

Part I.

Annex A

1. Introduction

This annex formally defines the semantics of OCL. It will proceed by describing the OCL semantics by a translation into a core language — called FeatherweightOCL— which has in itself a formally described semantics presented in Isabelle/HOL [25]¹. The semantic definitions are in large parts executable, in some parts only provable, namely the essence of Set-constructions. The first goal of its construction is *consistency*, i. e. it should be possible to apply logical rules and/or evaluation rules for OCL in an arbitrary manner always yielding the same result. Moreover, except in pathological cases, this result should be unambiguously defined, i. e. represent a value.

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

<code>Pair{true,invalid}.First() = invalid.First() = invalid</code>

and:

<code>Pair{true,invalid}.First() = true</code>

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.

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

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

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

OCL is an annotation language for UML models, in particular class models allowing for specifying data and operations on them. As such, it is a *typed* object-oriented language. This means that it is — like Java or C++ — based on the concept of a *static type*, that is the type that the type-checker infers from a UML class model and its OCL annotation, as well as a *dynamic type*, that is the type at which an object is dynamically 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 Russells Paradox in OCL typed set-theory. Moreover, object-oriented typing means that types there can be in sub-typing relation; technically speaking, this means that they can be *casted* via `oclIsTypeOf(T)` one to the other, and under particular conditions to be described in detail later, these casts are semantically *lossless*, i. e.

$$(X.\text{oclAsType}(C_j).\text{oclAsType}(C_i) = X) \quad (1.1)$$

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

Here is a feature-list of FeatherweightOCL:

- it specifies key built-in types such as `Boolean`, `Void`, `Integer`, `Real` and `String` as well as generic types such as `Pair(T,T')`, `Sequence(T)` and `Set(T)`.
- it defines the semantics of the operations of these types in *denotational form* — see explanation below —, and thus in an unambiguous (and in Isabelle/HOL executable or animatable) way.
- it develops the *theory* of these definitions, i.e. the collection of lemmas and theorems that can be proven from these definitions.
- all types in FeatherweightOCL contain the elements `null` and `invalid`; since this extends to `Boolean` type, this results in a four-valued logic. Consequently, FeatherweightOCL contains the derivation of the *logic* of OCL.
- collection types may contain `null` (so `Set{null}` is a defined set) but not `invalid` (`Set{invalid}` is just `invalid`).
- Wrt. to the static types, FeatherweightOCL is a strongly typed language in the Hindley-Milner tradition. We assume that a pre-process for full OCL elim-

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.

inates all implicit conversions due to subtyping by introducing explicit casts (e. g., `oclAsType(Class)`).⁴

- FeatherweightOCL types may be arbitrarily nested. For example, the expression `Set{Set{1,2}} = Set{Set{2,1}}` is legal and true.
- All objects types are represented in an object universe⁵. The universe construction also gives semantics to type casts, dynamic type tests, as well as functions such as `oclAllInstances()`, or `oclIsNew()`. The object universe onstruction is conceptually described and demonstrated at an example.
- As part of the OCL logic, FeatherweightOCL develops the theory of equality in UML/OCL. This includes the standard equality, which is a computable strict equality using the object references for comparison, and the not necessarily computable logical equality, which expresses the Leibniz principle that ‘equals may be replaced by equals’ in OCL terms.
- Technically, FeatherweightOCL is a *semantic embedding* into a powerful semantic meta-language and environment, namely Isabelle/HOL [25]. It is a so-called *shallow embedding* in HOL; this means that types in OCL were *injectively* represented by types in Isabelle/HOL. Ill-typed OCL specifications cannot therefore not be represented in FeatherweightOCL and a type in FeatherweightOCL contains exactly the values that are possible in OCL .

Context.

This document stands in a more than fifteen years tradition of giving a formal semantics to the core of UML and its annotation language OCL, starting from Richters [30] and [17, 20, 24], leading to a number of formal, machine-checked versions, most notably HOL-OCL [5, 6, 9] and more recent approaches [14]. All of them have in common the attempt to reconcile the conflicting demands of an industrially used specification language and its various stakeholders, the needs of OMG standardization process and the desire for sufficient logical precision for tool-implementors, in particular from the Formal Methods research community.

To discuss the future directions of the standard, several OCL experts met in November 2013 in Aachen to discuss possible mid-term improvements of OCL, strategies of standardization of OCL within the OMG, and a vision for possible long-term developments of the language [13]. During this meeting, a Request for Proposals (RFP) for OCL 2.5 was finalized and meanwhile proposed. In particular, this RFP requires that the future OCL 2.5 standard document shall be generated from a machine-checked source. This will ensure

- the absence of syntax errors,

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

⁵following the tradition of HOL-OCL [6]

FiXme: *Something like this ? Shorten Paragraph !*

- the consistency of the formal semantics,
- a suite of corner-cases relevant for OCL tool implementors.

Organization of this document.

This document is organized as follows. After a brief background section introducing a running example and basic knowledge on Isabelle/HOL and its formal notations, we present the formal semantics of FeatherweightOCL introducing:

1. A conceptual description of the formal semantics, highlighting the essentials and avoiding the definitions in detail.
2. A detailed formal description. This covers:
 - a) OCL Types and their presentation in Isabelle/HOL,
 - b) OCL Terms, i.e. the semantics of library operators, together with definitions, lemmas, and test cases for the implementor,
 - c) UML/OCL Constructs, i.e. a core of UML class models plus user-defined constructions on them such as class-invariants and operation constructs.
3. Since the latter, i.e. the construction of UML class models, has to be done on the meta-level (so not *inside* HOL, rather on the level of a pre-compiler), we will describe this process with two larger examples, namely formalizations of our running example.

2. Background

2.1. A Running Example for UML/OCL

The Unified Modeling Language (UML) [26, 27] comprises a variety of model types for describing static (e. g., class models, object models) and dynamic (e. g., state-machines, activity graphs) system properties. One of the more prominent model types of the UML is the *class model* (visualized as *class diagram*) for modeling the underlying data model of a system in an object-oriented manner. As a running example, we model a part of a conference management system. Such a system usually supports the conference organizing process, e. g., creating a conference Website, reviewing submissions, registering attendees, organizing the different sessions and tracks, and indexing and producing the resulting proceedings. In this example, we constrain ourselves to the process of organizing conference sessions; Figure 2.1 shows the class model. We model the hierarchy of roles of our system as a hierarchy of classes (e. g., **Hearer**, **Speaker**, or **Chair**) using an *inheritance* relation (also called *generalization*). In particular, *inheritance* establishes a *subtyping* relationship, i. e., every **Speaker** (*subclass*) is also a **Hearer** (*superclass*).

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



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

FiXme: REWRITE
THIS FOR THE
ANNEX A:
SHORTEN!

Relations between classes (called *associations* in UML) can be represented in a class diagram by connecting lines, e.g., **Participant** and **Session** or **Person** and **Role**. Associations may be labeled by a particular constraint called *multiplicity*, e.g., $0..*$ or $0..1$, which means that in a relation between participants and sessions, each **Participant** object is associated to at most one **Session** object, while each **Session** object may be associated to arbitrarily many **Participant** objects. Furthermore, associations may be labeled by projection functions like **person** and **role**; these implicit function definitions allow for OCL-expressions like **self.person**, where **self** is a variable of the class **Role**. The expression **self.person** denotes persons being related to the specific object **self** of type **role**. A particular feature of the UML are *association classes* (**Participant** in our example) which represent a concrete tuple of the relation within a system state as an object; i.e., associations classes allow also for defining attributes and operations for such tuples. In a class diagram, association classes are represented by a dotted line connecting the class with the association. Associations classes can take part in other associations. Moreover, UML supports also *n*-ary associations (not shown in our example).

We refine this data model using the Object Constraint Language (OCL) for specifying additional invariants, preconditions and postconditions of operations. For example, we specify that objects of the class **Person** are uniquely determined by the value of the **name** attribute and that the attribute **name** is not equal to the empty string (denoted by **' '**):

```
context Person
  inv: name <> '' and
       Person::allInstances()->isUnique(p:Person | p.name)
```

Moreover, we specify that every session has exactly one chair by the following invariant (called **onlyOneChair**) of the class **Session**:

```
context Session
  inv onlyOneChair: self.participants->one( p:Participant |
                                             p.role.oclIsTypeOf(Chair))
```

where **p.role.oclIsTypeOf(Chair)** evaluates to true, if **p.role** is of *dynamic type* **Chair**. Besides the usual *static types* (i.e., the types inferred by a static type inference), objects in UML and other object-oriented languages have a second *dynamic* type concept. This is a consequence of a family of *casting functions* (written $o_{[C]}$ for an object *o* into another class type *C*) that allows for converting the static type of objects along the class hierarchy. The dynamic type of an object can be understood as its “initial static type” and is unchanged by casts. We complete our example by describing the behavior of the operation **findRole** as follows:

```
context Session::findRole(person:Person):Role
  pre: self.participates.person->includes(person)
  post: result=self.participants->one(p:Participant |
                                     p.person = person ).role
       and self.participants = self.participants@pre
       and self.name = self.name@pre
```

where in post-conditions, the operator `@pre` allows for accessing the previous state.

In UML, classes can contain attributes of the type of the defining class. Thus, UML can represent (mutually) recursive datatypes. Moreover, OCL introduces also recursively specified operations.

A key idea of defining the semantics of UML and extensions like SecureUML [10] is to translate the diagrammatic UML features into a combination of more elementary features of UML and OCL expressions [19]. For example, associations are usually represented by collection-valued class attributes together with OCL constraints expressing the multiplicity. Thus, having a semantics for a subset of UML and OCL is tantamount for the foundation of the entire method.

2.2. Formal Foundation

2.2.1. Isabelle

Isabelle [25] is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and natural deduction inference rules. Among other logics, Isabelle supports first-order logic, Zermelo-Fraenkel set theory and the instance for Church’s higher-order logic (HOL).

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

$$\llbracket A_1; \dots; A_n \rrbracket \Longrightarrow A_{n+1} \quad \text{or, in mathematical notation,} \quad \frac{A_1 \quad \dots \quad A_n}{A_{n+1}}. \quad (2.1)$$

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

$$\bigwedge x_1, \dots, x_m. \llbracket A_1; \dots; A_n \rrbracket \Longrightarrow A_{n+1}. \quad (2.2)$$

Isabelle supports forward- and backward reasoning on rules. For backward-reasoning, a *proof-state* can be initialized and further transformed into others. For example, a proof of ϕ , using the Isar [33] language, will look as follows in Isabelle:

```

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

(2.3)

This proof script instructs Isabelle to prove ϕ by case distinction followed by a simplification of the resulting proof state. Such a proof state is an implicitly conjoint sequence

of generalized Horn-clauses (called *subgoals*) ϕ_1, \dots, ϕ_n and a *goal* ϕ . Proof states were usually denoted by:

$$\begin{array}{lcl} \text{label :} & \phi & \\ & 1. \phi_1 & \\ & \vdots & \\ & n. \phi_n & \end{array} \quad (2.4)$$

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

2.2.2. Higher-order Logic (HOL)

Higher-order logic (HOL) [1, 15] is a classical logic based on a simple type system. It provides the usual logical connectives like $_ \wedge _, _ \rightarrow _, \neg _$ as well as the object-logical quantifiers $\forall x. P x$ and $\exists x. P x$; in contrast to first-order logic, quantifiers may range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centered around extensional equality $_ = _ :: \alpha \Rightarrow \alpha \Rightarrow \text{bool}$. HOL is more expressive than first-order logic, since, e.g., induction schemes can be expressed inside the logic. Being based on some polymorphically typed λ -calculus, HOL can be viewed as a combination of a programming language like SML or Haskell and a specification language providing powerful logical quantifiers ranging over elementary and function types.

Isabelle/HOL is a logical embedding of HOL into Isabelle. The (original) simple-type system underlying HOL has been extended by Hindley-Milner style polymorphism with type-classes similar to Haskell. While Isabelle/HOL is usually seen as proof assistant, we use it as symbolic computation environment. Implementations on top of Isabelle/HOL can re-use existing powerful deduction mechanisms such as higher-order resolution, tableaux-based reasoners, rewriting procedures, Presburger arithmetic, and via various integration mechanisms, also external provers such as Vampire [29] and the SMT-solver Z3 [18].

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

For instance, the library includes the type constructor $\tau_\perp := \perp \mid _ : \alpha$ that assigns to each type τ a type τ_\perp *disjointly extended* by the exceptional element \perp . The function $\lceil _ : \alpha_\perp \rightarrow \alpha$ is the inverse of $_ : \alpha \rightarrow \alpha_\perp$ (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:¹

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

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

$$\begin{array}{ll}
\text{datatype} & \text{option} = \text{None} \mid \text{Some } \alpha \\
\text{datatype} & \alpha \text{ list} = \text{Nil} \mid \text{Cons } a \ l
\end{array} \quad (2.6)$$

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

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

respectively

$$\text{case } x \text{ of } [] \Rightarrow F \mid \text{Cons } a \ r \Rightarrow G \ a \ r. \quad (2.8)$$

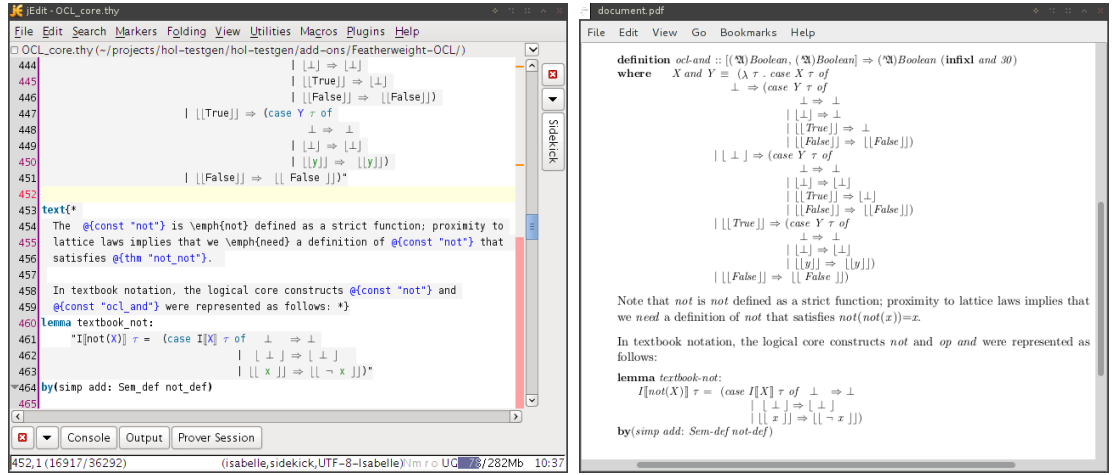
From the internal definitions (not shown here) several properties were automatically derived. We show only the case for lists:

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

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

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

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



(a) The Isabelle jEdit environment.

(b) The generated formal document.

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

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

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

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

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

FiXme: Here ? Or in chap 2 ?

Isabelle, as a framework for building formal tools [32], provides the means for generating *formal documents*. With formal documents (such as the one you are currently reading)

we refer to documents that are machine-generated and ensure certain formal guarantees. In particular, all formal content (e. g., definitions, formulae, types) are checked for consistency during the document generation.

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

Figure 2.2 illustrates this approach: Figure 2.2a shows the jEdit-based development environment of Isabelle with an excerpt of one of the core theories of FeatherweightOCL. Figure 2.2b shows the generated PDF document where all antiquotations are replaced. Moreover, the document generation tools allows for defining syntactic sugar as well as skipping technical details of the formalization.

Thus, applying the FeatherweightOCL approach to writing an updated Annex A that provides a formal semantics of the most fundamental concepts of OCL would ensure

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

Overall, this would contribute to one of the main goals of the OCL 2.5 RFP, as discussed at the OCL meeting in Aachen [13].

3. The Essence of UML-OCL Semantics

3.1. The Theory Organization

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

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

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

3.1.1. Denotational Semantics of Types

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

FiXme: *should we use explicit definitions ?*

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

Types were directly represented in FeatherweightOCL by types in HOL; consequently, any FeatherweightOCL type must provide elements for a bottom element (also denoted \perp) and a null element; this is enforced in Isabelle by a type-class `null` that contains two distinguishable elements `bot` and `null` (see Section 4.1.2 for the details of the construction).

Moreover, the representation mapping from OCL types to FeatherweightOCL is one-to-one (i. e. injective), and the corresponding FeatherweightOCL types were constructed

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

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

$$\text{type_synonym} \quad \text{Boolean}_{\text{base}} := \text{bool}_{\perp\perp} \quad (3.1)$$

As a consequence of this definition of the type, we have the elements $\perp, \perp_{\perp}, \perp_{\perp}\text{true}_{\perp\perp}, \perp_{\perp}\text{false}_{\perp\perp}$ 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 Section 4.1.2).

Fixme: *why does
backslash null not
work here ?*

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

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

$$\text{type_synonym} \quad V_{??}(\alpha) := \text{state}(??) \times \text{state}(??) \rightarrow \alpha :: \text{null} . \quad (3.2)$$

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

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

the other cases are analogous. In the subsequent sections, we will drop the index $??$ since it is constant in all formulas and expressions except for operations related to the object universe construction in $??$

The rules of the logical layer (there are no algebraic rules related to the semantics of types), and more details can be found in Section 4.1.2.

3.1.2. Denotational Semantics of Constants and Operations

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

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

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

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

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

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

we can therefore write:

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

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.

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

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

FiXme: we must uniformize the list - vs. `lfloor` notation. Either the one or the other.

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

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:

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

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.

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

$$\begin{aligned} I[S \text{ ->including}(y)]\tau = & \text{ if } I[\delta \ S]\tau = I[\text{true}]\tau \wedge I[v \ y]\tau = I[\text{true}]\tau \\ & \text{ then } \text{Abs_Set}_{\text{base}} \llbracket \text{Rep_Set}_{\text{base}} I[S]\tau^\top \cup \{I[y]\tau\} \rrbracket \\ & \text{ else } \perp \end{aligned}$$

Here, the operator $_ \cup _$ stems from the HOL set theory, together with the set inclusion $\{-\}$. The operator `Abs_Setbase` is the constructor for the FeatherweightOCL Set type, whereas `Rep_Setbase` is its destructor (see Section 4.1.2 for details). There is even one more variant of a strict basic OCL operation: the referential equality `=`. Since the comparison with must be possible and since the referential equality should be symmetric, should be allowed for *both* arguments and the expression:

$$\text{null} = \text{null} \tag{3.3}$$

should be valid and true. The details were discussed in the next session.

3.1.3. 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:

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

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,”¹ and is at the heart of the OCL logic,
3. The symbol $_ \doteq _$ is used for the strict referential equality, i.e. the equality the mandatory part of the OCL standard refers to by the $_ = _$ symbol.

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

$$I[X \triangleq Y]\tau \equiv \sqcup I[X]\tau = I[Y]\tau_{\sqcup}$$

It enjoys nearly the laws of a congruence:

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

where the predicate `cp` stands for *context-passing*, a property that is true for all pure OCL expressions (but not arbitrary mixtures of OCL and HOL) in FeatherweightOCL

¹Strong logical equality is also referred as “Leibniz”-equality.

. The necessary side-calculus for establishing cp can be fully automated; the reader interested in the details is referred to Section 5.1.3.

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

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

The logical layer of the FeatherweightOCL rules gives also a means to convert an OCL formula living in its four-valued world into a representation that is classically two-valued and can be processed by standard SMT solvers such as CVC3 [2] or Z3 [18]. δ -closure rules for all logical connectives have the following format, e. g.:

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

Together with the already mentioned general case-distinction

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

which is possible for any OCL type, a case distinction on the variables in a formula can be performed; due to strictness rules, formulae containing somewhere a variable x that is known to be **invalid** or **null** reduce 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.

3.1.4. Algebraic Layer

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

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

$$\begin{aligned}
v \text{ invalid} &= \text{false} & v \text{ null} &= \text{true} \\
v \text{ true} &= \text{true} & v \text{ false} &= \text{true}
\end{aligned}$$

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

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

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

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

The above equations explain the behavior for the most-important non-strict operations. The clarification of the exceptional behaviors is of key-importance for a semantic definition of the standard and the major deviation point from HOLOCL [5, 7], to FeatherweightOCL as presented here. Expressed in algebraic equations, “strictness-principles” boil down to:

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

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

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

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

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

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

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

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

3.2. Object-oriented Datatype Theories

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

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

FiXme: *TODO*

1. C is a set of class names (written as $\{C_1, \dots, C_n\}$). To each class name a type of data in OCL is associated. Moreover, class names declare two projector functions to the set of all objects in a state: $C_i.\text{allInstances}()$ and $C_i.\text{allInstances@pre}()$,
2. $_ < _$ is an inheritance relation on classes,

3. $Attrib(C_i)$ is a collection of attributes associated to classes C_i . It declares two families of accessors; for each attribute $a \in Attrib(C_i)$ in a class definition C_i (denoted $X.a :: C_i \rightarrow A$ and $X.a@pre :: C_i \rightarrow A$ for $A \in TYPES(C)$),
4. $Assoc(C_i, C_j)$ is a collection of associations.² An association $(n, rn_{from}, rn_{to}) \in Assoc(C_i, C_j)$ between to classes C_i and C_j is a triple consisting of a (unique) association name n , and the rolenames rn_{to} and rn_{from} . To each rolename belong two families of accessors denoted $X.a :: C_i \rightarrow A$ and $X.a@pre :: C_i \rightarrow A$ for $A \in TYPES(C)$,
5. for each pair $C_i < C_j$ ($C_i, C_j < C$), there is a cast operation of type $C_j \rightarrow C_i$ that can change the static type of an object of type C_i : $obj :: C_i.oclAsType(C_j)$,
6. for each class $C_i \in C$, there are two dynamic type tests ($X.oclIsTypeOf(C_i)$ and $X.oclIsKindOf(C_i)$),
7. and last but not least, for each class name $C_i \in C$ there is an instance of the overloaded referential equality (written $- \doteq -$).

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

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

3.2.1. 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:

$$\text{typedef} \quad \alpha \text{ state} := \{\sigma :: \text{oid} \rightarrow \alpha \mid \text{inv}_\sigma(\sigma)\} \quad (3.4)$$

where inv_σ is a to be discussed invariant on states.

The key point is that we need a common type α for the set of all possible *object representations*. Object representations model “a piece of typed memory,” i.e., a kind of record comprising administration information and the information for all attributes of an object; here, the primitive types as well as collections over them are stored directly

²Given the fact that there is at present no consensus on the semantics of n-ary associations, FeatherweightOCL restricts itself to binary associations.

in the object representations, class types and collections over them are represented by oid's (respectively lifted collections over them).

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

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

For the sake of simplicity, the present semantics chose the first option for FeatherweightOCL, while HOL-OCL [6] used an involved construction allowing the latter.

A naïve attempt to construct \mathfrak{A} would look like this: the class type C_i induced by a class will be the type of such an object representation: $C_i := (\text{oid} \times A_{i_1} \times \dots \times A_{i_k})$ 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:

$$\mathfrak{A} := C_1 + \dots + C_n. \quad (3.5)$$

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 losing information, such that casting up and casting down will *not* give the required identity:

$$\begin{aligned} X.\text{oclIsTypeOf}(C_k) \text{ implies } X.\text{oclAsType}(C_i).\text{oclAsType}(C_k) \doteq X & \quad (3.6) \\ \text{whenever } C_k < C_i \text{ and } X \text{ is valid.} & \quad (3.7) \end{aligned}$$

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

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

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

Example instances of this scheme—outlining a compiler—can be found in Chapter 7 and Chapter 8.

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

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

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

We will work out concrete examples for the construction of the object-universes in Chapter 7 and Chapter 8 and the derivation of the respective datatype theories. While an automatization is clearly possible and desirable for concrete applications of FeatherweightOCL, we consider this out of the scope of this paper which has a focus on the semantic construction and its presentation.

3.2.2. 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 FeatherweightOCL . Arguments and results of accessors are based on type-safe object representations and *not* oid’s. This implies the following scheme for an accessor:

- The *evaluation and extraction* phase. If the argument evaluation results in an object representation, the oid is extracted, if not, exceptional cases like `invalid` are reported.
- The *dereferentiation* phase. The oid is interpreted in the pre- or post-state, the resulting object is casted to the expected format. The exceptional case of nonexistence in this state must be treated.
- The *selection* phase. The corresponding attribute is extracted from the object representation.
- The *re-construction* phase. The resulting value has to be embedded in the adequate HOL type. If an attribute has the type of an object (not value), it is represented by an optional (set of) oid, which must be converted via dereferentiation in one of the states to produce an object representation again. The exceptional case of nonexistence in this state must be treated.

The first phase directly translates into the following formalization:

definition

$$\text{eval_extract } X \ f = (\lambda \tau. \text{ case } X \ \tau \text{ of } \begin{array}{ll} \perp & \Rightarrow \text{invalid } \tau \quad \text{exception} \\ | \ \lfloor \perp \rfloor & \Rightarrow \text{invalid } \tau \quad \text{deref. null} \\ | \ \lfloor \text{obj} \rfloor & \Rightarrow f \ (\text{oid_of } \text{obj}) \ \tau \end{array}) \quad (3.8)$$

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

$$\text{definition } \text{deref_oid}_C \ fst_snd \ f \ oid = (\lambda \tau. \text{ case } (\text{heap } (fst_snd \ \tau)) \ oid \text{ of } \begin{array}{ll} \lfloor \text{in}_C \text{obj} \rfloor & \Rightarrow f \ \text{obj} \ \tau \\ | \text{--} & \Rightarrow \text{invalid } \tau \end{array}) \quad (3.9)$$

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:

$$\text{definition } \text{select}_a \ f = (\lambda \quad \begin{array}{ll} \text{mk}_C \ oid \ \dots \perp \dots & C_{\text{ext}} \Rightarrow \text{null} \\ | \text{mk}_C \ oid \ \dots \lfloor a \rfloor \dots & C_{\text{ext}} \Rightarrow f \ (\lambda \ x \ \dots \lfloor x \rfloor) \ a \end{array}) \quad (3.10)$$

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:

$$\begin{array}{lll} \text{definition} & \text{in_pre_state} & = \text{fst} \quad \text{first component} \\ \text{definition} & \text{in_post_state} & = \text{snd} \quad \text{second component} \\ \text{definition} & \text{reconst_basetype} & = \text{id} \quad \text{identity function} \end{array} \quad (3.11)$$

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

$$\begin{array}{ll} \text{definition} & _.\text{getBase} \quad :: C \Rightarrow A_{base} \\ \text{where} & X.\text{getBase} = \text{eval_extract } X \ (\text{deref_oid}_C \ \text{in_post_state} \\ & \quad (\text{select}_{\text{getBase}} \ \text{reconst_basetype})) \end{array} \quad (3.12)$$

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

$$\begin{array}{ll} \text{definition} & _.\text{getObject} \quad :: C \Rightarrow A_{object} \\ \text{where} & X.\text{getObject} = \text{eval_extract } X \ (\text{deref_oid}_C \ \text{in_post_state} \\ & \quad (\text{select}_{\text{getObject}} \ (\text{deref_oid}_C \ \text{in_post_state}))) \end{array} \quad (3.13)$$

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

Examples for the construction of accessors via associations can be found in Section 7.8, the construction of accessors via attributes in Section 8.8. The construction of casts and type tests `->oclIsTypeOf()` and `->oclIsKindOf()` is similarly.

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

Single-Valued Attributes

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

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

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

Collection-Valued Attributes

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

We propose to resolve this dilemma by regarding multiplicities as optional. This point of view complies with the UML standard, that does not require lower and upper bounds to be defined for multiplicities.³ In case a multiplicity is specified for an attribute, i. e., a

³We are however aware that a well-formedness rule of the UML standard does define a default bound of one in case a lower or upper bound is not specified.

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

The Precise Meaning of Multiplicity Constraints

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

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

If the upper bound n is infinite, the second invariant is omitted. For the definition of these invariants we are making use of the conversion of single values to sets described in Section 3.2.2. If $n \leq 1$, the attribute `a` evaluates to a single value, which is then converted to a `Set` on which the `size` operation is called.

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

3.2.3. Logic Properties of Class-Models

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

```
\<tau> \<Turnstile> X .oclAsType(C_i) \<triangleq> X
\<tau> \<Turnstile> invalid .oclAsType(C_i) \<triangleq> invalid
```

```

\<tau> \<Turnstile> null .oclAsType(C_i) \<triangleq> null
\<tau> \<Turnstile> ((X::C_i) .oclAsType(C_j) .oclAsType(C_i) \<triangleq> X)
\<tau> \<Turnstile> X .oclAsType(C_j) .oclAsType(C_i) \<triangleq> X
\<tau> \<Turnstile> \<upsilon> (X :: C_i) \<Longrightarrow> \<tau> \<Turnstile> (X .oclAsType(C_j) .oclAsType(C_i) \<triangleq> X)
\<tau> \<Turnstile> (X::OclAny) .oclAsType(OclAny) \<triangleq> X
\<tau> \<Turnstile> \<upsilon> (X :: C_i) \<Longrightarrow> \<tau> \<Turnstile> (X .oclAsType(C_j) .oclAsType(C_i) \<triangleq> X)
\<tau> \<Turnstile> \<delta> X \<Longrightarrow> \<tau> \<Turnstile> X .oclAsType(C_j) .oclAsType(C_i) \<triangleq> X
\<tau> \<Turnstile> \<upsilon> X \<Longrightarrow> \<tau> \<Turnstile> X .oclIsTypeOf(C_j) \<triangleq> X
\<tau> \<Turnstile> X .oclIsTypeOf(C_j) \<Longrightarrow> \<tau> \<Turnstile> \<delta> X \<Longrightarrow> \<tau> \<Turnstile> X .oclIsTypeOf(C_j) \<triangleq> X
\<tau> \<Turnstile> invalid .oclIsTypeOf(C_i) \<triangleq> invalid
\<tau> \<Turnstile> null .oclIsTypeOf(C_i) \<triangleq> true
\<tau> \<Turnstile> (Person .allInstances()->forall(X|X .oclIsTypeOf(C_z)))
\<tau> \<Turnstile> (Person .allInstances@pre()->forall(X|X .oclIsTypeOf(C_z)))
\<tau> \<Turnstile> (Person .allInstances()->forall(X|X .oclIsKindOf(C_i)))
\<tau> \<Turnstile> (Person .allInstances@pre()->forall(X|X .oclIsKindOf(C_i)))
\<tau> \<Turnstile> (X::C_i).oclIsTypeOf(C_j) \<Longrightarrow> \<tau> \<Turnstile> (X::C_i).oclIsTypeOf(C_j) \<triangleq> X
(\<tau> \<Turnstile> (X::C_j) \<doteq> X) = (\<tau> \<Turnstile> if \<upsilon> X then \<tau> \<Turnstile> (X::C_j) \<doteq> Y \<Longrightarrow> \<tau> \<Turnstile> Y \<doteq> X \<Longrightarrow> \<tau> \<Turnstile> (X::C_j) \<doteq> Y \<Longrightarrow> \<tau> \<Turnstile> Y \<doteq> Z \<Longrightarrow> \<tau> \<Turnstile> X \<doteq> Z)

```

3.2.4. 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:

```

invalid .oclIsTypeOf(C_i) = invalid
null .oclIsTypeOf(C_i) = true
invalid .oclIsKindOf(C_i) = invalid
null .oclIsKindOf(C_i) = true
(X::C_i) .oclAsType(C_i) = X
invalid .oclAsType(C_i) = invalid
null .oclAsType(C_i) = null
(X::C_i) \<doteq> X = if \<upsilon> X then true else invalid endif
((X::C_i) .oclAsType(C_j) .oclAsType(C_i) = X)

```

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

$$\text{invalid}.a = \text{invalid} \quad (3.14)$$

$$\text{null}.a = \text{invalid} \quad (3.15)$$

$$\text{invalid}.a@pre = \text{invalid} \quad (3.16)$$

$$\text{null}.a@pre = \text{invalid} \quad (3.17)$$

$$\text{invalid}.r = \text{invalid} \quad (3.18)$$

$$\text{null}.r = \text{invalid} \quad (3.19)$$

$$\text{invalid}.r@pre = \text{invalid} \quad (3.20)$$

$$\text{null.r@pre} = \text{invalid} \quad (3.21)$$

$$(3.22)$$

3.2.5. 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 [21]). We define

$$(S : \text{Set}(\text{OclAny})) \rightarrow \text{oclIsModifiedOnly}() : \text{Boolean}$$

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:

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

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

3.3. Data Invariants

Since the present OCL semantics uses one interpretation function ⁴, 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. $(id, rn_{from}, rn_{to}) \in \text{Assoc}(C_i, C_j)$) were replaced by their counterparts `_.rn@pre`. For example, $(self.boss = null)_{pre}$ is transformed to $self.boss@pre = null$.

⁴This has been handled differently in previous versions of the Annex A.

- The operation `_ .allInstances()` is also substituted by its `@pre` counterpart.

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

$$\begin{aligned}
I[\text{context } c : C_i \text{ inv } n : \phi(c)]\tau &\equiv \\
\tau &\models (C_i . \text{allInstances}() \rightarrow \text{forall}(x | \phi(x))) \wedge \\
\tau &\models (C_i . \text{allInstances}() \rightarrow \text{forall}(x | \phi(x)))_{\text{pre}}
\end{aligned} \tag{3.23}$$

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

3.4. Operation Contracts

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

$$\begin{aligned}
I[\text{context } C :: op(a_1, \dots, a_n) : T] \\
\text{pre } \phi(\text{self}, a_1, \dots, a_n) \\
\text{post } \psi(\text{self}, a_1, \dots, a_n, \text{result}) &\equiv \\
\lambda s, x_1, \dots, x_n, \tau. & \\
\text{if } \tau \models \partial s \wedge \tau \models v x_1 \wedge \dots \wedge \tau \models v x_n & \\
\text{then SOME } \text{result}. \quad \tau \models \phi(s, x_1, \dots, x_n)_{\text{pre}} & \\
\quad \wedge \tau \models \psi(s, x_1, \dots, x_n, \text{result}) & \\
\text{else } \perp &
\end{aligned} \tag{3.24}$$

where *SOME* x . $P(x)$ is the Hilbert-Choice Operator that chooses an arbitrary element satisfying P ; if such an element does not exist, it chooses an arbitrary one⁵. Thus, using the Hilbert-Choice Operator, a contract can be associated to a function definition:

$$f_{op} \equiv I[\text{context } C :: op(a_1, \dots, a_n) : T \dots] \tag{3.25}$$

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 3.2.5), 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 constructs left to the user (it can be shown, for example, by a proof of termination, i. e. by showing that all recursive calls were applied to argument vectors that are smaller wrt. to a well-founded ordering). Under this condition, an f_{op} resulting from recursive query operations can be used safely inside pre- and post-conditions of other contracts.

⁵In HOL, the Hilbert-Choice operator is a first-class element of the logical language.

FiXme: Should we add in our notion of Class-Model also the Operations ?

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

$$\frac{\begin{array}{c} [\tau \models \psi \text{ self } a_1 \dots a_n \text{ res}]_{res} \\ \vdots \\ \tau \models E(res) \end{array}}{\tau \models E(f_{op} \text{ self } a_1 \dots a_n)} \quad (3.26)$$

under the conditions:

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

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

$$\frac{\tau \models \phi \text{ self } a_1 \dots a_n}{(\tau \models E(f_{op} \text{ self } a_1 \dots a_n)) = (\tau \models E(BODY \text{ self } a_1 \dots a_n))} \quad (3.27)$$

where

- E must be an OCL term.
- self must be defined, and the arguments valid in τ :
 $\tau \models \partial \text{ self} \wedge \tau \models v x_1 \wedge \dots \wedge \tau \models v x_n$
- the postcondition $\psi \text{ self } a_1 \dots a_n \text{ result}$ must be decomposable into a $\psi' \text{ self } a_1 \dots a_n$ and $\text{result} \triangleq BODY \text{ self } a_1 \dots a_n$.

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

FixMe: *correct?*

Part II.

Formal Semantics of UML-OCL 2.5

4. Formalization I: OCL Types and Core Definitions

```
theory    UML-Types
imports  Transcendental
keywords Assert :: thy-decl
         and Assert-local :: thy-decl
begin
```

4.1. Preliminaries

4.1.1. Notations for the Option Type

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

```
no-notation ceiling ([ $\lceil$ - $\rceil$ )
no-notation floor  ( $\lfloor$ - $\rfloor$ )

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

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

```
fun drop :: 'α option ⇒ 'α ( $\lceil$ (-) $\rceil$ )
where drop-lift[simp]:  $\lceil$  $\lfloor$  $v$  $\rfloor$  $\rceil$  =  $v$ 
```

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

```
definition Sem :: 'a ⇒ 'a ( $I$  $\llbracket$ - $\rrbracket$ )
where  $I$  $\llbracket$  $x$  $\rrbracket$   $\equiv$   $x$ 
```

4.1.2. Common Infrastructure for all OCL Types

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

This uniform interface consists in a type class requiring the existence of a bot and a null element. The construction proceeds by abstracting the null (defined by $\lfloor \perp \rfloor$ on *'a option option*) to a *null* element, which may have an arbitrary semantic structure, and an undefinedness element \perp to an abstract undefinedness element *bot* (also written \perp whenever no confusion arises). As a consequence, it is necessary to redefine the notions of invalid, defined, valuation etc. on top of this interface.

