

A Formal Machine-Checked Semantics for OCL 2.5

A Proposal for the "Annex A" of the OCL Standard

Achim D. Brucker* Frédéric Tuong^{†‡} Burkhardt Wolff^{‡‡}

February 6, 2026

*SAP SE

Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
achim.brucker@sap.com

[†]LRI, Univ. Paris-Sud, CNRS, CentraleSupélec, Université Paris-Saclay
bât. 650 Ada Lovelace, 91405 Orsay, France
frederic.tuong@lri.fr burkhart.wolff@lri.fr

^{‡‡}IRT SystemX

8 av. de la Vauve, 91120 Palaiseau, France
frederic.tuong@irt-systemx.fr burkhart.wolff@irt-systemx.fr

Annex A.

Formal Semantics of OCL

0.1. 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 OCL¹. Featherweight OCL has a formal semantics in Isabelle/HOL [17].

The semantic definitions are in large parts executable, in some parts only provable, namely the essence of Set- and Bag- 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.

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

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

and:

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

which then results in the absurd logical consequence that `invalid = true`. Obviously, we need to be more careful on the side-conditions of our rules². And obviously, only a mechanized check of these definitions, following a rigorous methodology, can establish strong guarantees for logical consistency of the OCL language.

This leads us to our second goal of this annex: it should not only be usable by logicians, but also by developers of compilers and proof-tools. For this end, we *derived* from the Isabelle definitions also *logical rules* allowing formal interactive and automated proofs on UML/OCL specifications, as well as *execution rules* and *test-cases* revealing corner-cases resulting from this semantics which give vital information for the implementor.

OCL is an annotation language for UML models, in particular class models allowing for specifying data and operations on them. As such, it is a *typed* object-oriented language. This means that it is—like Java or C++—based on the concept of a *static type*, that is the type that the type-checker infers from a UML class model and its OCL annotation, as well as a *dynamic type*, that is the type at which an object is dynamically created³. Types are not only a means for efficient compilation and a support of separation of concerns in programming, there are of fundamental importance for our goal of logical consistency: it is impossible to have sets that contain themselves, i. e., to state Russels Paradox in OCL typed set-theory. Moreover, object-oriented typing means that types there can be in sub-typing relation; technically speaking, this means that they can be *cast* via `oclIsTypeOf(T)` one to the other, and under particular conditions to be described in detail later, these casts are semantically *lossless*, i. e.,

$$(X.\text{oclAsType}(C_j).\text{oclAsType}(C_i) = X) \quad (0.1)$$

¹An updated, machine-checked version and formally complete version of the complete formalization is maintained by the Isabelle Archive of Formal Proofs (AFP), see http://isa-afp.org/entries/Featherweight_OCL.shtml

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

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

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

Here is a feature-list of Featherweight OCL:

- it specifies key built-in types such as `Boolean`, `Void`, `Integer`, `Real` and `String` as well as generic types such as `Pair (T, T')`, `Sequence (T)` and `Set (T)`.
- it defines the semantics of the operations of these types in *denotational form*—see explanation below—, and thus in an unambiguous (and in Isabelle/HOL executable or animatable) way.
- it develops the *theory* of these definitions, i. e., the collection of lemmas and theorems that can be proven from these definitions.
- all types in Featherweight OCL contain the elements `null` and `invalid`; since this extends to `Boolean` type, this results in a four-valued logic. Consequently, Featherweight OCL contains the derivation of the *logic* of OCL.
- collection types may contain `null` (so `Set {null}` is a defined set) but not `invalid (Set {invalid})` is just `invalid`.
- Wrt. to the static types, Featherweight OCL is a strongly typed language in the Hindley-Milner tradition. We assume that a pre-process for full OCL eliminates all implicit conversions due to subtyping by introducing explicit casts (e. g., `oclAsType (Class)`).⁴
- Featherweight OCL types may be arbitrarily nested. For example, the expression `Set {Set {1, 2}} = Set {Set {2, 1}}` is legal and true.
- Featherweight OCL types may be higher-order nested. For example, the expression `\<lambda> X. Set {X} = Set {Set {2, 1}}` is legal. Higher-order pattern-matching can be easily extended following the principles in the HOL library, which can be applied also to Featherweight OCL types.
- All objects types are represented in an object universe⁵. The universe construction also gives semantics to type casts, dynamic type tests, as well as functions such as `allInstances ()`, or `oclIsNew ()`. The object universe construction is conceptually described and demonstrated at an example.
- As part of the OCL logic, Featherweight OCL develops the theory of equality in UML/OCL. This includes the standard equality, which is a computable strict equality using the object references for comparison, and the not necessarily computable logical equality, which expresses the Leibniz principle that ‘equals may be replaced by equals’ in OCL terms.
- Technically, Featherweight OCL is a *semantic embedding* into a powerful semantic meta-language and environment, namely Isabelle/HOL [17]. It is a so-called *shallow embedding* in HOL; this means that types in OCL were mapped one-to-one to types in Isabelle/HOL. Ill-typed OCL specifications can therefore not be represented in Featherweight OCL and a type in Featherweight OCL contains exactly the values that are possible in OCL.

⁴The details of such a pre-processing are described in [3].

⁵following the tradition of HOL-OCL [5]

Context. This document stands in a more than fifteen years tradition of giving a formal semantics to the core of UML and its annotation language OCL, starting from Richters [20] and [11, 14, 16], 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 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.

0.2. Background

0.2.1. Formal Foundation

0.2.1.1. A Gentle Introduction to Isabelle

Isabelle [17] 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).

The core language of Isabelle is a typed λ -calculus providing a uniform term language T in which all logical entities where represented:⁶

$$T ::= C \mid V \mid \lambda V. T \mid T T$$

where:

- C is the set of *constant symbols* like "fst" or "snd" as operators on pairs. Note that Isabelle's syntax engine supports mixfix-notation for terms: " $(_ \implies _) A B$ " or " $(_ + _) A B$ " can be parsed and printed as " $A \implies B$ " or " $A + B$ ", respectively.

⁶In the Isabelle implementation, there are actually two further variants which were irrelevant for this presentation and are therefore omitted.

- V is the set of *variable symbols* like "x", "y", "z", ... Variables standing in the scope of a λ -operator were called *bound* variables, all others are *free* variables.
- " $\lambda V. T$ " is called λ -abstraction. For example, consider the identity function $\lambda x.x$. A λ -abstraction forms a scope for the variable V .
- $T T'$ is called an *application*.

These concepts are not at all Isabelle specific and can be found in many modern programming languages ranging from Haskell over Python to Java.

Terms were associated to *types* by a set of *type inference rules*⁷; only terms for which a type can be inferred—i. e., for *typed terms*—were considered as legal input to the Isabelle system. The type-terms τ for λ -terms are defined as:⁸

$$\tau ::= TV \mid TV :: \Xi \mid \tau \Rightarrow \tau \mid (\tau, \dots, \tau) TC \quad (0.2)$$

- TV is the set of *type variables* like $'\alpha, '\beta, \dots$. The syntactic categories V and TV are disjoint; thus, $'x$ is a perfectly possible type variable.
- Ξ is a set of *type-classes* like *ord, order, linorder, ...*. This feature in the Isabelle type system is inspired by Haskell type classes.⁹ A *type class constraint* such as " $\alpha :: \text{order}$ " expresses that the type variable $'\alpha$ may range over any type that has the algebraic structure of a partial ordering (as it is configured in the Isabelle/HOL library).
- The type $'\alpha \Rightarrow '\beta$ denotes the total function space from $'\alpha$ to $'\beta$.
- TC is a set of *type constructors* like $(''\alpha)\text{list}$ or $(''\alpha)\text{tree}$. Again, Isabelle's syntax engine supports mixfix-notation for type terms: cartesian products $'\alpha \times '\beta$ or type sums $'\alpha + '\beta$ are notations for $(''\alpha, '\beta)(_ < \text{times} > _)$ or $(''\alpha, '\beta)(_ + _)$, respectively. Also null-ary type-constructors like $()\text{bool}, ()\text{nat}$ and $()\text{int}$ are possible; note that the parentheses of null-ary type constructors are usually omitted.

Isabelle accepts also the notation $t :: \tau$ as type assertion in the term-language; $t :: \tau$ means "t is required to have type τ ". Note that typed terms *can* contain free variables; terms like $x + y = y + x$ reflecting common mathematical notation (and the convention that free variables are implicitly universally quantified) are possible and common in Isabelle theories.¹⁰ An environment providing Ξ, TC as well as a map from constant symbols C to types (built over these Ξ and TC) is called a *global context*; it provides a kind of signature, i. e., a mechanism to construct the syntactic material of a logical theory.

The most basic (built-in) global context of Isabelle provides just a language to construct logical rules. More concretely, it provides a constant declaration for the (built-in) *meta-level implication* $\Longrightarrow$ allowing to form constructs like $A_1 \Longrightarrow \dots \Longrightarrow A_n \Longrightarrow A_{n+1}$, which are viewed as a *rule* of the form "from assumptions A_1 to A_n , infer conclusion A_{n+1} " and which is written in Isabelle syntax as

$$\llbracket A_1; \dots; A_n \rrbracket \Longrightarrow A_{n+1} \quad \text{or, in mathematical notation,} \quad \frac{A_1 \quad \dots \quad A_n}{A_{n+1}}. \quad (0.3)$$

Moreover, the built-in meta-level quantification $\text{Forall}(\lambda x. E x)$ (pretty-printed and parsed as $\bigwedge x. E x$) captures the usual side-constraints "x must not occur free in the assumptions" for quantifier rules; meta-quantified variables can be considered as "fresh" free variables. Meta-level quantification leads to a generalization of Horn-clauses of the form:

$$\bigwedge x_1, \dots, x_m. \llbracket A_1; \dots; A_n \rrbracket \Longrightarrow A_{n+1}. \quad (0.4)$$

⁷Similar to https://en.wikipedia.org/w/index.php?title=Hindley%E2%80%93Milner_type_system&oldid=668548458

⁸Again, the Isabelle implementation is actually slightly different; our presentation is an abstraction in order to improve readability.

⁹See https://en.wikipedia.org/w/index.php?title=Type_class&oldid=672053941.

¹⁰Here, we assume that $_ + _$ and $_ = _$ are declared constant symbols having type $\text{int} \Rightarrow \text{int} \Rightarrow \text{int}$ and $'\alpha \Rightarrow '\beta \Rightarrow \text{bool}$, respectively.

Isabelle supports forward- and backward reasoning on rules. For backward-reasoning, a *proof-state* can be initialized in a given global context and further transformed into others. For example, a proof of ϕ , using the Isar [22] language, will look as follows in Isabelle:

```
lemma label:   $\phi$ 
  apply(case_tac)
  apply(simp_all)
done
```

(0.5)

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

```
label :   $\phi$ 
  1.   $\phi_1$ 
     $\vdots$ 
  n.   $\phi_n$ 
```

(0.6)

Subgoals and goals may be extracted from the proof state into theorems of the form $[\phi_1; \dots; \phi_n] \implies \phi$ at any time; By extensions of global contexts with axioms and proofs of theorems, *theories* can be constructed step by step. Beyond the basic mechanisms to extend a global context by a type-constructor-, type-class- constant-definition or an axiom, Isabelle offers a number of *commands* that allow for more complex extensions of theories in a logically safe way (avoiding the use of axioms directly).

0.2.1.2. Higher-order Logic (HOL)

Higher-order logic (HOL) [1, 10] is a classical logic based on a simple type system. Isabelle/HOL is a theory extension of the basic Isabelle core-language with operators and the 7 axioms of HOL; together with large libraries this constitutes an implementation of HOL. Isabelle/HOL provides the usual logical connectives like $_ \wedge _, _ \rightarrow _, \neg _$ as well as the object-logical quantifiers $\forall x. P x$ and $\exists x. P x$; in contrast to first-order logic, quantifiers may range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centered around extensional equality $_ = _ :: \alpha \Rightarrow \alpha \Rightarrow \text{bool}$. Extensional equality means that two functions f and g are equal if and only if they are point-wise equal; this is captured by the rule: $(\bigwedge x. f x = g x) \implies f = g$. HOL is more expressive than first-order logic, since, among many other things, induction schemes can be expressed inside the logic. For example, the standard induction rule on natural numbers in HOL:

$$P\ 0 \implies (\bigwedge x. P\ x \implies P\ (x+1)) \implies P\ x$$

is just an ordinary rule in Isabelle which is in fact a proven theorem in the theory of natural numbers. This example exemplifies an important design principle of Isabelle: theorems and rules are technically the same, paving the way to *derived rules* and automated decision procedures based on them. This has the consequence that these procedures are consequently sound by construction with respect to their logical aspects (they may be incomplete or failing, though). On the other hand, Isabelle/HOL can also be viewed as a functional programming language like SML or Haskell. Isabelle/HOL definitions can usually be read just as another functional **programming** language; if not interested in proofs and the possibilities of a **specification** language providing powerful logical quantifiers or equivalent free variables, the reader can just ignore these aspects in theories.

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 *well founded recursive definitions*.

For instance, the library includes the type constructor $\tau_{\perp} := \perp \mid \lfloor _ \rfloor : \alpha$ that assigns to each type τ a type $\tau_{\perp}$ *disjointly extended* by the exceptional element $\perp$. The function $\lceil _ \rceil : \alpha_{\perp} \rightarrow \alpha$ is the inverse of $\lfloor _ \rfloor$ (unspecified for $\perp$). Partial functions $\alpha \rightarrow \beta$ are defined as functions $\alpha \Rightarrow \beta_{\perp}$ supporting the usual concepts of domain ($\text{dom } _$) and range ($\text{ran } _$). As another example of a conservative extension, typed sets were built in the Isabelle libraries conservatively on top of the kernel of HOL as functions to bool; consequently, the constant definitions for membership is as follows:¹¹

$$\begin{array}{llll}
\text{types} & \alpha \text{ set} & = \alpha \Rightarrow \text{bool} & \\
\text{definition} & \text{Collect} & :: (\alpha \Rightarrow \text{bool}) \Rightarrow \alpha \text{ set} & \text{--- set comprehension} \\
\text{where} & \text{Collect } S & \equiv S & \\
\text{definition} & \text{member} & :: \alpha \Rightarrow \alpha \Rightarrow \text{bool} & \text{--- membership test} \\
\text{where} & \text{member } s \ S & \equiv Ss &
\end{array} \tag{0.7}$$

Isabelle's syntax engine is instructed to accept the notation $\{x \mid P\}$ for $\text{Collect } \lambda x. P$ and the notation $s \in S$ for $\text{member } s \ S$. As can be inferred from the example, constant definitions are axioms that introduce a fresh constant symbol by some non-recursive expressions not containing free variables; this type of axiom is logically safe since it works like an abbreviation. The syntactic side conditions of this axiom are mechanically checked. It is straightforward to express the usual operations on sets like $_ \cup _, _ \cap _ :: \alpha \text{ set} \Rightarrow \alpha \text{ set} \Rightarrow \alpha \text{ set}$ as conservative extensions, too, while the rules of typed set theory were derived by proofs from these definitions.

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

$$\begin{array}{ll}
\text{datatype} & \text{option} = \text{None} \mid \text{Some } \alpha \\
\text{datatype} & \alpha \text{ list} = \text{Nil} \mid \text{Cons } a \ l
\end{array} \tag{0.8}$$

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

$$\text{case } x \text{ of } \text{None} \Rightarrow F \mid \text{Some } a \Rightarrow G a \tag{0.9}$$

respectively

$$\text{case } x \text{ of } [] \Rightarrow F \mid \text{Cons } a \ r \Rightarrow G a \ r. \tag{0.10}$$

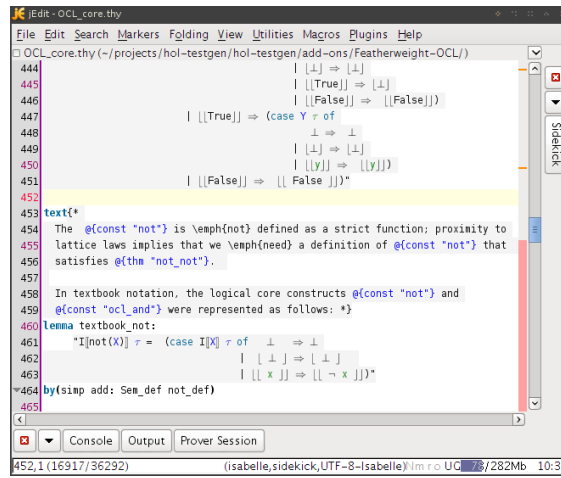
From the internal definitions (not shown here) several properties were automatically derived. We show only the case for lists:

$$\begin{array}{ll}
(\text{case } [] \text{ of } [] \Rightarrow F \mid (a\#r) \Rightarrow G a \ r) = F & \\
(\text{case } b\#t \text{ of } [] \Rightarrow F \mid (a\#r) \Rightarrow G a \ r) = G b \ t & \\
[] \neq a\#t & \text{--- distinctness} \\
\llbracket a = [] \rightarrow P; \exists x \ t. a = x\#t \rightarrow P \rrbracket \Longrightarrow P & \text{--- exhaust} \\
\llbracket P []; \forall a \ t. P t \rightarrow P(a\#t) \rrbracket \Longrightarrow P x & \text{--- induct}
\end{array} \tag{0.11}$$

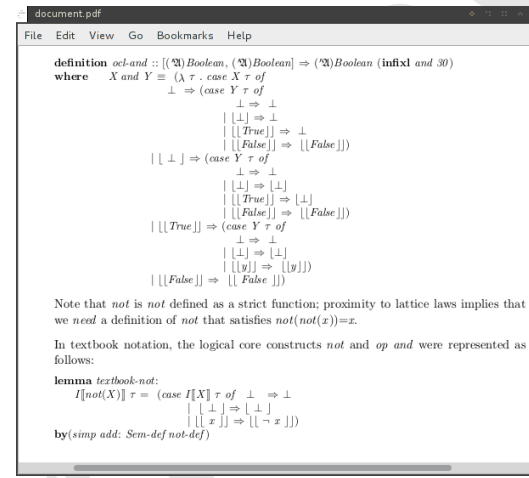
Finally, there is a compiler for primitive and well founded recursive function definitions. For example, we may define the sort operation on linearly ordered lists by:

$$\begin{array}{lll}
\text{fun} & \text{ins} & :: [\alpha :: \text{linorder}, \alpha \text{ list}] \Rightarrow \alpha \text{ list} \\
\text{where} & \text{ins } x \ [] & = [x] \\
& \text{ins } x \ (y\#ys) & = \text{if } x < y \text{ then } x\#y\#ys \text{ else } y\#(\text{ins } x \ ys)
\end{array} \tag{0.12}$$

¹¹To increase readability, we use a slightly simplified presentation.



(a) The Isabelle/jEdit environment.



(b) The generated formal document.

Figure 1. – Generating documents with guaranteed syntactical and semantical consistency.

$$\begin{array}{lll}
 \text{fun} & \text{sort} & :: (\alpha :: \text{linorder}) \text{ list} \Rightarrow \alpha \text{ list} \\
 \text{where} & \text{sort } [] & = [] \\
 & \text{sort } (x\#xs) & = \text{ins } x (\text{sort } xs)
 \end{array} \tag{0.13}$$

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, orderings, and various arithmetic theories which only contain rules derived from conservative definitions. This library constitutes a comfortable basis for defining the OCL library and language constructs.

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). This forms the basis that many OCL terms can be executed directly. Using the `value` command, it is possible to compile many OCL ground expressions (no free variables) to code and to execute them; for example value `"3 + 7"` just answers with 10 in Isabelle's output window. This is even true for many expressions containing types which in themselves are not executable. For example, the `Set` type, which is defined in Featherweight OCL as the type of potentially infinite sets, is consequently not in itself executable; however, due to special setups of the code-generator, expressions like value `"Set {1,2}"` are, because the underlying constructors in this expression allow for automatically establishing that this set is finite and reducible to constructs that *are* in this special case executable.

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

Isabelle, as a framework for building formal tools [21], 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 $\text{\LaTeX}$ -based markup language within

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

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

0.3. The Essence of UML-OCL Semantics

0.3.1. The Theory Organization

The semantic theory is organized in a quite conventional manner in three layers. The first layer, called the *denotational semantics* comprises a set of definitions of the operators of the language. Presented as *definitional axioms* inside Isabelle/HOL, this part assures the logical consistency of the overall construction. The denotational definitions of types, constants and operations, and OCL contracts represent the “gold standard” of the semantics. The second layer, called *logical layer*, is derived from the former and centered around the notion of validity of an OCL formula P . For a state-transition from pre-state σ to post-state σ' , a validity statement is 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.

0.3.2. Denotational Semantics of Types

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

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

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 $\perp$) and a null element; this is enforced in Isabelle by a type-class `null` that contains two distinguishable elements `bot` and `null` (see Chapter 1 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:

$$\text{type_synonym} \quad \text{Boolean}_{\text{base}} := \text{bool}_{\perp\perp} \quad (0.14)$$

As a consequence of this definition of the type, we have the elements $\perp, \perp_{\perp}, \perp\text{true}_{\perp}, \perp\text{false}_{\perp}$ in the carrier-set of $\text{Boolean}_{\text{base}}$. We can therefore use the element $\perp$ to define the generic type class element $\perp$ and $\perp_{\perp}$ for the generic type class null. For collection types and object types this definition is more evolved (see Chapter 1).

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

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

$$\text{type_synonym} \quad \text{V}_{\mathfrak{A}}(\alpha) := \text{state}(\mathfrak{A}) \times \text{state}(\mathfrak{A}) \rightarrow \alpha :: \text{null} . \quad (0.15)$$

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

$$\begin{aligned} \text{type_synonym} \quad \text{Boolean}_{\mathfrak{A}} &:= \text{V}_{\mathfrak{A}}(\text{Boolean}_{\text{base}}) \\ \text{type_synonym} \quad \text{Integer}_{\mathfrak{A}} &:= \text{V}_{\mathfrak{A}}(\text{Integer}_{\text{base}}) \\ \dots \end{aligned}$$

the other cases are analogous. In the subsequent subsections, we will drop the index $\mathfrak{A}$ since it is constant in all formulas and expressions except for operations related to the object universe construction in Section 3.1

The rules of the logical layer (there are no algebraic rules related to the semantics of types), and more details can be found in Chapter 1.

0.3.3. Denotational Semantics of Constants and Operations

We use the notation $I\llbracket E \rrbracket \tau$ for the semantic interpretation function as commonly used in mathematical textbooks and the variable τ standing for pairs of pre- and post state (σ, σ') . Note that we will also use τ to denote the *type* of a state-pair; since both syntactic categories are independent, we can do so without arising confusion. OCL provides for all OCL types the constants *invalid* for the exceptional computation result and *null* for the non-existing value. Thus we define:

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

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

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

For reasons of conciseness, we will write δX for $\text{not}(X.\text{oclIsUndefined}())$ and νX for $\text{not}(X.\text{oclIsValid}())$ throughout this document.

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

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

we can therefore write:

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

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

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

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

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

Previous versions of the OCL semantics [18, Annex A] used different interpretation functions for invariants and pre-conditions; we achieve their semantic effect by a syntactic transformation $_ \text{pre}$ which replaces, for example, all accessor functions $_.a$ by their counterparts $_.a@ \text{pre}$ (see Section 0.3.7). For example, $(\text{self}.a > 5)_{\text{pre}}$ is just $(\text{self}.a@ \text{pre} > 5)$. This way, also invariants and pre-conditions can be interpreted by the same interpretation function and have the same type of an evaluation $V(\alpha)$.

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

$$\begin{aligned} I[\![\text{not } X]\!] \tau &= (\text{case } I[\![X]\!] \tau \text{ of} \\ &\quad \perp \Rightarrow \perp \\ &\quad I_{\perp} \perp \Rightarrow \perp \\ &\quad I_{\perp} X \Rightarrow \perp \neg X) \\ \\ I[\![X \text{ and } Y]\!] \tau &= (\text{case } I[\![X]\!] \tau \text{ of} \\ &\quad \perp \Rightarrow (\text{case } I[\![Y]\!] \tau \text{ of} \\ &\quad \quad \perp \Rightarrow \perp \\ &\quad \quad I_{\perp} \perp \Rightarrow \perp \\ &\quad \quad I_{\perp} \text{true} \Rightarrow \perp \\ &\quad \quad I_{\perp} \text{false} \Rightarrow \perp \text{false}) \\ &\quad I_{\perp} \perp \Rightarrow (\text{case } I[\![Y]\!] \tau \text{ of} \\ &\quad \quad \perp \Rightarrow \perp \\ &\quad \quad I_{\perp} \perp \Rightarrow \perp \\ &\quad \quad I_{\perp} \text{true} \Rightarrow \perp \\ &\quad \quad I_{\perp} \text{false} \Rightarrow \perp \text{false}) \\ &\quad I_{\perp} \text{true} \Rightarrow (\text{case } I[\![Y]\!] \tau \text{ of} \\ &\quad \quad \perp \Rightarrow \perp \\ &\quad \quad I_{\perp} \perp \Rightarrow \perp \\ &\quad \quad I_{\perp} Y \Rightarrow Y) \\ &\quad I_{\perp} \text{false} \Rightarrow \perp \text{false}) \end{aligned}$$

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

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:

$$\begin{aligned} I[x + y] \tau = & \text{ if } I[\delta x] \tau = I[\text{true}] \tau \wedge I[\delta y] \tau = I[\text{true}] \tau \\ & \text{ then } \perp^{\top} I[x] \tau^{\top} + {}^{\top} I[y] \tau^{\top} \perp \\ & \text{ else } \perp \end{aligned}$$

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

0.3.4. Logical Layer

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

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

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

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

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

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

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

The mandatory part of the OCL standard refers to an equality (written $x = y$ or $x \neq y$ 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,”¹² and is at the heart of the OCL logic,

¹²Strong logical equality is also referred as “Leibniz”-equality.

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

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

$$I[X \triangleq Y]\tau \equiv _ I[X]\tau = I[Y]\tau _$$

It enjoys nearly the laws of a congruence:

$$\tau \models (x \triangleq x)$$

$$\tau \models (x \triangleq y) \implies \tau \models (y \triangleq x)$$

$$\tau \models (x \triangleq y) \implies \tau \models (y \triangleq z) \implies \tau \models (x \triangleq z)$$

$$\text{cp}P \implies \tau \models (x \triangleq y) \implies \tau \models (Px) \implies \tau \models (Py)$$

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 2.1.3.

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

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

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

$$(\tau \models A \triangleq \text{true}) = (\tau \models A) \quad (\tau \models A \triangleq \text{false}) = (\tau \models \text{not}A)$$

$$(\tau \models \text{not}(\delta x)) = (\neg \tau \models \delta x) \quad (\tau \models \text{not}(vx)) = (\neg \tau \models vx)$$

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 [12]. δ -closure rules for all logical connectives have the following format, e. g.:

$$\tau \models \delta x \implies (\tau \models \text{not} x) = (\neg(\tau \models x))$$

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

$$\tau \models \delta x \implies \tau \models \delta y$$

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

Together with the already mentioned general case-distinction

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

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

0.3.5. Algebraic Layer

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

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

```

v invalid = false      v null = true
v true = true          v false = true
δ invalid = false      δ null = false
δ true = true          δ false = true

not invalid = invalid   not null = null
not true = false        not false = true

(null and true) = null   (null and false) = false
(null and null) = null   (null and invalid) = invalid
(false and true) = false (false and false) = false
(false and null) = false (false and invalid) = false
(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 and X = X      X and Y = Y and X
false and X = false   X and false = false
true and X = X      X and true = X
X and (Y and Z) = X and Y and Z

```

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

The above equations explain the behavior for the most-important non-strict operations. The clarification of the exceptional behaviors is of key-importance for a semantic definition of the standard and the major deviation point from HOL-OCL [4, 6] to Featherweight OCL as presented here. Expressed in algebraic equations, “strictness-principles” boil down to:

```

invalid + X = invalid      X + invalid = invalid
invalid->including(X) = invalid   null->including(X) = invalid
X ÷ invalid = invalid      invalid ÷ X = invalid
S->including(invalid) = invalid
X ÷ X = (if v x then true else invalid endif)
1 / 0 = invalid           1 / null = invalid
invalid->isEmpty() = invalid   null->isEmpty() = null

```

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

```

δ Set{} = true

```

```

 $\delta(X \rightarrow \text{including}(x)) = \delta X$  and  $v x$ 
Set{}  $\rightarrow$  includes( $x$ ) = (if  $v x$  then false
                        else invalid endif)

( $X \rightarrow \text{including}(x) \rightarrow \text{includes}(y)$ ) =
  (if  $\delta X$ 
   then if  $x \doteq y$ 
        then true
        else  $X \rightarrow \text{includes}(y)$ 
        endif
   else invalid
   endif)

```

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

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

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:

value “ $\tau \models (\text{Set}\{\text{Set}\{2, \text{null}\}\} \doteq \text{Set}\{\text{Set}\{\text{null}, 2\}\})$ ”

make consult Section 2.9; these test-statements have been machine-checked and proven consistent with the denotational and logic semantics of Featherweight OCL.

0.3.6. Object-oriented Datatype Theories

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

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

1. C is a set of class names (written as $\{C_1, \dots, C_n\}$). To each class name a type of data in OCL is associated. Moreover, class names declare two projector functions to the set of all objects in a state: $C_i.\text{allInstances}()$ and $C_i.\text{allInstances@pre}()$,
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 $X.a@pre :: C_i \rightarrow A$ for $A \in \text{TYPES}(C)$),
4. $\text{Assoc}(C_i, C_j)$ is a collection of associations¹³. An association $(n, m_{\text{from}}, m_{\text{to}}) \in \text{Assoc}(C_i, C_j)$ between to classes C_i and C_j is a triple consisting of a (unique) association name n , and the role-names m_{to} and m_{from} . To each role-name belong two families of accessors denoted $X.a :: C_i \rightarrow A$ and $X.a@pre :: C_i \rightarrow A$ 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 : $\text{obj} :: C_i.\text{oclAsType}(C_j)$,

¹³Given the fact that there is at present no consensus on the semantics of n-ary associations, Featherweight OCL restricts itself to binary associations.

6. for each class $C_i \in C$, there are two dynamic type tests ($X.\text{oclIsTypeOf}(C_i)$ and $X.\text{oclIsKindOf}(C_i)$),
7. and last but not least, for each class name $C_i \in C$ there is an instance of the overloaded referential equality (written $_ \doteq _$).

Assuming a strong static type discipline in the sense of Hindley-Milner types, Featherweight OCL has no “syntactic subtyping.” In contrast, sub-typing can be expressed *semantically* in Featherweight OCL by adding suitable type-casts which do have a formal semantics. Thus, sub-typing becomes an issue of the front-end that can make implicit type-coercions explicit. 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 for constructors, accessors, tests and casts can be derived. A concrete universe constructed from a class model will be used to instantiate the implicit type parameter $\mathcal{A}$ of all OCL operations discussed so far.

0.3.6.1. 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:

$$\text{typedef} \quad \mathcal{A} \text{ state} := \{ \sigma :: \text{oid} \rightarrow \alpha \mid \text{inv}_\sigma(\sigma) \} \quad (0.16)$$

where inv_σ is a to be discussed invariant on states.

The key point is that we need a common type $\mathcal{A}$ 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 one-to-one by HOL types, there are two fundamentally different ways to construct such a set of object representations, which we call an *object universe* $\mathcal{A}$:

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

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

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

$$\mathcal{A} := C_1 + \dots + C_n. \quad (0.17)$$

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

$$X.\text{oclIsTypeOf}(C_k) \text{ implies } X.\text{oclAsType}(C_i).\text{oclAsType}(C_k) \doteq X \quad (0.18)$$

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

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

- Then the *class type extension* $C_{i\text{ext}}$ associated to C_i is $A_{i_1} \times \dots \times A_{i_n} \times (C_{j_1\text{ext}} + \dots + C_{j_m\text{ext}})_\perp$ where A_{i_k} ranges over the local attribute types of C_i and $C_{j\text{ext}}$ ranges over all class type extensions of the subclass C_j of C_i .
- Then the *class type* for C_i is $\text{oid} \times A_{i_1} \times \dots \times A_{i_n} \times (C_{j_1\text{ext}} + \dots + C_{j_m\text{ext}})_\perp$ where A_{i_k} ranges over the inherited *and* local attribute types of C_i and $C_{j\text{ext}}$ 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 Chapter 4.

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

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

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

We will work out concrete examples for the construction of the object-universes in Chapter 4 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.

0.3.6.2. 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 *de-referentiation* 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 de-referentiation in one of the states to produce an object representation again. The exceptional case of non-existence in this state must be treated.

The first phase directly translates into the following formalization:

definition

$$\begin{aligned} \text{eval_extract} X f = (\lambda \tau. \text{case } X \tau \text{ of } & \perp & \Rightarrow \text{invalid } \tau & \text{exception} \\ & | \perp_\perp & \Rightarrow \text{invalid } \tau & \text{deref. null} \\ & | \perp_{obj} & \Rightarrow f(\text{oid_of } obj) \tau) \end{aligned} \quad (0.19)$$

For each class C , we introduce the de-referentiation phase of this form:

$$\text{definition deref_oid}_C \text{ fst_snd f oid} = (\lambda \tau. \text{case}(\text{heap}(\text{fst_snd } \tau)) \text{ oid of} \\ \begin{array}{ll} \text{in}_C \text{ obj}_\perp & \Rightarrow \text{f obj } \tau \\ \text{!}_\perp & \Rightarrow \text{invalid } \tau \end{array}) \quad (0.20)$$

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:

$$\text{definition select}_a \text{ f} = (\lambda \text{ mk}_C \text{ oid } \dots \perp \dots \text{ C}_{\text{Xext}} \Rightarrow \text{null} \\ \mid \text{ mk}_C \text{ oid } \dots \text{!}_\perp \dots \text{ C}_{\text{Xext}} \Rightarrow \text{f}(\lambda x \text{ } _ \text{. } _ \text{!}_\perp x) a) \quad (0.21)$$

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

$$\begin{array}{lll} \text{definition} & \text{in_pre_state} & = \text{fst} & \text{first component} \\ \text{definition} & \text{in_post_state} & = \text{snd} & \text{second component} \\ \text{definition} & \text{reconst_basetype} & = \text{id} & \text{identity function} \end{array} \quad (0.22)$$

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

$$\begin{array}{ll} \text{definition} & _ \text{.getBase} :: C \Rightarrow A_{\text{base}} \\ \text{where} & X \text{.getBase} = \text{eval_extract } X(\text{deref_oid}_C \text{ in_post_state} \\ & \quad (\text{select}_{\text{getBase}} \text{ reconst_basetype})) \end{array} \quad (0.23)$$

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

$$\begin{array}{ll} \text{definition} & _ \text{.getObject} :: C \Rightarrow A_{\text{object}} \\ \text{where} & X \text{.getObject} = \text{eval_extract } X(\text{deref_oid}_C \text{ in_post_state} \\ & \quad (\text{select}_{\text{getObject}} (\text{deref_oid}_C \text{ in_post_state}))) \end{array} \quad (0.24)$$

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 $_ \text{.a@pre}$ were produced when in_post_state is replaced by in_pre_state .

Examples for the construction of accessors via associations can be found in Section 4.8. 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 $0 \dots 1$ 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.

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

To facilitate accessing attributes with multiplicity $0 \dots 1$, the OCL standard states that single values can be used as sets by calling collection operations on them. This implicit conversion of a value to a *Set* is not defined by the standard.

We argue that the resulting set cannot be constructed the same way as when evaluating a *Set* literal. Otherwise, *null* would be mapped to the singleton set containing *null*, but the standard demands that the resulting set is empty in this case. The conversion should instead be defined as follows:

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

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

We propose to resolve this dilemma by regarding multiplicities as optional. This point of view complies with the UML standard, that does not require lower and upper bounds to be defined for multiplicities.¹⁴ In case a multiplicity is specified for an attribute, i.e., a lower and an upper bound are provided, we require for any collection the attribute evaluates to a collection not containing `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.

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

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

If the upper bound `n` is infinite, the second invariant is omitted. For the definition of these invariants we are making use of the conversion of single values to sets described in paragraph 0.3.6.2.1. If $n \leq 1$, the attribute `a` evaluates to a single value, which is then converted to a `Set` on which the `size` operation is called.

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

¹⁴We 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.

0.3.6.3. 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:

$$\begin{aligned} \tau \models X.\text{oclAsType}(C_i) &\triangleq X \\ \tau \models \text{invalid}.\text{oclAsType}(C_i) &\triangleq \text{invalid} \\ \tau \models \text{null}.\text{oclAsType}(C_i) &\triangleq \text{null} \\ \tau \models (X :: C_i).\text{oclAsType}(C_j) &\triangleq X \\ \tau \models X.\text{oclAsType}(C_j) &\triangleq X \\ \tau \models (X :: \text{OclAny}).\text{oclAsType}(\text{OclAny}) &\triangleq X \\ \tau \models v(X :: C_i) \implies \tau \models (X.\text{oclIsTypeOf}(C_i) \text{ implies } (X.\text{oclAsType}(C_j).\text{oclAsType}(C_i)) \doteq X) \\ \tau \models v(X :: C_i) \implies \tau \models X.\text{oclIsTypeOf}(C_i) \text{ implies } (X.\text{oclAsType}(C_j).\text{oclAsType}(C_i)) \doteq X \\ \tau \models \delta X \implies \tau \models X.\text{oclAsType}(C_j).\text{oclAsType}(C_i) &\triangleq X \\ \tau \models vX \implies \tau \models X.\text{oclIsTypeOf}(C_i) \text{ implies } X.\text{oclAsType}(C_j).\text{oclAsType}(C_i) &\doteq X \\ \tau \models X.\text{oclIsTypeOf}(C_j) \implies \tau \models \delta X \implies \tau \models \text{not}(vX.\text{oclAsType}(C_i)) \\ \tau \models \text{invalid}.\text{oclIsTypeOf}(C_i) &\triangleq \text{invalid} \\ \tau \models \text{null}.\text{oclIsTypeOf}(C_i) &\triangleq \text{true} \\ \tau \models \text{Person}.\text{allInstances}() \text{ ->forAll } (X) X.\text{oclIsTypeOf}(C_z) \\ \tau \models \text{Person}.\text{allInstances@pre}() \text{ ->forAll } (X) X.\text{oclIsTypeOf}(C_z) \\ \tau \models \text{Person}.\text{allInstances}() \text{ ->forAll } (X) X.\text{oclIsKindOf}(C_i) \\ \tau \models \text{Person}.\text{allInstances@pre}() \text{ ->forAll } (X) X.\text{oclIsKindOf}(C_i) \\ \tau \models (X :: C_i).\text{oclIsTypeOf}(C_j) \implies \tau \models (X :: C_i).\text{oclIsKindOf}(C_i) \\ (\tau \models (X :: C_j) \doteq X) = (\tau \models \text{if } vX \text{ then true else invalid endif}) \\ \tau \models (X :: C_j) \doteq Y \implies \tau \models Y \doteq X \\ \tau \models (X :: C_j) \doteq Y \implies \tau \models Y \doteq Z \implies \tau \models X \doteq Z \end{aligned}$$

0.3.6.4. Algebraic Properties of the Class-Models

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

$$\begin{aligned} \text{invalid}.\text{oclIsTypeOf}(C_i) &= \text{invalid} & \text{null}.\text{oclIsTypeOf}(C_i) &= \text{true} \\ \text{invalid}.\text{oclIsKindOf}(C_i) &= \text{invalid} & \text{null}.\text{oclIsKindOf}(C_i) &= \text{true} \\ (X :: C_i).\text{oclAsType}(C_i) &= X & \text{invalid}.\text{oclAsType}(C_i) &= \text{invalid} \\ \text{null}.\text{oclAsType}(C_i) &= \text{null} & (X :: C_i).\text{oclAsType}(C_j).\text{oclAsType}(C_i) &= X \\ (X :: C_i) \doteq X &= \text{if } vX \text{ then true else invalid endif} \end{aligned}$$

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

$$\text{invalid}.a = \text{invalid} \quad \text{null}.a = \text{invalid}$$

```

invalid.a@pre=invalid    null.a@pre=invalid
invalid.r=invalid        null.r=invalid
invalid.r@pre=invalid    null.r@pre=invalid

```

0.3.6.5. Other Operations on States

Defining `_ .allInstances()` is straight-forward; the only difference is the property `T.allInstances() -> excludes(null)` which is a consequence of the fact that `null`'s are values and do not “live” in the state. OCL semantics admits states with “dangling references,”; it is the semantics of accessors or roles which maps these references to `invalid`, which makes it possible to rule out these situations in invariants. OCL does not guarantee that an operation only modifies the path-expressions mentioned in the postcondition, i. e., it allows arbitrary relations from pre-states to post-states. This framing problem is well-known (one of the suggested solutions is [15]). We define

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

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

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

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

0.3.7. Data Invariants

Since the present OCL semantics uses one interpretation function¹⁵, we express the effect of OCL terms occurring in preconditions and invariants by a syntactic transformation `_pre` which replaces:

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

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

$$\begin{aligned}
I[\text{context } c : C_i \text{ inv } n : \phi(c)] \tau \equiv \\
\tau \models (C_i.allInstances() \rightarrow \text{forall}(x \mid \phi(x))) \wedge \\
\tau \models (C_i.allInstances() \rightarrow \text{forall}(x \mid \phi(x)))_{pre}
\end{aligned} \tag{0.25}$$

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

¹⁵This has been handled differently in previous versions of the Annex A.

0.3.8. Operation Contracts

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

$$\begin{aligned}
& I[\text{context } C :: op(a_1, \dots, a_n) : T \\
& \quad \text{pre } \phi(self, a_1, \dots, a_n) \\
& \quad \text{post } \psi(self, a_1, \dots, a_n, result)] \equiv \\
& \quad \lambda s, x_1, \dots, x_n, \tau. \\
& \quad \text{if } \tau \models \partial s \wedge \tau \models v x_1 \wedge \dots \wedge \tau \models v x_n \\
& \quad \text{then SOME } result. \quad \tau \models \phi(s, x_1, \dots, x_n)_{\text{pre}} \\
& \quad \quad \quad \wedge \tau \models \psi(s, x_1, \dots, x_n, result)) \\
& \quad \text{else } \perp
\end{aligned} \tag{0.26}$$

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 one¹⁶. Thus, using the Hilbert-Choice Operator, a contract can be associated to a function definition:

$$f_{op} \equiv I[\text{context } C :: op(a_1, \dots, a_n) : T \dots] \tag{0.27}$$

provided that neither ϕ nor ψ contain recursive method calls of op . In the case of a query operation (i.e., τ must have the form: (σ, σ) , which means that query operations do not change the state; c.f. `oclIsModifiedOnly()` in Section 0.3.6.5), this constraint can be relaxed: the above equation is then stated as *axiom*. Note however, that the consistency of the overall theory is for recursive query contracts left to the user (it can be shown, for example, by a proof of termination, i.e., by showing that all recursive calls were applied to argument vectors that are smaller wrt. a well-founded ordering). Under this condition, an f_{op} resulting from recursive query operations can be used safely inside pre- and post-conditions of other contracts.

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

$$\frac{
\begin{array}{c}
[\tau \models \psi \text{ self } a_1 \dots a_n \text{ res}]_{\text{res}} \\
\vdots \\
\tau \models E(\text{res})
\end{array}
}{
\tau \models E(f_{op} \text{ self } a_1 \dots a_n)
} \tag{0.28}$$

under the conditions:

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

¹⁶In HOL, the Hilbert-Choice operator is a first-class element of the logical language.

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

$$\frac{\tau \models \phi \text{ self } a_1 \dots a_n}{(\tau \models E(f_{\text{op}} \text{ self } a_1 \dots a_n)) = e(\tau \models E(\text{BODY self } a_1 \dots a_n))} \quad (0.29)$$

where

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

Currently, Featherweight OCL neither supports overloading nor overriding for user-defined operations: the Featherweight OCL compiler needs to be extended to generate pre-conditions that constrain the classes on which an overridden function can be called as well as the dispatch order. This construction, overall, is similar to the virtual function table that, e.g., is generated by C++ compilers. Moreover, 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.

1. Formalization I: OCL Types and Core Definitions

1.1. Preliminaries

1.1.1. Notations for the Option Type

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

unbundle *no floor-ceiling-syntax*

type-notation *option* ($\langle \cdot \rangle_{\perp}$)

notation *Some* ($\langle \cdot \rangle_{\perp}$)

notation *None* ($\langle \cdot \rangle_{\perp}$)

These commands introduce an alternative, more compact notation for the type constructor $\langle 'a \rangle_{\perp}$, namely $\langle 'a \rangle_{\perp}$. Furthermore, the constructors $\perp_X$ and $\perp$ of the type $\langle 'a \rangle_{\perp}$, namely $\perp_X$ and $\perp$.

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

fun *drop* :: $'a \text{ option} \Rightarrow 'a$ ($\langle \cdot \rangle_{\perp}$)

where *drop-lift[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 *Sem* as defined below is just a textual marker for presentation purposes, i.e. intended for readers used to conventional textbook notations on semantics. Since we use a “shallow embedding”, i.e. since we represent the syntax of OCL directly by HOL constants, the interpretation function is semantically not only superfluous, but from an Isabelle perspective strictly in the way for certain consistency checks performed by the definitional packages.

definition *Sem* :: $'a \Rightarrow 'a$ ($\langle I[\cdot] \rangle$)

where $I[x] \equiv x$

1.1.2. Common Infrastructure for all OCL Types

In order to have the possibility to nest collection types, such that we can give semantics to expressions like $\text{Set}\{\text{Set}\{2\}, \text{null}\}$, it is necessary to introduce a uniform interface for types having the *invalid* (= bottom) element. The reason is that we impose a data-invariant on raw-collection **types_code** which assures that the *invalid* element is not allowed inside the collection; all raw-collections of this form were identified with the *invalid* element itself. The construction requires that the new collection type is not comparable with the raw-types (consisting of nested option type constructions), such that the data-invariant must be expressed in terms of the interface. In a second step, our base-types will be shown to be instances of this interface.

This uniform interface consists in a type class requiring the existence of a bot and a null element. The construction proceeds by abstracting the null (defined by $\perp_{\perp}$ on $'a \text{ option option}$) to a *null* element, which may have an arbitrary semantic structure, and an undefinedness element $\perp$ to an abstract undefinedness element *bot* (also written $\perp$ whenever no

confusion arises). As a consequence, it is necessary to redefine the notions of invalid, defined, valuation etc. on top of this interface.

