : \( \tau \vdash (\delta X) \)
and val-x : \( \tau \vdash (\nu x) \)
and val-y : \( \tau \vdash (\nu y) \)
and neq : \( \tau \vdash \text{not}(x \triangleq y) \)
shows \( \tau \vdash ((X \to including_{Set}(x)) \to excluding_{Set}(y)) \triangleq ((X \to excluding_{Set}(y)) \to including_{Set}(x)) \)

lemma OclExcluding-charn2:
assumes def-X : \( \tau \vdash (\delta X) \)
and val-x : \( \tau \vdash (\nu x) \)
shows \( \tau \vdash ((X \to including_{Set}(x)) \to excluding_{Set}(y)) \triangleq (X \to excluding_{Set}(y)) \)

theorem OclExcluding-charn3: \((X \to including_{Set}(x)) \to excluding_{Set}(x)) = (X \to excluding_{Set}(x))\)

One would like a generic theorem of the form:

lemma OclExcluding_charn_exec:
"((X \to including_{Set}(x)) \to excluding_{Set}(y)) =
(if \( \delta X \) then (if \( x \triangleq y \) then \( X \to excluding_{Set}(y) \) else \( X \to excluding_{Set}(y) \to including_{Set}(x) \) )
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 \( \triangleq y \) = invalid)
and $\text{strict2}: (x \equiv \text{invalid}) = \text{invalid}$
and $\text{StrictRefEq-valid-args-valid}: \forall (x::(\text{A},'a::\text{null})\text{val})\ y\ \tau.
\quad (\tau \models \delta (x \equiv y)) = ((\tau \models (\upsilon\ x)) \land (\tau \models \upsilon\ y))$
and $\text{cp-StrictRefEq}: \forall (X::(\text{A},'a::\text{null})\text{val})\ Y\ \tau.\ (X \equiv Y)\ \tau = ((\lambda\ -.\ X\ \tau) \triangleq (\lambda\ -.\ Y\ \tau))\ \tau$
and $\text{StrictRefEq-vs-StrongEq}: \forall (x::(\text{A},'a::\text{null})\text{val})\ \tau.
\quad \tau \models \upsilon\ x \implies \tau \models \upsilon\ y \implies ((\tau \models (x \equiv y) \triangleq (x \not\equiv y)))$
shows $(X \rightarrow \text{including}_{\text{set}}(x::(\text{A},'a::\text{null})\text{val}) \rightarrow \text{excluding}_{\text{set}}(y)) =$
\begin{align*}
& (\text{if } \delta\ \text{ X then if } x \equiv y \\
& \quad \text{ then } X \rightarrow \text{excluding}_{\text{set}}(y)) \\
& \quad \text{ else } X \rightarrow \text{excluding}_{\text{set}}(y) \rightarrow \text{including}_{\text{set}}(x)) \\
& \text{ endif}
endif$

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

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

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

\text{Execution Rules on Includes} \ \text{lemma} \quad \text{OclIncludes-charn0}[\text{simp}]:$
\begin{align*}
\text{assumes} & \quad \text{val-x:}\ \tau \models (\upsilon\ x) \\
\text{shows} & \quad \tau \models \text{not(set{} \rightarrow \text{inclusion}_{\text{set}}(x))}
\end{align*}

\text{lemma} \quad \text{OclIncludes-charn0}[\text{simp, code-unfold}]:$
\begin{align*}
\text{Set{}} \rightarrow \text{inclusion}_{\text{set}}(x) &= (\text{if } \upsilon\ x \text{ then false else invalid endif})
\end{align*}

\text{lemma} \quad \text{OclIncludes-charn1}:
\begin{align*}
\text{assumes} & \quad \text{def-x:}\ \tau \models (\delta\ X) \\
\text{assumes} & \quad \text{val-x:}\ \tau \models (\upsilon\ x) \\
\text{shows} & \quad \tau \models (X \rightarrow \text{including}_{\text{set}}(x) \rightarrow \text{inclusion}_{\text{set}}(x))
\end{align*}

\text{lemma} \quad \text{OclIncludes-charn2}:
\begin{align*}
\text{assumes} & \quad \text{def-x:}\ \tau \models (\delta\ X) \\
\text{and} & \quad \text{val-x:}\ \tau \models (\upsilon\ x) \\
\text{and} & \quad \text{val-y:}\ \tau \models (\upsilon\ y) \\
\text{and} & \quad \text{neq:}\ \tau \models \text{not}(x \not\equiv y) \\
\text{shows} & \quad \tau \models (X \rightarrow \text{including}_{\text{set}}(x) \rightarrow \text{inclusion}_{\text{set}}(y)) \not\equiv (X \rightarrow \text{including}_{\text{set}}(y))
\end{align*}

Here is again a generic theorem similar as above.

\text{lemma} \quad \text{OclIncludes-execute-generic}:
\begin{align*}
\text{assumes} & \quad \text{strict1}: (\text{invalid} \equiv y) = \text{invalid} \\
\text{and} & \quad \text{strict2}: (x \equiv \text{invalid}) = \text{invalid} \\
\text{and} & \quad \text{cp-StrictRefEq}: \forall (X::(\text{A},'a::\text{null})\text{val})\ Y\ \tau.\ (X \equiv Y)\ \tau = ((\lambda\ -.\ X\ \tau) \triangleq (\lambda\ -.\ Y\ \tau))\ \tau
\end{align*}
and \( \text{StrictRefEq-vs-StrongEq} \): \( \forall (x::(\forall a::null)\text{val}) \ y \ \tau. \)
\[
\tau \models \mathcal{U} x \implies \tau \models \mathcal{U} y \implies (\tau \models ((x \triangleright y) \triangleq (x \triangleq y)))
\]
shows
\[
(X \rightarrow \text{includesSet}((x::(\forall a::null)\text{val}) \rightarrow \text{includesSet}(y)) =
\]
(if \( \delta \) \( X \) then if \( x \triangleq y \) then true else \( X \rightarrow \text{includesSet}(y) \) endif else invalid endif)

**schematic-lemma** \( \text{OclIncludes-execute}_{\text{Integer}}[\text{simp},\text{code-unfold}] \): \?X

**schematic-lemma** \( \text{OclIncludes-execute}_{\text{Boolean}}[\text{simp},\text{code-unfold}] \): \?X

**schematic-lemma** \( \text{OclIncludes-execute}_{\text{Set}}[\text{simp},\text{code-unfold}] \): \?X

**lemma** \( \text{OclIncludes-including-generic} \):
assumes \( \text{OclIncludes-execute-generic} [\text{simp}] : \forall X. y. \\
(X \rightarrow \text{includesSet}((x::(\forall a::null)\text{val}) \rightarrow \text{includesSet}(y)) =
\]
(if \( \delta \) \( X \) then if \( x \triangleq y \) then true else \( X \rightarrow \text{includesSet}(y) \) endif else invalid endif)
and \( \text{StrictRefEq-strict\";} \) : \( \forall X. y. \delta = (\tau \rightarrow \text{includesSet}(y)) = (\tau \rightarrow \text{includesSet}(y))
and \( a\text{-val} : \tau \models \mathcal{U} a \\
and \( x\text{-val} : \tau \models \mathcal{U} x \\
and \( S\text{-incl} : \tau \models (S \rightarrow \text{includesSet}((x::(\forall a::null)\text{val}) )
\]
shows \( \tau \models \delta \rightarrow \text{includesSet}((a::(\forall a::null)\text{val}) \rightarrow \text{includesSet}(x))
\]

**lemmas** \( \text{OclIncludes-including}_{\text{Integer}} = \text{OclIncludes-including-generic} \):

**Execution Rules on Excludes**

**lemma** \( \text{OclExcludes-charn1} \):
assumes \( \text{def-X} : \tau \models (\delta \ X) \\
assumes \( \text{val-x} : \tau \models (\mathcal{U} x) \\
shows \tau \models (X \rightarrow \text{excludesSet}(x) \rightarrow \text{excludesSet}(x))
\]

**Execution Rules on Size**

**lemma** \( [\text{simp},\text{code-unfold}] : \text{Set}() \rightarrow \text{sizeSet}() = 0 
\]

**lemma** \( \text{OclSize-including-exec} [\text{simp},\text{code-unfold}] : \\
((X \rightarrow \text{includesSet}(x)) \rightarrow \text{sizeSet}()) = (if \delta \ X \ and \ \mathcal{U} x \ then \\
X \rightarrow \text{sizeSet}() + \text{int} \ if \ X \rightarrow \text{includesSet}(x) \ then \ 0 \ else \ 1 \ endif \\
else \\
invalid \\
endif)
\]

**Execution Rules on IsEmpty**

**lemma** \( [\text{simp},\text{code-unfold}] : \text{Set}() \rightarrow \text{isEmptySet}() = \text{true} 
\]

**lemma** \( \text{OclIsEmpty-including} [\text{simp}] : \\
assumes \( \text{X-def} : \tau \models \delta \ X \\
and \( \text{X-finite} : \text{finite}(X. \mathcal{T}) \\
and \( \text{a-val} : \tau \models \mathcal{U} a \\
shows X \rightarrow \text{includesSet}(a) \rightarrow \text{isEmptySet}() \ \tau = \text{false} \ \tau
\]
Execution Rules on NotEmpty

**Lemma** (simp, code-unfold): \( \text{Set} \{\} \rightarrow \text{notEmpty}_{\text{Set}}() = \text{false} \)

**Lemma** OclNotEmpty-including (simp, code-unfold):

- assumes \( X-\text{def} : \tau |\leq \delta X \)
- and \( X-\text{finite} : \text{finite} "\text{Rep-Set}_{\text{base}}(X \; \tau)" \)
- and \( a-\text{val} : \tau |\leq \upsilon a \)

**Shows** \( X \rightarrow \text{including}_{\text{Set}}(a) \rightarrow \text{notEmpty}_{\text{Set}}() \; \tau = \text{true} \)

Execution Rules on Any

**Lemma** (simp, code-unfold): \( \text{Set} \{\} \rightarrow \text{any}_{\text{Set}}() = \text{null} \)

**Lemma** OclANY-singleton-exec (simp, code-unfold):

\( \text{Set} \{\} \rightarrow \text{including}_{\text{Set}}(a) \rightarrow \text{any}_{\text{Set}}() = a \)

Execution Rules on Forall

**Lemma** OclForall-mtSet-exec (simp, code-unfold):

\( \text{Set} \{\} \rightarrow \forall_{\text{Set}}(z \; | \; P(z)) = \text{true} \)

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:

\( \text{Integer}_{\text{null}} \rightarrow \forall_{\text{Set}}(x | \text{Integer}_{\text{null}} \rightarrow \forall_{\text{Set}}(y | x + \text{int} \; y = y + \text{int} \; x)) \)

or even:

\( \text{Integer} \rightarrow \forall_{\text{Set}}(x | \text{Integer} \rightarrow \forall_{\text{Set}}(y | x + \text{int} \; y = y + \text{int} \; x)) \)

are valid OCL statements in any context \( \tau \).

**Theorem** OclForall-including-exec (simp, code-unfold):

- assumes \( cp0 : cp \; P \)
- shows \( (S \rightarrow \text{including}_{\text{Set}}(x)) \rightarrow \forall_{\text{Set}}(z \; | \; P(z)) = (\text{if } \delta \; S \text{ and } \upsilon \; x \; \text{then } P \; x \text{ and } (S \rightarrow \forall_{\text{Set}}(z \; | \; P(z))) \; \text{else } \text{invalid} \) \)

Execution Rules on Exists

**Lemma** OclExists-mtSet-exec (simp, code-unfold):

\( (\text{Set} \{\}) \rightarrow \exists_{\text{Set}}(z \; | \; P(z)) = \text{false} \)

**Lemma** OclExists-including-exec (simp, code-unfold):

- assumes \( cp : cp \; P \)
- shows \( (S \rightarrow \text{including}_{\text{Set}}(x)) \rightarrow \exists_{\text{Set}}(z \; | \; P(z)) = (\text{if } \delta \; S \text{ and } \upsilon \; x \; \text{then } P \; x \text{ or } (S \rightarrow \exists_{\text{Set}}(z \; | \; P(z))) \; \text{else } \text{invalid} \) \)

Execution Rules on Iterate

**Lemma** OclIterate-empty (simp, code-unfold):

\( (\text{Set} \{\}) \rightarrow \text{iterate}_{\text{Set}}(a \; | \; x = A \; | \; P \; a \; x) = A \)
In particular, this does hold for A = null.

**Lemma** OclIterate-including:

**Assumes** S-finite: \( \tau \models \delta (S \to size\ Set) \)

and F-valid-arg: \((v A) \tau = (v (F a A)) \tau \)

and F-commute: comp-fan-commute F

and F-cp: \( \lambda x y \tau. F x y \tau = F (\lambda \_ x \tau) y \tau \)

**Shows** \((S \to including\ Set(a)) \to iterate\ Set(a; x = A \mid F a x)) \tau =

\((S \to excluding\ Set(a)) \to iterate\ Set(a; x = F a A \mid F a x)) \tau \)

**Execution Rules on Select**

**Lemma** OclSelect-mtSet-exec [simp, code-unfold]: OclSelect mtSet P = mtSet

**Definition** OclSelect-body :: - \to - \to (\forall, \_ a \text{ option option}) Set

\[ \equiv (\lambda P x \text{ acc. if } P x. \text{ false then acc else acc} \to including\ Set(x) \text{ endif}) \]

**Theorem** OclSelect-including-exec [simp, code-unfold]:

**Assumes** P-cp : cp P

**Shows** OclSelect (X \to including\ Set(y)) P = OclSelect-body P y (OclSelect (X \to excluding\ Set(y)) P)

(is = ?select)

**Execution Rules on Reject**

**Lemma** OclReject-mtSet-exec [simp, code-unfold]: OclReject mtSet P = mtSet

**Assumes** P-cp : cp P

**Shows** OclReject (X \to including\ Set(y)) P = OclSelect-body (not o P) y (OclReject (X \to excluding\ Set(y)) P)

**Execution Rules Combining Previous Operators**

**OclIncluding**

**Lemma** OclIncluding-idem0:

**Assumes** \( \tau \models \delta S \)

and \( \tau \models \upsilon i \)

**Shows** \( \models (S \to including\ Set(i)) \to including\ Set(i) \equiv (S \to including\ Set(i)) \)

**Theorem** OclIncluding-idem [simp, code-unfold]: \((S :: (\forall, \_ : \text{null})\text{Set}) \to including\ Set(i) \to including\ Set(i) = (S \to including\ Set(i)) \)

**OclExcluding**

**Lemma** OclExcluding-idem0:

**Assumes** \( \tau \models \delta S \)

and \( \tau \models \upsilon i \)

**Shows** \( \models (S \to excluding\ Set(i)) \to excluding\ Set(i) \equiv (S \to excluding\ Set(i)) \)

**Theorem** OclExcluding-idem [simp, code-unfold]: \((S \to excluding\ Set(i)) \to excluding\ Set(i) = (S \to excluding\ Set(i)) \)

**OclIncludes**

**Lemma** OclIncludes-any [simp, code-unfold]:
\( X \rightarrow \text{includes}_{\text{Set}}(X \rightarrow \text{any}_{\text{Set}}()) = (\text{if } \delta X \text{ then} \)  
\[ \text{if } \delta (X \rightarrow \text{size}_{\text{Set}}()) \text{ then not}(X \rightarrow \text{isEmpty}_{\text{Set}}()) \]  
\[ \text{else } X \rightarrow \text{includes}_{\text{Set}}(\text{null}) \text{ endif} \]  
\[ \text{else invalid endif} \)

OclSize

**Lemma** [simp.code-unfold]: \( \delta (\text{Set}()) \rightarrow \text{size}_{\text{Set}}()) = \text{true} \)

**Lemma** [simp.code-unfold]: \( \delta ((X \rightarrow \text{including}_{\text{Set}}(x)) \rightarrow \text{size}_{\text{Set}}()) = (\delta(X \rightarrow \text{size}_{\text{Set}}()) \text{ and } \upsilon(x)) \)

**Lemma** [simp.code-unfold]: \( \delta ((X \rightarrow \text{excluding}_{\text{Set}}(x)) \rightarrow \text{size}_{\text{Set}}()) = (\delta(X \rightarrow \text{size}_{\text{Set}}()) \text{ and } \upsilon(x)) \)