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

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

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

4.1.3. Accommodation of Basic Types to the Abstract Interface

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

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

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

end

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

  instance  $\langle proof \rangle$ 
end
```

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

  instance  $\langle proof \rangle$ 
end
```

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

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

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

For the moment, we formalize the most common notions of objects, in particular the 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 \mathfrak{U} .

```

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

```

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

```

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

```

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

FiXme: *Get
Appropriate
Reference!*

```

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 <proof>
end

```

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

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

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

```

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

```

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

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

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

definition *invalid* :: (' \mathcal{A} , ' α ::*bot*) *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*: $I\llbracket \text{invalid} \rrbracket \tau = \text{bot}$
<proof>

Note that the definition :

definition *null* :: "(' \mathcal{A} , ' α ::*null*) *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*: $I\llbracket \text{null} :: (' \mathcal{A} , ' α ::*null*) *val* \rrbracket \tau = (\text{null} :: (' α ::*null*))$

<proof>

4.2. Basic OCL Value Types

The semantic domain of the (basic) boolean type is now defined as the Standard: the space of valuation to *bool option option*, i.e. the Boolean base type:

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

Because of the previous class definitions, Isabelle type-inference establishes that ' \mathcal{A} *Boolean* lives actually both in the type class *UML-Types.bot-class.bot* and *null*; this type is sufficiently rich to contain at least these two elements. Analogously we build:

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

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

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

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

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

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

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

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

4.3. Some OCL Collection Types

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

4.3.1. The Construction of the Pair Type (Tuples)

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

```
typedef  $(\alpha, \beta)\ Pair_{base} = \{X::(\alpha::null \times \beta::null)\ option\ option. \\ X = bot \vee X = null \vee (fst[[X]] \neq bot \wedge snd[[X]] \neq bot)\} \\ \langle proof \rangle$ 
```

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

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

```
begin
```

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

```
  instance  $\langle proof \rangle$ 
```

```
end
```

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

```
begin
```

```
  definition  $null\text{-}Pair_{base}\text{-}def: (null::(\alpha::null, \beta::null)\ Pair_{base}) \equiv Abs\text{-}Pair_{base}\ [None]$ 
```

```
  instance  $\langle proof \rangle$ 
```

```
end
```

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

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


4.3.2. The Construction of the Set Type

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

```
typedef  $'\alpha \text{ Set}_{base} = \{X :: ('a :: \text{null}) \text{ set option option. } X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$ 
       $\langle \text{proof} \rangle$ 
```

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

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

```
  instance  $\langle \text{proof} \rangle$ 
end
```

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

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

```
  instance  $\langle \text{proof} \rangle$ 
end
```

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

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

4.3.3. The Construction of the Sequence Type

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

```
typedef  $'\alpha \text{ Sequence}_{base} = \{X :: ('a :: \text{null}) \text{ list option option. } X = \text{bot} \vee X = \text{null} \vee (\forall x \in \text{set } \llbracket X \rrbracket. x \neq \text{bot})\}$ 
       $\langle \text{proof} \rangle$ 
```

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

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

```
  instance  $\langle \text{proof} \rangle$ 
end
```

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

begin

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

instance $\langle \text{proof} \rangle$
end

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

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

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

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

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

OCL Type	HOL Type
Boolean	$\mathcal{A} \text{ Boolean}$
Boolean \rightarrow Boolean	$\mathcal{A} \text{ Boolean} \Rightarrow \mathcal{A} \text{ Boolean}$
(Integer,Integer) \rightarrow Boolean	$\mathcal{A} \text{ Integer} \Rightarrow \mathcal{A} \text{ Integer} \Rightarrow \mathcal{A} \text{ Boolean}$
Set(Integer)	$(\mathcal{A}, \text{Integer}_{\text{base}}) \text{Set}$
Set(Integer) \rightarrow Real	$(\mathcal{A}, \text{Integer}_{\text{base}}) \text{Set} \Rightarrow \mathcal{A} \text{ Real}$
Set(Pair(Integer,Boolean))	$(\mathcal{A}, (\text{Integer}_{\text{base}}, \text{Boolean}_{\text{base}}) \text{Pair}_{\text{base}}) \text{Set}$
Set($\langle T \rangle$)	$(\mathcal{A}, \alpha) \text{Set}$

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

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

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

$\langle ML \rangle$

end

5. Formalization II: OCL Terms and Library Operations

```
theory UML-Logic
imports UML-Types
begin
```

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

5.1.1. Basic Constants

```
lemma bot-Boolean-def : (bot::('A)Boolean) = ( $\lambda \tau. \perp$ )
<proof>
```

```
lemma null-Boolean-def : (null::('A)Boolean) = ( $\lambda \tau. \lfloor \perp \rfloor$ )
<proof>
```

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

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

```
lemma bool-split-0:  $X \tau = \text{invalid } \tau \vee X \tau = \text{null } \tau \vee$   

 $X \tau = \text{true } \tau \quad \vee \quad X \tau = \text{false } \tau$ 
<proof>
```

```
lemma [simp]: false (a, b) =  $\lfloor \text{False} \rfloor$ 
<proof>
```

```
lemma [simp]: true (a, b) =  $\lfloor \text{True} \rfloor$ 
<proof>
```

```
lemma textbook-true:  $I \llbracket \text{true} \rrbracket \tau = \lfloor \text{True} \rfloor$ 
<proof>
```

```
lemma textbook-false:  $I \llbracket \text{false} \rrbracket \tau = \lfloor \text{False} \rfloor$ 
<proof>
```

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

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

5.1.2. Validity and Definedness

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

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

lemma *valid1[simp]*: $v \text{ invalid} = \text{false}$
 $\langle \text{proof} \rangle$

lemma *valid2[simp]*: $v \text{ null} = \text{true}$
 $\langle \text{proof} \rangle$

lemma *valid3[simp]*: $v \text{ true} = \text{true}$
 $\langle \text{proof} \rangle$

lemma *valid4[simp]*: $v \text{ false} = \text{true}$
 $\langle \text{proof} \rangle$

lemma *cp-valid*: $(v X) \tau = (v (\lambda \tau . X \tau)) \tau$
 $\langle \text{proof} \rangle$

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

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

lemma *defined1[simp]*: $\delta \text{ invalid} = \text{false}$
 $\langle \text{proof} \rangle$

lemma *defined2[simp]*: $\delta \text{ null} = \text{false}$
 $\langle \text{proof} \rangle$

lemma *defined3[simp]*: $\delta \text{ true} = \text{true}$
 $\langle \text{proof} \rangle$

lemma *defined4*[simp]: $\delta \text{ false} = \text{true}$
 $\langle \text{proof} \rangle$

lemma *defined5*[simp]: $\delta \delta X = \text{true}$
 $\langle \text{proof} \rangle$

lemma *defined6*[simp]: $\delta v X = \text{true}$
 $\langle \text{proof} \rangle$

lemma *valid5*[simp]: $v v X = \text{true}$
 $\langle \text{proof} \rangle$

lemma *valid6*[simp]: $v \delta X = \text{true}$
 $\langle \text{proof} \rangle$

lemma *cp-defined*: $(\delta X)\tau = (\delta (\lambda \cdot X \tau)) \tau$
 $\langle \text{proof} \rangle$

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

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

lemma *textbook-valid*: $I[v(X)] \tau = (\text{if } I[X] \tau = I[\text{bot}] \tau$
 $\text{then } I[\text{false}] \tau$
 $\text{else } I[\text{true}] \tau)$
 $\langle \text{proof} \rangle$

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

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

Table 5.2.: Basic predicate definitions of the logic.

5.1.3. The Equalities of OCL

The OCL contains a particular version of equality, written in Standard documents $_ = _$ and $_ <> _$ for its negation, which is referred as *weak referential equality* hereafter and

Name	Theorem
<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>	$\delta v X = \text{true}$

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

for which we use the symbol \doteq throughout the formal part of this document. Its semantics is motivated by the desire of fast execution, and similarity to languages like Java and C, but does not satisfy the needs of logical reasoning over OCL expressions and specifications. We therefore introduce a second equality, referred as *strong equality* or *logical equality* and written \triangleq which is not present in the current standard but was discussed in prior texts on OCL like the Amsterdam Manifesto [17] and was identified as desirable extension of OCL in the Aachen Meeting [13] in the future 2.5 OCL Standard. The purpose of strong equality is to define and reason over OCL. It is therefore a natural task in Featherweight OCL to formally investigate the somewhat quite complex relationship between these two.

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

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

$a.\text{boss} \doteq b.\text{boss@pre}$ **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:

$$s = t \implies P(s) = P(t) \quad (5.1)$$

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

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

Definition

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

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

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

From this follow already elementary properties like:

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

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

Fundamental Predicates on Strong Equality

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

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

lemma *StrongEq-sym*: $(X \triangleq Y) = (Y \triangleq X)$
 $\langle \text{proof} \rangle$

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

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

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

shows $(X \triangleq Z) = true$

$\langle proof \rangle$

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 $eq: (X \triangleq Y)\tau = true \tau$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *valid7*[simp]: $v (X \triangleq Y) = true$

$\langle proof \rangle$

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

$\langle proof \rangle$

5.1.4. Logical Connectives and their Universal Properties

It is a design goal to give OCL a semantics that is as closely as possible to a “logical system” in a known sense; a specification logic where the logical connectives can not be understood other 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* :: $(\mathfrak{A})Boolean \Rightarrow (\mathfrak{A})Boolean$ (*not*)

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

lemma *cp-OclNot*: $(not\ X)\tau = (not\ (\lambda\ -. X\ \tau))\ \tau$
 $\langle proof \rangle$

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

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

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

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

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

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

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

$$\begin{array}{lcl}
| \llbracket \text{False} \rrbracket & \Rightarrow & \llbracket \text{False} \rrbracket \\
| \perp & \Rightarrow & (\text{case } Y \text{ } \tau \text{ of} \\
& & \llbracket \text{False} \rrbracket \Rightarrow \llbracket \text{False} \rrbracket \\
& & | - \Rightarrow \perp) \\
| \llbracket \perp \rrbracket & \Rightarrow & (\text{case } Y \text{ } \tau \text{ of} \\
& & \llbracket \text{False} \rrbracket \Rightarrow \llbracket \text{False} \rrbracket \\
& & | \perp \Rightarrow \perp \\
& & | - \Rightarrow \llbracket \perp \rrbracket) \\
| \llbracket \text{True} \rrbracket & \Rightarrow & Y \text{ } \tau)
\end{array}$$

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

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

lemma *textbook-OclNot*:

$$I[\text{not}(X)] \tau = \begin{array}{l} \text{(case } I[X] \tau \text{ of } \perp \Rightarrow \perp \\ \quad | [\perp] \Rightarrow [\perp] \\ \quad | [[x]] \Rightarrow [[\neg x]]) \end{array}$$

$\langle proof \rangle$

lemma *textbook-OclAnd*:

$$\begin{aligned}
I\llbracket X \text{ and } Y \rrbracket \tau &= (\text{case } I\llbracket X \rrbracket \tau \text{ of} \\
&\quad \perp \Rightarrow (\text{case } I\llbracket Y \rrbracket \tau \text{ of} \\
&\quad \quad \perp \Rightarrow \perp \\
&\quad \quad | \llbracket \perp \rrbracket \Rightarrow \perp \\
&\quad \quad | \llbracket \text{True} \rrbracket \Rightarrow \perp \\
&\quad \quad | \llbracket \text{False} \rrbracket \Rightarrow \llbracket \text{False} \rrbracket)) \\
&| \llbracket \perp \rrbracket \Rightarrow (\text{case } I\llbracket Y \rrbracket \tau \text{ of} \\
&\quad \perp \Rightarrow \perp \\
&\quad | \llbracket \perp \rrbracket \Rightarrow \llbracket \perp \rrbracket \\
&\quad | \llbracket \text{True} \rrbracket \Rightarrow \llbracket \perp \rrbracket \\
&\quad | \llbracket \text{False} \rrbracket \Rightarrow \llbracket \text{False} \rrbracket)) \\
&| \llbracket \text{True} \rrbracket \Rightarrow (\text{case } I\llbracket Y \rrbracket \tau \text{ of} \\
&\quad \perp \Rightarrow \perp \\
&\quad | \llbracket \perp \rrbracket \Rightarrow \llbracket \perp \rrbracket \\
&\quad | \llbracket y \rrbracket \Rightarrow \llbracket y \rrbracket) \\
&| \llbracket \text{False} \rrbracket \Rightarrow \llbracket \text{False} \rrbracket)
\end{aligned}$$

$\langle proof \rangle$

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

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

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

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

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

lemma *OclAnd1[simp]*: $(\text{invalid and true}) = \text{invalid}$
 $\langle proof \rangle$

lemma *OclAnd2[simp]*: $(\text{invalid and false}) = \text{false}$
 $\langle proof \rangle$

lemma *OclAnd3[simp]*: $(\text{invalid and null}) = \text{invalid}$
 $\langle proof \rangle$

lemma *OclAnd4[simp]*: $(\text{invalid and invalid}) = \text{invalid}$
 $\langle proof \rangle$

lemma *OclAnd5[simp]*: $(\text{null and true}) = \text{null}$

$\langle \text{proof} \rangle$
lemma *OclAnd6[simp]*: $(\text{null and false}) = \text{false}$
 $\langle \text{proof} \rangle$
lemma *OclAnd7[simp]*: $(\text{null and null}) = \text{null}$
 $\langle \text{proof} \rangle$
lemma *OclAnd8[simp]*: $(\text{null and invalid}) = \text{invalid}$
 $\langle \text{proof} \rangle$

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

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

lemma *OclAnd-idem[simp]*: $(X \text{ and } X) = X$
 $\langle \text{proof} \rangle$

lemma *OclAnd-commute*: $(X \text{ and } Y) = (Y \text{ and } X)$
 $\langle \text{proof} \rangle$

lemma *OclAnd-false1[simp]*: $(\text{false and } X) = \text{false}$
 $\langle \text{proof} \rangle$

lemma *OclAnd-false2[simp]*: $(X \text{ and false}) = \text{false}$
 $\langle \text{proof} \rangle$

lemma *OclAnd-true1[simp]*: $(\text{true and } X) = X$
 $\langle \text{proof} \rangle$

lemma *OclAnd-true2[simp]*: $(X \text{ and true}) = X$
 $\langle \text{proof} \rangle$

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

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

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

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

lemma *OclAnd-assoc*: $(X \text{ and } (Y \text{ and } Z)) = (X \text{ and } Y \text{ and } Z)$
 $\langle \text{proof} \rangle$

lemma *OclOr1[simp]*: $(\text{invalid or true}) = \text{true}$
 $\langle \text{proof} \rangle$

lemma *OclOr2[simp]*: $(\text{invalid or false}) = \text{invalid}$
 $\langle \text{proof} \rangle$

lemma *OclOr3[simp]*: $(\text{invalid or null}) = \text{invalid}$
 $\langle \text{proof} \rangle$

lemma *OclOr4[simp]*: $(\text{invalid or invalid}) = \text{invalid}$
 $\langle \text{proof} \rangle$

lemma *OclOr5[simp]*: $(\text{null or true}) = \text{true}$
 $\langle \text{proof} \rangle$

lemma *OclOr6[simp]*: $(\text{null or false}) = \text{null}$
 $\langle \text{proof} \rangle$

lemma *OclOr7[simp]*: $(\text{null or null}) = \text{null}$
 $\langle \text{proof} \rangle$

lemma *OclOr8[simp]*: $(\text{null or invalid}) = \text{invalid}$
 $\langle \text{proof} \rangle$

lemma *OclOr-idem[simp]*: $(X \text{ or } X) = X$
 $\langle \text{proof} \rangle$

lemma *OclOr-commute*: $(X \text{ or } Y) = (Y \text{ or } X)$
 $\langle \text{proof} \rangle$

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

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

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

lemma *OclOr-true2*: $(Y \text{ or true}) = \text{true}$
 $\langle \text{proof} \rangle$

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

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

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

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

lemma *OclOr-assoc*: $(X \text{ or } (Y \text{ or } Z)) = (X \text{ or } Y \text{ or } Z)$
 $\langle \text{proof} \rangle$

lemma *OclImplies-true*: $(X \text{ implies } \text{true}) = \text{true}$
 $\langle \text{proof} \rangle$

lemma *deMorgan1*: $\text{not}(X \text{ and } Y) = ((\text{not } X) \text{ or } (\text{not } Y))$
 $\langle \text{proof} \rangle$

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

5.1.5. A Standard Logical Calculus for OCL

definition *OclValid* :: $[(\mathfrak{A})st, (\mathfrak{A})\text{Boolean}] \Rightarrow \text{bool} \ ((1(-)/ \models (-)) \ 50)$

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

Global vs. Local Judgements

lemma *transform1*: $P = \text{true} \implies \tau \models P$
 $\langle \text{proof} \rangle$

lemma *transform1-rev*: $\forall \tau. \tau \models P \implies P = \text{true}$
 $\langle \text{proof} \rangle$

lemma *transform2*: $(P = Q) \implies ((\tau \models P) = (\tau \models Q))$
 $\langle \text{proof} \rangle$

lemma *transform2-rev*: $\forall \tau. (\tau \models \delta \ P) \wedge (\tau \models \delta \ Q) \wedge (\tau \models P) = (\tau \models Q) \implies P = Q$
 $\langle \text{proof} \rangle$

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

lemma
assumes $H : P = \text{true} \implies Q = \text{true}$
shows $\tau \models P \implies \tau \models Q$
 $\langle \text{proof} \rangle$

Local Validity and Meta-logic

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

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

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

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

lemma *bool-split*[simp]:
 $(\tau \models (x \triangleq \text{invalid})) \vee (\tau \models (x \triangleq \text{null})) \vee (\tau \models (x \triangleq \text{true})) \vee (\tau \models (x \triangleq \text{false}))$
 $\langle \text{proof} \rangle$

lemma *defined-split*:
 $(\tau \models \delta x) = ((\neg(\tau \models (x \triangleq \text{invalid}))) \wedge (\neg(\tau \models (x \triangleq \text{null}))))$
 $\langle \text{proof} \rangle$

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

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

lemma *foundation5*:
 $\tau \models (P \text{ and } Q) \implies (\tau \models P) \wedge (\tau \models Q)$
 $\langle \text{proof} \rangle$

lemma *foundation6*:
 $\tau \models P \implies \tau \models \delta P$
 $\langle \text{proof} \rangle$

lemma *foundation7*[simp]:
 $(\tau \models \text{not } (\delta x)) = (\neg(\tau \models \delta x))$
 $\langle \text{proof} \rangle$

lemma *foundation7'*[simp]:
 $(\tau \models \text{not } (v x)) = (\neg(\tau \models v x))$
 $\langle \text{proof} \rangle$

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

lemma *foundation8*:

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

<proof>

lemma *foundation9*:

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

<proof>

lemma *foundation9'*:

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

<proof>

lemma *foundation9''*:

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

<proof>

lemma *foundation10*:

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

<proof>

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

<proof>

lemma *foundation11*:

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

<proof>

lemma *foundation12*:

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

$\langle proof \rangle$

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

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

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

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

lemma *foundation20*: $\tau \models (\delta X) \implies \tau \models v X$
 $\langle proof \rangle$

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

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

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

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

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

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

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

shows R
 $\langle proof \rangle$

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

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

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

lemma *defined-and-I* : $\tau \models \delta(x) \implies \tau \models \delta(y) \implies \tau \models \delta(x \text{ and } y)$
 $\langle proof \rangle$

lemma *valid-and-I* : $\tau \models v(x) \implies \tau \models v(y) \implies \tau \models v(x \text{ and } y)$
 $\langle proof \rangle$

lemma *defined-or-I* : $\tau \models \delta(x) \implies \tau \models \delta(y) \implies \tau \models \delta(x \text{ or } y)$
 $\langle proof \rangle$

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

Local Judgements and Strong Equality

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

lemma *StrongEq-L-sym*: $\tau \models (x \triangleq y) \implies \tau \models (y \triangleq x)$
 $\langle proof \rangle$

lemma *StrongEq-L-trans*: $\tau \models (x \triangleq y) \implies \tau \models (y \triangleq z) \implies \tau \models (x \triangleq z)$
 $\langle proof \rangle$

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

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

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

lemma *StrongEq-L-subst1*: $\bigwedge \tau. cp P \implies \tau \models (x \triangleq y) \implies \tau \models (P x \triangleq P y)$
 $\langle proof \rangle$

lemma *StrongEq-L-subst2*:

$\bigwedge \tau. \text{cp } P \implies \tau \models (x \triangleq y) \implies \tau \models (P \ x) \implies \tau \models (P \ y)$
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *StrongEq-L-subst3*:

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

and *eq*: $\tau \models (x \triangleq y)$

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

$\langle \text{proof} \rangle$

lemma *StrongEq-L-subst3-rev*:

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

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

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

$\langle \text{proof} \rangle$

lemma *StrongEq-L-subst4-rev*:

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

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

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

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

$\langle \text{proof} \rangle$

lemma *cpI1*:

$(\forall \ X \ \tau. f \ X \ \tau = f(\lambda_. X \ \tau) \ \tau) \implies \text{cp } P \implies \text{cp}(\lambda X. f \ (P \ X))$

$\langle \text{proof} \rangle$

lemma *cpI2*:

$(\forall \ X \ Y \ \tau. f \ X \ Y \ \tau = f(\lambda_. X \ \tau)(\lambda_. Y \ \tau) \ \tau) \implies$

$\text{cp } P \implies \text{cp } Q \implies \text{cp}(\lambda X. f \ (P \ X) \ (Q \ X))$

$\langle \text{proof} \rangle$

lemma *cpI3*:

$(\forall \ X \ Y \ Z \ \tau. f \ X \ Y \ Z \ \tau = f(\lambda_. X \ \tau)(\lambda_. Y \ \tau)(\lambda_. Z \ \tau) \ \tau) \implies$

$\text{cp } P \implies \text{cp } Q \implies \text{cp } R \implies \text{cp}(\lambda X. f \ (P \ X) \ (Q \ X) \ (R \ X))$

$\langle \text{proof} \rangle$

lemma *cpI4*:

$(\forall \ W \ X \ Y \ Z \ \tau. f \ W \ X \ Y \ Z \ \tau = f(\lambda_. W \ \tau)(\lambda_. X \ \tau)(\lambda_. Y \ \tau)(\lambda_. Z \ \tau) \ \tau) \implies$

$\text{cp } P \implies \text{cp } Q \implies \text{cp } R \implies \text{cp } S \implies \text{cp}(\lambda X. f \ (P \ X) \ (Q \ X) \ (R \ X) \ (S \ X))$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

lemmas *cp-intro*[*intro!*,*simp*,*code-unfold*] =
cp-const
cp-id
cp-defined[*THEN allI*[*THEN allI*[*THEN cpI1*], *of defined*]]
cp-valid[*THEN allI*[*THEN allI*[*THEN cpI1*], *of valid*]]
cp-OclNot[*THEN allI*[*THEN allI*[*THEN cpI1*], *of not*]]
cp-OclAnd[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of op and*]]
cp-OclOr[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of op or*]]
cp-OclImplies[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of op implies*]]
cp-StrongEq[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]],
of StrongEq]]

5.1.6. OCL's if then else endif

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

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

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

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

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

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

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

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

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

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

lemma *OclIf-idem1* [simp]: (if δX then A else A endif) = A
 <proof>

lemma *OclIf-idem2* [simp]: (if $v X$ then A else A endif) = A
 <proof>

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

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

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

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

with term “not” we can express the notation:

syntax

notequal :: $(\mathfrak{A}) \text{Boolean} \Rightarrow (\mathfrak{A}) \text{Boolean} \Rightarrow (\mathfrak{A}) \text{Boolean}$ (**infix** $<> 40$)

translations

$a <> b == \text{CONST } \text{OclNot}(a \doteq b)$

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

5.1.8. Laws to Establish Definedness (δ -closure)

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

lemma *OclNot-defargs*:
 $\tau \models (\text{not } P) \implies \tau \models \delta P$
 <proof>

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

assumes $C: \tau \models A \implies \tau \models B$
shows $\tau \models \text{not } A$
 $\langle \text{proof} \rangle$

5.1.9. A Side-calculus for Constant Terms

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

lemma *const-charn*: $\text{const } X \implies X \tau = X \tau'$
 $\langle \text{proof} \rangle$

lemma *const-subst*:
assumes *const-X*: $\text{const } X$
and *const-Y*: $\text{const } Y$
and *eq*: $X \tau = Y \tau$
and *cp-P*: $\text{cp } P$
and *pp*: $P Y \tau = P Y \tau'$
shows $P X \tau = P X \tau'$
 $\langle \text{proof} \rangle$

lemma *const-imply2* :
assumes $\bigwedge \tau \tau'. P \tau = P \tau' \implies Q \tau = Q \tau'$
shows $\text{const } P \implies \text{const } Q$
 $\langle \text{proof} \rangle$

lemma *const-imply3* :
assumes $\bigwedge \tau \tau'. P \tau = P \tau' \implies Q \tau = Q \tau' \implies R \tau = R \tau'$
shows $\text{const } P \implies \text{const } Q \implies \text{const } R$
 $\langle \text{proof} \rangle$

lemma *const-imply4* :
assumes $\bigwedge \tau \tau'. P \tau = P \tau' \implies Q \tau = Q \tau' \implies R \tau = R \tau' \implies S \tau = S \tau'$
shows $\text{const } P \implies \text{const } Q \implies \text{const } R \implies \text{const } S$
 $\langle \text{proof} \rangle$

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

lemma *const-true[simp]* : const true
 $\langle \text{proof} \rangle$

lemma *const-false[simp]* : const false
 $\langle \text{proof} \rangle$

lemma *const-null[simp]* : const null
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

lemma *const-StrongEq*:
assumes *const X*
assumes *const X'*
shows *const(X ≐ X')*
 ⟨*proof*⟩


```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

Miscellaneous: Overloading the syntax of “bottom”

```

notation bot ( $\perp$ )

```

```

end

```

```

theory UML-PropertyProfiles
imports UML-Logic
begin

```

5.2. Property Profiles for OCL Operators via Isabelle Locales

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

5.2.1. mono

```
locale profile-mono-scheme =
  fixes f :: ('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
```

```
locale profile-mono2 = profile-mono-scheme +
  assumes ∧ x. x ≠ bot ⇒ x ≠ null ⇒ g x ≠ bot
begin
  lemma strict[simp,code-unfold]: f invalid = invalid
  <proof>

  lemma null-strict[simp,code-unfold]: f null = invalid
  <proof>
```

```
lemma cp0 : f X τ = f (λ -. X τ) τ
<proof>
```

```
lemma cp[simp,code-unfold] : cp P ⇒ cp (λX. f (P X))
<proof>
```

```
lemma const[simp,code-unfold] :
  assumes C1 :const X
  shows      const(f X)
  <proof>
```

```
end
```

```
locale profile-mono0 = profile-mono-scheme +
  assumes def-body: ∧ x. x ≠ bot ⇒ x ≠ null ⇒ g x ≠ bot ∧ g x ≠ null
```

```
sublocale profile-mono0 < profile-mono2
<proof>
```

```
context profile-mono0
```

```
begin
```

```
lemma def-homo[simp,code-unfold]: δ(f x) = (δ x)
<proof>
```

```
lemma def-valid-then-def: v(f x) = (δ(f x))
<proof>
```

end

5.2.2. single

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

5.2.3. bin

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

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

lemmas $[simp, code-unfold] = bin'\text{-def} \ bin\text{-def}$

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

begin

interpretation $d_x : profile\text{-single} \ d_x \ \langle proof \rangle$
interpretation $d_y : profile\text{-single} \ d_y \ \langle proof \rangle$

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

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

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

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

$\langle proof \rangle$

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

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

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

lemma *const*[simp,code-unfold] :
assumes $C1 : \text{const}\ X$ **and** $C2 : \text{const}\ Y$
shows $\text{const}(f\ X\ Y)$
 $\langle proof \rangle$

end

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

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

sublocale *profile-bin-scheme-defined* < *profile-bin-scheme defined*
 $\langle proof \rangle$

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

```

    <proof>
end

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

locale profile-bin2 =
  fixes f::('A,'a::null)val => ('A,'b::null)val => ('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-bin2 < profile-bin-scheme-defined valid
  <proof>

locale profile-bin3 =
  fixes f :: ('A,'a::null)val => ('A,'a::null)val => ('A) Boolean
  assumes def-scheme[simplified]: bin' f StrongEq valid valid X Y

sublocale profile-bin3 < profile-bin-scheme valid valid f  $\lambda x y. \llbracket x = y \rrbracket$ 
  <proof>

context profile-bin3
begin
  lemma idem[simp,code-unfold]: f null null = true
    <proof>

  lemma defargs:  $\tau \models f\ x\ y \implies (\tau \models v\ x) \wedge (\tau \models v\ y)$ 
    <proof>

  lemma defined-args-valid':  $\delta\ (f\ x\ y) = (v\ x\ \text{and}\ v\ y)$ 
    <proof>

  lemma refl-ext[simp,code-unfold] :  $(f\ x\ x) = (\text{if}\ (v\ x)\ \text{then}\ \text{true}\ \text{else}\ \text{invalid}\ \text{endif})$ 
    <proof>

  lemma sym :  $\tau \models (f\ x\ y) \implies \tau \models (f\ y\ x)$ 
    <proof>

  lemma symmetric :  $(f\ x\ y) = (f\ y\ x)$ 
    <proof>

  lemma trans :  $\tau \models (f\ x\ y) \implies \tau \models (f\ y\ z) \implies \tau \models (f\ x\ z)$ 
    <proof>

  lemma StrictRefEq-vs-StrongEq:  $\tau \models (v\ x) \implies \tau \models (v\ y) \implies (\tau \models ((f\ x\ y) \triangleq (x \triangleq y)))$ 
    <proof>

```

```

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

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

end

```

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

$$\begin{aligned} \text{defs } \textit{StrictRefEq}_{\textit{Boolean}}[\textit{code-unfold}] : \\ (x :: (\mathfrak{A}) \textit{Boolean}) \doteq y \equiv \lambda \tau. \textit{if } (v \ x) \ \tau = \textit{true} \ \tau \wedge (v \ y) \ \tau = \textit{true} \ \tau \\ \textit{then } (x \triangleq y) \tau \\ \textit{else invalid } \tau \end{aligned}$$

lemma $[simp, code-unfold] : (true \dot{=} false) = false$
 $\langle proof \rangle$
lemma $[simp, code-unfold] : (false \dot{=} true) = false$
 $\langle proof \rangle$

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

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

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

interpretation *StrictRefEq_{Boolean}* : *profile-bin3* $\lambda x y. (x::(\mathfrak{A})\text{Boolean}) \doteq y$
 $\langle \text{proof} \rangle$

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

lemma $(\text{invalid} \doteq \text{false}) = \text{invalid} \langle \text{proof} \rangle$
lemma $(\text{invalid} \doteq \text{true}) = \text{invalid} \langle \text{proof} \rangle$
lemma $(\text{false} \doteq \text{invalid}) = \text{invalid} \langle \text{proof} \rangle$
lemma $(\text{true} \doteq \text{invalid}) = \text{invalid} \langle \text{proof} \rangle$
lemma $((\text{invalid}::(\mathfrak{A})\text{Boolean}) \doteq \text{invalid}) = \text{invalid} \langle \text{proof} \rangle$

Thus, the weak equality is *not* reflexive.

5.2.5. Test Statements on Boolean Operations.

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

Elementary computations on Boolean

```
Assert  $\tau \models v(\text{true})$ 
Assert  $\tau \models \delta(\text{false})$ 
Assert  $\neg(\tau \models \delta(\text{null}))$ 
Assert  $\neg(\tau \models \delta(\text{invalid}))$ 
Assert  $\tau \models v((\text{null}::(\mathfrak{A})\text{Boolean}))$ 
Assert  $\neg(\tau \models v(\text{invalid}))$ 
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  $\tau \models (\text{false} <> \text{true})$ 
```

end

```
theory UML-Void
imports ../UML-PropertyProfiles
begin
```

5.3. Basic Type Void

This *minimal* OCL type contains only two elements: *invalid* and *null*. *Void* could initially be defined as *unit option option*, however the cardinal of this type is more than two, so it would have the cost to consider *Some None* and *Some (Some ())* seemingly everywhere.

5.3.1. Fundamental Properties on Basic Types: Strict Equality

Definition

```

instantiation   Voidbase :: bot
begin
  definition bot-Void-def: (bot-class.bot :: Voidbase)  $\equiv$  Abs-Voidbase None

  instance  $\langle$ proof $\rangle$ 
end

instantiation   Voidbase :: null
begin
  definition null-Void-def: (null::Voidbase)  $\equiv$  Abs-Voidbase [ None ]

  instance  $\langle$ proof $\rangle$ 
end

```

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

```

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

```

Property proof in terms of *profile-bin3*

```

interpretation   StrictRefEqVoid : profile-bin3  $\lambda x y.$  (x::( $\mathfrak{A}$ ) Void)  $\doteq$  y
   $\langle$ proof $\rangle$ 

```

5.3.2. Test Statements

```

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

```

end

```

theory   UML-Integer
imports  ../UML-PropertyProfiles
begin

```


5.4. Basic Type Integer: Operations

5.4.1. Basic Integer Constants

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

definition *OclInt0* :: ('**A**)Integer (0)
where 0 = (λ - . $\llbracket 0::int \rrbracket$)

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

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

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

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

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

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

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

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

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

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

5.4.2. Validity and Definedness Properties

lemma $\delta(\text{null}::('A)\text{Integer}) = \text{false}$ *<proof>*

lemma $v(\text{null}::('A)\text{Integer}) = \text{true}$ *<proof>*

lemma [*simp,code-unfold*]: $\delta(\lambda -. \llbracket n \rrbracket) = \text{true}$
<proof>

lemma [*simp,code-unfold*]: $v(\lambda -. \llbracket n \rrbracket) = \text{true}$
<proof>

```

lemma [simp,code-unfold]:  $\delta \mathbf{0} = \text{true}$   $\langle \text{proof} \rangle$ 
lemma [simp,code-unfold]:  $v \mathbf{0} = \text{true}$   $\langle \text{proof} \rangle$ 
lemma [simp,code-unfold]:  $\delta \mathbf{1} = \text{true}$   $\langle \text{proof} \rangle$ 
lemma [simp,code-unfold]:  $v \mathbf{1} = \text{true}$   $\langle \text{proof} \rangle$ 
lemma [simp,code-unfold]:  $\delta \mathbf{2} = \text{true}$   $\langle \text{proof} \rangle$ 
lemma [simp,code-unfold]:  $v \mathbf{2} = \text{true}$   $\langle \text{proof} \rangle$ 
lemma [simp,code-unfold]:  $\delta \mathbf{6} = \text{true}$   $\langle \text{proof} \rangle$ 
lemma [simp,code-unfold]:  $v \mathbf{6} = \text{true}$   $\langle \text{proof} \rangle$ 
lemma [simp,code-unfold]:  $\delta \mathbf{8} = \text{true}$   $\langle \text{proof} \rangle$ 
lemma [simp,code-unfold]:  $v \mathbf{8} = \text{true}$   $\langle \text{proof} \rangle$ 
lemma [simp,code-unfold]:  $\delta \mathbf{9} = \text{true}$   $\langle \text{proof} \rangle$ 
lemma [simp,code-unfold]:  $v \mathbf{9} = \text{true}$   $\langle \text{proof} \rangle$ 

```

5.4.3. Arithmetical Operations

Definition

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

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

```

definition OclAddInteger :: ('A)Integer  $\Rightarrow$  ('A)Integer  $\Rightarrow$  ('A)Integer (infix  $+_{int}$  40)
where  $x +_{int} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \wedge (\delta y) \tau = \text{true} \wedge$ 
   $\text{then } \llbracket \llbracket x \tau \rrbracket + \llbracket y \tau \rrbracket \rrbracket$ 
   $\text{else invalid } \tau$ 
interpretation OclAddInteger : profile-bin1 op  $+_{int} \lambda x y. \llbracket \llbracket x \rrbracket + \llbracket y \rrbracket \rrbracket$ 
   $\langle \text{proof} \rangle$ 

```

```

definition OclMinusInteger :: ('A)Integer  $\Rightarrow$  ('A)Integer  $\Rightarrow$  ('A)Integer (infix  $-_{int}$  41)
where  $x -_{int} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \wedge (\delta y) \tau = \text{true} \wedge$ 
   $\text{then } \llbracket \llbracket x \tau \rrbracket - \llbracket y \tau \rrbracket \rrbracket$ 
   $\text{else invalid } \tau$ 
interpretation OclMinusInteger : profile-bin1 op  $-_{int} \lambda x y. \llbracket \llbracket x \rrbracket - \llbracket y \rrbracket \rrbracket$ 
   $\langle \text{proof} \rangle$ 

```

```

definition OclMultInteger :: ('A)Integer  $\Rightarrow$  ('A)Integer  $\Rightarrow$  ('A)Integer (infix  $*_{int}$  45)
where  $x *_{int} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \wedge (\delta y) \tau = \text{true} \wedge$ 
   $\text{then } \llbracket \llbracket x \tau \rrbracket * \llbracket y \tau \rrbracket \rrbracket$ 
   $\text{else invalid } \tau$ 
interpretation OclMultInteger : profile-bin1 op  $*_{int} \lambda x y. \llbracket \llbracket x \rrbracket * \llbracket y \rrbracket \rrbracket$ 
   $\langle \text{proof} \rangle$ 

```

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