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

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

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

1.1.3. Accommodation of Basic Types to the Abstract Interface

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

```
instantiation option :: (type)bot
begin
  definition bot-option-def: (bot::'a option)  $\equiv$  (None::'a option)
  instance
end
```

```
instantiation option :: (bot)null
begin
  definition null-option-def: (null::'a::bot option)  $\equiv$   $\lfloor bot \rfloor$ 
  instance
end
```

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

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

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

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

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

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

type-synonym $oid = nat$

We refrained from the alternative:

type-synonym $oid = ind$

which is slightly more abstract but non-executable.

States in UML/OCL are a pair of

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

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

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

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

type-synonym $(' \mathcal{A})_{st} = ' \mathcal{A}\ state \times ' \mathcal{A}\ state$

We will require for all objects that there is a function that projects the oid of an object in the state (we will settle the question how to define this function later). We will use the Isabelle type class mechanism [13] to capture this:

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

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

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

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

instantiation *option* :: (object)object
begin
 definition *oid-of-option-def*: $oid-of\ x = oid-of\ (the\ x)$
 instance
end

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

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

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

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

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:

1.1.6. The fundamental constants ‘invalid’ and ‘null’ in all OCL Types

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

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

where *invalid* $\equiv \lambda \tau. \text{bot}$

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

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

Note that the definition :

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

where *null* $\equiv \lambda \tau. \text{null}$

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

lemma *textbook-null-fun*: $I\llbracket \text{null}::(\mathcal{A}, \alpha::\text{null}) \text{ val} \rrbracket \tau = (\text{null}::(\alpha::\text{null}))$

1.2. Basic OCL Value Types

The structure of this section roughly follows the structure of Chapter 11 of the OCL standard [19], which introduces the OCL Library.

The semantic domain of the (basic) boolean type is now defined as the Standard: the space of valuation to $\langle \langle \text{bool} \rangle_{\perp} \rangle_{\perp}$, i. e. the Boolean base type:

type-synonym *Boolean*_{base} = *bool option option*

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

lemma *Boolean*_{base}-cases:

fixes *b* :: $\langle \text{Boolean}_{\text{base}} \rangle$

obtains $\langle b = \perp \rangle \mid \langle b = \perp_{\perp} \rangle \mid \langle b = \perp_{\perp} \text{False}_{\perp} \rangle \mid \langle b = \perp_{\perp} \text{True}_{\perp} \rangle$

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:

type-synonym *Integer*_{base} = *int option option*

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

type-synonym String_{base} = string option option

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

type-synonym Real_{base} = real option option

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

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

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

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

typedef Void_{base} = {X::unit option option. X = bot $\vee$ X = null }

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

1.3. Some OCL Collection Types

For the semantic construction of the collection types, we have two goals:

1. we want the types to be *fully abstract*, i. e., the type should not contain junk-elements that are not representable by OCL expressions, and
2. we want a possibility to nest collection types (so, we want the potential of talking about $Set(Set(Sequences(Pairs(X,Y))))$).

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

1.3.1. The Construction of the Pair Type (Tuples)

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

typedef (overloaded) ($'\alpha$, $'\beta$) Pair_{base} = {X::($'\alpha$::null $\times$ $'\beta$::null) option option.
X = bot $\vee$ X = null $\vee$ (fst[⌈]X[⌋] $\neq$ bot $\wedge$ snd[⌈]X[⌋] $\neq$ bot)}

setup-lifting type-definition-Pair_{base}

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

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

lift-definition $bot\text{-}Pair_{base} :: ('a, 'b) Pair_{base}$ **is** $None$

instance

end

lemma $Abs\text{-}Pair_{base}\text{-}invalid\text{-}eq$ [simp]:
 $\langle Abs\text{-}Pair_{base} (invalid\ \tau) = invalid\ \tau \rangle$

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

lift-definition $null\text{-}Pair_{base} :: ('a, 'b) Pair_{base}$ **is** $_None_$

instance

end

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

lift-definition $equal\text{-}Pair_{base} :: ('a, 'b) Pair_{base} \Rightarrow ('a, 'b) Pair_{base} \Rightarrow bool$
is $HOL.equal :: ('a \times 'b) option option \Rightarrow -$

instance

end

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

type-synonym $('\mathcal{A}, '\alpha, '\beta) Pair = (\mathcal{A}, (' \alpha, '\beta) Pair_{base})$ **val**
type-notation $Pair_{base} \langle Pair'(-, -') \rangle$

1.3.2. The Construction of the Set Type

The core of an own type construction is done via a type definition which provides the raw-type $'\alpha Set_{base}$. It is shown that this type “fits” indeed into the abstract type interface discussed in the previous section. Note that we make no restriction whatsoever to *finite* sets; while with the standards type-constructors only finite sets can be denoted, there is the possibility to define in fact infinite type constructors in Featherweight OCL (c.f. Section 2.9.1).

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

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

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

instance
end

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

begin

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

instance

end

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

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

type-notation $\text{Set}_{\text{base}} (\langle \text{Set}'(-)' \rangle)$

1.3.3. The Construction of the Bag Type

The core of an own type construction is done via a type definition which provides the raw-type $'a \text{Bag}_{\text{base}}$ based on multi-sets from the HOL library. As in Sets, it is shown that this type “fits” indeed into the abstract type interface discussed in the previous section, and as in sets, we make no restriction whatsoever to *finite* multi-sets; while with the standards type-constructors only finite sets can be denoted, there is the possibility to define in fact infinite type constructors in Featherweight OCL (c.f. Section 2.9.1). However, while several *null* elements are possible in a Bag, there can’t be no bottom (invalid) element in them.

typedef (overloaded) $'a \text{Bag}_{\text{base}} = \{X::('a::\text{null} \Rightarrow \text{nat}) \text{option option}. X = \text{bot} \vee X = \text{null} \vee \ulcorner X \urcorner \text{bot} = 0 \}$

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

begin

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

instance

end

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

begin

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

instance

end

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

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

type-notation $\text{Bag}_{\text{base}} (\langle \text{Bag}'(-)' \rangle)$

1.3.4. The Construction of the Sequence Type

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

typedef (overloaded) $'a \text{Sequence}_{\text{base}} = \{X::('a::\text{null}) \text{list option option}.$
 $X = \text{bot} \vee X = \text{null} \vee (\forall x \in \text{set } \ulcorner X \urcorner. x \neq \text{bot})\}$

setup-lifting *type-definition-Sequence_{base}*

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

begin

lift-definition $bot-Sequence_{base} :: 'a Sequence_{base}$ **is** $None$

instance

end

lemma $Sequence_{base}-invalid-eq [simp]:$

$\langle Abs-Sequence_{base} (invalid \ \tau) = invalid \ \tau \rangle$

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

begin

lift-definition $null-Sequence_{base} :: 'a Sequence_{base}$ **is** $\bot$

instance

end

instantiation $Sequence_{base} :: (\{null, equal\}) equal$

begin

lift-definition $equal-Sequence_{base} :: 'a Sequence_{base} \Rightarrow 'a Sequence_{base} \Rightarrow bool$

is $HOL.equal :: 'a list option option \Rightarrow -$

instance

end

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

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

type-notation $Sequence_{base} (\langle Sequence'(-') \rangle)$

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

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

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

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

1. cartesian products of arguments were curried,
2. constants of type T were mapped to valuations over the HOL-type for T ,
3. functions $T \rightarrow T'$ were mapped to functions in HOL, where T and T' were mapped to the valuations for them, and

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

Table 1.1. – Correspondance between OCL types and HOL types

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.

2. Formalization II: OCL Terms and Library Operations

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

2.1.1. Basic Constants

lemma *bot-Boolean-def* : (*bot*::(^ℓℓ)Boolean) = ($\lambda \tau. \perp$)

lemma *null-Boolean-def* : (*null*::(^ℓℓ)Boolean) = ($\lambda \tau. \perp_{\perp}$)

definition *true* :: (^ℓℓ)Boolean

where *true* $\equiv \lambda \tau. \perp_{\perp} \text{True}_{\perp}$

definition *false* :: (^ℓℓ)Boolean

where *false* $\equiv \lambda \tau. \perp_{\perp} \text{False}_{\perp}$

lemma *bool-split-0*: $X \tau = \text{invalid } \tau \vee X \tau = \text{null } \tau \vee$
 $X \tau = \text{true } \tau \quad \vee X \tau = \text{false } \tau$

lemma [*simp*]: *false* (*a*, *b*) = $\perp_{\perp} \text{False}_{\perp}$

lemma [*simp*]: *true* (*a*, *b*) = $\perp_{\perp} \text{True}_{\perp}$

lemma *textbook-true*: $I[\![\text{true}]\!] \tau = \perp_{\perp} \text{True}_{\perp}$

lemma *textbook-false*: $I[\![\text{false}]\!] \tau = \perp_{\perp} \text{False}_{\perp}$

Name	Theorem
<i>textbook-invalid</i>	$I[\![\text{invalid}]\!] \tau = \text{UML-Types.bot-class.bot}$
<i>textbook-null-fun</i>	$I[\![\text{null}]\!] \tau = \text{null}$
<i>textbook-true</i>	$I[\![\text{true}]\!] \tau = \perp_{\perp} \text{True}_{\perp}$
<i>textbook-false</i>	$I[\![\text{false}]\!] \tau = \perp_{\perp} \text{False}_{\perp}$

Table 2.1. – Basic semantic constant definitions of the logic

2.1.2. Validity and Definedness

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

definition *valid* :: (^ℓℓ, ^ℓℓ_a::null)val $\Rightarrow$ (^ℓℓ)Boolean ($\langle v \rightarrow [100]100 \rangle$)

```

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

```

The OCL contains a particular version of equality, written in Standard documents $_ = _$ and $_ \neq _$ for its negation, which is referred as *weak referential equality* hereafter and for which we use the symbol $_ \doteq _$ throughout the formal part of this document. Its semantics is motivated by the desire of fast execution, and similarity to languages like Java and C, but does not satisfy the needs of logical reasoning over OCL expressions and specifications. We therefore introduce a second equality, referred as *strong equality* or *logical equality* and written $_ \triangleq _$ which is not present in the current standard but was discussed in prior texts on OCL like the Amsterdam Manifesto [11] 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

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

Table 2.3. – Laws of the basic predicates of the logic.

and reason over OCL. It is therefore a natural task in Featherweight OCL to formally investigate the somewhat quite complex relationship between these two.

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

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

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

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

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

$$s = t \implies P(s) = P(t) \quad (2.1)$$

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

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

2.1.3.1. Definition

The strict equality on basic types (actually on all types) must be exceptionally defined on *null*—otherwise the entire concept of null in the language does not make much sense. This is an important exception from the general rule that null arguments—especially if passed as “self”-argument—lead to invalid results.

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

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

From this follow already elementary properties like:

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

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

2.1.3.2. Fundamental Predicates on Strong Equality

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

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

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

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

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

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

shows $(X \triangleq Z) = \text{true}$

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 $cp: \bigwedge X. P(X) \tau = P(\lambda \cdot. X \tau) \tau$

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

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

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

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

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

2.1.4. Logical Connectives and their Universal Properties

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

Practically, this means that we want to give a definition to the core operations to be as close as possible to the lattice laws; this makes also powerful symbolic normalization of OCL specifications possible as a pre-requisite for automated theorem provers. For example, it is still possible to compute without any definedness and validity reasoning the DNF

of an OCL specification; be it for test-case generations or for a smooth transition to a two-valued representation of the specification amenable to fast standard SMT-solvers, for example.

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

definition $OclNot :: (^{\mathcal{A}})Boolean \Rightarrow (^{\mathcal{A}})Boolean \langle not \rangle$

where $not\ X \equiv \lambda\ \tau . case\ X\ \tau\ of$

$$\begin{aligned} \perp &\Rightarrow \perp \\ | \perp \perp_{\perp} &\Rightarrow \perp \perp_{\perp} \\ | \perp x_{\perp} &\Rightarrow \perp \neg x_{\perp} \end{aligned}$$

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

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

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

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

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

lemma $OclNot\text{-}not[simp]: not\ (not\ X) = X$

lemma $OclNot\text{-}inject: \bigwedge x\ y. not\ x = not\ y \implies x = y$

definition $OclAnd :: [(^{\mathcal{A}})Boolean, (^{\mathcal{A}})Boolean] \Rightarrow (^{\mathcal{A}})Boolean \text{ (infixl } \langle and \rangle 30)$

where $X\ and\ Y \equiv (\lambda\ \tau . case\ X\ \tau\ of$

$$\begin{aligned} \perp_{False} &\Rightarrow \perp_{False} \\ | \perp &\Rightarrow (case\ Y\ \tau\ of \\ &\quad \perp_{False} \Rightarrow \perp_{False} \\ &\quad | - \Rightarrow \perp) \\ | \perp_{\perp} &\Rightarrow (case\ Y\ \tau\ of \\ &\quad \perp_{False} \Rightarrow \perp_{False} \\ &\quad | \perp \Rightarrow \perp \\ &\quad | - \Rightarrow \perp_{\perp}) \\ | \perp_{True} &\Rightarrow Y\ \tau \end{aligned}$$

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

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

lemma *textbook-OclNot:*

$$\begin{aligned} I[not(X)]\ \tau &= (case\ I[X]\ \tau\ of\ \perp \Rightarrow \perp \\ &\quad | \perp \perp_{\perp} \Rightarrow \perp \perp_{\perp} \\ &\quad | \perp x_{\perp} \Rightarrow \perp \neg x_{\perp}) \end{aligned}$$

lemma *textbook-OclAnd:*

$$\begin{aligned} I[X\ and\ Y]\ \tau &= (case\ I[X]\ \tau\ of \\ &\quad \perp \Rightarrow (case\ I[Y]\ \tau\ of \\ &\quad \quad \perp \Rightarrow \perp) \end{aligned}$$

$$\begin{aligned}
& \vdash \perp_{\perp} \Rightarrow \perp \\
& \vdash \text{True}_{\perp} \Rightarrow \perp \\
& \vdash \text{False}_{\perp} \Rightarrow \perp \text{False}_{\perp}) \\
& \vdash \perp_{\perp} \Rightarrow (\text{case } I[\![Y]\!] \text{ of} \\
& \quad \perp \Rightarrow \perp \\
& \quad \vdash \perp_{\perp} \Rightarrow \perp_{\perp} \\
& \quad \vdash \text{True}_{\perp} \Rightarrow \perp_{\perp} \\
& \quad \vdash \text{False}_{\perp} \Rightarrow \perp \text{False}_{\perp}) \\
& \vdash \text{True}_{\perp} \Rightarrow (\text{case } I[\![Y]\!] \text{ of} \\
& \quad \perp \Rightarrow \perp \\
& \quad \vdash \perp_{\perp} \Rightarrow \perp_{\perp} \\
& \quad \vdash y_{\perp} \Rightarrow \perp y_{\perp}) \\
& \vdash \text{False}_{\perp} \Rightarrow \perp \text{False}_{\perp})
\end{aligned}$$

definition *OclOr* :: [(^{'A})Boolean, (^{'A})Boolean] ⇒ (^{'A})Boolean (**infixl** <or> 25)
where *X or Y* ≡ not(not *X* and not *Y*)

definition *OclImplies* :: [(^{'A})Boolean, (^{'A})Boolean] ⇒ (^{'A})Boolean (**infixl** <implies> 25)
where *X implies Y* ≡ not *X or Y*

lemma *cp-OclAnd*:(*X and Y*) τ = ((λ -. *X* τ) and (λ -. *Y* τ)) τ

lemma *cp-OclOr*:(*X*::(^{'A})Boolean) or *Y*) τ = ((λ -. *X* τ) or (λ -. *Y* τ)) τ

lemma *cp-OclImplies*:(*X implies Y*) τ = ((λ -. *X* τ) implies (λ -. *Y* τ)) τ

lemma *OclAnd1[simp]*: (invalid and true) = invalid

lemma *OclAnd2[simp]*: (invalid and false) = false

lemma *OclAnd3[simp]*: (invalid and null) = invalid

lemma *OclAnd4[simp]*: (invalid and invalid) = invalid

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

lemma *OclAnd6[simp]*: (null and false) = false

lemma *OclAnd7[simp]*: (null and null) = null

lemma *OclAnd8[simp]*: (null and invalid) = invalid

lemma *OclAnd9[simp]*: (false and true) = false

lemma *OclAnd10[simp]*: (false and false) = false

lemma *OclAnd11[simp]*: (false and null) = false

lemma *OclAnd12[simp]*: (false and invalid) = false

lemma *OclAnd13[simp]*: (true and true) = true

lemma *OclAnd14[simp]*: (true and false) = false

lemma *OclAnd15[simp]*: (true and null) = null

lemma *OclAnd16[simp]*: (true and invalid) = invalid

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

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

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

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

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

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

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

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

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

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

lemma *OclAnd-assoc*: (*X and (Y and Z)*) = (*X and Y and Z*) (*is* $\langle ?lhs = ?rhs \rangle$)

lemma *OclOr1[simp]*: (*invalid or true*) = *true*

lemma *OclOr2[simp]*: (*invalid or false*) = *invalid*

lemma *OclOr3[simp]*: (*invalid or null*) = *invalid*

lemma *OclOr4[simp]*: (*invalid or invalid*) = *invalid*

lemma *OclOr5[simp]*: (*null or true*) = *true*

lemma *OclOr6[simp]*: (*null or false*) = *null*

lemma *OclOr7[simp]*: (*null or null*) = *null*

lemma *OclOr8[simp]*: (*null or invalid*) = *invalid*

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

lemma *OclOr-commute*: (*X or Y*) = (*Y or X*)

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

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

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

lemma *OclOr-true2*: (*Y or true*) = *true*

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

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

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

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

lemma *OclOr-assoc*: (*X or (Y or Z)*) = (*X or Y or Z*)

lemma *deMorgan1*: *not(X and Y)* = (*(not X) or (not Y)*)

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

lemma *OclImplies-true1[simp]*: $(\text{true implies } X) = X$

lemma *OclImplies-true2[simp]*: $(X \text{ implies true}) = \text{true}$

lemma *OclImplies-false1[simp]*: $(\text{false implies } X) = \text{true}$

2.1.5. A Standard Logical Calculus for OCL

definition *OclValid* :: $[(^{\mathcal{A}})\text{st}, (^{\mathcal{A}})\text{Boolean}] \Rightarrow \text{bool } (\langle I(-) \models (-) \rangle 50)$

where $\tau \models P \equiv ((P \ \tau) = \text{true } \tau)$

syntax *OclNonValid* :: $[(^{\mathcal{A}})\text{st}, (^{\mathcal{A}})\text{Boolean}] \Rightarrow \text{bool } (\langle I(-) \not\models (-) \rangle 50)$

syntax-consts *OclNonValid* == *Not*

translations $\tau \not\models P \equiv \neg(\tau \models P)$

2.1.5.1. Global vs. Local Judgements

lemma *transform1*: $P = \text{true} \Longrightarrow \tau \models P$

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

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

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

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

lemma

assumes $H : P = \text{true} \Longrightarrow Q = \text{true}$

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

2.1.5.2. Local Validity and Meta-logic

lemma *foundation1[simp]*: $\tau \models \text{true}$

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

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

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

lemma *bool-split[simp]*:

$(\tau \models (x \triangleq \text{invalid})) \vee (\tau \models (x \triangleq \text{null})) \vee (\tau \models (x \triangleq \text{true})) \vee (\tau \models (x \triangleq \text{false}))$

lemma *defined-split*:

$$(\tau \models \delta x) = ((\neg(\tau \models (x \triangleq \text{invalid}))) \wedge (\neg(\tau \models (x \triangleq \text{null}))))$$

$$\text{lemma valid-bool-split: } (\tau \models v A) = ((\tau \models A \triangleq \text{null}) \vee (\tau \models A) \vee (\tau \models \text{not } A))$$

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

lemma foundation5:

$$\tau \models (P \text{ and } Q) \implies (\tau \models P) \wedge (\tau \models Q)$$

lemma foundation6:

$$\tau \models P \implies \tau \models \delta P$$

lemma foundation7[simp]:

$$(\tau \models \text{not } (\delta x)) = (\neg(\tau \models \delta x))$$

lemma foundation7'[simp]:

$$(\tau \models \text{not } (v x)) = (\neg(\tau \models v x))$$

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

lemma foundation8:

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

lemma foundation9:

$$\tau \models \delta x \implies (\tau \models \text{not } x) = (\neg(\tau \models x))$$

lemma foundation9':

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

lemma foundation9'':

$$\tau \models \text{not } x \implies \tau \models \delta x$$

lemma foundation10:

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

lemma foundation10': $(\tau \models (A \text{ and } B)) = ((\tau \models A) \wedge (\tau \models B))$

lemma foundation11:

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

lemma foundation12:

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

lemma foundation13: $(\tau \models A \triangleq \text{true}) = (\tau \models A)$

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

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

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

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

lemma *foundation16'*: $(\tau \models (\delta X)) = (X \ \tau \neq \text{invalid } \tau \wedge X \ \tau \neq \text{null } \tau)$

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

lemma *foundation18'*: $(\tau \models (\text{v } X)) = (X \ \tau \neq \text{bot})$

lemma *foundation18''*: $(\tau \models (\text{v } X)) = (\neg(\tau \models (X \triangleq \text{invalid})))$

lemma *foundation20*: $\tau \models (\delta X) \implies \tau \models \text{v } X$

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*: $(\tau \models P) = (\tau \models (\lambda _ . P \ \tau))$

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

lemma *foundation25*: $\tau \models P \implies \tau \models (P \text{ or } Q)$

lemma *foundation25'*: $\tau \models Q \implies \tau \models (P \text{ or } Q)$

lemma *foundation26*:

assumes *defP*: $\tau \models \delta P$

assumes *defQ*: $\tau \models \delta Q$

assumes *H*: $\tau \models (P \text{ or } Q)$

assumes *P*: $\tau \models P \implies R$

assumes *Q*: $\tau \models Q \implies R$

shows *R*

lemma *foundation27*: $\tau \models A \implies (\tau \models A \text{ implies } B) = (\tau \models B)$

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

lemma *valid-not-I* : $\tau \models v(x) \implies \tau \models v(\text{not } x)$

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

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

2.1.5.3. Local Judgements and Strong Equality

lemma *StrongEq-L-refl*: $\tau \models (x \triangle x)$

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

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

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* :: $((\lambda \alpha. ' \alpha) \text{ val} \implies (\lambda \beta. ' \beta) \text{ val}) \implies \text{bool}$

where $\text{cp } P \equiv (\exists f. \forall X \tau. P X \tau = f(X \tau) \tau)$

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

lemma *StrongEq-L-subst1*: $\bigwedge \tau. \text{cp } P \implies \tau \models (x \triangle y) \implies \tau \models (P x \triangle P y)$

lemma *StrongEq-L-subst2*:

$\bigwedge \tau. \text{cp } P \implies \tau \models (x \triangle y) \implies \tau \models (P x) \implies \tau \models (P y)$

lemma *StrongEq-L-subst2-rev*: $\tau \models y \triangle x \implies \text{cp } P \implies \tau \models P x \implies \tau \models P y$

lemma *StrongEq-L-subst3*:

assumes *cp*: $\text{cp } P$

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*: $\text{cp } P$

shows $(\tau \models P x) = (\tau \models P y)$

lemma *StrongEq-L-subst4-rev*:

assumes *eq*: $\tau \models (x \triangle y)$

and *cp*: $\text{cp } P$

shows $(\neg(\tau \models P x)) = (\neg(\tau \models P y))$

thm *arg-cong[of - - Not]*

lemma *cpI1*:

$(\forall X \tau. f X \tau = f(\lambda -. X \tau) \tau) \implies cp P \implies cp(\lambda X. f (P X))$

lemma *cpI2*:

$(\forall X Y \tau. f X Y \tau = f(\lambda -. X \tau)(\lambda -. Y \tau) \tau) \implies$
 $cp P \implies cp Q \implies cp(\lambda X. f (P X) (Q X))$

lemma *cpI3*:

$(\forall X Y Z \tau. f X Y Z \tau = f(\lambda -. X \tau)(\lambda -. Y \tau)(\lambda -. Z \tau) \tau) \implies$
 $cp P \implies cp Q \implies cp R \implies cp(\lambda X. f (P X) (Q X) (R X))$

lemma *cpI4*:

$(\forall W X Y Z \tau. f W X Y Z \tau = f(\lambda -. W \tau)(\lambda -. X \tau)(\lambda -. Y \tau)(\lambda -. Z \tau) \tau) \implies$
 $cp P \implies cp Q \implies cp R \implies cp S \implies cp(\lambda X. f (P X) (Q X) (R X) (S X))$

lemma *cpI5*:

$(\forall V W X Y Z \tau. f V W X Y Z \tau = f(\lambda -. V \tau)(\lambda -. W \tau)(\lambda -. X \tau)(\lambda -. Y \tau)(\lambda -. Z \tau) \tau) \implies$
 $cp N \implies cp P \implies cp Q \implies cp R \implies cp S \implies cp(\lambda X. f (N X) (P X) (Q X) (R X) (S X))$

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

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

2.1.6. OCL's if then else endif

definition *OclIf* :: $[(^{\mathcal{A}})Boolean, (^{\mathcal{A}}, ^{\mathcal{A}}::null) val, (^{\mathcal{A}}, ^{\mathcal{A}}) val] \Rightarrow (^{\mathcal{A}}, ^{\mathcal{A}}) val$
 $\langle \text{if } (-) \text{ then } (-) \text{ else } (-) \text{ endif} \rangle [10, 10, 10] 50$

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

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

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

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

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

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

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

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

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

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

lemma *OclIf-idem2*[simp]:(if $\vee$ 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)

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

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

consts *StrictRefEq* :: $((\lambda a, 'a)val, (\lambda a, 'a)val) \Rightarrow (\lambda a)Boolean$ (**infixl** $\trianglelefteq$ 30)

with term "not" we can express the notation:

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

syntax-consts

notequal == *OclNot*

translations

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

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

2.1.8. Laws to Establish Definedness (δ -closure)

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

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

lemma *OclNot-contrapos-nn*:

assumes A : $\tau \models \delta A$

assumes B : $\tau \models not\ B$

assumes C : $\tau \models A \implies \tau \models B$

shows $\tau \models not\ A$

2.1.9. A Side-calculus for Constant Terms

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

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

lemma *const-subst*:

assumes *const-X*: $const\ X$

and *const-Y*: *const Y*
and *eq* : $X \tau = Y \tau$
and *cp-P*: $cp P$
and *pp* : $P Y \tau = P Y \tau'$
shows $P X \tau = P X \tau'$

lemma *const-impl2* :
assumes $\bigwedge \tau \tau'. P \tau = P \tau' \implies Q \tau = Q \tau'$
shows $const P \implies const Q$

lemma *const-impl3* :
assumes $\bigwedge \tau \tau'. P \tau = P \tau' \implies Q \tau = Q \tau' \implies R \tau = R \tau'$
shows $const P \implies const Q \implies const R$

lemma *const-impl4* :
assumes $\bigwedge \tau \tau'. P \tau = P \tau' \implies Q \tau = Q \tau' \implies R \tau = R \tau' \implies S \tau = S \tau'$
shows $const P \implies const Q \implies const R \implies const S$

lemma *const-lam* : *const* ($\lambda \cdot. e$)

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

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

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

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

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

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

lemma *const-valid* :
assumes *const X*
shows *const* ($\vee X$)

lemma *const-OclAnd* :
assumes *const X*
assumes *const X'*
shows *const* ($X \text{ and } X'$)

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

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

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

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

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

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

lemma *const-OclValid2*:
 assumes *const x*
 shows $(\tau \models v x) = (\tau' \models v x)$

lemma *const-HOL-if* : *const C $\implies$ const D $\implies$ const F $\implies$ const ($\lambda \tau. \text{if } C \ \tau \text{ then } D \ \tau \text{ else } F \ \tau$)*

lemma *const-HOL-and*: *const C $\implies$ const D $\implies$ const ($\lambda \tau. C \ \tau \wedge D \ \tau$)*

lemma *const-HOL-eq* : *const C $\implies$ const D $\implies$ const ($\lambda \tau. C \ \tau = D \ \tau$)*

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

Miscellaneous: Overloading the syntax of “bottom”

notation *bot* ($\langle \perp \rangle$)

2.2. Property Profiles for OCL Operators via Isabelle Locales

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

2.2.1. Property Profiles for Monadic Operators

```
locale profile-mono-scheme-defined =  
  fixes f :: ('A, 'α::null)val ⇒ ('A, 'β::null)val  
  fixes g  
  assumes def-scheme: (f x) ≡ λ τ. if (δ x) τ = true τ then g (x τ) else invalid τ  
begin  
  lemma strict[simp, code-unfold]: f invalid = invalid  
  
  lemma null-strict[simp, code-unfold]: f null = invalid  
  
  lemma cp0 : f X τ = f (λ -. X τ) τ  
  
  lemma cp[simp, code-unfold] : cp P ⇒ cp (λX. f (P X))  
  
end  
  
locale profile-mono-schemeV =  
  fixes f :: ('A, 'α::null)val ⇒ ('A, 'β::null)val  
  fixes g  
  assumes def-scheme: (f x) ≡ λ τ. if (v x) τ = true τ then g (x τ) else invalid τ  
begin  
  lemma strict[simp, code-unfold]: f invalid = invalid  
  
  lemma cp0 : f X τ = f (λ -. X τ) τ  
  
  lemma cp[simp, code-unfold] : cp P ⇒ cp (λX. f (P X))  
  
end  
  
locale profile-monoΔ = profile-mono-scheme-defined +  
  assumes ∧ x. x ≠ bot ⇒ x ≠ null ⇒ g x ≠ bot  
begin  
  
  lemma const[simp, code-unfold] :  
    assumes C1 : const X  
    shows      const(f X)  
end  
  
locale profile-mono0 = profile-mono-scheme-defined +  
  assumes def-body: ∧ x. x ≠ bot ⇒ x ≠ null ⇒ g x ≠ bot ∧ g x ≠ null  
  
sublocale profile-mono0 < profile-monoΔ  
  
context profile-mono0
```

```

begin
  lemma def-homo[simp,code-unfold]:  $\delta(fx) = (\delta x)$ 

  lemma def-valid-then-def:  $v(fx) = (\delta(fx))$ 
end

```

2.2.2. Property Profiles for Single

```

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

```

2.2.3. Property Profiles for Binary Operators

```

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

```

```

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

```

```

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

```

```

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

```

```

begin

```

```

  interpretation dx : profile-single dx

```

```

  interpretation dy : profile-single dy

```

```

  lemma strictI[simp,code-unfold]: f invalid y = invalid

```

```

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

```

```

  lemma cp0 : f X Y  $\tau$  = f ( $\lambda -. X \tau$ ) ( $\lambda -. Y \tau$ )  $\tau$ 

```

```

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

```

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

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

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

lemma *const*[simp,code-unfold] :

assumes *C1* :const *X* **and** *C2* : const *Y*

shows const(*f X Y*)

end

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

locale *profile-bin-scheme-defined* =

fixes *d_y*::('A,'b::null)val $\Rightarrow$ 'A Boolean

fixes *f*::('A,'a::null)val $\Rightarrow$ ('A,'b::null)val $\Rightarrow$ ('A,'c::null)val

fixes *g*

assumes *d_y* : profile-single *d_y*

assumes *d_y-homo*[simp,code-unfold]: *cp (f X) $\implies$*

f X invalid = invalid $\implies$

$\neg \tau \models d_y\ Y \implies$

$\tau \models \delta\ f\ X\ Y \triangleq (\delta\ X\ \text{and}\ d_y\ Y)$

assumes *def-scheme*'[simplified]: bin *f g defined d_y X Y*

assumes *def-body*': $\bigwedge x\ y\ \tau. x \neq \text{bot} \implies x \neq \text{null} \implies (d_y\ y)\ \tau = \text{true} \implies g\ x\ (y\ \tau) \neq \text{bot} \wedge g\ x\ (y\ \tau) \neq \text{null}$

begin

lemma *strict3*[simp,code-unfold]: *f null y = invalid*

end

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

locale *profile-bin_d-d* =

fixes *f*::('A,'a::null)val $\Rightarrow$ ('A,'b::null)val $\Rightarrow$ ('A,'c::null)val

fixes *g*

assumes *def-scheme*[simplified]: bin *f g defined defined X Y*

assumes *def-body*: $\bigwedge x\ y. x \neq \text{bot} \implies x \neq \text{null} \implies y \neq \text{bot} \implies y \neq \text{null} \implies$

$g\ x\ y \neq \text{bot} \wedge g\ x\ y \neq \text{null}$

begin

lemma *strict4*[simp,code-unfold]: *f x null = invalid*

end

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

locale *profile-bin_d-v* =

fixes *f*::('A,'a::null)val $\Rightarrow$ ('A,'b::null)val $\Rightarrow$ ('A,'c::null)val

fixes *g*

assumes *def-scheme*[simplified]: bin *f g defined valid X Y*

assumes *def-body*: $\bigwedge x\ y. x \neq \text{bot} \implies x \neq \text{null} \implies y \neq \text{bot} \implies g\ x\ y \neq \text{bot} \wedge g\ x\ y \neq \text{null}$

sublocale *profile-bin_d-v* < *profile-bin-scheme-defined valid*

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

sublocale *profile-bin*_{StrongEq^{-v-v}} < *profile-bin-scheme valid valid f λx y. $\sqsubseteq x = y_{\sqsubseteq}$*

context *profile-bin*_{StrongEq^{-v-v}}
begin
lemma *idem[simp,code-unfold]*: $f\ null\ null = true$

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

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

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

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

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

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

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

end

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

sublocale *profile-bin*_{v-v} < *profile-bin-scheme valid valid*

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

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

overloading *StrictRefEq* $\equiv$ *StrictRefEq* :: $[('A) Boolean, ('A) Boolean] \Rightarrow ('A) Boolean$

begin
definition *StrictRefEq*_{Boolean}[*code-unfold*]:
 $(x :: ('A) Boolean) \dot{=} y \equiv \lambda\ \tau. if\ (v\ x)\ \tau = true\ \tau \wedge (v\ y)\ \tau = true\ \tau$
 $\quad then\ (x \triangleq y)\ \tau$
 $\quad else\ invalid\ \tau$

end

which implies elementary properties like:

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

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

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

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

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

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

With respect to strictness properties and miscellaneous side-calculi, strict referential equality behaves on booleans as described in the *profile-bin*_{StrongEq^{-v-v}}:

interpretation StrictRefEqBoolean : *profile-bin*_{StrongEq^{-v-v}} λ x y. (x::('A)Boolean) $\doteq$ y

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

lemma (invalid $\doteq$ false) = invalid

lemma (invalid $\doteq$ true) = invalid

lemma (false $\doteq$ invalid) = invalid

lemma (true $\doteq$ invalid) = invalid

lemma ((invalid::('A)Boolean) $\doteq$ invalid) = invalid

Thus, the weak equality is *not* reflexive.

2.2.5. Test Statements on Boolean Operations.

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

Elementary computations on Boolean

Assert $\tau \models v(\text{true})$

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

Assert $\tau \not\models \delta(\text{null})$

Assert $\tau \not\models \delta(\text{invalid})$

Assert $\tau \models v((\text{null}::('A)\text{Boolean}))$

Assert $\tau \not\models v(\text{invalid})$

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

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

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

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

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

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

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

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

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

2.3. Basic Type Void: Operations

This *minimal* OCL type contains only two elements: *invalid* and *null*. *Void* could initially be defined as $\langle\langle unit \rangle_{\perp}\rangle_{\perp}$, 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.

2.3.1. Fundamental Properties on Voids: Strict Equality

2.3.1.1. Definition

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

instance
end

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

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 $\forall$ *Void*-case as strict extension of the strong equality:

overloading *StrictRefEq* $\equiv StrictRefEq :: [(\forall)Void, (\forall)Void] \Rightarrow (\forall)Boolean$
begin
 definition *StrictRefEqVoid*[*code-unfold*] :
 $(x :: (\forall)Void) \doteq y \equiv \lambda \tau. \text{if } (v \ x) \ \tau = true \ \tau \wedge (v \ y) \ \tau = true \ \tau$
 $\text{then } (x \triangleq y) \ \tau$
 $\text{else } invalid \ \tau$

end

Property proof in terms of *profile-bin*_{StrongEq^{v-v}}

interpretation *StrictRefEqVoid* : *profile-bin*_{StrongEq^{v-v}} $\lambda \ x \ y. (x :: (\forall)Void) \doteq y$

2.3.2. Basic Void Constants

2.3.3. Validity and Definedness Properties

lemma $\delta(null :: (\forall)Void) = false$
lemma $v(null :: (\forall)Void) = true$

lemma [*simp,code-unfold*]: $\delta(\lambda -. Abs-Void_{base} None) = false$

lemma [*simp,code-unfold*]: $v(\lambda -. Abs-Void_{base} None) = false$

lemma [*simp,code-unfold*]: $\delta(\lambda -. Abs-Void_{base} _ None _) = false$

lemma [*simp,code-unfold*]: $v(\lambda -. Abs-Void_{base} _ None _) = true$

2.3.4. Test Statements

Assert $\tau \models ((null::('A)Void) \doteq null)$

2.4. Basic Type Integer: Operations

2.4.1. 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 $'A$ Boolean-case as strict extension of the strong equality:

overloading $StrictRefEq \equiv StrictRefEq :: [('A)Integer, ('A)Integer] \Rightarrow ('A)Boolean$

begin

definition $StrictRefEq_{Integer}[code-unfold] :$

$(x::('A)Integer) \doteq y \equiv \lambda \tau. \text{if } (v\ x) \tau = true \ \tau \wedge (v\ y) \tau = true \ \tau$
 $\text{then } (x \doteq y) \tau$
 $\text{else invalid } \tau$

end

Property proof in terms of $profile-bin_{StrongEq}^{-v-v}$

interpretation $StrictRefEq_{Integer} : profile-bin_{StrongEq}^{-v-v} \lambda x\ y. (x::('A)Integer) \doteq y$

2.4.2. Basic Integer Constants

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

definition $OclInt0 :: ('A)Integer \langle 0 \rangle$ **where** $0 = (\lambda _ . _0::int_)$

definition $OclInt1 :: ('A)Integer \langle 1 \rangle$ **where** $1 = (\lambda _ . _1::int_)$

definition $OclInt2 :: ('A)Integer \langle 2 \rangle$ **where** $2 = (\lambda _ . _2::int_)$

Etc.

2.4.3. Arithmetical Operations

2.4.3.1. Definition

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

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

definition $OclAdd_{Integer} :: ('A)Integer \Rightarrow ('A)Integer \Rightarrow ('A)Integer$ (**infix** $\langle +_{int} \rangle$ 40)

where $x +_{int} y \equiv \lambda \tau. \text{if } (\delta\ x) \tau = true \ \tau \wedge (\delta\ y) \tau = true \ \tau$

$\text{then } _ \uparrow x \ \tau \uparrow + \uparrow y \ \tau \uparrow$
 $\text{else invalid } \tau$

interpretation $OclAdd_{Integer} : profile-bin_{d-d} (+_{int}) \lambda x\ y. _ \uparrow x \ \tau \uparrow + \uparrow y \ \tau \uparrow$

definition $OclMinus_{Integer} :: ('A)Integer \Rightarrow ('A)Integer \Rightarrow ('A)Integer$ (**infix** $\langle -_{int} \rangle$ 41)

where $x \neg_{\text{int}} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \tau \wedge (\delta y) \tau = \text{true} \tau$
 then $\llbracket x \neg_{\text{int}} y \rrbracket \tau \equiv \llbracket x \neg_{\text{int}} y \rrbracket \tau$
 else invalid τ
interpretation $\text{OclMinus}_{\text{Integer}} : \text{profile-bin}_{\text{d-d}} (\neg_{\text{int}}) \lambda x y. \llbracket x \neg_{\text{int}} y \rrbracket \tau$

definition $\text{OclMult}_{\text{Integer}} :: ('A) \text{Integer} \Rightarrow ('A) \text{Integer} \Rightarrow ('A) \text{Integer} \text{ (infix } \langle *_{\text{int}} \rangle 45)$
where $x *_{\text{int}} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \tau \wedge (\delta y) \tau = \text{true} \tau$
 then $\llbracket x *_{\text{int}} y \rrbracket \tau \equiv \llbracket x *_{\text{int}} y \rrbracket \tau$
 else invalid τ
interpretation $\text{OclMult}_{\text{Integer}} : \text{profile-bin}_{\text{d-d}} \text{OclMult}_{\text{Integer}} \lambda x y. \llbracket x *_{\text{int}} y \rrbracket \tau$

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

definition $\text{OclDivision}_{\text{Integer}} :: ('A) \text{Integer} \Rightarrow ('A) \text{Integer} \Rightarrow ('A) \text{Integer} \text{ (infix } \langle \text{div}_{\text{int}} \rangle 45)$
where $x \text{div}_{\text{int}} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \tau \wedge (\delta y) \tau = \text{true} \tau$
 then if $y \tau \neq \text{OclInt0} \tau$ *then* $\llbracket x \text{div}_{\text{int}} y \rrbracket \tau \equiv \llbracket x \text{div}_{\text{int}} y \rrbracket \tau$ *else invalid* τ
 else invalid τ

definition $\text{OclModulus}_{\text{Integer}} :: ('A) \text{Integer} \Rightarrow ('A) \text{Integer} \Rightarrow ('A) \text{Integer} \text{ (infix } \langle \text{mod}_{\text{int}} \rangle 45)$
where $x \text{mod}_{\text{int}} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \tau \wedge (\delta y) \tau = \text{true} \tau$
 then if $y \tau \neq \text{OclInt0} \tau$ *then* $\llbracket x \text{mod}_{\text{int}} y \rrbracket \tau \equiv \llbracket x \text{mod}_{\text{int}} y \rrbracket \tau$ *else invalid* τ
 else invalid τ

definition $\text{OclLess}_{\text{Integer}} :: ('A) \text{Integer} \Rightarrow ('A) \text{Integer} \Rightarrow ('A) \text{Boolean} \text{ (infix } \langle <_{\text{int}} \rangle 35)$
where $x <_{\text{int}} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \tau \wedge (\delta y) \tau = \text{true} \tau$
 then $\llbracket x <_{\text{int}} y \rrbracket \tau \equiv \llbracket x <_{\text{int}} y \rrbracket \tau$
 else invalid τ

interpretation $\text{OclLess}_{\text{Integer}} : \text{profile-bin}_{\text{d-d}} (<_{\text{int}}) \lambda x y. \llbracket x <_{\text{int}} y \rrbracket \tau$

definition $\text{OclLe}_{\text{Integer}} :: ('A) \text{Integer} \Rightarrow ('A) \text{Integer} \Rightarrow ('A) \text{Boolean} \text{ (infix } \langle \leq_{\text{int}} \rangle 35)$
where $x \leq_{\text{int}} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \tau \wedge (\delta y) \tau = \text{true} \tau$
 then $\llbracket x \leq_{\text{int}} y \rrbracket \tau \equiv \llbracket x \leq_{\text{int}} y \rrbracket \tau$
 else invalid τ

interpretation $\text{OclLe}_{\text{Integer}} : \text{profile-bin}_{\text{d-d}} (\leq_{\text{int}}) \lambda x y. \llbracket x \leq_{\text{int}} y \rrbracket \tau$

2.4.3.2. Basic Properties

lemma $\text{OclAdd}_{\text{Integer-commute}}: (X +_{\text{int}} Y) = (Y +_{\text{int}} X)$

2.4.3.3. Execution with Invalid or Null or Zero as Argument

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

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

2.4.4. 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 $\tau \models \neg ((4 +_{\text{int}} (4 +_{\text{int}} 4)) <_{\text{int}} 10)$
Assert $\tau \models \text{not } (v \text{ (null } +_{\text{int}} 1))$
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 $\tau \models \text{not } (v (1 \text{ div}_{\text{int}} 0))$

lemma *integer-non-null* [simp]: $((\lambda \cdot. \perp n_{\perp}) \doteq (\text{null}::({}^b\mathbb{A})\text{Integer})) = \text{false}$

lemma *null-non-integer* [simp]: $((\text{null}::({}^b\mathbb{A})\text{Integer}) \doteq (\lambda \cdot. \perp n_{\perp})) = \text{false}$

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

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

Elementary computations on Integer

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

Assert $\tau \models 1 <> 2$

Assert $\tau \models 2 <> 1$

Assert $\tau \models 2 \doteq 2$

Assert $\tau \models v \ 4$

Assert $\tau \models \delta \ 4$

Assert $\tau \models v \ (\text{null}::({}^b\mathbb{A})\text{Integer})$

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

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

Assert $\tau \models (4 \triangleq 4)$

Assert $\tau \models \neg (9 \triangleq 10)$

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

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

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

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

Assert $\tau \not\models (invalid \triangleleft (invalid::('A)Integer))$
Assert $\tau \not\models v \ (invalid \triangleleft (invalid::('A)Integer))$
Assert $\tau \models (null \doteq (null::('A)Integer))$
Assert $\tau \models (null \doteq (null::('A)Integer))$
Assert $\tau \models (4 \doteq 4)$
Assert $\tau \not\models (4 \triangleleft 4)$
Assert $\tau \not\models (4 \doteq 10)$
Assert $\tau \models (4 \triangleleft 10)$
Assert $\tau \not\models (0 <_{int} null)$
Assert $\tau \not\models (\delta \ (0 <_{int} null))$

2.5. Basic Type Real: Operations

2.5.1. 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 $'A$ Boolean-case as strict extension of the strong equality:

overloading $StrictRefEq \equiv StrictRefEq :: [('A)Real, ('A)Real] \Rightarrow ('A)Boolean$
begin
definition $StrictRefEq_{Real} \ [code-unfold] :$
 $(x::('A)Real) \doteq y \equiv \lambda \tau. \text{ if } (v \ x) \ \tau = true \ \tau \wedge (v \ y) \ \tau = true \ \tau$
 $\quad \text{then } (x \triangleq y) \ \tau$
 $\quad \text{else } invalid \ \tau$
end

Property proof in terms of $profile-bin_{StrongEq}^{-v-v}$

interpretation $StrictRefEq_{Real} : profile-bin_{StrongEq}^{-v-v} \lambda \ x \ y. (x::('A)Real) \doteq y$

2.5.2. Basic Real Constants

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

definition $OclReal0 :: ('A)Real \ (\langle 0.0 \rangle) \ \text{where} \ 0.0 = (\lambda _ . _ \ 0::real_{\perp})$
definition $OclReal1 :: ('A)Real \ (\langle 1.0 \rangle) \ \text{where} \ 1.0 = (\lambda _ . _ \ 1::real_{\perp})$
definition $OclReal2 :: ('A)Real \ (\langle 2.0 \rangle) \ \text{where} \ 2.0 = (\lambda _ . _ \ 2::real_{\perp})$

Etc.