**Lemma** [simp]:
assumes X-finite: \( \land \tau. \text{finite } \uparrow\text{Rep-Set}_{\text{base}} (X \tau)^\top \)
shows \( \delta ((X \rightarrow \text{including}_{\text{Set}}(x)) \rightarrow \text{size}_{\text{Set}}()) = (\delta(X) \text{ and } \upsilon(x)) \)

OclForall

**Lemma** OclForall-rep-set-true:
assumes \( \tau \models \delta X \)
shows \( (\text{OclForall } X P \tau = \text{false } \tau) = (\exists x \in \uparrow\text{Rep-Set}_{\text{base}} (X \tau)^\top. P (\lambda \tau.x) \tau = \text{false } \tau) \)

**Lemma** OclForall-rep-set-false:
assumes \( \tau \models \delta X \)
shows \( (\tau \models \text{OclForall } X P) = (\forall x \in \uparrow\text{Rep-Set}_{\text{base}} (X \tau)^\top. \tau \models P (\lambda \tau.x)) \)

**Lemma** OclForall-includes :
assumes x-def : \( \tau \models \delta x \)  
and y-def : \( \tau \models \delta y \)
shows \( (\tau \models \text{OclForall } x (\text{OclIncludes } y)) = (\uparrow\text{Rep-Set}_{\text{base}} (x \tau)^\top \subseteq \uparrow\text{Rep-Set}_{\text{base}} (y \tau)^\top) \)

**Lemma** OclForall-not-includes :
assumes x-def : \( \tau \models \delta x \)  
and y-def : \( \tau \models \delta y \)
shows \( (\tau \models \text{OclForall } x (\text{OclIncludes } y) \tau = \text{false } \tau) = (\neg(\uparrow\text{Rep-Set}_{\text{base}} (x \tau)^\top \subseteq \uparrow\text{Rep-Set}_{\text{base}} (y \tau)^\top)) \)

**Lemma** OclForall-iterate:
assumes S-finite: finite \( \uparrow\text{Rep-Set}_{\text{base}} (S \tau)^\top \)
shows \( S \rightarrow \text{forAll}_{\text{Set}}(x \mid P x) \tau = (S \rightarrow \text{iterate}_{\text{Set}}(\tau; \text{acc } = \text{true } \mid \text{acc } \text{ and } P x)) \tau \)

**Lemma** OclForall-cong:
assumes \( \forall x. x \in \uparrow\text{Rep-Set}_{\text{base}} (X \tau)^\top \models \tau \models P (\lambda \tau.x) \models \tau = Q (\lambda \tau.x) \)  
assumes \( P. \tau \models \text{OclForall } X P \)  
shows \( \tau \models \text{OclForall } X Q \)

**Lemma** OclForall-cong:
assumes \( \forall x. x \in \uparrow\text{Rep-Set}_{\text{base}} (X \tau)^\top \models \tau \models P (\lambda \tau.x) \models \tau = Q (\lambda \tau.x) \models \tau = R (\lambda \tau.x) \)  
assumes \( P. \tau \models \text{OclForall } X P \)  
assumes \( Q. \tau \models \text{OclForall } X Q \)  
shows \( \tau \models \text{OclForall } X R \)
Strict Equality

**lemma** StrictRefEq-set-defined :

**assumes** x-def : τ \models δ x

**assumes** y-def : τ \models δ y

**shows** ((x:('A,'a::null)Set) = y) τ = (x->forallSet(z) y->includesSet(z)) and (y->forallSet(z) x->includesSet(z))) τ

**lemma** StrictRefEq-exec[simp,code-unfold] :

((x:('A,'a::null)Set) = y) = 
(if δ x then (if δ y 
then ((x->forallSet(z) y->includesSet(z)) and (y->forallSet(z) x->includesSet(z)))) 
else if υ y 
then false (+ x'->includes = null) 
else invalid 
endif 
endif) 
else if υ x (* null = ??? *) 
then if υ y then not(δ y) else invalid endif 
else invalid 
endif 
endif

**lemma** OclIncluding-cong' : 

**shows** τ = δ s \implies τ = δ t \implies τ = υ x \implies 
τ = ((s::('A,'a::null)Set) = t) \implies τ = (s->includesSet(x) \equiv (t->includesSet(x)))

**lemma** OclIncluding-cong : \forall(s::('A,'a::null)Set) t x y. τ = δ t \implies τ = υ y \implies 
τ = s \equiv t \implies x = y \implies τ = s->includesSet(x) \equiv (t->includesSet(y))

**lemma** const-StrictRefEqSet-empty : const X \implies const (X = Set{ })

**lemma** const-StrictRefEqSet-including : 
const a \implies const S \implies const (X = S->includesSet(a))

**Test Statements**

Assert (τ = (Set{λ-..} \equiv Set{λ-..})))

Assert (τ = (Set{λ-..} \equiv Set{λ-..})))

**A.5.9. Collection Type Sequence: Operations**

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

**lemma** Sequence-inv-lemma : τ \models (δ X) \implies \forall x\in set^T Rep-Sequencebase (X τ)^T. x \neq bot

74
**Constants: mtSequence**

**Definition**

\[\text{mtSequence} ::= (\forall \alpha. \alpha::\texttt{null}) \texttt{Sequence} \ (\texttt{Sequence}\{\})\]

**Where**

\[\text{Sequence}\{\} \equiv (\lambda \tau. \texttt{Abs-Sequence}_{\texttt{base}} \triangleleft \triangleleft \lambda \lambda::\texttt{list}_{\texttt{base}})\]

**Declare**

\[\text{mtSequence-def}[\text{code-unfold}]\]

**Lemma**

\[\text{mtSequence-defined}[\text{simp}, \text{code-unfold}]: δ(\texttt{Sequence}\{\}) = \texttt{true}\]

\[\text{mtSequence-valid}[\text{simp}, \text{code-unfold}]: υ(\texttt{Sequence}\{\}) = \texttt{true}\]

**Lemma**

\[\text{mtSequence-rep-set}: [\![\texttt{Rep-Sequence}_{\texttt{base}} (\texttt{Sequence}\{} \tau \texttt{)}\!]\texttt{\tau}^\texttt{T} = []\]

**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:

**Defn**

\[\text{StrictRefEq}_{\texttt{Sequence}} [\text{code-unfold}]:\]

\[(x::(\forall \alpha. \alpha::\texttt{null}) \texttt{Sequence}) \triangleq y) \equiv (\lambda \tau. \texttt{if} (δ x) \tau = \texttt{true} \land (υ y) \tau = \texttt{true} \tau \texttt{then} (x = y) \tau \texttt{else invalid} \tau)\]

**Property proof in terms of profile-bin3**

**Interpretation**

\[\text{StrictRefEq}_{\texttt{Sequence}} : \text{profile-bin3} \lambda \ x \ y. (x::(\forall \alpha. \alpha::\texttt{null}) \texttt{Sequence}) \triangleq y\]

**Definition: including**

**Definition**

\[\text{OclIncluding} :: [((\forall \alpha. \alpha::\texttt{null}) \texttt{Sequence}, (\forall \alpha) \texttt{val}) \Rightarrow ((\forall \alpha) \texttt{Sequence})] \]

**Where**

\[\text{OclIncluding} \ x \ y = (\lambda \tau. \texttt{if} (δ x) \tau = \texttt{true} \land (υ y) \tau = \texttt{true} \tau \texttt{then Abs-Sequence}_{\texttt{base}_{\texttt{base}}} [\text{Rep-Sequence}_{\texttt{base}} (x \tau)^Τ @ [y \tau]_{\texttt{base}}] \texttt{else invalid} \tau)\]

**Notation**

\[\text{OclIncluding} (\rightarrow\rightarrow\text{including}_{\texttt{Seq}}'(\cdot'))\]

**Interpretation**

\[\text{OclIncluding} : \text{profile-bin2} \lambda \ x \ y. \texttt{Abs-Sequence}_{\texttt{base}_{\texttt{base}}} [\text{Rep-Sequence}_{\texttt{base}} x^Τ @ [y]_{\texttt{base}}]\]

**Syntax**

\[\text{-OclFinsequence} ::= \text{args} => (\forall \alpha. \alpha::\texttt{null}) \texttt{Sequence} \ (\texttt{Sequence}\{\})\]

**Translations**

\[\text{Sequence}\{x, xs\} \equiv \text{CONST OclIncluding} \ (\texttt{Sequence}\{xs\}) \ x\]

\[\text{Sequence}\{x\} \equiv \text{CONST OclIncluding} \ (\texttt{Sequence}\{\}) \ x\]

**Definition: excluding**

**Definition**

\[\text{OclExcluding} :: [((\forall \alpha. \alpha::\texttt{null}) \texttt{Sequence}, (\forall \alpha) \texttt{val}) \Rightarrow ((\forall \alpha) \texttt{Sequence})] \]

**Where**

\[\text{OclExcluding} \ x \ y = (\lambda \tau. \texttt{if} (δ x) \tau = \texttt{true} \land (υ y) \tau = \texttt{true} \tau \texttt{then Abs-Sequence}_{\texttt{base}_{\texttt{base}}} [\texttt{filter} (\lambda x. x = y) \tau]^Τ \texttt{else invalid} \tau)\]

75
\[ \text{Rep-Sequence}_{\text{base}} \left( x \ \tau \right)^{\uparrow} \]

\text{else invalid } \tau \}

\textbf{notation} \quad \textit{OclExcluding} \quad (-\rightarrow \text{excluding}_{\text{seq}} \left( \cdot \right))

\textbf{Definition: append}\n
\textit{identical to including}\n
\textbf{definition} \quad \textit{OclAppend} \quad :: \left( \forall \alpha, \alpha::\text{null} \right) \\text{Sequence}, \left( \forall \alpha, \alpha \right) \\text{val} \Rightarrow \left( \forall \alpha, \alpha \right) \\text{Sequence} \n
\textbf{where} \quad \textit{OclAppend} = \textit{OclIncluding}

\textbf{Definition: union}\n
\textbf{definition} \quad \textit{OclUnion} \quad :: \left( \forall \alpha, \alpha::\text{null} \right) \\text{Sequence}, \left( \forall \alpha, \alpha \right) \\text{Sequence} \Rightarrow \left( \forall \alpha, \alpha \right) \\text{Sequence} \n
\textbf{where} \quad \textit{OclUnion} \ x \ y = (\lambda \ \tau. \ \text{if } (\delta \ x) \ \tau = \text{true } \tau \land (\delta \ y) \ \tau = \text{true } \tau \text{ then } \text{Abs-Sequence}_{\text{base}} \left( \text{Rep-Sequence}_{\text{base}} \left( x \ \tau \right)^{\uparrow} \right), \text{else invalid } \tau) \]

\textbf{notation} \quad \textit{OclUnion} \quad (-\rightarrow \text{union}_{\text{seq}} \left( \cdot \right))

\textbf{Definition: prepend}\n
\textbf{definition} \quad \textit{OclPrepend} \quad :: \left( \forall \alpha, \alpha::\text{null} \right) \\text{Sequence}, \left( \forall \alpha, \alpha \right) \\text{val} \Rightarrow \left( \forall \alpha, \alpha \right) \\text{Sequence} \n
\textbf{where} \quad \textit{OclPrepend} \ x \ y = (\lambda \ \tau. \ \text{if } (\delta \ x) \ \tau = \text{true } \tau \land (\delta \ y) \ \tau = \text{true } \tau \text{ then } \text{Abs-Sequence}_{\text{base}} \left( \text{Rep-Sequence}_{\text{base}} \left( x \ \tau \right)^{\uparrow} \right), \text{else invalid } \tau) \]

\textbf{notation} \quad \textit{OclPrepend} \quad (-\rightarrow \text{prepend}_{\text{seq}} \left( \cdot \right))

\textbf{interpretation} \quad \textit{OclPrepend::profile-bin2} \ \textit{OclPrepend} \ \lambda \ x \ y. \ \text{Abs-Sequence}_{\text{base}} \left( \text{Rep-Sequence}_{\text{base}} \left( x \ \tau \right)^{\uparrow} \right)

\textbf{Definition: subSequence}\n
\textbf{Definition: at}\n
\textbf{definition} \quad \textit{OclAt} \quad :: \left( \forall \alpha, \alpha::\text{null} \right) \\text{Sequence}, \left( \forall \alpha \right) \\text{Integer} \Rightarrow \left( \forall \alpha, \alpha \right) \\text{val} \n
\textbf{where} \quad \textit{OclAt} \ x \ y = (\lambda \ \tau. \ \text{if } (\delta \ x) \ \tau = \text{true } \tau \land (\delta \ y) \ \tau = \text{true } \tau \text{ then } \text{Abs-Sequence}_{\text{base}} \left( \text{Rep-Sequence}_{\text{base}} \left( x \ \tau \right)^{\uparrow} \right) ! \left( \text{nat } y \ \tau + 1 \right), \text{else invalid } \tau) \]

\textbf{notation} \quad \textit{OclAt} \quad (-\rightarrow \text{at}_{\text{seq}} \left( \cdot \right))

\textbf{Definition: first}\n
\textbf{definition} \quad \textit{OclFirst} \quad :: \left( \forall \alpha, \alpha::\text{null} \right) \\text{Sequence} \Rightarrow \left( \forall \alpha, \alpha \right) \\text{val} \n
\textbf{where} \quad \textit{OclFirst} \ x = (\lambda \ \tau. \ \text{if } (\delta \ x) \ \tau = \text{true } \tau \text{ then } \text{hd } \text{Rep-Sequence}_{\text{base}} \left( x \ \tau \right)^{\uparrow}, \text{else invalid } \tau) \]

\textbf{notation} \quad \textit{OclFirst} \quad (-\rightarrow \text{first}_{\text{seq}} \left( \cdot \right))

\textbf{Definition: last}\n
\textbf{definition} \quad \textit{OclLast} \quad :: \left( \forall \alpha, \alpha::\text{null} \right) \\text{Sequence} \Rightarrow \left( \forall \alpha, \alpha \right) \\text{val}
where \( \text{OclLast } x = (\lambda \tau. \text{ if } (\delta x) \tau = \text{true } \tau \text{ then } \text{last } ''\text{Rep-Sequence}_{\text{base}} (x \tau)'' \text{ else invalid } \tau) \)

**Notation**: \( \text{OclLast } (\rightarrow\rightarrow\text{last}_{\text{Seq}} (\cdot)) \)

**Definition**: \textit{asSet}

\textit{Instantiation} \( \text{Sequence}_{\text{base}} :: (\text{equal})\text{equal} \)

\textit{begin}

\textit{definition} \( \text{HOL.equal } k \ l \longleftrightarrow (k::(\alpha::\text{equal})\text{Sequence}_{\text{base}}) = l \)

\textit{instance}

\textit{end}

\textit{Lemma} \textit{equal-Sequence}_{\text{base}}-\text{code} [\text{code}]:

\( \text{HOL.equal } k \ l :: (\alpha::\{\text{equal, null}\})\text{Sequence}_{\text{base}}) \longleftrightarrow \text{Rep-Sequence}_{\text{base}} k = \text{Rep-Sequence}_{\text{base}} l \)

**Test Statements**

\text{Assert } (\tau \models (\text{Sequence}\{\} \equiv \text{Sequence}\{\}))

\text{Assert } (\tau \models (\text{Sequence}\{1, \text{invalid}, 2\} \triangleq \text{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 \textit{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 \( \text{Set}(\text{Set}(\text{Sequences}(\text{Pairs}(X,Y)))))).