```

definition OclDivisionInteger :: ('A)Integer  $\Rightarrow$  ('A)Integer  $\Rightarrow$  ('A)Integer (infix  $\text{div}_{int}$  45)

```

where $x \text{ div}_{int} y \equiv \lambda \tau. \text{ if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
 then if $y \tau \neq \text{OclInt0 } \tau$ *then* $\llbracket \llbracket x \tau \rrbracket \text{ div } \llbracket y \tau \rrbracket \rrbracket$ *else invalid* τ
 else invalid τ

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

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

interpretation $\text{OclLess}_{Integer} : \text{profile-bin1 op } <_{int} \lambda x y. \llbracket \llbracket x \rrbracket < \llbracket y \rrbracket \rrbracket$
 $\langle \text{proof} \rangle$

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

interpretation $\text{OclLe}_{Integer} : \text{profile-bin1 op } \leq_{int} \lambda x y. \llbracket \llbracket x \rrbracket \leq \llbracket y \rrbracket \rrbracket$
 $\langle \text{proof} \rangle$

Basic Properties

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

Execution with Invalid or Null or Zero as Argument

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

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

Test Statements

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

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

Assert $\tau \models (((9 *_{int} 4) \text{ div}_{int} 10) \leq_{int} 4)$
Assert $\tau \models \text{not } (\delta (1 \text{ div}_{int} 0))$
Assert $\tau \models \text{not } (v (1 \text{ div}_{int} 0))$

5.4.4. Fundamental Predicates on Integers: Strict Equality

Definition

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

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

Property proof in terms of *profile-bin3*

interpretation $\text{StrictRefEq}_{Integer} : \text{profile-bin3} \lambda x y. (x::(\mathcal{A})Integer) \doteq y$
 $\langle \text{proof} \rangle$

lemma *integer-non-null* [simp]: $((\lambda-. \llbracket n \rrbracket) \doteq (\text{null}::(\mathcal{A})Integer)) = \text{false}$
 $\langle \text{proof} \rangle$

lemma *null-non-integer* [simp]: $((\text{null}::(\mathcal{A})Integer) \doteq (\lambda-. \llbracket n \rrbracket)) = \text{false}$
 $\langle \text{proof} \rangle$

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

5.4.5. Test Statements on Basic Integer

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

Elementary computations on Integer

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

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  $\tau \models v\ (null::(\mathfrak{A})Integer)$ 
Assert  $\tau \models (invalid \triangleq invalid)$ 
Assert  $\tau \models (null \triangleq null)$ 
Assert  $\tau \models (4 \triangleq 4)$ 
Assert  $\neg(\tau \models (9 \triangleq 10))$ 
Assert  $\neg(\tau \models (invalid \triangleq 10))$ 
Assert  $\neg(\tau \models (null \triangleq 10))$ 
Assert  $\neg(\tau \models (invalid \doteq (invalid::(\mathfrak{A})Integer)))$ 
Assert  $\neg(\tau \models v\ (invalid \doteq (invalid::(\mathfrak{A})Integer)))$ 
Assert  $\neg(\tau \models (invalid <> (invalid::(\mathfrak{A})Integer)))$ 
Assert  $\neg(\tau \models v\ (invalid <> (invalid::(\mathfrak{A})Integer)))$ 
Assert  $\tau \models (null \doteq (null::(\mathfrak{A})Integer))$ 
Assert  $\tau \models (null \doteq (null::(\mathfrak{A})Integer))$ 
Assert  $\tau \models (4 \doteq 4)$ 
Assert  $\neg(\tau \models (4 <> 4))$ 
Assert  $\neg(\tau \models (4 \doteq 10))$ 
Assert  $\tau \models (4 <> 10)$ 
Assert  $\neg(\tau \models (0 <_{int}\ null))$ 
Assert  $\neg(\tau \models (\delta\ (0 <_{int}\ null)))$ 

```

end

```

theory UML-Real
imports ../UML-PropertyProfiles
begin

```

5.5. Basic Type Real: Operations

5.5.1. Basic Real Constants

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

```

definition OclReal0 :: ( $\mathfrak{A}$ )Real (0.0)
where 0.0 = ( $\lambda$  . . [ $\lfloor 0::real \rfloor$ ])

```

```

definition OclReal1 :: ( $\mathfrak{A}$ )Real (1.0)
where 1.0 = ( $\lambda$  . . [ $\lfloor 1::real \rfloor$ ])

```

```

definition OclReal2 :: ( $\mathfrak{A}$ )Real (2.0)

```

where **2.0** = (λ - . $\llbracket 2::real \rrbracket$)

definition *OclReal3* :: (\mathfrak{A}) *Real* (**3.0**)

where **3.0** = (λ - . $\llbracket 3::real \rrbracket$)

definition *OclReal4* :: (\mathfrak{A}) *Real* (**4.0**)

where **4.0** = (λ - . $\llbracket 4::real \rrbracket$)

definition *OclReal5* :: (\mathfrak{A}) *Real* (**5.0**)

where **5.0** = (λ - . $\llbracket 5::real \rrbracket$)

definition *OclReal6* :: (\mathfrak{A}) *Real* (**6.0**)

where **6.0** = (λ - . $\llbracket 6::real \rrbracket$)

definition *OclReal7* :: (\mathfrak{A}) *Real* (**7.0**)

where **7.0** = (λ - . $\llbracket 7::real \rrbracket$)

definition *OclReal8* :: (\mathfrak{A}) *Real* (**8.0**)

where **8.0** = (λ - . $\llbracket 8::real \rrbracket$)

definition *OclReal9* :: (\mathfrak{A}) *Real* (**9.0**)

where **9.0** = (λ - . $\llbracket 9::real \rrbracket$)

definition *OclReal10* :: (\mathfrak{A}) *Real* (**10.0**)

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

definition *OclRealpi* :: (\mathfrak{A}) *Real* (π)

where π = (λ - . $\llbracket \pi \rrbracket$)

5.5.2. Validity and Definedness Properties

lemma $\delta(\text{null}::(\mathfrak{A})\text{Real}) = \text{false}$ $\langle \text{proof} \rangle$

lemma $v(\text{null}::(\mathfrak{A})\text{Real}) = \text{true}$ $\langle \text{proof} \rangle$

lemma $[simp, code-unfold]: \delta(\lambda -. \llbracket n \rrbracket) = \text{true}$
 $\langle \text{proof} \rangle$

lemma $[simp, code-unfold]: v(\lambda -. \llbracket n \rrbracket) = \text{true}$
 $\langle \text{proof} \rangle$

lemma $[simp, code-unfold]: \delta \mathbf{0.0} = \text{true}$ $\langle \text{proof} \rangle$

lemma $[simp, code-unfold]: v \mathbf{0.0} = \text{true}$ $\langle \text{proof} \rangle$

lemma $[simp, code-unfold]: \delta \mathbf{1.0} = \text{true}$ $\langle \text{proof} \rangle$

lemma $[simp, code-unfold]: v \mathbf{1.0} = \text{true}$ $\langle \text{proof} \rangle$

lemma $[simp, code-unfold]: \delta \mathbf{2.0} = \text{true}$ $\langle \text{proof} \rangle$

lemma $[simp, code-unfold]: v \mathbf{2.0} = \text{true}$ $\langle \text{proof} \rangle$

lemma $[simp, code-unfold]: \delta \mathbf{6.0} = \text{true}$ $\langle \text{proof} \rangle$

lemma $[simp, code-unfold]: v \mathbf{6.0} = \text{true}$ $\langle \text{proof} \rangle$

lemma $[simp, code-unfold]: \delta \text{ 8.0} = true \langle proof \rangle$
lemma $[simp, code-unfold]: v \text{ 8.0} = true \langle proof \rangle$
lemma $[simp, code-unfold]: \delta \text{ 9.0} = true \langle proof \rangle$
lemma $[simp, code-unfold]: v \text{ 9.0} = true \langle proof \rangle$

5.5.3. Arithmetical Operations

Definition

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

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

definition $OclAdd_{Real} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Real \text{ (infix } +_{real} \text{ 40)}$
where $x +_{real} y \equiv \lambda \tau. \text{ if } (\delta x) \tau = true \wedge (\delta y) \tau = true \tau$
 $\quad \text{then } \llbracket \llbracket x \tau \rrbracket + \llbracket y \tau \rrbracket \rrbracket$
 $\quad \text{else } invalid \tau$

interpretation $OclAdd_{Real} : profile-bin1 \text{ op } +_{real} \lambda x y. \llbracket \llbracket x \rrbracket + \llbracket y \rrbracket \rrbracket$
 $\langle proof \rangle$

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

interpretation $OclMinus_{Real} : profile-bin1 \text{ op } -_{real} \lambda x y. \llbracket \llbracket x \rrbracket - \llbracket y \rrbracket \rrbracket$
 $\langle proof \rangle$

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

interpretation $OclMult_{Real} : profile-bin1 \text{ op } *_{real} \lambda x y. \llbracket \llbracket x \rrbracket * \llbracket y \rrbracket \rrbracket$
 $\langle proof \rangle$

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

definition $OclDivision_{Real} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Real \text{ (infix } div_{real} \text{ 45)}$
where $x div_{real} y \equiv \lambda \tau. \text{ if } (\delta x) \tau = true \wedge (\delta y) \tau = true \tau$
 $\quad \text{then if } y \tau \neq OclReal0 \tau \text{ then } \llbracket \llbracket x \tau \rrbracket / \llbracket y \tau \rrbracket \rrbracket \text{ else } invalid \tau$
 $\quad \text{else } invalid \tau$

definition $mod_float \ a \ b = a - real \ (floor \ (a / b)) * b$

definition $OclModulus_{Real} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Real \text{ (infix } mod_{real} \text{ 45)}$

where $x mod_{real} y \equiv \lambda \tau. \text{ if } (\delta x) \tau = true \wedge (\delta y) \tau = true \tau$
 $\quad \text{then if } y \tau \neq OclReal0 \tau \text{ then } \llbracket mod_float \ \llbracket x \tau \rrbracket \ \llbracket y \tau \rrbracket \rrbracket \text{ else } invalid \tau$

else invalid τ

definition $OclLess_{Real} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Boolean$ (**infix** $<_{real}$ 35)
where $x <_{real} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \wedge (\delta y) \tau = \text{true} \tau$
 then $\llbracket [x \ \tau] \rrbracket < \llbracket [y \ \tau] \rrbracket$
 else invalid τ
interpretation $OclLess_{Real} : \text{profile-bin1 op } <_{real} \lambda x y. \llbracket [x] \rrbracket < \llbracket [y] \rrbracket$
 $\langle \text{proof} \rangle$

definition $OclLe_{Real} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Boolean$ (**infix** \leq_{real} 35)
where $x \leq_{real} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \wedge (\delta y) \tau = \text{true} \tau$
 then $\llbracket [x \ \tau] \rrbracket \leq \llbracket [y \ \tau] \rrbracket$
 else invalid τ
interpretation $OclLe_{Real} : \text{profile-bin1 op } \leq_{real} \lambda x y. \llbracket [x] \rrbracket \leq \llbracket [y] \rrbracket$
 $\langle \text{proof} \rangle$

Basic Properties

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

Execution with Invalid or Null or Zero as Argument

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

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

Test Statements

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

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

5.5.4. Fundamental Predicates on Reals: Strict Equality

Definition

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

defs *StrictRefEq_{Real}* [code-unfold] :

$$(x::(\mathcal{A})Real) \doteq y \equiv \lambda \tau. \text{ if } (v\ x) \tau = \text{true} \wedge (v\ y) \tau = \text{true} \wedge$$

$$\text{ then } (x \triangleq y) \tau$$

$$\text{ else invalid } \tau$$

Property proof in terms of *profile-bin3*

interpretation *StrictRefEq_{Real}* : *profile-bin3* $\lambda x\ y. (x::(\mathcal{A})Real) \doteq y$
 $\langle \text{proof} \rangle$

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

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

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

Const

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

5.5.5. Test Statements on Basic Real

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

Elementary computations on Real

```

Assert  $\tau \models 1.0 <> 2.0$ 
Assert  $\tau \models 2.0 <> 1.0$ 
Assert  $\tau \models 2.0 \doteq 2.0$ 

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

```

end

```

theory UML-String
imports ../UML-PropertyProfiles
begin

```

5.6. Basic Type String: Operations

5.6.1. Basic String Constants

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

```

definition OclStringa :: ('A)String (a)
where a = ( $\lambda$  - .  $[[\text{"a"}]]$ )

```

```

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

```

definition $OclStringc :: ('A)String \rightarrow c$
where $c = (\lambda - . \llbracket ''c'' \rrbracket)$

5.6.2. Validity and Definedness Properties

lemma $\delta(\text{null}::('A)String) = \text{false} \langle \text{proof} \rangle$

lemma $v(\text{null}::('A)String) = \text{true} \langle \text{proof} \rangle$

lemma $[simp, code-unfold]: \delta (\lambda -. \llbracket n \rrbracket) = \text{true} \langle \text{proof} \rangle$

lemma $[simp, code-unfold]: v (\lambda -. \llbracket n \rrbracket) = \text{true} \langle \text{proof} \rangle$

lemma $[simp, code-unfold]: \delta a = \text{true} \langle \text{proof} \rangle$

lemma $[simp, code-unfold]: v a = \text{true} \langle \text{proof} \rangle$

5.6.3. String Operations

Definition

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

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

definition $OclAddString :: ('A)String \Rightarrow ('A)String \Rightarrow ('A)String \text{ (infix } +_{string} 40)$

where $x +_{string} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \wedge (\delta y) \tau = \text{true} \wedge$
 $\text{then } \llbracket \text{concat } [\llbracket x \tau \rrbracket, \llbracket y \tau \rrbracket] \rrbracket$
 $\text{else } \text{invalid } \tau$

interpretation $OclAddString : \text{profile-bin1 op } +_{string} \lambda x y. \llbracket \text{concat } [\llbracket x \rrbracket, \llbracket y \rrbracket] \rrbracket$
 $\langle \text{proof} \rangle$

Basic Properties

lemma $OclAddString\text{-not-commute}: \exists X Y. (X +_{string} Y) \neq (Y +_{string} X)$
 $\langle \text{proof} \rangle$

Test Statements

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

5.6.4. Fundamental Properties on Strings: Strict Equality

Definition

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

defs $StrictRefEq_{String}[code-unfold]$:

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

Property proof in terms of *profile-bin3*

interpretation $StrictRefEq_{String} : profile-bin3 \lambda x\ y. (x::(\mathfrak{A})String) \doteq y$
 $\langle proof \rangle$

5.6.5. Test Statements on Basic String

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

Elementary computations on String

```

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

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

```

end

```

theory   UML-Pair
imports ../basic-types/UML-Boolean
          ../basic-types/UML-Integer
begin

```

5.7. Collection Type Pairs: Operations

The OCL standard provides the concept of *Tuples*, i.e. a family of record-types with projection functions. In FeatherWeight OCL, only the theory of a special case is 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, ...

5.7.1. Semantic Properties of the Type Constructor

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

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

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

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

5.7.2. Strict Equality

Definition

After the part of foundational operations on sets, we detail here equality on sets. Strong equality is inherited from the OCL core, but we have to consider the case of the strict equality. We decide to overload strict equality in the same way we do for other value's in OCL:

defs $StrictRefEq_{Pair} :$
 $((x::('A, 'A::null, 'B::null)Pair) \doteq y) \equiv (\lambda \tau. \text{if } (v \ x) \ \tau = true \ \tau \wedge (v \ y) \ \tau = true \ \tau$
 $\quad \text{then } (x \triangleq y) \tau$
 $\quad \text{else invalid } \tau)$

Property proof in terms of *profile-bin3*

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

5.7.3. Standard Operations

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

Definition: OclPair Constructor

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

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

Definition: OclFst

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

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

Definition: OclSnd

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

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

5.7.4. Logical Properties

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

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

5.7.5. Execution Properties

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

<proof>

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

<proof>

5.7.6. Test Statements

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

end

theory *UML-Set*
imports *../basic-types/UML-Boolean*
../basic-types/UML-Integer
begin

no-notation *None* (\perp)

5.8. Collection Type Set: Operations

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

Our notion of typed set goes beyond the usual notion of a finite executable set and is powerful enough to capture *the extension of a type* in UML and OCL. This means we can have in Featherweight OCL Sets containing all possible elements of a type, not only those (finite) ones representable in a state. This holds for base types as well as class types, although the notion for class-types — involving object id's not 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 $Integer :: (\mathfrak{A}, Integer_{base}) Set$
where $Integer \equiv (\lambda \tau. (Abs-Set_{base} \circ Some \circ Some) ((Some \circ Some) ' (UNIV::int set)))$

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

lemma $Integer\text{-}defined : \delta Integer = true$
 $\langle proof \rangle$

lemma $Integer_{null}\text{-}defined : \delta Integer_{null} = true$
 $\langle proof \rangle$

This allows the theorems:

$\tau \models \delta x \implies \tau \models (Integer \rightarrow includes(x)) \quad \tau \models \delta x \implies \tau \models Integer \triangleq (Integer \rightarrow including(x))$

and

$\tau \models v x \implies \tau \models (Integer_{null} \rightarrow includes(x)) \quad \tau \models v x \implies \tau \models Integer_{null} \triangleq (Integer_{null} \rightarrow including(x))$

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

5.8.2. Validity and Definedness Properties

Every element in a defined set is valid.

lemma $Set\text{-}inv\text{-}lemma : \tau \models (\delta X) \implies \forall x \in [[Rep-Set_{base} (X \tau)]] . x \neq bot$
 $\langle proof \rangle$

lemma $Set\text{-}inv\text{-}lemma' :$
assumes $x\text{-}def : \tau \models \delta X$
and $e\text{-}mem : e \in [[Rep-Set_{base} (X \tau)]]$
shows $\tau \models v (\lambda \cdot . e)$
 $\langle proof \rangle$

lemma $abs\text{-}rep\text{-}simp' :$
assumes $S\text{-}all\text{-}def : \tau \models \delta S$
shows $Abs-Set_{base} [[[[Rep-Set_{base} (S \tau)]]]] = S \tau$
 $\langle proof \rangle$

lemma $S\text{-}lift' :$
assumes $S\text{-}all\text{-}def : (\tau :: \mathfrak{A} \text{ st}) \models \delta S$
shows $\exists S'. (\lambda a (-::\mathfrak{A} \text{ st}). a) ' [[Rep-Set_{base} (S \tau)]] = (\lambda a (-::\mathfrak{A} \text{ st}). [a]) ' S'$
 $\langle proof \rangle$

lemma $invalid\text{-}set\text{-}OclNot\text{-}defined [simp, code\text{-}unfold] : \delta (invalid::(\mathfrak{A}, \alpha::null) Set) = false$
 $\langle proof \rangle$

lemma $null\text{-}set\text{-}OclNot\text{-}defined [simp, code\text{-}unfold] : \delta (null::(\mathfrak{A}, \alpha::null) Set) = false$
 $\langle proof \rangle$

lemma $invalid\text{-}set\text{-}valid [simp, code\text{-}unfold] : v (invalid::(\mathfrak{A}, \alpha::null) Set) = false$
 $\langle proof \rangle$

lemma *null-set-valid* $[simp, code-unfold]: v(null::('A, 'a::null) Set) = true$
 $\langle proof \rangle$

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

5.8.3. Constants on Sets

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

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

lemma $mtSet-valid[simp,code-unfold]:v(Set\{\}) = true$
 $\langle proof \rangle$

lemma *mtSet-rep-set*: $\llbracket Rep\text{-}Set_{base}(Set\{\} \tau) \rrbracket = \{\}$
 $\langle proof \rangle$

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

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

5.8.4. Operations

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

Definition: OclIncluding

definition $OclIncluding \quad :: [(\mathfrak{A}, \alpha :: null) \text{ Set}, (\mathfrak{A}, \alpha) \text{ val}] \Rightarrow (\mathfrak{A}, \alpha) \text{ Set}$
where $OclIncluding \ x \ y = (\lambda \ \tau. \text{ if } (\delta \ x) \ \tau = true \ \tau \wedge (v \ y) \ \tau = true \ \tau$
then $Abs\text{-}Set_{base} \llbracket \llbracket Rep\text{-}Set_{base} \ (x \ \tau) \rrbracket \cup \{y \ \tau\} \rrbracket$
else invalid τ)

notation $OclIncluding \quad (--->including'(-'))$

interpretation $OclIncluding : profile-bin2 \ OclIncluding \ \lambda x \ y. \ Abs\text{-}Set_{base}[[[Rep\text{-}Set_{base} \ x]] \cup \{y\}]]$
 $\langle proof \rangle$

syntax
 $-OclFinset :: args \Rightarrow ('A, 'a::null) \text{ Set } (Set\{-\})$

translations

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

Definition: OclExcluding

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

Definition: OclIncludes

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

Definition: OclExcludes

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

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

Definition: OclSize

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

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

Definition: OclIsEmpty

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

Definition: OclNotEmpty

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

Definition: OclANY

definition *OclANY* :: ($\mathfrak{A}, \alpha :: \text{null}$) *Set* \Rightarrow (\mathfrak{A}, α) *val*
where *OclANY* *x* = ($\lambda \tau$. *if* (*v x*) $\tau = \text{true}$ τ
 then if (δx *and* *OclNotEmpty x*) $\tau = \text{true}$ τ
 then SOME *y*. *y* \in [*Rep-Set_{base}* (*x* τ)]
 else null τ
 else \perp)
notation *OclANY* ($-->\text{any}'()$)

Definition: OclForall

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

definition *OclForall* :: ($\mathfrak{A}, \alpha :: \text{null}$) *Set*, (\mathfrak{A}, α) *val* \Rightarrow (\mathfrak{A}) *Boolean* \Rightarrow \mathfrak{A} *Boolean*
where *OclForall* *S P* = ($\lambda \tau$. *if* (δS) $\tau = \text{true}$ τ
 then if ($\exists x \in$ [*Rep-Set_{base}* (*S* τ)]). *P*($\lambda \cdot$ *x*) $\tau = \text{false}$ τ
 then false τ
 else if ($\exists x \in$ [*Rep-Set_{base}* (*S* τ)]). *P*($\lambda \cdot$ *x*) $\tau = \text{invalid}$ τ
 then invalid τ
 else if ($\exists x \in$ [*Rep-Set_{base}* (*S* τ)]). *P*($\lambda \cdot$ *x*) $\tau = \text{null}$ τ
 then null τ
 else true τ
 else \perp)

syntax

-*OclForall* :: ($\mathfrak{A}, \alpha :: \text{null}$) *Set*, *id*, (\mathfrak{A}) *Boolean* \Rightarrow \mathfrak{A} *Boolean* ($(-) \rightarrow \text{forAll}'(-|')$)

translations

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

Definition: OclExists

Like *OclForall*, *OclExists* is also not strict.

definition *OclExists* :: ($\mathfrak{A}, \alpha :: \text{null}$) *Set*, (\mathfrak{A}, α) *val* \Rightarrow (\mathfrak{A}) *Boolean* \Rightarrow \mathfrak{A} *Boolean*
where *OclExists* *S P* = *not*(*OclForall* *S* (λX . *not* (*P X*)))

syntax

-*OclExist* :: ($\mathfrak{A}, \alpha :: \text{null}$) *Set*, *id*, (\mathfrak{A}) *Boolean* \Rightarrow \mathfrak{A} *Boolean* ($(-) \rightarrow \text{exists}'(-|')$)

translations

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

Definition: OclIterate

definition *OclIterate* :: ($\mathfrak{A}, \alpha :: \text{null}$) *Set*, ($\mathfrak{A}, \beta :: \text{null}$) *val*,
 (\mathfrak{A}, α) *val* \Rightarrow (\mathfrak{A}, β) *val* \Rightarrow (\mathfrak{A}, β) *val* \Rightarrow (\mathfrak{A}, β) *val*

where $OclIterate\ S\ A\ F = (\lambda\ \tau. \text{if } (\delta\ S)\ \tau = \text{true } \tau \wedge (\nu\ A)\ \tau = \text{true } \tau \wedge \text{finite}[[Rep\text{-}Set_{base}\ (S\ \tau)]]$

$\text{then } (Finite\text{-}Set.fold\ (F)\ (A)\ ((\lambda a\ \tau. a) \text{ ' } [[Rep\text{-}Set_{base}\ (S\ \tau)]]))\tau$
 $\text{else } \perp)$

syntax

$-OclIterate :: [(\mathfrak{A}, ' \alpha :: null)\ Set, idt, idt, ' \alpha, ' \beta] \Rightarrow (\mathfrak{A}, ' \gamma)val$
 $(- \rightarrow iterate'(-; ==- | -'))$

translations

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

Definition: OclSelect

definition $OclSelect :: [(\mathfrak{A}, ' \alpha :: null)\ Set, (\mathfrak{A}, ' \alpha)val \Rightarrow (\mathfrak{A})Boolean] \Rightarrow (\mathfrak{A}, ' \alpha)Set$

where $OclSelect\ S\ P = (\lambda\ \tau. \text{if } (\delta\ S)\ \tau = \text{true } \tau$

$\text{then if } (\exists x \in [[Rep\text{-}Set_{base}\ (S\ \tau)]] . P(\lambda\ -. x)\ \tau = \text{invalid } \tau)$

$\text{then invalid } \tau$

$\text{else } Abs\text{-}Set_{base}\ [[\{x \in [[Rep\text{-}Set_{base}\ (S\ \tau)]] . P(\lambda\ -. x)\ \tau \neq \text{false}}$

$\tau\}]]$

$\text{else invalid } \tau)$

syntax

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

translations

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

Definition: OclReject

definition $OclReject :: [(\mathfrak{A}, ' \alpha :: null)\ Set, (\mathfrak{A}, ' \alpha)val \Rightarrow (\mathfrak{A})Boolean] \Rightarrow (\mathfrak{A}, ' \alpha :: null)Set$

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

syntax

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

translations

$X \rightarrow reject(x \mid P) == CONST\ OclReject\ X\ (\%x. P)$

Definition (futur operators)

consts

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

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

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

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

$OclComplement :: (\mathfrak{A}, ' \alpha :: null)\ Set \Rightarrow (\mathfrak{A}, ' \alpha) Set$

$OclUnion :: [(\mathfrak{A}, ' \alpha :: null)\ Set, (\mathfrak{A}, ' \alpha) Set] \Rightarrow (\mathfrak{A}, ' \alpha) Set$

$OclIntersection :: [(\mathfrak{A}, ' \alpha :: null)\ Set, (\mathfrak{A}, ' \alpha) Set] \Rightarrow (\mathfrak{A}, ' \alpha) Set$

notation

$OclCount\ \ \ \ \ \ (\rightarrow count'(-))$

notation

$OclSum\ \ \ \ \ \ (\rightarrow sum'(-))$

notation

$OclIncludesAll\ (\rightarrow includesAll'(-))$

notation

$$OclExcludesAll \ (-->excludesAll'(-) \)$$
notation

$$OclComplement \ (-->complement'('))$$
notation

$$OclUnion \ \ \ \ \ \ (-->union'(-' \ \ \ \ \ \))$$
notation

$$OclIntersection(-->intersection'(-' \ \ \ \ \ \))$$
Validity and Definedness Properties**OclIncluding**

lemma *OclIncluding-defined-args-valid*:

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

<proof>

lemma *OclIncluding-valid-args-valid*:

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

<proof>

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

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

<proof>

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

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

<proof>

OclExcluding

lemma *OclExcluding-defined-args-valid*:

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

<proof>

lemma *OclExcluding-valid-args-valid*:

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

<proof>

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

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

<proof>

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

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

<proof>

OclIncludes

lemma *OclIncludes-defined-args-valid*:

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

<proof>

lemma *OclIncludes-valid-args-valid*:

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

<proof>

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

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

<proof>

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

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

<proof>

OclExcludes

lemma *OclExcludes-defined-args-valid*:

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

<proof>

lemma *OclExcludes-valid-args-valid*:

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

<proof>

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

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

<proof>

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

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

<proof>

OclSize

lemma *OclSize-defined-args-valid*: $\tau \models \delta(X \rightarrow \text{size}()) \implies \tau \models \delta X$

<proof>

lemma *OclSize-infinite*:

assumes *non-finite*: $\tau \models \text{not}(\delta(S \rightarrow \text{size}()))$

shows $(\tau \models \text{not}(\delta(S))) \vee \neg \text{finite } [[\text{Rep-Set}_{\text{base}}(S \ \tau)]]$

<proof>

lemma $\tau \models \delta X \implies \neg \text{finite } [[\text{Rep-Set}_{\text{base}}(X \ \tau)]] \implies \neg \tau \models \delta(X \rightarrow \text{size}())$

<proof>

lemma *size-defined*:

assumes *X-finite*: $\bigwedge \tau. \text{finite } [[\text{Rep-Set}_{\text{base}}(X \ \tau)]]$

shows $\delta(X \rightarrow \text{size}()) = \delta X$

$\langle proof \rangle$

lemma *size-defined'*:

assumes *X-finite*: *finite* $[[Rep-Set_{base} (X \ \tau)]]$

shows $(\tau \models \delta (X \rightarrow size())) = (\tau \models \delta X)$

$\langle proof \rangle$

OclIsEmpty

lemma *OclIsEmpty-defined-args-valid*: $\tau \models \delta (X \rightarrow isEmpty()) \implies \tau \models v \ X$

$\langle proof \rangle$

lemma $\tau \models \delta (null \rightarrow isEmpty())$

$\langle proof \rangle$

lemma *OclIsEmpty-infinite*: $\tau \models \delta X \implies \neg \text{finite } [[Rep-Set_{base} (X \ \tau)]] \implies \neg \tau \models \delta (X \rightarrow isEmpty())$

$\langle proof \rangle$

OclNotEmpty

lemma *OclNotEmpty-defined-args-valid*: $\tau \models \delta (X \rightarrow notEmpty()) \implies \tau \models v \ X$

$\langle proof \rangle$

lemma $\tau \models \delta (null \rightarrow notEmpty())$

$\langle proof \rangle$

lemma *OclNotEmpty-infinite*: $\tau \models \delta X \implies \neg \text{finite } [[Rep-Set_{base} (X \ \tau)]] \implies \neg \tau \models \delta (X \rightarrow notEmpty())$

$\langle proof \rangle$

lemma *OclNotEmpty-has-elt* : $\tau \models \delta X \implies$
 $\tau \models X \rightarrow notEmpty() \implies$
 $\exists e. e \in [[Rep-Set_{base} (X \ \tau)]]$

$\langle proof \rangle$

OclANY

lemma *OclANY-defined-args-valid*: $\tau \models \delta (X \rightarrow any()) \implies \tau \models \delta X$

$\langle proof \rangle$

lemma $\tau \models \delta X \implies \tau \models X \rightarrow isEmpty() \implies \neg \tau \models \delta (X \rightarrow any())$

$\langle proof \rangle$

lemma *OclANY-valid-args-valid*:

$(\tau \models v(X \rightarrow any())) = (\tau \models v \ X)$

$\langle proof \rangle$

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

$v(X \rightarrow any()) = (v \ X)$

$\langle proof \rangle$

Execution with Invalid or Null or Infinite Set as Argument

OclIncluding

lemma *OclIncluding-invalid*[simp,code-unfold]:(*invalid*→*including*(*x*)) = *invalid*
(*proof*)

lemma *OclIncluding-invalid-args*[simp,code-unfold]:(*X*→*including*(*invalid*)) = *invalid*
(*proof*)

lemma *OclIncluding-null*[simp,code-unfold]:(*null*→*including*(*x*)) = *invalid*
(*proof*)

OclExcluding

lemma *OclExcluding-invalid*[simp,code-unfold]:(*invalid*→*excluding*(*x*)) = *invalid*
(*proof*)

lemma *OclExcluding-invalid-args*[simp,code-unfold]:(*X*→*excluding*(*invalid*)) = *invalid*
(*proof*)

lemma *OclExcluding-null*[simp,code-unfold]:(*null*→*excluding*(*x*)) = *invalid*
(*proof*)

OclIncludes

lemma *OclIncludes-invalid*[simp,code-unfold]:(*invalid*→*includes*(*x*)) = *invalid*
(*proof*)

lemma *OclIncludes-invalid-args*[simp,code-unfold]:(*X*→*includes*(*invalid*)) = *invalid*
(*proof*)

lemma *OclIncludes-null*[simp,code-unfold]:(*null*→*includes*(*x*)) = *invalid*
(*proof*)

OclExcludes

lemma *OclExcludes-invalid*[simp,code-unfold]:(*invalid*→*excludes*(*x*)) = *invalid*
(*proof*)

lemma *OclExcludes-invalid-args*[simp,code-unfold]:(*X*→*excludes*(*invalid*)) = *invalid*
(*proof*)

lemma *OclExcludes-null*[simp,code-unfold]:(*null*→*excludes*(*x*)) = *invalid*
(*proof*)

OclSize

lemma *OclSize-invalid*[simp,code-unfold]:(*invalid*→*size*()) = *invalid*
(*proof*)

lemma *OclSize-null*[simp,code-unfold]:(*null*→*size*()) = *invalid*
(*proof*)

OclIsEmpty

lemma *OclIsEmpty-invalid*[simp,code-unfold]:(*invalid*→*isEmpty()*) = *invalid*
 ⟨proof⟩

lemma *OclIsEmpty-null*[simp,code-unfold]:(*null*→*isEmpty()*) = *true*
 ⟨proof⟩

OclNotEmpty

lemma *OclNotEmpty-invalid*[simp,code-unfold]:(*invalid*→*notEmpty()*) = *invalid*
 ⟨proof⟩

lemma *OclNotEmpty-null*[simp,code-unfold]:(*null*→*notEmpty()*) = *false*
 ⟨proof⟩

OclANY

lemma *OclANY-invalid*[simp,code-unfold]:(*invalid*→*any()*) = *invalid*
 ⟨proof⟩

lemma *OclANY-null*[simp,code-unfold]:(*null*→*any()*) = *null*
 ⟨proof⟩

OclForall

lemma *OclForall-invalid*[simp,code-unfold]:*invalid*→*forall(a | P a)* = *invalid*
 ⟨proof⟩

lemma *OclForall-null*[simp,code-unfold]:*null*→*forall(a | P a)* = *invalid*
 ⟨proof⟩

OclExists

lemma *OclExists-invalid*[simp,code-unfold]:*invalid*→*exists(a | P a)* = *invalid*
 ⟨proof⟩

lemma *OclExists-null*[simp,code-unfold]:*null*→*exists(a | P a)* = *invalid*
 ⟨proof⟩

OclIterate

lemma *OclIterate-invalid*[simp,code-unfold]:*invalid*→*iterate(a; x = A | P a x)* = *invalid*
 ⟨proof⟩

lemma *OclIterate-null*[simp,code-unfold]:*null*→*iterate(a; x = A | P a x)* = *invalid*
 ⟨proof⟩

lemma *OclIterate-invalid-args*[simp,code-unfold]:*S*→*iterate(a; x = invalid | P a x)* = *invalid*
 ⟨proof⟩

An open question is this ...

lemma *S*→*iterate(a; x = null | P a x)* = *invalid*
 ⟨proof⟩

lemma *OclIterate-infinite*:
assumes *non-finite*: $\tau \models \text{not}(\delta(S \rightarrow \text{size}()))$
shows $(\text{OclIterate } S \ A \ F) \ \tau = \text{invalid } \tau$
 $\langle \text{proof} \rangle$

OclSelect

lemma *OclSelect-invalid[simp,code-unfold]*: $\text{invalid} \rightarrow \text{select}(a \mid P \ a) = \text{invalid}$
 $\langle \text{proof} \rangle$

lemma *OclSelect-null[simp,code-unfold]*: $\text{null} \rightarrow \text{select}(a \mid P \ a) = \text{invalid}$
 $\langle \text{proof} \rangle$

OclReject

lemma *OclReject-invalid[simp,code-unfold]*: $\text{invalid} \rightarrow \text{reject}(a \mid P \ a) = \text{invalid}$
 $\langle \text{proof} \rangle$

lemma *OclReject-null[simp,code-unfold]*: $\text{null} \rightarrow \text{reject}(a \mid P \ a) = \text{invalid}$
 $\langle \text{proof} \rangle$