2.5.3. Arithmetical Operations

2.5.3.1. Definition

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

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

definition $OclAdd_{\text{Real}} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Real$ (**infix** $\langle +_{\text{real}} \rangle$ 40)

where $x +_{\text{real}} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \tau \wedge (\delta y) \tau = \text{true} \tau$

then $\llbracket x \tau \rrbracket + \llbracket y \tau \rrbracket$

else invalid τ

interpretation $OclAdd_{\text{Real}} : \text{profile-bin}_{\text{d-d}} (+_{\text{real}}) \lambda x y. \llbracket x \tau \rrbracket + \llbracket y \tau \rrbracket$

definition $OclMinus_{\text{Real}} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Real$ (**infix** $\langle -_{\text{real}} \rangle$ 41)

where $x -_{\text{real}} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \tau \wedge (\delta y) \tau = \text{true} \tau$

then $\llbracket x \tau \rrbracket - \llbracket y \tau \rrbracket$

else invalid τ

interpretation $OclMinus_{\text{Real}} : \text{profile-bin}_{\text{d-d}} (-_{\text{real}}) \lambda x y. \llbracket x \tau \rrbracket - \llbracket y \tau \rrbracket$

definition $OclMult_{\text{Real}} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Real$ (**infix** $\langle *_{\text{real}} \rangle$ 45)

where $x *_{\text{real}} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \tau \wedge (\delta y) \tau = \text{true} \tau$

then $\llbracket x \tau \rrbracket * \llbracket y \tau \rrbracket$

else invalid τ

interpretation $OclMult_{\text{Real}} : \text{profile-bin}_{\text{d-d}} OclMult_{\text{Real}} \lambda x y. \llbracket x \tau \rrbracket * \llbracket y \tau \rrbracket$

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

definition $OclDivision_{\text{Real}} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Real$ (**infix** $\langle \text{div}_{\text{real}} \rangle$ 45)

where $x \text{div}_{\text{real}} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \tau \wedge (\delta y) \tau = \text{true} \tau$

then if $y \tau \neq OclReal0 \tau$ *then* $\llbracket x \tau \rrbracket / \llbracket y \tau \rrbracket$ *else invalid* τ

else invalid τ

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

definition $OclModulus_{\text{Real}} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Real$ (**infix** $\langle \text{mod}_{\text{real}} \rangle$ 45)

where $x \text{mod}_{\text{real}} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \tau \wedge (\delta y) \tau = \text{true} \tau$

then if $y \tau \neq OclReal0 \tau$ *then* $\llbracket \text{mod-float } \llbracket x \tau \rrbracket \llbracket y \tau \rrbracket \rrbracket$ *else invalid* τ

else invalid τ

definition $OclLess_{\text{Real}} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Boolean$ (**infix** $\langle <_{\text{real}} \rangle$ 35)

where $x <_{\text{real}} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \tau \wedge (\delta y) \tau = \text{true} \tau$

then $\llbracket x \tau \rrbracket < \llbracket y \tau \rrbracket$

else invalid τ

interpretation $OclLess_{\text{Real}} : \text{profile-bin}_{\text{d-d}} (<_{\text{real}}) \lambda x y. \llbracket x \tau \rrbracket < \llbracket y \tau \rrbracket$

definition $OclLe_{\text{Real}} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Boolean$ (**infix** $\langle \leq_{\text{real}} \rangle$ 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 $\llbracket x \tau \rrbracket \leq \llbracket y \tau \rrbracket$

else invalid τ

interpretation $OclLe_{\text{Real}} : \text{profile-bin}_{\text{d-d}} (\leq_{\text{real}}) \lambda x y. \llbracket x \tau \rrbracket \leq \llbracket y \tau \rrbracket$

2.5.3.2. Basic Properties

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

2.5.3.3. Execution with Invalid or Null or Zero as Argument

lemma *OclAddReal-zero1*[simp,code-unfold] :

$(x +_{\text{real}} \mathbf{0.0}) = (\text{if } \forall x \text{ and not } (\delta x) \text{ then invalid else } x \text{ endif})$

lemma *OclAddReal-zero2*[simp,code-unfold] :

$(\mathbf{0.0} +_{\text{real}} x) = (\text{if } \forall x \text{ and not } (\delta x) \text{ then invalid else } x \text{ endif})$

2.5.4. Test Statements

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

Assert $\tau \models (\mathbf{9.0} \leq_{\text{real}} \mathbf{10.0})$

Assert $\tau \models ((\mathbf{4.0} +_{\text{real}} \mathbf{4.0}) \leq_{\text{real}} \mathbf{10.0})$

Assert $\tau \not\models ((\mathbf{4.0} +_{\text{real}} (\mathbf{4.0} +_{\text{real}} \mathbf{4.0})) <_{\text{real}} \mathbf{10.0})$

Assert $\tau \models \text{not } (\forall (\text{null} +_{\text{real}} \mathbf{1.0}))$

Assert $\tau \models (((\mathbf{9.0} *_{\text{real}} \mathbf{4.0}) \text{div}_{\text{real}} \mathbf{10.0}) \leq_{\text{real}} \mathbf{4.0})$

Assert $\tau \models \text{not } (\delta (\mathbf{1.0} \text{div}_{\text{real}} \mathbf{0.0}))$

Assert $\tau \models \text{not } (\forall (\mathbf{1.0} \text{div}_{\text{real}} \mathbf{0.0}))$

lemma *real-non-null* [simp]: $((\lambda -. _ \sqsubseteq n_{\perp}) \doteq (\text{null}::({}^{\mathcal{A}}\text{Real})) = \text{false}$

lemma *null-non-real* [simp]: $((\text{null}::({}^{\mathcal{A}}\text{Real}) \doteq (\lambda -. _ \sqsubseteq n_{\perp})) = \text{false}$

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

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

lemma *OclReal1-non-null* [simp,code-unfold]: $(\mathbf{1.0} \doteq \text{null}) = \text{false}$

lemma *null-non-OclReal1* [simp,code-unfold]: $(\text{null} \doteq \mathbf{1.0}) = \text{false}$

lemma *OclReal2-non-null* [simp,code-unfold]: $(\mathbf{2.0} \doteq \text{null}) = \text{false}$

lemma *null-non-OclReal2* [simp,code-unfold]: $(\text{null} \doteq \mathbf{2.0}) = \text{false}$

lemma *OclReal6-non-null* [simp,code-unfold]: $(\mathbf{6.0} \doteq \text{null}) = \text{false}$

lemma *null-non-OclReal6* [simp,code-unfold]: $(\text{null} \doteq \mathbf{6.0}) = \text{false}$

lemma *OclReal8-non-null* [simp,code-unfold]: $(\mathbf{8.0} \doteq \text{null}) = \text{false}$

lemma *null-non-OclReal8* [simp,code-unfold]: $(\text{null} \doteq \mathbf{8.0}) = \text{false}$

lemma *OclReal9-non-null* [simp,code-unfold]: $(\mathbf{9.0} \doteq \text{null}) = \text{false}$

lemma *null-non-OclReal9* [simp,code-unfold]: $(\text{null} \doteq \mathbf{9.0}) = \text{false}$

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

Elementary computations on Real

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

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

Assert $\tau \models \mathbf{2.0} \doteq \mathbf{2.0}$

Assert $\tau \models \forall \mathbf{4.0}$

Assert $\tau \models \delta \mathbf{4.0}$

Assert $\tau \models \forall (\text{null}::({}^{\mathcal{A}}\text{Real}))$

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

Assert $\tau \models (\text{null} \triangleq \text{null})$
Assert $\tau \models (4.0 \triangleq 4.0)$
Assert $\tau \not\models (9.0 \triangleq 10.0)$
Assert $\tau \not\models (\text{invalid} \triangleq 10.0)$
Assert $\tau \not\models (\text{null} \triangleq 10.0)$
Assert $\tau \not\models (\text{invalid} \doteq (\text{invalid}::(^{\mathcal{A}})\text{Real}))$
Assert $\tau \not\models v \text{ } (\text{invalid} \doteq (\text{invalid}::(^{\mathcal{A}})\text{Real}))$
Assert $\tau \not\models (\text{invalid} \triangleleft (\text{invalid}::(^{\mathcal{A}})\text{Real}))$
Assert $\tau \not\models v \text{ } (\text{invalid} \triangleleft (\text{invalid}::(^{\mathcal{A}})\text{Real}))$
Assert $\tau \models (\text{null} \doteq (\text{null}::(^{\mathcal{A}})\text{Real}))$
Assert $\tau \models (\text{null} \doteq (\text{null}::(^{\mathcal{A}})\text{Real}))$
Assert $\tau \models (4.0 \doteq 4.0)$
Assert $\tau \not\models (4.0 \triangleleft 4.0)$
Assert $\tau \not\models (4.0 \doteq 10.0)$
Assert $\tau \models (4.0 \triangleleft 10.0)$
Assert $\tau \not\models (0.0 <_{\text{real}} \text{null})$
Assert $\tau \not\models (\delta \text{ } (0.0 <_{\text{real}} \text{null}))$

2.6. Basic Type String: Operations

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

overloading $\text{StrictRefEq} \equiv \text{StrictRefEq} :: [(^{\mathcal{A}})\text{String}, (^{\mathcal{A}})\text{String}] \Rightarrow (^{\mathcal{A}})\text{Boolean}$

begin

definition $\text{StrictRefEq}_{\text{String}}[\text{code-unfold}] :$

$$(x::(^{\mathcal{A}})\text{String}) \doteq y \equiv \lambda \tau. \text{if } (\forall x) \tau = \text{true} \wedge (\forall y) \tau = \text{true} \tau \text{ then } (x \triangleq y) \tau \text{ else } \text{invalid } \tau$$

end

Property proof in terms of $\text{profile-bin}_{\text{StrongEq}^{-\mathcal{V}}-\mathcal{V}}$

interpretation $\text{StrictRefEq}_{\text{String}} : \text{profile-bin}_{\text{StrongEq}^{-\mathcal{V}}-\mathcal{V}} \lambda x y. (x::(^{\mathcal{A}})\text{String}) \doteq y$

2.6.2. Basic String Constants

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

definition $\text{OclStringa} :: (^{\mathcal{A}})\text{String} \langle \mathbf{a} \rangle$ **where** $\mathbf{a} = (\lambda _ . _ \text{ } \underline{\underline{a}})$

definition $\text{OclStringb} :: (^{\mathcal{A}})\text{String} \langle \mathbf{b} \rangle$ **where** $\mathbf{b} = (\lambda _ . _ \text{ } \underline{\underline{b}})$

definition $\text{OclStringc} :: (^{\mathcal{A}})\text{String} \langle \mathbf{c} \rangle$ **where** $\mathbf{c} = (\lambda _ . _ \text{ } \underline{\underline{c}})$

Etc.

Etc.

2.6.3. String Operations

2.6.3.1. Definition

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

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

definition $OclAdd_{String} :: ('A)String \Rightarrow ('A)String \Rightarrow ('A)String$ (**infix** $\langle +_{string} \rangle$ 40)

where $x +_{string} y \equiv \lambda \tau. \text{if } (\delta x) \tau = true \wedge (\delta y) \tau = true$
then $\llbracket concat [\ulcorner x \urcorner, \ulcorner y \urcorner] \rrbracket$
else $invalid \tau$

interpretation $OclAdd_{String} : profile_bind_d (+_{string}) \lambda x y. \llbracket concat [\ulcorner x \urcorner, \ulcorner y \urcorner] \rrbracket$

2.6.3.2. Basic Properties

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

2.6.4. Test Statements

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

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

Elementary computations on String

Assert $\tau \models a \triangleleft b$

Assert $\tau \models b \triangleleft a$

Assert $\tau \models b \doteq b$

Assert $\tau \models v a$

Assert $\tau \models \delta a$

Assert $\tau \models v (null :: ('A)String)$

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

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

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

Assert $\tau \models \not\models (a \triangleq b)$

Assert $\tau \models \not\models (invalid \triangleq b)$

Assert $\tau \models \not\models (null \triangleq b)$

Assert $\tau \models \not\models (invalid \doteq (invalid :: ('A)String))$

Assert $\tau \models \not\models v (invalid \doteq (invalid :: ('A)String))$

Assert $\tau \models \not\models (invalid \triangleleft (invalid :: ('A)String))$

Assert $\tau \models \not\models v (invalid \triangleleft (invalid :: ('A)String))$

Assert $\tau \models (null \doteq (null :: ('A)String))$

Assert $\tau \models (null \doteq (null :: ('A)String))$

Assert $\tau \models (b \doteq b)$

Assert $\tau \models \not\models (b \triangleleft b)$

Assert $\tau \models \not\models (b \doteq c)$

Assert $\tau \models (b \triangleleft c)$

2.7. Collection Type Pairs: Operations

The OCL standard provides the concept of *Tuples*, i. e. a family of record-types with projection functions. In 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 mimick all tuples. In particular, it can be used to express operations with multiple arguments, roles of n-ary associations, ...

2.7.1. Semantic Properties of the Type Constructor

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

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

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

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

2.7.2. 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:

overloading

$StrictRefEq \equiv StrictRefEq :: [(\alpha, \alpha::null, \beta::null)Pair, (\alpha, \alpha::null, \beta::null)Pair] \Rightarrow (\alpha)Boolean$

begin

definition $StrictRefEq_{Pair} :$

$((x::(\alpha, \alpha::null, \beta::null)Pair) \doteq y) \equiv (\lambda \tau. \text{if } (v\ x) \tau = true \ \tau \wedge (v\ y) \tau = true \ \tau$
 $\text{then } (x \triangleq y) \tau$
 $\text{else invalid } \tau)$

end

Property proof in terms of $profile-bin_{StrongEq}^{-v-v}$

interpretation $StrictRefEq_{Pair} : profile-bin_{StrongEq}^{-v-v} \lambda x\ y. (x::(\alpha, \alpha::null, \beta::null)Pair) \doteq y$

2.7.3. Standard Operations Definitions

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

2.7.3.1. Definition: Pair Constructor

definition $OclPair::(\alpha, \alpha) val \Rightarrow$

$(\alpha, \beta) val \Rightarrow$

$(\alpha, \alpha::null, \beta::null) Pair \ (\langle Pair\{(-),(-)\} \rangle)$

where $Pair\{X,Y\} \equiv (\lambda \tau. \text{if } (v\ X) \tau = true \ \tau \wedge (v\ Y) \tau = true \ \tau$

then $Abs\text{-}Pair_{base} \sqcup (X \ \tau, Y \ \tau) \sqcup$
 else $invalid \ \tau$)

interpretation $OclPair : profile\text{-}bin_{\sim v}$
 $OclPair \ \lambda x y. Abs\text{-}Pair_{base} \sqcup (x, y) \sqcup$

2.7.3.2. Definition: First

definition $OclFirst :: ('A, 'a :: null, 'b :: null) \text{Pair} \Rightarrow ('A, 'a) \text{val} \ (\leftarrow \cdot \text{First}'('))$
where $X \text{.First}() \equiv (\lambda \ \tau. \text{if } (\delta \ X) \ \tau = \text{true} \ \tau$
 then $\text{fst } {}^{\top}Rep\text{-}Pair_{base} (X \ \tau)^{\top}$
 else $invalid \ \tau$)

interpretation $OclFirst : profile\text{-}mono_d \ OclFirst \ \lambda x. \text{fst } {}^{\top}Rep\text{-}Pair_{base} (x)^{\top}$

2.7.3.3. Definition: Second

definition $OclSecond :: ('A, 'a :: null, 'b :: null) \text{Pair} \Rightarrow ('A, 'b) \text{val} \ (\leftarrow \cdot \text{Second}'('))$
where $X \text{.Second}() \equiv (\lambda \ \tau. \text{if } (\delta \ X) \ \tau = \text{true} \ \tau$
 then $\text{snd } {}^{\top}Rep\text{-}Pair_{base} (X \ \tau)^{\top}$
 else $invalid \ \tau$)

interpretation $OclSecond : profile\text{-}mono_d \ OclSecond \ \lambda x. \text{snd } {}^{\top}Rep\text{-}Pair_{base} (x)^{\top}$

2.7.4. Logical Properties

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

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

2.7.5. Algebraic Execution Properties

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

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

2.7.6. Test Statements

Assert $\tau \models invalid \ \text{.First}() \triangleq invalid$

Assert $\tau \models null \ \text{.First}() \triangleq invalid$

Assert $\tau \models null \ \text{.Second}() \triangleq invalid \ \text{.Second}()$

Assert $\tau \models Pair\{invalid, true\} \triangleq invalid$

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

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

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

no-notation $None (\langle \perp \rangle)$

2.8. Collection Type Bag: Operations

definition $Rep\text{-}Bag\text{-}base' x = \{(x0, y). y < \ulcorner Rep\text{-}Bag_{base} x \urcorner x0\}$

definition $Rep\text{-}Bag\text{-}base x \tau = \{(x0, y). y < \ulcorner Rep\text{-}Bag_{base} (x \tau) \urcorner x0\}$

definition $Rep\text{-}Set\text{-}base x \tau = fst' \{(x0, y). y < \ulcorner Rep\text{-}Bag_{base} (x \tau) \urcorner x0\}$

definition $ApproxEq$ (infixl $\langle \approx \rangle$ 30)

where $X \approx Y \equiv \lambda \tau. \ulcorner Rep\text{-}Set\text{-}base X \tau = Rep\text{-}Set\text{-}base Y \tau \urcorner$

2.8.1. As a Motivation for the (infinite) Type Construction: Type-Extensions as Bags

Our notion of typed bag goes beyond the usual notion of a finite executable bag and is powerful enough to capture *the extension of a type* in UML and OCL. This means we can have in Featherweight OCL Bags 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 bag of all *defined* values of a type T (for which we will introduce the constant T)
2. the bag of all *valid* values of a type T , so including *null* (for which we will introduce the constant T_{null}).

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

definition $Integer :: (\mathcal{A}, Integer_{base}) Bag$

where $Integer \equiv (\lambda \tau. (Abs\text{-}Bag_{base} o Some o Some) (\lambda None \Rightarrow 0 \mid Some None \Rightarrow 0 \mid - \Rightarrow I))$

definition $Integer_{null} :: (\mathcal{A}, Integer_{base}) Bag$

where $Integer_{null} \equiv (\lambda \tau. (Abs\text{-}Bag_{base} o Some o Some) (\lambda None \Rightarrow 0 \mid - \Rightarrow I))$

lemma $Integer\text{-}defined : \delta Integer = true$

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

This allows the theorems:

$\tau \models \delta x \implies \tau \models (Integer \text{--} \> includes_{Bag}(x)) \tau \models \delta x \implies \tau \models Integer \triangleq (Integer \text{--} \> including_{Bag}(x))$

and

$\tau \models \nu x \implies \tau \models (Integer_{null} \text{--} \> includes_{Bag}(x)) \tau \models \nu x \implies \tau \models Integer_{null} \triangleq (Integer_{null} \text{--} \> including_{Bag}(x))$

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

In the same spirit, we proceed similarly for the remaining base types:

definition $Void_{null} :: (\mathcal{A}, Void_{base}) Bag$

where $Void_{null} \equiv (\lambda \tau. (Abs\text{-}Bag_{base} o Some o Some) (\lambda x. if x = Abs\text{-}Void_{base} (Some None) then 1 else 0))$

definition $Void_{empty} :: (\mathcal{A}, Void_{base}) Bag$

where $Void_{empty} \equiv (\lambda \tau. (Abs-Bag_{base} \circ Some \circ Some) (\lambda \cdot. 0))$

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

lemma $Void_{empty}-defined : \delta Void_{empty} = true$

lemma assumes $\tau \models \delta (V :: (^{\mathcal{A}}, Void_{base}) Bag)$

shows $\tau \models V \cong Void_{null} \vee \tau \models V \cong Void_{empty}$

definition $Boolean :: (^{\mathcal{A}}, Boolean_{base}) Bag$

where $Boolean \equiv (\lambda \tau. (Abs-Bag_{base} \circ Some \circ Some) (\lambda None \Rightarrow 0 \mid Some None \Rightarrow 0 \mid - \Rightarrow I))$

definition $Boolean_{null} :: (^{\mathcal{A}}, Boolean_{base}) Bag$

where $Boolean_{null} \equiv (\lambda \tau. (Abs-Bag_{base} \circ Some \circ Some) (\lambda None \Rightarrow 0 \mid - \Rightarrow I))$

lemma $Boolean-defined : \delta Boolean = true$

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

definition $String :: (^{\mathcal{A}}, String_{base}) Bag$

where $String \equiv (\lambda \tau. (Abs-Bag_{base} \circ Some \circ Some) (\lambda None \Rightarrow 0 \mid Some None \Rightarrow 0 \mid - \Rightarrow I))$

definition $String_{null} :: (^{\mathcal{A}}, String_{base}) Bag$

where $String_{null} \equiv (\lambda \tau. (Abs-Bag_{base} \circ Some \circ Some) (\lambda None \Rightarrow 0 \mid - \Rightarrow I))$

lemma $String-defined : \delta String = true$

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

definition $Real :: (^{\mathcal{A}}, Real_{base}) Bag$

where $Real \equiv (\lambda \tau. (Abs-Bag_{base} \circ Some \circ Some) (\lambda None \Rightarrow 0 \mid Some None \Rightarrow 0 \mid - \Rightarrow I))$

definition $Real_{null} :: (^{\mathcal{A}}, Real_{base}) Bag$

where $Real_{null} \equiv (\lambda \tau. (Abs-Bag_{base} \circ Some \circ Some) (\lambda None \Rightarrow 0 \mid - \Rightarrow I))$

lemma $Real-defined : \delta Real = true$

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

2.8.2. Basic Properties of the Bag Type

Every element in a defined bag is valid.

lemma $Bag-inv-lemma: \tau \models (\delta X) \implies \ulcorner Rep-Bag_{base} (X \tau) \urcorner bot = 0$

lemma $Bag-inv-lemma'$:

assumes $x-def : \tau \models \delta X$

and $e-mem : \ulcorner Rep-Bag_{base} (X \tau) \urcorner e \geq 1$

shows $\tau \models v (\lambda \cdot. e)$

lemma $abs-rep-simp'$:

assumes $S-all-def : \tau \models \delta S$

shows $Abs-Bag_{base} \sqsubseteq^{\top} Rep-Bag_{base} (S \tau)^{\top} \sqsubseteq S \tau$

lemma *invalid-bag-OclNot-defined* [simp,code-unfold]: $\delta(invalid::('A, 'A::null) Bag) = false$

lemma *null-bag-OclNot-defined* [simp,code-unfold]: $\delta(null::('A, 'A::null) Bag) = false$

lemma *invalid-bag-valid* [simp,code-unfold]: $v(invalid::('A, 'A::null) Bag) = false$

lemma *null-bag-valid* [simp,code-unfold]: $v(null::('A, 'A::null) Bag) = true$

... which means that we can have a type $('A, ('A, ('A) Integer) Bag) Bag$ corresponding exactly to $Bag(Bag(Integer))$ 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.

2.8.3. Definition: Strict Equality

After the part of foundational operations on bags, we detail here equality on bags. 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:

overloading *StrictRefEq* $\equiv StrictRefEq :: [('A, 'A::null) Bag, ('A, 'A::null) Bag] \Rightarrow ('A) Boolean$

begin

definition *StrictRefEqBag* :

$(x::('A, 'A::null) Bag) \doteq y \equiv \lambda \tau. \text{if } (v x) \tau = true \wedge (v y) \tau = true \wedge$
 $\text{then } (x \triangleq y) \tau$
 $\text{else invalid } \tau$

end

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

Property proof in terms of *profile-bin*_{StrongEq^{-v-v}}

interpretation *StrictRefEqBag* : *profile-bin*_{StrongEq^{-v-v}} $\lambda x y. (x::('A, 'A::null) Bag) \doteq y$

2.8.4. Constants: mtBag

definition *mtBag*: $(('A, 'A::null) Bag) (Bag\{\})$

where $Bag\{\} \equiv (\lambda \tau. Abs-Bag_{base} \sqsubseteq \lambda -. 0::nat_{\perp})$

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

lemma *mtBag-valid* [simp,code-unfold]: $v(Bag\{\}) = true$

lemma *mtBag-rep-bag*: $\top Rep-Bag_{base} (Bag\{\} \tau)^{\top} = (\lambda -. 0)$

2.8.5. Definition: Including

definition *OclIncluding* :: $[('A, 'A::null) Bag, ('A, 'A) val] \Rightarrow ('A, 'A) Bag$

where $OclIncluding\ x\ y = (\lambda\ \tau. \text{if } (\delta\ x)\ \tau = \text{true}\ \tau \wedge (\nu\ y)\ \tau = \text{true}\ \tau$
 $\text{then } Abs-Bag_{base}\ \perp \sqcup \ulcorner Rep-Bag_{base}(x\ \tau) \urcorner$
 $((y\ \tau) := \ulcorner Rep-Bag_{base}(x\ \tau) \urcorner (y\ \tau) + 1)$
 $\text{else } \text{invalid}\ \tau)$

notation $OclIncluding\ (\leftarrow\rightarrow\text{including}_{Bag}\ '(-)')$

interpretation $OclIncluding : \text{profile-bin}_d\text{-}\nu\ OclIncluding\ \lambda x\ y. Abs-Bag_{base}\ \perp \sqcup \ulcorner Rep-Bag_{base}\ x \urcorner$
 $(y := \ulcorner Rep-Bag_{base}\ x \urcorner y + 1) \sqcup$

syntax

- $OclFinbag :: \text{args} \Rightarrow ('A, 'a :: \text{null})\ Bag\ (\langle Bag\{(-)\} \rangle)$

syntax-consts

- $OclFinbag == OclIncluding$

translations

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

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

2.8.6. Definition: Excluding

definition $OclExcluding :: [('A, 'a :: \text{null})\ Bag, ('A, 'a)\ \text{val}] \Rightarrow ('A, 'a)\ Bag$

where $OclExcluding\ x\ y = (\lambda\ \tau. \text{if } (\delta\ x)\ \tau = \text{true}\ \tau \wedge (\nu\ y)\ \tau = \text{true}\ \tau$
 $\text{then } Abs-Bag_{base}\ \perp \sqcup \ulcorner Rep-Bag_{base}(x\ \tau) \urcorner ((y\ \tau) := 0 :: \text{nat}) \sqcup$
 $\text{else } \text{invalid}\ \tau)$

notation $OclExcluding\ (\leftarrow\rightarrow\text{excluding}_{Bag}\ '(-)')$

interpretation $OclExcluding : \text{profile-bin}_d\text{-}\nu\ OclExcluding$

$\lambda x\ y. Abs-Bag_{base}\ \perp \sqcup \ulcorner Rep-Bag_{base}(x) \urcorner (y := 0 :: \text{nat}) \sqcup$

2.8.7. Definition: Includes

definition $OclIncludes :: [('A, 'a :: \text{null})\ Bag, ('A, 'a)\ \text{val}] \Rightarrow 'A\ \text{Boolean}$

where $OclIncludes\ x\ y = (\lambda\ \tau. \text{if } (\delta\ x)\ \tau = \text{true}\ \tau \wedge (\nu\ y)\ \tau = \text{true}\ \tau$
 $\text{then } \perp \sqcup \ulcorner Rep-Bag_{base}(x\ \tau) \urcorner (y\ \tau) > 0 \sqcup$
 $\text{else } \perp)$

notation $OclIncludes\ (\leftarrow\rightarrow\text{includes}_{Bag}\ '(-)')$

interpretation $OclIncludes : \text{profile-bin}_d\text{-}\nu\ OclIncludes\ \lambda x\ y. \perp \sqcup \ulcorner Rep-Bag_{base}\ x \urcorner y > 0 \sqcup$

2.8.8. Definition: Excludes

definition $OclExcludes :: [('A, 'a :: \text{null})\ Bag, ('A, 'a)\ \text{val}] \Rightarrow 'A\ \text{Boolean}$

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

notation $OclExcludes\ (\leftarrow\rightarrow\text{excludes}_{Bag}\ '(-)')$

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

interpretation $OclExcludes : \text{profile-bin}_d\text{-}\nu\ OclExcludes\ \lambda x\ y. \perp \sqcup \ulcorner Rep-Bag_{base}\ x \urcorner y \leq 0 \sqcup$

2.8.9. Definition: Size

definition $OclSize :: (^{\mathcal{A}}, ' \alpha :: null) Bag \Rightarrow ^{\mathcal{A}} Integer$
where $OclSize x = (\lambda \tau. \text{if } (\delta x) \tau = true \ \tau \wedge finite (Rep-Bag-base \ x \ \tau) \\ \text{then } \perp \text{int } (card (Rep-Bag-base \ x \ \tau)) \perp \\ \text{else } \perp)$
notation $OclSize \quad (\leftarrow size_{Bag} '(\cdot))$

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

2.8.10. Definition: IsEmpty

definition $OclIsEmpty :: (^{\mathcal{A}}, ' \alpha :: null) Bag \Rightarrow ^{\mathcal{A}} Boolean$
where $OclIsEmpty x = ((\forall x \text{ and not } (\delta x)) \text{ or } ((OclSize \ x) \doteq 0))$
notation $OclIsEmpty \quad (\leftarrow isEmpty_{Bag} '(\cdot))$

2.8.11. Definition: NotEmpty

definition $OclNotEmpty :: (^{\mathcal{A}}, ' \alpha :: null) Bag \Rightarrow ^{\mathcal{A}} Boolean$
where $OclNotEmpty x = not(OclIsEmpty \ x)$
notation $OclNotEmpty \quad (\leftarrow notEmpty_{Bag} '(\cdot))$

2.8.12. Definition: Any

definition $OclANY :: [(^{\mathcal{A}}, ' \alpha :: null) Bag] \Rightarrow (^{\mathcal{A}}, ' \alpha) val$
where $OclANY x = (\lambda \tau. \text{if } (\forall x) \tau = true \ \tau \\ \text{then if } (\delta x \text{ and } OclNotEmpty \ x) \tau = true \ \tau \\ \text{then } SOME \ y. y \in (Rep-Set-base \ x \ \tau) \\ \text{else null } \tau \\ \text{else } \perp)$
notation $OclANY \quad (\leftarrow any_{Bag} '(\cdot))$

2.8.13. Definition: Forall

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

definition $OclForall :: [(^{\mathcal{A}}, ' \alpha :: null) Bag, (^{\mathcal{A}}, ' \alpha) val \Rightarrow (^{\mathcal{A}}) Boolean] \Rightarrow ^{\mathcal{A}} Boolean$
where $OclForall \ S \ P = (\lambda \tau. \text{if } (\delta \ S) \tau = true \ \tau \\ \text{then if } (\exists x \in Rep-Set-base \ S \ \tau. P \ (\lambda -. x) \ \tau) \tau = false \ \tau \\ \text{then false } \tau \\ \text{else if } (\exists x \in Rep-Set-base \ S \ \tau. P \ (\lambda -. x) \ \tau) \tau = invalid \ \tau \\ \text{then invalid } \tau \\ \text{else if } (\exists x \in Rep-Set-base \ S \ \tau. P \ (\lambda -. x) \ \tau) \tau = null \ \tau \\ \text{then null } \tau \\ \text{else true } \tau \\ \text{else } \perp)$

syntax
 $-OclForallBag :: [(^{\mathcal{A}}, ' \alpha :: null) Bag, id, (^{\mathcal{A}}) Boolean] \Rightarrow ^{\mathcal{A}} Boolean \quad (\leftarrow (-) \rightarrow forAll_{Bag} '(-))$

syntax-consts
 $-OclForallBag == UML-Bag.OclForall$

translations

$$X \rightarrow \text{forAll}_{\text{Bag}}(x \mid P) == \text{CONST UML-Bag.OclForall } X (\%x. P)$$

2.8.14. Definition: Exists

Like OclForall, OclExists is also not strict.

definition $\text{OclExists} :: [(\text{'A}, \text{'A}::\text{null}) \text{Bag}, (\text{'A}, \text{'A})\text{val} \Rightarrow (\text{'A})\text{Boolean}] \Rightarrow \text{'A Boolean}$

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

syntax

$$\text{-OclExistBag} :: [(\text{'A}, \text{'A}::\text{null}) \text{Bag}, \text{id}, (\text{'A})\text{Boolean}] \Rightarrow \text{'A Boolean} \quad (\langle (-) \rangle \rightarrow \text{exists}_{\text{Bag}} '(-) \rangle)$$

syntax-consts

$$\text{-OclExistBag} == \text{UML-Bag.OclExists}$$

translations

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

2.8.15. Definition: Iterate

definition $\text{OclIterate} :: [(\text{'A}, \text{'A}::\text{null}) \text{Bag}, (\text{'A}, \text{'B}::\text{null})\text{val},$

$$(\text{'A}, \text{'A})\text{val} \Rightarrow (\text{'A}, \text{'B})\text{val} \Rightarrow (\text{'A}, \text{'B})\text{val}] \Rightarrow (\text{'A}, \text{'B})\text{val}$$

where $\text{OclIterate } S A F = (\lambda \tau. \text{if } (\delta S) \tau = \text{true } \tau \wedge (\forall A) \tau = \text{true } \tau \wedge \text{finite } (\text{Rep-Bag-base } S \tau) \\ \text{then Finite-Set.fold } (F \circ (\lambda a \tau. a) \circ \text{fst}) A (\text{Rep-Bag-base } S \tau) \tau \\ \text{else } \perp)$

syntax

$$\text{-OclIterateBag} :: [(\text{'A}, \text{'A}::\text{null}) \text{Bag}, \text{id}, \text{id}, \text{'A}, \text{'B}] \Rightarrow (\text{'A}, \text{'B})\text{val} \\ (\langle (-) \rangle \rightarrow \text{iterate}_{\text{Bag}} '(-) \rangle)$$

syntax-consts

$$\text{-OclIterateBag} == \text{OclIterate}$$

translations

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

2.8.16. Definition: Select

definition $\text{OclSelect} :: [(\text{'A}, \text{'A}::\text{null})\text{Bag}, (\text{'A}, \text{'A})\text{val} \Rightarrow (\text{'A})\text{Boolean}] \Rightarrow (\text{'A}, \text{'A})\text{Bag}$

where $\text{OclSelect } S P = (\lambda \tau. \text{if } (\delta S) \tau = \text{true } \tau \\ \text{then if } (\exists x \in \text{Rep-Set-base } S \tau. P(\lambda -. x) \tau = \text{invalid } \tau) \\ \text{then invalid } \tau \\ \text{else Abs-Bag}_{\text{base}} \perp \lambda x. \\ \text{let } n = \sqcap \text{Rep-Bag}_{\text{base}} (S \tau) \sqcap x \text{ in} \\ \text{if } n = 0 \mid P(\lambda -. x) \tau = \text{false } \tau \text{ then} \\ 0 \\ \text{else} \\ n \sqcup \\ \text{else invalid } \tau)$

syntax

$$\text{-OclSelectBag} :: [(\text{'A}, \text{'A}::\text{null}) \text{Bag}, \text{id}, (\text{'A})\text{Boolean}] \Rightarrow \text{'A Boolean} \quad (\langle (-) \rangle \rightarrow \text{select}_{\text{Bag}} '(-) \rangle)$$

syntax-consts

$$\text{-OclSelectBag} == \text{OclSelect}$$

translations

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

2.8.17. Definition: Reject

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

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

syntax

$-OclRejectBag :: [(\text{'}\mathcal{A}, \text{'}\alpha :: \text{null}) Bag, id, (\text{'}\mathcal{A}) \text{Boolean}] \Rightarrow \text{'}\mathcal{A}\ \text{Boolean} \quad (\langle (-) \rightarrow reject_{Bag} \text{' } (-) \rangle)$

syntax-consts

$-OclRejectBag == OclReject$

translations

$X \rightarrow reject_{Bag} (x \mid P) == CONST\ OclReject\ X\ (\% x.\ P)$

2.8.18. Definition: IncludesAll

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

where $OclIncludesAll\ x\ y = (\lambda\ \tau.\ \text{if } (\delta\ x)\ \tau = \text{true } \tau \wedge (\delta\ y)\ \tau = \text{true } \tau$

$\text{then } \sqcup Rep\text{-}Bag\text{-}base\ y\ \tau \subseteq Rep\text{-}Bag\text{-}base\ x\ \tau \sqcup$

$\text{else } \perp)$

notation $OclIncludesAll\ (\langle \rightarrow includesAll_{Bag} \text{' } (-) \rangle)$

interpretation $OclIncludesAll : profile\text{-}bin_d\text{-}d\ OclIncludesAll\ \lambda x\ y.\ \sqcup Rep\text{-}Bag\text{-}base'\ y \subseteq Rep\text{-}Bag\text{-}base'\ x \sqcup$

2.8.19. Definition: ExcludesAll

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

where $OclExcludesAll\ x\ y = (\lambda\ \tau.\ \text{if } (\delta\ x)\ \tau = \text{true } \tau \wedge (\delta\ y)\ \tau = \text{true } \tau$

$\text{then } \sqcup Rep\text{-}Bag\text{-}base\ y\ \tau \cap Rep\text{-}Bag\text{-}base\ x\ \tau = \{ \} \sqcup$

$\text{else } \perp)$

notation $OclExcludesAll\ (\langle \rightarrow excludesAll_{Bag} \text{' } (-) \rangle)$

interpretation $OclExcludesAll : profile\text{-}bin_d\text{-}d\ OclExcludesAll\ \lambda x\ y.\ \sqcup Rep\text{-}Bag\text{-}base'\ y \cap Rep\text{-}Bag\text{-}base'\ x = \{ \} \sqcup$

2.8.20. Definition: Union

definition $OclUnion :: [(\text{'}\mathcal{A}, \text{'}\alpha :: \text{null}) Bag, (\text{'}\mathcal{A}, \text{'}\alpha) Bag] \Rightarrow (\text{'}\mathcal{A}, \text{'}\alpha) Bag$

where $OclUnion\ x\ y = (\lambda\ \tau.\ \text{if } (\delta\ x)\ \tau = \text{true } \tau \wedge (\delta\ y)\ \tau = \text{true } \tau$

$\text{then } Abs\text{-}Bag_{base} \sqcup \lambda X.\ \sqcap Rep\text{-}Bag_{base}\ (x\ \tau)^\top X +$

$\sqcap Rep\text{-}Bag_{base}\ (y\ \tau)^\top X \sqcup$

$\text{else } invalid\ \tau)$

notation $OclUnion\ (\langle \rightarrow union_{Bag} \text{' } (-) \rangle)$

interpretation $OclUnion :$

$profile\text{-}bin_d\text{-}d\ OclUnion\ \lambda x\ y.\ Abs\text{-}Bag_{base} \sqcup \lambda X.\ \sqcap Rep\text{-}Bag_{base}\ x^\top X +$
 $\sqcap Rep\text{-}Bag_{base}\ y^\top X \sqcup$

2.8.21. Definition: Intersection

definition $OclIntersection :: [(\text{'}\mathcal{A}, \text{'}\alpha :: \text{null}) Bag, (\text{'}\mathcal{A}, \text{'}\alpha) Bag] \Rightarrow (\text{'}\mathcal{A}, \text{'}\alpha) Bag$

where $OclIntersection\ x\ y = (\lambda\ \tau.\ \text{if } (\delta\ x)\ \tau = \text{true } \tau \wedge (\delta\ y)\ \tau = \text{true } \tau$

$\text{then } Abs\text{-}Bag_{base} \sqcup \lambda X.\ min\ (\sqcap Rep\text{-}Bag_{base}\ (x\ \tau)^\top X)$

$(\sqcap Rep\text{-}Bag_{base}\ (y\ \tau)^\top X) \sqcup$

$\text{else } \perp$)

notation $\text{OclIntersection}(\langle \rightarrow \rangle \text{intersection}_{\text{Bag}} '(-)') \rangle$)

interpretation OclIntersection :

$$\text{profile-bin}_{d-d} \text{OclIntersection } \lambda x y. \text{Abs-Bag}_{\text{base}} \perp \lambda X. \min ({}^{\top} \text{Rep-Bag}_{\text{base}} x^{\top} X) \\ ({}^{\top} \text{Rep-Bag}_{\text{base}} y^{\top} X)_{\perp}$$

2.8.22. Definition: Count

definition $\text{OclCount} :: [({}^b \mathbb{N}, {}^b \alpha :: \text{null}) \text{Bag}, ({}^b \mathbb{N}, {}^b \alpha) \text{val}] \Rightarrow ({}^b \mathbb{N}) \text{Integer}$

where $\text{OclCount } x y = (\lambda \tau. \text{if } (\delta x) \tau = \text{true} \wedge (\delta y) \tau = \text{true} \wedge$
 $\text{then } \perp \text{int}({}^{\top} \text{Rep-Bag}_{\text{base}} (x \tau)^{\top} (y \tau))^{\top})_{\perp}$
 $\text{else invalid } \tau)$

notation $\text{OclCount}(\langle \rightarrow \rangle \text{count}_{\text{Bag}} '(-)') \rangle$)

interpretation $\text{OclCount} : \text{profile-bin}_{d-d} \text{OclCount } \lambda x y. \perp \text{int}({}^{\top} \text{Rep-Bag}_{\text{base}} x^{\top} y)_{\perp}$

2.8.23. Definition (future operators)

consts

$\text{OclSum} :: ({}^b \mathbb{N}, {}^b \alpha :: \text{null}) \text{Bag} \Rightarrow {}^b \mathbb{N} \text{Integer}$

notation $\text{OclSum}(\langle \rightarrow \rangle \text{sum}_{\text{Bag}} '(-)') \rangle$)

2.8.24. Logical Properties

OclIncluding

lemma $\text{OclIncluding-valid-args-valid}$:

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

lemma $\text{OclIncluding-valid-args-valid}''[\text{simp}, \text{code-unfold}]$:

$v(X \rightarrow \text{including}_{\text{Bag}}(x)) = ((\delta X) \text{ and } (v x))$

etc. etc.

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

OclIncluding

OclExcluding

OclIncludes

OclExcludes

OclSize

lemma $\text{OclSize-invalid}[\text{simp}, \text{code-unfold}]: (\text{invalid} \rightarrow \text{size}_{\text{Bag}}()) = \text{invalid}$

lemma $\text{OclSize-null}[\text{simp}, \text{code-unfold}]: (\text{null} \rightarrow \text{size}_{\text{Bag}}()) = \text{invalid}$

OclIsEmpty

lemma *OclIsEmpty-invalid[simp,code-unfold]:(invalid $\rightarrow$ isEmptyBag()) = invalid*

lemma *OclIsEmpty-null[simp,code-unfold]:(null $\rightarrow$ isEmptyBag()) = true*

OclNotEmpty

lemma *OclNotEmpty-invalid[simp,code-unfold]:(invalid $\rightarrow$ notEmptyBag()) = invalid*

lemma *OclNotEmpty-null[simp,code-unfold]:(null $\rightarrow$ notEmptyBag()) = false*

OclANY

lemma *OclANY-invalid[simp,code-unfold]:(invalid $\rightarrow$ anyBag()) = invalid*

lemma *OclANY-null[simp,code-unfold]:(null $\rightarrow$ anyBag()) = null*

OclForall

lemma *OclForall-invalid[simp,code-unfold]:invalid $\rightarrow$ forallBag(a | P a) = invalid*

lemma *OclForall-null[simp,code-unfold]:null $\rightarrow$ forallBag(a | P a) = invalid*

OclExists

lemma *OclExists-invalid[simp,code-unfold]:invalid $\rightarrow$ existsBag(a | P a) = invalid*

lemma *OclExists-null[simp,code-unfold]:null $\rightarrow$ existsBag(a | P a) = invalid*

OclIterate

lemma *OclIterate-invalid[simp,code-unfold]:invalid $\rightarrow$ iterateBag(a; x = A | P a x) = invalid*

lemma *OclIterate-null[simp,code-unfold]:null $\rightarrow$ iterateBag(a; x = A | P a x) = invalid*

lemma *OclIterate-invalid-args[simp,code-unfold]:S $\rightarrow$ iterateBag(a; x = invalid | P a x) = invalid*

An open question is this ...

lemma *S $\rightarrow$ iterateBag(a; x = null | P a x) = invalid*

lemma *OclIterate-infinite:*

assumes *non-finite: $\tau \models \text{not}(\delta(S \rightarrow \text{sizeBag}()))$*

shows *(OclIterate S A F) τ = invalid τ*

OclSelect

lemma *OclSelect-invalid[simp,code-unfold]:invalid $\rightarrow$ selectBag(a | P a) = invalid*

lemma $OclSelect_null[simp,code-unfold]: null \rightarrow select_{Bag}(a \mid P a) = invalid$

OclReject

lemma $OclReject_invalid[simp,code-unfold]: invalid \rightarrow reject_{Bag}(a \mid P a) = invalid$

lemma $OclReject_null[simp,code-unfold]: null \rightarrow reject_{Bag}(a \mid P a) = invalid$

2.8.26. Test Statements

instantiation $Bag_{base} :: (equal)equal$

begin

definition $HOL.equal\ k\ l \longleftrightarrow (k::('a::equal)Bag_{base}) = l$

instance

end

lemma $equal_Bag_{base}\text{-}code\ [code]:$

$HOL.equal\ k\ (l::('a::\{equal,null\})Bag_{base}) \longleftrightarrow Rep_Bag_{base}\ k = Rep_Bag_{base}\ l$

Assert $\tau \models (Bag\{\} \doteq Bag\{\})$

no-notation $None\ (\langle \perp \rangle)$

2.9. Collection Type Set: Operations

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

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

where $Integer \equiv (\lambda\ \tau. (Abs_Set_{base}\ o\ Some\ o\ Some)\ ((Some\ o\ Some)\ ' (UNIV::int\ set)))$

definition $Integer_{null} :: ('A, Integer_{base}) Set$

where $Integer_{null} \equiv (\lambda\ \tau. (Abs_Set_{base}\ o\ Some\ o\ Some)\ (Some\ ' (UNIV::int\ option\ set)))$

lemma $Integer_defined : \delta\ Integer = true$

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

This allows the theorems:

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

and

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

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

In the same spirit, we proceed similarly for the remaining base types:

definition $Void_{null} :: ('A, Void_{base}) Set$

where $Void_{null} \equiv (\lambda \tau. (Abs\text{-}Set_{base} \circ Some \circ Some) \{Abs\text{-}Void_{base} (Some None)\})$

definition $Void_{empty} :: ('A, Void_{base}) Set$

where $Void_{empty} \equiv (\lambda \tau. (Abs\text{-}Set_{base} \circ Some \circ Some) \{\})$

lemma $Void_{null}\text{-defined} : \delta Void_{null} = true$

lemma $Void_{empty}\text{-defined} : \delta Void_{empty} = true$

lemma assumes $\tau \models \delta (V :: ('A, Void_{base}) Set)$

shows $\tau \models V \triangleq Void_{null} \vee \tau \models V \triangleq Void_{empty}$

definition $Boolean :: ('A, Boolean_{base}) Set$

where $Boolean \equiv (\lambda \tau. (Abs\text{-}Set_{base} \circ Some \circ Some) ((Some \circ Some) ' (UNIV::bool set)))$

definition $Boolean_{null} :: ('A, Boolean_{base}) Set$

where $Boolean_{null} \equiv (\lambda \tau. (Abs\text{-}Set_{base} \circ Some \circ Some) (Some ' (UNIV::bool option set)))$

lemma $Boolean\text{-defined} : \delta Boolean = true$

lemma $Boolean_{null}\text{-defined} : \delta Boolean_{null} = true$

definition $String :: ('A, String_{base}) Set$

where $String \equiv (\lambda \tau. (Abs\text{-}Set_{base} \circ Some \circ Some) ((Some \circ Some) ' (UNIV::string set)))$

definition $String_{null} :: ('A, String_{base}) Set$

where $String_{null} \equiv (\lambda \tau. (Abs\text{-}Set_{base} \circ Some \circ Some) (Some ' (UNIV::string option set)))$

lemma $String\text{-defined} : \delta String = true$

lemma $String_{null}\text{-defined} : \delta String_{null} = true$

definition $Real :: ('A, Real_{base}) Set$

where $Real \equiv (\lambda \tau. (Abs\text{-}Set_{base} \circ Some \circ Some) ((Some \circ Some) ' (UNIV::real set)))$

definition $Real_{null} :: ('A, Real_{base}) Set$

where $Real_{null} \equiv (\lambda \tau. (Abs\text{-}Set_{base} \circ Some \circ Some) (Some ' (UNIV::real option set)))$

lemma $Real\text{-defined} : \delta Real = true$

lemma $Real_{null}\text{-defined} : \delta Real_{null} = true$

2.9.2. Basic Properties of the Set Type

Every element in a defined set is valid.

lemma $Set\text{-inv-lemma} : \tau \models (\delta X) \implies \forall x \in {}^{\top}Rep\text{-Set}_{base} (X \tau)^{\top}. x \neq bot$

lemma $Set\text{-inv-lemma}' :$
assumes $x\text{-def} : \tau \models \delta X$
and $e\text{-mem} : e \in {}^{\top}Rep\text{-Set}_{base} (X \tau)^{\top}$
shows $\tau \models v(\lambda \cdot. e)$

lemma $abs\text{-rep-simp}' :$
assumes $S\text{-all-def} : \tau \models \delta S$
shows $Abs\text{-Set}_{base} \sqsubseteq {}^{\top}Rep\text{-Set}_{base} (S \tau)^{\top} \sqsubseteq S \tau$

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

lemma $invalid\text{-set-OclNot-defined} [simp,code-unfold] : \delta(invalid::('A,'A::null) Set) = false$

lemma $null\text{-set-OclNot-defined} [simp,code-unfold] : \delta(null::('A,'A::null) Set) = false$

lemma $invalid\text{-set-valid} [simp,code-unfold] : v(invalid::('A,'A::null) Set) = false$

lemma $null\text{-set-valid} [simp,code-unfold] : v(null::('A,'A::null) Set) = true$

... which means that we can have a type $('A, ('A, ('A) Integer) Set) Set$ corresponding exactly to $Set(Set(Integer))$ 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.