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

lemma syntax-test: $\text{Set}\{2,1\} = \langle \text{Set}\{\}\rangle : > \text{includes}_{\text{Set}}(1) : > \text{includes}_{\text{Set}}(2)\rangle$

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

lemma semantic-test2:
assumes $H: \langle \text{Set}\{2\} = null \rangle = (\text{false};(\forall)\text{Boolean})$
shows $(\tau;(\forall)\alpha t) : = \langle \text{Set}\{2\};\text{null} \rangle : > \text{includes}_{\text{Set}}(\text{null})\rangle$

lemma short-cut$\delta[simp,\text{code-unfold}]; (8 \neq 6) = \text{false}$
lemma short-cut$\delta[simp,\text{code-unfold}]; (2 \neq 1) = \text{false}$
lemma short-cut$\delta[simp,\text{code-unfold}]; (1 \neq 2) = \text{false}$

Elementary computations on Sets.

declare OclSelect-body-def [simp]

assert $\tau \vdash \langle \text{null}::(\forall)\alpha \alpha::\text{null} \rangle \text{Set}\rangle$
assert $\tau \vdash \langle \text{null}::(\forall)\alpha \alpha::\text{null} \rangle \text{Set}\rangle$
assert $\tau \vdash \langle \text{null}::(\forall)\alpha \alpha::\text{null} \rangle \text{Set}\rangle$
assert $\tau \vdash \langle \text{null}::(\forall)\alpha \alpha::\text{null} \rangle \text{Set}\rangle$
assert $\tau \vdash \langle \text{null}::(\forall)\alpha \alpha::\text{null} \rangle \text{Set}\rangle$
assert $\tau \vdash \langle \text{null}::(\forall)\alpha \alpha::\text{null} \rangle \text{Set}\rangle$
assert $\tau \vdash \langle \text{null}::(\forall)\alpha \alpha::\text{null} \rangle \text{Set}\rangle$
assert $\tau \vdash \langle \text{null}::(\forall)\alpha \alpha::\text{null} \rangle \text{Set}\rangle$
assert $\tau \vdash \langle \text{null}::(\forall)\alpha \alpha::\text{null} \rangle \text{Set}\rangle$
assert $\tau \vdash \langle \text{null}::(\forall)\alpha \alpha::\text{null} \rangle \text{Set}\rangle$
assert $\tau \vdash \langle \text{null}::(\forall)\alpha \alpha::\text{null} \rangle \text{Set}\rangle$
assert $\tau \vdash \langle \text{null}::(\forall)\alpha \alpha::\text{null} \rangle \text{Set}\rangle$
A.6. Formalization III: UML/OCL constructs: State Operations and Objects

no-notation None (⊥)

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 StrictRefEqObject :: (\x, \a::\{object, null\})val ⇒ (\a, \a)val ⇒ (\x)\Boolean
where  StrictRefEqObject x y
  ≡ λ τ. if (\nu x) τ = true τ ∧ (\nu y) τ = true τ
     then if x τ = null ∨ y τ = null
       then (\t. x τ = null) ∨ (\t. y τ = null)
     else invalid τ
     else (oid-of (\t. x τ)) = (oid-of (\t. y τ)) ⊢_

Strictness and context passing  lemma StrictRefEqObject-strict1[simp, code-unfold] :
  (StrictRefEqObject x invalid) = invalid

lemma StrictRefEqObject-strict2[simp, code-unfold] :
  (StrictRefEqObject invalid x) = invalid

lemma cp-StrictRefEqObject:
  (StrictRefEqObject x y τ) = (StrictRefEqObject (λ•. x τ) (λ•. y τ)) τ

Logic and Algebraic Layer on Object

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

lemma StrictRefEqObject-defargs:
  τ ⇒ (StrictRefEqObject x (y::(\x, \a::\{null\})val)) ⇒ (τ = (\nu x)) ∧ (τ = (\nu y))
lemma defined-StrictRefEq-1:
assumes val-x : τ |= υ x
assumes val-x : τ |= υ y
shows τ |= δ (StrictRefEqObject x y)

lemma StrictRefEqObject-def-homo:
δ(StrictRefEqObject x y:(\forall \alpha::(null,object)val)) = ((υ x) and (υ y))

Symmetry lemma StrictRefEqObject-sym:
assumes x-val : τ |= υ x
shows τ |= StrictRefEqObject x x

Behavior vs StrongEq It remains to clarify the role of the state invariant \(\text{inv}_\sigma(\sigma)\) mentioned above that states the condition that there is a “one-to-one” correspondence between object representations and oid’s: \(\forall \text{oid} \in \text{dom} \sigma, \text{oid} = \text{OidOf}^\sigma(\text{oid})\). 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 :: (\forall \alpha::object)st \Rightarrow bool
where WFF σ = ((\forall x \in \text{ran}(\text{heap}(\text{fst} \sigma)), \text{heap}(\text{fst} \sigma) (\text{oid-of} x) = x) \land
(\forall x \in \text{ran}(\text{heap}(\text{snd} \sigma)), \text{heap}(\text{snd} \sigma) (\text{oid-of} x) = 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 \(\models\) is defined by generic referential equality.

theorem StrictRefEqObject-vs-StrongEq:
assumes WFF: WFF σ
and valid-x: \(\sigma\models (\forall \alpha::\forall x::(null,object)val)\)
and valid-y: \(\sigma\models (\forall \alpha::\forall x::(null,object)val)\)
and x-present-pre: \(x \sigma\in \text{ran}(\text{heap}(\text{fst} \sigma))\)
and y-present-pre: \(y \sigma\in \text{ran}(\text{heap}(\text{fst} \sigma))\)
and x-present-post: \(x \sigma\in \text{ran}(\text{heap}(\text{snd} \sigma))\)
and y-present-post: \(y \sigma\in \text{ran}(\text{heap}(\text{snd} \sigma))\)
shows \(\sigma\models (\text{StrictRefEqObject } x y)\) = \((\sigma\models (x \models y))\)

theorem StrictRefEqObject-vs-StrongEq2:
assumes WFF: WFF σ
and valid-x: \(\sigma\models (\forall x::(\forall x::(null,object)val))\)
and valid-\(\nu\): \(\tau \models (\nu \ y)\)

and oid-preserve: \(\forall x. x \in \text{ran}(\text{heap}(\text{fst} \ \tau)) \lor x \in \text{ran}(\text{heap}(\text{snd} \ \tau)) \implies H \ x \neq \bot \implies \text{oid-of}(H \ x) = \text{oid-of} x\)

and xy-together: \(x \ \tau \in H \cdot \text{ran}(\text{heap}(\text{fst} \ \tau)) \land y \ \tau \in H \cdot \text{ran}(\text{heap}(\text{fst} \ \tau)) \lor x \ \tau \in H \cdot \text{ran}(\text{heap}(\text{snd} \ \tau)) \land y \ \tau \in H \cdot \text{ran}(\text{heap}(\text{snd} \ \tau))\)

shows \((\tau \models (\text{StrictRefEq}_\text{Object} \ x \ y)) = (\tau \models (x \triangleq y))\)

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 \(\tau_0 :: \{\forall \}_{\text{st}}\)

where \(\tau_0 \equiv ([\text{heap}=\text{Map}.\text{empty}, \text{assocs}=\text{Map}.\text{empty}]), ([\text{heap}=\text{Map}.\text{empty}, \text{assocs}=\text{Map}.\text{empty}])\)

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 :: ((\forall::\text{object}) \text{st} = \forall::\text{state}) \implies (\forall::\text{object} \rightarrow \forall) \Rightarrow (\forall, \forall\ \text{option}\ \text{option}) \text{Set}

where OclAllInstances-generic fst-snd H = (\lambda \tau. \text{Abs-Set}_{\text{base}} \Downarrow \text{Some} \ ((H \cdot \text{ran}(\text{heap}(\text{fst-snd} \ \tau)))) - \{\text{None}\} \Downarrow)

lemma OclAllInstances-generic-defined: \(\tau \models \delta(\text{OclAllInstances-generic pre-post} \ H)\)

lemma OclAllInstances-generic-init-empty:

assumes [simp]: \(\forall x. \text{pre-post} \ (x, x) = x\)

shows \(\tau_0 \models \text{OclAllInstances-generic pre-post} \ H \triangleq \text{Set}\{\}\)

lemma represented-generic-objectsnonnull:

assumes A: \(\tau \models (\text{OclAllInstances-generic pre-post} \ (H::(\forall::\text{object} \rightarrow \forall)) - \rightarrow \text{includes}_{\text{Set}}(x))\)

shows \(\tau \models \text{not}(x \triangleq \text{null})\)

lemma represented-generic-objects-defined:

assumes A: \(\tau \models (\text{OclAllInstances-generic pre-post} \ (H::(\forall::\text{object} \rightarrow \forall)) - \rightarrow \text{includes}_{\text{Set}}(x))\)

shows \(\tau \models \delta(\text{OclAllInstances-generic pre-post} \ H) \land \tau \models \delta x\)

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

lemma represented-generic-objects-in-state:

assumes A: \(\tau \models (\text{OclAllInstances-generic pre-post} \ H - \rightarrow \text{includes}_{\text{Set}}(x))\)

shows \(x \ \tau \in (\text{Some o} \ H) \cdot \text{ran}(\text{heap}(\text{pre-post} \ \tau))\)


Here comes a couple of operational rules that allow to infer the value of \( \text{oclAllInstances} \) from the context \( \tau \). These rules are a special-case in the sense that they are the only rules that relate statements with different \( \tau \)'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-empty:**

**Assumes** [simp]: \( \forall a. \text{pre-post} (mk \ a) = a \)

**Shows** \( (mk (\text{heap}=\text{empty}, \text{assocs}=A)) \models \text{OclAllInstances-generic pre-post Type} \equiv \text{Set}() \)

**Lemma state-update-vs-allInstances-generic-including:**

**Assumes** [simp]: \( \forall a. \text{pre-post} (mk \ a) = a \)

**Assumes** \( \forall x. \sigma' \text{oid} = \text{Some} \ x \Rightarrow x = \text{Object} \)

**And** Type \( \text{Object} \neq \text{None} \)

**Shows** \( \text{OclAllInstances-generic pre-post Type} \)

\[
\begin{aligned}
& (mk (\text{heap}=\sigma' (\text{oid} \to \text{Object}), \text{assocs}=A)) \\
= & \quad ((\text{OclAllInstances-generic pre-post Type}) \to \text{inclusion}_{\text{Set}} (\lambda \cdot \cdot \cdot \text{drop} (\text{Type Object} \cdot \cdot \cdot))) \\
\end{aligned}
\]

**Lemma state-update-vs-allInstances-generic-including:**

**Assumes** [simp]: \( \forall a. \text{pre-post} (mk \ a) = a \)

**Assumes** \( \forall x. \sigma' \text{oid} = \text{Some} \ x \Rightarrow x = \text{Object} \)

**And** Type \( \text{Object} \neq \text{None} \)

**Shows** \( \text{OclAllInstances-generic pre-post Type} \)

\[
\begin{aligned}
& (mk (\text{heap}=\sigma' (\text{oid} \to \text{Object}), \text{assocs}=A)) \\
= & \quad ((\lambda \cdot . (\text{OclAllInstances-generic pre-post Type}) \to \text{inclusion}_{\text{Set}} (\lambda \cdot . \cdot \text{drop} (\text{Type Object} \cdot \cdot \cdot))) \\
\end{aligned}
\]

**Lemma state-update-vs-allInstances-generic-noincluding:**

**Assumes** [simp]: \( \forall a. \text{pre-post} (mk \ a) = a \)

**Assumes** \( \forall x. \sigma' \text{oid} = \text{Some} \ x \Rightarrow x = \text{Object} \)

**And** Type \( \text{Object} = \text{None} \)

**Shows** \( \text{OclAllInstances-generic pre-post Type} \)

\[
\begin{aligned}
& (mk (\text{heap}=\sigma' (\text{oid} \to \text{Object}), \text{assocs}=A)) \\
= & \quad (\text{OclAllInstances-generic pre-post Type}) \\
\end{aligned}
\]

**Theorem state-update-vs-allInstances-generic-ntc:**

**Assumes** [simp]: \( \forall a. \text{pre-post} (mk \ a) = a \)

**Assumes** oid-def: \( \text{oid} \notin \text{dom} \sigma' \)

**And** non-type-conform: Type \( \text{Object} = \text{None} \)

**And** cp-cxt: \( \text{ep} \ P \)

**And** const-cxt: \( \forall X. \text{const} \ x \Rightarrow \text{const} \ (P \ x) \)

**Shows** \( (mk (\text{heap}=\sigma' (\text{oid} \to \text{Object}), \text{assocs}=A)) \models P (\text{OclAllInstances-generic pre-post Type}) = \\
(mk (\text{heap}=\sigma', \text{assocs}=A)) \models P (\text{OclAllInstances-generic pre-post Type}) \\
(\text{is} (\text{?} \models P \ ? \phi) = (\text{?} \models P \ ? \phi)) \)

82
One way to establish the actual presence of an object representation in a state is:

lemma state-update-vs-allInstances-at-post-empty:
assumes A: \( \tau \models H .allInstances() \Rightarrow \text{includes}_\mathcal{S}(x) \)
shows \((\sigma, (\text{heap}=\text{empty}, \text{assocs}=\text{A})) \models \text{Type .allInstances() \Rightarrow Set(\{\})}\)

Here comes a couple of operational rules that allow to infer the value of \( \text{oclAllInstances} \) from the context \( \tau \). These rules are a special-case in the sense that they are the only rules that relate statements with different \( \tau \)'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 straightforward by hand).

lemma state-update-vs-allInstances-at-post-including:
assumes \( \forall x. \sigma' .\text{oid} = \text{Some} \ x \Rightarrow x = \text{Object} \)
and Type Object \( \neq \text{None} \)
shows \((Type .\text{allInstances}()) \Rightarrow (\sigma, (\text{heap}=\sigma' .(\text{oid}=\text{Object}), \text{assocs}=\text{A})) \Rightarrow \)
\begin{verbatim}
((Type.allInstances()) -> includingSet(\lambda \_ . \_ drop (Type Object) \_))
(\sigma, ([heap=\sigma', assocs=A]))

lemma state-update-vs-allInstances-at-post-including:
assumes \( \forall x. \sigma' \_ oid = \text{Some} x \implies x = \text{Object} \)
and Type Object \neq \text{None}
shows \( (Type.allInstances())\)
\( (\sigma, ([heap=\sigma'(oid->Object), assocs=A])) \) =
\( (\lambda \_ . (Type.allInstances())) \)
\( (\sigma, ([heap=\sigma', assocs=A])) \rightarrow includingSet(\lambda \_ . \_ drop (Type Object) \_)) \)
(\sigma, ([heap=\sigma'(oid->Object), assocs=A]))

lemma state-update-vs-allInstances-at-post-noincluding:
assumes \( \forall x. \sigma' \_ oid = \text{Some} x \implies x = \text{Object} \)
and Type Object = \text{None}
shows \( (Type.allInstances())\)
\( (\sigma, ([heap=\sigma'(oid->Object), assocs=A])) \) =
\( (Type.allInstances()) \)
(\sigma, ([heap=\sigma', assocs=A]))

theorem state-update-vs-allInstances-at-post-ntc:
assumes oid-def: \( oid \notin \text{dom} \sigma' \)
and non-type-conform: Type Object = \text{None}
and cp-ctxt: \( \text{cp } P \)
and const-ctxt: \( \forall X. \text{const } X \implies \text{const } (P X) \)
shows \( ((\sigma, ([heap=\sigma'(oid->Object), assocs=A])) \downarrow (P(\text{Type.allInstances}))) \) =
\( ((\sigma, ([heap=\sigma', assocs=A])) \downarrow (P(\text{Type.allInstances}))) \)

theorem state-update-vs-allInstances-at-post-tc:
assumes oid-def: \( oid \notin \text{dom} \sigma' \)
and type-conform: Type Object \neq \text{None}
and cp-ctxt: \( \text{cp } P \)
and const-ctxt: \( \forall X. \text{const } X \implies \text{const } (P X) \)
shows \( ((\sigma, ([heap=\sigma'(oid->Object), assocs=A])) \downarrow (P(\text{Type.allInstances}))) \) =
\( ((\sigma, ([heap=\sigma', assocs=A])) \downarrow (P(\text{Type.allInstances}))) \rightarrow \text{includingSet}(\lambda \_ . (Type Object) \_)) \)
\end{verbatim}
lemma represented-at-pre-objects-nonnull:
assumes \( A : \tau |\ (H::\forall x::object \rightarrow 'a).allInstances@pre()) \rightarrow\text{includes}_{Set}(x) \)
shows \( \tau |\ \text{not}(x \triangleq \text{null}) \)

lemma represented-at-pre-objects-defined:
assumes \( A : \tau |\ ((H::\forall x::object \rightarrow 'a).allInstances@pre()) \rightarrow\text{includes}_{Set}(x) \)
shows \( \tau |\ \delta (H . allInstances@pre()) \land \tau |\ \delta x \)