Context Passing

lemma *cp-OclIncluding*:
 $(X \rightarrow \text{including}(x)) \ \tau = ((\lambda \ -. \ X \ \tau) \rightarrow \text{including}(\lambda \ -. \ x \ \tau)) \ \tau$
 $\langle \text{proof} \rangle$

lemma *cp-OclExcluding*:
 $(X \rightarrow \text{excluding}(x)) \ \tau = ((\lambda \ -. \ X \ \tau) \rightarrow \text{excluding}(\lambda \ -. \ x \ \tau)) \ \tau$
 $\langle \text{proof} \rangle$

lemma *cp-OclIncludes*:
 $(X \rightarrow \text{includes}(x)) \ \tau = ((\lambda \ -. \ X \ \tau) \rightarrow \text{includes}(\lambda \ -. \ x \ \tau)) \ \tau$
 $\langle \text{proof} \rangle$

lemma *cp-OclIncludes1*:
 $(X \rightarrow \text{includes}(x)) \ \tau = (X \rightarrow \text{includes}(\lambda \ -. \ x \ \tau)) \ \tau$
 $\langle \text{proof} \rangle$

lemma *cp-OclExcludes*:
 $(X \rightarrow \text{excludes}(x)) \ \tau = ((\lambda \ -. \ X \ \tau) \rightarrow \text{excludes}(\lambda \ -. \ x \ \tau)) \ \tau$
 $\langle \text{proof} \rangle$

lemma *cp-OclSize*: $X \rightarrow \text{size}() \ \tau = ((\lambda \ -. \ X \ \tau) \rightarrow \text{size}()) \ \tau$
 $\langle \text{proof} \rangle$

lemma *cp-OclIsEmpty*: $X \rightarrow \text{isEmpty}() \ \tau = ((\lambda \ -. \ X \ \tau) \rightarrow \text{isEmpty}()) \ \tau$
 $\langle \text{proof} \rangle$

lemma *cp-OclNotEmpty*: $X \rightarrow \text{notEmpty}() \ \tau = ((\lambda \ -. \ X \ \tau) \rightarrow \text{notEmpty}()) \ \tau$
 $\langle \text{proof} \rangle$

lemma *cp-OclANY*: $X \rightarrow any() \tau = ((\lambda -. X \tau) \rightarrow any()) \tau$
 $\langle proof \rangle$

lemma *cp-OclForall*:
 $(S \rightarrow forAll(x \mid P x)) \tau = ((\lambda -. S \tau) \rightarrow forAll(x \mid P (\lambda -. x \tau))) \tau$
 $\langle proof \rangle$

lemma *cp-OclForall1* [*simp,intro!*]:
 $cp S \implies cp (\lambda X. ((S X) \rightarrow forAll(x \mid P x)))$
 $\langle proof \rangle$

lemma
 $cp (\lambda X St x. P (\lambda \tau. x) X St) \implies cp S \implies cp (\lambda X. (S X) \rightarrow forAll(x \mid P x X))$
 $\langle proof \rangle$

lemma
 $cp S \implies$
 $(\bigwedge x. cp(P x)) \implies$
 $cp(\lambda X. ((S X) \rightarrow forAll(x \mid P x X)))$
 $\langle proof \rangle$

lemma *cp-OclExists*:
 $(S \rightarrow exists(x \mid P x)) \tau = ((\lambda -. S \tau) \rightarrow exists(x \mid P (\lambda -. x \tau))) \tau$
 $\langle proof \rangle$

lemma *cp-OclExists1* [*simp,intro!*]:
 $cp S \implies cp (\lambda X. ((S X) \rightarrow exists(x \mid P x)))$
 $\langle proof \rangle$

lemma *cp-OclIterate*: $(X \rightarrow iterate(a; x = A \mid P a x)) \tau =$
 $((\lambda -. X \tau) \rightarrow iterate(a; x = A \mid P a x)) \tau$
 $\langle proof \rangle$

lemma *cp-OclSelect*: $(X \rightarrow select(a \mid P a)) \tau =$
 $((\lambda -. X \tau) \rightarrow select(a \mid P a)) \tau$
 $\langle proof \rangle$

lemma *cp-OclReject*: $(X \rightarrow reject(a \mid P a)) \tau =$
 $((\lambda -. X \tau) \rightarrow reject(a \mid P a)) \tau$
 $\langle proof \rangle$

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

```

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

```

Const

```

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

```

5.8.5. Strict Equality

Definition

After the part of foundational operations on sets, we detail here equality on sets. Strong equality is inherited from the OCL core, but we have to consider the case of the strict equality. We decide to overload strict equality in the same way we do for other value's in OCL:

```

defs StrictRefEqSet :
  (x::('A,'α::null)Set)  $\doteq$  y  $\equiv$   $\lambda \tau$ . if (v x)  $\tau$  = true  $\tau \wedge$  (v y)  $\tau$  = true  $\tau$ 
    then (x  $\triangleq$  y) $\tau$ 
    else invalid  $\tau$ 

```

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

Property proof in terms of *profile-bin3*

```

interpretation StrictRefEqSet : profile-bin3  $\lambda x y$ . (x::('A,'α::null)Set)  $\doteq$  y
  <proof>

```

Execution Rules on OclIncluding

```

lemma OclIncluding-finite-rep-set :
  assumes X-def :  $\tau \models \delta X$ 
    and x-val :  $\tau \models v x$ 
  shows finite [[Rep-Setbase (X->including(x)  $\tau$ )] ] = finite [[Rep-Setbase (X  $\tau$ )] ]
  <proof>

```

```

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

```

shows $\llbracket \text{Rep-Set}_{base} (S \rightarrow \text{including}(\lambda x. \llbracket x \rrbracket) \tau) \rrbracket = \text{insert } \llbracket x \rrbracket \llbracket \text{Rep-Set}_{base} (S \tau) \rrbracket$
 $\langle \text{proof} \rangle$

lemma *OclIncluding-notempty-rep-set*:

assumes $X\text{-def}: \tau \models \delta X$
and $a\text{-val}: \tau \models v a$
shows $\llbracket \text{Rep-Set}_{base} (X \rightarrow \text{including}(a) \tau) \rrbracket \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *OclIncluding-includes0*:

assumes $\tau \models X \rightarrow \text{includes}(x)$
shows $X \rightarrow \text{including}(x) \tau = X \tau$
 $\langle \text{proof} \rangle$

lemma *OclIncluding-includes*:

assumes $\tau \models X \rightarrow \text{includes}(x)$
shows $\tau \models X \rightarrow \text{including}(x) \triangleq X$
 $\langle \text{proof} \rangle$

lemma *OclIncluding-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 $\tau \models ((S :: (\mathfrak{A}, 'a::\text{null}) \text{ Set}) \rightarrow \text{including}(i) \rightarrow \text{including}(j)) \triangleq$
 $(S \rightarrow \text{including}(j) \rightarrow \text{including}(i))$
 $\langle \text{proof} \rangle$

lemma *OclIncluding-commute[simp,code-unfold]*:

$((S :: (\mathfrak{A}, 'a::\text{null}) \text{ Set}) \rightarrow \text{including}(i) \rightarrow \text{including}(j)) = (S \rightarrow \text{including}(j) \rightarrow \text{including}(i))$
 $\langle \text{proof} \rangle$

Execution Rules on OclExcluding

lemma *OclExcluding-finite-rep-set* :

assumes $X\text{-def} : \tau \models \delta X$
and $x\text{-val} : \tau \models v x$
shows $\text{finite } \llbracket \text{Rep-Set}_{base} (X \rightarrow \text{excluding}(x) \tau) \rrbracket = \text{finite } \llbracket \text{Rep-Set}_{base} (X \tau) \rrbracket$
 $\langle \text{proof} \rangle$

lemma *OclExcluding-rep-set*:

assumes $S\text{-def}: \tau \models \delta S$
shows $\llbracket \text{Rep-Set}_{base} (S \rightarrow \text{excluding}(\lambda x. \llbracket x \rrbracket) \tau) \rrbracket = \llbracket \text{Rep-Set}_{base} (S \tau) \rrbracket - \{\llbracket x \rrbracket\}$
 $\langle \text{proof} \rangle$

lemma *OclExcluding-excludes0*:

assumes $\tau \models X \rightarrow \text{excludes}(x)$
shows $X \rightarrow \text{excluding}(x) \tau = X \tau$
 $\langle \text{proof} \rangle$

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

lemma *OclExcluding-cha0[simp]*:
assumes $\text{val-}x:\tau \models (v\ x)$
shows $\tau \models ((\text{Set}\{\}\rightarrow \text{excluding}(x)) \triangleq \text{Set}\{\})$
 $\langle \text{proof} \rangle$

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 $\tau \models ((S :: ('A, 'a::\text{null})\ \text{Set}) \rightarrow \text{excluding}(i) \rightarrow \text{excluding}(j)) \triangleq$
 $(S \rightarrow \text{excluding}(j) \rightarrow \text{excluding}(i)))$
 $\langle \text{proof} \rangle$

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

lemma *OclExcluding-cha0-exec[simp,code-unfold]*:
 $(\text{Set}\{\} \rightarrow \text{excluding}(x)) = (\text{if } (v\ x) \text{ then } \text{Set}\{\} \text{ else } \text{invalid endif})$
 $\langle \text{proof} \rangle$

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 $\tau \models ((X \rightarrow \text{including}(x)) \rightarrow \text{excluding}(y)) \triangleq ((X \rightarrow \text{excluding}(y)) \rightarrow \text{including}(x))$
 $\langle \text{proof} \rangle$

lemma *OclExcluding-cha2*:
assumes $\text{def-}X:\tau \models (\delta\ X)$
and $\text{val-}x:\tau \models (v\ x)$
shows $\tau \models (((X \rightarrow \text{including}(x)) \rightarrow \text{excluding}(x)) \triangleq (X \rightarrow \text{excluding}(x)))$
 $\langle \text{proof} \rangle$

theorem *OclExcluding-cha3*: $((X \rightarrow \text{including}(x)) \rightarrow \text{excluding}(x)) = (X \rightarrow \text{excluding}(x))$

<proof>

One would like a generic theorem of the form:

lemma *OclExcluding_chn_exec*:

```

”(X->including(x::('A,'a::null)val)->excluding(y)) =
  (if  $\delta$  X then if  $x \doteq y$ 
    then X->excluding(y)
    else X->excluding(y)->including(x)
  endif
  else invalid endif)”

```

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

lemma *OclExcluding-chn-exec*:

```

assumes strict1: (invalid  $\doteq$  y) = invalid
and strict2: ( $x \doteq$  invalid) = invalid
and StrictRefEq-valid-args-valid:  $\bigwedge (x::('A,'a::null)val) y \tau.$ 
                                     ( $\tau \models \delta (x \doteq y)$ ) = (( $\tau \models (v x)$ )  $\wedge$  ( $\tau \models v y$ ))
and cp-StrictRefEq:  $\bigwedge (X::('A,'a::null)val) Y \tau. (X \doteq Y) \tau = ((\lambda \cdot. X \tau) \doteq (\lambda \cdot. Y \tau)) \tau$ 
and StrictRefEq-vs-StrongEq:  $\bigwedge (x::('A,'a::null)val) y \tau.$ 
                                      $\tau \models v x \implies \tau \models v y \implies (\tau \models ((x \doteq y) \triangleq (x \triangleq y)))$ 
shows (X->including(x::('A,'a::null)val)->excluding(y)) =
  (if  $\delta$  X then if  $x \doteq y$ 
    then X->excluding(y)
    else X->excluding(y)->including(x)
  endif
  else invalid endif)

```

<proof>

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

<proof>

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

<proof>

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

<proof>

Execution Rules on OclIncludes

lemma *OclIncludes-cha0*[simp]:
assumes $val\text{-}x:\tau \models (v\ x)$
shows $\tau \models not(Set\{\}\text{-}>includes(x))$
 $\langle proof \rangle$

lemma *OclIncludes-cha0'*[simp,code-unfold]:
 $Set\{\}\text{-}>includes(x) = (if\ v\ x\ then\ false\ else\ invalid\ endif)$
 $\langle proof \rangle$

lemma *OclIncludes-cha1*:
assumes $def\text{-}X:\tau \models (\delta\ X)$
assumes $val\text{-}x:\tau \models (v\ x)$
shows $\tau \models (X\text{-}>including(x)\text{-}>includes(x))$
 $\langle proof \rangle$

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 $\tau \models (X\text{-}>including(x)\text{-}>includes(y)) \triangleq (X\text{-}>includes(y))$
 $\langle proof \rangle$

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 $cp\text{-}StrictRefEq: \bigwedge (X::('A,'a::null)val)\ Y\ \tau. (X \doteq Y)\ \tau = ((\lambda\cdot. X\ \tau) \doteq (\lambda\cdot. Y\ \tau))\ \tau$
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
 $(X\text{-}>including(x::('A,'a::null)val)\text{-}>includes(y)) =$
 $(if\ \delta\ X\ then\ if\ x \doteq y\ then\ true\ else\ X\text{-}>includes(y)\ endif\ else\ invalid\ endif)$
 $\langle proof \rangle$

schematic-lemma *OclIncludes-execute_{Integer}*[simp,code-unfold]: $?X$
 $\langle proof \rangle$

schematic-lemma *OclIncludes-execute_{Boolean}*[simp,code-unfold]: $?X$
 $\langle proof \rangle$

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

lemma *OclIncludes-including-generic* :
assumes *OclIncludes-execute-generic* [simp] : $\bigwedge X x y.$
 $(X \rightarrow \text{including}(x::(\mathfrak{A}, 'a::\text{null})\text{val}) \rightarrow \text{includes}(y)) =$
 $(\text{if } \delta X \text{ then if } x \doteq y \text{ then true else } X \rightarrow \text{includes}(y) \text{ endif else invalid endif})$
and *StrictRefEq-strict''* : $\bigwedge x y. \delta ((x::(\mathfrak{A}, 'a::\text{null})\text{val}) \doteq y) = (v(x) \text{ and } v(y))$
and *a-val* : $\tau \models v a$
and *x-val* : $\tau \models v x$
and *S-incl* : $\tau \models (S) \rightarrow \text{includes}((x::(\mathfrak{A}, 'a::\text{null})\text{val}))$
shows $\tau \models S \rightarrow \text{including}((a::(\mathfrak{A}, 'a::\text{null})\text{val})) \rightarrow \text{includes}(x)$
 ⟨proof⟩

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

Execution Rules on OclExcludes

lemma *OclExcludes-charn1*:
assumes *def-X*: $\tau \models (\delta X)$
assumes *val-x*: $\tau \models (v x)$
shows $\tau \models (X \rightarrow \text{excluding}(x) \rightarrow \text{excludes}(x))$
 ⟨proof⟩

Execution Rules on OclSize

lemma [simp,code-unfold]: *Set*{ } $\rightarrow \text{size}() = 0$
 ⟨proof⟩

lemma *OclSize-including-exec*[simp,code-unfold]:
 $((X \rightarrow \text{including}(x)) \rightarrow \text{size}()) = (\text{if } \delta X \text{ and } v x \text{ then}$
 $X \rightarrow \text{size}() +_{int} \text{if } X \rightarrow \text{includes}(x) \text{ then } 0 \text{ else } 1 \text{ endif}$
 else
 invalid
 $\text{endif})$
 ⟨proof⟩

Execution Rules on OclIsEmpty

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

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

Execution Rules on OclNotEmpty

lemma $[simp, code-unfold]: Set\{\} \rightarrow notEmpty() = false$
 $\langle proof \rangle$

lemma $OclNotEmpty-including [simp, code-unfold]:$
assumes $X-def: \tau \models \delta X$
and $X-finite: finite \ [\ Rep-Set_{base} (X \ \tau)]$
and $a-val: \tau \models v \ a$
shows $X \rightarrow including(a) \rightarrow notEmpty() \ \tau = true \ \tau$
 $\langle proof \rangle$

Execution Rules on OclANY

lemma $[simp, code-unfold]: Set\{\} \rightarrow any() = null$
 $\langle proof \rangle$

lemma $OclANY-singleton-exec[simp, code-unfold]:$
 $(Set\{\} \rightarrow including(a) \rightarrow any() = a$
 $\langle proof \rangle$

Execution Rules on OclForall

lemma $OclForall-mtSet-exec[simp, code-unfold] : ((Set\{\}) \rightarrow forAll(z \mid P(z))) = true$
 $\langle proof \rangle$

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:

$$Integer_{null} \rightarrow forAll(x \mid Integer_{null} \rightarrow forAll(y \mid x +_{int} y \triangleq y +_{int} x))$$

or even:

$$Integer \rightarrow forAll(x \mid Integer \rightarrow forAll(y \mid x +_{int} y \doteq y +_{int} x))$$

are valid OCL statements in any context τ .

theorem $OclForall-including-exec[simp, code-unfold] :$
assumes $cp0 : cp \ P$
shows $((S \rightarrow including(x)) \rightarrow forAll(z \mid P(z))) = (if \ \delta \ S \ and \ v \ x$
 $then \ P \ x \ and \ (S \rightarrow forAll(z \mid P(z)))$
 $else \ invalid$
 $endif)$
 $\langle proof \rangle$

Execution Rules on OclExists

lemma $OclExists-mtSet-exec[simp, code-unfold] :$
 $((Set\{\}) \rightarrow exists(z \mid P(z))) = false$

$\langle proof \rangle$

lemma *OclExists-including-exec*[simp,code-unfold] :
assumes *cp*: $cp\ P$
shows $((S \rightarrow including(x)) \rightarrow exists(z \mid P(z))) = (if\ \delta\ S\ and\ v\ x$
 $\quad then\ P\ x\ or\ (S \rightarrow exists(z \mid P(z)))$
 $\quad else\ invalid$
 $\quad endif)$

$\langle proof \rangle$

Execution Rules on OclIterate

lemma *OclIterate-empty*[simp,code-unfold]: $((Set\{\}) \rightarrow iterate(a; x = A \mid P\ a\ x)) = A$
 $\langle proof \rangle$

In particular, this does hold for $A = \text{null}$.

lemma *OclIterate-including*:
assumes *S-finite*: $\tau \models \delta(S \rightarrow size())$
and *F-valid-arg*: $(v\ A)\ \tau = (v\ (F\ a\ A))\ \tau$
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 $((S \rightarrow including(a)) \rightarrow iterate(a; x = A \mid F\ a\ x))\ \tau =$
 $((S \rightarrow excluding(a)) \rightarrow iterate(a; x = F\ a\ A \mid F\ a\ x))\ \tau$
 $\langle proof \rangle$

Execution Rules on OclSelect

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

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

theorem *OclSelect-including-exec*[simp,code-unfold]:
assumes *P-cp* : $cp\ P$
shows $OclSelect\ (X \rightarrow including(y))\ P = OclSelect\ body\ P\ y\ (OclSelect\ (X \rightarrow excluding(y))\ P)$
 $(is\ - = ?select)$
 $\langle proof \rangle$

Execution Rules on OclReject

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

lemma *OclReject-including-exec*[simp,code-unfold]:
assumes *P-cp* : $cp\ P$
shows $OclReject\ (X \rightarrow including(y))\ P = OclSelect\ body\ (not\ o\ P)\ y\ (OclReject\ (X \rightarrow excluding(y))\ P)$
 $\langle proof \rangle$

Execution Rules Combining Previous Operators

OclIncluding

lemma *OclIncluding-idem0* :

assumes $\tau \models \delta S$

and $\tau \models v i$

shows $\tau \models (S \rightarrow \text{including}(i) \rightarrow \text{including}(i)) \triangleq (S \rightarrow \text{including}(i))$

<proof>

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

<proof>

OclExcluding

lemma *OclExcluding-idem0* :

assumes $\tau \models \delta S$

and $\tau \models v i$

shows $\tau \models (S \rightarrow \text{excluding}(i) \rightarrow \text{excluding}(i)) \triangleq (S \rightarrow \text{excluding}(i))$

<proof>

theorem *OclExcluding-idem[simp,code-unfold]*: $((S \rightarrow \text{excluding}(i)) \rightarrow \text{excluding}(i)) = (S \rightarrow \text{excluding}(i))$

<proof>

OclIncludes

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

$X \rightarrow \text{includes}(X \rightarrow \text{any}()) = (\text{if } \delta X \text{ then}$

$\text{if } \delta (X \rightarrow \text{size}()) \text{ then } \text{not}(X \rightarrow \text{isEmpty}())$

$\text{else } X \rightarrow \text{includes}(\text{null}) \text{ endif}$

$\text{else invalid endif})$

<proof>

OclSize

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

<proof>

lemma *[simp,code-unfold]*: $\delta ((X \rightarrow \text{including}(x)) \rightarrow \text{size}()) = (\delta (X \rightarrow \text{size}()) \text{ and } v(x))$

<proof>

lemma *[simp,code-unfold]*: $\delta ((X \rightarrow \text{excluding}(x)) \rightarrow \text{size}()) = (\delta (X \rightarrow \text{size}()) \text{ and } v(x))$

<proof>

lemma *[simp]*:

assumes $X\text{-finite}; \bigwedge \tau. \text{finite } \llbracket \text{Rep-Set}_{\text{base}}(X \ \tau) \rrbracket$

shows $\delta ((X \rightarrow \text{including}(x)) \rightarrow \text{size}()) = (\delta (X) \text{ and } v(x))$

<proof>

OclForall

lemma *OclForall-rep-set-false:*

assumes $\tau \models \delta \ X$

shows $(\text{OclForall } X \ P \ \tau = \text{false } \tau) = (\exists x \in \llbracket \text{Rep-Set}_{\text{base}}(X \ \tau) \rrbracket. P \ (\lambda \tau. x) \ \tau = \text{false } \tau)$
 $\langle \text{proof} \rangle$

lemma *OclForall-rep-set-true:*

assumes $\tau \models \delta \ X$

shows $(\tau \models \text{OclForall } X \ P) = (\forall x \in \llbracket \text{Rep-Set}_{\text{base}}(X \ \tau) \rrbracket. \tau \models P \ (\lambda \tau. x))$
 $\langle \text{proof} \rangle$

lemma *OclForall-includes :*

assumes $x\text{-def} : \tau \models \delta \ x$

and $y\text{-def} : \tau \models \delta \ y$

shows $(\tau \models \text{OclForall } x \ (\text{OclIncludes } y)) = (\llbracket \text{Rep-Set}_{\text{base}}(x \ \tau) \rrbracket \subseteq \llbracket \text{Rep-Set}_{\text{base}}(y \ \tau) \rrbracket)$
 $\langle \text{proof} \rangle$

lemma *OclForall-not-includes :*

assumes $x\text{-def} : \tau \models \delta \ x$

and $y\text{-def} : \tau \models \delta \ y$

shows $(\text{OclForall } x \ (\text{OclIncludes } y) \ \tau = \text{false } \tau) = (\neg \llbracket \text{Rep-Set}_{\text{base}}(x \ \tau) \rrbracket \subseteq \llbracket \text{Rep-Set}_{\text{base}}(y \ \tau) \rrbracket)$
 $\langle \text{proof} \rangle$

lemma *OclForall-iterate:*

assumes $S\text{-finite: finite } \llbracket \text{Rep-Set}_{\text{base}}(S \ \tau) \rrbracket$

shows $S \rightarrow \text{forAll}(x \mid P \ x) \ \tau = (S \rightarrow \text{iterate}(x; \text{acc} = \text{true} \mid \text{acc and } P \ x)) \ \tau$
 $\langle \text{proof} \rangle$

lemma *OclForall-cong:*

assumes $\bigwedge x. x \in \llbracket \text{Rep-Set}_{\text{base}}(X \ \tau) \rrbracket \implies \tau \models P \ (\lambda \tau. x) \implies \tau \models Q \ (\lambda \tau. x)$

assumes $P: \tau \models \text{OclForall } X \ P$

shows $\tau \models \text{OclForall } X \ Q$

$\langle \text{proof} \rangle$

lemma *OclForall-cong':*

assumes $\bigwedge x. x \in \llbracket \text{Rep-Set}_{\text{base}}(X \ \tau) \rrbracket \implies \tau \models P \ (\lambda \tau. x) \implies \tau \models Q \ (\lambda \tau. x) \implies \tau \models R \ (\lambda \tau. x)$

assumes $P: \tau \models \text{OclForall } X \ P$

assumes $Q: \tau \models \text{OclForall } X \ Q$

shows $\tau \models \text{OclForall } X \ R$

$\langle \text{proof} \rangle$

Strict Equality

lemma *StrictRefEqSet-defined :*

assumes $x\text{-def}: \tau \models \delta \ x$

assumes $y\text{-def}: \tau \models \delta \ y$

shows $((x::(\mathfrak{A}, \alpha::\text{null})\text{Set}) \doteq y) \ \tau =$
 $(x \rightarrow \text{forAll}(z \mid y \rightarrow \text{includes}(z)) \text{ and } (y \rightarrow \text{forAll}(z \mid x \rightarrow \text{includes}(z)))) \ \tau$

$\langle \text{proof} \rangle$

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

$((x :: ('A, 'a :: \text{null}) \text{Set}) \doteq y) =$
 $(\text{if } \delta x \text{ then } (\text{if } \delta y$
 $\quad \text{then } ((x \rightarrow \text{forAll}(z \mid y \rightarrow \text{includes}(z)) \text{ and } (y \rightarrow \text{forAll}(z \mid x \rightarrow \text{includes}(z))))$
 $\quad \text{else if } v y$
 $\quad \quad \text{then false } (* x' \rightarrow \text{includes} = \text{null} *)$
 $\quad \quad \text{else invalid}$
 $\quad \quad \text{endif})$
 $\quad \text{endif})$
 $\text{else if } v x (* \text{null} = ??? *)$
 $\quad \text{then if } v y \text{ then not}(\delta y) \text{ else invalid endif}$
 $\quad \text{else invalid}$
 $\quad \text{endif}$
 $\text{endif})$

$\langle \text{proof} \rangle$

lemma *StrictRefEq_{Set}-L-subst1* : $cp P \implies \tau \models v x \implies \tau \models v y \implies \tau \models v P x \implies \tau \models v P y \implies$

$\tau \models (x :: ('A, 'a :: \text{null}) \text{Set}) \doteq y \implies \tau \models (P x :: ('A, 'a :: \text{null}) \text{Set}) \doteq P y$

$\langle \text{proof} \rangle$

lemma *OclIncluding-cong'* :

shows $\tau \models \delta s \implies \tau \models \delta t \implies \tau \models v x \implies$

$\tau \models ((s :: ('A, 'a :: \text{null}) \text{Set}) \doteq t) \implies \tau \models (s \rightarrow \text{including}(x) \doteq (t \rightarrow \text{including}(x)))$

$\langle \text{proof} \rangle$

lemma *OclIncluding-cong* : $\bigwedge (s :: ('A, 'a :: \text{null}) \text{Set}) t x y \tau. \tau \models \delta t \implies \tau \models v y \implies$

$\tau \models s \doteq t \implies x = y \implies \tau \models s \rightarrow \text{including}(x) \doteq (t \rightarrow \text{including}(y))$

$\langle \text{proof} \rangle$

lemma *const-StrictRefEq_{Set}-empty* : $\text{const } X \implies \text{const } (X \doteq \text{Set}\{\})$

$\langle \text{proof} \rangle$

lemma *const-StrictRefEq_{Set}-including* :

$\text{const } a \implies \text{const } S \implies \text{const } X \implies \text{const } (X \doteq S \rightarrow \text{including}(a))$

$\langle \text{proof} \rangle$

5.8.6. Test Statements

Assert $(\tau \models (\text{Set}\{\lambda \cdot \lfloor x \rfloor\} \doteq \text{Set}\{\lambda \cdot \lfloor x \rfloor\}))$

Assert $(\tau \models (\text{Set}\{\lambda \cdot \lfloor x \rfloor\} \doteq \text{Set}\{\lambda \cdot \lfloor x \rfloor\}))$

end

```

theory  UML-Sequence
imports ../basic-types/ UML-Boolean
          ../basic-types/ UML-Integer
begin

```

5.9. Collection Type Sequence: Operations

5.9.1. Constants: mtSequence

```

definition mtSequence :: ('A, 'α::null) Sequence (Sequence{})
where      Sequence{} ≡ (λ τ. Abs-Sequencebase [[]::'α list])

```

```

declare mtSequence-def[code-unfold]

```

```

lemma mtSequence-defined[simp,code-unfold]:δ(Sequence{}) = true
<proof>

```

```

lemma mtSequence-valid[simp,code-unfold]:v(Sequence{}) = true
<proof>

```

```

lemma mtSequence-rep-set: [[Rep-Sequencebase (Sequence{}) τ]] = []
<proof>

```

```

lemma [simp,code-unfold]: const Sequence{}
<proof>

```

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

```

lemmas cp-intro''Sequence[intro!,simp,code-unfold] = cp-intro'

```

Properties of Sequence Type:

Every element in a defined sequence is valid.

```

lemma Sequence-inv-lemma: τ ⊨ (δ X) ⇒ ∀ x∈set [[Rep-Sequencebase (X τ)]] . x ≠ bot
<proof>

```

5.9.2. Strict Equality

Definition

After the part of foundational operations on sets, we detail here equality on sets. Strong equality is inherited from the OCL core, but we have to consider the case of the strict equality. We decide to overload strict equality in the same way we do for other value's in OCL:

```

defs StrictRefEqSequence [code-unfold]:
  ((x::('A,'α::null)Sequence) ≐ y) ≡ (λ τ. if (v x) τ = true τ ∧ (v y) τ = true τ
      then (x ≐ y)τ
      else invalid τ)

```

Property proof in terms of *profile-bin3*

interpretation *StrictRefEqSequence* : *profile-bin3* $\lambda x y. (x::('A, 'a::null)Sequence) \doteq y$
 $\langle proof \rangle$

5.9.3. Standard Operations

Definition: including

definition *OclIncluding* :: $[('A, 'a::null) Sequence, ('A, 'a) val] \Rightarrow ('A, 'a) Sequence$
where $OclIncluding\ x\ y = (\lambda\ \tau. \text{if } (\delta\ x)\ \tau = \text{true}\ \tau \wedge (v\ y)\ \tau = \text{true}\ \tau$
 $\quad \text{then } Abs\text{-}Sequence_{base} \ll \ll Rep\text{-}Sequence_{base}\ (x\ \tau) \rr \ @\ [y\ \tau] \rr$
 $\quad \text{else } invalid\ \tau)$
notation *OclIncluding* $(-->including_{Seq}'(-))$

interpretation *OclIncluding* :
 $profile\text{-}bin2\ OclIncluding\ \lambda x y. Abs\text{-}Sequence_{base} \ll \ll Rep\text{-}Sequence_{base}\ x \rr \ @\ [y] \rr$
 $\langle proof \rangle$

syntax

$-OclFinsequence :: args \Rightarrow ('A, 'a::null) Sequence \quad (Sequence\{(-)\})$

translations

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

typ *int*

typ *num*

Definition: excluding

Definition: union

Definition: append

identical to including

Definition: prepend

Definition: subSequence

Definition: at

Definition: first

Definition: last

Definition: asSet

instantiation *Sequence_base* :: $(equal)equal$

begin

definition *HOL.equal* $k\ l \longleftrightarrow (k::('a::equal)Sequence_{base}) = l$

instance $\langle proof \rangle$

end

lemma *equal-Sequence_{base}-code* [code]:

$HOL.equal\ k\ (l::('a::\{equal,null\})Sequence_{base}) \longleftrightarrow Rep-Sequence_{base}\ k = Rep-Sequence_{base}\ l$
 $\langle proof \rangle$

5.9.4. Test Statements

Assert $(\tau \models (Sequence\{\} \doteq Sequence\{\}))$

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

end

theory *UML-Library*

imports

basic-types / UML-Boolean

basic-types / UML-Void

basic-types / UML-Integer

basic-types / UML-Real

basic-types / UML-String

collection-types / UML-Pair

collection-types / UML-Set

collection-types / UML-Sequence

begin

5.10. Miscellaneous Stuff

5.10.1. Properties on Collection Types: Strict Equality

The structure of this chapter roughly follows the structure of Chapter 10 of the OCL standard [28], which introduces the OCL Library.

5.10.2. MOVE TEXT : Collection Types

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

1. we want the types to be *fully abstract*, i.e., the type should not contain junk-elements that are not representable by OCL expressions, and

2. we want a possibility to nest collection types (so, we want the potential to talking about $Set(Set(Sequences(Pairs(X, Y))))$).

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

```

lemmas cp-intro'' [intro!, simp, code-unfold] =
  cp-intro'

  cp-intro''Set
  cp-intro''Sequence

```

5.10.3. MOVE TEXT: Test Statements

```

lemma syntax-test:  $Set\{2,1\} = (Set\{\} \rightarrow including(1) \rightarrow including(2))$ 
<proof>

```

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

```

lemma semantic-test2:
assumes  $H: (Set\{2\} \doteq null) = (false::(\mathfrak{A}) Boolean)$ 
shows  $(\tau::(\mathfrak{A}) st) \models (Set\{Set\{2\}, null\} \rightarrow includes(null))$ 
<proof>

```

```

lemma short-cut' [simp, code-unfold]:  $(8 \doteq 6) = false$ 
<proof>

```

```

lemma short-cut'' [simp, code-unfold]:  $(2 \doteq 1) = false$ 
<proof>

```

```

lemma short-cut''' [simp, code-unfold]:  $(1 \doteq 2) = false$ 
<proof>

```

Elementary computations on Sets.

```

declare OclSelect-body-def [simp]

```

```

Assert  $\neg (\tau \models v(invalid::(\mathfrak{A}, '\alpha::null) Set))$ 
Assert  $\tau \models v(null::(\mathfrak{A}, '\alpha::null) Set)$ 
Assert  $\neg (\tau \models \delta(null::(\mathfrak{A}, '\alpha::null) Set))$ 
Assert  $\tau \models v(Set\{\})$ 
Assert  $\tau \models v(Set\{Set\{2\}, null\})$ 
Assert  $\tau \models \delta(Set\{Set\{2\}, null\})$ 
Assert  $\tau \models (Set\{2,1\} \rightarrow includes(1))$ 

```

```

Assert  $\neg (\tau \models (\text{Set}\{\mathbf{2}\} \rightarrow \text{includes}(\mathbf{1})))$ 
Assert  $\neg (\tau \models (\text{Set}\{\mathbf{2}, \mathbf{1}\} \rightarrow \text{includes}(\text{null})))$ 
Assert  $\tau \models (\text{Set}\{\mathbf{2}, \text{null}\} \rightarrow \text{includes}(\text{null}))$ 
Assert  $\tau \models (\text{Set}\{\text{null}, \mathbf{2}\} \rightarrow \text{includes}(\text{null}))$ 

Assert  $\tau \models ((\text{Set}\{\}) \rightarrow \text{forAll}(z \mid \mathbf{0} <_{\text{int}} z))$ 

Assert  $\tau \models ((\text{Set}\{\mathbf{2}, \mathbf{1}\}) \rightarrow \text{forAll}(z \mid \mathbf{0} <_{\text{int}} z))$ 
Assert  $\tau \models (\mathbf{0} <_{\text{int}} \mathbf{2}) \text{ and } (\mathbf{0} <_{\text{int}} \mathbf{1})$ 
Assert  $\neg (\tau \models ((\text{Set}\{\mathbf{2}, \mathbf{1}\}) \rightarrow \text{exists}(z \mid z <_{\text{int}} \mathbf{0})))$ 
Assert  $\neg (\tau \models (\delta(\text{Set}\{\mathbf{2}, \text{null}\}) \rightarrow \text{forAll}(z \mid \mathbf{0} <_{\text{int}} z)))$ 
Assert  $\neg (\tau \models ((\text{Set}\{\mathbf{2}, \text{null}\}) \rightarrow \text{forAll}(z \mid \mathbf{0} <_{\text{int}} z)))$ 
Assert  $\tau \models ((\text{Set}\{\mathbf{2}, \text{null}\}) \rightarrow \text{exists}(z \mid \mathbf{0} <_{\text{int}} z))$ 

Assert  $\neg (\tau \models (\text{Set}\{\text{null}::'a \text{ Boolean}\} \doteq \text{Set}\{\}))$ 
Assert  $\neg (\tau \models (\text{Set}\{\text{null}::'a \text{ Integer}\} \doteq \text{Set}\{\}))$ 

Assert  $\neg (\tau \models (\text{Set}\{\text{true}\} \doteq \text{Set}\{\text{false}\}))$ 
Assert  $\neg (\tau \models (\text{Set}\{\text{true}, \text{true}\} \doteq \text{Set}\{\text{false}\}))$ 
Assert  $\neg (\tau \models (\text{Set}\{\mathbf{2}\} \doteq \text{Set}\{\mathbf{1}\}))$ 
Assert  $\tau \models (\text{Set}\{\mathbf{2}, \text{null}, \mathbf{2}\} \doteq \text{Set}\{\text{null}, \mathbf{2}\})$ 
Assert  $\tau \models (\text{Set}\{\mathbf{1}, \text{null}, \mathbf{2}\} <> \text{Set}\{\text{null}, \mathbf{2}\})$ 
Assert  $\tau \models (\text{Set}\{\text{Set}\{\mathbf{2}, \text{null}\}\} \doteq \text{Set}\{\text{Set}\{\text{null}, \mathbf{2}\}\})$ 
Assert  $\tau \models (\text{Set}\{\text{Set}\{\mathbf{2}, \text{null}\}\} <> \text{Set}\{\text{Set}\{\text{null}, \mathbf{2}\}, \text{null}\})$ 
Assert  $\tau \models (\text{Set}\{\text{null}\} \rightarrow \text{select}(x \mid \text{not } x) \doteq \text{Set}\{\text{null}\})$ 
Assert  $\tau \models (\text{Set}\{\text{null}\} \rightarrow \text{reject}(x \mid \text{not } x) \doteq \text{Set}\{\text{null}\})$ 

lemma  $\text{const } (\text{Set}\{\text{Set}\{\mathbf{2}, \text{null}\}, \text{invalid}\}) \langle \text{proof} \rangle$ 

```

end

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

```
theory UML-State
imports UML-Library
begin
```

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

6.1. Introduction: States over Typed Object Universes

In the following, we will refine the concepts of a user-defined data-model (implied by a class-diagram) as well as the notion of state used in the previous section to much more detail. Surprisingly, even without a concrete notion of an objects and a universe of object representation, the generic infrastructure of state-related operations is fairly rich.