2.9.3. 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:

overloading

$StrictRefEq \equiv StrictRefEq :: [('A,'A::null)Set, ('A,'A::null)Set] \Rightarrow ('A)Boolean$

begin

definition $StrictRefEq_{Set} :$

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

end

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

Property proof in terms of $profile-bin_{StrongEq}^{-v-v}$

interpretation $StrictRefEq_{Set} : profile-bin_{StrongEq}^{-v-v} \lambda x y. (x::('A, 'a::null)Set) \doteq y$

2.9.4. Constants: mtSet

definition $mtSet::('A, 'a::null) Set \ (\langle Set\{ \} \rangle)$

where $Set\{ \} \equiv (\lambda \tau. Abs-Set_{base \perp} \{ \}::'a set_{\perp})$

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

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

lemma $mtSet-rep-set: \ulcorner Rep-Set_{base} (Set\{ \} \tau) \urcorner = \{ \}$

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.

2.9.5. Definition: Including

definition $OclIncluding :: [('A, 'a::null) Set, ('A, 'a) val] \Rightarrow ('A, 'a) Set$

where $OclIncluding x y = (\lambda \tau. if (\delta x) \tau = true \tau \wedge (v y) \tau = true \tau$
 $then Abs-Set_{base \perp} \ulcorner Rep-Set_{base} (x \tau) \urcorner \cup \{y \tau\}_{\perp}$
 $else invalid \tau)$

notation $OclIncluding \ (\leftarrow\rightarrow including_{Set} '(-'))$

interpretation $OclIncluding : profile-bin_d^{-v} OclIncluding \lambda x y. Abs-Set_{base \perp} \ulcorner Rep-Set_{base} x \urcorner \cup \{y\}_{\perp}$

syntax

$-OclFinset :: args \Rightarrow ('A, 'a::null) Set \ (\langle Set\{(-)\} \rangle)$

syntax-consts

$-OclFinset == OclIncluding$

translations

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

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

2.9.6. Definition: Excluding

definition $OclExcluding :: [('A, 'a::null) Set, ('A, 'a) val] \Rightarrow ('A, 'a) Set$

where $OclExcluding x y = (\lambda \tau. if (\delta x) \tau = true \tau \wedge (v y) \tau = true \tau$
 $then Abs-Set_{base \perp} \ulcorner Rep-Set_{base} (x \tau) \urcorner - \{y \tau\}_{\perp}$
 $else \perp)$

notation $OclExcluding \ (\leftarrow\rightarrow excluding_{Set} '(-'))$

lemma $OclExcluding-inv: (x::Set('b::\{null\})) \neq \perp \implies x \neq null \implies y \neq \perp \implies$
 $\ulcorner Rep-Set_{base} x \urcorner - \{y\}_{\perp} \in \{X. X = bot \vee X = null \vee (\forall x \in \ulcorner X \urcorner. x \neq bot)\}$

interpretation $OclExcluding : profile-bin_d-v \ OclExcluding \lambda x y. Abs-Set_{base} \ulcorner Rep-Set_{base} x^{\top} - \{y\} \urcorner$

2.9.7. Definition: Includes

definition $OclIncludes :: (^A, 'a::null) Set, (^A, 'a) val] \Rightarrow ^A Boolean$

where $OclIncludes \ x \ y = (\lambda \ \tau. \text{ if } (\delta \ x) \ \tau = true \ \tau \wedge (v \ y) \ \tau = true \ \tau$
 $\text{ then } \ulcorner (y \ \tau) \in \ulcorner Rep-Set_{base} (x \ \tau) \urcorner \urcorner$
 $\text{ else } \perp)$

notation $OclIncludes \ (\longleftrightarrow includes_{Set} '(-)')$

interpretation $OclIncludes : profile-bin_d-v \ OclIncludes \lambda x y. \ulcorner y \in \ulcorner Rep-Set_{base} x^{\top} \urcorner \urcorner$

2.9.8. Definition: Excludes

definition $OclExcludes :: (^A, 'a::null) Set, (^A, 'a) val] \Rightarrow ^A Boolean$

where $OclExcludes \ x \ y = (not(OclIncludes \ x \ y))$

notation $OclExcludes \ (\longleftrightarrow excludes_{Set} '(-)')$

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.

interpretation $OclExcludes : profile-bin_d-v \ OclExcludes \lambda x y. \ulcorner y \notin \ulcorner Rep-Set_{base} x^{\top} \urcorner \urcorner$

2.9.9. Definition: Size

definition $OclSize :: (^A, 'a::null) Set \Rightarrow ^A Integer$

where $OclSize \ x = (\lambda \ \tau. \text{ if } (\delta \ x) \ \tau = true \ \tau \wedge finite(\ulcorner Rep-Set_{base} (x \ \tau) \urcorner)$
 $\text{ then } \ulcorner int(card \ \ulcorner Rep-Set_{base} (x \ \tau) \urcorner) \urcorner$
 $\text{ else } \perp)$

notation

$OclSize \ (\longleftrightarrow size_{Set} '(-)')$

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

2.9.10. Definition: IsEmpty

definition $OclIsEmpty :: (^A, 'a::null) Set \Rightarrow ^A Boolean$

where $OclIsEmpty \ x = ((v \ x \text{ and not } (\delta \ x)) \text{ or } ((OclSize \ x) \doteq 0))$

notation $OclIsEmpty \ (\longleftrightarrow isEmpty_{Set} '(-)')$

2.9.11. Definition: NotEmpty

definition $OclNotEmpty :: (^A, 'a::null) Set \Rightarrow ^A Boolean$

where $OclNotEmpty \ x = not(OclIsEmpty \ x)$

notation $OclNotEmpty \ (\longleftrightarrow notEmpty_{Set} '(-)')$

2.9.12. Definition: Any

definition $OclANY :: [(\mathcal{A}, ' \alpha :: null) Set] \Rightarrow (\mathcal{A}, ' \alpha) val$
where $OclANY x = (\lambda \tau. \text{if } (\vee x) \tau = true \tau$
 $\text{then if } (\delta x \text{ and } OclNotEmpty x) \tau = true \tau$
 $\text{then SOME } y. y \in {}^{\top}Rep-Set_{base} (x \tau)^{\top}$
 $\text{else null } \tau$
 $\text{else } \perp)$
notation $OclANY \ (\leftarrow>any_{Set} \ ' \langle \rangle)$

2.9.13. Definition: Forall

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

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

syntax

$-OclForallSet :: [(\mathcal{A}, ' \alpha :: null) Set, id, (\mathcal{A}) Boolean] \Rightarrow \mathcal{A} Boolean \ (\leftarrow(-)>forAll_{Set} \ ' \langle - | - \rangle)$

syntax-consts

$-OclForallSet == UML-Set.OclForall$

translations

$X \rightarrow>forAll_{Set} (x \mid P) == CONST \ UML-Set.OclForall \ X \ (\%x. P)$

2.9.14. Definition: Exists

Like $OclForall$, $OclExists$ is also not strict.

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

syntax

$-OclExistSet :: [(\mathcal{A}, ' \alpha :: null) Set, id, (\mathcal{A}) Boolean] \Rightarrow \mathcal{A} Boolean \ (\leftarrow(-)>exists_{Set} \ ' \langle - | - \rangle)$

syntax-consts

$-OclExistSet == UML-Set.OclExists$

translations

$X \rightarrow>exists_{Set} (x \mid P) == CONST \ UML-Set.OclExists \ X \ (\%x. P)$

2.9.15. Definition: Iterate

definition $OclIterate :: [(\mathcal{A}, ' \alpha :: null) Set, (\mathcal{A}, ' \beta :: null) val,$
 $(\mathcal{A}, ' \alpha) val \Rightarrow (\mathcal{A}, ' \beta) val \Rightarrow (\mathcal{A}, ' \beta) val] \Rightarrow (\mathcal{A}, ' \beta) val$
where $OclIterate S A F = (\lambda \tau. \text{if } (\delta S) \tau = true \tau \wedge (\vee A) \tau = true \tau \wedge finite {}^{\top}Rep-Set_{base} (S \tau)^{\top}$
 $\text{then } (Finite-Set.fold (F) (A) ((\lambda a \tau. a) \cdot {}^{\top}Rep-Set_{base} (S \tau)^{\top})) \tau$
 $\text{else } \perp)$

syntax

$-OclIterateSet :: [(\mathcal{A}, ' \alpha :: null) Set, id, id, ' \alpha, ' \beta] \Rightarrow (\mathcal{A}, ' \gamma) val$

$(\leftarrow \rightarrow \text{iterate}_{\text{Set}} '(-; :- | -)') \rangle$

syntax-consts

$\text{-OclIterateSet} == \text{OclIterate}$

translations

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

2.9.16. Definition: Select

definition $\text{OclSelect} :: [('A, 'A::\text{null}) \text{Set}, ('A, 'A) \text{val} \Rightarrow ('A) \text{Boolean}] \Rightarrow ('A, 'A) \text{Set}$

where $\text{OclSelect } S \ P = (\lambda \tau. \text{if } (\delta \ S) \ \tau = \text{true } \tau$

$\text{then if } (\exists x \in {}^{\top}\text{Rep-Set}_{\text{base}} (S \ \tau)^{\top}. P(\lambda -. x) \ \tau = \text{invalid } \tau$

$\text{then invalid } \tau$

$\text{else Abs-Set}_{\text{base}} \perp \{x \in {}^{\top}\text{Rep-Set}_{\text{base}} (S \ \tau)^{\top}. P(\lambda -. x) \ \tau \neq \text{false } \tau\} \perp$

$\text{else invalid } \tau)$

syntax

$\text{-OclSelectSet} :: [('A, 'A::\text{null}) \text{Set}, \text{id}, ('A) \text{Boolean}] \Rightarrow 'A \ \text{Boolean} \quad (\leftarrow \rightarrow \text{select}_{\text{Set}} '(-|-)') \rangle$

syntax-consts

$\text{-OclSelectSet} == \text{OclSelect}$

translations

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

2.9.17. Definition: Reject

definition $\text{OclReject} :: [('A, 'A::\text{null}) \text{Set}, ('A, 'A) \text{val} \Rightarrow ('A) \text{Boolean}] \Rightarrow ('A, 'A::\text{null}) \text{Set}$

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

syntax

$\text{-OclRejectSet} :: [('A, 'A::\text{null}) \text{Set}, \text{id}, ('A) \text{Boolean}] \Rightarrow 'A \ \text{Boolean} \quad (\leftarrow \rightarrow \text{reject}_{\text{Set}} '(-|-)') \rangle$

syntax-consts

$\text{-OclRejectSet} == \text{OclReject}$

translations

$X \rightarrow \text{reject}_{\text{Set}}(x \mid P) == \text{CONST OclReject } X \ (\% x. P)$

2.9.18. Definition: IncludesAll

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

where $\text{OclIncludesAll } x \ y = (\lambda \tau. \text{if } (\delta \ x) \ \tau = \text{true } \tau \wedge (\delta \ y) \ \tau = \text{true } \tau$

$\text{then } \perp \{x \in {}^{\top}\text{Rep-Set}_{\text{base}} (y \ \tau)^{\top} \subseteq {}^{\top}\text{Rep-Set}_{\text{base}} (x \ \tau)^{\top}\} \perp$

$\text{else } \perp)$

notation $\text{OclIncludesAll } (\leftarrow \rightarrow \text{includesAll}_{\text{Set}} '(-|-)') \rangle$

interpretation $\text{OclIncludesAll} : \text{profile-bin}_{\text{d-d}} \ \text{OclIncludesAll } \lambda x \ y. \perp \{x \in {}^{\top}\text{Rep-Set}_{\text{base}} y^{\top} \subseteq {}^{\top}\text{Rep-Set}_{\text{base}} x^{\top}\} \perp$

2.9.19. Definition: ExcludesAll

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

where $\text{OclExcludesAll } x \ y = (\lambda \tau. \text{if } (\delta \ x) \ \tau = \text{true } \tau \wedge (\delta \ y) \ \tau = \text{true } \tau$

$\text{then } \perp \{x \in {}^{\top}\text{Rep-Set}_{\text{base}} (y \ \tau)^{\top} \cap {}^{\top}\text{Rep-Set}_{\text{base}} (x \ \tau)^{\top} = \{\}\} \perp$

$\text{else } \perp)$

notation $\text{OclExcludesAll } (\leftarrow \rightarrow \text{excludesAll}_{\text{Set}} '(-|-)') \rangle$

interpretation $\text{OclExcludesAll} : \text{profile-bin}_{\text{d-d}} \ \text{OclExcludesAll } \lambda x \ y. \perp \{x \in {}^{\top}\text{Rep-Set}_{\text{base}} y^{\top} \cap {}^{\top}\text{Rep-Set}_{\text{base}} x^{\top} = \{\}\} \perp$

2.9.20. Definition: Union

definition $OclUnion :: [({\mathcal{A}}, {\alpha}::null) Set, ({\mathcal{A}}, {\alpha}) Set] \Rightarrow ({\mathcal{A}}, {\alpha}) Set$

where $OclUnion\ x\ y = (\lambda\ \tau.\ \text{if } (\delta\ x)\ \tau = \text{true } \tau \wedge (\delta\ y)\ \tau = \text{true } \tau$
 $\text{then } Abs\text{-}Set_{base\perp} \ulcorner Rep\text{-}Set_{base}\ (y\ \tau) \urcorner \cup \ulcorner Rep\text{-}Set_{base}\ (x\ \tau) \urcorner _ _$
 $\text{else } \perp\)$

notation $OclUnion\ (\leftarrow\!\!\rightarrow)_{union_{Set}}'(-)'$

lemma $OclUnion\text{-}inv: (x:: Set'({\mathcal{B}}::\{null\})) \neq \perp \implies x \neq null \implies y \neq \perp \implies y \neq null \implies$
 $\ulcorner Rep\text{-}Set_{base}\ y \urcorner \cup \ulcorner Rep\text{-}Set_{base}\ x \urcorner _ _ \in \{X.\ X = bot \vee X = null \vee (\forall x \in \ulcorner X \urcorner.\ x \neq bot)\}$

interpretation $OclUnion : profile\text{-}bind\text{-}d\ OclUnion\ \lambda x\ y.\ Abs\text{-}Set_{base\perp} \ulcorner Rep\text{-}Set_{base}\ y \urcorner \cup \ulcorner Rep\text{-}Set_{base}\ x \urcorner _ _$

2.9.21. Definition: Intersection

definition $OclIntersection :: [({\mathcal{A}}, {\alpha}::null) Set, ({\mathcal{A}}, {\alpha}) Set] \Rightarrow ({\mathcal{A}}, {\alpha}) Set$

where $OclIntersection\ x\ y = (\lambda\ \tau.\ \text{if } (\delta\ x)\ \tau = \text{true } \tau \wedge (\delta\ y)\ \tau = \text{true } \tau$
 $\text{then } Abs\text{-}Set_{base\perp} \ulcorner Rep\text{-}Set_{base}\ (y\ \tau) \urcorner$
 $\cap \ulcorner Rep\text{-}Set_{base}\ (x\ \tau) \urcorner _ _$
 $\text{else } \perp\)$

notation $OclIntersection(\leftarrow\!\!\rightarrow)_{intersection_{Set}}'(-)'$

lemma $OclIntersection\text{-}inv: (x:: Set'({\mathcal{B}}::\{null\})) \neq \perp \implies x \neq null \implies y \neq \perp \implies y \neq null \implies$
 $\ulcorner Rep\text{-}Set_{base}\ y \urcorner \cap \ulcorner Rep\text{-}Set_{base}\ x \urcorner _ _ \in \{X.\ X = bot \vee X = null \vee (\forall x \in \ulcorner X \urcorner.\ x \neq bot)\}$

interpretation $OclIntersection : profile\text{-}bind\text{-}d\ OclIntersection\ \lambda x\ y.\ Abs\text{-}Set_{base\perp} \ulcorner Rep\text{-}Set_{base}\ y \urcorner \cap \ulcorner Rep\text{-}Set_{base}\ x \urcorner _ _$

2.9.22. Definition (future operators)

consts

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

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

notation $OclCount\ (\leftarrow\!\!\rightarrow)_{count_{Set}}'(-)'$

notation $OclSum\ (\leftarrow\!\!\rightarrow)_{sum_{Set}}'(-)'$

2.9.23. Logical Properties

$OclIncluding$

lemma $OclIncluding\text{-}valid\text{-}args\text{-}valid:$

$(\tau \models v(X \rightarrow including_{Set}(x))) = ((\tau \models (\delta\ X)) \wedge (\tau \models (v\ x)))$

lemma $OclIncluding\text{-}valid\text{-}args\text{-}valid''[simp, code-unfold]:$

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

etc. etc.

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

OclIncluding

OclExcluding

OclIncludes

OclExcludes

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

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

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

OclForall

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

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

OclExists

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

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

OclIterate

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

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

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

An open question is this ...

lemma $S \rightarrow \text{iterate}_{\text{Set}}(a; x = \text{null} \mid P \ a \ x) = \text{invalid}$

lemma *OclIterate-infinite*:

assumes *non-finite*: $\tau \models \text{not}(\delta(S \rightarrow \text{size}_{\text{Set}}()))$

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

OclSelect

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

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

OclReject

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

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

2.9.25. General Algebraic Execution Rules

2.9.25.1. Execution Rules on Including

lemma *OclIncluding-finite-rep-set* :

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

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

shows $\text{finite } \ulcorner \text{Rep-Set}_{\text{base}} (X \rightarrow \text{including}_{\text{Set}}(x) \ \tau) \urcorner = \text{finite } \ulcorner \text{Rep-Set}_{\text{base}} (X \ \tau) \urcorner$

lemma *OclIncluding-rep-set*:

assumes *S-def*: $\tau \models \delta \ S$

shows $\ulcorner \text{Rep-Set}_{\text{base}} (S \rightarrow \text{including}_{\text{Set}}(\lambda \cdot \ulcorner x \urcorner) \ \tau) \urcorner = \text{insert } \ulcorner x \urcorner \ulcorner \text{Rep-Set}_{\text{base}} (S \ \tau) \urcorner$

lemma *OclIncluding-notempty-rep-set*:

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

and *a-val*: $\tau \models v \ a$

shows $\ulcorner \text{Rep-Set}_{\text{base}} (X \rightarrow \text{including}_{\text{Set}}(a) \ \tau) \urcorner \neq \{\}$

lemma *OclIncluding-includes0*:

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

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

lemma *OclIncluding-includes*:

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

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

lemma *OclIncluding-commute0* :

assumes *S-def* : $\tau \models \delta \ S$

and *i-val* : $\tau \models v \ i$

and $j\text{-val} : \tau \models v j$
shows $\tau \models ((S :: ('A, 'a::null) \text{Set}) \rightarrow \text{including}_{\text{Set}}(i) \rightarrow \text{including}_{\text{Set}}(j)) \triangleq (S \rightarrow \text{including}_{\text{Set}}(j) \rightarrow \text{including}_{\text{Set}}(i))$

lemma *OclIncluding-commute[simp,code-unfold]*:
 $((S :: ('A, 'a::null) \text{Set}) \rightarrow \text{including}_{\text{Set}}(i) \rightarrow \text{including}_{\text{Set}}(j)) = (S \rightarrow \text{including}_{\text{Set}}(j) \rightarrow \text{including}_{\text{Set}}(i))$

2.9.25.2. Execution Rules on Excluding

lemma *OclExcluding-finite-rep-set* :
assumes $X\text{-def} : \tau \models \delta X$
and $x\text{-val} : \tau \models v x$
shows $\text{finite } {}^{\top}\text{Rep-Set}_{\text{base}} (X \rightarrow \text{excluding}_{\text{Set}}(x) \tau)^{\top} = \text{finite } {}^{\top}\text{Rep-Set}_{\text{base}} (X \tau)^{\top}$

lemma *OclExcluding-rep-set*:
assumes $S\text{-def} : \tau \models \delta S$
shows ${}^{\top}\text{Rep-Set}_{\text{base}} (S \rightarrow \text{excluding}_{\text{Set}}(\lambda \cdot \cdot \text{⌊}x\text{⌋}) \tau)^{\top} = {}^{\top}\text{Rep-Set}_{\text{base}} (S \tau)^{\top} - \{\text{⌊}x\text{⌋}\}$

lemma *OclExcluding-excludes0*:
assumes $\tau \models X \rightarrow \text{excludes}_{\text{Set}}(x)$
shows $X \rightarrow \text{excluding}_{\text{Set}}(x) \tau = X \tau$

lemma *OclExcluding-excludes*:
assumes $\tau \models X \rightarrow \text{excludes}_{\text{Set}}(x)$
shows $\tau \models X \rightarrow \text{excluding}_{\text{Set}}(x) \triangleq X$

lemma *OclExcluding-cha0[simp]*:
assumes $\text{val-}x : \tau \models (v x)$
shows $\tau \models ((\text{Set}\{ \} \rightarrow \text{excluding}_{\text{Set}}(x)) \triangleq \text{Set}\{ \})$

lemma *OclExcluding-commute0* :
assumes $S\text{-def} : \tau \models \delta S$
and $i\text{-val} : \tau \models v i$
and $j\text{-val} : \tau \models v j$
shows $\tau \models ((S :: ('A, 'a::null) \text{Set}) \rightarrow \text{excluding}_{\text{Set}}(i) \rightarrow \text{excluding}_{\text{Set}}(j)) \triangleq (S \rightarrow \text{excluding}_{\text{Set}}(j) \rightarrow \text{excluding}_{\text{Set}}(i))$

lemma *OclExcluding-commute[simp,code-unfold]*:
 $((S :: ('A, 'a::null) \text{Set}) \rightarrow \text{excluding}_{\text{Set}}(i) \rightarrow \text{excluding}_{\text{Set}}(j)) = (S \rightarrow \text{excluding}_{\text{Set}}(j) \rightarrow \text{excluding}_{\text{Set}}(i))$

lemma *OclExcluding-cha0-exec[simp,code-unfold]*:
 $(\text{Set}\{ \} \rightarrow \text{excluding}_{\text{Set}}(x)) = (\text{if } (v x) \text{ then } \text{Set}\{ \} \text{ else invalid endif})$

lemma *OclExcluding-cha1*:
assumes $\text{def-}X : \tau \models (\delta X)$
and $\text{val-}x : \tau \models (v x)$
and $\text{val-}y : \tau \models (v y)$
and $\text{neq} : \tau \models \text{not}(x \triangleq y)$
shows $\tau \models ((X \rightarrow \text{including}_{\text{Set}}(x)) \rightarrow \text{excluding}_{\text{Set}}(y)) \triangleq ((X \rightarrow \text{excluding}_{\text{Set}}(y)) \rightarrow \text{including}_{\text{Set}}(x))$

lemma *OclExcluding-charn2*:
assumes $\text{def } X:\tau \models (\delta \ X)$
and $\text{val-}x:\tau \models (\text{v } x)$
shows $\tau \models (((X \rightarrow \text{including}_{\text{Set}}(x)) \rightarrow \text{excluding}_{\text{Set}}(x)) \triangleq (X \rightarrow \text{excluding}_{\text{Set}}(x)))$

theorem *OclExcluding-charn3*: $((X \rightarrow \text{including}_{\text{Set}}(x)) \rightarrow \text{excluding}_{\text{Set}}(x)) = (X \rightarrow \text{excluding}_{\text{Set}}(x))$

One would like a generic theorem of the form:

lemma *OclExcluding_charn_exec*:

$$\begin{aligned} & \text{"}(X \rightarrow \text{including}_{\text{Set}}(x::('A, 'a::\text{null})\text{val})) \rightarrow \text{excluding}_{\text{Set}}(y)) = \\ & \quad (\text{if } \delta \ X \text{ then if } x \doteq y \\ & \quad \quad \text{then } X \rightarrow \text{excluding}_{\text{Set}}(y) \\ & \quad \quad \text{else } X \rightarrow \text{excluding}_{\text{Set}}(y) \rightarrow \text{including}_{\text{Set}}(x) \\ & \quad \quad \text{endif} \\ & \quad \text{else invalid endif)"} \end{aligned}$$

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 equality).

lemma *OclExcluding-charn-exec*:
assumes *strict1*: $(\text{invalid} \doteq y) = \text{invalid}$
and *strict2*: $(x \doteq \text{invalid}) = \text{invalid}$
and *StrictRefEq-valid-args-valid*: $\bigwedge (x::('A, 'a::\text{null})\text{val}) \ y \ \tau. \\ (\tau \models \delta \ (x \doteq y)) = ((\tau \models (\text{v } x)) \wedge (\tau \models \text{v } y))$
and *cp-StrictRefEq*: $\bigwedge (X::('A, 'a::\text{null})\text{val}) \ Y \ \tau. (X \doteq Y) \ \tau = ((\lambda \cdot. X \ \tau) \doteq (\lambda \cdot. Y \ \tau)) \ \tau$
and *StrictRefEq-vs-StrongEq*: $\bigwedge (x::('A, 'a::\text{null})\text{val}) \ y \ \tau. \\ \tau \models \text{v } x \implies \tau \models \text{v } y \implies (\tau \models ((x \doteq y) \triangleq (x \triangleq y)))$
shows $(X \rightarrow \text{including}_{\text{Set}}(x::('A, 'a::\text{null})\text{val})) \rightarrow \text{excluding}_{\text{Set}}(y)) =$

$$\begin{aligned} & (\text{if } \delta \ X \text{ then if } x \doteq y \\ & \quad \text{then } X \rightarrow \text{excluding}_{\text{Set}}(y) \\ & \quad \text{else } X \rightarrow \text{excluding}_{\text{Set}}(y) \rightarrow \text{including}_{\text{Set}}(x) \\ & \quad \text{endif} \\ & \text{else invalid endif}) \end{aligned}$$

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

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

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

2.9.25.3. Execution Rules on Includes

lemma *OclIncludes-cha0*[simp]:

assumes $val\text{-}x:\tau \models (v\ x)$

shows $\tau \models not(Set\ \{\}\rightarrow includes_{Set}(x))$

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

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

lemma *OclIncludes-cha1*:

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

assumes $val\text{-}x:\tau \models (v\ x)$

shows $\tau \models (X\rightarrow including_{Set}(x)\rightarrow includes_{Set}(x))$

lemma *OclIncludes-cha2*:

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

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

and $val\text{-}y:\tau \models (v\ y)$

and $neq\ :\tau \models not(x \triangleq y)$

shows $\tau \models (X\rightarrow including_{Set}(x)\rightarrow includes_{Set}(y)) \triangleq (X\rightarrow includes_{Set}(y))$

Here is again a generic theorem similar as above.

lemma *OclIncludes-execute-generic*:

assumes $strict1: (invalid \doteq y) = invalid$

and $strict2: (x \doteq invalid) = invalid$

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

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

$\tau \models v\ x \implies \tau \models v\ y \implies (\tau \models ((x \doteq y) \triangleq (x \triangleq y)))$

shows

$(X\rightarrow including_{Set}(x::('A, 'a::null)val)\rightarrow includes_{Set}(y)) =$

$(if\ \delta\ X\ then\ if\ x \doteq y\ then\ true\ else\ X\rightarrow includes_{Set}(y)\ endif\ else\ invalid\ endif)$

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

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

schematic-goal *OclIncludes-executes*_{Set}[simp,code-unfold]: ?X

lemma *OclIncludes-including-generic* :

assumes *OclIncludes-execute-generic* [simp] : $\bigwedge X\ x\ y.$

$(X\rightarrow including_{Set}(x::('A, 'a::null)val)\rightarrow includes_{Set}(y)) =$

$(if\ \delta\ X\ then\ if\ x \doteq y\ then\ true\ else\ X\rightarrow includes_{Set}(y)\ endif\ else\ invalid\ endif)$

and $StrictRefEq\text{-}strict'' : \bigwedge x\ y. \delta\ ((x::('A, 'a::null)val) \doteq y) = (v(x)\ and\ v(y))$

and $a\text{-}val : \tau \models v\ a$

and $x\text{-}val : \tau \models v\ x$

and $S\text{-incl} : \tau \models (S) \rightarrow \text{includes}_{\text{Set}}((x::('A, 'a::\text{null})\text{val}))$
 shows $\tau \models S \rightarrow \text{including}_{\text{Set}}((a::('A, 'a::\text{null})\text{val})) \rightarrow \text{includes}_{\text{Set}}(x)$

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

2.9.25.4. Execution Rules on Excludes

lemma *OclExcludes-charn1*:
assumes *def-X*: $\tau \models (\delta X)$
assumes *val-x*: $\tau \models (v x)$
shows $\tau \models (X \rightarrow \text{excluding}_{\text{Set}}(x) \rightarrow \text{excludes}_{\text{Set}}(x))$

2.9.25.5. Execution Rules on Size

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

lemma *OclSize-including-exec*[*simp,code-unfold*]:
 $((X \rightarrow \text{including}_{\text{Set}}(x)) \rightarrow \text{size}_{\text{Set}}()) = (\text{if } \delta X \text{ and } v x \text{ then}$
 $\quad X \rightarrow \text{size}_{\text{Set}}() +_{\text{int}} \text{if } X \rightarrow \text{includes}_{\text{Set}}(x) \text{ then } 0 \text{ else } 1 \text{ endif}$
 $\quad \text{else}$
 $\quad \text{invalid}$
 $\quad \text{endif})$

2.9.25.6. Execution Rules on IsEmpty

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

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

2.9.25.7. 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-def*: $\tau \models \delta X$
and *X-finite*: $\text{finite } \ulcorner \text{Rep-Set}_{\text{base}}(X \ \tau) \urcorner$
and *a-val*: $\tau \models v a$
shows $X \rightarrow \text{including}_{\text{Set}}(a) \rightarrow \text{notEmpty}_{\text{Set}}() \ \tau = \text{true } \tau$

2.9.25.8. Execution Rules on Any

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

lemma *OclANY-singleton-exec*[*simp,code-unfold*]:

$$(Set\{\} \rightarrow including_{Set}(a)) \rightarrow any_{Set}() = a$$

2.9.25.9. Execution Rules on Forall

lemma *OclForall-mtSet-exec[simp,code-unfold]* : $((Set\{\}) \rightarrow forAll_{Set}(z \mid P(z))) = true$

The following rule is a main theorem of our approach: From a denotational definition that assures consistency, but may be — as in the case of the *OclForall X P* — dauntingly complex, we derive operational rules that can serve as a gold-standard for operational execution, since they may be evaluated in whatever situation and according to whatever strategy. In the case of *OclForall X P*, the operational rule gives immediately a way to evaluation in any finite (in terms of conventional OCL: denotable) set, although the rule also holds for the infinite case:

$$Integer_{null} \rightarrow forAll_{Set}(x \mid Integer_{null} \rightarrow forAll_{Set}(y \mid x +_{int} y \triangleq y +_{int} x))$$

or even:

$$Integer \rightarrow forAll_{Set}(x \mid Integer \rightarrow forAll_{Set}(y \mid x +_{int} y \doteq y +_{int} x))$$

are valid OCL statements in any context τ .

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

assumes *cp0* : *cp P*

shows $((S \rightarrow including_{Set}(x)) \rightarrow forAll_{Set}(z \mid P(z))) = (if \ \delta \ S \ and \ \forall \ x$
 $\quad then \ P \ x \ and \ (S \rightarrow forAll_{Set}(z \mid P(z)))$
 $\quad else \ invalid$
 $\quad endif)$

2.9.25.10. Execution Rules on Exists

lemma *OclExists-mtSet-exec[simp,code-unfold]* :

$$((Set\{\}) \rightarrow exists_{Set}(z \mid P(z))) = false$$

lemma *OclExists-including-exec[simp,code-unfold]* :

assumes *cp*: *cp P*

shows $((S \rightarrow including_{Set}(x)) \rightarrow exists_{Set}(z \mid P(z))) = (if \ \delta \ S \ and \ \forall \ x$
 $\quad then \ P \ x \ or \ (S \rightarrow exists_{Set}(z \mid P(z)))$
 $\quad else \ invalid$
 $\quad endif)$

2.9.25.11. Execution Rules on Iterate

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

In particular, this does hold for $A = null$.

lemma *OclIterate-including*:

assumes *S-finite*: $\tau \models \delta(S \rightarrow size_{Set}())$

and *F-valid-arg*: $(\forall A) \ \tau = (\forall (F \ a \ A)) \ \tau$

and *F-commute*: *comp-fun-commute F*

and *F-cp*: $\bigwedge x \ y \ \tau. F \ x \ y \ \tau = F \ (\lambda \ -. \ x \ \tau) \ y \ \tau$

shows $((S \rightarrow including_{Set}(a)) \rightarrow iterate_{Set}(a; x = A \mid F \ a \ x)) \ \tau =$
 $((S \rightarrow excluding_{Set}(a)) \rightarrow iterate_{Set}(a; x = F \ a \ A \mid F \ a \ x)) \ \tau$

2.9.25.12. Execution Rules on Select

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

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

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

assumes $P\text{-}cp : cp\ P$

shows $OclSelect\ (X \rightarrow including_{Set}(y))\ P = OclSelect\text{-}body\ P\ y\ (OclSelect\ (X \rightarrow excluding_{Set}(y))\ P)$

(is - = ?select)

2.9.25.13. Execution Rules on Reject

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

lemma *OclReject-including-exec*[simp,code-unfold]:

assumes $P\text{-}cp : cp\ P$

shows $OclReject\ (X \rightarrow including_{Set}(y))\ P = OclSelect\text{-}body\ (not\ o\ P)\ y\ (OclReject\ (X \rightarrow excluding_{Set}(y))\ P)$

2.9.25.14. Execution Rules Combining Previous Operators

OclIncluding

lemma *OclIncluding-idem0* :

assumes $\tau \models \delta\ S$

and $\tau \models v\ i$

shows $\tau \models (S \rightarrow including_{Set}(i) \rightarrow including_{Set}(i)) \hat{=} (S \rightarrow including_{Set}(i))$

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

OclExcluding

lemma *OclExcluding-idem0* :

assumes $\tau \models \delta\ S$

and $\tau \models v\ i$

shows $\tau \models (S \rightarrow excluding_{Set}(i) \rightarrow excluding_{Set}(i)) \hat{=} (S \rightarrow excluding_{Set}(i))$

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

OclIncludes

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

$X \rightarrow includes_{Set}(X \rightarrow any_{Set}()) = (if\ \delta\ X\ then$

$if\ \delta\ (X \rightarrow size_{Set}())\ then\ not(X \rightarrow isEmpty_{Set}())$

$else\ X \rightarrow includes_{Set}(null)\ endif$

$else\ invalid\ endif)$

OclSize

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

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

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

lemma [simp]:

assumes $X\text{-finite}$: $\bigwedge \tau. \text{finite } \ulcorner Rep\text{-Set}_{base} (X \ \tau) \urcorner$

shows $\delta ((X \rightarrow including_{Set}(x)) \rightarrow size_{Set}()) = (\delta(X)) \text{ and } v(x)$

OclForall

lemma *OclForall-rep-set-false*:

assumes $\tau \models \delta X$

shows $(OclForall \ X \ P \ \tau = false \ \tau) = (\exists x \in \ulcorner Rep\text{-Set}_{base} (X \ \tau) \urcorner. P (\lambda \tau. x) \ \tau = false \ \tau)$

lemma *OclForall-rep-set-true*:

assumes $\tau \models \delta X$

shows $(\tau \models OclForall \ X \ P) = (\forall x \in \ulcorner Rep\text{-Set}_{base} (X \ \tau) \urcorner. \tau \models P (\lambda \tau. x))$

lemma *OclForall-includes* :

assumes $x\text{-def}$: $\tau \models \delta x$

and $y\text{-def}$: $\tau \models \delta y$

shows $(\tau \models OclForall \ x \ (OclIncludes \ y)) = (\ulcorner Rep\text{-Set}_{base} (x \ \tau) \urcorner \subseteq \ulcorner Rep\text{-Set}_{base} (y \ \tau) \urcorner)$

lemma *OclForall-not-includes* :

assumes $x\text{-def}$: $\tau \models \delta x$

and $y\text{-def}$: $\tau \models \delta y$

shows $(OclForall \ x \ (OclIncludes \ y) \ \tau = false \ \tau) = (\neg \ulcorner Rep\text{-Set}_{base} (x \ \tau) \urcorner \subseteq \ulcorner Rep\text{-Set}_{base} (y \ \tau) \urcorner)$

lemma *OclForall-iterate*:

assumes $S\text{-finite}$: $\text{finite } \ulcorner Rep\text{-Set}_{base} (S \ \tau) \urcorner$

shows $S \rightarrow forAll_{Set}(x \mid P \ x) \ \tau = (S \rightarrow iterate_{Set}(x; acc = true \mid acc \text{ and } P \ x)) \ \tau$

lemma *OclForall-cong*:

assumes $\bigwedge x. x \in \ulcorner Rep\text{-Set}_{base} (X \ \tau) \urcorner \implies \tau \models P (\lambda \tau. x) \implies \tau \models Q (\lambda \tau. x)$

assumes P : $\tau \models OclForall \ X \ P$

shows $\tau \models OclForall \ X \ Q$

lemma *OclForall-cong'*:

assumes $\bigwedge x. x \in \ulcorner Rep\text{-Set}_{base} (X \ \tau) \urcorner \implies \tau \models P (\lambda \tau. x) \implies \tau \models Q (\lambda \tau. x) \implies \tau \models R (\lambda \tau. x)$

assumes P : $\tau \models OclForall \ X \ P$

assumes Q : $\tau \models OclForall \ X \ Q$

shows $\tau \models OclForall \ X \ R$

Strict Equality

lemma *StrictRefEq_{Set}-defined* :

assumes $x\text{-def}$: $\tau \models \delta x$

assumes $y\text{-def}$: $\tau \models \delta y$

shows $((x::(^{\mathcal{A}}, 'a::null)Set) \doteq y) \ \tau =$

$(x \rightarrow \text{forAll}_{\text{Set}}(\lambda y. \rightarrow \text{includes}_{\text{Set}}(z))) \text{ and } (y \rightarrow \text{forAll}_{\text{Set}}(\lambda x. \rightarrow \text{includes}_{\text{Set}}(z)))) \tau$

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

$((x::('A, 'a::\text{null})\text{Set}) \doteq y) =$
 (if δx then (if δy
 then $((x \rightarrow \text{forAll}_{\text{Set}}(\lambda y. \rightarrow \text{includes}_{\text{Set}}(z))) \text{ and } (y \rightarrow \text{forAll}_{\text{Set}}(\lambda x. \rightarrow \text{includes}_{\text{Set}}(z))))$
 else if $\forall y$
 then false — $x' \rightarrow \text{includes} = \text{null}$
 else invalid
 endif)
 else if $\forall x$ — $\text{null} = ???$
 then if $\forall y$ then $\text{not}(\delta y)$ else invalid endif
 else invalid
 endif
endif)

lemma *StrictRefEq_{Set}-L-subst1* : $cp P \implies \tau \models \forall x \implies \tau \models \forall y \implies \tau \models \forall P x \implies \tau \models \forall P y \implies$
 $\tau \models (x::('A, 'a::\text{null})\text{Set}) \doteq y \implies \tau \models (P x::('A, 'a::\text{null})\text{Set}) \doteq P y$

lemma *OclIncluding-cong'* :

shows $\tau \models \delta s \implies \tau \models \delta t \implies \tau \models \forall x \implies$
 $\tau \models ((s::('A, 'a::\text{null})\text{Set}) \doteq t) \implies \tau \models (s \rightarrow \text{including}_{\text{Set}}(x) \doteq (t \rightarrow \text{including}_{\text{Set}}(x)))$

lemma *OclIncluding-cong* : $\bigwedge (s::('A, 'a::\text{null})\text{Set}) t x y \tau. \tau \models \delta t \implies \tau \models \forall y \implies$
 $\tau \models s \doteq t \implies x = y \implies \tau \models s \rightarrow \text{including}_{\text{Set}}(x) \doteq (t \rightarrow \text{including}_{\text{Set}}(y))$

lemma *const-StrictRefEq_{Set}-empty* : $\text{const } X \implies \text{const } (X \doteq \text{Set}\{\})$

lemma *const-StrictRefEq_{Set}-including* :

$\text{const } a \implies \text{const } S \implies \text{const } X \implies \text{const } (X \doteq S \rightarrow \text{including}_{\text{Set}}(a))$

2.9.26. Test Statements

Assert $(\tau \models (\text{Set}\{\lambda \cdot. \text{⊥} x_{\perp}\} \doteq \text{Set}\{\lambda \cdot. \text{⊥} x_{\perp}\}))$

Assert $(\tau \models (\text{Set}\{\lambda \cdot. \text{⊥} x_{\perp}\} \doteq \text{Set}\{\lambda \cdot. \text{⊥} x_{\perp}\}))$

instantiation *Set_{base}* :: (equal)equal

begin

definition *HOL.equal* $k l \longleftrightarrow (k::('a::\text{equal})\text{Set}_{\text{base}}) = l$

instance

end

lemma *equal-Set_{base}-code* [code]:

$\text{HOL.equal } k (l::('a::\{\text{equal}, \text{null}\})\text{Set}_{\text{base}}) \longleftrightarrow \text{Rep-Set}_{\text{base}} k = \text{Rep-Set}_{\text{base}} l$

Assert $\tau \models (\text{Set}\{\} \doteq \text{Set}\{\})$

Assert $\tau \models (\text{Set}\{\mathbf{1}, \mathbf{2}\} \triangleq \text{Set}\{\} \rightarrow \text{including}_{\text{Set}}(\mathbf{2}) \rightarrow \text{including}_{\text{Set}}(\mathbf{1}))$

Assert $\tau \models (\text{Set}\{\mathbf{1}, \text{invalid}, \mathbf{2}\} \triangleq \text{invalid})$

Assert $\tau \models (\text{Set}\{\mathbf{1}, \mathbf{2}\} \rightarrow \text{including}_{\text{Set}}(\text{null}) \triangleq \text{Set}\{\text{null}, \mathbf{1}, \mathbf{2}\})$

definition *OclPrepend* :: $[(\mathcal{A}, \alpha :: \text{null}) \text{ Sequence}, (\mathcal{A}, \alpha) \text{ val}] \Rightarrow (\mathcal{A}, \alpha) \text{ Sequence}$