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

lemma
assumes \( A : \tau |\ H . allInstances@pre() \rightarrow\text{includes}_{Set}(x) \)
shows \( \tau |\ x \tau \in (\text{Some o H} \ ' \ \text{ran} (heap(fst \tau)) \)

lemma state-update-vs-allInstances-at-pre-empty:
shows \((\exists \| \text{heap=empty}, \text{assocs}=A\|, \sigma) |\ \text{Type . allInstances@pre()} = \text{Set}{} \)

Here comes a couple of operational rules that allow to infer the value of oclAllInstances@pre from the context \( \tau \). These rules are a special-case in the sense that they are the only rules that relate statements with different \( \tau \)'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 \( \forall x . \sigma' \text{oid} = \text{Some x} \Rightarrow x = \text{Object} \)
and \( \text{Type Object} \neq \text{None} \)
shows \( (\text{Type . allInstances@pre()} ) \)
\( (\| \text{heap=\sigma'(oid->Object), assocs=A\|}, \sigma) = (\text{Type . allInstances@pre()}) \rightarrow\text{including}_{Set}(\lambda . \text{drop (Type Object)}) \)
\( (\| \text{heap=\sigma', assocs=A\|}, \sigma) \)

lemma state-update-vs-allInstances-at-pre-including:
assumes \( \forall x . \sigma' \text{oid} = \text{Some x} \Rightarrow x = \text{Object} \)
and \( \text{Type Object} \neq \text{None} \)
shows \( (\text{Type . allInstances@pre()} ) \)
\( (\| \text{heap=\sigma'(oid->Object), assocs=A\|}, \sigma) = (\lambda . (\text{Type . allInstances@pre()} ) \rightarrow\text{including}_{Set}(\lambda . \text{drop (Type Object)}) \)
\( (\| \text{heap=\sigma', assocs=A\|}, \sigma) \)

lemma state-update-vs-allInstances-at-pre-noincluding' :
assumes \( \forall x . \sigma' \text{oid} = \text{Some x} \Rightarrow x = \text{Object} \)
and \( \text{Type Object} = \text{None} \)
shows \( (\text{Type . allInstances@pre()} ) \)
\( (\| \text{heap=\sigma'(oid->Object), assocs=A\|}, \sigma) = (\text{Type . allInstances@pre()}) \)

85
shows (((heap=σ', assocs=A], σ)\]

\textbf{theorem} \textit{state-update-vs-allInstances-at-pre-ntc}:
\textbf{assumes} oid-def: oid\notin\text{dom} \ σ'
\textbf{and} non-type-conform: Type Object = None
\textbf{and} cp-ctxt: cp \ P
\textbf{and} const-ctxt: \(\forall X. \text{const} X \implies \text{const} (P X)\)
\textbf{shows} (((heap=σ'(oid→Object), assocs=A], σ)\] |= (P(Type.allInstances@pre())) = (((heap=σ', assocs=A], σ)\] |= (P(Type.allInstances@pre()))

\textbf{theorem} \textit{state-update-vs-allInstances-at-pre-tc}:
\textbf{assumes} oid-def: oid\notin\text{dom} \ σ'
\textbf{and} type-conform: Type Object \neq \text{None}
\textbf{and} cp-ctxt: cp \ P
\textbf{and} const-ctxt: \(\forall X. \text{const} X \implies \text{const} (P X)\)
\textbf{shows} (((heap=σ'(oid→Object), assocs=A], σ)\] |= (P(Type.allInstances@pre())) = (((heap=σ', assocs=A], σ)\] |= (P((Type.allInstances@pre()))

\(\text{\texttt{@post or @pre theorem} StrictRefEq\_Object-vs-StrongEq'':}
\textbf{assumes} \text{WFF}: \text{WFF} \ τ
\textbf{and} valid-x: \(\tau\models (u (x :: (\mathcal{A}:\text{object}, \mathcal{A}:\text{object option option}|val))
\textbf{and} valid-y: \(\tau\models (u y)
\textbf{and} oid-preserve: \(\forall x. x \in \text{ran} (\text{heap} (\text{fst} \ τ)) \lor x \in \text{ran} (\text{heap} (\text{snd} \ τ)) \implies \text{oid-of} (H x) = \text{oid-of} x
\textbf{and} xy-together: \(\tau\models ((H .\text{allInstances}) \rightarrow \text{includes}_{\text{Set}}(x) \text{ and } H .\text{allInstances}) \rightarrow \text{includes}_{\text{Set}}(y)) \text{ or } ((H .\text{allInstances}@pre()) \rightarrow \text{includes}_{\text{Set}}(x) \text{ and } H .\text{allInstances}@pre@) \rightarrow \text{includes}_{\text{Set}}(y))
\textbf{shows} ((\tau \models (\text{StrictRefEq}\_\text{Object} \ x \ y)) = (\tau \models (x \triangleleft y))

\textbf{OclIsNew, OclIsDeleted, OclIsMaintained, OclIsAbsent}

\textbf{definition} \textit{OclIsNew}:: (X, \mathcal{A}:null.object)val ⇒ (\mathcal{A}Boolean (\text{-}.oclIsNew''))
\textbf{where} \text{X .oclIsNew()} \equiv (\lambda \tau. \text{if } (\delta X) \ \tau = \text{true} \ \tau)
\text{then } \text{oid-of} (X \ \tau) \notin \text{dom} (\text{heap} (\text{fst} \ \tau)) \land
\text{oid-of} (X \ \tau) \in \text{dom} (\text{heap} (\text{snd} \ \tau)) \land
\text{else } \text{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.

\textbf{definition} \textit{OclIsDeleted}:: (X, \mathcal{A}:null.object)val ⇒ (\mathcal{A}Boolean (\text{-}.oclIsDeleted''))
\textbf{where} \text{X .oclIsDeleted()} \equiv (\lambda \tau. \text{if } (\delta X) \ \tau = \text{true} \ \tau)
\text{then } \text{oid-of} (X \ \tau) \in \text{dom} (\text{heap} (\text{fst} \ \tau)) \land
\text{oid-of} (X \ \tau) \notin \text{dom} (\text{heap} (\text{snd} \ \tau)) \land
\text{else } \text{invalid} \ \tau)

\textbf{definition} \textit{OclIsMaintained}:: (X, \mathcal{A}:null.object)val ⇒ (\mathcal{A}Boolean (\text{-}.oclIsMaintained''))
\textbf{where} \text{X .oclIsMaintained()} \equiv (\lambda \tau. \text{if } (\delta X) \ \tau = \text{true} \ \tau)
\text{then } \text{oid-of} (X \ \tau) \in \text{dom} (\text{heap} (\text{fst} \ \tau)) \land
\text{oid-of} (X \ \tau) \in \text{dom} (\text{heap} (\text{snd} \ \tau)) \land
\text{else } \text{invalid} \ \tau)
\[ \text{definition } OclIsAbsent \:: (\forall \alpha::\{null,\text{object}\})\text{val} \Rightarrow (\forall)\text{Boolean} \quad ((\cdot).\text{oclIsAbsent})' \]

\text{where } X .\text{oclIsAbsent()} \equiv (\lambda \tau . \text{if } (\delta X) \tau = \text{true } \tau \text{ then } \text{oid-of } (X \tau) \notin \text{dom(heap(fst } \tau) ) \wedge \text{oid-of } (X \tau) \notin \text{dom(heap(snd } \tau) )' \text{ else invalid } \tau) \]

\[ \text{lemma } \text{state-split } : \tau \models \delta X \implies \]
\[ \tau \models (X .\text{oclIsNew()} ) \vee \tau \models (X .\text{oclIsDeleted()} ) \vee \]
\[ \tau \models (X .\text{oclIsMaintained()} ) \vee \tau \models (X .\text{oclIsAbsent()} ) \]

\[ \text{lemma } \text{notNew-vs-others } : \tau \models \delta X \implies \]
\[ (\neg \tau \models (X .\text{oclIsNew()} )) \Rightarrow (\tau \models (X .\text{oclIsDeleted()} ) \vee \]
\[ \tau \models (X .\text{oclIsMaintained()} ) \vee \tau \models (X .\text{oclIsAbsent()} ) \]

\textbf{OclIsModifiedOnly}

\textbf{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.

\[ \text{definition } \text{OclIsModifiedOnly } :: (\forall \alpha::\{null,\text{object}\})\text{Set} \Rightarrow (\forall)\text{Boolean} \quad ((\cdot).\text{oclIsModifiedOnly})' \]

\text{where } X \rightarrow\text{oclIsModifiedOnly}() \equiv (\lambda (\sigma,\sigma'). \text{let } \sigma' = (\text{oid-of } '(\text{Rep-Set_base}(X(\sigma,\sigma'))')) ; \]
\[ S = ((\text{dom (heap } \sigma) \cap \text{dom (heap } \sigma')) - X') \]
\[ \text{in } (\delta X) ((\sigma,\sigma')) = \text{true } ((\forall x \in (\text{Rep-Set_base}(X(\sigma,\sigma')))). x \neq \text{null} ) \]
\[ \text{then } (\forall x \in S . (\text{heap } \sigma) x = (\text{heap } \sigma') x \wedge \text{else invalid } ((\sigma,\sigma')) ) \]

\textbf{Execution with Invalid or Null or Null Element as Argument} \textbf{lemma } \text{invalid } \rightarrow\text{oclIsModifiedOnly}() = \text{invalid}

\[ \text{lemma } \text{null } \rightarrow\text{oclIsModifiedOnly}() = \text{invalid} \]

\[ \text{lemma } \text{assumes } X\text{-null } : \tau \models X \rightarrow\text{includesSet}(\text{null}) \]
\[ \text{shows } \tau \models X \rightarrow\text{oclIsModifiedOnly}() \equiv \text{invalid} \]

\textbf{Context Passing} \textbf{lemma } cp\text{-OclIsModifiedOnly } : X \rightarrow\text{oclIsModifiedOnly}() \tau = (\lambda . X \tau) \rightarrow\text{oclIsModifiedOnly}() \tau

\textbf{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.

\[ \text{definition } \text{simp } \text{OclSelf } x \text{ if } \text{fst-snd } = (\lambda \tau . \text{if } (\delta x) \tau = \text{true } \tau \text{ then } \text{oid-of } (x \tau) \in \text{dom(heap(fst } \tau)) \wedge \text{oid-of } (x \tau) \in \text{dom(heap(snd } \tau)) \]
then $H \vdash (\text{heap}(\text{fst } \tau)) (\text{oid-of } (x \tau))$
else invalid $\tau$
else invalid $\tau$

**Definition** $\text{OclSelf-at-pre} :: (\text{def-}x : \tau) \Rightarrow (\exists x : \text{object} \Rightarrow (\forall x : \text{object} \Rightarrow \text{val} ((\cdot) @ \text{pre}(-))))$

**Definition** $\text{OclSelf-at-post} :: (\text{def-}x : \tau) \Rightarrow (\exists x : \text{object} \Rightarrow (\forall x : \text{object} \Rightarrow \text{val} ((\cdot) @ \text{post}(-))))$

**Framing Theorem**

**Lemma** $\text{all-oid-diff}$

**assumes** $\text{def-}x : \tau \models \delta x$

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

**assumes** $\text{def-}X' : \forall x \in \text{Rep-Set} \Rightarrow (X \tau) \Rightarrow x \neq \text{null}$

**defines** $P \equiv \lambda a. \text{not} ((\text{StrictEqObject } x a))$

**shows** $\text{modifies clause} \vdash (\forall x \in \text{Rep-Set} (a) P a) = (\text{oid-of } (x \tau) \notin \text{oid-of } (\text{Rep-Set} (X \tau))]$

**Theorem** $\text{framing}$

**assumes** $\text{modifies clause} : \tau \models (X \Rightarrow \text{forAllSet}(a) \Rightarrow \text{oclIsModifiedOnly}())$

**and** $\text{oid-is-typerepr} : \tau \models (X \Rightarrow \text{forAllSet}(a) \Rightarrow \text{not} ((\text{StrictEqObject } x a))$

**shows** $\tau \models (x @ \text{pre } P \triangleq (x @ \text{post } P))$

As corollary, the framing property can be expressed with only the strong equality as comparison operator.

**Theorem** $\text{framing'}$

**assumes** $\text{wff} : WFF \tau$

**assumes** $\text{modifies clause} : \tau \models (X \Rightarrow \text{forAllSet}(a) \Rightarrow \text{oclIsModifiedOnly}())$

**and** $\text{oid-is-typerepr} : \tau \models (X \Rightarrow \text{forAllSet}(a) \Rightarrow \text{not} ((x \triangleq a))$

**and** $\text{oid-preserve} : \forall x . \text{ran} (\text{heap}(\text{fst } \tau)) \Rightarrow x \in \text{ran} (\text{heap}(\text{snd } \tau)) \Rightarrow$

$\text{oid-of } (H x) = \text{oid-of } x$

**and** $\text{xy-together}$

$\tau \models (X \Rightarrow \text{forAllSet}(y) \Rightarrow (H \text{allInstances}(\Rightarrow ) \Rightarrow \text{includesSet}(x) \Rightarrow H \text{allInstances}(\Rightarrow ) \Rightarrow \text{includesSet}(y))$ or

$\tau \models (H \Rightarrow \text{allInstances}(\Rightarrow ) \Rightarrow \text{includesSet}(x) \Rightarrow H \Rightarrow \text{allInstances}(\Rightarrow ) \Rightarrow \text{includesSet}(y))$

**shows** $\tau \models (x @ \text{pre } P \triangleq (x @ \text{post } P))$

**Miscellaneous**

**Lemma** $\text{pre-post-new} : \tau \models (x \Rightarrow \text{oclIsNew}()) \Rightarrow (\neg (\tau \models \text{val}(x @ \text{pre } H1)) \land \neg (\tau \models \text{val}(x @ \text{post } H2))$

**Lemma** $\text{pre-post-old} : \tau \models (x \Rightarrow \text{oclIsDeleted}()) \Rightarrow (\neg (\tau \models \text{val}(x @ \text{pre } H1)) \land \neg (\tau \models \text{val}(x @ \text{post } H2))$

**Lemma** $\text{pre-post-absent} : \tau \models (x \Rightarrow \text{oclIsAbsent}()) \Rightarrow (\neg (\tau \models \text{val}(x @ \text{pre } H1)) \land \neg (\tau \models \text{val}(x @ \text{post } H2))$

**Lemma** $\text{pre-post-maintained} : \tau \models (x \Rightarrow \text{oclIsModified}()) \Rightarrow (\tau \models (x @ \text{pre } H1) \lor \tau \models (x @ \text{post } H2)) \Rightarrow \tau \models (x \Rightarrow \text{oclIsMaintained}())$

88
lemma pre-post-maintained:\n\[ \tau \models (x \text{oclIsMaintained}()) \implies (\tau \models \psi x @ \text{pre} (\text{Some } H1)) \land \tau \models \psi x @ \text{post} (\text{Some } H2)) \]

lemma framing-same-state: \((\sigma, \sigma) \models (x @ \text{pre} H \triangleq (x @ \text{post} H))\)