6.1.1. Fundamental Properties on Objects: Core Referential Equality

Definition

Generic referential equality - to be used for instantiations with concrete object types ...

```
definition StrictRefEqObject :: ('A, 'a :: {object, null}) val  $\Rightarrow$  ('A, 'a) val  $\Rightarrow$  ('A) Boolean
where      StrictRefEqObject x y
             $\equiv \lambda \tau. \text{if } (v\ x)\ \tau = \text{true}\ \tau \wedge (v\ y)\ \tau = \text{true}\ \tau$ 
               $\text{then if } x\ \tau = \text{null} \vee y\ \tau = \text{null}$ 
                 $\text{then } \llbracket x\ \tau = \text{null} \wedge y\ \tau = \text{null} \rrbracket$ 
                 $\text{else } \llbracket (\text{oid-of } (x\ \tau)) = (\text{oid-of } (y\ \tau)) \rrbracket$ 
               $\text{else invalid } \tau$ 
```

Strictness and context passing

```
lemma StrictRefEqObject-strict1[simp, code-unfold] :
  (StrictRefEqObject x invalid) = invalid
  <proof>
```

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

```
lemma cp-StrictRefEqObject:
  (StrictRefEqObject x y  $\tau$ ) = (StrictRefEqObject ( $\lambda\cdot. x\ \tau$ ) ( $\lambda\cdot. y\ \tau$ ))  $\tau$ 
```

$\langle proof \rangle$

lemmas $cp0\text{-}StrictRefEq_{Object} = cp\text{-}StrictRefEq_{Object}[THEN \quad all[THEN \quad all[THEN$
 $all[THEN \quad cpI2]],$
 $of \quad StrictRefEq_{Object}]]$

lemmas $cp\text{-}intro''[intro!,simp,code\text{-}unfold] =$
 $cp\text{-}intro''$
 $cp\text{-}StrictRefEq_{Object}[THEN \quad all[THEN \quad all[THEN \quad all[THEN \quad cpI2]],$
 $of \quad StrictRefEq_{Object}]]$

6.1.2. Logic and Algebraic Layer on Object

Validity and Definedness Properties

We derive the usual laws on definedness for (generic) object equality:

lemma $StrictRefEq_{Object}\text{-}defargs:$
 $\tau \models (StrictRefEq_{Object} \ x \ (y::(\mathfrak{A}, 'a::\{null, object\})val)) \implies (\tau \models (v \ x)) \wedge (\tau \models (v \ y))$
 $\langle proof \rangle$

lemma $defined\text{-}StrictRefEq_{Object}\text{-}I:$
assumes $val\text{-}x : \tau \models v \ x$
assumes $val\text{-}x : \tau \models v \ y$
shows $\tau \models \delta \ (StrictRefEq_{Object} \ x \ y)$
 $\langle proof \rangle$

lemma $StrictRefEq_{Object}\text{-}def\text{-}homo :$
 $\delta(StrictRefEq_{Object} \ x \ (y::(\mathfrak{A}, 'a::\{null, object\})val)) = ((v \ x) \text{ and } (v \ y))$
 $\langle proof \rangle$

Symmetry

lemma $StrictRefEq_{Object}\text{-}sym :$
assumes $x\text{-}val : \tau \models v \ x$
shows $\tau \models StrictRefEq_{Object} \ x \ x$
 $\langle proof \rangle$

Behavior vs StrongEq

It remains to clarify the role of the state invariant $inv_{\sigma}(\sigma)$ mentioned above that states the condition that there is a “one-to-one” correspondence between object representations and oid’s: $\forall oid \in \text{dom } \sigma. oid = \text{OidOf } \lceil \sigma(oid) \rceil$. This condition is also mentioned in [28, Annex A] and goes back to Richters [30]; however, we state this condition as an invariant on states rather than a global axiom. It can, therefore, not be taken for granted that an oid makes sense both in pre- and post-states of OCL expressions.

We capture this invariant in the predicate WFF :

definition $WFF :: (\mathfrak{A}::object)st \Rightarrow bool$

where $WFF \tau = ((\forall x \in \text{ran}(\text{heap}(\text{fst } \tau)). [\text{heap}(\text{fst } \tau) (\text{oid-of } x)] = x) \wedge$
 $(\forall x \in \text{ran}(\text{heap}(\text{snd } \tau)). [\text{heap}(\text{snd } \tau) (\text{oid-of } x)] = x))$

It turns out that WFF is a key-concept for linking strict referential equality to logical equality: in well-formed states (i.e. those states where the self (oid-of) field contains the pointer to which the object is associated to in the state), referential equality coincides with logical equality.

We turn now to the generic definition of referential equality on objects: Equality on objects in a state is reduced to equality on the references to these objects. As in HOL-OCL [5, 7], we will store the reference of an object inside the object in a (ghost) field. By establishing certain invariants (“consistent state”), it can be assured that there is a “one-to-one-correspondence” of objects to their references—and therefore the definition below behaves as we expect.

Generic Referential Equality enjoys the usual properties: (quasi) reflexivity, symmetry, transitivity, substitutivity for defined values. For type-technical reasons, for each concrete object type, the equality \doteq is defined by generic referential equality.

theorem *StrictRefEqObject-vs-StrongEq:*

assumes *WFF:* $WFF \tau$

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

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

and *x-present-pre:* $x \tau \in \text{ran} (\text{heap}(\text{fst } \tau))$

and *y-present-pre:* $y \tau \in \text{ran} (\text{heap}(\text{fst } \tau))$

and *x-present-post:* $x \tau \in \text{ran} (\text{heap}(\text{snd } \tau))$

and *y-present-post:* $y \tau \in \text{ran} (\text{heap}(\text{snd } \tau))$

shows $(\tau \models (\text{StrictRefEqObject } x \ y)) = (\tau \models (x \doteq y))$

<proof>

theorem *StrictRefEqObject-vs-StrongEq':*

assumes *WFF:* $WFF \tau$

and *valid-x:* $\tau \models (v \ (x :: (\mathfrak{A}::\text{object}, 'a::\{\text{null}, \text{object}\})\text{val}))$

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

and *oid-preserve:* $\bigwedge x. x \in \text{ran} (\text{heap}(\text{fst } \tau)) \vee x \in \text{ran} (\text{heap}(\text{snd } \tau)) \implies$

$H \ x \neq \perp \implies \text{oid-of } (H \ x) = \text{oid-of } x$

and *xy-together:* $x \tau \in H \text{ ' } \text{ran} (\text{heap}(\text{fst } \tau)) \wedge y \tau \in H \text{ ' } \text{ran} (\text{heap}(\text{fst } \tau)) \vee$

$x \tau \in H \text{ ' } \text{ran} (\text{heap}(\text{snd } \tau)) \wedge y \tau \in H \text{ ' } \text{ran} (\text{heap}(\text{snd } \tau))$

shows $(\tau \models (\text{StrictRefEqObject } x \ y)) = (\tau \models (x \doteq y))$

<proof>

So, if two object descriptions live in the same state (both pre or post), the referential equality on objects implies in a WFF state the logical equality.

6.2. Operations on Object

6.2.1. Initial States (for testing and code generation)

definition $\tau_0 :: ('A)st$
where $\tau_0 \equiv (\langle heap = Map.empty, assoc = Map.empty \rangle,$
 $\langle heap = Map.empty, assoc = Map.empty \rangle)$

6.2.2. OclAllInstances

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

definition $OclAllInstances-generic :: (('A::object) st \Rightarrow 'A state) \Rightarrow ('A::object \rightarrow 'A) \Rightarrow$
 $(('A, 'A option option) Set$
where $OclAllInstances-generic fst-snd H =$
 $(\lambda \tau. Abs-Set_{base} \ll Some \text{ ' } ((H \text{ ' } ran (heap (fst-snd \tau))) - \{ None \}) \ll)$

lemma $OclAllInstances-generic-defined: \tau \models \delta (OclAllInstances-generic pre-post H)$
 $\langle proof \rangle$

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

lemma $represented-generic-objects-nonnul:$
assumes $A: \tau \models ((OclAllInstances-generic pre-post (H::('A::object \rightarrow 'A))) \rightarrow includes(x))$
shows $\tau \models not(x \triangleq null)$
 $\langle proof \rangle$

lemma $represented-generic-objects-defined:$
assumes $A: \tau \models ((OclAllInstances-generic pre-post (H::('A::object \rightarrow 'A))) \rightarrow includes(x))$
shows $\tau \models \delta (OclAllInstances-generic pre-post H) \wedge \tau \models \delta x$
 $\langle proof \rangle$

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

lemma $represented-generic-objects-in-state:$
assumes $A: \tau \models (OclAllInstances-generic pre-post H) \rightarrow includes(x)$
shows $x \tau \in (Some o H) \text{ ' } ran (heap(pre-post \tau))$
 $\langle proof \rangle$

lemma $state-update-vs-allInstances-generic-empty:$
assumes $[simp]: \bigwedge a. pre-post (mk a) = a$
shows $(mk \langle heap = empty, assoc = A \rangle) \models OclAllInstances-generic pre-post Type \doteq Set\{\}$
 $\langle proof \rangle$

Here comes a couple of operational rules that allow to infer the value of `oclAllInstances`

from the context τ . These rules are a special-case in the sense that they are the only rules that relate statements with *different* τ 's. For that reason, new concepts like “constant contexts P” are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

lemma *state-update-vs-allInstances-generic-including'*:
assumes [simp]: $\bigwedge a. \text{pre-post } (mk \ a) = a$
assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and $\text{Type Object} \neq \text{None}$
shows $(\text{OclAllInstances-generic pre-post Type})$
 $(mk \ (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$
 $=$
 $((\text{OclAllInstances-generic pre-post Type}) \rightarrow \text{including}(\lambda \cdot. \llbracket \text{drop } (\text{Type Object}) \rrbracket))$
 $(mk \ (\text{heap}=\sigma', \text{assocs}=A))$
 $\langle \text{proof} \rangle$

lemma *state-update-vs-allInstances-generic-including*:
assumes [simp]: $\bigwedge a. \text{pre-post } (mk \ a) = a$
assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and $\text{Type Object} \neq \text{None}$
shows $(\text{OclAllInstances-generic pre-post Type})$
 $(mk \ (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$
 $=$
 $((\lambda \cdot. (\text{OclAllInstances-generic pre-post Type})$
 $(mk \ (\text{heap}=\sigma', \text{assocs}=A))) \rightarrow \text{including}(\lambda \cdot. \llbracket \text{drop } (\text{Type Object}) \rrbracket))$
 $(mk \ (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$
 $\langle \text{proof} \rangle$

lemma *state-update-vs-allInstances-generic-noincluding'*:
assumes [simp]: $\bigwedge a. \text{pre-post } (mk \ a) = a$
assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and $\text{Type Object} = \text{None}$
shows $(\text{OclAllInstances-generic pre-post Type})$
 $(mk \ (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$
 $=$
 $(\text{OclAllInstances-generic pre-post Type})$
 $(mk \ (\text{heap}=\sigma', \text{assocs}=A))$
 $\langle \text{proof} \rangle$

theorem *state-update-vs-allInstances-generic-ntc*:
assumes [simp]: $\bigwedge a. \text{pre-post } (mk \ a) = a$
assumes *oid-def*: $\text{oid} \notin \text{dom } \sigma'$
and *non-type-conform*: $\text{Type Object} = \text{None}$
and *cp-ctxt*: $cp \ P$
and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P \ X)$

shows $(mk \ (\!|heap=\sigma'(oid \mapsto Object), assoc s=A|) \models P \ (OclAllInstances-generic \ pre-post \ Type)) =$
 $(mk \ (\!|heap=\sigma', \ assoc s=A|) \models P \ (OclAllInstances-generic \ pre-post \ Type))$
is $(? \tau \models P \ ? \varphi) = (? \tau' \models P \ ? \varphi)$
 $\langle proof \rangle$

theorem *state-update-vs-allInstances-generic-tc*:

assumes $[simp]: \bigwedge a. \ pre-post \ (mk \ a) = a$

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 $(mk \ (\!|heap=\sigma'(oid \mapsto Object), assoc s=A|) \models P \ (OclAllInstances-generic \ pre-post \ Type)) =$
 $(mk \ (\!|heap=\sigma', \ assoc s=A|) \models P \ ((OclAllInstances-generic \ pre-post \ Type)$
 $\quad \rightarrow including(\lambda \ -. \ [(Type \ Object)])))$

is $(? \tau \models P \ ? \varphi) = (? \tau' \models P \ ? \varphi')$

$\langle proof \rangle$

declare *OclAllInstances-generic-def* $[simp]$

OclAllInstances (@post)

definition *OclAllInstances-at-post* :: $('A :: object \rightarrow 'a) \Rightarrow ('A, 'a \ option \ option) \ Set$
 $(- \ .allInstances'('a))$

where *OclAllInstances-at-post* = *OclAllInstances-generic snd*

lemma *OclAllInstances-at-post-defined*: $\tau \models \delta \ (H \ .allInstances())$

$\langle proof \rangle$

lemma $\tau_0 \models H \ .allInstances() \triangleq Set\{\}$

$\langle proof \rangle$

lemma *represented-at-post-objects-nonnull*:

assumes $A: \tau \models (((H::('A::object \rightarrow 'a)).allInstances()) \rightarrow includes(x))$

shows $\tau \models not(x \triangleq null)$

$\langle proof \rangle$

lemma *represented-at-post-objects-defined*:

assumes $A: \tau \models (((H::('A::object \rightarrow 'a)).allInstances()) \rightarrow includes(x))$

shows $\tau \models \delta \ (H \ .allInstances()) \wedge \tau \models \delta \ x$

$\langle proof \rangle$

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

lemma

assumes $A: \tau \models H \ .allInstances() \rightarrow includes(x)$

shows $x \in (Some \ o \ H) \ 'ran \ (heap(snd \ \tau))$

$\langle proof \rangle$

lemma *state-update-vs-allInstances-at-post-empty*:

shows $(\sigma, (\text{heap}=\text{empty}, \text{assocs}=A)) \models \text{Type} . \text{allInstances}() \doteq \text{Set}\{\}$
 <proof>

Here comes a couple of operational rules that allow to infer the value of `oclAllInstances` from the context τ . These rules are a special-case in the sense that they are the only rules that relate statements with *different* τ 's. For that reason, new concepts like “constant contexts P” are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

lemma *state-update-vs-allInstances-at-post-including'*:
assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and $\text{Type } \text{Object} \neq \text{None}$
shows $(\text{Type} . \text{allInstances}())$
 $(\sigma, (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$
 $=$
 $((\text{Type} . \text{allInstances}()) \rightarrow \text{including}(\lambda -. \llbracket \text{drop } (\text{Type } \text{Object}) \rrbracket))$
 $(\sigma, (\text{heap}=\sigma', \text{assocs}=A))$
 <proof>

lemma *state-update-vs-allInstances-at-post-including*:
assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and $\text{Type } \text{Object} \neq \text{None}$
shows $(\text{Type} . \text{allInstances}())$
 $(\sigma, (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$
 $=$
 $((\lambda -. (\text{Type} . \text{allInstances}())$
 $(\sigma, (\text{heap}=\sigma', \text{assocs}=A))) \rightarrow \text{including}(\lambda -. \llbracket \text{drop } (\text{Type } \text{Object}) \rrbracket))$
 $(\sigma, (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$
 <proof>

lemma *state-update-vs-allInstances-at-post-noincluding'*:
assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and $\text{Type } \text{Object} = \text{None}$
shows $(\text{Type} . \text{allInstances}())$
 $(\sigma, (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$
 $=$
 $(\text{Type} . \text{allInstances}())$
 $(\sigma, (\text{heap}=\sigma', \text{assocs}=A))$
 <proof>

theorem *state-update-vs-allInstances-at-post-ntc*:
assumes *oid-def*: $\text{oid} \notin \text{dom } \sigma'$
and *non-type-conform*: $\text{Type } \text{Object} = \text{None}$
and *cp-ctxt*: $\text{cp } P$
and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P X)$
shows $((\sigma, (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A)) \models (P(\text{Type} . \text{allInstances}())))) =$

$$\langle proof \rangle \quad ((\sigma, \langle heap=\sigma', assoc=A \rangle) \models (P(Type.allInstances())))$$

theorem *state-update-vs-allInstances-at-post-tc:*

assumes *oid-def:* $oid \notin dom \sigma'$

and *type-conform:* $Type \ Object \neq None$

and *cp-ctxt:* $cp \ P$

and *const-ctxt:* $\bigwedge X. const \ X \implies const \ (P \ X)$

shows $((\sigma, \langle heap=\sigma'(oid \mapsto Object), assoc=A \rangle) \models (P(Type.allInstances()))) =$
 $((\sigma, \langle heap=\sigma', assoc=A \rangle) \models (P((Type.allInstances())$
 $\quad \rightarrow including(\lambda -. [(Type \ Object)]))))$

$\langle proof \rangle$

OclAllInstances (@pre)

definition *OclAllInstances-at-pre* :: $('A :: object \rightarrow 'a) \Rightarrow ('A, 'a \ option \ option) \ Set$
 $(- .allInstances@pre('))$

where *OclAllInstances-at-pre* = *OclAllInstances-generic fst*

lemma *OclAllInstances-at-pre-defined:* $\tau \models \delta \ (H .allInstances@pre())$

$\langle proof \rangle$

lemma $\tau_0 \models H .allInstances@pre() \triangleq Set\{\}$

$\langle proof \rangle$

lemma *represented-at-pre-objects-nonnul:*

assumes *A:* $\tau \models (((H::('A::object \rightarrow 'a)).allInstances@pre()) \rightarrow includes(x))$

shows $\tau \models not(x \triangleq null)$

$\langle proof \rangle$

lemma *represented-at-pre-objects-defined:*

assumes *A:* $\tau \models (((H::('A::object \rightarrow 'a)).allInstances@pre()) \rightarrow includes(x))$

shows $\tau \models \delta \ (H .allInstances@pre()) \wedge \tau \models \delta \ x$

$\langle proof \rangle$

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

lemma

assumes *A:* $\tau \models H .allInstances@pre() \rightarrow includes(x)$

shows $x \ \tau \in (Some \ o \ H) \ 'ran \ (heap(fst \ \tau))$

$\langle proof \rangle$

lemma *state-update-vs-allInstances-at-pre-empty:*

shows $((\langle heap=empty, assoc=A \rangle, \sigma) \models Type.allInstances@pre() \doteq Set\{\})$

$\langle proof \rangle$

Here comes a couple of operational rules that allow to infer the value of *oclAllInstances@pre* from the context τ . These rules are a special-case in the sense that they are the only rules that relate statements with *different* τ 's. For that reason, new

concepts like “constant contexts P” are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

lemma *state-update-vs-allInstances-at-pre-including'*:
assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and $\text{Type Object} \neq \text{None}$
shows $(\text{Type .allInstances@pre}())$
 $(\llbracket \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \rrbracket, \sigma)$
 $=$
 $((\text{Type .allInstances@pre}()) \text{--> including}(\lambda -. \llbracket \text{drop}(\text{Type Object}) \rrbracket))$
 $(\llbracket \text{heap} = \sigma', \text{assocs} = A \rrbracket, \sigma)$
 $\langle \text{proof} \rangle$

lemma *state-update-vs-allInstances-at-pre-including*:
assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and $\text{Type Object} \neq \text{None}$
shows $(\text{Type .allInstances@pre}())$
 $(\llbracket \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \rrbracket, \sigma)$
 $=$
 $((\lambda -. (\text{Type .allInstances@pre}())$
 $(\llbracket \text{heap} = \sigma', \text{assocs} = A \rrbracket, \sigma)) \text{--> including}(\lambda -. \llbracket \text{drop}(\text{Type Object}) \rrbracket))$
 $(\llbracket \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \rrbracket, \sigma)$
 $\langle \text{proof} \rangle$

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 $(\text{Type .allInstances@pre}())$
 $(\llbracket \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \rrbracket, \sigma)$
 $=$
 $(\text{Type .allInstances@pre}())$
 $(\llbracket \text{heap} = \sigma', \text{assocs} = A \rrbracket, \sigma)$
 $\langle \text{proof} \rangle$

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 $((\llbracket \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \rrbracket, \sigma) \models (P(\text{Type .allInstances@pre}())) =$
 $((\llbracket \text{heap} = \sigma', \text{assocs} = A \rrbracket, \sigma) \models (P(\text{Type .allInstances@pre}())))$
 $\langle \text{proof} \rangle$

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*: $cp\ P$
and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P\ X)$
shows $((\llbracket \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \rrbracket, \sigma) \models (P(\text{Type}.\text{allInstances}@pre())) =$
 $((\llbracket \text{heap} = \sigma', \text{assocs} = A \rrbracket, \sigma) \models (P((\text{Type}.\text{allInstances}@pre())$
 $\quad \rightarrow \text{including}(\lambda \cdot \llbracket (\text{Type } \text{Object}) \rrbracket)))$
 $\langle \text{proof} \rangle$

@post or @pre

theorem *StrictRefEqObject-vs-StrongEq''*:
assumes *WFF*: $WFF\ \tau$
and *valid-x*: $\tau \models (v\ (x :: ('A :: \text{object}, 'a :: \text{object option option})val))$
and *valid-y*: $\tau \models (v\ y)$
and *oid-preserve*: $\bigwedge x. x \in \text{ran } (\text{heap}(\text{fst } \tau)) \vee x \in \text{ran } (\text{heap}(\text{snd } \tau)) \implies$
 $\quad \text{oid-of } (H\ x) = \text{oid-of } x$
and *xy-together*: $\tau \models ((H.\text{allInstances}() \rightarrow \text{includes}(x) \text{ and } H.\text{allInstances}() \rightarrow \text{includes}(y))$
 or
 $(H.\text{allInstances}@pre() \rightarrow \text{includes}(x) \text{ and } H.\text{allInstances}@pre() \rightarrow \text{includes}(y)))$

shows $(\tau \models (\text{StrictRefEqObject } x\ y)) = (\tau \models (x \triangleq y))$
 $\langle \text{proof} \rangle$

6.2.3. OclIsNew, OclIsDeleted, OclIsMaintained, OclIsAbsent

definition *OclIsNew*:: $('A, 'a :: \{\text{null}, \text{object}\})val \Rightarrow ('A)Boolean\ ((-).\text{oclIsNew}'())$
where $X.\text{oclIsNew}() \equiv (\lambda \tau. \text{if } (\delta\ X)\ \tau = \text{true } \tau$
 $\quad \text{then } \llbracket \text{oid-of } (X\ \tau) \notin \text{dom}(\text{heap}(\text{fst } \tau)) \wedge$
 $\quad \text{oid-of } (X\ \tau) \in \text{dom}(\text{heap}(\text{snd } \tau)) \rrbracket$
 $\quad \text{else invalid } \tau)$

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*:: $('A, 'a :: \{\text{null}, \text{object}\})val \Rightarrow ('A)Boolean\ ((-).\text{oclIsDeleted}'())$
where $X.\text{oclIsDeleted}() \equiv (\lambda \tau. \text{if } (\delta\ X)\ \tau = \text{true } \tau$
 $\quad \text{then } \llbracket \text{oid-of } (X\ \tau) \in \text{dom}(\text{heap}(\text{fst } \tau)) \wedge$
 $\quad \text{oid-of } (X\ \tau) \notin \text{dom}(\text{heap}(\text{snd } \tau)) \rrbracket$
 $\quad \text{else invalid } \tau)$

definition *OclIsMaintained*:: $('A, 'a :: \{\text{null}, \text{object}\})val \Rightarrow ('A)Boolean\ ((-).\text{oclIsMaintained}'())$
where $X.\text{oclIsMaintained}() \equiv (\lambda \tau. \text{if } (\delta\ X)\ \tau = \text{true } \tau$
 $\quad \text{then } \llbracket \text{oid-of } (X\ \tau) \in \text{dom}(\text{heap}(\text{fst } \tau)) \wedge$
 $\quad \text{oid-of } (X\ \tau) \in \text{dom}(\text{heap}(\text{snd } \tau)) \rrbracket$
 $\quad \text{else invalid } \tau)$

definition *OclIsAbsent*:: $('A, 'a :: \{\text{null}, \text{object}\})val \Rightarrow ('A)Boolean\ ((-).\text{oclIsAbsent}'())$
where $X.\text{oclIsAbsent}() \equiv (\lambda \tau. \text{if } (\delta\ X)\ \tau = \text{true } \tau$
 $\quad \text{then } \llbracket \text{oid-of } (X\ \tau) \notin \text{dom}(\text{heap}(\text{fst } \tau)) \wedge$
 $\quad \text{oid-of } (X\ \tau) \notin \text{dom}(\text{heap}(\text{snd } \tau)) \rrbracket$
 $\quad \text{else invalid } \tau)$

lemma *state-split* : $\tau \models \delta X \implies$
 $\tau \models (X \text{ .oclIsNew}()) \vee \tau \models (X \text{ .oclIsDeleted}()) \vee$
 $\tau \models (X \text{ .oclIsMaintained}()) \vee \tau \models (X \text{ .oclIsAbsent}())$
<proof>

lemma *notNew-vs-others* : $\tau \models \delta X \implies$
 $(\neg \tau \models (X \text{ .oclIsNew}())) = (\tau \models (X \text{ .oclIsDeleted}()) \vee$
 $\tau \models (X \text{ .oclIsMaintained}()) \vee \tau \models (X \text{ .oclIsAbsent}()))$
<proof>

6.2.4. OclIsModifiedOnly

Definition

The following predicate—which is not part of the OCL standard—provides a simple, but powerful means to describe framing conditions. For any formal approach, be it animation of OCL contracts, test-case generation or die-hard theorem proving, the specification of the part of a system transition that *does not change* is of primordial importance. The following operator establishes the equality between old and new objects in the state (provided that they exist in both states), with the exception of those objects.

definition *OclIsModifiedOnly* :: (' \mathcal{A} ::object, ' α ::{null,object})Set \Rightarrow ' \mathcal{A} Boolean
 $(\rightarrow \text{oclIsModifiedOnly})(')$
where $X \rightarrow \text{oclIsModifiedOnly}() \equiv (\lambda(\sigma, \sigma').$
 $\text{let } X' = (\text{oid-of } ' \llbracket \text{Rep-Set}_{\text{base}}(X(\sigma, \sigma')) \rrbracket \rrbracket);$
 $S = ((\text{dom } (\text{heap } \sigma) \cap \text{dom } (\text{heap } \sigma')) - X')$
 $\text{in if } (\delta X) (\sigma, \sigma') = \text{true } (\sigma, \sigma') \wedge (\forall x \in \llbracket \text{Rep-Set}_{\text{base}}(X(\sigma, \sigma')) \rrbracket. x \neq$
 $\text{null})$
 $\text{then } \llbracket \forall x \in S. (\text{heap } \sigma) x = (\text{heap } \sigma') x \rrbracket$
 $\text{else invalid } (\sigma, \sigma')$

Execution with Invalid or Null or Null Element as Argument

lemma $\text{invalid} \rightarrow \text{oclIsModifiedOnly}() = \text{invalid}$
<proof>

lemma $\text{null} \rightarrow \text{oclIsModifiedOnly}() = \text{invalid}$
<proof>

lemma
assumes $X\text{-null} : \tau \models X \rightarrow \text{includes}(\text{null})$
shows $\tau \models X \rightarrow \text{oclIsModifiedOnly}() \triangleq \text{invalid}$
<proof>

Context Passing

lemma *cp-OclIsModifiedOnly* : $X \rightarrow \text{oclIsModifiedOnly}() \tau = (\lambda\tau. X \tau) \rightarrow \text{oclIsModifiedOnly}()$
 τ
<proof>

6.2.5. OclSelf

The following predicate—which is not part of the OCL standard—explicitly retrieves in the pre or post state the original OCL expression given as argument.

definition *[simp]*: $OclSelf\ x\ H\ fst\ snd = (\lambda\tau . \text{if } (\delta\ x)\ \tau = \text{true}\ \tau$
 then if $oid\text{-}of\ (x\ \tau) \in \text{dom}(\text{heap}(fst\ \tau)) \wedge oid\text{-}of\ (x\ \tau) \in \text{dom}(\text{heap}\ (snd\ \tau))$
 then $H\ \lceil(\text{heap}(fst\ snd\ \tau))(oid\text{-}of\ (x\ \tau))\rceil$
 else invalid τ
 else invalid τ)

definition $OclSelf\text{-}at\text{-}pre :: ('A::object, 'a::\{null, object\})val \Rightarrow$
 $('A \Rightarrow 'a) \Rightarrow$
 $('A::object, 'a::\{null, object\})val\ ((-)\text{@}pre(-))$
where $x\ \text{@}pre\ H = OclSelf\ x\ H\ fst$

definition $OclSelf\text{-}at\text{-}post :: ('A::object, 'a::\{null, object\})val \Rightarrow$
 $('A \Rightarrow 'a) \Rightarrow$
 $('A::object, 'a::\{null, object\})val\ ((-)\text{@}post(-))$
where $x\ \text{@}post\ H = OclSelf\ x\ H\ snd$

6.2.6. Framing Theorem

lemma *all-oid-diff*:

assumes $def\text{-}x : \tau \models \delta\ x$
assumes $def\text{-}X : \tau \models \delta\ X$
assumes $def\text{-}X' : \bigwedge x. x \in \llbracket Rep\text{-}Set_{base}\ (X\ \tau) \rrbracket \implies x \neq null$

defines $P \equiv (\lambda a. \text{not}\ (StrictRefEq_{Object}\ x\ a))$
shows $(\tau \models X \text{->forAll}(a\ |\ P\ a)) = (oid\text{-}of\ (x\ \tau) \notin oid\text{-}of\ ' \llbracket Rep\text{-}Set_{base}\ (X\ \tau) \rrbracket)$
<proof>

theorem *framing*:

assumes $modifies\text{clause}:\tau \models (X \text{->excluding}(x)) \text{->oclIsModifiedOnly}()$
and $oid\text{-}is\text{-}typerepr : \tau \models X \text{->forAll}(a\ |\ \text{not}\ (StrictRefEq_{Object}\ x\ a))$
shows $\tau \models (x\ \text{@}pre\ P \triangleq (x\ \text{@}post\ P))$
<proof>

As corollary, the framing property can be expressed with only the strong equality as comparison operator.

theorem *framing'*:

assumes $wff : WFF\ \tau$
assumes $modifies\text{clause}:\tau \models (X \text{->excluding}(x)) \text{->oclIsModifiedOnly}()$
and $oid\text{-}is\text{-}typerepr : \tau \models X \text{->forAll}(a\ |\ \text{not}\ (x \triangleq a))$
and $oid\text{-}preserve: \bigwedge x. x \in \text{ran}\ (\text{heap}(fst\ \tau)) \vee x \in \text{ran}\ (\text{heap}(snd\ \tau)) \implies$
 $oid\text{-}of\ (H\ x) = oid\text{-}of\ x$
and *xy-together*:
 $\tau \models X \text{->forAll}(y\ |\ (H.\text{allInstances}() \text{->includes}(x)\ \text{and}\ H.\text{allInstances}() \text{->includes}(y))\ \text{or}$
 $(H.\text{allInstances}@pre() \text{->includes}(x)\ \text{and}\ H.\text{allInstances}@pre() \text{->includes}(y)))$
shows $\tau \models (x\ \text{@}pre\ P \triangleq (x\ \text{@}post\ P))$

$\langle proof \rangle$

6.2.7. Miscellaneous

lemma *pre-post-new*: $\tau \models (x \text{ .oclIsNew}()) \implies \neg (\tau \models v(x \text{ @pre } H1)) \wedge \neg (\tau \models v(x \text{ @post } H2))$
 $\langle proof \rangle$

lemma *pre-post-old*: $\tau \models (x \text{ .oclIsDeleted}()) \implies \neg (\tau \models v(x \text{ @pre } H1)) \wedge \neg (\tau \models v(x \text{ @post } H2))$
 $\langle proof \rangle$

lemma *pre-post-absent*: $\tau \models (x \text{ .oclIsAbsent}()) \implies \neg (\tau \models v(x \text{ @pre } H1)) \wedge \neg (\tau \models v(x \text{ @post } H2))$
 $\langle proof \rangle$

lemma *pre-post-maintained*: $(\tau \models v(x \text{ @pre } H1) \vee \tau \models v(x \text{ @post } H2)) \implies \tau \models (x \text{ .oclIsMaintained}())$
 $\langle proof \rangle$

lemma *pre-post-maintained'*:
 $\tau \models (x \text{ .oclIsMaintained}()) \implies (\tau \models v(x \text{ @pre } (Some \ o \ H1)) \wedge \tau \models v(x \text{ @post } (Some \ o \ H2)))$
 $\langle proof \rangle$

lemma *framing-same-state*: $(\sigma, \sigma) \models (x \text{ @pre } H \triangleq (x \text{ @post } H))$
 $\langle proof \rangle$

end

theory *UML-Tools*
imports *UML-Logic*
begin

lemmas *substs1 = StrongEq-L-subst2-rev*
 foundation15[THEN iffD2, THEN StrongEq-L-subst2-rev]
 foundation7'[THEN iffD2, THEN foundation15[THEN iffD2,
 THEN StrongEq-L-subst2-rev]]
 foundation14[THEN iffD2, THEN StrongEq-L-subst2-rev]
 foundation13[THEN iffD2, THEN StrongEq-L-subst2-rev]

lemmas *substs2 = StrongEq-L-subst3-rev*
 foundation15[THEN iffD2, THEN StrongEq-L-subst3-rev]
 foundation7'[THEN iffD2, THEN foundation15[THEN iffD2,
 THEN StrongEq-L-subst3-rev]]

foundation14[*THEN iffD2*, *THEN StrongEq-L-subst3-rev*]
foundation13[*THEN iffD2*, *THEN StrongEq-L-subst3-rev*]

lemmas *substs4* = *StrongEq-L-subst4-rev*
foundation15[*THEN iffD2*, *THEN StrongEq-L-subst4-rev*]
foundation7'[*THEN iffD2*, *THEN foundation15*[*THEN iffD2*,
THEN StrongEq-L-subst4-rev]]
foundation14[*THEN iffD2*, *THEN StrongEq-L-subst4-rev*]
foundation13[*THEN iffD2*, *THEN StrongEq-L-subst4-rev*]

lemmas *substs* = *substs1 substs2 substs4* [*THEN iffD2*] *substs4*

thm *substs*
 $\langle ML \rangle$

lemma *test1* : $\tau \models A \implies \tau \models (A \text{ and } B \triangleq B)$
 $\langle proof \rangle$

lemma *test2* : $\tau \models A \implies \tau \models (A \text{ and } B \triangleq B)$
 $\langle proof \rangle$

lemma *test3* : $\tau \models A \implies \tau \models (A \text{ and } A)$
 $\langle proof \rangle$

lemma *test4* : $\tau \models \text{not } A \implies \tau \models (A \text{ and } B \triangleq \text{false})$
 $\langle proof \rangle$

lemma *test5* : $\tau \models (A \triangleq \text{null}) \implies \tau \models (B \triangleq \text{null}) \implies \neg (\tau \models (A \text{ and } B))$
 $\langle proof \rangle$

lemma *test6* : $\tau \models \text{not } A \implies \neg (\tau \models (A \text{ and } B))$
 $\langle proof \rangle$

lemma *test7* : $\neg (\tau \models (v \ A)) \implies \tau \models (\text{not } B) \implies \neg (\tau \models (A \text{ and } B))$
 $\langle proof \rangle$

lemma *X*: $\neg (\tau \models (\text{invalid and } B))$
 $\langle proof \rangle$

lemma *X'*: $\neg (\tau \models (\text{invalid and } B))$
 $\langle proof \rangle$

lemma *Y*: $\neg (\tau \models (\text{null and } B))$

```

⟨proof⟩
lemma  $Z$ :  $\neg (\tau \models (\text{false and } B))$ 
⟨proof⟩
lemma  $Z'$ :  $(\tau \models (\text{true and } B)) = (\tau \models B)$ 
⟨proof⟩

```

end

```

theory UML-Contracts
imports UML-State UML-Library
begin

```

Modeling of an operation contract for an operation with 2 arguments, (so depending on three parameters if one takes "self" into account).

```

locale contract0 =
  fixes  $f$  ::  $(\mathfrak{A}, 'a0::\text{null})\text{val} \Rightarrow$ 
     $(\mathfrak{A}, 'res::\text{null})\text{val}$ 
  fixes  $PRE$  ::  $(\mathfrak{A}, 'a0::\text{null})\text{val} \Rightarrow$ 
     $(\mathfrak{A}, \text{Boolean}_{base})\text{val}$ 
  fixes  $POST$  ::  $(\mathfrak{A}, 'a0::\text{null})\text{val} \Rightarrow$ 
     $(\mathfrak{A}, 'res::\text{null})\text{val} \Rightarrow$ 
     $(\mathfrak{A}, \text{Boolean}_{base})\text{val}$ 
  assumes  $\text{def-scheme: } f \text{ self} \equiv (\lambda \tau. \text{if } (\tau \models (\delta \text{ self}))$ 
     $\text{then SOME } res. (\tau \models PRE \text{ self}) \wedge$ 
     $(\tau \models POST \text{ self } (\lambda -. res))$ 
     $\text{else invalid } \tau)$ 
  assumes  $\forall \sigma \sigma' \sigma''. ((\sigma, \sigma') \models PRE \text{ self}) = ((\sigma, \sigma'') \models PRE \text{ self})$ 

  assumes  $cp_{PRE}: PRE \text{ (self)} \tau = PRE (\lambda -. \text{self } \tau) \tau$ 

  assumes  $cp_{POST}: POST \text{ (self)} (res) \tau = POST (\lambda -. \text{self } \tau) (\lambda -. res) \tau$ 

begin
  lemma  $\text{strict0 } [simp]: f \text{ invalid} = \text{invalid}$ 
  ⟨proof⟩

```

lemma *nullstrict0[simp]*: $f \text{ null} = \text{invalid}$
 $\langle \text{proof} \rangle$

lemma *cp-pre*: $cp \text{ self}' \implies cp (\lambda X. PRE (\text{self}' X))$
 $\langle \text{proof} \rangle$

lemma *cp-post*: $cp \text{ self}' \implies cp \text{ res}' \implies cp (\lambda X. POST (\text{self}' X) (\text{res}' X))$
 $\langle \text{proof} \rangle$

lemma *cp0* : $f \text{ self } \tau = f (\lambda -. \text{self } \tau) \tau$
 $\langle \text{proof} \rangle$

lemma *cp [simp]*: $cp \text{ self}' \implies cp \text{ res}' \implies cp (\lambda X. f (\text{self}' X))$
 $\langle \text{proof} \rangle$

theorem *unfold* :

assumes *context-ok*: $cp E$
and *args-def-or-valid*: $(\tau \models \delta \text{ self})$
and *pre-satisfied*: $\tau \models PRE \text{ self}$
and *post-satisfiable*: $\exists \text{ res}. (\tau \models POST \text{ self } (\lambda -. \text{res}))$
and *sat-for-sols-post*: $(\bigwedge \text{res}. \tau \models POST \text{ self } (\lambda -. \text{res}) \implies \tau \models E (\lambda -. \text{res}))$
shows $\tau \models E(f \text{ self})$