2.10.8. Definition: Union

definition $\text{OclUnion} :: [(\text{'A}, \text{'A}::\text{null}) \text{Sequence}, (\text{'A}, \text{'A}) \text{Sequence}] \Rightarrow (\text{'A}, \text{'A}) \text{Sequence}$

where $\text{OclUnion } x y = (\lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
 $\text{then } \text{Abs-Sequence}_{\text{base}} \llcorner \lceil \text{Rep-Sequence}_{\text{base}} (x \tau)^\top @$
 $\lceil \text{Rep-Sequence}_{\text{base}} (y \tau)^\top \llcorner$
 $\text{else invalid } \tau)$

notation $\text{OclUnion } (\langle \leftrightarrow \rangle_{\text{unionSeq}} '(-) \rangle)$

interpretation $\text{OclUnion} :$

$\text{profile-bin}_{d-d} \text{OclUnion } \lambda x y. \text{Abs-Sequence}_{\text{base}} \llcorner \lceil \text{Rep-Sequence}_{\text{base}} x^\top @ \lceil \text{Rep-Sequence}_{\text{base}} y^\top \llcorner$

2.10.9. Definition: At

definition $\text{OclAt} :: [(\text{'A}, \text{'A}::\text{null}) \text{Sequence}, (\text{'A}) \text{Integer}] \Rightarrow (\text{'A}, \text{'A}) \text{val}$

where $\text{OclAt } x y = (\lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
 $\text{then if } 1 \leq \lceil y \tau^\top \wedge \lceil y \tau^\top \leq \text{length} \lceil \text{Rep-Sequence}_{\text{base}} (x \tau)^\top$
 $\text{then } \lceil \text{Rep-Sequence}_{\text{base}} (x \tau)^\top ! (\text{nat } \lceil y \tau^\top - 1)$
 $\text{else invalid } \tau$
 $\text{else invalid } \tau)$

notation $\text{OclAt } (\langle \leftrightarrow \rangle_{\text{atSeq}} '(-) \rangle)$

2.10.10. Definition: First

definition $\text{OclFirst} :: [(\text{'A}, \text{'A}::\text{null}) \text{Sequence}] \Rightarrow (\text{'A}, \text{'A}) \text{val}$

where $\text{OclFirst } x = (\lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \text{ then}$
 $\text{case } \lceil \text{Rep-Sequence}_{\text{base}} (x \tau)^\top \text{ of } [] \Rightarrow \text{invalid } \tau$
 $\mid x \# - \Rightarrow x$
 $\text{else invalid } \tau)$

notation $\text{OclFirst } (\langle \leftrightarrow \rangle_{\text{firstSeq}} '(-) \rangle)$

2.10.11. Definition: Last

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

where $\text{OclLast } x = (\lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \text{ then}$
 $\text{if } \lceil \text{Rep-Sequence}_{\text{base}} (x \tau)^\top = [] \text{ then}$
 $\text{invalid } \tau$
 else
 $\text{last } \lceil \text{Rep-Sequence}_{\text{base}} (x \tau)^\top$
 $\text{else invalid } \tau)$

notation $\text{OclLast } (\langle \leftrightarrow \rangle_{\text{lastSeq}} '(-) \rangle)$

2.10.12. Definition: Iterate

definition $\text{OclIterate} :: [(\text{'A}, \text{'A}::\text{null}) \text{Sequence}, (\text{'A}, \text{'B}::\text{null}) \text{val},$
 $(\text{'A}, \text{'A}) \text{val} \Rightarrow (\text{'A}, \text{'B}) \text{val} \Rightarrow (\text{'A}, \text{'B}) \text{val}] \Rightarrow (\text{'A}, \text{'B}) \text{val}$

where $\text{OclIterate } S A F = (\lambda \tau. \text{if } (\delta S) \tau = \text{true } \tau \wedge (\delta A) \tau = \text{true } \tau$
 $\text{then } (\text{foldr } (F) (\text{map } (\lambda a \tau. a) \lceil \text{Rep-Sequence}_{\text{base}} (S \tau)^\top))(A) \tau$

else $\perp$)

syntax

-OclIterateSeq :: [($'\mathcal{A}, ' \alpha :: \text{null}$) Sequence, idt, idt, $' \alpha, ' \beta$] $\Rightarrow$ ($' \mathcal{A}, ' \gamma$)val
($\langle \cdot \rangle \rightarrow \text{iterateSeq} '(-; \text{--} | \cdot) \rangle$)

syntax-consts

-OclIterateSeq == OclIterate

translations

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

2.10.13. Definition: Forall

definition OclForall :: [($'\mathcal{A}, ' \alpha :: \text{null}$) Sequence, ($' \mathcal{A}, ' \alpha$)val $\Rightarrow$ ($' \mathcal{A}$)Boolean] $\Rightarrow$ $' \mathcal{A}$ Boolean

where OclForall $S \ P = (S \rightarrow \text{iterateSeq}(b; x = \text{true} \mid x \text{ and } (P \ b)))$

syntax

-OclForallSeq :: [($'\mathcal{A}, ' \alpha :: \text{null}$) Sequence, id, ($' \mathcal{A}$)Boolean] $\Rightarrow$ $' \mathcal{A}$ Boolean ($\langle \cdot \rangle \rightarrow \text{forallSeq} '(- | \cdot) \rangle$)

syntax-consts

-OclForallSeq == UML-Sequence.OclForall

translations

$X \rightarrow \text{forallSeq}(x \mid P) == \text{CONST UML-Sequence.OclForall } X \ (\%x. P)$

2.10.14. Definition: Exists

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

where OclExists $S \ P = (S \rightarrow \text{iterateSeq}(b; x = \text{false} \mid x \text{ or } (P \ b)))$

syntax

-OclExistSeq :: [($'\mathcal{A}, ' \alpha :: \text{null}$) Sequence, id, ($' \mathcal{A}$)Boolean] $\Rightarrow$ $' \mathcal{A}$ Boolean ($\langle \cdot \rangle \rightarrow \text{existsSeq} '(- | \cdot) \rangle$)

syntax-consts

-OclExistSeq == OclExists

translations

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

2.10.15. Definition: Collect

definition OclCollect :: [($'\mathcal{A}, ' \alpha :: \text{null}$) Sequence, ($' \mathcal{A}, ' \alpha$)val $\Rightarrow$ ($' \mathcal{A}, ' \beta$)val $\Rightarrow$ ($' \mathcal{A}, ' \beta :: \text{null}$) Sequence

where OclCollect $S \ P = (S \rightarrow \text{iterateSeq}(b; x = \text{Sequence}\{ \} \mid x \rightarrow \text{prependSeq}(P \ b)))$

syntax

-OclCollectSeq :: [($'\mathcal{A}, ' \alpha :: \text{null}$) Sequence, id, ($' \mathcal{A}$)Boolean] $\Rightarrow$ $' \mathcal{A}$ Boolean ($\langle \cdot \rangle \rightarrow \text{collectSeq} '(- | \cdot) \rangle$)

syntax-consts

-OclCollectSeq == OclCollect

translations

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

2.10.16. Definition: Select

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

where OclSelect $S \ P =$
($S \rightarrow \text{iterateSeq}(b; x = \text{Sequence}\{ \} \mid \text{if } P \ b \text{ then } x \rightarrow \text{prependSeq}(b) \text{ else } x \text{ endif})$)

syntax

-OclSelectSeq :: [(^ℓℳ, ^ℓα::null) Sequence, id, (^ℓℳ) Boolean] ⇒ ^ℓℳ Boolean (⟨(-)→selectSeq ^ℓ(-·)⟩)

syntax-consts

-OclSelectSeq == UML-Sequence.OclSelect

translations

$X \rightarrow \text{selectSeq}(x \mid P) == \text{CONST } \text{UML-Sequence.OclSelect } X \ (\%x. P)$

2.10.17. Definition: Size

definition OclSize :: [(^ℓℳ, ^ℓα::null) Sequence] ⇒ (^ℓℳ) Integer (⟨(-)→sizeSeq ^ℓ(·)⟩)

where OclSize S = (S→iterateSeq(b; x = 0 | x +_{int} 1))

2.10.18. Definition: IsEmpty

definition OclIsEmpty :: (^ℓℳ, ^ℓα::null) Sequence ⇒ ^ℓℳ Boolean

where OclIsEmpty x = ((∀ x and not (δ x)) or ((OclSize x) ≐ 0))

notation OclIsEmpty (⟨(-)→isEmptySeq ^ℓ(·)⟩)

2.10.19. Definition: NotEmpty

definition OclNotEmpty :: (^ℓℳ, ^ℓα::null) Sequence ⇒ ^ℓℳ Boolean

where OclNotEmpty x = not(OclIsEmpty x)

notation OclNotEmpty (⟨(-)→notEmptySeq ^ℓ(·)⟩)

2.10.20. Definition: Any

definition OclANY x = (λ τ.

if x τ = invalid τ then

⊥

else

case drop (drop (Rep-Sequence_{base} (x τ))) of [] ⇒ ⊥
| l ⇒ hd l)

notation OclANY (⟨(-)→anySeq ^ℓ(·)⟩)

2.10.21. Definition (future operators)

consts

OclCount :: [(^ℓℳ, ^ℓα::null) Sequence, (^ℓℳ, ^ℓα) Sequence] ⇒ ^ℓℳ Integer

OclSum :: (^ℓℳ, ^ℓα::null) Sequence ⇒ ^ℓℳ Integer

notation OclCount (⟨(-)→countSeq ^ℓ(·)⟩)

notation OclSum (⟨(-)→sumSeq ^ℓ(·)⟩)

2.10.22. Logical Properties

2.10.23. Execution Laws with Invalid or Null as Argument

OclIterate

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

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

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

2.10.24. General Algebraic Execution Rules

2.10.24.1. Execution Rules on Iterate

lemma *OclIterate-empty*[simp,code-unfold]:Sequence{ } $\rightarrow$ iterate_{Seq}(a; x = A | P a x) = A

In particular, this does hold for A = null.

lemma *OclIterate-including*[simp,code-unfold]:
assumes *strict1* : $\bigwedge X. P \text{ invalid } X = \text{invalid}$
and *P-valid-arg*: $\bigwedge \tau. (\forall A) \tau = (\forall (P a A)) \tau$
and *P-cp* : $\bigwedge x y \tau. P x y \tau = P (\lambda -. x \tau) y \tau$
and *P-cp'* : $\bigwedge x y \tau. P x y \tau = P x (\lambda -. y \tau) \tau$
shows (S $\rightarrow$ including_{Seq}(a)) $\rightarrow$ iterate_{Seq}(b; x = A | P b x) = S $\rightarrow$ iterate_{Seq}(b; x = P a A | P b x)

lemma *OclIterate-prepend*[simp,code-unfold]:
assumes *strict1* : $\bigwedge X. P \text{ invalid } X = \text{invalid}$
and *strict2* : $\bigwedge X. P X \text{ invalid} = \text{invalid}$
and *P-cp* : $\bigwedge x y \tau. P x y \tau = P (\lambda -. x \tau) y \tau$
and *P-cp'* : $\bigwedge x y \tau. P x y \tau = P x (\lambda -. y \tau) \tau$
shows (S $\rightarrow$ prepend_{Seq}(a)) $\rightarrow$ iterate_{Seq}(b; x = A | P b x) = P a (S $\rightarrow$ iterate_{Seq}(b; x = A | P b x))

2.10.25. Test Statements

Assert $\tau \models (\text{Sequence}\{ \} \doteq \text{Sequence}\{ \})$
Assert $\tau \models (\text{Sequence}\{ \mathbf{1}, \mathbf{2} \} \triangleq \text{Sequence}\{ \} \rightarrow \text{prepend}_{\text{Seq}}(\mathbf{2}) \rightarrow \text{prepend}_{\text{Seq}}(\mathbf{1}))$
Assert $\tau \models (\text{Sequence}\{ \mathbf{1}, \text{invalid}, \mathbf{2} \} \triangleq \text{invalid})$
Assert $\tau \models (\text{Sequence}\{ \mathbf{1}, \mathbf{2} \} \rightarrow \text{prepend}_{\text{Seq}}(\text{null}) \triangleq \text{Sequence}\{ \text{null}, \mathbf{1}, \mathbf{2} \})$
Assert $\tau \models (\text{Sequence}\{ \mathbf{1}, \mathbf{2} \} \rightarrow \text{including}_{\text{Seq}}(\text{null}) \triangleq \text{Sequence}\{ \mathbf{1}, \mathbf{2}, \text{null} \})$

2.11. Miscellaneous Stuff

2.11.1. Definition: asBoolean

definition *OclAsBoolean*_{Int} :: (^bℕ) Integer $\Rightarrow$ (^bℕ) Boolean ($\langle (-) \rightarrow \text{oclAsType}_{\text{Int}}'(\text{Boolean}') \rangle$)
where *OclAsBoolean*_{Int} X = ($\lambda \tau. \text{if } (\delta X) \tau = \text{true } \tau$
 then $\bigwedge X. \tau^{\top} \neq 0_{\perp}$
 else invalid τ)

interpretation $OclAsBoolean_{Int} : profile-mono_d \ OclAsBoolean_{Int} \ \lambda x. \ \perp \ulcorner x \urcorner \neq 0_{\perp}$

definition $OclAsBoolean_{Real} :: ('A) \ Real \Rightarrow ('A) \ Boolean \ (\langle(-) \rangle_{OclAsType_{Real}} 'Boolean')$
where $OclAsBoolean_{Real} \ X = (\lambda \tau. \text{if } (\delta \ X) \ \tau = true \ \tau$
 $\text{then } \perp \ulcorner X \ \tau \urcorner \neq 0_{\perp}$
 $\text{else } invalid \ \tau)$

interpretation $OclAsBoolean_{Real} : profile-mono_d \ OclAsBoolean_{Real} \ \lambda x. \ \perp \ulcorner x \urcorner \neq 0_{\perp}$

2.11.2. Definition: asInteger

definition $OclAsInteger_{Real} :: ('A) \ Real \Rightarrow ('A) \ Integer \ (\langle(-) \rangle_{OclAsType_{Real}} 'Integer')$
where $OclAsInteger_{Real} \ X = (\lambda \tau. \text{if } (\delta \ X) \ \tau = true \ \tau$
 $\text{then } \perp floor \ulcorner X \ \tau \urcorner$
 $\text{else } invalid \ \tau)$

interpretation $OclAsInteger_{Real} : profile-mono_d \ OclAsInteger_{Real} \ \lambda x. \ \perp floor \ulcorner x \urcorner$

2.11.3. Definition: asReal

definition $OclAsReal_{Int} :: ('A) \ Integer \Rightarrow ('A) \ Real \ (\langle(-) \rangle_{OclAsType_{Int}} 'Real')$
where $OclAsReal_{Int} \ X = (\lambda \tau. \text{if } (\delta \ X) \ \tau = true \ \tau$
 $\text{then } \perp real-of-int \ulcorner X \ \tau \urcorner$
 $\text{else } invalid \ \tau)$

interpretation $OclAsReal_{Int} : profile-mono_d \ OclAsReal_{Int} \ \lambda x. \ \perp real-of-int \ulcorner x \urcorner$

lemma *Integer-subtype-of-Real:*

assumes $\tau \models \delta \ X$

shows $\tau \models X \rightarrow_{OclAsType_{Int}(Real)} \rightarrow_{OclAsType_{Real}(Integer)} \triangleq X$

2.11.4. Definition: asPair

definition $OclAsPair_{Seq} :: [('A, 'A::null)Sequence] \Rightarrow ('A, 'A::null, 'A::null) \ Pair \ (\langle(-) \rangle_{asPair_{Seq}} 'Pair)$
where $OclAsPair_{Seq} \ S = (\text{if } S \rightarrow size_{Seq}() \doteq 2$
 $\text{then } Pair\{S \rightarrow at_{Seq}(0), S \rightarrow at_{Seq}(1)\}$
 $\text{else } invalid$
 $endif)$

definition $OclAsPair_{Set} :: [('A, 'A::null)Set] \Rightarrow ('A, 'A::null, 'A::null) \ Pair \ (\langle(-) \rangle_{asPair_{Set}} 'Pair)$
where $OclAsPair_{Set} \ S = (\text{if } S \rightarrow size_{Set}() \doteq 2$
 $\text{then let } v = S \rightarrow any_{Set}() \text{ in}$
 $\text{Pair}\{v, S \rightarrow excluding_{Set}(v) \rightarrow any_{Set}()\}$
 $\text{else } invalid$
 $endif)$

definition $OclAsPair_{Bag} :: [('A, 'A::null)Bag] \Rightarrow ('A, 'A::null, 'A::null) \ Pair \ (\langle(-) \rangle_{asPair_{Bag}} 'Pair)$
where $OclAsPair_{Bag} \ S = (\text{if } S \rightarrow size_{Bag}() \doteq 2$
 $\text{then let } v = S \rightarrow any_{Bag}() \text{ in}$

$Pair\{v, S \rightarrow excluding_{Bag}(v) \rightarrow any_{Bag}()\}$
 else invalid
 endif)

2.11.5. Definition: asSet

definition $OclAsSet_{Seq} :: [(^{\mathcal{A}}, ^{\alpha}::null)Sequence] \Rightarrow (^{\mathcal{A}}, ^{\alpha})Set (\langle(-) \rightarrow asSet_{Seq} (^{\mathcal{A}}) \rangle)$
where $OclAsSet_{Seq} S = (S \rightarrow iterate_{Seq}(b; x = Set\{\} \mid x \rightarrow including_{Set}(b)))$

definition $OclAsSet_{Pair} :: [(^{\mathcal{A}}, ^{\alpha}::null, ^{\alpha}::null) Pair] \Rightarrow (^{\mathcal{A}}, ^{\alpha})Set (\langle(-) \rightarrow asSet_{Pair} (^{\mathcal{A}}) \rangle)$
where $OclAsSet_{Pair} S = Set\{S.First(), S.Second()\}$

definition $OclAsSet_{Bag} :: (^{\mathcal{A}}, ^{\alpha}::null) Bag \Rightarrow (^{\mathcal{A}}, ^{\alpha})Set (\langle(-) \rightarrow asSet_{Bag} (^{\mathcal{A}}) \rangle)$
where $OclAsSet_{Bag} S = (\lambda \tau. \text{if } (\delta S) \tau = \text{true } \tau$
 then $Abs\text{-}Set_{base \perp} Rep\text{-}Set\text{-}base S \tau \perp$
 else if $(\vee S) \tau = \text{true } \tau$ then null τ
 else invalid τ)

2.11.6. Definition: asSequence

definition $OclAsSeq_{Set} :: [(^{\mathcal{A}}, ^{\alpha}::null)Set] \Rightarrow (^{\mathcal{A}}, ^{\alpha})Sequence (\langle(-) \rightarrow asSequence_{Set} (^{\mathcal{A}}) \rangle)$
where $OclAsSeq_{Set} S = (S \rightarrow iterate_{Set}(b; x = Sequence\{\} \mid x \rightarrow including_{Seq}(b)))$

definition $OclAsSeq_{Bag} :: [(^{\mathcal{A}}, ^{\alpha}::null)Bag] \Rightarrow (^{\mathcal{A}}, ^{\alpha})Sequence (\langle(-) \rightarrow asSequence_{Bag} (^{\mathcal{A}}) \rangle)$
where $OclAsSeq_{Bag} S = (S \rightarrow iterate_{Bag}(b; x = Sequence\{\} \mid x \rightarrow including_{Seq}(b)))$

definition $OclAsSeq_{Pair} :: [(^{\mathcal{A}}, ^{\alpha}::null, ^{\alpha}::null) Pair] \Rightarrow (^{\mathcal{A}}, ^{\alpha})Sequence (\langle(-) \rightarrow asSequence_{Pair} (^{\mathcal{A}}) \rangle)$
where $OclAsSeq_{Pair} S = Sequence\{S.First(), S.Second()\}$

2.11.7. Definition: asBag

definition $OclAsBag_{Seq} :: [(^{\mathcal{A}}, ^{\alpha}::null)Sequence] \Rightarrow (^{\mathcal{A}}, ^{\alpha})Bag (\langle(-) \rightarrow asBag_{Seq} (^{\mathcal{A}}) \rangle)$
where $OclAsBag_{Seq} S = (\lambda \tau. Abs\text{-}Bag_{base \perp} \lambda s. \text{if list-ex } ((=) s) \uparrow Rep\text{-}Sequence_{base} (S \tau) \uparrow \text{ then } 1 \text{ else } 0_{\perp})$

definition $OclAsBag_{Set} :: [(^{\mathcal{A}}, ^{\alpha}::null)Set] \Rightarrow (^{\mathcal{A}}, ^{\alpha})Bag (\langle(-) \rightarrow asBag_{Set} (^{\mathcal{A}}) \rangle)$
where $OclAsBag_{Set} S = (\lambda \tau. Abs\text{-}Bag_{base \perp} \lambda s. \text{if } s \in \uparrow Rep\text{-}Set_{base} (S \tau) \uparrow \text{ then } 1 \text{ else } 0_{\perp})$

lemma assumes $\tau \models \delta (S \rightarrow size_{Set}())$
shows $OclAsBag_{Set} S = (S \rightarrow iterate_{Set}(b; x = Bag\{\} \mid x \rightarrow including_{Bag}(b)))$

definition $OclAsBag_{Pair} :: [(^{\mathcal{A}}, ^{\alpha}::null, ^{\alpha}::null) Pair] \Rightarrow (^{\mathcal{A}}, ^{\alpha})Bag (\langle(-) \rightarrow asBag_{Pair} (^{\mathcal{A}}) \rangle)$
where $OclAsBag_{Pair} S = Bag\{S.First(), S.Second()\}$

2.11.8. Test Statements

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

Here is an example of a nested collection.

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

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

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

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

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

Elementary computations on Sets.

declare *OclSelect-body-def* [simp]

Assert $\neg (\tau \models v(\text{invalid}::('A, 'a::\text{null}) \text{Set}))$

Assert $\tau \models v(\text{null}::('A, 'a::\text{null}) \text{Set})$

Assert $\neg (\tau \models \delta(\text{null}::('A, 'a::\text{null}) \text{Set}))$

Assert $\tau \models v(\text{Set}\{\})$

Assert $\tau \models v(\text{Set}\{\text{Set}\{2\}, \text{null}\})$

Assert $\tau \models \delta(\text{Set}\{\text{Set}\{2\}, \text{null}\})$

Assert $\tau \models (\text{Set}\{2, 1\} \rightarrow \text{includes}_{\text{Set}}(1))$

Assert $\neg (\tau \models (\text{Set}\{2\} \rightarrow \text{includes}_{\text{Set}}(1)))$

Assert $\neg (\tau \models (\text{Set}\{2, 1\} \rightarrow \text{includes}_{\text{Set}}(\text{null})))$

Assert $\tau \models (\text{Set}\{2, \text{null}\} \rightarrow \text{includes}_{\text{Set}}(\text{null}))$

Assert $\tau \models (\text{Set}\{\text{null}, 2\} \rightarrow \text{includes}_{\text{Set}}(\text{null}))$

Assert $\tau \models ((\text{Set}\{\}) \rightarrow \text{forAll}_{\text{Set}}(z \mid 0 <_{\text{int}} z))$

Assert $\tau \models ((\text{Set}\{2, 1\}) \rightarrow \text{forAll}_{\text{Set}}(z \mid 0 <_{\text{int}} z))$

Assert $\neg (\tau \models ((\text{Set}\{2, 1\}) \rightarrow \text{exists}_{\text{Set}}(z \mid z <_{\text{int}} 0)))$

Assert $\neg (\tau \models (\delta(\text{Set}\{2, \text{null}\}) \rightarrow \text{forAll}_{\text{Set}}(z \mid 0 <_{\text{int}} z)))$

Assert $\neg (\tau \models ((\text{Set}\{2, \text{null}\}) \rightarrow \text{forAll}_{\text{Set}}(z \mid 0 <_{\text{int}} z)))$

Assert $\tau \models ((\text{Set}\{2, \text{null}\}) \rightarrow \text{exists}_{\text{Set}}(z \mid 0 <_{\text{int}} z))$

Assert $\neg (\tau \models (\text{Set}\{\text{null}::'a \text{ Boolean}\} \doteq \text{Set}\{\}))$

Assert $\neg (\tau \models (\text{Set}\{\text{null}::'a \text{ Integer}\} \doteq \text{Set}\{\}))$

Assert $\neg (\tau \models (\text{Set}\{\text{true}\} \doteq \text{Set}\{\text{false}\}))$

Assert $\neg (\tau \models (\text{Set}\{\text{true}, \text{true}\} \doteq \text{Set}\{\text{false}\}))$

lemma $\neg (\tau \models (\text{Set}\{2\} \doteq \text{Set}\{1\}))$

lemma $\tau \models (\text{Set}\{2, \text{null}, 2\} \doteq \text{Set}\{\text{null}, 2\})$

lemma $\tau \models (\text{Set}\{1, \text{null}, 2\} \triangleleft \text{Set}\{\text{null}, 2\})$

lemma $\tau \models (\text{Set}\{\text{Set}\{2, \text{null}\}\} \doteq \text{Set}\{\text{Set}\{\text{null}, 2\}\})$

lemma $\tau \models (\text{Set}\{\text{Set}\{2, \text{null}\}\} \triangleleft \text{Set}\{\text{Set}\{\text{null}, 2\}, \text{null}\})$

lemma $\tau \models (\text{Set}\{\text{null}\} \rightarrow \text{select}_{\text{Set}}(x \mid \text{not } x) \doteq \text{Set}\{\text{null}\})$

lemma $\tau \models (\text{Set}\{\text{null}\} \rightarrow \text{reject}_{\text{Set}}(x \mid \text{not } x) \doteq \text{Set}\{\text{null}\})$

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

Elementary computations on Sequences.

Assert $\neg (\tau \models v(\text{invalid}::('A, 'a::\text{null}) \text{Sequence}))$

Assert $\tau \models v(\text{null}::('A, 'a::\text{null}) \text{Sequence})$

Assert $\neg (\tau \models \delta(\text{null}::(\lambda, \alpha::\text{null}) \text{Sequence}))$

Assert $\tau \models v(\text{Sequence}\{\})$

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

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

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

3.1. Introduction: States over Typed Object Universes

In the following, we will refine the concepts of a user-defined data-model (implied by a class-diagram) as well as the notion of state used in the previous section to much more detail. Surprisingly, even without a concrete notion of an objects and a universe of object representation, the generic infrastructure of state-related operations is fairly rich.

3.1.1. Fundamental Properties on Objects: Core Referential Equality

3.1.1.1. Definition

Generic referential equality - to be used for instantiations with concrete object types ...

definition $StrictRefEq_{Object} :: ('A, 'a :: \{object, null\})val \Rightarrow ('A, 'a)val \Rightarrow ('A)Boolean$
where $StrictRefEq_{Object} x y$
 $\equiv \lambda \tau. \text{if } (\forall x) \tau = true \ \tau \wedge (\forall y) \tau = true \ \tau$
 $\text{then if } x \ \tau = null \vee y \ \tau = null$
 $\text{then } \perp x \ \tau = null \wedge y \ \tau = null \perp$
 $\text{else } \perp (oid-of \ (x \ \tau)) = (oid-of \ (y \ \tau)) \perp$
 $\text{else invalid } \tau$

3.1.1.2. Strictness and context passing

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

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

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

3.1.2. Logic and Algebraic Layer on Object

3.1.2.1. Validity and Definedness Properties

We derive the usual laws on definedness for (generic) object equality:

lemma $StrictRefEq_{Object}\text{-defargs} :$

$\tau \models (\text{StrictRefEq}_{\text{Object}} x (y :: (\lambda a. \{ \text{null}, \text{object} \}) \text{val})) \implies (\tau \models (v x)) \wedge (\tau \models (v y))$

lemma *defined-StrictRefEqObject-I:*

assumes $\text{val-}x : \tau \models v x$

assumes $\text{val-}x : \tau \models v y$

shows $\tau \models \delta (\text{StrictRefEq}_{\text{Object}} x y)$

lemma *StrictRefEqObject-def-homo :*

$\delta(\text{StrictRefEq}_{\text{Object}} x (y :: (\lambda a. \{ \text{null}, \text{object} \}) \text{val})) = ((v x) \text{ and } (v y))$

3.1.2.2. Symmetry

lemma *StrictRefEqObject-sym :*

assumes $x\text{-val} : \tau \models v x$

shows $\tau \models \text{StrictRefEq}_{\text{Object}} x x$

3.1.2.3. 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: $\forall \text{oid} \in \text{dom } \sigma. \text{oid} = \text{OidOf} \lceil \sigma(\text{oid}) \rceil$. This condition is also mentioned in [19, Annex A] and goes back to Richters [20]; 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 $\text{WFF} :: (\lambda a. \text{object}) \text{st} \Rightarrow \text{bool}$

where $\text{WFF } \tau = ((\forall x \in \text{ran}(\text{heap}(\text{fst } \tau)). \lceil \text{heap}(\text{fst } \tau) (\text{oid-of } x) \rceil = x) \wedge$
 $(\forall x \in \text{ran}(\text{heap}(\text{snd } \tau)). \lceil \text{heap}(\text{snd } \tau) (\text{oid-of } x) \rceil = x))$

It turns out that WFF is a key-concept for linking strict referential equality to logical equality: in well-formed states (i.e. those states where the self (oid-of) field contains the pointer to which the object is associated to in the state), referential equality coincides with logical equality.

We turn now to the generic definition of referential equality on objects: Equality on objects in a state is reduced to equality on the references to these objects. As in HOL-OCL [4, 6], we will store the reference of an object inside the object in a (ghost) field. By establishing certain invariants (“consistent state”), it can be assured that there is a “one-to-one-correspondence” of objects to their references—and therefore the definition below behaves as we expect.

Generic Referential Equality enjoys the usual properties: (quasi) reflexivity, symmetry, transitivity, substitutivity for defined values. For type-technical reasons, for each concrete object type, the equality $\doteq$ is defined by generic referential equality.

theorem *StrictRefEqObject-vs-StrongEq:*

assumes $\text{WFF} : \text{WFF } \tau$

and $\text{valid-}x : \tau \models (v x)$

and $\text{valid-}y : \tau \models (v y)$

and $x\text{-present-pre} : x \tau \in \text{ran} (\text{heap}(\text{fst } \tau))$

and $y\text{-present-pre} : y \tau \in \text{ran} (\text{heap}(\text{fst } \tau))$

and $x\text{-present-post} : x \tau \in \text{ran} (\text{heap}(\text{snd } \tau))$

and $y\text{-present-post} : y \tau \in \text{ran} (\text{heap}(\text{snd } \tau))$

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

theorem *StrictRefEq_{Object}-vs-StrongEq'*:

assumes *WFF*: $\text{WFF } \tau$

and *valid-x*: $\tau \models (\text{v } (x :: ('A::\text{object}, 'A::\{\text{null}, \text{object}\})\text{val}))$

and *valid-y*: $\tau \models (\text{v } y)$

and *oid-preserve*: $\bigwedge x. x \in \text{ran } (\text{heap}(\text{fst } \tau)) \vee x \in \text{ran } (\text{heap}(\text{snd } \tau)) \implies$

$H x \neq \perp \implies \text{oid-of } (H x) = \text{oid-of } x$

and *xy-together*: $x \tau \in H \text{ ' ran } (\text{heap}(\text{fst } \tau)) \wedge y \tau \in H \text{ ' ran } (\text{heap}(\text{fst } \tau)) \vee$

$x \tau \in H \text{ ' ran } (\text{heap}(\text{snd } \tau)) \wedge y \tau \in H \text{ ' ran } (\text{heap}(\text{snd } \tau))$

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

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.

3.2. Operations on Object

3.2.1. Initial States (for testing and code generation)

definition $\tau_0 :: ('A)\text{st}$

where $\tau_0 \equiv (\langle \text{heap} = \text{Map.empty}, \text{assocs} = \text{Map.empty} \rangle,$

$\langle \text{heap} = \text{Map.empty}, \text{assocs} = \text{Map.empty} \rangle)$

3.2.2. OclAllInstances

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

definition *OclAllInstances-generic* $:: ((('A::\text{object}) \text{st} \Rightarrow 'A \text{ state}) \Rightarrow ('A::\text{object} \rightarrow 'A) \Rightarrow ('A, 'A \text{ option option}) \text{Set})$

where *OclAllInstances-generic* $\text{fst-snd } H =$

$(\lambda \tau. \text{Abs-Set}_{\text{base}} \perp \text{Some } ((H \text{ ' ran } (\text{heap } (\text{fst-snd } \tau))) - \{ \text{None} \})) \perp$

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

lemma *OclAllInstances-generic-init-empty*:

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

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

lemma *represented-generic-objects-nonnull*:

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

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

lemma *represented-generic-objects-defined*:

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

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

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

definition *is-represented-in-state fst-snd* $x H \tau = (x \tau \in (\text{Some } o H) \cdot \text{ran } (\text{heap } (\text{fst-snd } \tau)))$

lemma *represented-generic-objects-in-state*:

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

shows *is-represented-in-state pre-post* $x H \tau$

lemma *state-update-vs-allInstances-generic-empty*:

assumes $[\text{simp}]: \bigwedge a. \text{pre-post } (\text{mk } a) = a$

shows $(\text{mk } (\text{heap} = \text{Map.empty}, \text{assocs} = A)) \models \text{OclAllInstances-generic pre-post Type} \doteq \text{Set}\{\}$

Here comes a couple of operational rules that allow to infer the value of `oclAllInstances` from the context τ . These rules are a special-case in the sense that they are the only rules that relate statements with *different* τ 's. For that reason, new concepts like “constant contexts P” are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

lemma *state-update-vs-allInstances-generic-including'*:

assumes $[\text{simp}]: \bigwedge a. \text{pre-post } (\text{mk } a) = a$

assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$

and $\text{Type Object} \neq \text{None}$

shows $(\text{OclAllInstances-generic pre-post Type})$

$(\text{mk } (\text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A))$

$=$

$((\text{OclAllInstances-generic pre-post Type}) \rightarrow \text{including}_{\text{Set}}(\lambda \cdot \cdot \cdot \perp \text{ drop } (\text{Type Object}) \cdot \cdot \cdot))$

$(\text{mk } (\text{heap} = \sigma', \text{assocs} = A))$

lemma *state-update-vs-allInstances-generic-including*:

assumes $[\text{simp}]: \bigwedge a. \text{pre-post } (\text{mk } a) = a$

assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$

and $\text{Type Object} \neq \text{None}$

shows $(\text{OclAllInstances-generic pre-post Type})$

$(\text{mk } (\text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A))$

$=$

$((\lambda \cdot. (\text{OclAllInstances-generic pre-post Type})$

$(\text{mk } (\text{heap} = \sigma', \text{assocs} = A))) \rightarrow \text{including}_{\text{Set}}(\lambda \cdot \cdot \cdot \perp \text{ drop } (\text{Type Object}) \cdot \cdot \cdot))$

$(\text{mk } (\text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A))$

lemma *state-update-vs-allInstances-generic-noincluding'*:

assumes $[\text{simp}]: \bigwedge a. \text{pre-post } (\text{mk } a) = a$

assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$

and $\text{Type Object} = \text{None}$

shows $(\text{OclAllInstances-generic pre-post Type})$

$(\text{mk } (\text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A))$

$=$

$(\text{OclAllInstances-generic pre-post Type})$

$(\text{mk } (\text{heap} = \sigma', \text{assocs} = A))$

theorem *state-update-vs-allInstances-generic-ntc*:

assumes $[\text{simp}]: \bigwedge a. \text{pre-post } (\text{mk } a) = a$

assumes *oid-def*: $oid \notin dom \sigma'$
and *non-type-conform*: $Type \ Object = None$
and *cp-ctxt*: $cp \ P$
and *const-ctxt*: $\bigwedge X. const \ X \implies const \ (P \ X)$
shows $(mk \ (\langle heap = \sigma' (oid \mapsto Object), assoc = A \rangle) \models P \ (OclAllInstances-generic \ pre-post \ Type)) =$
 $(mk \ (\langle heap = \sigma', assoc = A \rangle) \models P \ (OclAllInstances-generic \ pre-post \ Type))$
(is $(? \tau \models P \ ? \varphi) = (? \tau' \models P \ ? \varphi))$

theorem *state-update-vs-allInstances-generic-tc*:
assumes [*simp*]: $\bigwedge a. pre-post \ (mk \ a) = a$
assumes *oid-def*: $oid \notin dom \sigma'$
and *type-conform*: $Type \ Object \neq None$
and *cp-ctxt*: $cp \ P$
and *const-ctxt*: $\bigwedge X. const \ X \implies const \ (P \ X)$
shows $(mk \ (\langle heap = \sigma' (oid \mapsto Object), assoc = A \rangle) \models P \ (OclAllInstances-generic \ pre-post \ Type)) =$
 $(mk \ (\langle heap = \sigma', assoc = A \rangle) \models P \ ((OclAllInstances-generic \ pre-post \ Type)$
 $\rightarrow including_{Set}(\lambda \cdot. \lfloor (Type \ Object) \rfloor)))$
(is $(? \tau \models P \ ? \varphi) = (? \tau' \models P \ ? \varphi'))$

declare *OclAllInstances-generic-def* [*simp*]

3.2.2.1. OclAllInstances (@post)

definition *OclAllInstances-at-post* :: $('A :: object \mapsto ' \alpha) \Rightarrow ('A, ' \alpha \ option \ option) \ Set$
 $(\leftarrow .allInstances'(\cdot))$

where *OclAllInstances-at-post* = *OclAllInstances-generic* *snd*

lemma *OclAllInstances-at-post-defined*: $\tau \models \delta \ (H \ .allInstances())$

lemma $\tau_0 \models H \ .allInstances() \triangleq Set\{\}$

lemma *represented-at-post-objects-nonnul*:
assumes *A*: $\tau \models (((H::('A::object \mapsto ' \alpha)).allInstances()) \rightarrow includes_{Set}(x))$
shows $\tau \models not(x \triangleq null)$

lemma *represented-at-post-objects-defined*:
assumes *A*: $\tau \models (((H::('A::object \mapsto ' \alpha)).allInstances()) \rightarrow includes_{Set}(x))$
shows $\tau \models \delta \ (H \ .allInstances()) \wedge \tau \models \delta \ x$

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

lemma
assumes *A*: $\tau \models H \ .allInstances() \rightarrow includes_{Set}(x)$
shows *is-represented-in-state* *snd* *x* *H* τ

lemma *state-update-vs-allInstances-at-post-empty*:
shows $(\sigma, \langle heap = Map.empty, assoc = A \rangle) \models Type \ .allInstances() \triangleq Set\{\}$

Here comes a couple of operational rules that allow to infer the value of *oclAllInstances* from the context τ . These rules are a special-case in the sense that they are the only rules that relate statements with *different* τ 's. For that reason,

new concepts like “constant contexts P” are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

lemma *state-update-vs-allInstances-at-post-including'*:

assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$

and $\text{Type Object} \neq \text{None}$

shows $(\text{Type .allInstances}())$

$(\sigma, \llbracket \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \rrbracket)$

$=$

$((\text{Type .allInstances}()) \rightarrow \text{includingSet}(\lambda \cdot \perp \text{ drop } (\text{Type Object}) \perp))$

$(\sigma, \llbracket \text{heap} = \sigma', \text{assocs} = A \rrbracket)$

lemma *state-update-vs-allInstances-at-post-including*:

assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$

and $\text{Type Object} \neq \text{None}$

shows $(\text{Type .allInstances}())$

$(\sigma, \llbracket \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \rrbracket)$

$=$

$((\lambda \cdot (\text{Type .allInstances}())$

$(\sigma, \llbracket \text{heap} = \sigma', \text{assocs} = A \rrbracket)) \rightarrow \text{includingSet}(\lambda \cdot \perp \text{ drop } (\text{Type Object}) \perp))$

$(\sigma, \llbracket \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \rrbracket)$

lemma *state-update-vs-allInstances-at-post-noincluding'*:

assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$

and $\text{Type Object} = \text{None}$

shows $(\text{Type .allInstances}())$

$(\sigma, \llbracket \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \rrbracket)$

$=$

$(\text{Type .allInstances}())$

$(\sigma, \llbracket \text{heap} = \sigma', \text{assocs} = A \rrbracket)$

theorem *state-update-vs-allInstances-at-post-ntc*:

assumes *oid-def*: $\text{oid} \notin \text{dom } \sigma'$

and *non-type-conform*: $\text{Type Object} = \text{None}$

and *cp-ctxt*: $\text{cp } P$

and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P X)$

shows $((\sigma, \llbracket \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \rrbracket) \models (P(\text{Type .allInstances}())))) =$

$((\sigma, \llbracket \text{heap} = \sigma', \text{assocs} = A \rrbracket) \models (P(\text{Type .allInstances}()))))$

theorem *state-update-vs-allInstances-at-post-tc*:

assumes *oid-def*: $\text{oid} \notin \text{dom } \sigma'$

and *type-conform*: $\text{Type Object} \neq \text{None}$

and *cp-ctxt*: $\text{cp } P$

and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P X)$

shows $((\sigma, \llbracket \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \rrbracket) \models (P(\text{Type .allInstances}())))) =$

$((\sigma, \llbracket \text{heap} = \sigma', \text{assocs} = A \rrbracket) \models (P((\text{Type .allInstances}())$

$\rightarrow \text{includingSet}(\lambda \cdot \perp \text{ drop } (\text{Type Object}) \perp))))$

3.2.2.2. OclAllInstances (@pre)

definition $OclAllInstances\text{-}at\text{-}pre :: ('A :: object \rightarrow 'a) \Rightarrow ('A, 'a\ option\ option)\ Set$
 $(\leftarrow .allInstances@pre('))$

where $OclAllInstances\text{-}at\text{-}pre = OclAllInstances\text{-}generic\ fst$

lemma $OclAllInstances\text{-}at\text{-}pre\text{-}defined: \tau \models \delta\ (H.allInstances@pre())$

lemma $\tau_0 \models H.allInstances@pre() \triangleq Set\{\}$

lemma $represented\text{-}at\text{-}pre\text{-}objects\text{-}nonnull:$

assumes $A: \tau \models (((H::('A::object \rightarrow 'a)).allInstances@pre()) \rightarrow includes_{Set}(x))$

shows $\tau \models not(x \triangleq null)$

lemma $represented\text{-}at\text{-}pre\text{-}objects\text{-}defined:$

assumes $A: \tau \models (((H::('A::object \rightarrow 'a)).allInstances@pre()) \rightarrow includes_{Set}(x))$

shows $\tau \models \delta\ (H.allInstances@pre()) \wedge \tau \models \delta\ x$

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

lemma

assumes $A: \tau \models H.allInstances@pre() \rightarrow includes_{Set}(x)$

shows $is\text{-}represented\text{-}in\text{-}state\ fst\ x\ H\ \tau$

lemma $state\text{-}update\text{-}vs\text{-}allInstances\text{-}at\text{-}pre\text{-}empty:$

shows $((\llbracket heap=Map.empty, assocs=A \rrbracket, \sigma) \models Type.allInstances@pre() \triangleq Set\{\})$

Here comes a couple of operational rules that allow to infer the value of $oclAllInstances@pre$ from the context τ . These rules are a special-case in the sense that they are the only rules that relate statements with *different* τ 's. For that reason, new concepts like “constant contexts P” are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

lemma $state\text{-}update\text{-}vs\text{-}allInstances\text{-}at\text{-}pre\text{-}including':$

assumes $\bigwedge x. \sigma' oid = Some\ x \implies x = Object$

and $Type\ Object \neq None$

shows $(Type.allInstances@pre())$

$((\llbracket heap=\sigma'(oid \mapsto Object), assocs=A \rrbracket, \sigma)$

$=$

$((Type.allInstances@pre()) \rightarrow including_{Set}(\lambda \cdot \perp \cdot drop\ (Type\ Object)\ \perp))$

$((\llbracket heap=\sigma', assocs=A \rrbracket, \sigma)$

lemma $state\text{-}update\text{-}vs\text{-}allInstances\text{-}at\text{-}pre\text{-}including:$

assumes $\bigwedge x. \sigma' oid = Some\ x \implies x = Object$

and $Type\ Object \neq None$

shows $(Type.allInstances@pre())$

$((\llbracket heap=\sigma'(oid \mapsto Object), assocs=A \rrbracket, \sigma)$

$=$

$((\lambda \cdot (Type.allInstances@pre())$

$((\llbracket heap=\sigma', assocs=A \rrbracket, \sigma) \rightarrow including_{Set}(\lambda \cdot \perp \cdot drop\ (Type\ Object)\ \perp))$

$((\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=\text{A}), \sigma)$

lemma *state-update-vs-allInstances-at-pre-noincluding'*:

assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$

and $\text{Type Object} = \text{None}$

shows $(\text{Type} .\text{allInstances}@pre())$

$((\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=\text{A}), \sigma)$

$=$

$(\text{Type} .\text{allInstances}@pre())$

$((\text{heap}=\sigma', \text{assocs}=\text{A}), \sigma)$

theorem *state-update-vs-allInstances-at-pre-ntc*:

assumes *oid-def*: $\text{oid} \notin \text{dom } \sigma'$

and *non-type-conform*: $\text{Type Object} = \text{None}$

and *cp-ctxt*: $\text{cp } P$

and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P X)$

shows $((\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=\text{A}), \sigma) \models (P(\text{Type} .\text{allInstances}@pre())) =$

$((\text{heap}=\sigma', \text{assocs}=\text{A}), \sigma) \models (P(\text{Type} .\text{allInstances}@pre()))$

theorem *state-update-vs-allInstances-at-pre-tc*:

assumes *oid-def*: $\text{oid} \notin \text{dom } \sigma'$

and *type-conform*: $\text{Type Object} \neq \text{None}$

and *cp-ctxt*: $\text{cp } P$

and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P X)$

shows $((\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=\text{A}), \sigma) \models (P(\text{Type} .\text{allInstances}@pre())) =$

$((\text{heap}=\sigma', \text{assocs}=\text{A}), \sigma) \models (P((\text{Type} .\text{allInstances}@pre()) \rightarrow \text{includingSet}(\lambda -. \sqcup(\text{Type Object})_{\sqcup})))$

3.2.2.3. @post or @pre

theorem *StrictRefEqObject-vs-StrongEq''*:

assumes *WFF*: $\text{WFF } \tau$

and *valid-x*: $\tau \models (v \ (x :: ('A::\text{object}, 'B::\text{object option option})\text{val}))$

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

and *oid-preserve*: $\bigwedge x. x \in \text{ran}(\text{heap}(\text{fst } \tau)) \vee x \in \text{ran}(\text{heap}(\text{snd } \tau)) \implies$