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 \(f-\psi\)
  fixes \(f-\lambda\)
  fixes \(f::(\forall \alpha . \text{null} ) \text{val} \Rightarrow \)
  \(\lambda \Rightarrow (\forall \alpha . \text{null} ) \text{val}\)
  fixes \(\text{PRE}\)
  fixes \(\text{POST}\)
  assumes def-scheme:\n  \(f \text{ self } x \equiv (\lambda \tau . \text{ if } (\tau \models (\delta \text{ self})) \land f-\psi x \tau \)
  \(\text{ then } \text{ SOME res. (}\tau \models \text{ PRE self x} ) \land \)
  \(\tau \models \text{ POST self x (}\lambda . \text{ res} )) \)
  \(\text{ else invalid } \tau)\)
  assumes all-post:\n  \(\forall \sigma \sigma' . ((\sigma, \sigma') \models \text{ PRE self x} ) = ((\sigma, \sigma') \models \text{ PRE self x})\)
  assumes \(\text{cp PRE} : \text{ PRE (self) } x \tau = \text{ PRE (}\lambda . \text{ self x} ) (f-\lambda x \tau) \tau\)
  assumes \(\text{cp POST} : \text{ POST (self) x (res) } \tau = \text{ POST (}\lambda . \text{ self x}) (f-\lambda x \tau) (\lambda . \text{ res) } \tau\)
  assumes \(f-\psi-\text{val} : \forall a1 . f-\psi (f-\lambda a1 \tau) \tau = f-\psi a1 \tau\)
begin
lemma strict0 [simp]: \(f \text{ invalid X} = \text{ invalid}\)
lemma nullstrict0 [simp]: \(f \text{ null X} = \text{ invalid}\)
lemma cp0 : \(f \text{ self a1} ) \tau = f (\lambda . \text{ self x} ) (f-\lambda a1 \tau) \tau\)

theorem unfold:\n  assumes context-ok: \(\text{ cp E}\)
  and args-def-or-valid: \(\tau \models (\delta \text{ self}) \land f-\psi a1 \tau\)
  and pre-satisfied: \(\tau \models \text{ PRE self a1}\)
  and post-satisfiable: \(\exists \text{ res. (}\tau \models \text{ POST self a1 (}\lambda . \text{ res) )}\)
  and sat-for-sols-post: \(\left( \forall \text{ res. } \tau \models \text{ POST self a1 (}\lambda . \text{ res) } \implies \tau \models E (\lambda . \text{ res) }}\right)\)
  shows \(\tau \models E(f \text{ self a1})\)

lemma unfold2:\n  assumes context-ok: \(\text{ cp E}\)
  and args-def-or-valid: \(\tau \models (\delta \text{ self}) \land (f-\psi a1 \tau)\)
  and pre-satisfied: \(\tau \models \text{ PRE self a1}\)
  and post-split-satisfied: \(\tau \models \text{ POST' self a1}\)
  and post-decomposable: \(\left( \forall \text{ res. (POST self a1 res) } = \right)\)
  \(\left( (\text{POST' self a1}) \text{ and (res } \triangleq \text{ (BODY self a1)) }\right)\)
  shows \(\tau \models E(f \text{ self a1}) = (\tau \models E(\text{BODY self a1}))\)
end

locale contract0 =
fixes $f :: (\mathbb{N}, \alpha_0::\text{null})\text{val} \Rightarrow (\mathbb{N}, \alpha_1::\text{null})\text{val}$

fixes $\text{PRE}$

fixes $\text{POST}$

assumes $\text{def-scheme}: f \text{self} \equiv (\lambda \tau. \text{if } (\tau \models (\delta \text{self})) \text{ then SOME res. } (\tau \models \text{PRE self}) \land (\tau \models \text{POST self } (\lambda -. \text{res})) \text{ else invalid } \tau)$

assumes $\text{all-post}: \forall \sigma \sigma' \sigma''. ((\sigma, \sigma') \models \text{PRE self}) = ((\sigma, \sigma'') \models \text{PRE self})$

assumes $\text{cp\text{-}PRE}: \text{PRE } (\lambda -. \text{self } \tau) \tau$

assumes $\text{cp\text{-}POST}: (\text{POST } (\lambda -. \text{self } \tau)) (\lambda -. \text{res } \tau) \tau$

sublocale contract0 < contract-scheme \lambda -. True \lambda x -. \lambda x -. \lambda x -. \lambda x -. \text{PRE } \lambda x -. \lambda x -. \text{POST } x$

context contract0

begin

lemma $\text{cp\text{-}pre}: \text{cp } \text{self } \tau \Rightarrow \text{cp } (\lambda X. \text{PRE } (\text{self } X))$

lemma $\text{cp\text{-}post}: \text{cp } \text{self } \tau \Rightarrow \text{cp } \text{res } \tau \Rightarrow \text{cp } (\lambda X. \text{POST } (\text{self } X))$

lemma $\text{cp\text{-}simp}: \text{cp } \text{self } \tau \Rightarrow \text{cp } \text{res } \tau \Rightarrow \text{cp } (\lambda X. f (\text{self } X))$

lemmas unfold = unfold'[simplified]

lemma unfold2 :

assumes $\text{cp } E$

and $(\tau \models (\delta \text{self}))$

and $(\tau \models \text{PRE self})$

and $(\tau \models \text{POST self})$

and $\land \text{res. } (\text{POST self } \text{res})$

shows $(\tau \models E(\text{f self})) = (\tau \models E(\text{BODY self}))$
assumes \( cp_{\text{POST}} : \text{POST} \ (\text{self}) \ (a1) \ (\text{res}) \ \tau = \text{POST} \ (\lambda \cdot \text{self} \ \tau)(\lambda \cdot \ a1 \ \tau) \ (\lambda \cdot \ \text{res} \ \tau) \ \tau \)

**sublocale** contract1 < contract-scheme \( \lambda a1 \ \tau . \ (\tau \vdash \nu \ a1) \ \lambda a1 \ \tau . \ (\lambda \cdot \ a1 \ \tau) \)

**context** contract1

**begin**

lemma strict1[simp]: \( f \text{ invalid} = \text{invalid} \)

lemma cp-pre: cp self' \( \Rightarrow \) cp a1' \( \Rightarrow \) cp \( \lambda \chi. \ \text{PRE} \ (\text{self'} X) \ (a1' X) \)

lemma cp-post: cp self' \( \Rightarrow \) cp a1' \( \Rightarrow \) cp res'

\[ \Rightarrow cp \ (\lambda \chi. \ \text{POST} \ (\text{self'} X) \ (a1' X) \ (\text{res'} X)) \]

lemma cp[simp]: cp self' \( \Rightarrow \) cp a1' \( \Rightarrow \) cp res'

\[ \Rightarrow cp \ (\lambda \chi. \ f \ (\text{self'} X) \ (a1' X)) \]

lemmas unfold = unfold'

lemmas unfold2 = unfold2'

**end**

**locale** contract2 =

**fixes** \( f :: (\forall \alpha0 : \text{null}) \text{val} \Rightarrow \ (\forall \alpha1 : \text{null}) \text{val} \Rightarrow \ (\forall \alpha2 : \text{null}) \text{val} \Rightarrow \ (\forall \alpha \text{res} : \text{null}) \text{val} \)

**fixes** \( \text{PRE} \)

**fixes** \( \text{POST} \)

assumes def-scheme: \( f \text{ self a1 a2} \equiv \)

\[ (\lambda \ \tau. \ \text{if} \ (\tau \vdash (\delta \ \text{self})) \ \wedge \ (\tau \vdash v \ a1) \ \wedge \ (\tau \vdash v \ a2)) \]

then SOME res. \( \ (\tau \vdash \text{PRE} \ \text{self a1 a2}) \ \wedge \)

\( (\tau \vdash \text{POST} \ \text{self a1 a2} \ (\lambda \cdot \ \text{res})) \)

else invalid \( \tau \)

assumes all-post: \( \forall \ \sigma \ \sigma' \ \sigma''. \ ((\sigma, \sigma') \vdash \text{PRE} \ \text{self a1 a2}) = ((\sigma, \sigma'') \vdash \text{PRE} \ \text{self a1 a2}) \)

assumes cppre: \( \text{PRE} \ (\text{self}) \ (a1) \ (a2) \ \tau = \text{PRE} \ (\lambda \cdot \ \text{self} \ \tau)(\lambda \cdot \ a1 \ \tau)(\lambda \cdot \ a2 \ \tau) \ \tau \)

assumes cpPOST : \( \text{res. POST} \ (\text{self}) \ (a1) \ (a2) \ (\text{res}) \ \tau = \)

\[ \text{POST} \ (\lambda \cdot \ \text{self} \ \tau)(\lambda \cdot \ a1 \ \tau)(\lambda \cdot \ a2 \ \tau)(\lambda \cdot \ \text{res} \ \tau) \ \tau \]

**sublocale** contract2 < contract-scheme \( \lambda (a1,a2) \ \tau . \ (\tau \vdash \nu \ a1) \ \wedge \ (\tau \vdash \nu \ a2) \)

\[ \lambda (a1,a2) \ \tau . (\lambda \cdot \ a1 \ \tau, \lambda \cdot \ a2 \ \tau) \]

\[ (\lambda x \ (a,b), \ f x \ a \ b) \]

\[ (\lambda x \ (a,b), \ \text{PRE} \ x \ a \ b) \]

\[ (\lambda x \ (a,b), \ \text{POST} \ x \ a \ b) \]

**context** contract2

**begin**

lemma strict0[simp]: \( f \text{ invalid X Y} = \text{invalid} \)

lemma nullstrict0[simp]: \( \text{null X Y} = \text{invalid} \)

lemma strict1[simp]: \( f \text{ self invalid Y} = \text{invalid} \)
lemma strict2[simp]: f self X invalid = invalid

lemma cp-pre: cp self' ===> cp a1' ===> cp a2' ===> cp (λX. PRE (self'X) (a1'X) (a2'X) )

lemma cp-post: cp self' ===> cp a1' ===> cp a2' ===> cp res'
  ===> cp (λX. POST (self'X) (a1'X) (a2'X) (res'X))

lemma cp0: f self a1 a2 τ = f (λ - . self) (λ - . a1) (λ - . a2) τ

lemma cp [simp]: cp self' ===> cp a1' ===> cp a2' ===> cp res'
  ===> cp (λX. f (self'X) (a1'X) (a2'X))

theorem unfold :
  assumes cp E
  and (τ |= δ self) ∧ (τ |= υ a1) ∧ (τ |= υ a2)
  and τ |= PRE self a1 a2
  and ∃ res. (τ |= POST self a1 a2 (λ - . res))
  and (∀ res. τ |= POST self a1 a2 (λ - . res) ===> τ |= E (λ - . res))
  shows τ |= E(f self a1 a2)

lemma unfold2 :
  assumes cp E
  and (τ |= δ self) ∧ (τ |= υ a1) ∧ (τ |= υ a2)
  and τ |= PRE self a1 a2
  and τ |= POST' self a1 a2
  and (∀ res. (POST self a1 a2 res) =
        (POST' self a1 a2) and (res ≜ (BODY self a1 a2))))
  shows (τ |= E(f self a1 a2)) = (τ |= E(BODY self a1 a2))
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].

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 typePerson = mkPerson oid
  int option
```

```
datatype typeOclAny = mkOclAny oid
  (int option) option
```

Now, we construct a concrete “universe of OclAny types” by injection into a sum type containing the class types. This type of OclAny will be used as instance for all respective type-variables.

```
datatype A = inPerson typePerson | inOclAny typeOclAny
```

Having fixed the object universe, we can introduce type synonyms that exactly correspond to OCL types. Again, we exploit that our representation of OCL is a “shallow embedding” with a one-to-one correspondance of OCL-types to types of the meta-language HOL.

```
type-synonym Boolean = A Boolean
type-synonym Integer = A Integer
type-synonym Void = A Void
type-synonym OclAny = (A, typeOclAny option option) val
type-synonym Person = (A, typePerson option option) val
type-synonym Set-Integer = (A, int option option) Set
```

Just a little check:
To reuse key-elements of the library like referential equality, we have to show that the object universe belongs to the type class “oclany,” i.e., each class type has to provide a function oid-of yielding the object id (oid) of the object.

```
instantiation typePerson :: object
begin
  definition oid-of-typePerson-def: oid-of x = (case x of mkPerson oid - => oid)
  instance
end

instantiation typeOclAny :: object
begin
  definition oid-of-typeOclAny-def: oid-of x = (case x of mkOclAny oid - => oid)
  instance
end

instantiation A :: object
begin
  definition oid-of-A-def: oid-of x = (case x of
    inPerson person => oid-of person
    | inOclAny oclany => oid-of oclany)
  instance
end
```

### A.7.3. Instantiation of the Generic Strict Equality

We instantiate the referential equality on `Person` and `OclAny`

```
defs(overloaded) StrictRefEqObject-Person : (x::Person) = y  StrictRefEqObject x y
defs(overloaded) StrictRefEqObject-OclAny : (x::OclAny) = y  StrictRefEqObject x y
```

```
lemmas cps23[standard] =
  cp-StrictRefEqObject[def,of x::Person y::Person τ,]
  simplified StrictRefEqObject-Person[of symmetric]
  cp-intro(9) of P::Person =>PersonQ::Person =>Person, simplified StrictRefEqObject-Person[of symmetric]
StrictRefEqObject-def [of x::Person y::Person, simplified StrictRefEqObject-Person[of symmetric]]
StrictRefEqObject-defargs [of - x::Person y::Person, simplified StrictRefEqObject-Person[of symmetric]]
StrictRefEqObject-strict1 [of x::Person, simplified StrictRefEqObject-Person[of symmetric]]
StrictRefEqObject-strict2 [of x::Person, simplified StrictRefEqObject-Person[of 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 OclAsTypeOclAny :: 'α ⇒ OclAny ((-).oclAsType("OclAny"))
consts OclAsTypePerson :: 'α ⇒ Person ((-).oclAsType("Person"))

definition OclAsTypeOclAny.∀X = (λu. case u of inOclAny a ⇒ a
| inPerson (mkPerson oid a) ⇒ mkOclAny oid a)

lemma OclAsTypeOclAny.∀some: OclAsTypeOclAny.∀x ≠ None

defs (overloaded) OclAsTypeOclAny-OclAny:
(X::OclAny) .oclAsType(OclAny) ≡ X

defs (overloaded) OclAsTypeOclAny-Person:
(X::Person) .oclAsType(OclAny) ≡ (λτ. case X τ of
  ⊥ ⇒ invalid τ
| ⊥ ⇒ null τ
| mkPerson oid a ⇒ mkOclAny oid a)

definition OclAsTypePerson.∀X = (λu. case u of inPerson p ⇒ p
| inOclAny (mkOclAny oid a) ⇒ mkPerson oid a
| _ ⇒ None)

defs (overloaded) OclAsTypePerson-OclAny:
(X::OclAny) .oclAsType(Person) ≡ (λτ. case X τ of
  ⊥ ⇒ invalid τ
| ⊥ ⇒ null τ
| mkOclAny oid ⊥ ⇒ invalid τ (* down-cast exception *)
| mkOclAny oid a ⇒ mkPerson oid a)

defs (overloaded) OclAsTypePerson-Person:
(X::Person) .oclAsType(Person) ≡ X

Execution with Invalid or Null as Argument

lemma OclAsTypeOclAny-OclAny-strict : (invalid::OclAny) .oclAsType(OclAny) = invalid
lemma OclAsTypeOclAny-OclAny-nullstrict : (null::OclAny) .oclAsType(OclAny) = null
lemma OclAsTypeOclAny-Person-strict[simp] : (invalid::Person) .oclAsType(OclAny) = invalid
lemma OclAsTypeOclAny-Person-nullstrict[simp] : (null::Person) .oclAsType(OclAny) = null
lemma OclAsTypePersonOclAny-OclAny-strict[simp] : (invalid::OclAny) .oclAsType(Person) = invalid
lemma OclAsTypePersonOclAny-Person-nullstrict[simp] : (null::OclAny) .oclAsType(Person) = null
lemma OclAsTypePerson-Person-strict : (invalid::Person) .oclAsType(Person) = invalid
lemma OclAsTypePerson-Person-nullstrict : (null::Person) .oclAsType(Person) = null

A.7.5. OclIsTypeOf

Definition

consts OclIsTypeOfOclAny :: 'α ⇒ Boolean ((-).oclIsTypeOf("OclAny"))
consts OclIsTypeOfPerson :: 'α ⇒ Boolean ((-).oclIsTypeOf("Person"))
defs (overloaded) OclIsTypeOf OclAny-OclAny:
(X::OclAny).oclIsTypeOf(OclAny) ≡
(λ τ. case X τ of
    ⊥ ⇒ invalid τ
    | ⊥⇒ true τ (* invalid ?? *)
    | mkOclAny oid ⊥⇒ true τ
    | mkOclAny oid ⊣⇒ false τ)

lemma OclIsTypeOf OclAny-OclAny:
(X::OclAny).oclIsTypeOf(OclAny) =
(λ τ. if τ = ν X then (case X τ of
    ⊥⇒ true τ (* invalid ?? *)
    | mkOclAny oid ⊥⇒ true τ
    | mkOclAny oid ⊣⇒ false τ)
else invalid τ)

interpretation OclIsTypeOf OclAny-OclAny:
profile-mono-schemeV
OclIsTypeOf OclAny::OclAny ⇒ Boolean
λ X. (case X of
    None ⇒ True ∣ (* invalid ?? *)
    | mkOclAny oid None⇒ True
    | mkOclAny oid ⊣⇒ False)

defs (overloaded) OclIsTypeOf OclAny-Person:
(X::Person).oclIsTypeOf(OclAny) ≡
(λ τ. case X τ of
    ⊥⇒ invalid τ
    | ⊥⇒ true τ (* invalid ?? *)
    | ⊣⇒ false τ)

defs (overloaded) OclIsTypeOf Person-OclAny:
(X::OclAny).oclIsTypeOf(Person) ≡
(λ τ. case X τ of
    ⊥⇒ invalid τ
    | ⊥⇒ true τ
    | mkOclAny oid ⊥⇒ false τ
    | mkOclAny oid ⊣⇒ true τ)

defs (overloaded) OclIsTypeOf Person-Person:
(X::Person).oclIsTypeOf(Person) ≡
(λ τ. case X τ of
    ⊥⇒ invalid τ
    | ⊥⇒ true τ)