$\langle \text{proof} \rangle$

lemma *unfold2* :

assumes *context-ok*: $cp E$
and *args-def-or-valid*: $(\tau \models \delta \text{ self})$
and *pre-satisfied*: $\tau \models PRE \text{ self}$
and *postsplit-satisfied*: $\tau \models POST' \text{ self}$
and *post-decomposable* : $\bigwedge \text{res}. (POST \text{ self } \text{res} = ((POST' \text{ self}) \text{ and } (\text{res} \triangleq (BODY \text{ self}))))$
shows $(\tau \models E(f \text{ self})) = (\tau \models E(BODY \text{ self}))$

$\langle \text{proof} \rangle$

end

locale *contract1* =

fixes $f :: ('A, 'a0::\text{null}) \text{val} \Rightarrow ('A, 'a1::\text{null}) \text{val} \Rightarrow ('A, 'res::\text{null}) \text{val}$
fixes $PRE :: ('A, 'a0::\text{null}) \text{val} \Rightarrow ('A, 'a1::\text{null}) \text{val} \Rightarrow ('A, \text{Boolean}_{base}) \text{val}$
fixes $POST :: ('A, 'a0::\text{null}) \text{val} \Rightarrow ('A, 'a1::\text{null}) \text{val} \Rightarrow ('A, 'res::\text{null}) \text{val} \Rightarrow ('A, \text{Boolean}_{base}) \text{val}$

assumes *def-scheme*: $f \text{ self } a1 \equiv$
 $(\lambda \tau. \text{ if } (\tau \models (\delta \text{ self})) \wedge (\tau \models v \ a1)$
 $\text{ then SOME } res. (\tau \models PRE \text{ self } a1) \wedge$
 $(\tau \models POST \text{ self } a1 \ (\lambda -. res))$
 $\text{ else invalid } \tau)$
assumes $\forall \sigma \sigma' \sigma''. ((\sigma, \sigma') \models PRE \text{ self } a1) = ((\sigma, \sigma'') \models PRE \text{ self } a1)$
assumes $cp_{PRE}: PRE \ (self) \ (a1) \ \tau = PRE \ (\lambda -. self \ \tau) \ (\lambda -. a1 \ \tau) \ \tau$
assumes $cp_{POST}: POST \ (self) \ (a1) \ (res) \ \tau = POST \ (\lambda -. self \ \tau) (\lambda -. a1 \ \tau) \ (\lambda -. res \ \tau) \ \tau$

begin

lemma *strict0 [simp]*: $f \text{ invalid } X = \text{ invalid }$
 $\langle \text{proof} \rangle$

lemma *nullstrict0 [simp]*: $f \text{ null } X = \text{ invalid }$
 $\langle \text{proof} \rangle$

lemma *strict1 [simp]*: $f \text{ self invalid} = \text{ invalid }$
 $\langle \text{proof} \rangle$

lemma *cp-pre*: $cp \text{ self}' \implies cp \ a1' \implies cp \ (\lambda X. PRE \ (self' \ X) \ (a1' \ X) \)$
 $\langle \text{proof} \rangle$

lemma *cp-post*: $cp \text{ self}' \implies cp \ a1' \implies cp \ res'$
 $\implies cp \ (\lambda X. POST \ (self' \ X) \ (a1' \ X) \ (res' \ X))$
 $\langle \text{proof} \rangle$

lemma *cp0* : $f \text{ self } a1 \ \tau = f \ (\lambda -. self \ \tau) \ (\lambda -. a1 \ \tau) \ \tau$
 $\langle \text{proof} \rangle$

lemma *cp [simp]*: $cp \text{ self}' \implies cp \ a1' \implies cp \ res' \implies cp \ (\lambda X. f \ (self' \ X) \ (a1' \ X))$
 $\langle \text{proof} \rangle$

theorem *unfold* :

assumes *context-ok*: $cp \ E$
and *args-def-or-valid*: $(\tau \models \delta \text{ self}) \wedge (\tau \models v \ a1)$
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*: $(\bigwedge res. \tau \models POST \text{ self } a1 \ (\lambda -. res) \implies \tau \models E \ (\lambda -. res))$
shows $\tau \models E(f \text{ self } a1)$
 $\langle \text{proof} \rangle$

lemma *unfold2* :

assumes *context-ok*: $cp \ E$
and *args-def-or-valid*: $(\tau \models \delta \text{ self}) \wedge (\tau \models v \ a1)$
and *pre-satisfied*: $\tau \models PRE \text{ self } a1$
and *postsplit-satisfied*: $\tau \models POST' \text{ self } a1$

and *post-decomposable* : $\bigwedge \text{res. } (POST \text{ self } a1 \text{ res}) =$
 $((POST' \text{ self } a1) \text{ and } (res \triangleq (BODY \text{ self } a1)))$
shows $(\tau \models E(f \text{ self } a1)) = (\tau \models E(BODY \text{ self } a1))$
 $\langle \text{proof} \rangle$
end

locale *contract2* =
fixes *f* :: $(\mathfrak{A}, 'a0::null)val \Rightarrow$
 $(\mathfrak{A}, 'a1::null)val \Rightarrow (\mathfrak{A}, 'a2::null)val \Rightarrow$
 $(\mathfrak{A}, 'res::null)val$
fixes *PRE* :: $(\mathfrak{A}, 'a0::null)val \Rightarrow$
 $(\mathfrak{A}, 'a1::null)val \Rightarrow (\mathfrak{A}, 'a2::null)val \Rightarrow$
 $(\mathfrak{A}, Boolean_{base})val$
fixes *POST* :: $(\mathfrak{A}, 'a0::null)val \Rightarrow$
 $(\mathfrak{A}, 'a1::null)val \Rightarrow (\mathfrak{A}, 'a2::null)val \Rightarrow$
 $(\mathfrak{A}, 'res::null)val \Rightarrow$
 $(\mathfrak{A}, Boolean_{base})val$
assumes *def-scheme*: $f \text{ self } a1 \text{ } a2 \equiv$
 $(\lambda \tau. \text{if } (\tau \models (\delta \text{ self})) \wedge (\tau \models v \text{ } a1) \wedge (\tau \models v \text{ } a2)$
 $\text{then } SOME \text{ res. } (\tau \models PRE \text{ self } a1 \text{ } a2) \wedge$
 $(\tau \models POST \text{ self } a1 \text{ } a2 \text{ } (\lambda -. \text{res}))$
 $\text{else } invalid \text{ } \tau)$
assumes $\forall \sigma \sigma' \sigma''. ((\sigma, \sigma') \models PRE \text{ self } a1 \text{ } a2) = ((\sigma, \sigma'') \models PRE \text{ self } a1 \text{ } a2)$
assumes *cp_{PRE}*: $PRE \text{ (self) (} a1 \text{) (} a2 \text{) } \tau = PRE \text{ (} \lambda -. \text{self } \tau \text{) (} \lambda -. \text{ } a1 \text{ } \tau \text{) (} \lambda -. \text{ } a2 \text{ } \tau \text{) } \tau$
assumes *cp_{POST}*: $\bigwedge res. POST \text{ (self) (} a1 \text{) (} a2 \text{) (} res \text{) } \tau =$
 $POST \text{ (} \lambda -. \text{self } \tau \text{) (} \lambda -. \text{ } a1 \text{ } \tau \text{) (} \lambda -. \text{ } a2 \text{ } \tau \text{) (} \lambda -. \text{ } res \text{ } \tau \text{) } \tau$

begin

lemma *strict0* [*simp*]: $f \text{ invalid } X \text{ } Y = invalid$
 $\langle \text{proof} \rangle$

lemma *nullstrict0* [*simp*]: $f \text{ null } X \text{ } Y = invalid$
 $\langle \text{proof} \rangle$

lemma *strict1* [*simp*]: $f \text{ self } invalid \text{ } Y = invalid$
 $\langle \text{proof} \rangle$

lemma *strict2* [*simp*]: $f \text{ self } X \text{ } invalid = invalid$
 $\langle \text{proof} \rangle$

lemma *cp-pre*: $cp \text{ self}' \Longrightarrow cp \text{ } a1' \Longrightarrow cp \text{ } a2' \Longrightarrow cp \text{ (} \lambda X. PRE \text{ (self}' X \text{) (} a1' X \text{) (} a2' X \text{))}$
 $\langle \text{proof} \rangle$

lemma *cp-post*: $cp \text{ self}' \Longrightarrow cp \text{ } a1' \Longrightarrow cp \text{ } a2' \Longrightarrow cp \text{ } res'$

$\implies cp (\lambda X. POST (self' X) (a1' X) (a2' X) (res' X))$
 $\langle proof \rangle$

lemma *cp0* : $f self a1 a2 \tau = f (\lambda -. self \tau) (\lambda -. a1 \tau) (\lambda -. a2 \tau) \tau$
 $\langle proof \rangle$

lemma *cp [simp]*: $cp self' \implies cp a1' \implies cp a2' \implies cp res'$
 $\implies cp (\lambda X. f (self' X) (a1' X) (a2' X))$
 $\langle proof \rangle$

theorem *unfold* :

assumes *context-ok*: $cp E$
and *args-def-or-valid*: $(\tau \models \delta self) \wedge (\tau \models v a1) \wedge (\tau \models v a2)$
and *pre-satisfied*: $\tau \models PRE self a1 a2$
and *post-satisfiable*: $\exists res. (\tau \models POST self a1 a2 (\lambda -. res))$
and *sat-for-sols-post*: $(\bigwedge res. \tau \models POST self a1 a2 (\lambda -. res) \implies \tau \models E (\lambda -. res))$
shows $\tau \models E(f self a1 a2)$

$\langle proof \rangle$

lemma *unfold2* :

assumes *context-ok*: $cp E$
and *args-def-or-valid*: $(\tau \models \delta self) \wedge (\tau \models v a1) \wedge (\tau \models v a2)$
and *pre-satisfied*: $\tau \models PRE self a1 a2$
and *postsplit-satisfied*: $\tau \models POST' self a1 a2$
and *post-decomposable* : $\bigwedge res. (POST self a1 a2 res) =$
 $((POST' self a1 a2) \text{ and } (res \triangleq (BODY self a1 a2)))$
shows $(\tau \models E(f self a1 a2)) = (\tau \models E(BODY self a1 a2))$

$\langle proof \rangle$

end

end

theory *UML-Main*

imports *UML-State UML-Tools UML-Contracts*

begin

end

7. Example I : The Employee Analysis Model (UML)

```
theory
  Analysis-UML
imports
  ../../src/UML-Main
begin
```

7.1. Introduction

For certain concepts like classes and class-types, only a generic definition for its resulting semantics can be given. Generic means, there is a function outside HOL that “compiles” a concrete, closed-world class diagram into a “theory” of this data model, consisting of a bunch of definitions for classes, accessors, method, casts, and tests for actual types, as well as proofs for the fundamental properties of these operations in this concrete data model.

Such generic function or “compiler” can be implemented in Isabelle on the ML level. This has been done, for a semantics following the open-world assumption, for UML 2.0 in [4, 6]. In this paper, we follow another approach for UML 2.4: we define the concepts of the compilation informally, and present a concrete example which is verified in Isabelle/HOL.

7.1.1. Outlining the Example

We are presenting here an “analysis-model” of the (slightly modified) example Figure 7.3, page 20 of the OCL standard [28]. Here, analysis model means that associations were really represented as relation on objects on the state—as is intended by the standard—rather by pointers between objects as is done in our “design model” (see Chapter 8). To be precise, this theory contains the formalization of the data-part covered by the UML class model (see Figure 7.1):

This means that the association (attached to the association class **EmployeeRanking**) with the association ends **boss** and **employees** is implemented by the attribute **boss** and the operation **employees** (to be discussed in the OCL part captured by the subsequent theory).

object universe belongs to the type class “oclany,” i. e., each class type has to provide a function *oid-of* yielding the object id (oid) of the object.

```

instantiation typePerson :: object
begin
  definition oid-of-typePerson-def: oid-of x = (case x of mkPerson oid - ⇒ oid)
  instance ⟨proof⟩
end

```

```

instantiation typeOclAny :: object
begin
  definition oid-of-typeOclAny-def: oid-of x = (case x of mkOclAny oid - ⇒ oid)
  instance ⟨proof⟩
end

```

```

instantiation  $\mathfrak{A}$  :: object
begin
  definition oid-of- $\mathfrak{A}$ -def: oid-of x = (case x of
    inPerson person ⇒ oid-of person
    | inOclAny oclany ⇒ oid-of oclany)
  instance ⟨proof⟩
end

```

7.3. Instantiation of the Generic Strict Equality

We instantiate the referential equality on *Person* and *OclAny*

```

defs(overloaded) StrictRefEqObject-Person : (x::Person)  $\doteq$  y  $\equiv$  StrictRefEqObject x y
defs(overloaded) StrictRefEqObject-OclAny : (x::OclAny)  $\doteq$  y  $\equiv$  StrictRefEqObject x y

```

lemmas

```

cp-StrictRefEqObject[of x::Person y::Person  $\tau$ ,
  simplified StrictRefEqObject-Person[symmetric]]
cp-intro(9) [of P::Person ⇒ Person Q::Person ⇒ Person,
  simplified StrictRefEqObject-Person[symmetric] ]
StrictRefEqObject-def [of x::Person y::Person,
  simplified StrictRefEqObject-Person[symmetric]]
StrictRefEqObject-defargs [of - x::Person y::Person,
  simplified StrictRefEqObject-Person[symmetric]]
StrictRefEqObject-strict1
  [of x::Person,
  simplified StrictRefEqObject-Person[symmetric]]
StrictRefEqObject-strict2
  [of x::Person,
  simplified StrictRefEqObject-Person[symmetric]]

```

For each Class *C*, we will have a casting operation *.oclAsType*(*C*), a test on the actual type *.oclIsTypeOf*(*C*) as well as its relaxed form *.oclIsKindOf*(*C*) (corresponding exactly to Java’s *instanceof*-operator).

Thus, since we have two class-types in our concrete class hierarchy, we have two operations to declare and to provide two overloading definitions for the two static types.

7.4. OclAsType

7.4.1. Definition

consts $OclAsType_{OclAny} :: 'α \Rightarrow OclAny \ ((-) .oclAsType' (OclAny'))$

consts $OclAsType_{Person} :: 'α \Rightarrow Person \ ((-) .oclAsType' (Person'))$

definition $OclAsType_{OclAny}\text{-}\mathfrak{A} = (\lambda u. \lfloor \text{case } u \text{ of } in_{OclAny} \ a \Rightarrow a \mid in_{Person} \ (mk_{Person} \ oid \ a) \Rightarrow mk_{OclAny} \ oid \ \lfloor a \rfloor \rfloor)$

lemma $OclAsType_{OclAny}\text{-}\mathfrak{A}\text{-some}: OclAsType_{OclAny}\text{-}\mathfrak{A} \ x \neq None$
 $\langle proof \rangle$

defs (overloaded) $OclAsType_{OclAny}\text{-}OclAny:$
 $(X :: OclAny) .oclAsType(OclAny) \equiv X$

defs (overloaded) $OclAsType_{OclAny}\text{-}Person:$
 $(X :: Person) .oclAsType(OclAny) \equiv$
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad \mid \lfloor \perp \rfloor \Rightarrow \text{null } \tau$
 $\quad \mid \lfloor \lfloor mk_{Person} \ oid \ a \rfloor \rfloor \Rightarrow \lfloor \lfloor (mk_{OclAny} \ oid \ \lfloor a \rfloor) \rfloor \rfloor)$

definition $OclAsType_{Person}\text{-}\mathfrak{A} = (\lambda u. \text{case } u \text{ of } in_{Person} \ p \Rightarrow \lfloor p \rfloor \mid in_{OclAny} \ (mk_{OclAny} \ oid \ \lfloor a \rfloor) \Rightarrow \lfloor mk_{Person} \ oid \ a \rfloor \mid - \Rightarrow None)$

defs (overloaded) $OclAsType_{Person}\text{-}OclAny:$
 $(X :: OclAny) .oclAsType(Person) \equiv$
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad \mid \lfloor \perp \rfloor \Rightarrow \text{null } \tau$
 $\quad \mid \lfloor \lfloor mk_{OclAny} \ oid \ \perp \rfloor \rfloor \Rightarrow \text{invalid } \tau \quad (* \text{ down-cast exception } *)$
 $\quad \mid \lfloor \lfloor mk_{OclAny} \ oid \ \lfloor a \rfloor \rfloor \rfloor \Rightarrow \lfloor \lfloor mk_{Person} \ oid \ a \rfloor \rfloor)$

defs (overloaded) $OclAsType_{Person}\text{-}Person:$
 $(X :: Person) .oclAsType(Person) \equiv X$

lemmas $[simp] =$
 $OclAsType_{OclAny}\text{-}OclAny$
 $OclAsType_{Person}\text{-}Person$

7.4.2. Context Passing

lemma $cp\text{-}OclAsType_{OclAny}\text{-}Person\text{-}Person$: $cp\ P \implies cp(\lambda X. (P\ (X::Person)::Person))$
 $.oclAsType(OclAny))$
 $\langle proof \rangle$

lemma $cp\text{-}OclAsType_{OclAny}\text{-}OclAny\text{-}OclAny$: $cp\ P \implies cp(\lambda X. (P\ (X::OclAny)::OclAny))$
 $.oclAsType(OclAny))$
 $\langle proof \rangle$

lemma $cp\text{-}OclAsType_{Person}\text{-}Person\text{-}Person$: $cp\ P \implies cp(\lambda X. (P\ (X::Person)::Person))$
 $.oclAsType(Person))$
 $\langle proof \rangle$

lemma $cp\text{-}OclAsType_{Person}\text{-}OclAny\text{-}OclAny$: $cp\ P \implies cp(\lambda X. (P\ (X::OclAny)::OclAny))$
 $.oclAsType(Person))$
 $\langle proof \rangle$

lemma $cp\text{-}OclAsType_{OclAny}\text{-}Person\text{-}OclAny$: $cp\ P \implies cp(\lambda X. (P\ (X::Person)::OclAny))$
 $.oclAsType(OclAny))$
 $\langle proof \rangle$

lemma $cp\text{-}OclAsType_{OclAny}\text{-}OclAny\text{-}Person$: $cp\ P \implies cp(\lambda X. (P\ (X::OclAny)::Person))$
 $.oclAsType(OclAny))$
 $\langle proof \rangle$

lemma $cp\text{-}OclAsType_{Person}\text{-}Person\text{-}OclAny$: $cp\ P \implies cp(\lambda X. (P\ (X::Person)::OclAny))$
 $.oclAsType(Person))$
 $\langle proof \rangle$

lemma $cp\text{-}OclAsType_{Person}\text{-}OclAny\text{-}Person$: $cp\ P \implies cp(\lambda X. (P\ (X::OclAny)::Person))$
 $.oclAsType(Person))$
 $\langle proof \rangle$

lemmas $[simp] =$
 $cp\text{-}OclAsType_{OclAny}\text{-}Person\text{-}Person$
 $cp\text{-}OclAsType_{OclAny}\text{-}OclAny\text{-}OclAny$
 $cp\text{-}OclAsType_{Person}\text{-}Person\text{-}Person$
 $cp\text{-}OclAsType_{Person}\text{-}OclAny\text{-}OclAny$

 $cp\text{-}OclAsType_{OclAny}\text{-}Person\text{-}OclAny$
 $cp\text{-}OclAsType_{OclAny}\text{-}OclAny\text{-}Person$
 $cp\text{-}OclAsType_{Person}\text{-}Person\text{-}OclAny$
 $cp\text{-}OclAsType_{Person}\text{-}OclAny\text{-}Person$

7.4.3. Execution with Invalid or Null as Argument

lemma $OclAsType_{OclAny}\text{-}OclAny\text{-}strict$: $(invalid::OclAny) .oclAsType(OclAny) = invalid$
 $\langle proof \rangle$

lemma $OclAsType_{OclAny}\text{-}OclAny\text{-}nullstrict$: $(null::OclAny) .oclAsType(OclAny) = null$
 $\langle proof \rangle$

lemma $OclAsType_{OclAny}\text{-}Person\text{-}strict[simp]$: $(invalid::Person) .oclAsType(OclAny) = invalid$
 $\langle proof \rangle$

lemma *OclAsType_{OclAny}-Person-nullstrict*[simp] : (null::Person) .oclAsType(OclAny) = null
 <proof>

lemma *OclAsType_{Person}-OclAny-strict*[simp] : (invalid::OclAny) .oclAsType(Person) = invalid
 <proof>

lemma *OclAsType_{Person}-OclAny-nullstrict*[simp] : (null::OclAny) .oclAsType(Person) = null
 <proof>

lemma *OclAsType_{Person}-Person-strict* : (invalid::Person) .oclAsType(Person) = invalid
 <proof>

lemma *OclAsType_{Person}-Person-nullstrict* : (null::Person) .oclAsType(Person) = null
 <proof>

7.5. OclIsTypeOf

7.5.1. Definition

consts *OclIsTypeOf_{OclAny}* :: 'α ⇒ Boolean ((-).oclIsTypeOf'(OclAny'))

consts *OclIsTypeOf_{Person}* :: 'α ⇒ Boolean ((-).oclIsTypeOf'(Person'))

defs (overloaded) *OclIsTypeOf_{OclAny}-OclAny*:
 (X::OclAny) .oclIsTypeOf(OclAny) ≡
 (λτ. case X τ of
 ⊥ ⇒ invalid τ
 | ⊥ ⇒ true τ (* invalid ?? *)
 | [[mk_{OclAny} oid ⊥]] ⇒ true τ
 | [[mk_{OclAny} oid [-]]] ⇒ false τ)

defs (overloaded) *OclIsTypeOf_{OclAny}-Person*:
 (X::Person) .oclIsTypeOf(OclAny) ≡
 (λτ. case X τ of
 ⊥ ⇒ invalid τ
 | ⊥ ⇒ true τ (* invalid ?? *)
 | [[-]] ⇒ false τ)

defs (overloaded) *OclIsTypeOf_{Person}-OclAny*:
 (X::OclAny) .oclIsTypeOf(Person) ≡
 (λτ. case X τ of
 ⊥ ⇒ invalid τ
 | ⊥ ⇒ true τ
 | [[mk_{OclAny} oid ⊥]] ⇒ false τ
 | [[mk_{OclAny} oid [-]]] ⇒ true τ)

defs (overloaded) *OclIsTypeOf_{Person}-Person*:
 (X::Person) .oclIsTypeOf(Person) ≡
 (λτ. case X τ of
 ⊥ ⇒ invalid τ)

| - \Rightarrow true τ)

7.5.2. Context Passing

lemma $cp\text{-}OclIsTypeOf_{OclAny}\text{-}Person\text{-}Person:$ $cp(\lambda X.(P(X::Person)::Person).oclIsTypeOf(OclAny))$ $\langle proof \rangle$	cp	P	\Rightarrow
lemma $cp\text{-}OclIsTypeOf_{OclAny}\text{-}OclAny\text{-}OclAny:$ $cp(\lambda X.(P(X::OclAny)::OclAny).oclIsTypeOf(OclAny))$ $\langle proof \rangle$	cp	P	\Rightarrow
lemma $cp\text{-}OclIsTypeOf_{Person}\text{-}Person\text{-}Person:$ $cp(\lambda X.(P(X::Person)::Person).oclIsTypeOf(Person))$ $\langle proof \rangle$	cp	P	\Rightarrow
lemma $cp\text{-}OclIsTypeOf_{Person}\text{-}OclAny\text{-}OclAny:$ $cp(\lambda X.(P(X::OclAny)::OclAny).oclIsTypeOf(Person))$ $\langle proof \rangle$	cp	P	\Rightarrow
lemma $cp\text{-}OclIsTypeOf_{OclAny}\text{-}Person\text{-}OclAny:$ $cp(\lambda X.(P(X::Person)::OclAny).oclIsTypeOf(OclAny))$ $\langle proof \rangle$	cp	P	\Rightarrow
lemma $cp\text{-}OclIsTypeOf_{OclAny}\text{-}OclAny\text{-}Person:$ $cp(\lambda X.(P(X::OclAny)::Person).oclIsTypeOf(OclAny))$ $\langle proof \rangle$	cp	P	\Rightarrow
lemma $cp\text{-}OclIsTypeOf_{Person}\text{-}Person\text{-}OclAny:$ $cp(\lambda X.(P(X::Person)::OclAny).oclIsTypeOf(Person))$ $\langle proof \rangle$	cp	P	\Rightarrow
lemma $cp\text{-}OclIsTypeOf_{Person}\text{-}OclAny\text{-}Person:$ $cp(\lambda X.(P(X::OclAny)::Person).oclIsTypeOf(Person))$ $\langle proof \rangle$	cp	P	\Rightarrow
lemmas $[simp] =$ $cp\text{-}OclIsTypeOf_{OclAny}\text{-}Person\text{-}Person$ $cp\text{-}OclIsTypeOf_{OclAny}\text{-}OclAny\text{-}OclAny$ $cp\text{-}OclIsTypeOf_{Person}\text{-}Person\text{-}Person$ $cp\text{-}OclIsTypeOf_{Person}\text{-}OclAny\text{-}OclAny$ $cp\text{-}OclIsTypeOf_{OclAny}\text{-}Person\text{-}OclAny$ $cp\text{-}OclIsTypeOf_{OclAny}\text{-}OclAny\text{-}Person$ $cp\text{-}OclIsTypeOf_{Person}\text{-}Person\text{-}OclAny$ $cp\text{-}OclIsTypeOf_{Person}\text{-}OclAny\text{-}Person$			

7.5.3. Execution with Invalid or Null as Argument

lemma $OclIsTypeOf_{OclAny}\text{-}OclAny\text{-}strict1[simp]:$ $(invalid::OclAny).oclIsTypeOf(OclAny) = invalid$ $\langle proof \rangle$
lemma $OclIsTypeOf_{OclAny}\text{-}OclAny\text{-}strict2[simp]:$ $(null::OclAny).oclIsTypeOf(OclAny) = true$ $\langle proof \rangle$

lemma *OclIsTypeOf_{OclAny}-Person-strict1*[simp]:
 $(invalid::Person) .oclIsTypeOf(OclAny) = invalid$
 $\langle proof \rangle$
lemma *OclIsTypeOf_{OclAny}-Person-strict2*[simp]:
 $(null::Person) .oclIsTypeOf(OclAny) = true$
 $\langle proof \rangle$
lemma *OclIsTypeOf_{Person}-OclAny-strict1*[simp]:
 $(invalid::OclAny) .oclIsTypeOf(Person) = invalid$
 $\langle proof \rangle$
lemma *OclIsTypeOf_{Person}-OclAny-strict2*[simp]:
 $(null::OclAny) .oclIsTypeOf(Person) = true$
 $\langle proof \rangle$
lemma *OclIsTypeOf_{Person}-Person-strict1*[simp]:
 $(invalid::Person) .oclIsTypeOf(Person) = invalid$
 $\langle proof \rangle$
lemma *OclIsTypeOf_{Person}-Person-strict2*[simp]:
 $(null::Person) .oclIsTypeOf(Person) = true$
 $\langle proof \rangle$

7.5.4. Up Down Casting

lemma *actualType-larger-staticType*:
assumes *isdef*: $\tau \models (\delta X)$
shows $\tau \models (X::Person) .oclIsTypeOf(OclAny) \triangleq false$
 $\langle proof \rangle$

lemma *down-cast-type*:
assumes *isOclAny*: $\tau \models (X::OclAny) .oclIsTypeOf(OclAny)$
and *non-null*: $\tau \models (\delta X)$
shows $\tau \models (X .oclAsType(Person)) \triangleq invalid$
 $\langle proof \rangle$

lemma *down-cast-type'*:
assumes *isOclAny*: $\tau \models (X::OclAny) .oclIsTypeOf(OclAny)$
and *non-null*: $\tau \models (\delta X)$
shows $\tau \models not (v (X .oclAsType(Person)))$
 $\langle proof \rangle$

lemma *up-down-cast* :
assumes *isdef*: $\tau \models (\delta X)$
shows $\tau \models ((X::Person) .oclAsType(OclAny) .oclAsType(Person) \triangleq X)$
 $\langle proof \rangle$

lemma *up-down-cast-Person-OclAny-Person* [simp]:
shows $((X::Person) .oclAsType(OclAny) .oclAsType(Person) = X)$
 $\langle proof \rangle$

lemma *up-down-cast-Person-OclAny-Person'*:

assumes $\tau \models v \ X$
shows $\tau \models (((X :: \text{Person}) . \text{oclAsType}(\text{OclAny}) . \text{oclAsType}(\text{Person})) \doteq X)$
 $\langle \text{proof} \rangle$

lemma *up-down-cast-Person-OclAny-Person''*:

assumes $\tau \models v \ (X :: \text{Person})$

shows $\tau \models (X . \text{oclIsTypeOf}(\text{Person}) \text{ implies } (X . \text{oclAsType}(\text{OclAny}) . \text{oclAsType}(\text{Person})) \doteq X)$
 $\langle \text{proof} \rangle$

7.6. OclIsKindOf

7.6.1. Definition

consts $\text{OclIsKindOf}_{\text{OclAny}} :: 'a \Rightarrow \text{Boolean} \ ((-). \text{oclIsKindOf}'(\text{OclAny}'))$

consts $\text{OclIsKindOf}_{\text{Person}} :: 'a \Rightarrow \text{Boolean} \ ((-). \text{oclIsKindOf}'(\text{Person}'))$

defs (overloaded) $\text{OclIsKindOf}_{\text{OclAny-OclAny}}$:

$(X :: \text{OclAny}) . \text{oclIsKindOf}(\text{OclAny}) \equiv$
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | _ \Rightarrow \text{true } \tau)$

defs (overloaded) $\text{OclIsKindOf}_{\text{OclAny-Person}}$:

$(X :: \text{Person}) . \text{oclIsKindOf}(\text{OclAny}) \equiv$
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | _ \Rightarrow \text{true } \tau)$

defs (overloaded) $\text{OclIsKindOf}_{\text{Person-OclAny}}$:

$(X :: \text{OclAny}) . \text{oclIsKindOf}(\text{Person}) \equiv$
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \lfloor \perp \rfloor \Rightarrow \text{true } \tau$
 $\quad | \lfloor \text{mk}_{\text{OclAny}} \text{ oid } \perp \rfloor \Rightarrow \text{false } \tau$
 $\quad | \lfloor \text{mk}_{\text{OclAny}} \text{ oid } \lfloor _ \rfloor \rfloor \Rightarrow \text{true } \tau)$

defs (overloaded) $\text{OclIsKindOf}_{\text{Person-Person}}$:

$(X :: \text{Person}) . \text{oclIsKindOf}(\text{Person}) \equiv$
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | _ \Rightarrow \text{true } \tau)$

7.6.2. Context Passing

lemma $\text{cp-OclIsKindOf}_{\text{OclAny-Person-Person}}: \quad \text{cp} \quad P \quad \Rightarrow$
 $\text{cp}(\lambda X. (P(X :: \text{Person}) :: \text{Person}). \text{oclIsKindOf}(\text{OclAny}))$

$\langle proof \rangle$
lemma $cp\text{-}OclIsKindOf_{OclAny}\text{-}OclAny\text{-}OclAny:$ cp P \implies
 $cp(\lambda X.(P(X::OclAny)::OclAny).oclIsKindOf(OclAny))$
 $\langle proof \rangle$
lemma $cp\text{-}OclIsKindOf_{Person}\text{-}Person\text{-}Person:$ cp P \implies
 $cp(\lambda X.(P(X::Person)::Person).oclIsKindOf(Person))$
 $\langle proof \rangle$
lemma $cp\text{-}OclIsKindOf_{Person}\text{-}OclAny\text{-}OclAny:$ cp P \implies
 $cp(\lambda X.(P(X::OclAny)::OclAny).oclIsKindOf(Person))$
 $\langle proof \rangle$
lemma $cp\text{-}OclIsKindOf_{OclAny}\text{-}Person\text{-}OclAny:$ cp P \implies
 $cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(OclAny))$
 $\langle proof \rangle$
lemma $cp\text{-}OclIsKindOf_{OclAny}\text{-}OclAny\text{-}Person:$ cp P \implies
 $cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(OclAny))$
 $\langle proof \rangle$
lemma $cp\text{-}OclIsKindOf_{Person}\text{-}Person\text{-}OclAny:$ cp P \implies
 $cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(Person))$
 $\langle proof \rangle$
lemma $cp\text{-}OclIsKindOf_{Person}\text{-}OclAny\text{-}Person:$ cp P \implies
 $cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(Person))$
 $\langle proof \rangle$
lemmas $[simp] =$
 $cp\text{-}OclIsKindOf_{OclAny}\text{-}Person\text{-}Person$
 $cp\text{-}OclIsKindOf_{OclAny}\text{-}OclAny\text{-}OclAny$
 $cp\text{-}OclIsKindOf_{Person}\text{-}Person\text{-}Person$
 $cp\text{-}OclIsKindOf_{Person}\text{-}OclAny\text{-}OclAny$

 $cp\text{-}OclIsKindOf_{OclAny}\text{-}Person\text{-}OclAny$
 $cp\text{-}OclIsKindOf_{OclAny}\text{-}OclAny\text{-}Person$
 $cp\text{-}OclIsKindOf_{Person}\text{-}Person\text{-}OclAny$
 $cp\text{-}OclIsKindOf_{Person}\text{-}OclAny\text{-}Person$

7.6.3. Execution with Invalid or Null as Argument

lemma $OclIsKindOf_{OclAny}\text{-}OclAny\text{-}strict1[simp] : (invalid::OclAny).oclIsKindOf(OclAny) =$
 $invalid$
 $\langle proof \rangle$

lemma $OclIsKindOf_{OclAny}\text{-}OclAny\text{-}strict2[simp] : (null::OclAny).oclIsKindOf(OclAny) =$
 $true$
 $\langle proof \rangle$

lemma $OclIsKindOf_{OclAny}\text{-}Person\text{-}strict1[simp] : (invalid::Person).oclIsKindOf(OclAny) =$
 $invalid$
 $\langle proof \rangle$

lemma *OclIsKindOf_{OclAny}-Person-strict2*[simp]: (null::Person) .oclIsKindOf(OclAny) = true
 ⟨proof⟩

lemma *OclIsKindOf_{Person}-OclAny-strict1*[simp]: (invalid::OclAny) .oclIsKindOf(Person) = invalid
 ⟨proof⟩

lemma *OclIsKindOf_{Person}-OclAny-strict2*[simp]: (null::OclAny) .oclIsKindOf(Person) = true
 ⟨proof⟩

lemma *OclIsKindOf_{Person}-Person-strict1*[simp]: (invalid::Person) .oclIsKindOf(Person) = invalid
 ⟨proof⟩

lemma *OclIsKindOf_{Person}-Person-strict2*[simp]: (null::Person) .oclIsKindOf(Person) = true
 ⟨proof⟩

7.6.4. Up Down Casting

lemma *actualKind-larger-staticKind*:
assumes *isdef*: $\tau \models (\delta X)$
shows $\tau \models ((X::Person) .oclIsKindOf(OclAny)) \triangleq true$
 ⟨proof⟩

lemma *down-cast-kind*:
assumes *isOclAny*: $\neg (\tau \models ((X::OclAny).oclIsKindOf(Person)))$
and *non-null*: $\tau \models (\delta X)$
shows $\tau \models ((X .oclAsType(Person)) \triangleq invalid)$
 ⟨proof⟩

7.7. OclAllInstances

To denote OCL-types 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}-A*
definition *OclAny* \equiv *OclAsType_{OclAny}-A*
lemmas [simp] = *Person-def OclAny-def*

lemma *OclAllInstances-generic_{OclAny-exec}*: *OclAllInstances-generic pre-post OclAny* =
 ($\lambda\tau. Abs\text{-}Set_{base} \llbracket Some \text{ ‘ } OclAny \text{ ‘ } ran (heap (pre\text{-}post \tau)) \rrbracket$)
 ⟨proof⟩

lemma *OclAllInstances-at-post_{OclAny-exec}*: *OclAny .allInstances()* =
 ($\lambda\tau. Abs\text{-}Set_{base} \llbracket Some \text{ ‘ } OclAny \text{ ‘ } ran (heap (snd \tau)) \rrbracket$)
 ⟨proof⟩

lemma *OclAllInstances-at-pre_{OclAny-exec}*: *OclAny .allInstances@pre()* =

$\langle \text{proof} \rangle$ $(\lambda \tau. \text{Abs-Set}_{\text{base}} \llbracket \text{Some } \text{'OclAny'} \text{ ran (heap (fst } \tau)) \rrbracket)$

7.7.1. OclIsTypeOf

lemma *OclAny-allInstances-generic-oclIsTypeOf_{OclAny}1*:

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

shows $\exists \tau. (\tau \models ((\text{OclAllInstances-generic pre-post OclAny}) \rightarrow \text{forAll}(X|X \text{ .oclIsTypeOf}(\text{OclAny}))))$

$\langle \text{proof} \rangle$

lemma *OclAny-allInstances-at-post-oclIsTypeOf_{OclAny}1*:

$\exists \tau. (\tau \models (\text{OclAny .allInstances}() \rightarrow \text{forAll}(X|X \text{ .oclIsTypeOf}(\text{OclAny}))))$

$\langle \text{proof} \rangle$

lemma *OclAny-allInstances-at-pre-oclIsTypeOf_{OclAny}1*:

$\exists \tau. (\tau \models (\text{OclAny .allInstances@pre}() \rightarrow \text{forAll}(X|X \text{ .oclIsTypeOf}(\text{OclAny}))))$

$\langle \text{proof} \rangle$

lemma *OclAny-allInstances-generic-oclIsTypeOf_{OclAny}2*:

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

shows $\exists \tau. (\tau \models \text{not } ((\text{OclAllInstances-generic pre-post OclAny}) \rightarrow \text{forAll}(X|X \text{ .oclIsTypeOf}(\text{OclAny}))))$

$\langle \text{proof} \rangle$

lemma *OclAny-allInstances-at-post-oclIsTypeOf_{OclAny}2*:

$\exists \tau. (\tau \models \text{not } (\text{OclAny .allInstances}() \rightarrow \text{forAll}(X|X \text{ .oclIsTypeOf}(\text{OclAny}))))$

$\langle \text{proof} \rangle$

lemma *OclAny-allInstances-at-pre-oclIsTypeOf_{OclAny}2*:

$\exists \tau. (\tau \models \text{not } (\text{OclAny .allInstances@pre}() \rightarrow \text{forAll}(X|X \text{ .oclIsTypeOf}(\text{OclAny}))))$

$\langle \text{proof} \rangle$

lemma *Person-allInstances-generic-oclIsTypeOf_{Person}*:

$\tau \models ((\text{OclAllInstances-generic pre-post Person}) \rightarrow \text{forAll}(X|X \text{ .oclIsTypeOf}(\text{Person})))$

$\langle \text{proof} \rangle$

lemma *Person-allInstances-at-post-oclIsTypeOf_{Person}*:

$\tau \models (\text{Person .allInstances}() \rightarrow \text{forAll}(X|X \text{ .oclIsTypeOf}(\text{Person})))$

$\langle \text{proof} \rangle$

lemma *Person-allInstances-at-pre-oclIsTypeOf_{Person}*:

$\tau \models (\text{Person .allInstances@pre}() \rightarrow \text{forAll}(X|X \text{ .oclIsTypeOf}(\text{Person})))$

$\langle \text{proof} \rangle$

7.7.2. OclIsKindOf

lemma *OclAny-allInstances-generic-oclIsKindOf_{OclAny}*:

$\tau \models ((\text{OclAllInstances-generic pre-post OclAny}) \rightarrow \text{forAll}(X|X \text{ .oclIsKindOf}(\text{OclAny})))$

$\langle \text{proof} \rangle$

lemma *OclAny-allInstances-at-post-oclIsKindOf_{OclAny}*:
 $\tau \models (OclAny.allInstances() \rightarrow \text{forAll}(X|X.oclIsKindOf(OclAny)))$
 $\langle \text{proof} \rangle$

lemma *OclAny-allInstances-at-pre-oclIsKindOf_{OclAny}*:
 $\tau \models (OclAny.allInstances@pre() \rightarrow \text{forAll}(X|X.oclIsKindOf(OclAny)))$
 $\langle \text{proof} \rangle$

lemma *Person-allInstances-generic-oclIsKindOf_{OclAny}*:
 $\tau \models ((OclAllInstances-generic \text{ pre-post } Person) \rightarrow \text{forAll}(X|X.oclIsKindOf(OclAny)))$
 $\langle \text{proof} \rangle$

lemma *Person-allInstances-at-post-oclIsKindOf_{OclAny}*:
 $\tau \models (Person.allInstances() \rightarrow \text{forAll}(X|X.oclIsKindOf(OclAny)))$
 $\langle \text{proof} \rangle$

lemma *Person-allInstances-at-pre-oclIsKindOf_{OclAny}*:
 $\tau \models (Person.allInstances@pre() \rightarrow \text{forAll}(X|X.oclIsKindOf(OclAny)))$
 $\langle \text{proof} \rangle$

lemma *Person-allInstances-generic-oclIsKindOf_{Person}*:
 $\tau \models ((OclAllInstances-generic \text{ pre-post } Person) \rightarrow \text{forAll}(X|X.oclIsKindOf(Person)))$
 $\langle \text{proof} \rangle$

lemma *Person-allInstances-at-post-oclIsKindOf_{Person}*:
 $\tau \models (Person.allInstances() \rightarrow \text{forAll}(X|X.oclIsKindOf(Person)))$
 $\langle \text{proof} \rangle$

lemma *Person-allInstances-at-pre-oclIsKindOf_{Person}*:
 $\tau \models (Person.allInstances@pre() \rightarrow \text{forAll}(X|X.oclIsKindOf(Person)))$
 $\langle \text{proof} \rangle$

7.8. The Accessors (any, boss, salary)

Should be generated entirely from a class-diagram.