$\text{oid-of } (H \ x) = \text{oid-of } x$

and *xy-together*: $\tau \models ((H .\text{allInstances}() \rightarrow \text{includesSet}(x) \text{ and } H .\text{allInstances}() \rightarrow \text{includesSet}(y)) \text{ or }$

$(H .\text{allInstances}@pre() \rightarrow \text{includesSet}(x) \text{ and } H .\text{allInstances}@pre() \rightarrow \text{includesSet}(y)))$

shows $(\tau \models (\text{StrictRefEqObject } x \ y)) = (\tau \models (x \triangleq y))$

3.2.3. OclIsNew, OclIsDeleted, OclIsMaintained, OclIsAbsent

definition *OclIsNew*:: $('A, 'B::\{\text{null}, \text{object}\})\text{val} \Rightarrow ('A)\text{Boolean} \ (\langle(-). \text{oclIsNew}'(\cdot)\rangle)$

where $X .\text{oclIsNew}() \equiv (\lambda \tau . \text{if } (\delta \ X) \ \tau = \text{true } \tau$

$\text{then } \sqcup \text{oid-of } (X \ \tau) \notin \text{dom}(\text{heap}(\text{fst } \tau)) \wedge$

$\text{oid-of } (X \ \tau) \in \text{dom}(\text{heap}(\text{snd } \tau))_{\sqcup}$

$\text{else invalid } \tau)$

The following predicates — which are not part of the OCL standard descriptions — complete the goal of `oclIsNew` by describing where an object belongs.

definition *OclIsDeleted*:: ($\mathcal{A}, \alpha::\{\text{null}, \text{object}\}$)val $\Rightarrow (\mathcal{A})\text{Boolean} \quad (\langle(-).oclIsDeleted\rangle')$
where $X.oclIsDeleted() \equiv (\lambda \tau . \text{if } (\delta X) \tau = \text{true } \tau$
 then $\perp_{oid-of} (X \tau) \in \text{dom}(\text{heap}(\text{fst } \tau)) \wedge$
 $oid-of (X \tau) \notin \text{dom}(\text{heap}(\text{snd } \tau))_{\perp}$
 else *invalid* τ)

definition *OclIsMaintained*:: ($\mathcal{A}, \alpha::\{\text{null}, \text{object}\}$)val $\Rightarrow (\mathcal{A})\text{Boolean} \quad (\langle(-).oclIsMaintained\rangle')$
where $X.oclIsMaintained() \equiv (\lambda \tau . \text{if } (\delta X) \tau = \text{true } \tau$
 then $\perp_{oid-of} (X \tau) \in \text{dom}(\text{heap}(\text{fst } \tau)) \wedge$
 $oid-of (X \tau) \in \text{dom}(\text{heap}(\text{snd } \tau))_{\perp}$
 else *invalid* τ)

definition *OclIsAbsent*:: ($\mathcal{A}, \alpha::\{\text{null}, \text{object}\}$)val $\Rightarrow (\mathcal{A})\text{Boolean} \quad (\langle(-).oclIsAbsent\rangle')$
where $X.oclIsAbsent() \equiv (\lambda \tau . \text{if } (\delta X) \tau = \text{true } \tau$
 then $\perp_{oid-of} (X \tau) \notin \text{dom}(\text{heap}(\text{fst } \tau)) \wedge$
 $oid-of (X \tau) \notin \text{dom}(\text{heap}(\text{snd } \tau))_{\perp}$
 else *invalid* τ)

lemma *state-split* : $\tau \models \delta X \Rightarrow$
 $\tau \models (X.oclIsNew()) \vee \tau \models (X.oclIsDeleted()) \vee$
 $\tau \models (X.oclIsMaintained()) \vee \tau \models (X.oclIsAbsent())$

lemma *notNew-vs-others* : $\tau \models \delta X \Rightarrow$
 $(\neg \tau \models (X.oclIsNew())) = (\tau \models (X.oclIsDeleted()) \vee$
 $\tau \models (X.oclIsMaintained()) \vee \tau \models (X.oclIsAbsent()))$

3.2.4. OclIsModifiedOnly

3.2.4.1. Definition

The following predicate—which is not part of the OCL standard—provides a simple, but powerful means to describe framing conditions. For any formal approach, be it animation of OCL contracts, test-case generation or die-hard theorem proving, the specification of the part of a system transition that *does not change* is of primordial importance. The following operator establishes the equality between old and new objects in the state (provided that they exist in both states), with the exception of those objects.

definition *OclIsModifiedOnly* :: ($\mathcal{A}::\text{object}, \alpha::\{\text{null}, \text{object}\}$)Set $\Rightarrow \mathcal{A} \text{ Boolean}$
 $(\langle-->oclIsModifiedOnly\rangle')$
where $X-->oclIsModifiedOnly() \equiv (\lambda (\sigma, \sigma').$
 let $X' = (oid-of \text{ ' }^{\top} \text{Rep-Set}_{\text{base}}(X(\sigma, \sigma'))^{\top})$;
 $S = ((\text{dom}(\text{heap } \sigma) \cap \text{dom}(\text{heap } \sigma')) - X')$
 in if $(\delta X) (\sigma, \sigma') = \text{true } (\sigma, \sigma') \wedge (\forall x \in \text{ }^{\top} \text{Rep-Set}_{\text{base}}(X(\sigma, \sigma'))^{\top}. x \neq \text{null})$
 then $\perp_{\forall} x \in S. (\text{heap } \sigma) x = (\text{heap } \sigma') x_{\perp}$
 else *invalid* (σ, σ'))

3.2.4.2. Execution with Invalid or Null or Null Element as Argument

lemma *invalid->oclIsModifiedOnly* = *invalid*

lemma $null \rightarrow oclIsModifiedOnly() = invalid$

lemma

assumes $X \text{-} null : \tau \models X \rightarrow includes_{Set}(null)$

shows $\tau \models X \rightarrow oclIsModifiedOnly() \triangleq invalid$

3.2.4.3. Context Passing

lemma $cp\text{-}OclIsModifiedOnly : X \rightarrow oclIsModifiedOnly() \tau = (\lambda \cdot. X \tau) \rightarrow oclIsModifiedOnly() \tau$

3.2.5. OclSelf

The following predicate—which is not part of the OCL standard—explicitly retrieves in the pre or post state the original OCL expression given as argument.

definition $[simp]: OclSelf\ x\ H\ fst\ snd = (\lambda \tau \cdot \text{if } (\delta\ x)\ \tau = true\ \tau$
 $\text{then if } oid\text{-of } (x\ \tau) \in dom(heap(fst\ \tau)) \wedge oid\text{-of } (x\ \tau) \in dom(heap\ (snd\ \tau))$
 $\text{then } H\ \lceil heap(fst\ snd\ \tau)(oid\text{-of } (x\ \tau)) \rceil$
 $\text{else } invalid\ \tau$
 $\text{else } invalid\ \tau)$

definition $OclSelf\text{-}at\text{-}pre :: ('A::object, 'a::\{null,object\})val \Rightarrow$
 $('A \Rightarrow 'a) \Rightarrow$
 $('A::object, 'a::\{null,object\})val \langle \langle \cdot \rangle \rangle @pre(\cdot) \rangle$

where $x @pre\ H = OclSelf\ x\ H\ fst$

definition $OclSelf\text{-}at\text{-}post :: ('A::object, 'a::\{null,object\})val \Rightarrow$
 $('A \Rightarrow 'a) \Rightarrow$
 $('A::object, 'a::\{null,object\})val \langle \langle \cdot \rangle \rangle @post(\cdot) \rangle$

where $x @post\ H = OclSelf\ x\ H\ snd$

3.2.6. Framing Theorem

lemma $all\text{-}oid\text{-}diff:$

assumes $def\text{-}x : \tau \models \delta\ x$

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

assumes $def\text{-}X' : \bigwedge x. x \in {}^{\top}Rep\text{-}Set_{base}(X\ \tau)^{\top} \implies x \neq null$

defines $P \equiv (\lambda a. not\ (StrictRefEq_{Object}\ x\ a))$

shows $(\tau \models X \rightarrow forAll_{Set}(al\ P\ a)) = (oid\text{-of } (x\ \tau) \notin oid\text{-of } \langle {}^{\top}Rep\text{-}Set_{base}(X\ \tau)^{\top} \rangle)$

theorem $framing:$

assumes $modifies\ clause : \tau \models (X \rightarrow excluding_{Set}(x)) \rightarrow oclIsModifiedOnly()$

and $oid\text{-is-type}\ repr : \tau \models X \rightarrow forAll_{Set}(al\ not\ (StrictRefEq_{Object}\ x\ a))$

shows $\tau \models (x @pre\ P \triangleq (x @post\ P))$

As corollary, the framing property can be expressed with only the strong equality as comparison operator.

theorem $framing'$:

assumes $wff : WFF\ \tau$

assumes $modifies\ clause : \tau \models (X \rightarrow excluding_{Set}(x)) \rightarrow oclIsModifiedOnly()$

and *oid-is-typerrepr* : $\tau \models X \rightarrow \text{forAll}_{\text{Set}}(\text{al not } (x \triangleq a))$
and *oid-preserve*: $\bigwedge x. x \in \text{ran } (\text{heap}(\text{fst } \tau)) \vee x \in \text{ran } (\text{heap}(\text{snd } \tau)) \implies$
 $\text{oid-of } (H x) = \text{oid-of } x$
and *xy-together*:
 $\tau \models X \rightarrow \text{forAll}_{\text{Set}}(y \mid (H .\text{allInstances}() \rightarrow \text{includes}_{\text{Set}}(x) \text{ and } H .\text{allInstances}() \rightarrow \text{includes}_{\text{Set}}(y)) \text{ or }$
 $(H .\text{allInstances}@pre() \rightarrow \text{includes}_{\text{Set}}(x) \text{ and } H .\text{allInstances}@pre() \rightarrow \text{includes}_{\text{Set}}(y)))$
shows $\tau \models (x @pre P \triangleq (x @post P))$

3.2.7. Miscellaneous

lemma *pre-post-new*: $\tau \models (x .\text{ocIsNew}()) \implies \neg (\tau \models v(x @pre H1)) \wedge \neg (\tau \models v(x @post H2))$
lemma *pre-post-old*: $\tau \models (x .\text{ocIsDeleted}()) \implies \neg (\tau \models v(x @pre H1)) \wedge \neg (\tau \models v(x @post H2))$
lemma *pre-post-absent*: $\tau \models (x .\text{ocIsAbsent}()) \implies \neg (\tau \models v(x @pre H1)) \wedge \neg (\tau \models v(x @post H2))$
lemma *pre-post-maintained*: $(\tau \models v(x @pre H1) \vee \tau \models v(x @post H2)) \implies \tau \models (x .\text{ocIsMaintained}())$
lemma *pre-post-maintained'*:
 $\tau \models (x .\text{ocIsMaintained}()) \implies (\tau \models v(x @pre (\text{Some } o H1)) \wedge \tau \models v(x @post (\text{Some } o H2)))$
lemma *framing-same-state*: $(\sigma, \sigma) \models (x @pre H \triangleq (x @post H))$

3.3. Accessors on Object

3.3.1. Definition

definition *select-object* $mt \text{ incl } \text{smash } \text{deref } l = \text{smash } (\text{foldl } \text{incl } mt (\text{map } \text{deref } l))$
 — *smash* returns null with *mt* in input (in this case, object contains null pointer)

The continuation *f* is usually instantiated with a smashing function which is either the identity *id* or, for $0 \dots 1$ cardinalities of associations, the *UML-Sequence.OclANY*-selector which also handles the *null*-cases appropriately. A standard use-case for this combinator is for example:

term $(\text{select-object } mt \text{ Set } \text{UML-Set.OclIncluding } \text{UML-Set.OclANY } f \text{ l oid })::('A, 'a::\text{null})\text{val}$

definition $\text{select-object}_{\text{Set}} = \text{select-object } mt \text{ Set } \text{UML-Set.OclIncluding } id$

definition $\text{select-object-any0}_{\text{Set}} f \text{ s-set} = \text{UML-Set.OclANY } (\text{select-object}_{\text{Set}} f \text{ s-set})$

definition $\text{select-object-any}_{\text{Set}} f \text{ s-set} =$

$(\text{let } s = \text{select-object}_{\text{Set}} f \text{ s-set} \text{ in}$

$\text{if } s \rightarrow \text{size}_{\text{Set}}() \triangleq 1 \text{ then}$

$s \rightarrow \text{any}_{\text{Set}}()$

else

$\perp$

$\text{endif})$

definition $\text{select-object}_{\text{Seq}} = \text{select-object } mt \text{ Sequence } \text{UML-Sequence.OclIncluding } id$

definition $\text{select-object-any}_{\text{Seq}} f \text{ s-set} = \text{UML-Sequence.OclANY } (\text{select-object}_{\text{Seq}} f \text{ s-set})$

definition $\text{select-object}_{\text{Pair}} f1 f2 = (\lambda(a,b). \text{OclPair } (f1 a) (f2 b))$

3.3.2. Validity and Definedness Properties

lemma *select-fold-execSeq*:

assumes *list-all* ($\lambda f. (\tau \models v f)$) *l*

shows $\ulcorner \text{Rep-Sequence}_{\text{base}} (\text{foldl } \text{UML-Sequence.OclIncluding } \text{Sequence}\{\} \text{ } l \ \tau) \urcorner = \text{List.map } (\lambda f. f \ \tau) \ l$

lemma *select-fold-execSet*:

assumes *list-all* ($\lambda f. (\tau \models v f)$) *l*

shows $\ulcorner \text{Rep-Set}_{\text{base}} (\text{foldl } \text{UML-Set.OclIncluding } \text{Set}\{\} \text{ } l \ \tau) \urcorner = \text{set } (\text{List.map } (\lambda f. f \ \tau) \ l)$

lemma *fold-val-elemSeq*:

assumes $\tau \models v$ (*foldl* *UML-Sequence.OclIncluding* *Sequence*{ } (*List.map* (*f p*) *s-set*))

shows *list-all* ($\lambda x. (\tau \models v (f p x))$) *s-set*

lemma *fold-val-elemSet*:

assumes $\tau \models v$ (*foldl* *UML-Set.OclIncluding* *Set*{ } (*List.map* (*f p*) *s-set*))

shows *list-all* ($\lambda x. (\tau \models v (f p x))$) *s-set*

lemma *select-object-any-definedSeq*:

assumes *def-sel*: $\tau \models \delta$ (*select-object-anySeq* *f s-set*)

shows *s-set* $\neq []$

lemma

assumes *def-sel*: $\tau \models \delta$ (*select-object-any0Set* *f s-set*)

shows *s-set* $\neq []$

lemma *select-object-any-definedSet*:

assumes *def-sel*: $\tau \models \delta$ (*select-object-anySet* *f s-set*)

shows *s-set* $\neq []$

lemma *select-object-any-exec0Seq*:

assumes *def-sel*: $\tau \models \delta$ (*select-object-anySeq* *f s-set*)

shows $\tau \models (\text{select-object-anySeq } f \ s\text{-set} \triangleq f \ (\text{hd } s\text{-set}))$

lemma *select-object-any-execSeq*:

assumes *def-sel*: $\tau \models \delta$ (*select-object-anySeq* *f s-set*)

shows $\exists e. \text{List.member } s\text{-set } e \wedge (\tau \models (\text{select-object-anySeq } f \ s\text{-set} \triangleq f \ e))$

lemma *select-object-any-execSet*:

assumes *def-sel*: $\tau \models \delta$ (*select-object-anySet* *f s-set*)

shows $\exists e. \text{List.member } s\text{-set } e \wedge (\tau \models (\text{select-object-anySet } f \ s\text{-set} \triangleq f \ e))$

Modeling of an operation contract for an operation with 2 arguments, (so depending on three parameters if one takes "self" into account).

locale *contract-scheme* =

fixes *f-v*

fixes *f-lam*

fixes *f* :: ($\text{'A}, \text{'A0}::\text{null}$)*val* $\Rightarrow$

'b $\Rightarrow$

($\text{'A}, \text{'res}::\text{null}$)*val*

fixes *PRE*

fixes *POST*

assumes *def-scheme*¹: $f \text{ self } x \equiv (\lambda \tau. \text{SOME } res. \text{let } res = \lambda -. res \text{ in}$
 $\text{if } (\tau \models (\delta \text{ self})) \wedge f\text{-v } x \tau$
 $\text{then } (\tau \models PRE \text{ self } x) \wedge$
 $(\tau \models POST \text{ self } x \text{ res})$
 $\text{else } \tau \models res \triangleq \text{invalid})$
assumes *all-post*¹: $\forall \sigma \sigma' \sigma''. ((\sigma, \sigma') \models PRE \text{ self } x) = ((\sigma, \sigma'') \models PRE \text{ self } x)$

assumes *cp_{PRE}*¹: $PRE (\text{self}) x \tau = PRE (\lambda -. \text{self } \tau) (f\text{-lam } x \tau) \tau$

assumes *cp_{POST}*¹: $POST (\text{self}) x (res) \tau = POST (\lambda -. \text{self } \tau) (f\text{-lam } x \tau) (\lambda -. res \tau) \tau$

assumes *f-v-val*: $\lambda a1. f\text{-v } (f\text{-lam } a1 \tau) \tau = f\text{-v } a1 \tau$

begin

lemma *strict0* [*simp*]: $f \text{ invalid } X = \text{invalid}$

lemma *nullstrict0* [*simp*]: $f \text{ null } X = \text{invalid}$

lemma *cp0*: $f \text{ self } a1 \tau = f (\lambda -. \text{self } \tau) (f\text{-lam } a1 \tau) \tau$

theorem *unfold*¹:

assumes *context-ok*: $cp \ E$

and *args-def-or-valid*: $(\tau \models \delta \text{ self}) \wedge f\text{-v } a1 \tau$

and *pre-satisfied*: $\tau \models PRE \text{ self } a1$

and *post-satisfiable*: $\exists res. (\tau \models POST \text{ self } a1 (\lambda -. res))$

and *sat-for-sols-post*: $(\wedge res. \tau \models POST \text{ self } a1 (\lambda -. res)) \implies \tau \models E (\lambda -. res)$

shows $\tau \models E(f \text{ self } a1)$

lemma *unfold2*¹:

assumes *context-ok*: $cp \ E$

and *args-def-or-valid*: $(\tau \models \delta \text{ self}) \wedge (f\text{-v } a1 \tau)$

and *pre-satisfied*: $\tau \models PRE \text{ self } a1$

and *postsplit-satisfied*: $\tau \models POST' \text{ self } a1$

and *post-decomposable* : $\wedge res. (POST \text{ self } a1 \text{ res}) =$
 $((POST' \text{ self } a1) \text{ and } (res \triangleq (BODY \text{ self } a1)))$

shows $(\tau \models E(f \text{ self } a1)) = (\tau \models E(BODY \text{ self } a1))$

end

locale *contract0* =

fixes $f :: ('A, 'A0::\text{null})\text{val} \Rightarrow$
 $('A, 'res::\text{null})\text{val}$

fixes *PRE*

fixes *POST*

assumes *def-scheme*: $f \text{ self } \equiv (\lambda \tau. \text{SOME } res. \text{let } res = \lambda -. res \text{ in}$

$\text{if } (\tau \models (\delta \text{ self}))$

$\text{then } (\tau \models PRE \text{ self}) \wedge$

$(\tau \models POST \text{ self } res)$

$\text{else } \tau \models res \triangleq \text{invalid})$

assumes *all-post*: $\forall \sigma \sigma' \sigma''. ((\sigma, \sigma') \models PRE \text{ self}) = ((\sigma, \sigma'') \models PRE \text{ self})$

assumes *cp_{PRE}*: $PRE (\text{self}) \tau = PRE (\lambda -. \text{self } \tau) \tau$

assumes *cp_{POST}*: $POST (\text{self}) (res) \tau = POST (\lambda -. \text{self } \tau) (\lambda -. res \tau) \tau$

sublocale *contract0* < *contract-scheme* $\lambda x \cdot \cdot$. *True* $\lambda x \cdot \cdot$. $x \lambda x \cdot \cdot$. $f x \lambda x \cdot \cdot$. *PRE* $x \lambda x \cdot \cdot$. *POST* x

context *contract0*

begin

lemma *cp-pre*: $cp \ self' \implies cp \ (\lambda X. \ PRE \ (self' \ X))$

lemma *cp-post*: $cp \ self' \implies cp \ res' \implies cp \ (\lambda X. \ POST \ (self' \ X) \ (res' \ X))$

lemma *cp [simp]*: $cp \ self' \implies cp \ res' \implies cp \ (\lambda X. \ f \ (self' \ X))$

lemmas *unfold* = *unfold'*[*simplified*]

lemma *unfold2* :

assumes $cp \ E$
and $(\tau \models \delta \ self)$
and $\tau \models PRE \ self$
and $\tau \models POST' \ self$
and $\bigwedge res. (POST \ self \ res) =$
 $((POST' \ self) \ and \ (res \triangleq (BODY \ self)))$
shows $(\tau \models E(f \ self)) = (\tau \models E(BODY \ self))$

end

locale *contract1* =

fixes *f* :: $(\alpha, \alpha 0 :: null)val \Rightarrow$
 $(\alpha, \alpha 1 :: null)val \Rightarrow$
 $(\alpha, res :: null)val$

fixes *PRE*

fixes *POST*

assumes *def-scheme*: $f \ self \ a1 \equiv$

$(\lambda \tau. \ SOME \ res. \ let \ res = \lambda \cdot \cdot. \ res \ in$
 $if \ (\tau \models (\delta \ self)) \wedge (\tau \models v \ a1)$
 $then \ (\tau \models PRE \ self \ a1) \wedge$
 $(\tau \models POST \ self \ a1 \ res)$
 $else \ \tau \models res \triangleq invalid)$

assumes *all-post*: $\forall \sigma \sigma' \sigma''. ((\sigma, \sigma') \models PRE \ self \ a1) = ((\sigma, \sigma'') \models PRE \ self \ a1)$

assumes *cpPRE*: $PRE \ (self) \ (a1) \ \tau = PRE \ (\lambda \cdot \cdot. \ self \ \tau) \ (\lambda \cdot \cdot. \ a1 \ \tau) \ \tau$

assumes *cpPOST*: $POST \ (self) \ (a1) \ (res) \ \tau = POST \ (\lambda \cdot \cdot. \ self \ \tau) (\lambda \cdot \cdot. \ a1 \ \tau) (\lambda \cdot \cdot. \ res \ \tau) \ \tau$

sublocale *contract1* < *contract-scheme* $\lambda a1 \ \tau. (\tau \models v \ a1) \ \lambda a1 \ \tau. (\lambda \cdot \cdot. \ a1 \ \tau)$

context *contract1*

begin

lemma *strict1[simp]*: $f \ self \ invalid = invalid$

lemma *defined-mono* : $\tau \models v(f \ Y \ Z) \implies (\tau \models \delta \ Y) \wedge (\tau \models v \ Z)$

lemma *cp-pre*: $cp \ self' \implies cp \ a1' \implies cp \ (\lambda X. \ PRE \ (self' \ X) \ (a1' \ X))$

lemma *cp-post*: $cp\ self' \implies cp\ a1' \implies cp\ res'$
 $\implies cp\ (\lambda X. POST\ (self'\ X)\ (a1'\ X)\ (res'\ X))$

lemma *cp [simp]*: $cp\ self' \implies cp\ a1' \implies cp\ res' \implies cp\ (\lambda X. f\ (self'\ X)\ (a1'\ X))$

lemmas *unfold* = *unfold'*
lemmas *unfold2* = *unfold2'*

end

locale *contract2* =

fixes *f* :: (^{'A},^{'α0}::*null*)*val* $\Rightarrow$
(^{'A},^{'α1}::*null*)*val* $\Rightarrow$ (^{'A},^{'α2}::*null*)*val* $\Rightarrow$
(^{'A},^{'res}::*null*)*val*

fixes *PRE*

fixes *POST*

assumes *def-scheme*: $f\ self\ a1\ a2 \equiv$

$(\lambda\ \tau. SOME\ res. let\ res = \lambda\ -. res\ in$
 $if\ (\tau \models (\delta\ self)) \wedge (\tau \models v\ a1) \wedge (\tau \models v\ a2)$
 $then\ (\tau \models PRE\ self\ a1\ a2) \wedge$
 $(\tau \models POST\ self\ a1\ a2\ res)$
 $else\ \tau \models res \triangleq invalid)$

assumes *all-post*: $\forall\ \sigma\ \sigma'\ \sigma''. ((\sigma, \sigma') \models PRE\ self\ a1\ a2) = ((\sigma, \sigma'') \models PRE\ self\ a1\ a2)$

assumes *cpPRE*: $PRE\ (self)\ (a1)\ (a2)\ \tau = PRE\ (\lambda\ -. self\ \tau)\ (\lambda\ -. a1\ \tau)\ (\lambda\ -. a2\ \tau)\ \tau$

assumes *cpPOST*: $\wedge res. POST\ (self)\ (a1)\ (a2)\ (res)\ \tau =$
 $POST\ (\lambda\ -. self\ \tau)(\lambda\ -. a1\ \tau)(\lambda\ -. a2\ \tau)\ (\lambda\ -. res\ \tau)\ \tau$

sublocale *contract2* < *contract-scheme* $\lambda(a1,a2)\ \tau. (\tau \models v\ a1) \wedge (\tau \models v\ a2)$
 $\lambda(a1,a2)\ \tau. (\lambda\ -. a1\ \tau, \lambda\ -. a2\ \tau)$
 $(\lambda x\ (a,b). f\ x\ a\ b)$
 $(\lambda x\ (a,b). PRE\ x\ a\ b)$
 $(\lambda x\ (a,b). POST\ x\ a\ b)$

context *contract2*

begin

lemma *strict0'[simp]*: $f\ invalid\ X\ Y = invalid$

lemma *nullstrict0'[simp]*: $f\ null\ X\ Y = invalid$

lemma *strict1[simp]*: $f\ self\ invalid\ Y = invalid$

lemma *strict2[simp]*: $f\ self\ X\ invalid = invalid$

lemma *defined-mono*: $\tau \models v(f\ X\ Y\ Z) \implies (\tau \models \delta\ X) \wedge (\tau \models v\ Y) \wedge (\tau \models v\ Z)$

lemma *cp-pre*: $cp\ self' \implies cp\ a1' \implies cp\ a2' \implies cp\ (\lambda X. PRE\ (self'\ X)\ (a1'\ X)\ (a2'\ X))$

lemma *cp-post*: $cp\ self' \implies cp\ a1' \implies cp\ a2' \implies cp\ res'$
 $\implies cp\ (\lambda X. POST\ (self'\ X)\ (a1'\ X)\ (a2'\ X)\ (res'\ X))$

lemma $cp0' : f \text{ self } a1 \ a2 \ \tau = f \ (\lambda \ -. \ \text{self } \tau) \ (\lambda \ -. \ a1 \ \tau) \ (\lambda \ -. \ a2 \ \tau) \ \tau$

lemma $cp \ [simp] : cp \ \text{self}' \implies cp \ a1' \implies cp \ a2' \implies cp \ \text{res}'$
 $\implies cp \ (\lambda X. f \ (\text{self}' X) \ (a1' X) \ (a2' X))$

theorem $unfold :$

assumes $cp \ E$
and $(\tau \models \delta \ \text{self}) \wedge (\tau \models v \ a1) \wedge (\tau \models v \ a2)$
and $\tau \models PRE \ \text{self } a1 \ a2$
and $\exists \text{res}. (\tau \models POST \ \text{self } a1 \ a2 \ (\lambda \ -. \ \text{res}))$
and $(\wedge \text{res}. \tau \models POST \ \text{self } a1 \ a2 \ (\lambda \ -. \ \text{res}) \implies \tau \models E \ (\lambda \ -. \ \text{res}))$
shows $\tau \models E(f \ \text{self } a1 \ a2)$

lemma $unfold2 :$

assumes $cp \ E$
and $(\tau \models \delta \ \text{self}) \wedge (\tau \models v \ a1) \wedge (\tau \models v \ a2)$
and $\tau \models PRE \ \text{self } a1 \ a2$
and $\tau \models POST' \ \text{self } a1 \ a2$
and $\wedge \text{res}. (POST \ \text{self } a1 \ a2 \ \text{res}) =$
 $((POST' \ \text{self } a1 \ a2) \ \text{and} \ (\text{res} \triangleq (BODY \ \text{self } a1 \ a2)))$
shows $(\tau \models E(f \ \text{self } a1 \ a2)) = (\tau \models E(BODY \ \text{self } a1 \ a2))$

end

locale $contract3 =$

fixes $f :: ('A, 'A0::\text{null})\text{val} \Rightarrow$
 $('A, 'A1::\text{null})\text{val} \Rightarrow$
 $('A, 'A2::\text{null})\text{val} \Rightarrow$
 $('A, 'A3::\text{null})\text{val} \Rightarrow$
 $('A, 'res::\text{null})\text{val}$

fixes PRE

fixes $POST$

assumes $\text{def-scheme} : f \ \text{self } a1 \ a2 \ a3 \equiv$

$(\lambda \ \tau. \text{SOME } \text{res}. \text{let } \text{res} = \lambda \ -. \ \text{res} \text{ in}$
 $\text{if } (\tau \models (\delta \ \text{self})) \wedge (\tau \models v \ a1) \wedge (\tau \models v \ a2) \wedge (\tau \models v \ a3)$
 $\text{then } (\tau \models PRE \ \text{self } a1 \ a2 \ a3) \wedge$
 $(\tau \models POST \ \text{self } a1 \ a2 \ a3 \ \text{res})$
 $\text{else } \tau \models \text{res} \triangleq \text{invalid})$

assumes $\text{all-post} : \forall \ \sigma \ \sigma' \ \sigma''. ((\sigma, \sigma') \models PRE \ \text{self } a1 \ a2 \ a3) = ((\sigma, \sigma'') \models PRE \ \text{self } a1 \ a2 \ a3)$

assumes $cp_{PRE} : PRE \ (\text{self}) \ (a1) \ (a2) \ (a3) \ \tau = PRE \ (\lambda \ -. \ \text{self } \tau) \ (\lambda \ -. \ a1 \ \tau) \ (\lambda \ -. \ a2 \ \tau) \ (\lambda \ -. \ a3 \ \tau) \ \tau$

assumes $cp_{POST} : \wedge \text{res}. POST \ (\text{self}) \ (a1) \ (a2) \ (a3) \ (\text{res}) \ \tau =$
 $POST \ (\lambda \ -. \ \text{self } \tau) \ (\lambda \ -. \ a1 \ \tau) \ (\lambda \ -. \ a2 \ \tau) \ (\lambda \ -. \ a3 \ \tau) \ (\lambda \ -. \ \text{res } \tau) \ \tau$

sublocale $contract3 < contract\text{-scheme } \lambda(a1, a2, a3) \ \tau. (\tau \models v \ a1) \wedge (\tau \models v \ a2) \wedge (\tau \models v \ a3)$

$\lambda(a1, a2, a3) \ \tau. (\lambda \ -. a1 \ \tau, \lambda \ -. a2 \ \tau, \lambda \ -. a3 \ \tau)$
 $(\lambda x \ (a, b, c). f \ x \ a \ b \ c)$
 $(\lambda x \ (a, b, c). PRE \ x \ a \ b \ c)$
 $(\lambda x \ (a, b, c). POST \ x \ a \ b \ c)$

context $contract3$

begin

lemma *strict0*[simp] : $f \text{ invalid } X \ Y \ Z = \text{invalid}$

lemma *nullstrict0*[simp]: $f \text{ null } X \ Y \ Z = \text{invalid}$

lemma *strict1*[simp]: $f \text{ self invalid } Y \ Z = \text{invalid}$

lemma *strict2*[simp]: $f \text{ self } X \text{ invalid } Z = \text{invalid}$

lemma *defined-mono* : $\tau \models v(f \ W \ X \ Y \ Z) \implies (\tau \models \delta \ W) \wedge (\tau \models v \ X) \wedge (\tau \models v \ Y) \wedge (\tau \models v \ Z)$

lemma *cp-pre*: $cp \ self' \implies cp \ a1' \implies cp \ a2' \implies cp \ a3'$
 $\implies cp \ (\lambda X. \ PRE \ (self' \ X) \ (a1' \ X) \ (a2' \ X) \ (a3' \ X))$

lemma *cp-post*: $cp \ self' \implies cp \ a1' \implies cp \ a2' \implies cp \ a3' \implies cp \ res'$
 $\implies cp \ (\lambda X. \ POST \ (self' \ X) \ (a1' \ X) \ (a2' \ X) \ (a3' \ X) \ (res' \ X))$

lemma *cp0'*: $f \ self \ a1 \ a2 \ a3 \ \tau = f \ (\lambda \ -. \ self \ \tau) \ (\lambda \ -. \ a1 \ \tau) \ (\lambda \ -. \ a2 \ \tau) \ (\lambda \ -. \ a3 \ \tau) \ \tau$

lemma *cp* [simp]: $cp \ self' \implies cp \ a1' \implies cp \ a2' \implies cp \ a3' \implies cp \ res'$
 $\implies cp \ (\lambda X. f \ (self' \ X) \ (a1' \ X) \ (a2' \ X) \ (a3' \ X))$

theorem *unfold* :

assumes $cp \ E$
and $(\tau \models \delta \ self) \wedge (\tau \models v \ a1) \wedge (\tau \models v \ a2) \wedge (\tau \models v \ a3)$
and $\tau \models PRE \ self \ a1 \ a2 \ a3$
and $\exists res. (\tau \models POST \ self \ a1 \ a2 \ a3 \ (\lambda \ -. \ res))$
and $(\wedge res. \tau \models POST \ self \ a1 \ a2 \ a3 \ (\lambda \ -. \ res) \implies \tau \models E \ (\lambda \ -. \ res))$
shows $\tau \models E(f \ self \ a1 \ a2 \ a3)$

lemma *unfold2* :

assumes $cp \ E$
and $(\tau \models \delta \ self) \wedge (\tau \models v \ a1) \wedge (\tau \models v \ a2) \wedge (\tau \models v \ a3)$
and $\tau \models PRE \ self \ a1 \ a2 \ a3$
and $\tau \models POST' \ self \ a1 \ a2 \ a3$
and $\wedge res. (POST \ self \ a1 \ a2 \ a3 \ res) =$
 $((POST' \ self \ a1 \ a2 \ a3) \text{ and } (res \triangleq (BODY \ self \ a1 \ a2 \ a3)))$
shows $(\tau \models E(f \ self \ a1 \ a2 \ a3)) = (\tau \models E(BODY \ self \ a1 \ a2 \ a3))$

end

4. Example: The Employee Analysis Model

4.1. Introduction

For certain concepts like classes and class-types, only a generic definition for its resulting semantics can be given. Generic means, there is a function outside HOL that “compiles” a concrete, closed-world class diagram into a “theory” of this data model, consisting of a bunch of definitions for classes, accessors, method, casts, and tests for actual types, as well as proofs for the fundamental properties of these operations in this concrete data model.

Such generic function or “compiler” can be implemented in Isabelle on the ML level. This has been done, for a semantics following the open-world assumption, for UML 2.0 in [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.

4.1.1. Outlining the Example

We are presenting here an “analysis-model” of the (slightly modified) example Figure 7.3, page 20 of the OCL standard [19]. 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 http://isa-afp.org/entries/Featherweight_OCL.shtml). To be precise, this theory contains the formalization of the data-part covered by the UML class model (see Figure 4.1):

This means that the association (attached to the association class `EmployeeRanking`) with the association ends `boss` and `employees` is implemented by the attribute `boss` and the operation `employees` (to be discussed in the OCL part captured by the subsequent theory).

4.2. Example Data-Universe and its Infrastructure

Ideally, the following is generated automatically from a UML class model.

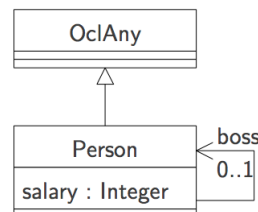


Figure 4.1. — A simple UML class model drawn from Figure 7.3, page 20 of [19].

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  $\mathcal{A}$  = 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    =  $\mathcal{A}$  Boolean
type-synonym Integer   =  $\mathcal{A}$  Integer
type-synonym Void      =  $\mathcal{A}$  Void
type-synonym OclAny     = ( $\mathcal{A}$ , typeOclAny option option) val
type-synonym Person     = ( $\mathcal{A}$ , typePerson option option) val
type-synonym Set-Integer = ( $\mathcal{A}$ , int option option) Set
type-synonym Set-Person  = ( $\mathcal{A}$ , typePerson option option) Set
```

Just a little check:

```
typ Boolean
```

To reuse key-elements of the library like referential equality, we have to show that the object universe belongs to the type class “oclaney,” i. e., each class type has to provide a function *oid-of* yielding the object id (oid) of the object.

```
instantiation typePerson :: object
begin
  definition oid-of-typePerson-def: oid-of x = (case x of mkPerson oid -  $\Rightarrow$  oid)
  instance
end
```

```
instantiation typeOclAny :: object
begin
  definition oid-of-typeOclAny-def: oid-of x = (case x of mkOclAny oid -  $\Rightarrow$  oid)
  instance
end
```