Execution with Invalid or Null as Argument

lemma OclIsTypeOf OclAny-Person-strict1![simp]:
(invalid::OclAny).oclIsTypeOf(Person) = invalid

lemma OclIsTypeOf OclAny-Person-strict2![simp]:
(null::OclAny).oclIsTypeOf(Person) = true

96
lemma OclIsTypeOf\(_\text{OclAny-Person-strict1}\) [simp]:
(invalid::Person).oclIsTypeOf(OclAny) = invalid
lemma OclIsTypeOf\(_\text{OclAny-Person-strict2}\) [simp]:
(null::Person).oclIsTypeOf(OclAny) = true
lemma OclIsTypeOf\(_\text{Person-OclAny-strict1}\) [simp]:
(invalid::OclAny).oclIsTypeOf(Person) = invalid
lemma OclIsTypeOf\(_\text{Person-OclAny-strict2}\) [simp]:
(null::OclAny).oclIsTypeOf(Person) = true
lemma OclIsTypeOf\(_\text{Person-Person-strict1}\) [simp]:
(invalid::Person).oclIsTypeOf(Person) = invalid
lemma OclIsTypeOf\(_\text{Person-Person-strict2}\) [simp]:
(null::Person).oclIsTypeOf(Person) = true

Up Down Casting

lemma actualType-larger-staticType:
assumes isdef: \(\tau \models (\delta X)\)
shows \(\tau \models (X::\text{Person}).\text{oclIsTypeOf}(\text{OclAny}) \triangleq \false\)

lemma down-cast-type:
assumes isOclAny: \(\tau \models (X::\text{OclAny}).\text{oclIsTypeOf}(\text{OclAny})\)
and non-null: \(\tau \models (\delta X)\)
shows \(\tau \models (X.\text{oclAsType}(\text{Person})) \triangleq \false\)

lemma down-cast-type':
assumes isOclAny: \(\tau \models (X::\text{OclAny}).\text{oclIsTypeOf}(\text{OclAny})\)
and non-null: \(\tau \models (\delta X)\)
shows \(\tau \models \not\left(\nu (X.\text{oclAsType}(\text{Person}))\right)\)

lemma up-down-cast:
assumes isdef: \(\tau \models (\delta X)\)
shows \(\tau \models ((X::\text{Person}).\text{oclAsType}(\text{OclAny}).\text{oclAsType}(\text{Person}) \triangleq X)\)

lemma up-down-cast-Person-OclAny-Person [simp]:
shows \((X::\text{Person}).\text{oclAsType}(\text{OclAny}).\text{oclAsType}(\text{Person}) = X)\)

lemma up-down-cast-Person-OclAny-Person':
assumes \(\tau \models \nu X\)
shows \(\tau \models ((X::\text{Person}).\text{oclAsType}(\text{OclAny}).\text{oclAsType}(\text{Person})) \triangleq X)\)

lemma up-down-cast-Person-OclAny-Person'':
assumes \(\tau \models \nu (X::\text{Person})\)
shows \(\tau \models (X.\text{oclIsTypeOf}(\text{Person}) \implies (X.\text{oclAsType}(\text{OclAny}).\text{oclAsType}(\text{Person})) \triangleq X)\)

A.7.6. OclIsKindOf

Definition
consts OclIsKindOf\(_\text{OclAny} :: \alpha \Rightarrow \text{Boolean}(\nu.\text{oclIsKindOf}(\nu\text{OclAny}))\)
consts OclIsKindOf\(_\text{Person} :: \alpha \Rightarrow \text{Boolean}(\nu.\text{oclIsKindOf}(\nu\text{Person}))\)
defs (overloaded) `OclIsKindOf_OclAny-OclAny`:
(X::OclAny).oclIsKindOf(OclAny) ≡
(λ τ. case X τ of
  ⊥ ⇒ invalid τ
  | - ⇒ true τ)

defs (overloaded) `OclIsKindOf_OclAny-Person`:
(X::Person).oclIsKindOf(OclAny) ≡
(λ τ. case X τ of
  ⊥ ⇒ invalid τ
  | - ⇒ true τ)

defs (overloaded) `OclIsKindOf_OclAny-Person`:
(X::OclAny).oclIsKindOf(Person) ≡
(λ τ. case X τ of
  ⊥ ⇒ invalid τ
  | ⊥ ⇒ true τ
  | mkOclAny oid ⊥ ⇒ false τ
  | mkOclAny oid ⊥ ⇒ true τ)

defs (overloaded) `OclIsKindOf_Person-Person`:
(X::Person).oclIsKindOf(Person) ≡
(λ τ. case X τ of
  ⊥ ⇒ invalid τ
  | - ⇒ true τ)

Execution with Invalid or Null as Argument

lemma `OclIsKindOf_OclAny-OclAny-strict1`[simp]: (invalid::OclAny).oclIsKindOf(OclAny) = invalid
lemma `OclIsKindOf_OclAny-OclAny-strict2`[simp]: (null::OclAny).oclIsKindOf(OclAny) = true
lemma `OclIsKindOf_OclAny-Person-strict1`[simp]: (invalid::Person).oclIsKindOf(OclAny) = invalid
lemma `OclIsKindOf_OclAny-Person-strict2`[simp]: (null::Person).oclIsKindOf(OclAny) = true
lemma `OclIsKindOf_Person-OclAny-strict1`[simp]: (invalid::Person).oclIsKindOf(Person) = invalid
lemma `OclIsKindOf_Person-OclAny-strict2`[simp]: (null::Person).oclIsKindOf(Person) = true
lemma `OclIsKindOf_Person-Person-strict1`[simp]: (invalid::Person).oclIsKindOf(Person) = invalid
lemma `OclIsKindOf_Person-Person-strict2`[simp]: (null::Person).oclIsKindOf(Person) = true

Up Down Casting

lemma `actualKind-larger-staticKind`:
assumes `isdef`: τ \models (δ X)
shows τ \models (X::Person).oclIsKindOf(OclAny) \equiv true

lemma `down-cast-kind`:
assumes `isOclAny`: ¬ (τ \models (X::OclAny).oclIsKindOf(Person))
and `non-null`: τ \models (δ X)
shows τ \models (X.oclAsType(Person)) \equiv invalid
A.7.7. OclAllInstances

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

**Definition**

\[
\text{Person} \equiv \text{OclAsType} \text{Person} \wedge \text{A}
\]

**Definition**

\[
\text{OclAny} \equiv \text{OclAsType} \text{OclAny} \wedge \text{A}
\]

**Lemmas**

\[
\text{OclAllInstances-generic}_{\text{OclAny}}: \text{pre-post OclAny} = (\lambda \tau. \text{Abs-Set}_{\text{base}} \downarrow \downarrow \text{Some } \downarrow \downarrow \text{OclAny } \downarrow \downarrow \text{ran (heap (pre-post \ \tau))})
\]

\[
\text{OclAllInstances-at-post}_{\text{OclAny}}: \text{OclAny} \downarrow \text{allInstances}() = (\lambda \tau. \text{Abs-Set}_{\text{base}} \downarrow \downarrow \text{Some } \downarrow \downarrow \text{OclAny } \downarrow \downarrow \text{ran (heap (snd \ \tau))})
\]

\[
\text{OclAllInstances-at-pre}_{\text{OclAny}}: \text{OclAny} \downarrow \text{allInstances}()@\text{pre}() = (\lambda \tau. \text{Abs-Set}_{\text{base}} \downarrow \downarrow \text{Some } \downarrow \downarrow \text{OclAny } \downarrow \downarrow \text{ran (heap (fst \ \tau))})
\]

**OclIsTypeOf**

\[
\text{OclAny-allInstances-generic-oclIsTypeOf}_{\text{OclAny}}: \exists \tau. (\tau \downarrow \downarrow (\text{OclAllInstances-generic pre-post OclAny} - \downarrow \downarrow \text{forAllSet}(X[X . oclIsTypeOf(OclAny)))))
\]

\[
\text{OclAny-allInstances-at-post-oclIsTypeOf}_{\text{OclAny}}: \exists \tau. (\tau \downarrow \downarrow (\text{OclAny-allInstances}() \downarrow \downarrow \text{forAllSet}(X[X . oclIsTypeOf(OclAny)))))
\]

\[
\text{OclAny-allInstances-at-pre-oclIsTypeOf}_{\text{OclAny}}: \exists \tau. (\tau \downarrow \downarrow (\text{OclAny-allInstances}()@\text{pre}() \downarrow \downarrow \text{forAllSet}(X[X . oclIsTypeOf(OclAny)))))
\]

\[
\text{Person-allInstances-generic-oclIsTypeOf}_{\text{OclAny}}: \tau \downarrow \downarrow (\text{OclAllInstances-generic pre-post Person} - \downarrow \downarrow \text{forAllSet}(X[X . oclIsTypeOf(Person))))
\]

\[
\text{Person-allInstances-at-post-oclIsTypeOf}_{\text{Person}}: \tau \downarrow \downarrow (\text{Person-allInstances}() \downarrow \downarrow \text{forAllSet}(X[X . oclIsTypeOf(Person))))
\]

\[
\text{Person-allInstances-at-pre-oclIsTypeOf}_{\text{Person}}: \tau \downarrow \downarrow (\text{Person-allInstances}()@\text{pre}() \downarrow \downarrow \text{forAllSet}(X[X . oclIsTypeOf(Person))))
\]
OclIsKindOf

lemma OclAny-allInstances-generic-oclIsKindOf_OclAny:
\( \tau \models (\text{OclAllInstances-generic pre-post OclAny}) \rightarrow \text{forallSet}(X | X .\text{oclIsKindOf}(\text{OclAny}))) \)

lemma OclAny-allInstances-at-post-oclIsKindOf_OclAny:
\( \tau \models (\text{OclAny .allInstances()} \rightarrow \text{forallSet}(X | X .\text{oclIsKindOf}(\text{OclAny}))) \)

lemma OclAny-allInstances-at-pre-oclIsKindOf_OclAny:
\( \tau \models (\text{OclAny .allInstances@pre()} \rightarrow \text{forallSet}(X | X .\text{oclIsKindOf}(\text{OclAny}))) \)

lemma Person-allInstances-generic-oclIsKindOf_OclAny:
\( \tau \models (\text{Person .allInstances-generic pre-post Person}) \rightarrow \text{forallSet}(X | X .\text{oclIsKindOf}(\text{OclAny}))) \)

lemma Person-allInstances-at-post-oclIsKindOf_Person:
\( \tau \models (\text{Person .allInstances()} \rightarrow \text{forallSet}(X | X .\text{oclIsKindOf}(\text{Person}))) \)

lemma Person-allInstances-at-pre-oclIsKindOf_Person:
\( \tau \models (\text{Person .allInstances@pre()} \rightarrow \text{forallSet}(X | X .\text{oclIsKindOf}(\text{Person}))) \)

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 oid_Person_Boss :: oid where oid_Person_Boss = 10

From there on, we can already define an empty state which must contain for oid_Person_Boss the empty relation (encoded as association list, since there are associations with a Sequence-like structure).

definition eval-extract :: (\( \lambda \mathcal{A}, \lambda x : \text{object} \) \) option option) \Rightarrow (oid \Rightarrow (\( \lambda \mathcal{A}, \lambda c : \text{null} \) ) \) val
\Rightarrow (\( \lambda \mathcal{A}, \lambda c : \text{null} \) ) \) val

where eval-extract \( X f = (\lambda \tau . \text{case } X \tau \text{ of} \\
\perp \Rightarrow \text{invalid } \tau \text{ (* exception propagation *)} \\
\perp, \perp \Rightarrow \text{invalid } \tau \text{ (* dereferencing null pointer *)} \\
\perp, \text{obj} \Rightarrow f \text{ (oid-of obj ) } \tau) \)

100
definition \( \text{choose}_2 \cdot \lambda = \text{fst} \)
definition \( \text{choose}_2 \cdot 2 = \text{snd} \)

definition List-flatten = (\lambda l. (\text{foldl}\ ((\lambda \text{acc}. \ (\lambda l. (\text{foldl}\ ((\lambda \text{acc}. \ (\lambda l. \ (\text{cons} \ l \ (\text{acc}))))\ (\text{acc}))))\ ((\text{rev} \ l))))))) \ (\text{Nil} \ ((\text{rev} \ l))))
definition deref-assocs : (\forall \text{state} \times \forall \text{state} \Rightarrow \forall \text{state})
\Rightarrow (\text{oid} \ \text{list} \ \text{list} \Rightarrow \text{oid} \ \text{list} \ \text{oid} \ \text{list})
\Rightarrow \text{oid}
\Rightarrow (\text{oid} \ \text{list} \Rightarrow (\forall \text{state}, f)\text{val})
\Rightarrow \text{oid}
\Rightarrow (\forall \text{state}, f:\text{null})\text{val}

where deref-assocs \pre-post \text{to-from assoc-oid} f oid =
(\lambda \tau. \text{case} \ (\text{assoc-oid} (\pre-post \tau)) \Rightarrow \text{oid} \ \text{oid} \ \text{oid} \ \text{oid} \ \text{oid} \ \text{oid} \ \text{oid} \ \text{oid} \ \text{oid} \ \text{oid})

The pre-post-parameter is configured with \( \text{fst} \) or \( \text{snd} \), the to-from-parameter either with the identity \( \text{id} \) or the following combinators switch:
definition switch2-1 = (\lambda [x,y] => (x,y))
definition switch2-2 = (\lambda [x,y] => (y,x))
definition switch3-1 = (\lambda [x,y,z] => (x,y))
definition switch3-2 = (\lambda [x,y,z] => (x,z))
definition switch3-3 = (\lambda [x,y,z] => (y,x))
definition switch3-4 = (\lambda [x,y,z] => (y,z))
definition switch3-5 = (\lambda [x,y,z] => (z,x))
definition switch3-6 = (\lambda [x,y,z] => (z,y))

definition select-object :: ((\forall \text{state}, b:\text{null})\text{val})
\Rightarrow ((\forall \text{state}, b)\text{val} => (\forall \text{state}, c)\text{val} => (\forall \text{state}, b)\text{val})
\Rightarrow ((\forall \text{state}, b)\text{val} => (\forall \text{state}, d)\text{val})
\Rightarrow (\text{oid} => (\forall \text{state}, e:\text{null})\text{val})
\Rightarrow (\text{oid list})
\Rightarrow (\forall \text{state}, d)\text{val}

where select-object \mt \text{incl} \text{smash deref l} = \text{smash} (\text{foldl}\ \text{incl} \mt \text{map deref l})
(+ \text{smash returns null with} \mt \text{in} \text{input} \text{in this case, object contains null pointer} +)