7.8.1. Definition (of the association Employee-Boss)

We start with a oid for the association; this oid can be used in presence of association classes to represent the association inside an object, pretty much similar to the Design_UML, where we stored an oid inside the class as “pointer.”

definition $oid_{Person}BOSS :: oid$ **where** $oid_{Person}BOSS = 10$

From there on, we can already define an empty state which must contain for $oid_{Person}BOSS$ the empty relation (encoded as association list, since there are associations with a Sequence-like structure).

definition *eval-extract* :: (' \mathcal{A} ,('a::object) option option) val
 \Rightarrow (oid \Rightarrow (' \mathcal{A} , 'c::null) val)
 \Rightarrow (' \mathcal{A} , 'c::null) val

where *eval-extract* $X f = (\lambda \tau. \text{case } X \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau \quad (* \text{ exception propagation } *)$
 $\quad | \perp \Rightarrow \text{invalid } \tau \quad (* \text{ dereferencing null pointer } *)$
 $\quad | [\![\text{obj}]\!] \Rightarrow f \text{ (oid-of obj) } \tau)$

definition *choose₂-1* = *fst*

definition *choose₂-2* = *snd*

definition *List-flatten* = ($\lambda l. (\text{foldl } ((\lambda \text{acc}. (\lambda l. (\text{foldl } ((\lambda \text{acc}. (\lambda l. (\text{Cons } l) (\text{acc})))) (\text{acc}))$
 $((\text{rev } l)))))) (\text{Nil } ((\text{rev } l))))$

definition *deref-assocs₂* :: (' \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-assocs₂* *pre-post to-from assoc-oid f oid* =
 $(\lambda \tau. \text{case } (\text{assocs } (\text{pre-post } \tau)) \text{ assoc-oid of}$
 $\quad [S] \Rightarrow f (\text{List-flatten } (\text{map } (\text{choose}_2\text{-2} \circ \text{to-from})$
 $\quad (\text{filter } (\lambda p. \text{List.member } (\text{choose}_2\text{-1 } (\text{to-from } p)) \text{ oid } S)))$
 $\quad \tau$
 $\quad | - \Rightarrow \text{invalid } \tau)$

The *pre-post*-parameter is configured with *fst* or *snd*, the *to-from*-parameter either with the identity *id* or the following combinator *switch*:

definition *switch₂-1* = ($\lambda [x,y] \Rightarrow (x,y)$)

definition *switch₂-2* = ($\lambda [x,y] \Rightarrow (y,x)$)

definition *switch₃-1* = ($\lambda [x,y,z] \Rightarrow (x,y)$)

definition *switch₃-2* = ($\lambda [x,y,z] \Rightarrow (x,z)$)

definition *switch₃-3* = ($\lambda [x,y,z] \Rightarrow (y,x)$)

definition *switch₃-4* = ($\lambda [x,y,z] \Rightarrow (y,z)$)

definition *switch₃-5* = ($\lambda [x,y,z] \Rightarrow (z,x)$)

definition *switch₃-6* = ($\lambda [x,y,z] \Rightarrow (z,y)$)

definition *select-object* :: ((' \mathcal{A} , 'b::null)val)
 \Rightarrow ((' \mathcal{A} , 'b)val \Rightarrow (' \mathcal{A} , 'c)val \Rightarrow (' \mathcal{A} , 'b)val)
 \Rightarrow ((' \mathcal{A} , 'b)val \Rightarrow (' \mathcal{A} , 'd)val)
 \Rightarrow (oid \Rightarrow (' \mathcal{A} , 'c::null)val)
 \Rightarrow oid list
 \Rightarrow (' \mathcal{A} , 'd)val

where *select-object* *mt incl smash deref l* = *smash*(*foldl* *incl* *mt* (*map deref l*))
 $(* \text{ smash returns null with mt in input (in this case, object contains null pointer) } *)$

The continuation *f* is usually instantiated with a smashing function which is either the identity *id* or, for 0..1 cardinalities of associations, the *OclANY*-selector which also handles the *null*-cases appropriately. A standard use-case for this combinator is for

example:

term (*select-object mtSet UML-Set.OclIncluding OclANY f l oid*) :: (\mathfrak{A} , 'a::null)val

definition $deref-oid_{Person} :: (\mathfrak{A} \text{ state} \times \mathfrak{A} \text{ state} \Rightarrow \mathfrak{A} \text{ state})$
 $\Rightarrow (type_{Person} \Rightarrow (\mathfrak{A}, 'c::null)val)$
 $\Rightarrow oid$
 $\Rightarrow (\mathfrak{A}, 'c::null)val$

where $deref-oid_{Person} \text{ fst-snd } f \text{ oid} = (\lambda \tau. \text{case } (heap \text{ (fst-snd } \tau)) \text{ oid of}$
 $\quad | \text{ in}_{Person} \text{ obj } \rfloor \Rightarrow f \text{ obj } \tau$
 $\quad | - \Rightarrow \text{invalid } \tau)$

definition $deref-oid_{OclAny} :: (\mathfrak{A} \text{ state} \times \mathfrak{A} \text{ state} \Rightarrow \mathfrak{A} \text{ state})$
 $\Rightarrow (type_{OclAny} \Rightarrow (\mathfrak{A}, 'c::null)val)$
 $\Rightarrow oid$
 $\Rightarrow (\mathfrak{A}, 'c::null)val$

where $deref-oid_{OclAny} \text{ fst-snd } f \text{ oid} = (\lambda \tau. \text{case } (heap \text{ (fst-snd } \tau)) \text{ oid of}$
 $\quad | \text{ in}_{OclAny} \text{ obj } \rfloor \Rightarrow f \text{ obj } \tau$
 $\quad | - \Rightarrow \text{invalid } \tau)$

pointer undefined in state or not referencing a type conform object representation

definition $select_{OclAny} \mathcal{ANY} f = (\lambda X. \text{case } X \text{ of}$
 $\quad (mk_{OclAny} - \perp) \Rightarrow \text{null}$
 $\quad | (mk_{OclAny} - \lfloor any \rfloor) \Rightarrow f (\lambda x -. \lfloor \lfloor x \rfloor \rfloor) \text{ any})$

definition $select_{Person} \mathcal{BOSS} f = \text{select-object mtSet UML-Set.OclIncluding OclANY } (f (\lambda x -. \lfloor \lfloor x \rfloor \rfloor))$

definition $select_{Person} \mathcal{SALARY} f = (\lambda X. \text{case } X \text{ of}$
 $\quad (mk_{Person} - \perp) \Rightarrow \text{null}$
 $\quad | (mk_{Person} - \lfloor salary \rfloor) \Rightarrow f (\lambda x -. \lfloor \lfloor x \rfloor \rfloor) \text{ salary})$

definition $deref-assocs_2 \mathcal{BOSS} \text{ fst-snd } f = (\lambda mk_{Person} \text{ oid} -. \Rightarrow$
 $\quad deref-assocs_2 \text{ fst-snd } switch_{2-1} \text{ oid}_{Person} \mathcal{BOSS} f \text{ oid})$

definition $in\text{-pre-state} = \text{fst}$

definition $in\text{-post-state} = \text{snd}$

definition $reconst\text{-basetype} = (\lambda \text{convert } x. \text{convert } x)$

definition $dot_{OclAny} \mathcal{ANY} :: OclAny \Rightarrow - \ ((1(-).any) \ 50)$
where $(X).any = \text{eval-extract } X$
 $\quad (deref-oid_{OclAny} \text{ in-post-state}$
 $\quad \quad (select_{OclAny} \mathcal{ANY}$
 $\quad \quad \quad reconst\text{-basetype}))$

definition $\text{dot}_{\text{Person}}\text{BOSS} :: \text{Person} \Rightarrow \text{Person} \ ((1(-).\text{boss}) \ 50)$

where $(X).\text{boss} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{Person}} \text{ in-post-state}$
 $(\text{deref-assocs}_2\text{BOSS} \text{ in-post-state}$
 $(\text{select}_{\text{Person}}\text{BOSS}$
 $(\text{deref-oid}_{\text{Person}} \text{ in-post-state}))))$

definition $\text{dot}_{\text{Person}}\text{SALARY} :: \text{Person} \Rightarrow \text{Integer} \ ((1(-).\text{salary}) \ 50)$

where $(X).\text{salary} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{Person}} \text{ in-post-state}$
 $(\text{select}_{\text{Person}}\text{SALARY}$
 $\text{reconst-basetype}))$

definition $\text{dot}_{\text{OclAny}}\text{ANY-at-pre} :: \text{OclAny} \Rightarrow - \ ((1(-).\text{any@pre}) \ 50)$

where $(X).\text{any@pre} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{OclAny}} \text{ in-pre-state}$
 $(\text{select}_{\text{OclAny}}\text{ANY}$
 $\text{reconst-basetype}))$

definition $\text{dot}_{\text{Person}}\text{BOSS-at-pre} :: \text{Person} \Rightarrow \text{Person} \ ((1(-).\text{boss@pre}) \ 50)$

where $(X).\text{boss@pre} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{Person}} \text{ in-pre-state}$
 $(\text{deref-assocs}_2\text{BOSS} \text{ in-pre-state}$
 $(\text{select}_{\text{Person}}\text{BOSS}$
 $(\text{deref-oid}_{\text{Person}} \text{ in-pre-state}))))$

definition $\text{dot}_{\text{Person}}\text{SALARY-at-pre} :: \text{Person} \Rightarrow \text{Integer} \ ((1(-).\text{salary@pre}) \ 50)$

where $(X).\text{salary@pre} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{Person}} \text{ in-pre-state}$
 $(\text{select}_{\text{Person}}\text{SALARY}$
 $\text{reconst-basetype}))$

lemmas $[\text{simp}] =$

$\text{dot}_{\text{OclAny}}\text{ANY-def}$
 $\text{dot}_{\text{Person}}\text{BOSS-def}$
 $\text{dot}_{\text{Person}}\text{SALARY-def}$
 $\text{dot}_{\text{OclAny}}\text{ANY-at-pre-def}$
 $\text{dot}_{\text{Person}}\text{BOSS-at-pre-def}$
 $\text{dot}_{\text{Person}}\text{SALARY-at-pre-def}$

7.8.2. Context Passing

lemmas $[\text{simp}] = \text{eval-extract-def}$

lemma $\text{cp-dot}_{\text{OclAny}}\text{ANY}: ((X).\text{any}) \ \tau = ((\lambda-. X \ \tau).\text{any}) \ \tau \ \langle \text{proof} \rangle$

lemma $\text{cp-dot}_{\text{Person}}\text{BOSS}: ((X).\text{boss}) \ \tau = ((\lambda-. X \ \tau).\text{boss}) \ \tau \ \langle \text{proof} \rangle$

lemma $\text{cp-dot}_{\text{Person}}\text{SALARY}: ((X).\text{salary}) \ \tau = ((\lambda-. X \ \tau).\text{salary}) \ \tau \ \langle \text{proof} \rangle$

lemma $cp\text{-}dot_{OclAny} \mathcal{ANY}\text{-}at\text{-}pre$: $((X).any@pre) \tau = ((\lambda\text{-}. X \tau).any@pre) \tau \langle proof \rangle$
lemma $cp\text{-}dot_{Person} \mathcal{BOSS}\text{-}at\text{-}pre$: $((X).boss@pre) \tau = ((\lambda\text{-}. X \tau).boss@pre) \tau \langle proof \rangle$
lemma $cp\text{-}dot_{Person} \mathcal{SALARY}\text{-}at\text{-}pre$: $((X).salary@pre) \tau = ((\lambda\text{-}. X \tau).salary@pre) \tau \langle proof \rangle$

lemmas $cp\text{-}dot_{OclAny} \mathcal{ANY}\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{OclAny} \mathcal{ANY}[THEN allI[THEN allI],$
 $of \lambda X \text{-}. X \lambda \text{-} \tau. \tau, THEN cpI1]$

lemmas $cp\text{-}dot_{OclAny} \mathcal{ANY}\text{-}at\text{-}pre\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{OclAny} \mathcal{ANY}\text{-}at\text{-}pre[THEN allI[THEN allI],$
 $of \lambda X \text{-}. X \lambda \text{-} \tau. \tau, THEN cpI1]$

lemmas $cp\text{-}dot_{Person} \mathcal{BOSS}\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{Person} \mathcal{BOSS}[THEN allI[THEN allI],$
 $of \lambda X \text{-}. X \lambda \text{-} \tau. \tau, THEN cpI1]$

lemmas $cp\text{-}dot_{Person} \mathcal{BOSS}\text{-}at\text{-}pre\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{Person} \mathcal{BOSS}\text{-}at\text{-}pre[THEN allI[THEN allI],$
 $of \lambda X \text{-}. X \lambda \text{-} \tau. \tau, THEN cpI1]$

lemmas $cp\text{-}dot_{Person} \mathcal{SALARY}\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{Person} \mathcal{SALARY}[THEN allI[THEN allI],$
 $of \lambda X \text{-}. X \lambda \text{-} \tau. \tau, THEN cpI1]$

lemmas $cp\text{-}dot_{Person} \mathcal{SALARY}\text{-}at\text{-}pre\text{-}I$ [simp, intro!]=
 $cp\text{-}dot_{Person} \mathcal{SALARY}\text{-}at\text{-}pre[THEN allI[THEN allI],$
 $of \lambda X \text{-}. X \lambda \text{-} \tau. \tau, THEN cpI1]$

7.8.3. Execution with Invalid or Null as Argument

lemma $dot_{OclAny} \mathcal{ANY}\text{-}nullstrict$ [simp]: $(null).any = invalid$
 $\langle proof \rangle$
lemma $dot_{OclAny} \mathcal{ANY}\text{-}at\text{-}pre\text{-}nullstrict$ [simp]: $(null).any@pre = invalid$
 $\langle proof \rangle$
lemma $dot_{OclAny} \mathcal{ANY}\text{-}strict$ [simp]: $(invalid).any = invalid$
 $\langle proof \rangle$
lemma $dot_{OclAny} \mathcal{ANY}\text{-}at\text{-}pre\text{-}strict$ [simp]: $(invalid).any@pre = invalid$
 $\langle proof \rangle$

lemma $dot_{Person} \mathcal{BOSS}\text{-}nullstrict$ [simp]: $(null).boss = invalid$
 $\langle proof \rangle$
lemma $dot_{Person} \mathcal{BOSS}\text{-}at\text{-}pre\text{-}nullstrict$ [simp]: $(null).boss@pre = invalid$
 $\langle proof \rangle$
lemma $dot_{Person} \mathcal{BOSS}\text{-}strict$ [simp]: $(invalid).boss = invalid$
 $\langle proof \rangle$
lemma $dot_{Person} \mathcal{BOSS}\text{-}at\text{-}pre\text{-}strict$ [simp]: $(invalid).boss@pre = invalid$
 $\langle proof \rangle$

lemma $dot_{Person} \mathcal{SALARY}\text{-}nullstrict$ [simp]: $(null).salary = invalid$
 $\langle proof \rangle$

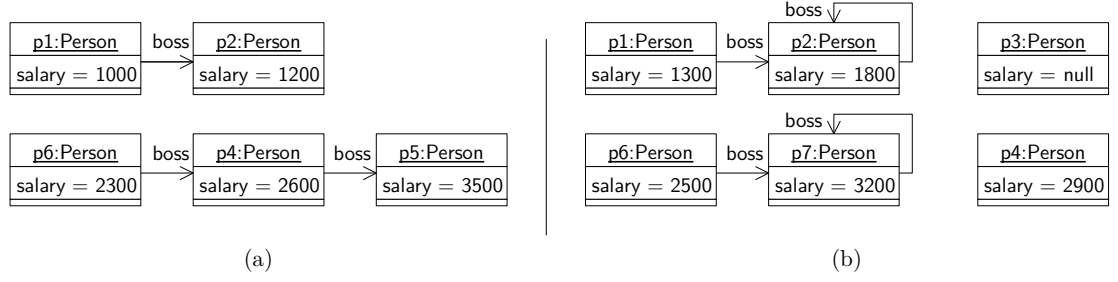


Figure 7.2.: (a) pre-state σ_1 and (b) post-state σ'_1 .

lemma $\text{dot}_{Person} \text{SALARY-at-pre-nullstrict} [\text{simp}] : (\text{null}).\text{salary}@pre = \text{invalid}$
 $\langle \text{proof} \rangle$
lemma $\text{dot}_{Person} \text{SALARY-strict} [\text{simp}] : (\text{invalid}).\text{salary} = \text{invalid}$
 $\langle \text{proof} \rangle$
lemma $\text{dot}_{Person} \text{SALARY-at-pre-strict} [\text{simp}] : (\text{invalid}).\text{salary}@pre = \text{invalid}$
 $\langle \text{proof} \rangle$

7.9. A Little Infra-structure on Example States

The example we are defining in this section comes from the figure 7.2.

definition $\text{OclInt1000} \text{ (1000) where } \text{OclInt1000} = (\lambda . . \llbracket 1000 \rrbracket)$
definition $\text{OclInt1200} \text{ (1200) where } \text{OclInt1200} = (\lambda . . \llbracket 1200 \rrbracket)$
definition $\text{OclInt1300} \text{ (1300) where } \text{OclInt1300} = (\lambda . . \llbracket 1300 \rrbracket)$
definition $\text{OclInt1800} \text{ (1800) where } \text{OclInt1800} = (\lambda . . \llbracket 1800 \rrbracket)$
definition $\text{OclInt2600} \text{ (2600) where } \text{OclInt2600} = (\lambda . . \llbracket 2600 \rrbracket)$
definition $\text{OclInt2900} \text{ (2900) where } \text{OclInt2900} = (\lambda . . \llbracket 2900 \rrbracket)$
definition $\text{OclInt3200} \text{ (3200) where } \text{OclInt3200} = (\lambda . . \llbracket 3200 \rrbracket)$
definition $\text{OclInt3500} \text{ (3500) where } \text{OclInt3500} = (\lambda . . \llbracket 3500 \rrbracket)$

definition $\text{oid0} \equiv 0$
definition $\text{oid1} \equiv 1$
definition $\text{oid2} \equiv 2$
definition $\text{oid3} \equiv 3$
definition $\text{oid4} \equiv 4$
definition $\text{oid5} \equiv 5$
definition $\text{oid6} \equiv 6$
definition $\text{oid7} \equiv 7$
definition $\text{oid8} \equiv 8$

definition $\text{person1} \equiv \text{mk}_{Person} \text{ oid0 } [1300]$
definition $\text{person2} \equiv \text{mk}_{Person} \text{ oid1 } [1800]$
definition $\text{person3} \equiv \text{mk}_{Person} \text{ oid2 } \text{None}$
definition $\text{person4} \equiv \text{mk}_{Person} \text{ oid3 } [2900]$
definition $\text{person5} \equiv \text{mk}_{Person} \text{ oid4 } [3500]$
definition $\text{person6} \equiv \text{mk}_{Person} \text{ oid5 } [2500]$

definition $person7 \equiv mk_{OclAny} \text{ oid6 } \llbracket 3200 \rrbracket$

definition $person8 \equiv mk_{OclAny} \text{ oid7 } None$

definition $person9 \equiv mk_{Person} \text{ oid8 } \llbracket 0 \rrbracket$

definition

$$\begin{aligned} \sigma_1 \equiv & \langle \text{ heap} = \text{empty}(\text{oid0} \mapsto \text{in}_{Person} (mk_{Person} \text{ oid0 } \llbracket 1000 \rrbracket)) \\ & (\text{oid1} \mapsto \text{in}_{Person} (mk_{Person} \text{ oid1 } \llbracket 1200 \rrbracket)) \\ & (*oid2*) \\ & (\text{oid3} \mapsto \text{in}_{Person} (mk_{Person} \text{ oid3 } \llbracket 2600 \rrbracket)) \\ & (\text{oid4} \mapsto \text{in}_{Person} person5) \\ & (\text{oid5} \mapsto \text{in}_{Person} (mk_{Person} \text{ oid5 } \llbracket 2300 \rrbracket)) \\ & (*oid6*) \\ & (*oid7*) \\ & (\text{oid8} \mapsto \text{in}_{Person} person9), \\ & \text{assocs} = \text{empty}(\text{oid}_{Person} BOSS \mapsto [[[\text{oid0}], [\text{oid1}]], [[\text{oid3}], [\text{oid4}]], [[\text{oid5}], [\text{oid3}]]]) \rangle \end{aligned}$$

definition

$$\begin{aligned} \sigma_1' \equiv & \langle \text{ heap} = \text{empty}(\text{oid0} \mapsto \text{in}_{Person} person1) \\ & (\text{oid1} \mapsto \text{in}_{Person} person2) \\ & (\text{oid2} \mapsto \text{in}_{Person} person3) \\ & (\text{oid3} \mapsto \text{in}_{Person} person4) \\ & (*oid4*) \\ & (\text{oid5} \mapsto \text{in}_{Person} person6) \\ & (\text{oid6} \mapsto \text{in}_{OclAny} person7) \\ & (\text{oid7} \mapsto \text{in}_{OclAny} person8) \\ & (\text{oid8} \mapsto \text{in}_{Person} person9), \\ & \text{assocs} = \text{empty}(\text{oid}_{Person} BOSS \mapsto \\ & [[[\text{oid0}], [\text{oid1}]], [[\text{oid1}], [\text{oid1}]], [[\text{oid5}], [\text{oid6}]], [[\text{oid6}], [\text{oid6}]]]) \rangle \end{aligned}$$

definition $\sigma_0 \equiv \langle \text{ heap} = \text{empty}, \text{assocs} = \text{empty} \rangle$

lemma $\text{basic-}\tau\text{-wff}: WFF(\sigma_1, \sigma_1')$

$\langle \text{proof} \rangle$

lemma $[\text{simp}, \text{code-unfold}]: \text{dom}(\text{heap } \sigma_1) = \{\text{oid0}, \text{oid1}, (*, \text{oid2}*)\text{oid3}, \text{oid4}, \text{oid5}(*, \text{oid6}, \text{oid7}*), \text{oid8}\}$

$\langle \text{proof} \rangle$

lemma $[\text{simp}, \text{code-unfold}]: \text{dom}(\text{heap } \sigma_1') = \{\text{oid0}, \text{oid1}, \text{oid2}, \text{oid3}, (*, \text{oid4}*)\text{oid5}, \text{oid6}, \text{oid7}, \text{oid8}\}$

$\langle \text{proof} \rangle$

definition $X_{Person} 1 :: Person \equiv \lambda - . \llbracket person1 \rrbracket$

definition $X_{Person} 2 :: Person \equiv \lambda - . \llbracket person2 \rrbracket$

definition $X_{Person} 3 :: Person \equiv \lambda - . \llbracket person3 \rrbracket$

definition $X_{Person} 4 :: Person \equiv \lambda - . \llbracket person4 \rrbracket$

definition $X_{Person} 5 :: Person \equiv \lambda - . \llbracket person5 \rrbracket$

definition $X_{Person} 6 :: Person \equiv \lambda - . \llbracket person6 \rrbracket$

definition $X_{Person} 7 :: OclAny \equiv \lambda - . \llbracket person7 \rrbracket$

definition $X_{Person} 8 :: OclAny \equiv \lambda - . \llbracket person8 \rrbracket$

definition $X_{Person9} :: Person \equiv \lambda - . [\mid person9 \mid]$

lemma $[code-unfold]: ((x::Person) \doteq y) = StrictRefEqObject\ x\ y\ \langle proof \rangle$

lemma $[code-unfold]: ((x::OclAny) \doteq y) = StrictRefEqObject\ x\ y\ \langle proof \rangle$

lemmas $[simp, code-unfold] =$

$OclAsType_{OclAny-OclAny}$

$OclAsType_{OclAny-Person}$

$OclAsType_{Person-OclAny}$

$OclAsType_{Person-Person}$

$OclIsTypeOf_{OclAny-OclAny}$

$OclIsTypeOf_{OclAny-Person}$

$OclIsTypeOf_{Person-OclAny}$

$OclIsTypeOf_{Person-Person}$

$OclIsKindOf_{OclAny-OclAny}$

$OclIsKindOf_{OclAny-Person}$

$OclIsKindOf_{Person-OclAny}$

$OclIsKindOf_{Person-Person}$

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} . salary <> 1000)$

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} . salary \doteq 1300)$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} . salary@pre \doteq 1000)$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} . salary@pre <> 1300)$

lemma $(\sigma_1, \sigma_1') \models (X_{Person1} . oclIsMaintained())$
 $\langle proof \rangle$

lemma $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models ((X_{Person1} . oclAsType(OclAny) . oclAsType(Person)) \doteq X_{Person1})$
 $\langle proof \rangle$

Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person1} . oclIsTypeOf(Person))$

Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models not(X_{Person1} . oclIsTypeOf(OclAny))$

Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person1} . oclIsKindOf(Person))$

Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person1} . oclIsKindOf(OclAny))$

Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models not(X_{Person1} . oclAsType(OclAny) . oclIsTypeOf(OclAny))$

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person2} . salary \doteq 1800)$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person2} . salary@pre \doteq 1200)$

lemma $(\sigma_1, \sigma_1') \models (X_{Person2} . oclIsMaintained())$
 $\langle proof \rangle$

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person3} . salary \doteq null)$

Assert $\bigwedge_{s_{post}. (\sigma_1, s_{post}) \models \text{not}(v(X_{Person3}.salary@pre))}$
lemma $(\sigma_1, \sigma_1') \models (X_{Person3}.oclIsNew())$
 $\langle proof \rangle$

lemma $(\sigma_1, \sigma_1') \models (X_{Person4}.oclIsMaintained())$
 $\langle proof \rangle$

Assert $\bigwedge_{s_{pre}}. (s_{pre}, \sigma_1') \models \text{not}(v(X_{Person5}.salary))$
Assert $\bigwedge_{s_{post}. (\sigma_1, s_{post}) \models (X_{Person5}.salary@pre \doteq 3500)}$

lemma $(\sigma_1, \sigma_1') \models (X_{Person5}.oclIsDeleted())$
 $\langle proof \rangle$

lemma $(\sigma_1, \sigma_1') \models (X_{Person6}.oclIsMaintained())$
 $\langle proof \rangle$

Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post}) \models v(X_{Person7}.oclAsType(Person))}$

lemma $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post}) \models ((X_{Person7}.oclAsType(Person).oclAsType(OclAny).oclAsType(Person)) \doteq (X_{Person7}.oclAsType(Person)))}$

$\langle proof \rangle$

lemma $(\sigma_1, \sigma_1') \models (X_{Person7}.oclIsNew())$
 $\langle proof \rangle$

Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post}) \models (X_{Person8} <> X_{Person7})}$
Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post}) \models \text{not}(v(X_{Person8}.oclAsType(Person)))}$
Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post}) \models (X_{Person8}.oclIsTypeOf(OclAny))}$
Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post}) \models \text{not}(X_{Person8}.oclIsTypeOf(Person))}$
Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post}) \models \text{not}(X_{Person8}.oclIsKindOf(Person))}$
Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post}) \models (X_{Person8}.oclIsKindOf(OclAny))}$

lemma $\sigma\text{-modifiedonly: } (\sigma_1, \sigma_1') \models (\text{Set}\{ X_{Person1}.oclAsType(OclAny), X_{Person2}.oclAsType(OclAny), (*, X_{Person3}.oclAsType(OclAny)*), X_{Person4}.oclAsType(OclAny), (*, X_{Person5}.oclAsType(OclAny)*})$

$$\begin{aligned}
& , X_{Person6} .oclAsType(OclAny) \\
& (*, X_{Person7} .oclAsType(OclAny)*) \\
& (*, X_{Person8} .oclAsType(OclAny)*) \\
& (*, X_{Person9} .oclAsType(OclAny)*) \} \rightarrow oclIsModifiedOnly()
\end{aligned}$$
 $\langle proof \rangle$

lemma $(\sigma_1, \sigma_1') \models ((X_{Person9} @pre (\lambda x. \lfloor OclAsType_{Person} \mathfrak{A} x \rfloor)) \triangleq X_{Person9})$
 $\langle proof \rangle$

lemma $(\sigma_1, \sigma_1') \models ((X_{Person9} @post (\lambda x. \lfloor OclAsType_{Person} \mathfrak{A} x \rfloor)) \triangleq X_{Person9})$
 $\langle proof \rangle$

lemma $(\sigma_1, \sigma_1') \models (((X_{Person9} .oclAsType(OclAny)) @pre (\lambda x. \lfloor OclAsType_{OclAny} \mathfrak{A} x \rfloor)) \triangleq$
 $((X_{Person9} .oclAsType(OclAny)) @post (\lambda x. \lfloor OclAsType_{OclAny} \mathfrak{A} x \rfloor)))$
 $\langle proof \rangle$

lemma $perm\text{-}\sigma_1' : \sigma_1' = () \text{ heap} = \text{empty}$

$$\begin{aligned}
& (oid8 \mapsto in_{Person} person9) \\
& (oid7 \mapsto in_{OclAny} person8) \\
& (oid6 \mapsto in_{OclAny} person7) \\
& (oid5 \mapsto in_{Person} person6) \\
& (*oid4*) \\
& (oid3 \mapsto in_{Person} person4) \\
& (oid2 \mapsto in_{Person} person3) \\
& (oid1 \mapsto in_{Person} person2) \\
& (oid0 \mapsto in_{Person} person1) \\
& , \text{assoc} = \text{assoc} \sigma_1' ()
\end{aligned}$$
 $\langle proof \rangle$

declare $const\text{-}ss [simp]$

lemma $\bigwedge \sigma_1.$
 $(\sigma_1, \sigma_1') \models (Person .allInstances() \doteq Set\{ X_{Person1}, X_{Person2}, X_{Person3}, X_{Person4}(*,$
 $X_{Person5}*), X_{Person6},$
 $X_{Person7} .oclAsType(Person)(*, X_{Person8}*), X_{Person9} \})$
 $\langle proof \rangle$

lemma $\bigwedge \sigma_1.$
 $(\sigma_1, \sigma_1') \models (OclAny .allInstances() \doteq Set\{ X_{Person1} .oclAsType(OclAny), X_{Person2}$
 $.oclAsType(OclAny),$
 $X_{Person3} .oclAsType(OclAny), X_{Person4} .oclAsType(OclAny)$
 $(*, X_{Person5}*), X_{Person6} .oclAsType(OclAny),$
 $X_{Person7}, X_{Person8}, X_{Person9} .oclAsType(OclAny) \})$
 $\langle proof \rangle$

end

```

theory
  Analysis-OCL
imports
  Analysis-UML
begin

```

7.10. OCL Part: Standard State Infrastructure

Ideally, these definitions are automatically generated from the class model.