```
instantiation  $\mathcal{A}$  :: object
begin
  definition oid-of- $\mathcal{A}$ -def: oid-of x = (case x of
    inPerson person  $\Rightarrow$  oid-of person
  | inOclAny oclany  $\Rightarrow$  oid-of oclany)
  instance
end
```

4.3. Instantiation of the Generic Strict Equality

We instantiate the referential equality on *Person* and *OclAny*

overloading *StrictRefEq* $\equiv$ *StrictRefEq* :: [*Person*,*Person*] $\Rightarrow$ *Boolean*

begin

definition *StrictRefEqObject*-*Person* : (*x*::*Person*) $\dot{=}$ *y* $\equiv$ *StrictRefEqObject* *x y*

end

overloading *StrictRefEq* $\equiv$ *StrictRefEq* :: [*OclAny*,*OclAny*] $\Rightarrow$ *Boolean*

begin

definition *StrictRefEqObject*-*OclAny* : (*x*::*OclAny*) $\dot{=}$ *y* $\equiv$ *StrictRefEqObject* *x y*

end

lemmas *cps23* =

cp-StrictRefEqObject [of *x*::*Person* *y*::*Person* τ ,
simplified *StrictRefEqObject*-*Person* [symmetric]]
cp-intro(9) [of *P*::*Person* $\Rightarrow$ *Person* *Q*::*Person* $\Rightarrow$ *Person*,
simplified *StrictRefEqObject*-*Person* [symmetric]]
StrictRefEqObject-def [of *x*::*Person* *y*::*Person*,
simplified *StrictRefEqObject*-*Person* [symmetric]]
StrictRefEqObject-defargs [of - *x*::*Person* *y*::*Person*,
simplified *StrictRefEqObject*-*Person* [symmetric]]
StrictRefEqObject-strict1
[of *x*::*Person*,
simplified *StrictRefEqObject*-*Person* [symmetric]]
StrictRefEqObject-strict2
[of *x*::*Person*,
simplified *StrictRefEqObject*-*Person* [symmetric]]

for *x y* τ *P Q*

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.

4.4. OclAsType

4.4.1. Definition

consts *OclAsType*_{*OclAny*} :: ' $\alpha \Rightarrow$ *OclAny* ($\langle(-) .oclAsType' (OclAny') \rangle$)

consts *OclAsType*_{*Person*} :: ' $\alpha \Rightarrow$ *Person* ($\langle(-) .oclAsType' (Person') \rangle$)

definition *OclAsType*_{*OclAny*}- $\mathcal{A}$ = ($\lambda u. \text{case } u \text{ of in}_{OclAny} a \Rightarrow a$
| $\text{in}_{Person} (mk_{Person} \text{oid } a) \Rightarrow mk_{OclAny} \text{oid } \lfloor a_{\perp} \rfloor$)

lemma *OclAsType*_{*OclAny*}- $\mathcal{A}$ -some: *OclAsType*_{*OclAny*}- $\mathcal{A}$ *x* $\neq$ *None*

overloading *OclAsType*_{*OclAny*} $\equiv$ *OclAsType*_{*OclAny*} :: *OclAny* $\Rightarrow$ *OclAny*

begin

definition *OclAsType*_{*OclAny*}-*OclAny*:

(*X*::*OclAny*) *.oclAsType* (*OclAny*) $\equiv$ *X*

end

overloading $OclAsType_{OclAny} \equiv OclAsType_{OclAny} :: Person \Rightarrow OclAny$

begin

definition $OclAsType_{OclAny}\text{-}Person$:

$(X::Person) .oclAsType(OclAny) \equiv$

$(\lambda \tau. \text{case } X \ \tau \text{ of}$

$\perp \Rightarrow \text{invalid } \tau$

$\mid \perp \Rightarrow \text{null } \tau$

$\mid \text{mk}_{Person} \text{ oid } a \Rightarrow \perp (mk_{OclAny} \text{ oid } \perp)$

end

definition $OclAsType_{Person}\text{-}\mathcal{A} =$

$(\lambda u. \text{case } u \text{ of } in_{Person} p \Rightarrow \perp p$

$\mid in_{OclAny} (mk_{OclAny} \text{ oid } \perp) \Rightarrow \text{mk}_{Person} \text{ oid } a$

$\mid - \Rightarrow \text{None})$

overloading $OclAsType_{Person} \equiv OclAsType_{Person} :: OclAny \Rightarrow Person$

begin

definition $OclAsType_{Person}\text{-}OclAny$:

$(X::OclAny) .oclAsType(Person) \equiv$

$(\lambda \tau. \text{case } X \ \tau \text{ of}$

$\perp \Rightarrow \text{invalid } \tau$

$\mid \perp \Rightarrow \text{null } \tau$

$\mid \text{mk}_{OclAny} \text{ oid } \perp \Rightarrow \text{invalid } \tau \quad \text{— down-cast exception}$

$\mid \text{mk}_{OclAny} \text{ oid } a \Rightarrow \text{mk}_{Person} \text{ oid } a$

end

overloading $OclAsType_{Person} \equiv OclAsType_{Person} :: Person \Rightarrow Person$

begin

definition $OclAsType_{Person}\text{-}Person$:

$(X::Person) .oclAsType(Person) \equiv X$

end

4.4.2. Execution with Invalid or Null as Argument

lemma $OclAsType_{OclAny}\text{-}OclAny\text{-}strict : (\text{invalid}::OclAny) .oclAsType(OclAny) = \text{invalid}$

lemma $OclAsType_{OclAny}\text{-}OclAny\text{-}nullstrict : (\text{null}::OclAny) .oclAsType(OclAny) = \text{null}$

lemma $OclAsType_{OclAny}\text{-}Person\text{-}strict[simp] : (\text{invalid}::Person) .oclAsType(OclAny) = \text{invalid}$

lemma $OclAsType_{OclAny}\text{-}Person\text{-}nullstrict[simp] : (\text{null}::Person) .oclAsType(OclAny) = \text{null}$

lemma $OclAsType_{Person}\text{-}OclAny\text{-}strict[simp] : (\text{invalid}::OclAny) .oclAsType(Person) = \text{invalid}$

lemma $OclAsType_{Person}\text{-}OclAny\text{-}nullstrict[simp] : (\text{null}::OclAny) .oclAsType(Person) = \text{null}$

lemma $OclAsType_{Person}\text{-}Person\text{-}strict : (\text{invalid}::Person) .oclAsType(Person) = \text{invalid}$

lemma $OclAsType_{Person}\text{-}Person\text{-}nullstrict : (\text{null}::Person) .oclAsType(Person) = \text{null}$

4.5. OclIsTypeOf

4.5.1. Definition

consts $OclIsTypeOf_{OclAny} :: 'a \Rightarrow \text{Boolean } (\lambda(-).oclIsTypeOf' (OclAny'))$

consts $OclIsTypeOf_{Person} :: 'a \Rightarrow \text{Boolean } (\lambda(-).oclIsTypeOf' (Person'))$

overloading $OclIsTypeOf_{OclAny} \equiv OclIsTypeOf_{OclAny} :: OclAny \Rightarrow Boolean$

begin

definition $OclIsTypeOf_{OclAny-OclAny}$:

$(X::OclAny) .oclIsTypeOf(OclAny) \equiv$

$(\lambda \tau. \text{case } X \text{ of}$

$\perp \Rightarrow \text{invalid } \tau$

$\mid \perp_{\perp} \Rightarrow \text{true } \tau \text{ — invalid ??}$

$\mid \perp_{mk_{OclAny} \text{ oid } \perp_{\perp}} \Rightarrow \text{true } \tau$

$\mid \perp_{mk_{OclAny} \text{ oid } \perp_{\perp}} \Rightarrow \text{false } \tau)$

end

lemma $OclIsTypeOf_{OclAny-OclAny}'$:

$(X::OclAny) .oclIsTypeOf(OclAny) =$

$(\lambda \tau. \text{if } \tau \models \forall X \text{ then } (\text{case } X \text{ of}$

$\perp_{\perp} \Rightarrow \text{true } \tau \text{ — invalid ??}$

$\mid \perp_{mk_{OclAny} \text{ oid } \perp_{\perp}} \Rightarrow \text{true } \tau$

$\mid \perp_{mk_{OclAny} \text{ oid } \perp_{\perp}} \Rightarrow \text{false } \tau)$

$\text{else invalid } \tau)$

interpretation $OclIsTypeOf_{OclAny-OclAny}$:

profile-mono-schemeV

$OclIsTypeOf_{OclAny} :: OclAny \Rightarrow Boolean$

$\lambda X. (\text{case } X \text{ of}$

$\perp_{None} \Rightarrow \perp_{True} \text{ — invalid ??}$

$\mid \perp_{mk_{OclAny} \text{ oid } None} \Rightarrow \perp_{True}$

$\mid \perp_{mk_{OclAny} \text{ oid } \perp_{\perp}} \Rightarrow \perp_{False})$

overloading $OclIsTypeOf_{OclAny} \equiv OclIsTypeOf_{OclAny} :: Person \Rightarrow Boolean$

begin

definition $OclIsTypeOf_{OclAny-Person}$:

$(X::Person) .oclIsTypeOf(Person) \equiv$

$(\lambda \tau. \text{case } X \text{ of}$

$\perp \Rightarrow \text{invalid } \tau$

$\mid \perp_{\perp} \Rightarrow \text{true } \tau \text{ — invalid ??}$

$\mid \perp_{\perp} \Rightarrow \text{false } \tau) \text{ — must have actual type } Person \text{ otherwise}$

end

overloading $OclIsTypeOf_{Person} \equiv OclIsTypeOf_{Person} :: OclAny \Rightarrow Boolean$

begin

definition $OclIsTypeOf_{Person-OclAny}$:

$(X::OclAny) .oclIsTypeOf(Person) \equiv$

$(\lambda \tau. \text{case } X \text{ of}$

$\perp \Rightarrow \text{invalid } \tau$

$\mid \perp_{\perp} \Rightarrow \text{true } \tau$

$\mid \perp_{mk_{OclAny} \text{ oid } \perp_{\perp}} \Rightarrow \text{false } \tau$

$\mid \perp_{mk_{OclAny} \text{ oid } \perp_{\perp}} \Rightarrow \text{true } \tau)$

end

overloading $OclIsTypeOf_{Person} \equiv OclIsTypeOf_{Person} :: Person \Rightarrow Boolean$

begin

definition $OclIsTypeOf_{Person-Person}$:

$(X::Person) .oclIsTypeOf(Person) \equiv$

$(\lambda \tau. \text{case } X \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | - \Rightarrow \text{true } \tau)$
end

4.5.2. Execution with Invalid or Null as Argument

lemma $\text{OclIsTypeOf}_{\text{OclAny-OclAny-strict1}}[\text{simp}]$:
 $(\text{invalid}::\text{OclAny}).\text{oclIsTypeOf}(\text{OclAny}) = \text{invalid}$
lemma $\text{OclIsTypeOf}_{\text{OclAny-OclAny-strict2}}[\text{simp}]$:
 $(\text{null}::\text{OclAny}).\text{oclIsTypeOf}(\text{OclAny}) = \text{true}$
lemma $\text{OclIsTypeOf}_{\text{OclAny-Person-strict1}}[\text{simp}]$:
 $(\text{invalid}::\text{Person}).\text{oclIsTypeOf}(\text{OclAny}) = \text{invalid}$
lemma $\text{OclIsTypeOf}_{\text{OclAny-Person-strict2}}[\text{simp}]$:
 $(\text{null}::\text{Person}).\text{oclIsTypeOf}(\text{OclAny}) = \text{true}$
lemma $\text{OclIsTypeOf}_{\text{Person-OclAny-strict1}}[\text{simp}]$:
 $(\text{invalid}::\text{OclAny}).\text{oclIsTypeOf}(\text{Person}) = \text{invalid}$
lemma $\text{OclIsTypeOf}_{\text{Person-OclAny-strict2}}[\text{simp}]$:
 $(\text{null}::\text{OclAny}).\text{oclIsTypeOf}(\text{Person}) = \text{true}$
lemma $\text{OclIsTypeOf}_{\text{Person-Person-strict1}}[\text{simp}]$:
 $(\text{invalid}::\text{Person}).\text{oclIsTypeOf}(\text{Person}) = \text{invalid}$
lemma $\text{OclIsTypeOf}_{\text{Person-Person-strict2}}[\text{simp}]$:
 $(\text{null}::\text{Person}).\text{oclIsTypeOf}(\text{Person}) = \text{true}$

4.5.3. Up Down Casting

lemma $\text{actualType-larger-staticType}$:
assumes $\text{isdef}: \tau \models (\delta \ X)$
shows $\tau \models (X::\text{Person}).\text{oclIsTypeOf}(\text{OclAny}) \triangleq \text{false}$

lemma down-cast-type :
assumes $\text{isOclAny}: \tau \models (X::\text{OclAny}).\text{oclIsTypeOf}(\text{OclAny})$
and $\text{non-null}: \tau \models (\delta \ X)$
shows $\tau \models (X.\text{oclAsType}(\text{Person})) \triangleq \text{invalid}$

lemma $\text{down-cast-type}'$:
assumes $\text{isOclAny}: \tau \models (X::\text{OclAny}).\text{oclIsTypeOf}(\text{OclAny})$
and $\text{non-null}: \tau \models (\delta \ X)$
shows $\tau \models \text{not } (\vee (X.\text{oclAsType}(\text{Person})))$

lemma up-down-cast :
assumes $\text{isdef}: \tau \models (\delta \ X)$
shows $\tau \models ((X::\text{Person}).\text{oclAsType}(\text{OclAny}).\text{oclAsType}(\text{Person})) \triangleq X$

lemma $\text{up-down-cast-Person-OclAny-Person}[\text{simp}]$:
shows $((X::\text{Person}).\text{oclAsType}(\text{OclAny}).\text{oclAsType}(\text{Person})) = X$

lemma $\text{up-down-cast-Person-OclAny-Person}'$:
assumes $\tau \models \vee X$
shows $\tau \models (((X::\text{Person}).\text{oclAsType}(\text{OclAny}).\text{oclAsType}(\text{Person}))) \doteq X$

lemma *up-down-cast-Person-OclAny-Person''*;
assumes $\tau \models v \ (X :: \text{Person})$
shows $\tau \models (X.\text{oclIsTypeOf}(\text{Person}) \text{ implies } (X.\text{oclAsType}(\text{OclAny}).\text{oclAsType}(\text{Person})) \doteq X)$

4.6. OclIsKindOf

4.6.1. Definition

consts $\text{OclIsKindOf}_{\text{OclAny}} :: 'a \Rightarrow \text{Boolean} \ (\langle(-).\text{oclIsKindOf}'(\text{OclAny}')\rangle)$
consts $\text{OclIsKindOf}_{\text{Person}} :: 'a \Rightarrow \text{Boolean} \ (\langle(-).\text{oclIsKindOf}'(\text{Person}')\rangle)$

overloading $\text{OclIsKindOf}_{\text{OclAny}} \equiv \text{OclIsKindOf}_{\text{OclAny}} :: \text{OclAny} \Rightarrow \text{Boolean}$

begin

definition $\text{OclIsKindOf}_{\text{OclAny}}\text{-OclAny}$:
 $(X :: \text{OclAny}).\text{oclIsKindOf}(\text{OclAny}) \equiv$
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | - \Rightarrow \text{true } \tau)$

end

overloading $\text{OclIsKindOf}_{\text{OclAny}} \equiv \text{OclIsKindOf}_{\text{OclAny}} :: \text{Person} \Rightarrow \text{Boolean}$

begin

definition $\text{OclIsKindOf}_{\text{OclAny}}\text{-Person}$:
 $(X :: \text{Person}).\text{oclIsKindOf}(\text{OclAny}) \equiv$
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | - \Rightarrow \text{true } \tau)$

end

overloading $\text{OclIsKindOf}_{\text{Person}} \equiv \text{OclIsKindOf}_{\text{Person}} :: \text{OclAny} \Rightarrow \text{Boolean}$

begin

definition $\text{OclIsKindOf}_{\text{Person}}\text{-OclAny}$:
 $(X :: \text{OclAny}).\text{oclIsKindOf}(\text{Person}) \equiv$
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \perp_{\perp} \Rightarrow \text{true } \tau$
 $\quad | \perp_{\perp} \text{mk}_{\text{OclAny}} \text{oid } \perp_{\perp} \Rightarrow \text{false } \tau$
 $\quad | \perp_{\perp} \text{mk}_{\text{OclAny}} \text{oid } \perp_{\perp} \Rightarrow \text{true } \tau)$

end

overloading $\text{OclIsKindOf}_{\text{Person}} \equiv \text{OclIsKindOf}_{\text{Person}} :: \text{Person} \Rightarrow \text{Boolean}$

begin

definition $\text{OclIsKindOf}_{\text{Person}}\text{-Person}$:
 $(X :: \text{Person}).\text{oclIsKindOf}(\text{Person}) \equiv$
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | - \Rightarrow \text{true } \tau)$

end

4.6.2. 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 $OclIsKindOf_{Person-OclAny-strict1}[simp] : (invalid::OclAny) .oclIsKindOf(Person) = invalid$
lemma $OclIsKindOf_{Person-OclAny-strict2}[simp] : (null::OclAny) .oclIsKindOf(Person) = true$
lemma $OclIsKindOf_{Person-Person-strict1}[simp] : (invalid::Person) .oclIsKindOf(Person) = invalid$
lemma $OclIsKindOf_{Person-Person-strict2}[simp] : (null::Person) .oclIsKindOf(Person) = true$

4.6.3. Up Down Casting

lemma $actualKind-larger-staticKind$:
assumes $isdef: \tau \models (\delta X)$
shows $\tau \models ((X::Person) .oclIsKindOf(OclAny) \triangleq true)$

lemma $down-cast-kind$:
assumes $isOclAny: \neg (\tau \models ((X::OclAny) .oclIsKindOf(Person)))$
and $non-null: \tau \models (\delta X)$
shows $\tau \models ((X .oclAsType(Person)) \triangleq invalid)$

4.7. OclAllInstances

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

definition $Person \equiv OclAsType_{Person-\mathfrak{A}}$

definition $OclAny \equiv OclAsType_{OclAny-\mathfrak{A}}$

lemmas $[simp] = Person-def\ OclAny-def$

lemma $OclAllInstances-generic_{OclAny-exec}: OclAllInstances-generic\ pre-post\ OclAny =$
 $(\lambda \tau. Abs-Set_{base} \sqcup Some\ 'OclAny'\ ran\ (heap\ (pre-post\ \tau)) \sqcup)$

lemma $OclAllInstances-at-post_{OclAny-exec}: OclAny .allInstances() =$
 $(\lambda \tau. Abs-Set_{base} \sqcup Some\ 'OclAny'\ ran\ (heap\ (snd\ \tau)) \sqcup)$

lemma $OclAllInstances-at-pre_{OclAny-exec}: OclAny .allInstances@pre() =$
 $(\lambda \tau. Abs-Set_{base} \sqcup Some\ 'OclAny'\ ran\ (heap\ (fst\ \tau)) \sqcup)$

4.7.1. OclIsTypeOf

lemma $OclAny-allInstances-generic-oclIsTypeOf_{OclAny}I$:
assumes $[simp]: \bigwedge x. pre-post\ (x, x) = x$
shows $\exists \tau. (\tau \models ((OclAllInstances-generic\ pre-post\ OclAny) \rightarrow forAll_{Set}(X \setminus X .oclIsTypeOf(OclAny))))$

lemma $OclAny-allInstances-at-post-oclIsTypeOf_{OclAny}I$:
 $\exists \tau. (\tau \models (OclAny .allInstances() \rightarrow forAll_{Set}(X \setminus X .oclIsTypeOf(OclAny))))$

lemma *OclAny-allInstances-at-pre-oclIsTypeOf*_{OclAny1}:
 $\exists \tau. (\tau \models (OclAny.allInstances@pre() \rightarrow forAll_{Set}(XIX.oclIsTypeOf(OclAny))))$

lemma *OclAny-allInstances-generic-oclIsTypeOf*_{OclAny2}:
assumes [simp]: $\bigwedge x. pre\text{-}post(x, x) = x$
shows $\exists \tau. (\tau \models not((OclAllInstances-generic\ pre\text{-}post\ OclAny) \rightarrow forAll_{Set}(XIX.oclIsTypeOf(OclAny))))$

lemma *OclAny-allInstances-at-post-oclIsTypeOf*_{OclAny2}:
 $\exists \tau. (\tau \models not(OclAny.allInstances() \rightarrow forAll_{Set}(XIX.oclIsTypeOf(OclAny))))$

lemma *OclAny-allInstances-at-pre-oclIsTypeOf*_{OclAny2}:
 $\exists \tau. (\tau \models not(OclAny.allInstances@pre() \rightarrow forAll_{Set}(XIX.oclIsTypeOf(OclAny))))$

lemma *Person-allInstances-generic-oclIsTypeOf*_{Person}:
 $\tau \models ((OclAllInstances-generic\ pre\text{-}post\ Person) \rightarrow forAll_{Set}(XIX.oclIsTypeOf(Person)))$

lemma *Person-allInstances-at-post-oclIsTypeOf*_{Person}:
 $\tau \models (Person.allInstances() \rightarrow forAll_{Set}(XIX.oclIsTypeOf(Person)))$

lemma *Person-allInstances-at-pre-oclIsTypeOf*_{Person}:
 $\tau \models (Person.allInstances@pre() \rightarrow forAll_{Set}(XIX.oclIsTypeOf(Person)))$

4.7.2. OclIsKindOf

lemma *OclAny-allInstances-generic-oclIsKindOf*_{OclAny}:
 $\tau \models ((OclAllInstances-generic\ pre\text{-}post\ OclAny) \rightarrow forAll_{Set}(XIX.oclIsKindOf(OclAny)))$

lemma *OclAny-allInstances-at-post-oclIsKindOf*_{OclAny}:
 $\tau \models (OclAny.allInstances() \rightarrow forAll_{Set}(XIX.oclIsKindOf(OclAny)))$

lemma *OclAny-allInstances-at-pre-oclIsKindOf*_{OclAny}:
 $\tau \models (OclAny.allInstances@pre() \rightarrow forAll_{Set}(XIX.oclIsKindOf(OclAny)))$

lemma *Person-allInstances-generic-oclIsKindOf*_{OclAny}:
 $\tau \models ((OclAllInstances-generic\ pre\text{-}post\ Person) \rightarrow forAll_{Set}(XIX.oclIsKindOf(OclAny)))$

lemma *Person-allInstances-at-post-oclIsKindOf*_{OclAny}:
 $\tau \models (Person.allInstances() \rightarrow forAll_{Set}(XIX.oclIsKindOf(OclAny)))$

lemma *Person-allInstances-at-pre-oclIsKindOf*_{OclAny}:
 $\tau \models (Person.allInstances@pre() \rightarrow forAll_{Set}(XIX.oclIsKindOf(OclAny)))$

lemma *Person-allInstances-generic-oclIsKindOf*_{Person}:
 $\tau \models ((OclAllInstances-generic\ pre\text{-}post\ Person) \rightarrow forAll_{Set}(XIX.oclIsKindOf(Person)))$

lemma *Person-allInstances-at-post-oclIsKindOf*_{Person}:
 $\tau \models (Person.allInstances() \rightarrow forAll_{Set}(XIX.oclIsKindOf(Person)))$

lemma *Person-allInstances-at-pre-oclIsKindOf*_{Person}:
 $\tau \models (Person.allInstances@pre() \rightarrow forAll_{Set}(XIX.oclIsKindOf(Person)))$

4.8. The Accessors (any, boss, salary)

Should be generated entirely from a class-diagram.

4.8.1. Definition (of the association Employee-Boss)

We start with a `oid` for the association; this `oid` can be used in presence of association classes to represent the association inside an object, pretty much similar to the `Design_UML`, where we stored an `oid` inside the class as “pointer.”

definition $oid_{\text{PersonBOS}}$:: oid **where** $oid_{\text{PersonBOS}} = 10$

From there on, we can already define an empty state which must contain for $oid_{\text{PersonBOS}}$ the empty relation (encoded as association list, since there are associations with a Sequence-like structure).

definition $eval_extract$:: $(\mathcal{A}, ('a::object) option option) val$
 $\Rightarrow (oid \Rightarrow (\mathcal{A}, 'c::null) val)$
 $\Rightarrow (\mathcal{A}, 'c::null) val$

where $eval_extract\ X\ f = (\lambda\ \tau. case\ X\ \tau\ of$
 $\quad \perp \Rightarrow invalid\ \tau \quad \text{— exception propagation}$
 $\quad | \perp \perp \Rightarrow invalid\ \tau \quad \text{— dereferencing null pointer}$
 $\quad | \perp obj \perp \Rightarrow f\ (oid_of\ obj)\ \tau)$

definition $choose_2-1 = fst$

definition $choose_2-2 = snd$

definition $List_flatten = (\lambda\ l. foldl\ ((\lambda\ acc. (\lambda\ l. foldl\ ((\lambda\ acc. (\lambda\ l. (Cons\ l)\ (acc))))\ (acc)\ ((rev\ l))))\ (Nil)\ ((rev\ l))))$

definition $deref_assoc_s_2$:: $(\mathcal{A}\ state \times \mathcal{A}\ state \Rightarrow \mathcal{A}\ state)$

$\Rightarrow (oid\ list\ list \Rightarrow oid\ list \times oid\ list)$
 $\Rightarrow oid$
 $\Rightarrow (oid\ list \Rightarrow (\mathcal{A}, 'f)val)$
 $\Rightarrow oid$
 $\Rightarrow (\mathcal{A}, 'f::null)val$

where $deref_assoc_s_2\ pre_post\ to_from\ assoc_oid\ f\ oid =$
 $(\lambda\ \tau. case\ (assoc\ (pre_post\ \tau))\ assoc_oid\ of$
 $\quad \perp \perp \Rightarrow f\ (List_flatten\ (map\ (choose_2-2 \circ to_from)$
 $\quad (filter\ (\lambda\ p. List.member\ (choose_2-1\ (to_from\ p))\ oid)\ S)))$
 $\quad \tau$
 $\quad | - \Rightarrow invalid\ \tau)$

The *pre-post*-parameter is configured with *fst* or *snd*, the *to-from*-parameter either with the identity *id* or the following combinator *switch*:

definition $switch_2-1 = (\lambda\ [x,y] \Rightarrow (x,y))$

definition $switch_2-2 = (\lambda\ [x,y] \Rightarrow (y,x))$

definition $switch_3-1 = (\lambda\ [x,y,z] \Rightarrow (x,y))$

definition $switch_3-2 = (\lambda\ [x,y,z] \Rightarrow (x,z))$

definition $switch_3-3 = (\lambda\ [x,y,z] \Rightarrow (y,x))$

definition $switch_3-4 = (\lambda\ [x,y,z] \Rightarrow (y,z))$

definition $switch_3-5 = (\lambda\ [x,y,z] \Rightarrow (z,x))$

definition $switch_3-6 = (\lambda\ [x,y,z] \Rightarrow (z,y))$

definition $deref-oid_{\text{Person}} :: (\mathcal{A} \text{ state} \times \mathcal{A} \text{ state} \Rightarrow \mathcal{A} \text{ state})$

$\Rightarrow (type_{\text{Person}} \Rightarrow (\mathcal{A}, 'c::null)val)$

$\Rightarrow oid$

$\Rightarrow (\mathcal{A}, 'c::null)val$

where $deref-oid_{\text{Person}} \text{fst-snd } f \text{oid} = (\lambda \tau. \text{case } (heap \text{fst-snd } \tau)) \text{oid of}$

$\quad \perp \text{in}_{\text{Person}} \text{obj} _ \Rightarrow f \text{obj } \tau$

$\quad | _ \Rightarrow \text{invalid } \tau)$

definition $deref-oid_{\text{OclAny}} :: (\mathcal{A} \text{ state} \times \mathcal{A} \text{ state} \Rightarrow \mathcal{A} \text{ state})$

$\Rightarrow (type_{\text{OclAny}} \Rightarrow (\mathcal{A}, 'c::null)val)$

$\Rightarrow oid$

$\Rightarrow (\mathcal{A}, 'c::null)val$

where $deref-oid_{\text{OclAny}} \text{fst-snd } f \text{oid} = (\lambda \tau. \text{case } (heap \text{fst-snd } \tau)) \text{oid of}$

$\quad \perp \text{in}_{\text{OclAny}} \text{obj} _ \Rightarrow f \text{obj } \tau$

$\quad | _ \Rightarrow \text{invalid } \tau)$

pointer undefined in state or not referencing a type conform object representation

definition $select_{\text{OclAny}} \mathcal{A} \mathcal{N} \mathcal{Y} f = (\lambda X. \text{case } X \text{ of}$

$\quad (mk_{\text{OclAny}} _ \perp) \Rightarrow null$

$\quad | (mk_{\text{OclAny}} _ \text{any} _) \Rightarrow f (\lambda x _ . _ \perp x _) \text{any})$

definition $select_{\text{Person}} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} f = \text{select-object miSet UML-Set.OclIncluding UML-Set.OclANY } (f (\lambda x _ . _ \perp x _))$

definition $select_{\text{Person}} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y} f = (\lambda X. \text{case } X \text{ of}$

$\quad (mk_{\text{Person}} _ \perp) \Rightarrow null$

$\quad | (mk_{\text{Person}} _ \text{salary} _) \Rightarrow f (\lambda x _ . _ \perp x _) \text{salary})$

definition $deref-assocs_2 \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} \text{fst-snd } f = (\lambda mk_{\text{Person}} \text{oid} _ \Rightarrow$

$\quad deref-assocs_2 \text{fst-snd switch}_2 \text{-1 oid}_{\text{Person}} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} f \text{oid})$

definition $\text{in-pre-state} = \text{fst}$

definition $\text{in-post-state} = \text{snd}$

definition $\text{reconst-basetype} = (\lambda \text{convert } x. \text{convert } x)$

definition $\text{dot}_{\text{OclAny}} \mathcal{A} \mathcal{N} \mathcal{Y} :: \text{OclAny} \Rightarrow _ \langle (I(-).\text{any}) \rangle 50)$

where $(X).\text{any} = \text{eval-extract } X$

$\quad (deref-oid_{\text{OclAny}} \text{in-post-state}$

$\quad (select_{\text{OclAny}} \mathcal{A} \mathcal{N} \mathcal{Y}$

$\quad \text{reconst-basetype}))$

definition $\text{dot}_{\text{Person}} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} :: \text{Person} \Rightarrow \text{Person} \langle (I(-).\text{boss}) \rangle 50)$

where $(X).\text{boss} = \text{eval-extract } X$

$\quad (deref-oid_{\text{Person}} \text{in-post-state}$

$\quad (deref-assocs_2 \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} \text{in-post-state}$

$\quad (select_{\text{Person}} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S}$

$(deref-oid_{Person} \text{ in-post-state})))))$

definition $dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y} :: Person \Rightarrow Integer \ (\langle (I(-).salary) \rangle 50)$
where $(X).salary = eval-extract \ X$
 $(deref-oid_{Person} \text{ in-post-state}$
 $(select_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}$
 $reconst-basetype))$

definition $dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y} \text{-at-pre} :: OclAny \Rightarrow - \ (\langle (I(-).any@pre) \rangle 50)$
where $(X).any@pre = eval-extract \ X$
 $(deref-oid_{OclAny} \text{ in-pre-state}$
 $(select_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y}$
 $reconst-basetype))$

definition $dot_{Person} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} \text{-at-pre} :: Person \Rightarrow Person \ (\langle (I(-).boss@pre) \rangle 50)$
where $(X).boss@pre = eval-extract \ X$
 $(deref-oid_{Person} \text{ in-pre-state}$
 $(deref-assocs_2 \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} \text{ in-pre-state}$
 $(select_{Person} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S}$
 $(deref-oid_{Person} \text{ in-pre-state}))))$

definition $dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y} \text{-at-pre} :: Person \Rightarrow Integer \ (\langle (I(-).salary@pre) \rangle 50)$
where $(X).salary@pre = eval-extract \ X$
 $(deref-oid_{Person} \text{ in-pre-state}$
 $(select_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y}$
 $reconst-basetype))$

lemmas $dot-accessor =$
 $dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y} \text{-def}$
 $dot_{Person} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} \text{-def}$
 $dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y} \text{-def}$
 $dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y} \text{-at-pre-def}$
 $dot_{Person} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} \text{-at-pre-def}$
 $dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y} \text{-at-pre-def}$

4.8.2. Context Passing

lemmas $[simp] = eval-extract-def$

lemma $cp-dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y} : ((X).any) \ \tau = ((\lambda -. X \ \tau).any) \ \tau$

lemma $cp-dot_{Person} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} : ((X).boss) \ \tau = ((\lambda -. X \ \tau).boss) \ \tau$

lemma $cp-dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y} : ((X).salary) \ \tau = ((\lambda -. X \ \tau).salary) \ \tau$

lemma $cp-dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y} \text{-at-pre} : ((X).any@pre) \ \tau = ((\lambda -. X \ \tau).any@pre) \ \tau$

lemma $cp-dot_{Person} \mathcal{B} \mathcal{O} \mathcal{S} \mathcal{S} \text{-at-pre} : ((X).boss@pre) \ \tau = ((\lambda -. X \ \tau).boss@pre) \ \tau$

lemma $cp-dot_{Person} \mathcal{S} \mathcal{A} \mathcal{L} \mathcal{A} \mathcal{R} \mathcal{Y} \text{-at-pre} : ((X).salary@pre) \ \tau = ((\lambda -. X \ \tau).salary@pre) \ \tau$

lemmas $cp-dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y} \text{-I} [simp, intro!]=$
 $cp-dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y} [THEN \ allI [THEN \ allI],$
 $of \ \lambda \ X -. X \ \lambda -. \tau, THEN \ cpII]$

lemmas $cp-dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y} \text{-at-pre-I} [simp, intro!]=$
 $cp-dot_{OclAny} \mathcal{A} \mathcal{N} \mathcal{Y} \text{-at-pre} [THEN \ allI [THEN \ allI],$

of $\lambda X \cdot X \lambda \cdot \tau$, THEN $cpII$]

lemmas $cp\text{-}dot_{\text{Person}}\mathcal{BOSS}\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{\text{Person}}\mathcal{BOSS}$ [THEN allI[THEN allI],
 of $\lambda X \cdot X \lambda \cdot \tau$, THEN $cpII$]
lemmas $cp\text{-}dot_{\text{Person}}\mathcal{BOSS}\text{-}at\text{-}pre\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{\text{Person}}\mathcal{BOSS}\text{-}at\text{-}pre$ [THEN allI[THEN allI],
 of $\lambda X \cdot X \lambda \cdot \tau$, THEN $cpII$]

lemmas $cp\text{-}dot_{\text{Person}}\mathcal{SALARY}\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{\text{Person}}\mathcal{SALARY}$ [THEN allI[THEN allI],
 of $\lambda X \cdot X \lambda \cdot \tau$, THEN $cpII$]
lemmas $cp\text{-}dot_{\text{Person}}\mathcal{SALARY}\text{-}at\text{-}pre\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{\text{Person}}\mathcal{SALARY}\text{-}at\text{-}pre$ [THEN allI[THEN allI],
 of $\lambda X \cdot X \lambda \cdot \tau$, THEN $cpII$]

4.8.3. Execution with Invalid or Null as Argument

lemma $dot_{\text{OclAny}}\mathcal{ANY}\text{-}nullstrict$ [simp]: $(null).any = invalid$
lemma $dot_{\text{OclAny}}\mathcal{ANY}\text{-}at\text{-}pre\text{-}nullstrict$ [simp]: $(null).any@pre = invalid$
lemma $dot_{\text{OclAny}}\mathcal{ANY}\text{-}strict$ [simp]: $(invalid).any = invalid$
lemma $dot_{\text{OclAny}}\mathcal{ANY}\text{-}at\text{-}pre\text{-}strict$ [simp]: $(invalid).any@pre = invalid$

lemma $dot_{\text{Person}}\mathcal{BOSS}\text{-}nullstrict$ [simp]: $(null).boss = invalid$
lemma $dot_{\text{Person}}\mathcal{BOSS}\text{-}at\text{-}pre\text{-}nullstrict$ [simp]: $(null).boss@pre = invalid$
lemma $dot_{\text{Person}}\mathcal{BOSS}\text{-}strict$ [simp]: $(invalid).boss = invalid$
lemma $dot_{\text{Person}}\mathcal{BOSS}\text{-}at\text{-}pre\text{-}strict$ [simp]: $(invalid).boss@pre = invalid$

lemma $dot_{\text{Person}}\mathcal{SALARY}\text{-}nullstrict$ [simp]: $(null).salary = invalid$
lemma $dot_{\text{Person}}\mathcal{SALARY}\text{-}at\text{-}pre\text{-}nullstrict$ [simp]: $(null).salary@pre = invalid$
lemma $dot_{\text{Person}}\mathcal{SALARY}\text{-}strict$ [simp]: $(invalid).salary = invalid$
lemma $dot_{\text{Person}}\mathcal{SALARY}\text{-}at\text{-}pre\text{-}strict$ [simp]: $(invalid).salary@pre = invalid$

4.8.4. Representation in States

lemma $dot_{\text{Person}}\mathcal{BOSS}\text{-}def\text{-}mono$: $\tau \models \delta(X.boss) \implies \tau \models \delta(X)$

lemma $repr\text{-}boss$:
assumes $A : \tau \models \delta(x.boss)$
shows $is\text{-}represented\text{-}in\text{-}state\text{-}in\text{-}post\text{-}state\ (x.boss)\ \text{Person}\ \tau$

lemma $repr\text{-}bossX$:
assumes $A : \tau \models \delta(x.boss)$
shows $\tau \models ((\text{Person}.allInstances()) \rightarrow includes_{\text{Set}}(x.boss))$

4.9. A Little Infra-structure on Example States

The example we are defining in this section comes from the figure 4.2.

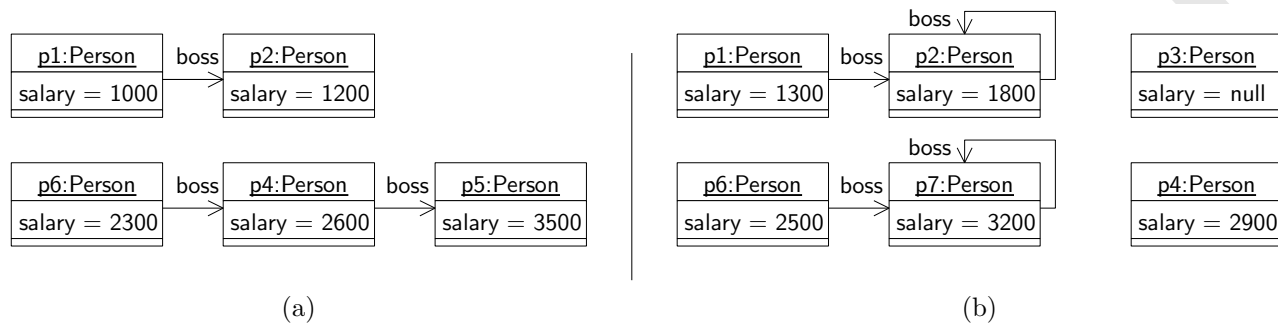


Figure 4.2. – (a) pre-state σ_1 and (b) post-state σ'_1 .

definition

$$\begin{aligned} \sigma_1 \equiv \llbracket \text{heap} = \text{Map.empty}(\text{oid0} \mapsto \text{inPerson}(\text{mkPerson oid0 } \llbracket 1000 \rrbracket), \\ \text{oid1} \mapsto \text{inPerson}(\text{mkPerson oid1 } \llbracket 1200 \rrbracket), \\ \text{oid2} \mapsto \text{inPerson}(\text{mkPerson oid2 } \llbracket 2600 \rrbracket), \\ \text{oid3} \mapsto \text{inPerson}(\text{mkPerson oid3 } \llbracket 2600 \rrbracket), \\ \text{oid4} \mapsto \text{inPerson}(\text{person5}), \\ \text{oid5} \mapsto \text{inPerson}(\text{mkPerson oid5 } \llbracket 2300 \rrbracket), \\ \text{oid6} \mapsto \text{inPerson}(\text{person9}), \\ \text{oid7} \mapsto \text{inPerson}(\text{person9}), \\ \text{oids8} \mapsto \text{inPerson}(\text{person9}), \\ \text{assocs} = \text{Map.empty}(\text{oidPerson} \cdot \mathcal{BOS} \mapsto [[[\text{oid0}], [\text{oid1}]], [[\text{oid3}], [\text{oid4}]], [[\text{oid5}], [\text{oid3}]]]) \rrbracket \end{aligned}$$

definition

$$\begin{aligned} \sigma'_1 \equiv \llbracket \text{heap} = \text{Map.empty}(\text{oid0} \mapsto \text{inPerson}(\text{person1}), \\ \text{oid1} \mapsto \text{inPerson}(\text{person2}), \\ \text{oid2} \mapsto \text{inPerson}(\text{person3}), \\ \text{oid3} \mapsto \text{inPerson}(\text{person4}), \\ \text{oid4} \mapsto \text{inPerson}(\text{person6}), \\ \text{oid5} \mapsto \text{inPerson}(\text{person6}), \\ \text{oid6} \mapsto \text{inOclAny}(\text{person7}), \\ \text{oid7} \mapsto \text{inOclAny}(\text{person8}), \\ \text{oids8} \mapsto \text{inPerson}(\text{person9}), \\ \text{assocs} = \text{Map.empty}(\text{oidPerson} \cdot \mathcal{BOS} \mapsto [[[\text{oid0}], [\text{oid1}]], [[\text{oid1}], [\text{oid1}]], [[\text{oid5}], [\text{oid6}]], [[\text{oid6}], [\text{oid6}]]]) \rrbracket \end{aligned}$$

definition $\sigma_0 \equiv \llbracket \text{heap} = \text{Map.empty}, \text{assocs} = \text{Map.empty} \rrbracket$

lemma *basic- τ -wff*: $\text{WFF}(\sigma_1, \sigma'_1)$

lemma [*simp, code-unfold*]: $\text{dom}(\text{heap } \sigma_1) = \{\text{oid0}, \text{oid1}, \text{oid2}, \text{oid3}, \text{oid4}, \text{oid5}, \text{oid6}, \text{oid7}, \text{oid8}\}$

lemma [*simp, code-unfold*]: $\text{dom}(\text{heap } \sigma'_1) = \{\text{oid0}, \text{oid1}, \text{oid2}, \text{oid3}, \text{oid4}, \text{oid5}, \text{oid6}, \text{oid7}, \text{oid8}\}$

Assert $\bigwedge s_{\text{pre}}. (s_{\text{pre}}, \sigma_1) \models (X_{\text{Person1}}.\text{salary} < 1000)$

Assert $\bigwedge s_{\text{pre}}. (s_{\text{pre}}, \sigma_1) \models (X_{\text{Person1}}.\text{salary} \doteq 1300)$

Assert $\bigwedge s_{\text{post}}. (\sigma_1, s_{\text{post}}) \models (X_{\text{Person1}}.\text{salary}@pre \doteq 1000)$

Assert $\bigwedge s_{\text{post}}. (\sigma_1, s_{\text{post}}) \models (X_{\text{Person1}}.\text{salary}@pre < 1300)$

lemma $(\sigma_1, \sigma_1') \models (X_{\text{Person1}}.oclIsMaintained())$

lemma $\wedge s_{\text{pre}} s_{\text{post}}. (s_{\text{pre}}, s_{\text{post}}) \models ((X_{\text{Person1}}.oclAsType(OclAny).oclAsType(Person)) \doteq X_{\text{Person1}})$

Assert $\wedge s_{\text{pre}} s_{\text{post}}. (s_{\text{pre}}, s_{\text{post}}) \models (X_{\text{Person1}}.oclIsTypeOf(Person))$

Assert $\wedge s_{\text{pre}} s_{\text{post}}. (s_{\text{pre}}, s_{\text{post}}) \models \text{not}(X_{\text{Person1}}.oclIsTypeOf(OclAny))$

Assert $\wedge s_{\text{pre}} s_{\text{post}}. (s_{\text{pre}}, s_{\text{post}}) \models (X_{\text{Person1}}.oclIsKindOf(Person))$

Assert $\wedge s_{\text{pre}} s_{\text{post}}. (s_{\text{pre}}, s_{\text{post}}) \models (X_{\text{Person1}}.oclIsKindOf(OclAny))$

Assert $\wedge s_{\text{pre}} s_{\text{post}}. (s_{\text{pre}}, s_{\text{post}}) \models \text{not}(X_{\text{Person1}}.oclAsType(OclAny).oclIsTypeOf(OclAny))$

Assert $\wedge s_{\text{pre}}. (s_{\text{pre}}, \sigma_1') \models (X_{\text{Person2}}.salary \doteq 1800)$

Assert $\wedge s_{\text{post}}. (\sigma_1, s_{\text{post}}) \models (X_{\text{Person2}}.salary@pre \doteq 1200)$

lemma $(\sigma_1, \sigma_1') \models (X_{\text{Person2}}.oclIsMaintained())$

Assert $\wedge s_{\text{pre}}. (s_{\text{pre}}, \sigma_1') \models (X_{\text{Person3}}.salary \doteq \text{null})$

Assert $\wedge s_{\text{post}}. (\sigma_1, s_{\text{post}}) \models \text{not}(v(X_{\text{Person3}}.salary@pre))$

lemma $(\sigma_1, \sigma_1') \models (X_{\text{Person3}}.oclIsNew())$

lemma $(\sigma_1, \sigma_1') \models (X_{\text{Person4}}.oclIsMaintained())$

Assert $\wedge s_{\text{pre}}. (s_{\text{pre}}, \sigma_1') \models \text{not}(v(X_{\text{Person5}}.salary))$

Assert $\wedge s_{\text{post}}. (\sigma_1, s_{\text{post}}) \models (X_{\text{Person5}}.salary@pre \doteq 3500)$

lemma $(\sigma_1, \sigma_1') \models (X_{\text{Person5}}.oclIsDeleted())$

lemma $(\sigma_1, \sigma_1') \models (X_{\text{Person6}}.oclIsMaintained())$

Assert $\wedge s_{\text{pre}} s_{\text{post}}. (s_{\text{pre}}, s_{\text{post}}) \models v(X_{\text{Person7}}.oclAsType(Person))$

lemma $\wedge s_{\text{pre}} s_{\text{post}}. (s_{\text{pre}}, s_{\text{post}}) \models ((X_{\text{Person7}}.oclAsType(Person).oclAsType(OclAny).oclAsType(Person))$

$\doteq (X_{\text{Person7}}.oclAsType(Person)))$

lemma $(\sigma_1, \sigma_1') \models (X_{\text{Person7}}.oclIsNew())$

Assert $\wedge s_{\text{pre}} s_{\text{post}}. (s_{\text{pre}}, s_{\text{post}}) \models (X_{\text{Person8}} <> X_{\text{Person7}})$

Assert $\wedge s_{\text{pre}} s_{\text{post}}. (s_{\text{pre}}, s_{\text{post}}) \models \text{not}(v(X_{\text{Person8}}.oclAsType(Person)))$

Assert $\wedge s_{\text{pre}} s_{\text{post}}. (s_{\text{pre}}, s_{\text{post}}) \models (X_{\text{Person8}}.oclIsTypeOf(OclAny))$

Assert $\wedge s_{\text{pre}} s_{\text{post}}. (s_{\text{pre}}, s_{\text{post}}) \models \text{not}(X_{\text{Person8}}.oclIsTypeOf(Person))$

Assert $\wedge s_{\text{pre}} s_{\text{post}}. (s_{\text{pre}}, s_{\text{post}}) \models \text{not}(X_{\text{Person8}}.oclIsKindOf(Person))$

Assert $\wedge s_{\text{pre}} s_{\text{post}}. (s_{\text{pre}}, s_{\text{post}}) \models (X_{\text{Person8}}.oclIsKindOf(OclAny))$

lemma σ -modifiedonly: $(\sigma_1, \sigma_1') \models (\text{Set}\{ X_{\text{Person1}}.oclAsType(OclAny)$

$, X_{\text{Person2}}.oclAsType(OclAny)$

$, X_{\text{Person3}}.oclAsType(OclAny))$

```

, XPerson4.oclAsType(OclAny)
/XPerson5.oclAsType(OclAny)
, XPerson6.oclAsType(OclAny)
/XPerson7.oclAsType(OclAny)
/XPerson8.oclAsType(OclAny)
/XPerson9.oclAsType(OclAny) }->oclIsModifiedOnly())

```

lemma $(\sigma_1, \sigma_1') \models ((X_{\text{Person9}} @pre (\lambda x. _OclAsType_{\text{Person}} \cdot \mathfrak{A} x)) \triangleq X_{\text{Person9}})$

lemma $(\sigma_1, \sigma_1') \models ((X_{\text{Person9}} @post (\lambda x. _OclAsType_{\text{Person}} \cdot \mathfrak{A} x)) \triangleq X_{\text{Person9}})$

lemma $(\sigma_1, \sigma_1') \models (((X_{\text{Person9}}.oclAsType(OclAny)) @pre (\lambda x. _OclAsType_{OclAny} \cdot \mathfrak{A} x)) \triangleq ((X_{\text{Person9}}.oclAsType(OclAny)) @post (\lambda x. _OclAsType_{OclAny} \cdot \mathfrak{A} x)))$

lemma $perm\text{-}\sigma_1' : \sigma_1' = \langle \text{heap} = \text{Map.empty}$
 $\text{oid8} \mapsto \text{in}_{\text{Person}} \text{person9},$
 $\text{oid7} \mapsto \text{in}_{OclAny} \text{person8},$
 $\text{oid6} \mapsto \text{in}_{OclAny} \text{person7},$
 $\text{oid5} \mapsto \text{in}_{\text{Person}} \text{person6},$
 ~~oid4~~
 $\text{oid3} \mapsto \text{in}_{\text{Person}} \text{person4},$
 $\text{oid2} \mapsto \text{in}_{\text{Person}} \text{person3},$
 $\text{oid1} \mapsto \text{in}_{\text{Person}} \text{person2},$
 $\text{oid0} \mapsto \text{in}_{\text{Person}} \text{person1})$
 $, \text{assocs} = \text{assocs } \sigma_1' \rangle$

declare *const-ss* [simp]

lemma $\wedge \sigma_1.$
 $(\sigma_1, \sigma_1') \models (\text{Person} .allInstances() \doteq \text{Set}\{ X_{\text{Person1}}, X_{\text{Person2}}, X_{\text{Person3}}, X_{\text{Person4}}, \text{Person5}, X_{\text{Person6}},$
 $X_{\text{Person7}}.oclAsType(\text{Person}), \text{Person8}, X_{\text{Person9}} \})$

lemma $\wedge \sigma_1.$
 $(\sigma_1, \sigma_1') \models (OclAny .allInstances() \doteq \text{Set}\{ X_{\text{Person1}}.oclAsType(OclAny), X_{\text{Person2}}.oclAsType(OclAny),$
 $X_{\text{Person3}}.oclAsType(OclAny), X_{\text{Person4}}.oclAsType(OclAny)$
 ~~$/X_{\text{Person5}}, X_{\text{Person6}}.oclAsType(OclAny),$~~
 $X_{\text{Person7}}, X_{\text{Person8}}, X_{\text{Person9}}.oclAsType(OclAny) \})$

4.10. OCL Part: Invariant

These recursive predicates can be defined conservatively by greatest fix-point constructions—automatically. See [3, 4] for details. For the purpose of this example, we state them as axioms here.

```

context Person
inv label : self .boss <> null implies (self .salary \<le> ((self .boss) .salary))

```

definition $\text{Person-label}_{\text{inv}} :: \text{Person} \Rightarrow \text{Boolean}$

where $\text{Person-label}_{\text{inv}}(\text{self}) \equiv$
 $(\text{self} .\text{boss} \triangleq \text{null} \text{ implies } (\text{self} .\text{salary} \leq_{\text{int}} ((\text{self} .\text{boss}) .\text{salary})))$

definition $Person\text{-}label_{invATpre} :: Person \Rightarrow Boolean$

where $Person\text{-}label_{invATpre}(self) \equiv$
 $(self.boss@pre \neq null \implies (self.salary@pre \leq_{int} ((self.boss@pre).salary@pre)))$

definition $Person\text{-}label_{globalinv} :: Boolean$

where $Person\text{-}label_{globalinv} \equiv (Person.allInstances() \rightarrow \text{forAll}_{Set}(x \mid Person\text{-}label_{inv}(x)) \text{ and } (Person.allInstances@pre() \rightarrow \text{forAll}_{Set}(x \mid Person\text{-}label_{invATpre}(x))))$

lemma $\tau \models \delta(X.boss) \implies \tau \models Person.allInstances() \rightarrow \text{includes}_{Set}(X.boss) \wedge$
 $\tau \models Person.allInstances() \rightarrow \text{includes}_{Set}(X)$

lemma $REC\text{-}pre : \tau \models Person\text{-}label_{globalinv}$

$\implies \tau \models Person.allInstances() \rightarrow \text{includes}_{Set}(X) \text{ — } X \text{ represented object in state}$
 $\implies \exists REC. \tau \models REC(X) \triangleq (Person\text{-}label_{inv}(X) \text{ and } (X.boss \neq null \implies REC(X.boss)))$

This allows to state a predicate:

axiomatization $inv_{Person\text{-}label} :: Person \Rightarrow Boolean$

where $inv_{Person\text{-}label}\text{-}def:$

$(\tau \models Person.allInstances() \rightarrow \text{includes}_{Set}(self)) \implies$
 $(\tau \models (inv_{Person\text{-}label}(self) \triangleq (self.boss \neq null \implies$
 $(self.salary \leq_{int} ((self.boss).salary)) \text{ and }$
 $inv_{Person\text{-}label}(self.boss))))$

axiomatization $inv_{Person\text{-}labelATpre} :: Person \Rightarrow Boolean$

where $inv_{Person\text{-}labelATpre}\text{-}def:$

$(\tau \models Person.allInstances@pre() \rightarrow \text{includes}_{Set}(self)) \implies$
 $(\tau \models (inv_{Person\text{-}labelATpre}(self) \triangleq (self.boss@pre \neq null \implies$
 $(self.salary@pre \leq_{int} ((self.boss@pre).salary@pre)) \text{ and }$
 $inv_{Person\text{-}labelATpre}(self.boss@pre))))$

lemma $inv\text{-}1 :$

$(\tau \models Person.allInstances() \rightarrow \text{includes}_{Set}(self)) \implies$
 $(\tau \models inv_{Person\text{-}label}(self) = ((\tau \models (self.boss \neq null)) \vee$
 $(\tau \models (self.boss \neq null) \wedge$
 $\tau \models ((self.salary) \leq_{int} (self.boss.salary)) \wedge$
 $\tau \models (inv_{Person\text{-}label}(self.boss))))$

lemma $inv\text{-}2 :$

$(\tau \models Person.allInstances@pre() \rightarrow \text{includes}_{Set}(self)) \implies$
 $(\tau \models inv_{Person\text{-}labelATpre}(self) = ((\tau \models (self.boss@pre \neq null)) \vee$
 $(\tau \models (self.boss@pre \neq null) \wedge$
 $(\tau \models (self.boss@pre.salary@pre \leq_{int} self.salary@pre)) \wedge$
 $(\tau \models (inv_{Person\text{-}labelATpre}(self.boss@pre))))$

A very first attempt to characterize the axiomatization by an inductive definition - this can not be the last word since too

weak (should be equality!)