The continuation \( f \) is usually instantiated with a smashing function which is either the identity \( \text{id} \) or, for \( 0..1 \) cardinalities of associations, the \( \text{OclANY} \)-selector which also handles the \( \text{null} \)-cases appropriately. A standard use-case for this combinators is for example:
term (select-object \mt \text{Set} \text{UML-Set} \text{OclIncluding} \text{OclANY} f \ 1 \ \text{oid} : ((\forall \text{state}, t) => (\forall \text{state}, t))

definition deref-oidPerson :: (\forall \text{state} \times \forall \text{state} \Rightarrow \forall \text{state})
\Rightarrow (\text{typePerson} \Rightarrow (\forall \text{state}, \text{c}::\text{null})\text{val})
\Rightarrow \text{oid}
\Rightarrow (\forall \text{state}, \text{c}::\text{null})\text{val}

where deref-oidPerson \fst-snd f \oid = (\lambda \tau. \text{case} \ (\text{heap} (\fst-snd \tau)) \text{oid} \Rightarrow
\Rightarrow f \text{oid} \ \tau
\Rightarrow \Rightarrow \text{invalid} \ \tau)
definition deref-oid \text{OclAny} :: (\mathbb{A} \text{ state} \times \mathbb{A} \text{ state} \Rightarrow \mathbb{A} \text{ state})
\Rightarrow (\text{type}\text{OclAny} \Rightarrow (\mathbb{A}, \text{c:null})\text{val})
\Rightarrow \text{oid}
\Rightarrow (\mathbb{A}, \text{c:oid})\text{val}

where deref-oid\text{OclAny} \text{fst-snd f oid} = (\lambda \tau. \text{case (heap (\text{fst-snd} \tau)) oid of}
\text{in}\text{OclAny obj} \Rightarrow f \text{ obj} \tau
| - \Rightarrow \text{invalid} \tau)

pointer undefined in state or not referencing a type conform object representation

definition select\text{OclAny} f = (\lambda X. \text{case X of}
\text{mk}\text{OclAny} - \bot \Rightarrow \text{null}
| \text{mk}\text{OclAny} - \text{any} \Rightarrow f (\lambda x - \text{any})
)

definition select\text{Person} f = \text{select-object mSet UML-Set. OclIncluding OclANY} (f (\lambda x - \text{any}))

definition deref-assocs2 \text{fst-snd f oid} = (\lambda \text{mk}\text{Person} \text{oid} - \Rightarrow deref-assocs2 \text{fst-snd switch2-1 oid}\text{Person}\text{fst-snd f oid})

definition in-pre-state = \text{fst}
definition in-post-state = \text{snd}

definition reconst-basetype = (\lambda \text{convert x. convert x})

definition \text{dot}\text{OclAny} f :: \text{OclAny} \Rightarrow - ((1(-).\text{any}) 50)
where \text{(X).any} = \text{eval-extract X}
\text{(deref-oid}\text{OclAny in-post-state}
\text{(select}\text{OclAny}\text{f})
\text{reconst-basetype})

definition \text{dot}\text{Person} f :: \text{Person} \Rightarrow \text{Person} ((1(-).\text{boss}) 50)
where \text{(X).boss} = \text{eval-extract X}
\text{(deref-oid}\text{Person in-post-state}
\text{(deref-assocs2} \text{in-post-state}
\text{(select}\text{Person}\text{f})
\text{reconst-basetype}))

definition \text{dot}\text{Person} f :: \text{Person} \Rightarrow \text{Integer} ((1(-).\text{salary}) 50)
where \text{(X).salary} = \text{eval-extract X}
\text{(deref-oid}\text{Person in-post-state}
\text{(select}\text{Person}\text{f})
\text{reconst-basetype}))

definition \text{dot}\text{OclAny} f :: \text{OclAny} \Rightarrow \text{OclAny} \Rightarrow - ((1(-).\text{any}@\text{pre}) 50)
where \text{(X).any}@\text{pre} = \text{eval-extract X}
\text{(deref-oid}\text{OclAny in-pre-state}
(selectOclAny�.V
  reconst-basetype))

definition dotPerson BOSS-at-pre:: Person ⇒ Person (((λ.-.boss@pre) 50)
where (X).boss@pre = eval-extract X
(deref-oidPerson in-pre-state
 (deref-assocs2 BOSS in-pre-state
 (selectPerson BOSS)
 (deref-oidPerson in-pre-state))))

definition dotPerson / \ of L \ of \ any\at-pre:: Person ⇒ Integer (((λ.-.salary@pre) 50)
where (X).salary@pre = eval-extract X
(deref-oidPerson in-pre-state
 (selectPerson / \ of L \ of \ any\at-pre
 (reconst-basetype)))

lemmas dot-accessor =
dotOclAny�.V-def
dotPerson BOSS-def
dotPerson / \ of L \ of \ any\-at-pre-def
dotOclAny�.V-at-pre-def
dotPerson BOSS-at-pre-def
dotPerson / \ of L \ of \ any\-at-pre-def

Context Passing

lemmas [simp] = eval-extract-def

lemma cp-dotOclAny�.V\: ((X).any) τ = ((λ.-.X τ).any) τ
lemma cp-dotPerson BOSS\: ((X).boss) τ = ((λ.-.X τ).boss) τ
lemma cp-dotPerson / \ of L \ of \ any\-at-pre\: ((X).salary) τ = ((λ.-.X τ).salary) τ

lemma cp-dotOclAny�.V-at-pre\: ((X).any@pre) τ = ((λ.-.X τ).any@pre) τ
lemma cp-dotPerson BOSS-at-pre\: ((X).boss@pre) τ = ((λ.-.X τ).boss@pre) τ
lemma cp-dotPerson / \ of L \ of \ any\-at-pre\: ((X).salary@pre) τ = ((λ.-.X τ).salary@pre) τ

lemmas cp-dotOclAny�.V \-I [simp, intro!]=
cp-dotOclAny�.V\[THEN all][THEN all],
  of λ \ X \ -. X λ - τ. τ, THEN cpII]
lemmas cp-dotOclAny�.V-at-pre-I [simp, intro!]=
cp-dotOclAny�.V-at-pre\[THEN all][THEN all],
  of λ \ X \ -. X λ - τ. τ, THEN cpII]

lemmas cp-dotPerson BOSS \-I [simp, intro!]=
cp-dotPerson BOSS\[THEN all][THEN all],
  of λ \ X \ -. X λ - τ. τ, THEN cpII]
lemmas cp-dotPerson BOSS-at-pre-I [simp, intro!]=
cp-dotPerson BOSS-at-pre\[THEN all][THEN all],
  of λ \ X \ -. X λ - τ. τ, THEN cpII]

lemmas cp-dotPerson / \ of L \ of \ any\-I [simp, intro!]=
cp-dotPerson / \ of L \ of \ any\[THEN all][THEN all],
  of λ \ X \ -. X λ - τ. τ, THEN cpII]
Figure A.3.: (a) pre-state $\sigma_1$ and (b) post-state $\sigma'_1$.

A.7.9. A Little Infrastructure on Example States

The example we are defining in this section comes from the figure A.3.

definition

$\sigma_1 \equiv (\langle \text{heap} = \text{empty}(\text{mkPerson oid0 \_1000)})$
$(\text{oid1} \text{\_inPerson (mkPerson oid1 \_1200)})$
$(\text{oid2})$
$(\text{oid3} \text{\_inPerson (mkPerson oid3 \_2600)})$
$(\text{oid4} \text{\_inPerson person5})$
$(\text{oid5} \text{\_inPerson (mkPerson oid5 \_2300)})$
$(\text{oid6})$
$(\text{oid7})$
(oid8 \rightarrow \text{inPerson}\ person9),
\text{assocs} = \text{empty}(\text{oidPerson}\ \mathcal{O}\ \mathcal{F} \rightarrow [[[\text{oid0}},[[\text{oid1}}],[[[\text{oid3}}],[[[\text{oid4}}],[[[\text{oid5}}],[[[\text{oid6}}],[[[\text{oid3}}]])])

\text{definition} \quad \sigma_1' \equiv \{} \quad \text{heap} = \text{empty}(\text{oid0} \rightarrow \text{inPerson}\ person1)
(\text{oid1} \rightarrow \text{inPerson}\ person2)
(\text{oid2} \rightarrow \text{inPerson}\ person3)
(\text{oid3} \rightarrow \text{inPerson}\ person4)
(*\text{oid4+})
(\text{oid5} \rightarrow \text{inPerson}\ person6)
(\text{oid6} \rightarrow \text{inOclAny}\ person7)
(\text{oid7} \rightarrow \text{inOclAny}\ person8)
(\text{oid8} \rightarrow \text{inPerson}\ person9),
\text{assocs} = \text{empty}(\text{oidPerson}\ \mathcal{O}\ \mathcal{F} \rightarrow [[[\text{oid0}},[[\text{oid1}}],[[[\text{oid1}},[[\text{oid5}}],[[[\text{oid6}},[[\text{oid6}},[[\text{oid6}}]])])

\text{definition} \quad \sigma_0 \equiv \{} \quad \text{heap} = \text{empty}, \text{assocs} = \text{empty} \}

\text{lemma basic-\tau-wff } \forall \text{WFF}(\sigma_1, \sigma_1')

\text{lemma simp.code-unfold } : \text{dom}(\text{heap} \sigma_1) = \{\text{oid0},\text{oid1},(\text{oid2+})\text{oid3},\text{oid4},\text{oid5},(\text{oid6,oid7+}),\text{oid8}\}

\text{Assert } \Delta s_{pre} : (s_{pre,\sigma_1'}) = (X_{\text{Person}}.\text{salary} < 1000)
\text{Assert } \Delta s_{pre} : (s_{pre,\sigma_1'}) = (X_{\text{Person}}.\text{salary} = 1000)
\text{Assert } \Delta s_{post} : (s_{\sigma_1,s_{\sigma_1'}}) = (X_{\text{Person}}.\text{salary}@pre \geq 1000)
\text{Assert } \Delta s_{post} : (s_{\sigma_1,s_{\sigma_1'}}) = (X_{\text{Person}}.\text{salary}@pre \leq 1300)

\text{lemma } (\sigma_1, \sigma_1') \models (X_{\text{Person}}.\text{ocIsMaintained}())

\text{lemma simp.code-unfold } : \text{dom}(\text{heap} \sigma_1') = \{\text{oid0},\text{oid1},\text{oid2},\text{oid3},(\text{oid4+})\text{oid5},\text{oid6},\text{oid7},\text{oid8}\}

\text{Assert } \Delta s_{pre} : (s_{pre,\sigma_1'}) = (X_{\text{Person}}.\text{salary} \geq 1800)
\text{Assert } \Delta s_{post} : (s_{\sigma_1,s_{\sigma_1'}}) = (X_{\text{Person}}.\text{salary}@pre \geq 1200)

\text{lemma } (\sigma_1, \sigma_1') \models (X_{\text{Person}}.\text{ocIsMaintained}())

\text{Assert } \Delta s_{pre} : (s_{pre,\sigma_1'}) = (X_{\text{Person}}.\text{salary} = \text{null})
\text{Assert } \Delta s_{post} : (s_{\sigma_1,s_{\sigma_1'}}) = (\text{null}(X_{\text{Person}}.\text{salary}@pre))

\text{lemma } (\sigma_1, \sigma_1') \models (X_{\text{Person}}.\text{ocIsNew}())

\text{lemma } (\sigma_1, \sigma_1') \models (X_{\text{Person}}.\text{ocIsMaintained}())

\text{Assert } \Delta s_{pre} : (s_{pre,\sigma_1'}) = (\text{not}(X_{\text{Person}}.\text{salary}))
Assert $\land s_{post}$: $(\sigma_1.s_{post}) \models (X_{Person5}.salary@pre \geq 3500)$

lemma $(\sigma_1,\sigma'_1) \models (X_{Person5}.oclIsDeleted())$

lemma $(\sigma_1,\sigma'_1) \models (X_{Person6}.oclIsMaintained())$

Assert $\land s_{pre} s_{post}$: $(s_{pre}.s_{post}) \models u(X_{Person7}.oclAsType(Person))$

lemma $\land s_{pre} s_{post}$: $(s_{pre}.s_{post}) \models (((X_{Person7}.oclAsType(Person).oclAsType(OclAny).oclAsType(Person)) \models (X_{Person7}.oclIsTypeOf(Person)))$

lemma $(\sigma_1,\sigma'_1) \models (X_{Person7}.oclIsNew())$

Assert $\land s_{pre} s_{post}$: $(s_{pre}.s_{post}) \models (X_{Person8} <> X_{Person7})$

Assert $\land s_{pre} s_{post}$: $(s_{pre}.s_{post}) \models not(u(X_{Person8}.oclAsType(Person)))$

Assert $\land s_{pre} s_{post}$: $(s_{pre}.s_{post}) \models (X_{Person8}.oclIsTypeOf(OclAny))$

Assert $\land s_{pre} s_{post}$: $(s_{pre}.s_{post}) \models not(X_{Person8}.oclIsTypeOf(Person))$

Assert $\land s_{pre} s_{post}$: $(s_{pre}.s_{post}) \models not(X_{Person8}.oclIsKindOf(Person))$

Assert $\land s_{pre} s_{post}$: $(s_{pre}.s_{post}) \models (X_{Person8}.oclIsKindOf(OclAny))$

lemma $\sigma$-modifiedonly: $(\sigma_1,\sigma'_1) \models (\text{Set}(X_{Person1}.oclAsType(OclAny), X_{Person2}.oclAsType(OclAny), X_{Person3}.oclAsType(OclAny)*, X_{Person4}.oclAsType(OclAny), X_{Person5}.oclAsType(OclAny)*, X_{Person6}.oclAsType(OclAny), X_{Person7}.oclAsType(OclAny)*, X_{Person8}.oclAsType(OclAny)*, X_{Person9}.oclAsType(OclAny)*) \rightarrow \text{oclIsModifiedOnly}())$

lemma $(\sigma_1,\sigma'_1) \models ((X_{Person9} @ pre (\lambda x. OclAsTypePerson@\lambda x)) \triangleq X_{Person9})$

lemma $(\sigma_1,\sigma'_1) \models ((X_{Person9} @ post (\lambda x. OclAsTypePerson@\lambda x)) \triangleq X_{Person9})$

lemma $(\sigma_1,\sigma'_1) \models (((X_{Person9}.oclAsType(OclAny)) @ pre (\lambda x. OclAsTypeOclAny@\lambda x)) \triangleq (((X_{Person9}.oclAsType(OclAny)) @ post (\lambda x. OclAsTypeOclAny@\lambda x))))$

lemma $\text{perm-}\sigma_3': \sigma'_1 = \{ \text{heap} = \text{empty} \}$

<table>
<thead>
<tr>
<th>oid8</th>
<th>inPerson person9</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid7</td>
<td>inOclAny person8</td>
</tr>
<tr>
<td>oid6</td>
<td>inOclAny person7</td>
</tr>
<tr>
<td>oid5</td>
<td>inPerson person6</td>
</tr>
<tr>
<td>*oid4+</td>
<td></td>
</tr>
<tr>
<td>oid3</td>
<td>inPerson person4</td>
</tr>
<tr>
<td>oid2</td>
<td>inPerson person3</td>
</tr>
<tr>
<td>oid1</td>
<td>inPerson person2</td>
</tr>
<tr>
<td>oid0</td>
<td>inPerson person1</td>
</tr>
</tbody>
</table>
\[ \text{const-ss} \approx \text{assocs} \sigma_1' \]

\[ \text{declare} \]
\[ \text{const-ss} [\text{simp}] \]

\[ \text{lemma} \wedge \sigma_1. \]
\[ (\sigma_1, \sigma_1') = (\text{Person}.\text{allInstances}() \wedge \text{Set}(X_{\text{Person}1}, X_{\text{Person}2}, X_{\text{Person}3}, X_{\text{Person}4}(\ast, X_{\text{Person}5*}), X_{\text{Person}6}, X_{\text{Person}7}.\text{oclAsType(Person)}(\ast, X_{\text{Person}8*})) \]

\[ \text{lemma} \wedge \sigma_1. \]
\[ (\sigma_1, \sigma_1') = (\text{OclAny}.\text{allInstances}() \wedge \text{Set}(X_{\text{OclAny}1}.\text{oclAsType(OclAny)}, X_{\text{OclAny}2}.\text{oclAsType(OclAny)}, X_{\text{OclAny}3}.\text{oclAsType(OclAny)}(\ast, X_{\text{OclAny}5*}), X_{\text{OclAny}6}.\text{oclAsType(OclAny)}, X_{\text{OclAny}7}, X_{\text{OclAny}8}, X_{\text{OclAny}9}.\text{oclAsType(OclAny)})) \]

### 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.

```plaintext
context Person

inv label : self .boss <> null implies (self .salary \leq (self .boss) .salary)

definition Person-label_{inv} :: Person \Rightarrow Boolean
where

Person-label_{inv} (self) \equiv

(self .boss <> null implies (self .salary \leq (self .boss) .salary))

definition Person-label_{invAT pre} :: Person \Rightarrow Boolean
where

Person-label_{invAT pre} (self) \equiv

(self .boss@pre <> null implies (self .salary@pre \leq (self .boss@pre) .salary@pre))

definition Person-label_{global inv} :: Boolean
where

Person-label_{global inv} \equiv (Person .allInstances() \rightarrow \forall x | Person-label_{inv} (x) and
(Person .allInstances@pre() \rightarrow \forall x | Person-label_{invAT pre} (x)))

lemma \[ \tau \models \delta(X .boss) \Rightarrow \tau \models Person .allInstances() \rightarrow include_{Set}(X .boss) \land
\tau \models Person .allInstances() \rightarrow include_{Set}(X) \]

lemma REC-pre : \[ \tau \models Person-label_{global inv} \Rightarrow \tau \models Person .allInstances() \rightarrow include_{Set}(X \ast \text{object represented in state } \ast) \Rightarrow
\exists \text{REC}. \tau \models \text{REC}(X) \triangleq (Person-label_{inv} (X) and (X .boss <> null implies \text{REC}(X .boss))) \]
```

This allows to state a predicate:
**axiomatization** \( \text{inv}_{\text{Person-label}} : \text{Person} \Rightarrow \text{Boolean} \)

where \( \text{inv}_{\text{Person-label}}-\text{def} : \)
\[
(\tau \models \text{Person.allInstances()} \rightarrow \text{includes}_{\text{Set}}(self)) \implies \\
(\tau \models (\text{inv}_{\text{Person-label}})(self) \triangleq (\text{self.boss} <> \text{null} \implies \\
(\text{self.salary} \leq \text{int}((\text{self.boss}.\text{salary})) \text{and} \\
\text{inv}_{\text{Person-label}}(\text{self.boss}))))
\]

**axiomatization** \( \text{inv}_{\text{Person-labelAT pre}} : \text{Person} \Rightarrow \text{Boolean} \)

where \( \text{inv}_{\text{Person-labelAT pre}}-\text{def} : \)
\[
(\tau \models \text{Person.allInstances@pre()} \rightarrow \text{includes}_{\text{Set}}(self)) \implies \\
(\tau \models (\text{inv}_{\text{Person-labelAT pre}})(self) \triangleq (\text{self.boss@pre} <> \text{null} \implies \\
(\text{self.salary@pre} \leq \text{int}((\text{self.boss@pre}.\text{salary@pre})) \text{and} \\
\text{inv}_{\text{Person-labelAT pre}}(\text{self.boss@pre}))))
\]

**lemma** \( \text{inv-1} : \)
\[
(\tau \models \text{Person.allInstances()} \rightarrow \text{includes}_{\text{Set}}(self)) \implies \\
(\tau \models \text{inv}_{\text{Person-label}}(self) = ((\tau \models (\text{self.boss} = \text{null})) \lor \\
(\tau \models (\text{self.boss} <> \text{null}) \land \\
\tau \models ((\text{self.salary} \leq \text{int}((\text{self.boss}.\text{salary}))) \land \\
\tau \models (\text{inv}_{\text{Person-label}}(\text{self.boss})))))
\]

**lemma** \( \text{inv-2} : \)
\[
(\tau \models \text{Person.allInstances@pre()} \rightarrow \text{includes}_{\text{Set}}(self)) \implies \\
(\tau \models \text{inv}_{\text{Person-labelAT pre}}(self) = ((\tau \models (\text{self.boss@pre} = \text{null})) \lor \\
(\tau \models (\text{self.boss@pre} <> \text{null}) \land \\
(\tau \models ((\text{self.boss@pre}.\text{salary}@pre) \leq \text{int}((\text{self.boss@pre}.\text{salary}@pre))) \land \\
(\tau \models (\text{inv}_{\text{Person-labelAT pre}}(\text{self.boss@pre})))))
\]

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** \( \text{inv} : \text{Person} \Rightarrow (\exists \text{st}) \Rightarrow \text{bool} \)

where \( \text{inv}(\delta, self) \Rightarrow ((\tau \models (\text{self.boss} = \text{null})) \lor \\
(\tau \models (\text{self.boss} <> \text{null}) \land (\tau \models (\text{self.boss}.\text{salary} \leq \text{int}(\text{self.salary}))) \land \\
((\text{inv}(\text{self.boss})) \tau)) \\
\Rightarrow (\text{inv} \text{self} \tau)
\)

**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 ⇒ Set-Integer \(((1\cdot).\text{contents}'(\tau)) 50\)

where contents-def:
\[
\text{contents}() = (\lambda \tau. \text{if } \tau \models (\delta \text{self}) \text{ then SOME res.}(\tau \models \text{true}) \land (
\tau \models (\lambda \cdot . \text{res}) \triangleq \text{if } (\text{self}\text{.boss} = \text{null}) \text{ then } (\text{Set}\{\text{self}\text{.salary}\}) \\
\text{else (self}\text{.boss}\text{.contents()} \rightarrow \text{incl}\text{uding}_{\text{Set}}(\text{self}\text{.salary}))} \\
\text{endif}) \\
\text{else invalid } \tau)\]
and cp0-contents:(X .\text{contents}()) \tau = ((\lambda \cdot X \tau) .\text{contents}()) \tau

interpretation contents : contract0 contents \lambda self. true
\[
\lambda self. res \triangleq \text{if } (\text{self}\text{.boss} = \text{null}) \text{ then } (\text{Set}\{\text{self}\text{.salary}\}) \\
\text{else (self}\text{.boss}\text{.contents()} \rightarrow \text{incl}\text{uding}_{\text{Set}}(\text{self}\text{.salary}))} \\
\text{endif}
\]

Specializing \[\text{cp } E; \tau \models \delta \text{self}; \tau \models \text{true}; \tau \models \text{POST}' \text{self}; \lambda res. (\text{res} \triangleq \text{if } \text{self}\text{.boss} = \text{null} \text{ then } \text{Set}\{\text{self}\text{.salary}\}) \\
\text{else (self}\text{.boss}\text{.contents()} \rightarrow \text{incl}\text{uding}_{\text{Set}}(\text{self}\text{.salary}))} = (\text{POST}' \text{self} \text{ and } (\text{res} \triangleq \text{BODY } \text{self})) \Rightarrow (\tau \models E \\text{(self}\text{.contents}()) = (\tau \models E \\text{(BODY self)}),\text{ one gets the following more practical rewrite rule that is amenable to symbolic evaluation:}

theorem unfold-contents :
\[\text{assumes cp } E\]
\[\text{and } \tau \models \delta \text{self}\]
\[\text{shows } (\tau \models E \\text{(self}\text{.contents}()) = \text{if } \text{self}\text{.boss} = \text{null} \text{ then } \text{Set}\{\text{self}\text{.salary}\} \\
\text{else self}\text{.boss}\text{.contents()} \rightarrow \text{incl}\text{uding}_{\text{Set}}(\text{self}\text{.salary}))} \]

Since we have only one interpretation function, we need the corresponding operation on the pre-state:

consts contentsATpre :: Person ⇒ Set-Integer \(((1\cdot).\text{contents}@\text{pre}'(\tau)) 50\)

axiomatization where contentsATpre-def:
\[
\text{contentsATpre}() = (\lambda \tau. \text{if } \tau \models (\delta \text{self}) \text{ then SOME res.}(\tau \models \text{true}) \land (
\tau \models (\lambda \cdot . \text{res}) \triangleq \text{if } (\text{self}\text{.boss} = \text{null}) \text{ then } (\text{Set}\{\text{self}\text{.salary}@\text{pre}\}) \\
\text{else (self}\text{.boss}@\text{pre}\text{.contents}@\text{pre}() \rightarrow \text{incl}\text{uding}_{\text{Set}}(\text{self}\text{.salary}@\text{pre}))} \\
\text{endif}) \\
\text{else invalid } \tau)\]
and cp0-contents-at-pre:(X .\text{contents}@\text{pre}()) \tau = ((\lambda \cdot X \tau) .\text{contents}@\text{pre}()) \tau

interpretation contentsATpre : contract0 contentsATpre \lambda self. true
\[
\lambda self. res \triangleq \text{if } (\text{self}\text{.boss}@\text{pre} = \text{null}) \text{ then } (\text{Set}\{\text{self}\text{.salary}@\text{pre}\}) \\
\text{else (self}\text{.boss}@\text{pre}\text{.contents}@\text{pre}())
\]
Again, we derive via contents.unfold2 a Knaster-Tarski like Fixpoint rule that is amenable to symbolic evaluation:

\[
\text{\textbf{theorem unfold-contentsATpre :}}
\]
\[
\text{assumes cp E}
\]
\[
\text{and } \tau \models \delta \text{ self}
\]
\[
\text{shows } (\tau \models E (\text{self.contents@pre}())) =
\]
\[
(\tau \models E (\text{if self.boss@pre = null then Set{self.salary@pre} \\
else self.boss@pre.contents@pre() ->including}_{\text{self.salary@pre}} endif)}
\]

Note that these \texttt{@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:

<table>
<thead>
<tr>
<th>context</th>
<th>Person::insert(x:Integer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre</td>
<td>true</td>
</tr>
<tr>
<td>post</td>
<td>contents():Set(Integer)</td>
</tr>
<tr>
<td>contents() = contents@pre()-&gt;including(x)</td>
<td></td>
</tr>
</tbody>
</table>

This boils down to:

\[
\text{\textbf{definition insert :: Person $\Rightarrow$ Integer $\Rightarrow$ Void \ ($(1(-).insert(\cdot)) \equiv 50$)}}
\]
\[
\text{where self . insert(x) $\equiv$}
\]
\[
(\lambda \tau. \text{if } (\tau \models (\delta \text{ self})) \land (\tau \models \nu x) \land \\
\text{then SOME res. } (\tau \models \text{true} \land \\
(\tau \models ((\text{self.contents()} \triangleq (\text{self.contents@pre}()->including}_{\text{self}}(x)))) \\
\text{else invalid } \tau)
\]

The semantic consequences of this definition were computed inside this locale interpretation:

\[
\text{\textbf{interpretation insert : contract1 insert } \lambda \text{ self x. true}}
\]
\[
\lambda \text{ self x res. ((self.contents()) \triangleq (self.contents@pre()->including}_{\text{self}}(x)))}
\]

The result of this locale interpretation for our \textit{Analysis-OCL.insert} contract is the following set of properties, which serves as basis for automated deduction on them:
<table>
<thead>
<tr>
<th>Name</th>
<th>Theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert.strict0</td>
<td>(invalid.insert(X)) = invalid</td>
</tr>
<tr>
<td>insert.nullstrict0</td>
<td>(null.insert(X)) = invalid</td>
</tr>
<tr>
<td>insert.strict1</td>
<td>(self.insert(invalid)) = invalid</td>
</tr>
<tr>
<td>insert.cpPRE</td>
<td>true ( \tau = \text{true} \tau )</td>
</tr>
<tr>
<td>insert.cpPOST</td>
<td>(self.contents() ( \triangleq ) self.contents@pre() ( \rightarrow ) ( \text{including}<em>{\text{Sel}}(a1.0) )) ( \tau = (\lambda \cdot \tau \cdot \text{contents()} \triangleq \lambda \cdot \tau \cdot \text{contents}@pre() ( \rightarrow ) ( \text{including}</em>{\text{Sel}}(\lambda \cdot a1.0 \tau) )) ( \tau )</td>
</tr>
<tr>
<td>insert.cp-pre</td>
<td>( [\text{cp self}^0; \text{cp a1}^0; \text{cp res}] \Rightarrow \text{cp} (\lambda X. \text{self}^0 X.\text{contents()} \triangleq \text{self}^0 X.\text{contents}@pre() ( \rightarrow ) ( \text{including}_{\text{Sel}}(a1 \lambda') )) ( \tau )</td>
</tr>
<tr>
<td>insert.cp-post</td>
<td>( [\text{cp self}^0; \text{cp a1}^0; \text{cp res}] \Rightarrow \text{cp} (\lambda X. \text{self}^0 X.\text{insert}(a1'X)) )</td>
</tr>
<tr>
<td>insert.cp0</td>
<td>(self.insert(a1.0)) ( \tau = (\lambda \cdot \text{self} \cdot \text{insert}(\lambda \cdot a1.0 \tau) ) ( \tau )</td>
</tr>
<tr>
<td>insert.def-scheme</td>
<td>self.insert(a1.0) ( \equiv \lambda \tau. \text{if } \tau \models \delta \text{ self } \land \tau \models \nu \text{ a1.0 then SOME res. } \tau \models \text{true } \land \tau \models ) self.contents() ( \triangleq ) self.contents@pre() ( \rightarrow ) ( \text{including}_{\text{Sel}}(a1.0) ) else invalid ( \tau )</td>
</tr>
<tr>
<td>insert.unfold</td>
<td>( [\text{cp E}; \tau \models \delta \text{ self } \land \tau \models \nu \text{ a1.0}; \tau \models \text{true}; \exists \text{res. } \tau \models \text{self.contents()} ) ( \triangleq ) self.contents@pre() ( \rightarrow ) ( \text{including}<em>{\text{Sel}}(a1.0) ); ( \tau \models \text{self.contents()} ) ( \triangleq ) self.contents@pre() ( \rightarrow ) ( \text{including}</em>{\text{Sel}}(a1.0) ) ( \Rightarrow \tau \models E (\lambda \cdot \text{res}) ) ( \Rightarrow \tau \models E (\text{self.insert}(a1.0)) )</td>
</tr>
<tr>
<td>insert.unfold2</td>
<td>( [\text{cp E}; \tau \models \delta \text{ self } \land \tau \models \nu \text{ a1.0}; \tau \models \text{true}; \tau \models \text{POST}^\prime \text{ self a1.0}; \exists \text{res. } \tau \models \text{self.contents()} ) ( \triangleq ) self.contents@pre() ( \rightarrow ) ( \text{including}_{\text{Sel}}(a1.0) ) ( \Rightarrow \tau \models (\text{POST}^\prime \text{ self a1.0 and res } \triangleq \text{BODY self a1.0}) ) ( \Rightarrow (\tau \models E (\text{self.insert}(a1.0))) = (\tau \models E (\text{BODY self a1.0})) )</td>
</tr>
</tbody>
</table>

Table A.5: Semantic properties resulting from a user-defined operation contract.
Bibliography


# Contents

## A. Formal Semantics of OCL

A.1. Introduction .................................................. 3
A.2. Background ..................................................... 5
  A.2.1. Formal Foundation ......................................... 5
    Isabelle ......................................................... 5
    Higher-order Logic (HOL) ....................................... 6
  A.2.2. How this Annex A was Generated from Isabelle/HOL Theories ....... 8
A.3. The Essence of UML-OCL Semantics .............................. 9
  A.3.1. The Theory Organization .................................. 9
  A.3.2. Denotational Semantics of Constants and Operations ............... 10
  A.3.3. Logical Layer ............................................ 11
    Algebraic Layer .............................................. 13
  A.3.4. Object-oriented Datatype Theories .......................... 15
    A Denotational Space for Class-Models: Object Universes ............. 16
    Denotational Semantics of Accessors on Objects and Associations .... 17
    Logic Properties of Class-Models ................................ 19
    Algebraic Properties of the Class-Models ........................ 20
    Other Operations on States .................................... 20
  A.3.5. Data Invariants ........................................... 21
  A.3.6. Operation Contracts ...................................... 21
A.4. Formalization I: OCL Types and Core Definitions ............... 23
  A.4.1. Preliminaries ............................................. 23
    Notations for the Option Type ................................ 23
    Common Infrastructure for all OCL Types ........................ 23
    Accommodation of Basic Types to the Abstract Interface ............. 24
    The Common Infrastructure of Object Types (Class Types) and States .. 25
    Common Infrastructure for all OCL Types (II): Valuations as OCL Types .... 26
    The fundamental constants 'invalid' and 'null' in all OCL Types ....... 26
  A.4.2. Basic OCL Value Types .................................... 26
  A.4.3. Some OCL Collection Types ................................ 27
    The Construction of the Pair Type (Tuples) ........................ 27
    The Construction of the Set Type ................................ 28
    The Construction of the Sequence Type ............................ 28
    Discussion: The Representation of UML/OCL Types in Featherweight OCL .... 29
A.5. Formalization II: OCL Terms and Library Operations ............. 29
  A.5.1. The Operations of the Boolean Type and the OCL Logic .......... 29
    Basic Constants ............................................... 29
    Validity and Definedness ...................................... 30
    The Equalities of OCL ......................................... 31
    Logical Connectives and their Universal Properties ................... 33
    A Standard Logical Calculus for OCL ................................ 36
    OCL's if then else endif ...................................... 41
    Fundamental Predicates on Basic Types: Strict (Referential) Equality .... 41