```
coinductive inv :: Person ⇒ (ℚ)st ⇒ bool where
  (τ ⊨ (δ self)) ⇒ ((τ ⊨ (self.boss ≐ null)) ∨
    (τ ⊨ (self.boss <> null) ∧ (τ ⊨ (self.boss.salary ≤int self.salary)) ∧
    (inv(self.boss)τ)))
  ⇒ (inv self τ)
```

4.11. OCL Part: The Contract of a Recursive Query

The original specification of a recursive query :

```
context Person::contents():Set(Integer)
pre:    true
post:  result = if self.boss = null
               then Set{i}
               else self.boss.contents()->including(i)
               endif
```

For the case of recursive queries, we use at present just axiomatizations:

axiomatization contents :: Person ⇒ Set-Integer (⟨(1(-).contents'('))⟩ 50)

where contents-def:

```
(self.contents()) = (λ τ. SOME res. let res = λ -. res in
  if τ ⊨ (δ self)
  then ((τ ⊨ true) ∧
    (τ ⊨ res ≐ if (self.boss ≐ null)
      then (Set{self.salary})
      else (self.boss.contents()
        ->includingSet(self.salary))
    endif))
  else τ ⊨ res ≐ invalid)
and cp0-contents:(X.contents()) τ = ((λ -. X τ).contents()) τ
```

interpretation contents : contract0 contents λ self. true

```
λ self res. res ≐ if (self.boss ≐ null)
  then (Set{self.salary})
  else (self.boss.contents()
    ->includingSet(self.salary))
endif
```

Specializing $\llbracket cp\ E; \tau \models \delta\ self; \tau \models true; \tau \models POST'\ self; \bigwedge res. (res \triangleq \text{if } self.boss \doteq null \text{ then } Set\{self.salary\} \text{ else } self.boss.contents() \rightarrow including_{Set}(self.salary) \text{ endif}) = (POST'\ self \text{ and } (res \triangleq BODY\ self)) \rrbracket \implies (\tau \models E\ (self.contents())) = (\tau \models E\ (BODY\ self))$, one gets the following more practical rewrite rule that is amenable to symbolic evaluation:

theorem unfold-contents :

assumes cp E

and τ ⊨ δ self

shows (τ ⊨ E (self.contents())) =

```
(τ ⊨ E (if self.boss ≐ null
  then Set{self.salary}
```

```
else self .boss .contents()->includingSet(self .salary) endif))
```

Since we have only one interpretation function, we need the corresponding operation on the pre-state:

```
consts contentsATpre :: Person => Set-Integer (<(I(-).contents@pre'('))> 50)
```

axiomatization where *contentsATpre-def*:

```
(self).contents@pre() = (λ τ.
  SOME res. let res = λ -. res in
  if τ ⊨ (δ self)
  then ((τ ⊨ true) ∧
    (τ ⊨ (res ≐ if (self).boss@pre ≐ null
      then Set{(self).salary@pre}
      else (self).boss@pre .contents@pre()
      ->includingSet(self .salary@pre)
      endif)))
  else τ ⊨ res ≐ invalid)
and cp0-contents-at-pre:(X .contents@pre()) τ = ((λ -. X τ) .contents@pre()) τ
```

interpretation *contentsATpre* : *contract0 contentsATpre λ self. true*

```
λ self res. res ≐ if (self .boss@pre ≐ null)
  then (Set{self .salary@pre})
  else (self .boss@pre .contents@pre()
    ->includingSet(self .salary@pre))
endif
```

Again, we derive via *contents.unfold2* a Knaster-Tarski like Fixpoint rule that is amenable to symbolic evaluation:

theorem *unfold-contentsATpre* :

```
assumes cp E
and τ ⊨ δ self
shows (τ ⊨ E (self .contents@pre())) =
  (τ ⊨ E (if self .boss@pre ≐ null
    then Set{self .salary@pre}
    else self .boss@pre .contents@pre()->includingSet(self .salary@pre) endif))
```

Note that these @pre variants on methods are only available on queries, i. e., operations without side-effect.

4.12. OCL Part: The Contract of a User-defined Method

The example specification in high-level OCL input syntax reads as follows:

```
context Person::insert(x:Integer)
pre: true
post: contents():Set(Integer)
contents() = contents@pre()->including(x)
```

This boils down to:

```
definition insert :: Person => Integer => Void (<(I(-).insert'(-'))> 50)
where self .insert(x) ≡
```

$$\begin{aligned}
&(\lambda \tau. \text{SOME } res. \text{let } res = \lambda -. res \text{ in} \\
&\quad \text{if } (\tau \models (\delta \text{ self})) \wedge (\tau \models v \ x) \\
&\quad \text{then } (\tau \models \text{true} \wedge \\
&\quad \quad (\tau \models ((\text{self}).\text{contents}() \triangleq (\text{self}).\text{contents}@pre() \rightarrow \text{including}_{\text{Set}}(x)))) \\
&\quad \text{else } \tau \models res \triangleq \text{invalid})
\end{aligned}$$

The semantic consequences of this definition were computed inside this locale interpretation:

interpretation *insert* : *contract1 insert* $\lambda \text{ self } x. \text{true}$
 $\lambda \text{ self } x \text{ res. } ((\text{self}.\text{contents}()) \triangleq$
 $\quad (\text{self}.\text{contents}@pre() \rightarrow \text{including}_{\text{Set}}(x)))$

The result of this locale interpretation for our *Analysis-OCL.insert* contract is the following set of properties, which serves as basis for automated deduction on them:

Name	Theorem
<i>insert.strict0</i>	$(\text{invalid.insert}(X)) = \text{invalid}$
<i>insert.nullstrict0</i>	$(\text{null.insert}(X)) = \text{invalid}$
<i>insert.strict1</i>	$(\text{self.insert}(\text{invalid})) = \text{invalid}$
<i>insert.cppRE</i>	$\text{true } \tau = \text{true } \tau$
<i>insert.cppOST</i>	$(\text{self}.\text{contents}() \triangleq \text{self}.\text{contents}@pre() \rightarrow \text{including}_{\text{Set}}(a1.0)) \tau = (\lambda -. \text{self } \tau.\text{contents}() \triangleq \lambda -. \text{self } \tau.\text{contents}@pre() \rightarrow \text{including}_{\text{Set}}(\lambda -. a1.0 \ \tau)) \tau$
<i>insert.cp-pre</i>	$\llbracket \text{cp self}'; \text{cp a1}' \rrbracket \implies \text{cp } (\lambda X. \text{true})$
<i>insert.cp-post</i>	$\llbracket \text{cp self}'; \text{cp a1}'; \text{cp res} \rrbracket \implies \text{cp } (\lambda X. \text{self}' X.\text{contents}() \triangleq \text{self}' X.\text{contents}@pre() \rightarrow \text{including}_{\text{Set}}(a1' X))$
<i>insert.cp</i>	$\llbracket \text{cp self}'; \text{cp a1}'; \text{cp res} \rrbracket \implies \text{cp } (\lambda X. \text{self}' X.\text{insert}(a1' X))$
<i>insert.cp0</i>	$(\text{self.insert}(a1.0)) \tau = (\lambda -. \text{self } \tau.\text{insert}(\lambda -. a1.0 \ \tau)) \tau$
<i>insert.def-scheme</i>	$\text{self.insert}(a1.0) \equiv \lambda \tau. \text{SOME } res. \text{let } res = \lambda -. res \text{ in if } \tau \models \delta \text{ self} \wedge \tau \models v \ a1.0 \text{ then } \tau \models \text{true} \wedge \tau \models \text{self}.\text{contents}() \triangleq \text{self}.\text{contents}@pre() \rightarrow \text{including}_{\text{Set}}(a1.0) \text{ else } \tau \models res \triangleq \text{invalid}$
<i>insert.unfold</i>	$\llbracket \text{cp } E; \tau \models \delta \text{ self} \wedge \tau \models v \ a1.0; \tau \models \text{true}; \exists res. \tau \models \text{self}.\text{contents}() \triangleq \text{self}.\text{contents}@pre() \rightarrow \text{including}_{\text{Set}}(a1.0); \wedge res. \tau \models \text{self}.\text{contents}() \triangleq \text{self}.\text{contents}@pre() \rightarrow \text{including}_{\text{Set}}(a1.0) \implies \tau \models E (\lambda -. res) \rrbracket \implies \tau \models E (\text{self.insert}(a1.0))$
<i>insert.unfold2</i>	$\llbracket \text{cp } E; \tau \models \delta \text{ self} \wedge \tau \models v \ a1.0; \tau \models \text{true}; \tau \models \text{POST}' \text{ self } a1.0; \wedge res. (\text{self}.\text{contents}() \triangleq \text{self}.\text{contents}@pre() \rightarrow \text{including}_{\text{Set}}(a1.0)) = (\text{POST}' \text{ self } a1.0 \text{ and } (res \triangleq \text{BODY self } a1.0)) \rrbracket \implies (\tau \models E (\text{self.insert}(a1.0))) = (\tau \models E (\text{BODY self } a1.0))$

Table 4.1. – Semantic properties resulting from a user-defined operation contract.

Annex B.

Bibliography

Bibliography

- [1] P. B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Kluwer Academic Publishers, Dordrecht, 2nd edition, 2002. ISBN 1-402-00763-9.
- [2] C. Barrett and C. Tinelli. Cvc3. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 2007. ISBN 978-3-540-73367-6. doi: 10.1007/978-3-540-73368-3_34.
- [3] A. D. Brucker. *An Interactive Proof Environment for Object-oriented Specifications*. PhD thesis, ETH Zurich, Mar. 2007. URL <http://www.brucker.ch/bibliography/abstract/brucker-interactive-2007>. ETH Dissertation No. 17097.
- [4] A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-ocl-book-2006>.
- [5] A. D. Brucker and B. Wolff. An extensible encoding of object-oriented data models in hol. *Journal of Automated Reasoning*, 41:219–249, 2008. ISSN 0168-7433. doi: 10.1007/s10817-008-9108-3. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-extensible-2008-b>.
- [6] A. D. Brucker and B. Wolff. HOL-OCL – A Formal Proof Environment for UML/OCL. In J. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering (FASE08)*, number 4961 in *Lecture Notes in Computer Science*, pages 97–100. Springer-Verlag, Heidelberg, 2008. doi: 10.1007/978-3-540-78743-3_8. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-ocl-2008>.
- [7] A. D. Brucker, J. Doser, and B. Wolff. Semantic issues of OCL: Past, present, and future. *Electronic Communications of the EASST*, 5, 2006. ISSN 1863-2122. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-semantic-2006-b>.
- [8] A. D. Brucker, D. Chiorean, T. Clark, B. Demuth, M. Gogolla, D. Plotnikov, B. Rumpe, E. D. Willink, and B. Wolff. Report on the Aachen OCL meeting. In J. Cabot, M. Gogolla, I. Rath, and E. Willink, editors, *Proceedings of the MODELS 2013 OCL Workshop (OCL 2013)*, volume 1092 of *CEUR Workshop Proceedings*, pages 103–111. CEUR-WS.org, 2013. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-summary-aachen-2013>.
- [9] A. D. Brucker, D. Longuet, F. Tuong, and B. Wolff. On the semantics of object-oriented data structures and path expressions. In J. Cabot, M. Gogolla, I. Ráth, and E. D. Willink, editors, *Proceedings of the MODELS 2013 OCL Workshop (OCL 2013)*, volume 1092 of *CEUR Workshop Proceedings*, pages 23–32. CEUR-WS.org, 2013. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-path-expressions-2013>. An extended version of this paper is available as LRI Technical Report 1565.
- [10] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.
- [11] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The amsterdam manifesto on OCL. In T. Clark and J. Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *Lecture Notes in Computer Science*, pages 115–149, Heidelberg, 2002. Springer-Verlag. ISBN 3-540-43169-1.

- [12] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-78799-0. doi: 10.1007/978-3-540-78800-3_24.
- [13] F. Haftmann and M. Wenzel. Constructive type classes in isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 160–174. Springer, 2006. ISBN 978-3-540-74463-4. doi: 10.1007/978-3-540-74464-1_11. URL https://doi.org/10.1007/978-3-540-74464-1_11.
- [14] A. Hamie, F. Civello, J. Howse, S. Kent, and R. Mitchell. Reflections on the Object Constraint Language. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language. «UML»'98: Beyond the Notation*, volume 1618 of *Lecture Notes in Computer Science*, pages 162–172, Heidelberg, 1998. Springer-Verlag. ISBN 3-540-66252-9. doi: 10.1007/b72309.
- [15] P. Kosiuczenko. Specification of invariability in OCL. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *Lecture Notes in Computer Science*, pages 676–691, Heidelberg, 2006. Springer-Verlag. ISBN 978-3-540-45772-5. doi: 10.1007/11880240_47.
- [16] L. Mandel and M. V. Cengarle. On the expressive power of OCL. In J. M. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems (FM)*, volume 1708 of *Lecture Notes in Computer Science*, pages 854–874, Heidelberg, 1999. Springer-Verlag. ISBN 3-540-66587-0.
- [17] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2002. doi: 10.1007/3-540-45949-9.
- [18] Object Management Group. UML 2.0 OCL specification, Oct. 2003. Available as OMG document ptc/03-10-14.
- [19] Object Management Group. UML 2.3.1 OCL specification, Feb. 2012. Available as OMG document formal/2012-01-01.
- [20] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.
- [21] M. Wenzel and B. Wolff. Building formal method tools in the Isabelle/Isar framework. In K. Schneider and J. Brandt, editors, *TPHOLs 2007*, number 4732 in *Lecture Notes in Computer Science*, pages 352–367. Springer-Verlag, Heidelberg, 2007. doi: 10.1007/978-3-540-74591-4_26.
- [22] M. M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, TU München, München, Feb. 2002. URL <http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html>.

Annex C.

The OCL And Featherweight OCL Syntax

Table 4.2. – Comparison of different concrete syntax variants for OCL

	OCL	Featherweight OCL	Logical Constant
OclAny	$_ = _$	$op \triangleq$	<i>UML-Logic.StrongEq</i>
	$_ <> _$	$op \triangleleft \triangleright$	<i>notequal</i>
	$_ \rightarrow oclAsSet (_)$		
	$_ .oclIsNew ()$	$_ .oclIsNew ()$	<i>UML-State.OclIsNew</i>
	$not (_ \rightarrow oclIsUndefined ())$	$\delta _$	<i>UML-Logic.defined</i>
	$not (_ \rightarrow oclIsInvalid ())$	$v _$	<i>UML-Logic.valid</i>
	$_ \rightarrow oclAsType (_)$		
	$_ \rightarrow oclIsTypeOf (_)$		
	$_ \rightarrow oclIsKindOf (_)$		
	$_ \rightarrow oclIsInState (_)$		
	$_ \rightarrow oclType ()$		
	$_ \rightarrow oclLocale ()$		
OclVoid	$_ = _$	$op \triangleq$	<i>UML-Logic.StrongEq</i>
	$_ <> _$	$op \triangleleft \triangleright$	<i>notequal</i>
	$_ \rightarrow oclAsSet (_)$		
	$_ .oclIsNew ()$	$_ .oclIsNew ()$	<i>UML-State.OclIsNew</i>
	$not (_ \rightarrow oclIsUndefined ())$	$\delta _$	<i>UML-Logic.defined</i>
	$not (_ \rightarrow oclIsInvalid ())$	$v _$	<i>UML-Logic.valid</i>
	$_ \rightarrow oclAsType (_)$		
	$_ \rightarrow oclIsTypeOf (_)$		
	$_ \rightarrow oclIsKindOf (_)$		
	$_ \rightarrow oclIsInState (_)$		
	$_ \rightarrow oclType ()$		
	$_ \rightarrow oclLocale ()$		
OclInvalid	$_ = _$	$op \triangleq$	<i>UML-Logic.StrongEq</i>
	$_ <> _$	$op \triangleleft \triangleright$	<i>notequal</i>
	$_ \rightarrow oclAsSet (_)$		
	$_ .oclIsNew ()$	$_ .oclIsNew ()$	<i>UML-State.OclIsNew</i>
	$not (_ \rightarrow oclIsUndefined ())$	$\delta _$	<i>UML-Logic.defined</i>
	$not (_ \rightarrow oclIsInvalid ())$	$v _$	<i>UML-Logic.valid</i>
	$_ \rightarrow oclAsType (_)$		
	$_ \rightarrow oclIsTypeOf (_)$		
	$_ \rightarrow oclIsKindOf (_)$		
	$_ \rightarrow oclIsInState (_)$		
	$_ \rightarrow oclType ()$		
	$_ \rightarrow oclLocale ()$		
Real	$_ + _$	$op \text{+}_{\text{real}}$	<i>UML-Real.OclAdd_{Real}</i>
	$_ - _$	$op \text{-}_{\text{real}}$	<i>UML-Real.OclMinus_{Real}</i>
	$_ * _$	$op \text{*}_{\text{real}}$	<i>UML-Real.OclMult_{Real}</i>
	$_ / _$		
	$_ .abs ()$		

Continued on next page

	OCL	Featherweight OCL	Logical Constant
Real Literals	<code>_ .floor()</code>		
	<code>_ .round()</code>		
	<code>_ .max()</code>		
	<code>_ .min()</code>		
	<code>_ < _</code>	$op <_{\text{real}}$	<i>UML-Real.OclLess_{Real}</i>
	<code>_ > _</code>		
	<code>_ <= _</code>	$op \leq_{\text{real}}$	<i>UML-Real.OclLe_{Real}</i>
	<code>_ >= _</code>		
	<code>_ .toString()</code>		
	<code>_ .div(_)</code>	$op \text{div}_{\text{real}}$	<i>UML-Real.OclDivision_{Real}</i>
	<code>_ .mod(_)</code>	$op \text{mod}_{\text{real}}$	<i>UML-Real.OclModulus_{Real}</i>
	<code>_ ->oclAsType(Integer)</code>	<code>_ ->oclAsType_{Real}(Integer)</code>	<i>UML-Library.OclAsInteger_{Real}</i>
	<code>_ ->oclAsType(Boolean)</code>	<code>_ ->oclAsType_{Real}(Boolean)</code>	<i>UML-Library.OclAsBoolean_{Real}</i>
Integer	<code>0.0</code>	0.0	<i>UML-Real.OclReal0</i>
	<code>1.0</code>	1.0	<i>UML-Real.OclReal1</i>
	<code>2.0</code>	2.0	<i>UML-Real.OclReal2</i>
	<code>3.0</code>	3.0	<i>UML-Real.OclReal3</i>
	<code>4.0</code>	4.0	<i>UML-Real.OclReal4</i>
	<code>5.0</code>	5.0	<i>UML-Real.OclReal5</i>
	<code>6.0</code>	6.0	<i>UML-Real.OclReal6</i>
	<code>7.0</code>	7.0	<i>UML-Real.OclReal7</i>
	<code>8.0</code>	8.0	<i>UML-Real.OclReal8</i>
	<code>9.0</code>	9.0	<i>UML-Real.OclReal9</i>
	<code>10.0</code>	10.0	<i>UML-Real.OclReal10</i>
		π	<i>UML-Real.OclRealpi</i>
	<code>- -</code>	$op \neg_{\text{int}}$	<i>UML-Integer.OclMinus_{Integer}</i>
Integer Literals	<code>- +</code>	$op +_{\text{int}}$	<i>UML-Integer.OclAdd_{Integer}</i>
	<code>- -</code>		
	<code>- *</code>	$op *_{\text{int}}$	<i>UML-Integer.OclMult_{Integer}</i>
	<code>- /</code>		
	<code>_ .abs()</code>		
	<code>_ div (_)</code>	$op \text{div}_{\text{int}}$	<i>UML-Integer.OclDivision_{Integer}</i>
	<code>_ mod (_)</code>	$op \text{mod}_{\text{int}}$	<i>UML-Integer.OclModulus_{Integer}</i>
	<code>_ .max()</code>		
	<code>_ .min()</code>		
	<code>_ .toString()</code>		
	<code>_ < _</code>	$op <_{\text{int}}$	<i>UML-Integer.OclLess_{Integer}</i>
	<code>_ <= _</code>	$op \leq_{\text{int}}$	<i>UML-Integer.OclLe_{Integer}</i>
	<code>_ ->oclAsType(Real)</code>	<code>_ ->oclAsType_{Int}(Real)</code>	<i>UML-Library.OclAsReal_{Int}</i>
	<code>_ ->oclAsType(Boolean)</code>	<code>_ ->oclAsType_{Int}(Boolean)</code>	<i>UML-Library.OclAsBoolean_{Int}</i>
Integer Literals	<code>0</code>	0	<i>UML-Integer.OclInt0</i>
	<code>1</code>	1	<i>UML-Integer.OclInt1</i>
	<code>2</code>	2	<i>UML-Integer.OclInt2</i>
	<code>3</code>	3	<i>UML-Integer.OclInt3</i>

Continued on next page

	OCL	Featherweight OCL	Logical Constant
	4	4	<i>UML-Integer.OclInt4</i>
	5	5	<i>UML-Integer.OclInt5</i>
	6	6	<i>UML-Integer.OclInt6</i>
	7	7	<i>UML-Integer.OclInt7</i>
	8	8	<i>UML-Integer.OclInt8</i>
	9	9	<i>UML-Integer.OclInt9</i>
	10	10	<i>UML-Integer.OclInt10</i>
String and String Literals	<code>_ + _</code>	<i>op</i> <code>+string</code>	<i>UML-String.OclAddString</i>
	<code>_ .size()</code>		
	<code>_ .concat(_)</code>		
	<code>_ .substring(_ , _)</code>		
	<code>_ .toInteger()</code>		
	<code>_ .toReal()</code>		
	<code>_ .toUpperCase()</code>		
	<code>_ .toLowerCase()</code>		
	<code>_ .indexOf()</code>		
	<code>_ .equalsIgnoreCase(_)</code>		
	<code>_ .at(_)</code>		
	<code>_ .characters()</code>		
	<code>_ .toBoolean()</code>		
	<code>_ < _</code>		
	<code>_ > _</code>		
	<code>_ <= _</code>		
	<code>_ >= _</code>		
	<code>a</code>	<code>a</code>	<i>UML-String.OclStringa</i>
	<code>b</code>	<code>b</code>	<i>UML-String.OclStringb</i>
	<code>c</code>	<code>c</code>	<i>UML-String.OclStringc</i>
Boolean and Core Logic	<code>_ or _</code>	<i>op</i> <code>or</code>	<i>UML-Logic.OclOr</i>
	<code>_ xor _</code>		
	<code>_ and _</code>	<i>op</i> <code>and</code>	<i>UML-Logic.OclAnd</i>
	<code>not _</code>	<i>not</i>	<i>UML-Logic.OclNot</i>
	<code>_ implies _</code>	<i>op</i> <code>implies</code>	<i>UML-Logic.OclImplies</i>
	<code>_ .toString()</code>		
	<code>if _ then _ else _ endif</code>	<i>if</i> <code>_ then _ else _ endif</code>	<i>UML-Logic.OclIf</i>
	<code>_ = _</code>	<i>op</i> <code>=</code>	<i>UML-Logic.StrictRefEq</i>
	<code>_ <> _</code>	<i>op</i> <code><></code>	<i>notequal</i>
		<code>_ != _</code>	<i>OclNonValid</i>
		<code>_ = _</code>	<i>UML-Logic.OclValid</i>
	<code>_ = _</code>	<i>op</i> <code>=</code>	<i>UML-Logic.StrongEq</i>
Iterators on Set	<code>Set (_)</code>	<i>Set</i> (<i>type</i> ⁰)	<i>UML-Types.Set_{base} type</i>
	<code>Set{ }</code>	<i>Set</i> { }	<i>UML-Set.mtSet</i>
	<code>Set{ _ }</code>	<i>Set</i> { <i>args</i> ⁰ }	<i>OclFinset</i>
	<code>_ ->union(_)</code>	<code>_ ->union_{Set}(_)</code>	<i>UML-Set.OclUnion</i>
	<code>_ = _</code>	<i>op</i> <code>=</code>	<i>UML-Logic.StrongEq</i>

Continued on next page

OCL	Featherweight OCL	Logical Constant
<code>_ ->intersection(_)</code>	<code>_ ->intersectionSet(_)</code>	<i>UML-Set.OclIntersection</i>
<code>_ - _</code>		
<code>_ ->including(_)</code>	<code>_ ->includingSet(_)</code>	<i>UML-Set.OclIncluding</i>
<code>_ ->excluding(_)</code>	<code>_ ->excludingSet(_)</code>	<i>UML-Set.OclExcluding</i>
<code>_ ->symmetricDifference(_)</code>		
<code>_ ->count(_)</code>	<code>_ ->countSet(_)</code>	<i>UML-Set.OclCount</i>
<code>_ ->flatten()</code>		
<code>_ ->selectByKind(_)</code>		
<code>_ ->selectByType(_)</code>		
<code>_ ->reject(_ _)</code>	<code>_ ->rejectSet(id _)</code>	<i>OclRejectSet</i>
<code>_ ->select(_ _)</code>	<code>_ ->selectSet(id _)</code>	<i>OclSelectSet</i>
<code>_ ->iterate(_ ; _ = _ _)</code>	<code>_ ->iterateSet(idt⁰ ; idt⁰ = any⁰ any⁰)</code>	<i>OclIterateSet</i>
<code>_ ->exists(_ _)</code>	<code>_ ->existsSet(id _)</code>	<i>OclExistSet</i>
<code>_ ->forall(_ _)</code>	<code>_ ->forallSet(id _)</code>	<i>OclForallSet</i>
<code>_ ->asSequence()</code>	<code>_ ->asSequenceSet()</code>	<i>UML-Library.OclAsSeqSet</i>
<code>_ ->asBag()</code>	<code>_ ->asBagSet()</code>	<i>UML-Library.OclAsBagSet</i>
<code>_ ->asPair()</code>	<code>_ ->asPairSet()</code>	<i>UML-Library.OclAsPairSet</i>
<code>_ ->sum()</code>	<code>_ ->sumSet()</code>	<i>UML-Set.OclSum</i>
<code>_ ->excludesAll(_)</code>	<code>_ ->excludesAllSet(_)</code>	<i>UML-Set.OclExcludesAll</i>
<code>_ ->includesAll(_)</code>	<code>_ ->includesAllSet(_)</code>	<i>UML-Set.OclIncludesAll</i>
<code>_ ->any()</code>	<code>_ ->anySet()</code>	<i>UML-Set.OclANY</i>
<code>_ ->notEmpty()</code>	<code>_ ->notEmptySet()</code>	<i>UML-Set.OclNotEmpty</i>
<code>_ ->isEmpty()</code>	<code>_ ->isEmptySet()</code>	<i>UML-Set.OclIsEmpty</i>
<code>_ ->size()</code>	<code>_ ->sizeSet()</code>	<i>UML-Set.OclSize</i>
<code>_ ->excludes(_)</code>	<code>_ ->excludesSet(_)</code>	<i>UML-Set.OclExcludes</i>
<code>_ ->includes(_)</code>	<code>_ ->includesSet(_)</code>	<i>UML-Set.OclIncludes</i>
Sequence and Iterators on Sequence	<code>Sequence(_)</code>	<i>UML-Types.Sequence_{base type}</i>
	<code>Sequence{ }</code>	<i>UML-Sequence.mtSequence</i>
	<code>Sequence{ _ }</code>	<i>OclFinsequence</i>
	<code>_ ->any()</code>	<i>UML-Sequence.OclANY</i>
	<code>_ ->notEmpty()</code>	<i>UML-Sequence.OclNotEmpty</i>
	<code>_ ->isEmpty()</code>	<i>UML-Sequence.OclIsEmpty</i>
	<code>_ ->size()</code>	<i>UML-Sequence.OclSize</i>
	<code>_ ->select(_ _)</code>	<i>OclSelectSeq</i>
	<code>_ ->collect(_ _)</code>	<i>OclCollectSeq</i>
	<code>_ ->exists(_ _)</code>	<i>OclExistSeq</i>
	<code>_ ->forall(_ _)</code>	<i>OclForallSeq</i>
	<code>_ ->iterate(_ ; _ : _ = _ _)</code>	<i>OclIterateSeq</i>
	<code>_ ->last()</code>	<i>UML-Sequence.OclLast</i>
	<code>_ ->first()</code>	<i>UML-Sequence.OclFirst</i>
	<code>_ ->at(_)</code>	<i>UML-Sequence.OclAt</i>
	<code>_ ->union(_)</code>	<i>UML-Sequence.OclUnion</i>
	<code>_ ->append(_)</code>	<i>UML-Sequence.OclAppend</i>
	<code>_ ->excluding(_)</code>	<i>UML-Sequence.OclExcluding</i>

Continued on next page

	OCL	Featherweight OCL	Logical Constant
	<code>_ ->including(_)</code>	<code>_ ->includingSeq(_)</code>	<i>UML-Sequence.OclIncluding</i>
	<code>_ ->prepend(_)</code>	<code>_ ->prependSeq(_)</code>	<i>UML-Sequence.OclPrepend</i>
	<code>_ ->asSet()</code>	<code>_ ->asSetSeq()</code>	<i>UML-Library.OclAsSetSeq</i>
	<code>_ ->asBag()</code>	<code>_ ->asBagSeq()</code>	<i>UML-Library.OclAsBagSeq</i>
	<code>_ ->asPair()</code>	<code>_ ->asPairSeq()</code>	<i>UML-Library.OclAsPairSeq</i>
	<code>Bag(_)</code>	<code>Bag(type⁰)</code>	<i>UML-Types.Bag_{base} type</i>
	<code>Bag{ }</code>	<code>Bag{ }</code>	<i>UML-Bag.mtBag</i>
	<code>Bag{ _ }</code>	<code>Bag{ args⁰ }</code>	<i>OclFinbag</i>
Bag and Iterators on Bag	<code>_ ->sum()</code>	<code>_ ->sumBag()</code>	<i>UML-Bag.OclSum</i>
	<code>_ ->count(_)</code>	<code>_ ->countBag(_)</code>	<i>UML-Bag.OclCount</i>
	<code>_ ->intersection(_)</code>	<code>_ ->intersectionBag(_)</code>	<i>UML-Bag.OclIntersection</i>
	<code>_ ->union(_)</code>	<code>_ ->unionBag(_)</code>	<i>UML-Bag.OclUnion</i>
	<code>_ ->excludesAll(_)</code>	<code>_ ->excludesAllBag(_)</code>	<i>UML-Bag.OclExcludesAll</i>
	<code>_ ->includesAll(_)</code>	<code>_ ->includesAllBag(_)</code>	<i>UML-Bag.OclIncludesAll</i>
	<code>_ ->reject(_ _)</code>	<code>_ ->rejectBag(id _)</code>	<i>OclRejectBag</i>
	<code>_ ->select(_ _)</code>	<code>_ ->selectBag(id _)</code>	<i>OclSelectBag</i>
	<code>_ ->iterate(_ ; _ = _ _)</code>	<code>_ ->iterateBag(idt ; idt = any any)</code>	<i>OclIterateBag</i>
	<code>_ ->exists(_ _)</code>	<code>_ ->existsBag(id _)</code>	<i>OclExistBag</i>
	<code>_ ->forAll(_ _)</code>	<code>_ ->forAllBag(id _)</code>	<i>OclForallBag</i>
	<code>_ ->any()</code>	<code>_ ->anyBag()</code>	<i>UML-Bag.OclANY</i>
	<code>_ ->notEmpty()</code>	<code>_ ->notEmptyBag()</code>	<i>UML-Bag.OclNotEmpty</i>
	<code>_ ->isEmpty()</code>	<code>_ ->isEmptyBag()</code>	<i>UML-Bag.OclIsEmpty</i>
	<code>_ ->size()</code>	<code>_ ->sizeBag()</code>	<i>UML-Bag.OclSize</i>
	<code>_ ->excludes(_)</code>	<code>_ ->excludesBag(_)</code>	<i>UML-Bag.OclExcludes</i>
	<code>_ ->includes(_)</code>	<code>_ ->includesBag(_)</code>	<i>UML-Bag.OclIncludes</i>
	<code>_ ->excluding(_)</code>	<code>_ ->excludingBag(_)</code>	<i>UML-Bag.OclExcluding</i>
	<code>_ ->including(_)</code>	<code>_ ->includingBag(_)</code>	<i>UML-Bag.OclIncluding</i>
	<code>_ ->asSet()</code>	<code>_ ->asSetBag()</code>	<i>UML-Library.OclAsSetBag</i>
	<code>_ ->asSeq()</code>	<code>_ ->asSeqBag()</code>	<i>UML-Library.OclAsSeqBag</i>
	<code>_ ->asPair()</code>	<code>_ ->asPairBag()</code>	<i>UML-Library.OclAsPairBag</i>
		<code>Pair(type⁰ , type⁰)</code>	<i>UML-Types.Pair_{base} type</i>
		<code>Pair{ _ , _ }</code>	<i>UML-Pair.OclPair</i>
		<code>_ .Second()</code>	<i>UML-Pair.OclSecond</i>
		<code>_ .First()</code>	<i>UML-Pair.OclFirst</i>
	<code>_ ->asSequence()</code>	<code>_ ->asSequencePair()</code>	<i>UML-Library.OclAsSeqPair</i>
	<code>_ ->asSet()</code>	<code>_ ->asSetPair()</code>	<i>UML-Library.OclAsSetPair</i>
State Access	<code>_ .allInstances()</code>	<code>_ .allInstances()</code>	<i>UML-State.OclAllInstances-at-post</i>
		<code>_ .allInstances@pre()</code>	<i>UML-State.OclAllInstances-at-pre</i>
		<code>_ .oclIsDeleted()</code>	<i>UML-State.OclIsDeleted</i>
		<code>_ .oclIsMaintained()</code>	<i>UML-State.OclIsMaintained</i>
		<code>_ .oclIsAbsent()</code>	<i>UML-State.OclIsAbsent</i>
		<code>_ ->oclIsModifiedOnly()</code>	<i>UML-State.OclIsModifiedOnly</i>
	<code>_ @pre _</code>	<code>_ @pre _</code>	<i>UML-State.OclSelf-at-pre</i>

Continued on next page

OCL	Featherweight OCL	Logical Constant
	<i>_ @post _</i>	<i>UML-State.OclSelf-at-post</i>

Annex D.

Table of Contents

Contents

A. Formal Semantics of OCL	3
0.1. Introduction	4
0.2. Background	6
0.2.1. Formal Foundation	6
0.2.1.1. A Gentle Introduction to Isabelle	6
0.2.1.2. Higher-order Logic (HOL)	8
0.2.2. How this Annex A was Generated from Isabelle/HOL Theories	10
0.3. The Essence of UML-OCL Semantics	11
0.3.1. The Theory Organization	11
0.3.2. Denotational Semantics of Types	11
0.3.3. Denotational Semantics of Constants and Operations	12
0.3.4. Logical Layer	14
0.3.5. Algebraic Layer	15
0.3.6. Object-oriented Datatype Theories	17
0.3.6.1. A Denotational Space for Class-Models: Object Universes	18
0.3.6.2. Denotational Semantics of Accessors on Objects and Associations	19
0.3.6.3. Logic Properties of Class-Models	22
0.3.6.4. Algebraic Properties of the Class-Models	22
0.3.6.5. Other Operations on States	23
0.3.7. Data Invariants	23
0.3.8. Operation Contracts	24
1. Formalization I: OCL Types and Core Definitions	26
1.1. Preliminaries	26
1.1.1. Notations for the Option Type	26
1.1.2. Common Infrastructure for all OCL Types	26
1.1.3. Accommodation of Basic Types to the Abstract Interface	27
1.1.4. The Common Infrastructure of Object Types (Class Types) and States.	28
1.1.5. Common Infrastructure for all OCL Types (II): Valuations as OCL Types	28
1.1.6. The fundamental constants 'invalid' and 'null' in all OCL Types	29
1.2. Basic OCL Value Types	29
1.3. Some OCL Collection Types	30
1.3.1. The Construction of the Pair Type (Tuples)	30
1.3.2. The Construction of the Set Type	31
1.3.3. The Construction of the Bag Type	32
1.3.4. The Construction of the Sequence Type	32
1.3.5. Discussion: The Representation of UML/OCL Types in Featherweight OCL	33
2. Formalization II: OCL Terms and Library Operations	35
2.1. The Operations of the Boolean Type and the OCL Logic	35
2.1.1. Basic Constants	35
2.1.2. Validity and Definedness	35
2.1.3. The Equalities of OCL	36
2.1.3.1. Definition	37
2.1.3.2. Fundamental Predicates on Strong Equality	38

2.1.4.	Logical Connectives and their Universal Properties	38
2.1.5.	A Standard Logical Calculus for OCL	42
2.1.5.1.	Global vs. Local Judgements	42
2.1.5.2.	Local Validity and Meta-logic	42
2.1.5.3.	Local Judgements and Strong Equality	45
2.1.6.	OCL's if then else endif	46
2.1.7.	Fundamental Predicates on Basic Types: Strict (Referential) Equality	47
2.1.8.	Laws to Establish Definedness (δ -closure)	47
2.1.9.	A Side-calculus for Constant Terms	47
2.2.	Property Profiles for OCL Operators via Isabelle Locales	50
2.2.1.	Property Profiles for Monadic Operators	50
2.2.2.	Property Profiles for Single	51
2.2.3.	Property Profiles for Binary Operators	51
2.2.4.	Fundamental Predicates on Basic Types: Strict (Referential) Equality	53
2.2.5.	Test Statements on Boolean Operations.	54
2.3.	Basic Type Void: Operations	55
2.3.1.	Fundamental Properties on Voids: Strict Equality	55
2.3.1.1.	Definition	55
2.3.2.	Basic Void Constants	55
2.3.3.	Validity and Definedness Properties	55
2.3.4.	Test Statements	56
2.4.	Basic Type Integer: Operations	56
2.4.1.	Fundamental Predicates on Integers: Strict Equality	56
2.4.2.	Basic Integer Constants	56
2.4.3.	Arithmetical Operations	56
2.4.3.1.	Definition	56
2.4.3.2.	Basic Properties	57
2.4.3.3.	Execution with Invalid or Null or Zero as Argument	57
2.4.4.	Test Statements	58
2.5.	Basic Type Real: Operations	59
2.5.1.	Fundamental Predicates on Reals: Strict Equality	59
2.5.2.	Basic Real Constants	59
2.5.3.	Arithmetical Operations	59
2.5.3.1.	Definition	59
2.5.3.2.	Basic Properties	60
2.5.3.3.	Execution with Invalid or Null or Zero as Argument	61
2.5.4.	Test Statements	61
2.6.	Basic Type String: Operations	62
2.6.1.	Fundamental Properties on Strings: Strict Equality	62
2.6.2.	Basic String Constants	62
2.6.3.	String Operations	63
2.6.3.1.	Definition	63
2.6.3.2.	Basic Properties	63
2.6.4.	Test Statements	63
2.7.	Collection Type Pairs: Operations	64
2.7.1.	Semantic Properties of the Type Constructor	64
2.7.2.	Fundamental Properties of Strict Equality	64
2.7.3.	Standard Operations Definitions	64
2.7.3.1.	Definition: Pair Constructor	64
2.7.3.2.	Definition: First	65
2.7.3.3.	Definition: Second	65
2.7.4.	Logical Properties	65

2.7.5.	Algebraic Execution Properties	65
2.7.6.	Test Statements	65
2.8.	Collection Type Bag: Operations	66
2.8.1.	As a Motivation for the (infinite) Type Construction: Type-Extensions as Bags	66
2.8.2.	Basic Properties of the Bag Type	67
2.8.3.	Definition: Strict Equality	68
2.8.4.	Constants: mtBag	68
2.8.5.	Definition: Including	68
2.8.6.	Definition: Excluding	69
2.8.7.	Definition: Includes	69
2.8.8.	Definition: Excludes	69
2.8.9.	Definition: Size	70
2.8.10.	Definition: IsEmpty	70
2.8.11.	Definition: NotEmpty	70
2.8.12.	Definition: Any	70
2.8.13.	Definition: Forall	70
2.8.14.	Definition: Exists	71
2.8.15.	Definition: Iterate	71
2.8.16.	Definition: Select	71
2.8.17.	Definition: Reject	72
2.8.18.	Definition: IncludesAll	72
2.8.19.	Definition: ExcludesAll	72
2.8.20.	Definition: Union	72
2.8.21.	Definition: Intersection	72
2.8.22.	Definition: Count	73
2.8.23.	Definition (future operators)	73
2.8.24.	Logical Properties	73
2.8.25.	Execution Laws with Invalid or Null or Infinite Set as Argument	73
2.8.26.	Test Statements	75
2.9.	Collection Type Set: Operations	75
2.9.1.	As a Motivation for the (infinite) Type Construction: Type-Extensions as Sets	75
2.9.2.	Basic Properties of the Set Type	77
2.9.3.	Definition: Strict Equality	77
2.9.4.	Constants: mtSet	78
2.9.5.	Definition: Including	78
2.9.6.	Definition: Excluding	78
2.9.7.	Definition: Includes	79
2.9.8.	Definition: Excludes	79
2.9.9.	Definition: Size	79
2.9.10.	Definition: IsEmpty	79
2.9.11.	Definition: NotEmpty	79
2.9.12.	Definition: Any	80
2.9.13.	Definition: Forall	80
2.9.14.	Definition: Exists	80
2.9.15.	Definition: Iterate	80
2.9.16.	Definition: Select	81
2.9.17.	Definition: Reject	81
2.9.18.	Definition: IncludesAll	81
2.9.19.	Definition: ExcludesAll	81
2.9.20.	Definition: Union	82
2.9.21.	Definition: Intersection	82
2.9.22.	Definition (future operators)	82

2.9.23. Logical Properties	82
2.9.24. Execution Laws with Invalid or Null or Infinite Set as Argument	83
2.9.25. General Algebraic Execution Rules	84
2.9.25.1. Execution Rules on Including	84
2.9.25.2. Execution Rules on Excluding	85
2.9.25.3. Execution Rules on Includes	87
2.9.25.4. Execution Rules on Excludes	88
2.9.25.5. Execution Rules on Size	88
2.9.25.6. Execution Rules on IsEmpty	88
2.9.25.7. Execution Rules on NotEmpty	88
2.9.25.8. Execution Rules on Any	88
2.9.25.9. Execution Rules on Forall	89
2.9.25.10. Execution Rules on Exists	89
2.9.25.11. Execution Rules on Iterate	89
2.9.25.12. Execution Rules on Select	90
2.9.25.13. Execution Rules on Reject	90
2.9.25.14. Execution Rules Combining Previous Operators	90
2.9.26. Test Statements	92
2.10. Collection Type Sequence: Operations	93
2.10.1. Basic Properties of the Sequence Type	93
2.10.2. Definition: Strict Equality	93
2.10.3. Constants: mtSequence	93
2.10.4. Definition: Prepend	93
2.10.5. Definition: Including	94
2.10.6. Definition: Excluding	94
2.10.7. Definition: Append	94
2.10.8. Definition: Union	95
2.10.9. Definition: At	95
2.10.10. Definition: First	95
2.10.11. Definition: Last	95
2.10.12. Definition: Iterate	95
2.10.13. Definition: Forall	96
2.10.14. Definition: Exists	96
2.10.15. Definition: Collect	96
2.10.16. Definition: Select	96
2.10.17. Definition: Size	97
2.10.18. Definition: IsEmpty	97
2.10.19. Definition: NotEmpty	97
2.10.20. Definition: Any	97
2.10.21. Definition (future operators)	97
2.10.22. Logical Properties	97
2.10.23. Execution Laws with Invalid or Null as Argument	97
2.10.24. General Algebraic Execution Rules	98
2.10.24.1. Execution Rules on Iterate	98
2.10.25. Test Statements	98
2.11. Miscellaneous Stuff	98
2.11.1. Definition: asBoolean	98
2.11.2. Definition: asInteger	99
2.11.3. Definition: asReal	99
2.11.4. Definition: asPair	99
2.11.5. Definition: asSet	100
2.11.6. Definition: asSequence	100

2.11.7. Definition: asBag	100
2.11.8. Test Statements	100
3. Formalization III: UML/OCL constructs: State Operations and Objects	103
3.1. Introduction: States over Typed Object Universes	103
3.1.1. Fundamental Properties on Objects: Core Referential Equality	103
3.1.1.1. Definition	103
3.1.1.2. Strictness and context passing	103
3.1.2. Logic and Algebraic Layer on Object	103
3.1.2.1. Validity and Definedness Properties	103
3.1.2.2. Symmetry	104
3.1.2.3. Behavior vs StrongEq	104
3.2. Operations on Object	105
3.2.1. Initial States (for testing and code generation)	105
3.2.2. OclAllInstances	105
3.2.2.1. OclAllInstances (@post)	107
3.2.2.2. OclAllInstances (@pre)	109
3.2.2.3. @post or @pre	110
3.2.3. OclIsNew, OclIsDeleted, OclIsMaintained, OclIsAbsent	110
3.2.4. OclIsModifiedOnly	111
3.2.4.1. Definition	111
3.2.4.2. Execution with Invalid or Null or Null Element as Argument	111
3.2.4.3. Context Passing	112
3.2.5. OclSelf	112
3.2.6. Framing Theorem	112
3.2.7. Miscellaneous	113
3.3. Accessors on Object	113
3.3.1. Definition	113
3.3.2. Validity and Definedness Properties	114
4. Example: The Employee Analysis Model	120
4.1. Introduction	120
4.1.1. Outlining the Example	120
4.2. Example Data-Universe and its Infrastructure	120
4.3. Instantiation of the Generic Strict Equality	122
4.4. OclAsType	122
4.4.1. Definition	122
4.4.2. Execution with Invalid or Null as Argument	123
4.5. OclIsTypeOf	123
4.5.1. Definition	123
4.5.2. Execution with Invalid or Null as Argument	125
4.5.3. Up Down Casting	125
4.6. OclIsKindOf	126
4.6.1. Definition	126
4.6.2. Execution with Invalid or Null as Argument	127
4.6.3. Up Down Casting	127
4.7. OclAllInstances	127
4.7.1. OclIsTypeOf	127
4.7.2. OclIsKindOf	128
4.8. The Accessors (any, boss, salary)	129
4.8.1. Definition (of the association Employee-Boss)	129
4.8.2. Context Passing	131
4.8.3. Execution with Invalid or Null as Argument	132

4.8.4. Representation in States	132
4.9. A Little Infra-structure on Example States	132
4.10. OCL Part: Invariant	135
4.11. OCL Part: The Contract of a Recursive Query	137
4.12. OCL Part: The Contract of a User-defined Method	138
B. Bibliography	140
C. The OCL And Featherweight OCL Syntax	143
D. Table of Contents	150