7.11. Invariant

These recursive predicates can be defined conservatively by greatest fix-point constructions—automatically. See [4, 5] for details. For the purpose of this example, we state them as axioms here.

```

context Person
  inv label : self .boss <> null implies (self .salary \<le>
    ((self .boss) .salary))

```

definition $Person\text{-}label_{inv} :: Person \Rightarrow Boolean$
where $Person\text{-}label_{inv} (self) \equiv$
 $(self .boss <> null \text{ implies } (self .salary \leq_{int} ((self .boss) .salary)))$

definition $Person\text{-}label_{invATpre} :: Person \Rightarrow Boolean$
where $Person\text{-}label_{invATpre} (self) \equiv$
 $(self .boss@pre <> null \text{ implies } (self .salary@pre \leq_{int} ((self .boss@pre) .salary@pre)))$

definition $Person\text{-}label_{globalinv} :: Boolean$
where $Person\text{-}label_{globalinv} \equiv (Person .allInstances() \text{--}> \text{forAll}(x \mid Person\text{-}label_{inv} (x)) \text{ and }$
 $(Person .allInstances@pre() \text{--}> \text{forAll}(x \mid Person\text{-}label_{invATpre} (x))))$

lemma $\tau \models \delta (X .boss) \implies \tau \models Person .allInstances() \text{--}> \text{includes}(X .boss) \wedge$
 $\tau \models Person .allInstances() \text{--}> \text{includes}(X)$
 $\langle proof \rangle$

lemma $REC\text{-}pre : \tau \models Person\text{-}label_{globalinv}$
 $\implies \tau \models Person .allInstances() \text{--}> \text{includes}(X) \text{ (* } X \text{ represented object in state *)}$
 $\implies \exists REC. \tau \models REC(X) \triangleq (Person\text{-}label_{inv} (X) \text{ and } (X .boss <> null \text{ implies } REC(X .boss)))$
 $\langle proof \rangle$

This allows to state a predicate:

axiomatization $inv_{Person-label} :: Person \Rightarrow Boolean$

where $inv_{Person-label-def}$:

$(\tau \models Person.allInstances() \rightarrow includes(self)) \implies$
 $(\tau \models (inv_{Person-label}(self) \triangleq (self.boss \neq null \implies$
 $(self.salary \leq_{int} ((self.boss).salary)) \text{ and }$
 $inv_{Person-label}(self.boss))))$

axiomatization $inv_{Person-labelATpre} :: Person \Rightarrow Boolean$

where $inv_{Person-labelATpre-def}$:

$(\tau \models Person.allInstances@pre() \rightarrow includes(self)) \implies$
 $(\tau \models (inv_{Person-labelATpre}(self) \triangleq (self.boss@pre \neq null \implies$
 $(self.salary@pre \leq_{int} ((self.boss@pre).salary@pre)) \text{ and }$
 $inv_{Person-labelATpre}(self.boss@pre))))$

lemma $inv-1$:

$(\tau \models Person.allInstances() \rightarrow includes(self)) \implies$
 $(\tau \models inv_{Person-label}(self) = ((\tau \models (self.boss \neq null)) \vee$
 $(\tau \models (self.boss \neq null) \wedge$
 $\tau \models ((self.salary) \leq_{int} (self.boss.salary)) \wedge$
 $\tau \models (inv_{Person-label}(self.boss))))$

$\langle proof \rangle$

lemma $inv-2$:

$(\tau \models Person.allInstances@pre() \rightarrow includes(self)) \implies$
 $(\tau \models inv_{Person-labelATpre}(self) = ((\tau \models (self.boss@pre \neq null)) \vee$
 $(\tau \models (self.boss@pre \neq null) \wedge$
 $(\tau \models (self.boss@pre.salary@pre \leq_{int} self.salary@pre)) \wedge$
 $(\tau \models (inv_{Person-labelATpre}(self.boss@pre))))$

$\langle proof \rangle$

A very first attempt to characterize the axiomatization by an inductive definition - this can not be the last word since too weak (should be equality!)

coinductive $inv :: Person \Rightarrow (\mathcal{A})st \Rightarrow bool$ **where**

$(\tau \models (\delta self)) \implies ((\tau \models (self.boss \neq null)) \vee$
 $(\tau \models (self.boss \neq null) \wedge (\tau \models (self.boss.salary \leq_{int} self.salary)) \wedge$
 $(inv(self.boss)\tau)))$
 $\implies (inv self \tau)$

7.12. The Contract of a Recursive Query

The original specification of a recursive query :

```
context Person :: contents() : Set(Integer)
pre:   true
post:  result = if self.boss = null
           then Set{i}
```



```

else self.boss.contents()->including(i)
endif

```

For the case of recursive queries, we use at present just axiomatizations:

axiomatization *contents* :: *Person* \Rightarrow *Set-Integer* $((1(-).contents'()) \ 50)$

where *contents-def*:

```

(self.contents()) = ( $\lambda \tau. (if \tau \models (\delta \ self)$ 
                      then SOME res.  $((\tau \models true) \wedge$ 
                       $(\tau \models (\lambda - . res) \triangleq if (self.boss \doteq null)$ 
                      then  $(Set\{self.salary\})$ 
                      else  $(self.boss.contents()$ 
                       $->including(self.salary))$ 
                      endif))
                      else invalid  $\tau$ ))

```

declare *dot_{Person}SALARY-def* [*simp del*]

declare *dot_{Person}BOSS-def* [*simp del*]

interpretation *contents* : *contract0 contents* $\lambda self. true$

```

 $\lambda self \ res. \ res \triangleq if (self.boss \doteq null)$ 
    then  $(Set\{self.salary\})$ 
    else  $(self.boss.contents()$ 
     $->including(self.salary))$ 
    endif

```

$\langle proof \rangle$

Specializing $\llbracket cp \ E; \tau \models \delta \ self; \tau \models true; \tau \models POST' \ self; \bigwedge res. (res \triangleq if \ self.boss \doteq null \ then \ Set\{self.salary\} \ else \ self.boss.contents() \rightarrow including(self.salary) \ endif) = (POST' \ self \ and \ (res \triangleq BODY \ self)) \rrbracket \implies (\tau \models E \ (self.contents())) = (\tau \models E \ (BODY \ self))$, one gets the following more practical rewrite rule that is amenable to symbolic evaluation:

theorem *unfold-contents* :

assumes *cp E*

and $\tau \models \delta \ self$

shows $(\tau \models E \ (self.contents())) =$
 $(\tau \models E \ (if \ self.boss \doteq null$
 $\quad then \ Set\{self.salary\}$
 $\quad else \ self.boss.contents() \rightarrow including(self.salary) \ endif))$

$\langle proof \rangle$

Since we have only one interpretation function, we need the corresponding operation on the pre-state:

consts *contentsATpre* :: *Person* \Rightarrow *Set-Integer* $((1(-).contents@pre'()) \ 50)$

axiomatization where *contentsATpre-def*:

```

(self).contents@pre() = ( $\lambda \tau. (if \tau \models (\delta \ self)$ 
    then SOME res.  $((\tau \models true) \wedge$ 
     $(\tau \models (self.boss.contents@pre'() \rightarrow including(self.salary))$ 
     $(\tau \models pre))$ 
    else invalid  $\tau$ ))

```

```

      (τ ⊨ ((λ-. res)  $\triangleq$  if (self).boss@pre  $\doteq$  null (* post *)
        then Set{(self).salary@pre}
        else (self).boss@pre .contents@pre()
               $\rightarrow$  including(self .salary@pre)
        endif)))
    else invalid τ))

declare dotPersonSALARY-at-pre-def [simp del]
declare dotPersonBOSS-at-pre-def [simp del]

interpretation contentsATpre : contract0 contentsATpre λ self. true
      λ self res. res  $\triangleq$  if (self .boss@pre  $\doteq$  null)
        then (Set{self .salary@pre})
        else (self .boss@pre .contents@pre()
               $\rightarrow$  including(self .salary@pre))
        endif

    <proof>

```

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  $\doteq$  null
      then Set{self .salary@pre}
      else self .boss@pre .contents@pre()  $\rightarrow$  including(self .salary@pre) endif))

    <proof>

```

Note that these @pre variants on methods are only available on queries, i. e., operations without side-effect.

7.13. The Contract of a User-defined Method

The example specification in high-level OCL input syntax reads as follows:

```

context Person::insert(x:Integer)
pre: true
post: contents():Set(Integer)
contents() = contents@pre() $\rightarrow$ including(x)

```

This boils down to:

```

definition insert :: Person  $\Rightarrow$  Integer  $\Rightarrow$  Void ((1(-).insert'(-)) 50)
where self .insert(x)  $\equiv$ 
  (λ τ. if (τ ⊨ (δ self)) ∧ (τ ⊨ v x)
    then SOME res. (τ ⊨ true ∧
      (τ ⊨ ((self).contents()  $\triangleq$  (self).contents@pre() $\rightarrow$ including(x))))
    else invalid τ)

```

The semantic consequences of this definition were computed inside this locale interpretation:

interpretation *insert* : *contract1 insert* $\lambda self\ x.\ true$
 $\lambda self\ x\ res.\ ((self.\ contents()) \triangleq$
 $(self.\ contents@pre() \rightarrow including(x)))$
 $\langle proof \rangle$

The result of this locale interpretation for our *Analysis-OCL.insert* contract is the following set of properties, which serves as basis for automated deduction on them:

Name	Theorem
<i>insert.strict0</i>	$(invalid.insert(X)) = invalid$
<i>insert.nullstrict0</i>	$(null.insert(X)) = invalid$
<i>insert.strict1</i>	$(self.insert(invalid)) = invalid$
<i>insert.cpPRE</i>	$true\ \tau = true\ \tau$
<i>insert.cpPOST</i>	$(self.contents() \triangleq self.contents@pre() \rightarrow including(a1.0))\ \tau =$ $(\lambda-. self\ \tau.contents() \triangleq \lambda-. self$ $\tau.contents@pre() \rightarrow including(\lambda-. a1.0\ \tau))\ \tau$
<i>insert.cp-pre</i>	$\llbracket cp\ self';\ cp\ a1' \rrbracket \implies cp\ (\lambda X.\ true)$
<i>insert.cp-post</i>	$\llbracket cp\ self';\ cp\ a1';\ cp\ res \rrbracket \implies cp\ (\lambda X.\ self'\ X.contents() \triangleq self'$ $X.contents@pre() \rightarrow including(a1'\ X))$
<i>insert.cp</i>	$\llbracket cp\ self';\ cp\ a1';\ cp\ res \rrbracket \implies cp\ (\lambda X.\ self'\ X.insert(a1'\ X))$
<i>insert.cp0</i>	$(self.insert(a1.0))\ \tau = (\lambda-. self\ \tau.insert(\lambda-. a1.0\ \tau))\ \tau$
<i>insert.def-scheme</i>	$self.insert(a1.0) \equiv \lambda\tau.\ if\ \tau \models \delta\ self \wedge \tau \models v\ a1.0\ then\ SOME$ $res.\ \tau \models true \wedge \tau \models self.contents() \triangleq$ $self.contents@pre() \rightarrow including(a1.0)\ else\ invalid\ \tau$
<i>insert.unfold</i>	$\llbracket cp\ E;\ \tau \models \delta\ self \wedge \tau \models v\ a1.0;\ \tau \models true;\ \exists res.\ \tau \models$ $self.contents() \triangleq self.contents@pre() \rightarrow including(a1.0);\ \bigwedge res.$ $\tau \models self.contents() \triangleq self.contents@pre() \rightarrow including(a1.0)$ $\implies \tau \models E\ (\lambda-. res) \rrbracket \implies \tau \models E\ (self.insert(a1.0))$
<i>insert.unfold2</i>	$\llbracket cp\ E;\ \tau \models \delta\ self \wedge \tau \models v\ a1.0;\ \tau \models true;\ \tau \models POST'\ self$ $a1.0;\ \bigwedge res.\ (self.contents() \triangleq$ $self.contents@pre() \rightarrow including(a1.0)) = (POST'\ self\ a1.0\ and$ $(res \triangleq BODY\ self\ a1.0)) \rrbracket \implies (\tau \models E\ (self.insert(a1.0))) =$ $(\tau \models E\ (BODY\ self\ a1.0))$

Table 7.1.: Semantic properties resulting from a user-defined operation contract.

end

8. Example II: The Employee Design Model (UML)

```
theory
  Design-UML
imports
  ../../../../src/UML-Main
begin
```

8.1. Introduction

For certain concepts like classes and class-types, only a generic definition for its resulting semantics can be given. Generic means, there is a function outside HOL that “compiles” a concrete, closed-world class diagram into a “theory” of this data model, consisting of a bunch of definitions for classes, accessors, method, casts, and tests for actual types, as well as proofs for the fundamental properties of these operations in this concrete data model.

Such generic function or “compiler” can be implemented in Isabelle on the ML level. This has been done, for a semantics following the open-world assumption, for UML 2.0 in [4, 6]. In this paper, we follow another approach for UML 2.4: we define the concepts of the compilation informally, and present a concrete example which is verified in Isabelle/HOL.

8.1.1. Outlining the Example

We are presenting here a “design-model” of the (slightly modified) example Figure 7.3, page 20 of the OCL standard [28]. To be precise, this theory contains the formalization of the data-part covered by the UML class model (see Figure 8.1):

This means that the association (attached to the association class **EmployeeRanking**) with the association ends **boss** and **employees** is implemented by the attribute **boss** and the operation **employees** (to be discussed in the OCL part captured by the subsequent theory).

8.2. Example Data-Universe and its Infrastructure

Ideally, the following is generated automatically from a UML class model.

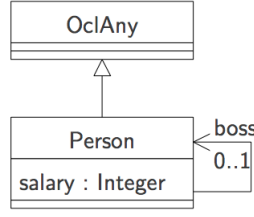


Figure 8.1.: A simple UML class model drawn from Figure 7.3, page 20 of [28].

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 $type_{Person} = mk_{Person} \text{ oid}$
 $int \text{ option}$
 $oid \text{ option}$

datatype $type_{OclAny} = mk_{OclAny} \text{ oid}$
 $(int \text{ option} \times oid \text{ option}) \text{ option}$

Now, we construct a concrete “universe of OclAny types” by injection into a sum type containing the class types. This type of OclAny will be used as instance for all respective type-variables.

datatype $\mathfrak{A} = in_{Person} type_{Person} \mid in_{OclAny} type_{OclAny}$

Having fixed the object universe, we can introduce type synonyms that exactly correspond to OCL types. Again, we exploit that our representation of OCL is a “shallow embedding” with a one-to-one correspondance of OCL-types to types of the meta-language HOL.

type-synonym $Boolean = \mathfrak{A} \text{ Boolean}$
type-synonym $Integer = \mathfrak{A} \text{ Integer}$
type-synonym $Void = \mathfrak{A} \text{ Void}$
type-synonym $OclAny = (\mathfrak{A}, type_{OclAny} \text{ option option}) \text{ val}$
type-synonym $Person = (\mathfrak{A}, type_{Person} \text{ option option}) \text{ val}$
type-synonym $Set-Integer = (\mathfrak{A}, int \text{ option option}) \text{ Set}$
type-synonym $Set-Person = (\mathfrak{A}, type_{Person} \text{ option option}) \text{ Set}$

Just a little check:

typ $Boolean$

To reuse key-elements of the library like referential equality, we have to show that the object universe belongs to the type class “oclany,” i. e., each class type has to provide a function *oid-of* yielding the object id (oid) of the object.

```

instantiation typePerson :: object
begin
  definition oid-of-typePerson-def: oid-of x = (case x of mkPerson oid - -  $\Rightarrow$  oid)
  instance  $\langle$ proof $\rangle$ 
end

instantiation typeOclAny :: object
begin
  definition oid-of-typeOclAny-def: oid-of x = (case x of mkOclAny oid - -  $\Rightarrow$  oid)
  instance  $\langle$ proof $\rangle$ 
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  $\langle$ proof $\rangle$ 
end

```

8.3. Instantiation of the Generic Strict Equality

We instantiate the referential equality on *Person* and *OclAny*

```

defs(overloaded)   StrictRefEqObject-Person   : (x::Person)  $\doteq$  y  $\equiv$  StrictRefEqObject x y
defs(overloaded)   StrictRefEqObject-OclAny   : (x::OclAny)  $\doteq$  y  $\equiv$  StrictRefEqObject x y

```

lemmas

```

cp-StrictRefEqObject[of x::Person y::Person  $\tau$ ,
  simplified StrictRefEqObject-Person[symmetric]]
cp-intro(9)      [of P::Person  $\Rightarrow$  PersonQ::Person  $\Rightarrow$  Person,
  simplified StrictRefEqObject-Person[symmetric] ]
StrictRefEqObject-def  [of x::Person y::Person,
  simplified StrictRefEqObject-Person[symmetric]]
StrictRefEqObject-defargs [of - x::Person y::Person,
  simplified StrictRefEqObject-Person[symmetric]]
StrictRefEqObject-strict1
  [of x::Person,
  simplified StrictRefEqObject-Person[symmetric]]
StrictRefEqObject-strict2
  [of x::Person,
  simplified StrictRefEqObject-Person[symmetric]]

```

For each Class *C*, we will have a casting operation *.oclAsType*(*C*), a test on the actual type *.oclIsTypeOf*(*C*) as well as its relaxed form *.oclIsKindOf*(*C*) (corresponding exactly to Java's *instanceof*-operator).

Thus, since we have two class-types in our concrete class hierarchy, we have two operations to declare and to provide two overloading definitions for the two static types.

8.4. OclAsType

8.4.1. Definition

consts $OclAsType_{OclAny} :: 'α \Rightarrow OclAny \ ((-) .oclAsType' (OclAny'))$

consts $OclAsType_{Person} :: 'α \Rightarrow Person \ ((-) .oclAsType' (Person'))$

definition $OclAsType_{OclAny}\text{-}\mathfrak{A} = (\lambda u. \text{case } u \text{ of } in_{OclAny} \ a \Rightarrow a \mid in_{Person} \ (mk_{Person} \ oid \ a \ b) \Rightarrow mk_{OclAny} \ oid \ [(a,b)]])$

lemma $OclAsType_{OclAny}\text{-}\mathfrak{A}\text{-some}$: $OclAsType_{OclAny}\text{-}\mathfrak{A} \ x \neq None$
 $\langle proof \rangle$

defs (overloaded) $OclAsType_{OclAny}\text{-}OclAny$:
 $(X :: OclAny) .oclAsType(OclAny) \equiv X$

defs (overloaded) $OclAsType_{OclAny}\text{-}Person$:
 $(X :: Person) .oclAsType(OclAny) \equiv$
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow invalid \ \tau$
 $\quad \mid \lfloor \perp \rfloor \Rightarrow null \ \tau$
 $\quad \mid \lfloor \lfloor mk_{Person} \ oid \ a \ b \rfloor \rfloor \Rightarrow \lfloor \lfloor (mk_{OclAny} \ oid \ [(a,b)]) \rfloor \rfloor)$

definition $OclAsType_{Person}\text{-}\mathfrak{A} = (\lambda u. \text{case } u \text{ of } in_{Person} \ p \Rightarrow \lfloor p \rfloor \mid in_{OclAny} \ (mk_{OclAny} \ oid \ [(a,b)]) \Rightarrow \lfloor mk_{Person} \ oid \ a \ b \rfloor \mid - \Rightarrow None)$

defs (overloaded) $OclAsType_{Person}\text{-}OclAny$:
 $(X :: OclAny) .oclAsType(Person) \equiv$
 $(\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow invalid \ \tau$
 $\quad \mid \lfloor \perp \rfloor \Rightarrow null \ \tau$
 $\quad \mid \lfloor \lfloor mk_{OclAny} \ oid \ \perp \rfloor \rfloor \Rightarrow invalid \ \tau \quad (* \text{down-cast exception} *)$
 $\quad \mid \lfloor \lfloor mk_{OclAny} \ oid \ [(a,b)] \rfloor \rfloor \Rightarrow \lfloor \lfloor mk_{Person} \ oid \ a \ b \rfloor \rfloor)$

defs (overloaded) $OclAsType_{Person}\text{-}Person$:
 $(X :: Person) .oclAsType(Person) \equiv X$

lemmas $[simp] =$
 $OclAsType_{OclAny}\text{-}OclAny$
 $OclAsType_{Person}\text{-}Person$

8.4.2. Context Passing

lemma $cp\text{-}OclAsType_{OclAny}\text{-}Person\text{-}Person$: $cp \ P \Longrightarrow cp(\lambda X. (P \ (X :: Person)) :: Person) .oclAsType(OclAny))$
 $\langle proof \rangle$

lemma $cp\text{-}OclAsType_{OclAny}\text{-}OclAny\text{-}OclAny$: $cp \ P \Longrightarrow cp(\lambda X. (P \ (X :: OclAny)) :: OclAny) .oclAsType(OclAny))$

<proof>

lemma *cp-OclAsType_{Person}-Person-Person*: *cp P* \implies *cp*($\lambda X. (P (X::Person)::Person)$
.oclAsType(Person))

<proof>

lemma *cp-OclAsType_{Person}-OclAny-OclAny*: *cp P* \implies *cp*($\lambda X. (P (X::OclAny)::OclAny)$
.oclAsType(Person))

<proof>

lemma *cp-OclAsType_{OclAny}-Person-OclAny*: *cp P* \implies *cp*($\lambda X. (P (X::Person)::OclAny)$
.oclAsType(OclAny))

<proof>

lemma *cp-OclAsType_{OclAny}-OclAny-Person*: *cp P* \implies *cp*($\lambda X. (P (X::OclAny)::Person)$
.oclAsType(OclAny))

<proof>

lemma *cp-OclAsType_{Person}-Person-OclAny*: *cp P* \implies *cp*($\lambda X. (P (X::Person)::OclAny)$
.oclAsType(Person))

<proof>

lemma *cp-OclAsType_{Person}-OclAny-Person*: *cp P* \implies *cp*($\lambda X. (P (X::OclAny)::Person)$
.oclAsType(Person))

<proof>

lemmas [*simp*] =

cp-OclAsType_{OclAny}-Person-Person
cp-OclAsType_{OclAny}-OclAny-OclAny
cp-OclAsType_{Person}-Person-Person
cp-OclAsType_{Person}-OclAny-OclAny

cp-OclAsType_{OclAny}-Person-OclAny
cp-OclAsType_{OclAny}-OclAny-Person
cp-OclAsType_{Person}-Person-OclAny
cp-OclAsType_{Person}-OclAny-Person

8.4.3. Execution with Invalid or Null as Argument

lemma *OclAsType_{OclAny}-OclAny-strict* : (*invalid::OclAny*) *.oclAsType(OclAny)* = *invalid*
<proof>

lemma *OclAsType_{OclAny}-OclAny-nullstrict* : (*null::OclAny*) *.oclAsType(OclAny)* = *null*
<proof>

lemma *OclAsType_{OclAny}-Person-strict*[*simp*] : (*invalid::Person*) *.oclAsType(OclAny)* = *invalid*
<proof>

lemma *OclAsType_{OclAny}-Person-nullstrict*[*simp*] : (*null::Person*) *.oclAsType(OclAny)* = *null*
<proof>

lemma *OclAsType_{Person}-OclAny-strict*[*simp*] : (*invalid::OclAny*) *.oclAsType(Person)* = *invalid*
<proof>

lemma $OclAsType_{Person-OclAny-nullstrict[simp]} : (null::OclAny) .oclAsType(Person) = null$
 $\langle proof \rangle$

lemma $OclAsType_{Person-Person-strict} : (invalid::Person) .oclAsType(Person) = invalid$
 $\langle proof \rangle$

lemma $OclAsType_{Person-Person-nullstrict} : (null::Person) .oclAsType(Person) = null$
 $\langle proof \rangle$

8.5. OclIsTypeOf

8.5.1. Definition

consts $OclIsTypeOf_{OclAny} :: 'α \Rightarrow Boolean \ ((-).oclIsTypeOf'(OclAny'))$
consts $OclIsTypeOf_{Person} :: 'α \Rightarrow Boolean \ ((-).oclIsTypeOf'(Person'))$

defs (overloaded) $OclIsTypeOf_{OclAny-OclAny}$:
 $(X::OclAny) .oclIsTypeOf(OclAny) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow invalid \ \tau$
 $\quad | \lfloor \perp \rfloor \Rightarrow true \ \tau \quad (* \text{ invalid } ?? *)$
 $\quad | \lfloor [mk_{OclAny} \text{ oid } \perp] \rfloor \Rightarrow true \ \tau$
 $\quad | \lfloor [mk_{OclAny} \text{ oid } _] \rfloor \Rightarrow false \ \tau)$

defs (overloaded) $OclIsTypeOf_{OclAny-Person}$:
 $(X::Person) .oclIsTypeOf(OclAny) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow invalid \ \tau$
 $\quad | \lfloor \perp \rfloor \Rightarrow true \ \tau \quad (* \text{ invalid } ?? *)$
 $\quad | \lfloor _ \rfloor \Rightarrow false \ \tau)$

defs (overloaded) $OclIsTypeOf_{Person-OclAny}$:
 $(X::OclAny) .oclIsTypeOf(Person) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow invalid \ \tau$
 $\quad | \lfloor \perp \rfloor \Rightarrow true \ \tau$
 $\quad | \lfloor [mk_{OclAny} \text{ oid } \perp] \rfloor \Rightarrow false \ \tau$
 $\quad | \lfloor [mk_{OclAny} \text{ oid } _] \rfloor \Rightarrow true \ \tau)$

defs (overloaded) $OclIsTypeOf_{Person-Person}$:
 $(X::Person) .oclIsTypeOf(Person) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow invalid \ \tau$
 $\quad | _ \Rightarrow true \ \tau)$

8.5.2. Context Passing

lemma $cp-OclIsTypeOf_{OclAny-Person-Person} : cp \quad P \quad \Rightarrow$
 $cp(\lambda X. (P(X::Person)::Person) .oclIsTypeOf(OclAny))$

$\langle proof \rangle$
lemma $cp-OclIsTypeOf_{OclAny-OclAny-OclAny}:$ cp P \implies
 $cp(\lambda X.(P(X::OclAny)::OclAny).oclIsTypeOf(OclAny))$
 $\langle proof \rangle$
lemma $cp-OclIsTypeOf_{Person-Person-Person}:$ cp P \implies
 $cp(\lambda X.(P(X::Person)::Person).oclIsTypeOf(Person))$
 $\langle proof \rangle$
lemma $cp-OclIsTypeOf_{Person-OclAny-OclAny}:$ cp P \implies
 $cp(\lambda X.(P(X::OclAny)::OclAny).oclIsTypeOf(Person))$
 $\langle proof \rangle$

lemma $cp-OclIsTypeOf_{OclAny-Person-OclAny}:$ cp P \implies
 $cp(\lambda X.(P(X::Person)::OclAny).oclIsTypeOf(OclAny))$
 $\langle proof \rangle$
lemma $cp-OclIsTypeOf_{OclAny-OclAny-Person}:$ cp P \implies
 $cp(\lambda X.(P(X::OclAny)::Person).oclIsTypeOf(OclAny))$
 $\langle proof \rangle$
lemma $cp-OclIsTypeOf_{Person-Person-OclAny}:$ cp P \implies
 $cp(\lambda X.(P(X::Person)::OclAny).oclIsTypeOf(Person))$
 $\langle proof \rangle$
lemma $cp-OclIsTypeOf_{Person-OclAny-Person}:$ cp P \implies
 $cp(\lambda X.(P(X::OclAny)::Person).oclIsTypeOf(Person))$
 $\langle proof \rangle$

lemmas $[simp] =$
 $cp-OclIsTypeOf_{OclAny-Person-Person}$
 $cp-OclIsTypeOf_{OclAny-OclAny-OclAny}$
 $cp-OclIsTypeOf_{Person-Person-Person}$
 $cp-OclIsTypeOf_{Person-OclAny-OclAny}$

 $cp-OclIsTypeOf_{OclAny-Person-OclAny}$
 $cp-OclIsTypeOf_{OclAny-OclAny-Person}$
 $cp-OclIsTypeOf_{Person-Person-OclAny}$
 $cp-OclIsTypeOf_{Person-OclAny-Person}$

8.5.3. Execution with Invalid or Null as Argument

lemma $OclIsTypeOf_{OclAny-OclAny-strict1}[simp]:$
 $(invalid::OclAny).oclIsTypeOf(OclAny) = invalid$
 $\langle proof \rangle$
lemma $OclIsTypeOf_{OclAny-OclAny-strict2}[simp]:$
 $(null::OclAny).oclIsTypeOf(OclAny) = true$
 $\langle proof \rangle$
lemma $OclIsTypeOf_{OclAny-Person-strict1}[simp]:$
 $(invalid::Person).oclIsTypeOf(OclAny) = invalid$
 $\langle proof \rangle$
lemma $OclIsTypeOf_{OclAny-Person-strict2}[simp]:$
 $(null::Person).oclIsTypeOf(OclAny) = true$

$\langle \text{proof} \rangle$
lemma $\text{OclIsTypeOf}_{\text{Person-OclAny-strict1}}[\text{simp}]$:
 $(\text{invalid}::\text{OclAny}) .\text{ocIsTypeOf}(\text{Person}) = \text{invalid}$
 $\langle \text{proof} \rangle$
lemma $\text{OclIsTypeOf}_{\text{Person-OclAny-strict2}}[\text{simp}]$:
 $(\text{null}::\text{OclAny}) .\text{ocIsTypeOf}(\text{Person}) = \text{true}$
 $\langle \text{proof} \rangle$
lemma $\text{OclIsTypeOf}_{\text{Person-Person-strict1}}[\text{simp}]$:
 $(\text{invalid}::\text{Person}) .\text{ocIsTypeOf}(\text{Person}) = \text{invalid}$
 $\langle \text{proof} \rangle$
lemma $\text{OclIsTypeOf}_{\text{Person-Person-strict2}}[\text{simp}]$:
 $(\text{null}::\text{Person}) .\text{ocIsTypeOf}(\text{Person}) = \text{true}$
 $\langle \text{proof} \rangle$

8.5.4. Up Down Casting

lemma $\text{actualType-larger-staticType}$:
assumes $\text{isdef}: \tau \models (\delta \ X)$
shows $\tau \models (X::\text{Person}) .\text{ocIsTypeOf}(\text{OclAny}) \triangleq \text{false}$
 $\langle \text{proof} \rangle$

lemma down-cast-type :
assumes $\text{isOclAny}: \tau \models (X::\text{OclAny}) .\text{ocIsTypeOf}(\text{OclAny})$
and $\text{non-null}: \tau \models (\delta \ X)$
shows $\tau \models (X .\text{oclAsType}(\text{Person})) \triangleq \text{invalid}$
 $\langle \text{proof} \rangle$

lemma $\text{down-cast-type}'$:
assumes $\text{isOclAny}: \tau \models (X::\text{OclAny}) .\text{ocIsTypeOf}(\text{OclAny})$
and $\text{non-null}: \tau \models (\delta \ X)$
shows $\tau \models \text{not } (v \ (X .\text{oclAsType}(\text{Person})))$
 $\langle \text{proof} \rangle$

lemma up-down-cast :
assumes $\text{isdef}: \tau \models (\delta \ X)$
shows $\tau \models ((X::\text{Person}) .\text{oclAsType}(\text{OclAny}) .\text{oclAsType}(\text{Person}) \triangleq X)$
 $\langle \text{proof} \rangle$

lemma $\text{up-down-cast-Person-OclAny-Person}[\text{simp}]$:
shows $((X::\text{Person}) .\text{oclAsType}(\text{OclAny}) .\text{oclAsType}(\text{Person}) = X)$
 $\langle \text{proof} \rangle$

lemma $\text{up-down-cast-Person-OclAny-Person}'$: **assumes** $\tau \models v \ X$
shows $\tau \models (((X :: \text{Person}) .\text{oclAsType}(\text{OclAny}) .\text{oclAsType}(\text{Person})) \doteq X)$
 $\langle \text{proof} \rangle$

lemma $\text{up-down-cast-Person-OclAny-Person}''$: **assumes** $\tau \models v \ (X :: \text{Person})$
shows $\tau \models (X .\text{ocIsTypeOf}(\text{Person}) \text{ implies } (X .\text{oclAsType}(\text{OclAny}) .\text{oclAsType}(\text{Person})) \doteq$

$X)$
 $\langle proof \rangle$

8.6. OclIsKindOf

8.6.1. Definition

consts $OclIsKindOf_{OclAny} :: 'α \Rightarrow Boolean \ ((-).oclIsKindOf'(OclAny'))$
consts $OclIsKindOf_{Person} :: 'α \Rightarrow Boolean \ ((-).oclIsKindOf'(Person'))$

defs (overloaded) $OclIsKindOf_{OclAny-OclAny}$:
 $(X::OclAny).oclIsKindOf(OclAny) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | - \Rightarrow \text{true } \tau)$

defs (overloaded) $OclIsKindOf_{OclAny-Person}$:
 $(X::Person).oclIsKindOf(OclAny) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | - \Rightarrow \text{true } \tau)$

defs (overloaded) $OclIsKindOf_{Person-OclAny}$:
 $(X::OclAny).oclIsKindOf(Person) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | [\perp] \Rightarrow \text{true } \tau$
 $\quad | [[mk_{OclAny} \text{ oid } \perp]] \Rightarrow \text{false } \tau$
 $\quad | [[mk_{OclAny} \text{ oid } [-]]] \Rightarrow \text{true } \tau)$

defs (overloaded) $OclIsKindOf_{Person-Person}$:
 $(X::Person).oclIsKindOf(Person) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | - \Rightarrow \text{true } \tau)$

8.6.2. Context Passing

lemma $cp-OclIsKindOf_{OclAny-Person-Person}$: $cp \quad P \quad \Rightarrow$
 $cp(\lambda X.(P(X::Person)::Person).oclIsKindOf(OclAny))$
 $\langle proof \rangle$

lemma $cp-OclIsKindOf_{OclAny-OclAny-OclAny}$: $cp \quad P \quad \Rightarrow$
 $cp(\lambda X.(P(X::OclAny)::OclAny).oclIsKindOf(OclAny))$
 $\langle proof \rangle$

lemma $cp-OclIsKindOf_{Person-Person-Person}$: $cp \quad P \quad \Rightarrow$
 $cp(\lambda X.(P(X::Person)::Person).oclIsKindOf(Person))$
 $\langle proof \rangle$

lemma $cp\text{-}OclIsKindOf_{Person-OclAny-OclAny} :$ cp P \implies
 $cp(\lambda X.(P(X::OclAny)::OclAny).oclIsKindOf(Person))$
 $\langle proof \rangle$

lemma $cp\text{-}OclIsKindOf_{OclAny-Person-OclAny} :$ cp P \implies
 $cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(OclAny))$
 $\langle proof \rangle$

lemma $cp\text{-}OclIsKindOf_{OclAny-OclAny-Person} :$ cp P \implies
 $cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(OclAny))$
 $\langle proof \rangle$

lemma $cp\text{-}OclIsKindOf_{Person-Person-OclAny} :$ cp P \implies
 $cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(Person))$
 $\langle proof \rangle$

lemma $cp\text{-}OclIsKindOf_{Person-OclAny-Person} :$ cp P \implies
 $cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(Person))$
 $\langle proof \rangle$

lemmas $[simp] =$
 $cp\text{-}OclIsKindOf_{OclAny-Person-Person}$
 $cp\text{-}OclIsKindOf_{OclAny-OclAny-OclAny}$
 $cp\text{-}OclIsKindOf_{Person-Person-Person}$
 $cp\text{-}OclIsKindOf_{Person-OclAny-OclAny}$

 $cp\text{-}OclIsKindOf_{OclAny-Person-OclAny}$
 $cp\text{-}OclIsKindOf_{OclAny-OclAny-Person}$
 $cp\text{-}OclIsKindOf_{Person-Person-OclAny}$
 $cp\text{-}OclIsKindOf_{Person-OclAny-Person}$

8.6.3. Execution with Invalid or Null as Argument

lemma $OclIsKindOf_{OclAny-OclAny-strict1}[simp] : (invalid::OclAny) .oclIsKindOf(OclAny) =$
 $invalid$
 $\langle proof \rangle$

lemma $OclIsKindOf_{OclAny-OclAny-strict2}[simp] : (null::OclAny) .oclIsKindOf(OclAny) =$
 $true$
 $\langle proof \rangle$

lemma $OclIsKindOf_{OclAny-Person-strict1}[simp] : (invalid::Person) .oclIsKindOf(OclAny) =$
 $invalid$
 $\langle proof \rangle$

lemma $OclIsKindOf_{OclAny-Person-strict2}[simp] : (null::Person) .oclIsKindOf(OclAny) = true$
 $\langle proof \rangle$

lemma $OclIsKindOf_{Person-OclAny-strict1}[simp] : (invalid::OclAny) .oclIsKindOf(Person) =$
 $invalid$
 $\langle proof \rangle$

lemma *OclIsKindOf_{Person}-OclAny-strict2*[simp]: (*null*::*OclAny*) .*oclIsKindOf*(*Person*) = *true*
 ⟨*proof*⟩

lemma *OclIsKindOf_{Person}-Person-strict1*[simp]: (*invalid*::*Person*) .*oclIsKindOf*(*Person*) = *invalid*
 ⟨*proof*⟩

lemma *OclIsKindOf_{Person}-Person-strict2*[simp]: (*null*::*Person*) .*oclIsKindOf*(*Person*) = *true*
 ⟨*proof*⟩

8.6.4. Up Down Casting

lemma *actualKind-larger-staticKind*:
assumes *isdef*: $\tau \models (\delta \ X)$
shows $\tau \models ((X::\textit{Person}) .\textit{oclIsKindOf}(\textit{OclAny}) \triangleq \textit{true})$
 ⟨*proof*⟩

lemma *down-cast-kind*:
assumes *isOclAny*: $\neg (\tau \models ((X::\textit{OclAny}).\textit{oclIsKindOf}(\textit{Person})))$
and *non-null*: $\tau \models (\delta \ X)$
shows $\tau \models ((X .\textit{oclAsType}(\textit{Person})) \triangleq \textit{invalid})$
 ⟨*proof*⟩

8.7. 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* $\equiv \textit{OclAsType}_{\textit{Person}}\text{-}\mathfrak{A}$
definition *OclAny* $\equiv \textit{OclAsType}_{\textit{OclAny}}\text{-}\mathfrak{A}$
lemmas [simp] = *Person-def OclAny-def*

lemma *OclAllInstances-generic_{OclAny-exec}*: *OclAllInstances-generic pre-post OclAny* =
 ($\lambda\tau. \textit{Abs-Set}_{\textit{base}} \llbracket \textit{Some } ' \textit{OclAny } ' \textit{ran } (\textit{heap } (\textit{pre-post } \tau)) \rrbracket$)
 ⟨*proof*⟩

lemma *OclAllInstances-at-post_{OclAny-exec}*: *OclAny .allInstances*() =
 ($\lambda\tau. \textit{Abs-Set}_{\textit{base}} \llbracket \textit{Some } ' \textit{OclAny } ' \textit{ran } (\textit{heap } (\textit{snd } \tau)) \rrbracket$)
 ⟨*proof*⟩

lemma *OclAllInstances-at-pre_{OclAny-exec}*: *OclAny .allInstances@pre*() =
 ($\lambda\tau. \textit{Abs-Set}_{\textit{base}} \llbracket \textit{Some } ' \textit{OclAny } ' \textit{ran } (\textit{heap } (\textit{fst } \tau)) \rrbracket$)
 ⟨*proof*⟩

8.7.1. OclIsTypeOf

lemma *OclAny-allInstances-generic-oclIsTypeOf_{OclAny1}*:
assumes [simp]: $\bigwedge x. \textit{pre-post } (x, x) = x$

shows $\exists \tau. (\tau \models ((\text{OclAllInstances-generic } \text{pre-post } \text{OclAny}) \rightarrow \text{forAll}(X|X .\text{oclIsTypeOf}(\text{OclAny}))))$
 $\langle \text{proof} \rangle$

lemma *OclAny-allInstances-at-post-oclIsTypeOf_{OclAny}1*:
 $\exists \tau. (\tau \models (\text{OclAny} .\text{allInstances}() \rightarrow \text{forAll}(X|X .\text{oclIsTypeOf}(\text{OclAny}))))$
 $\langle \text{proof} \rangle$

lemma *OclAny-allInstances-at-pre-oclIsTypeOf_{OclAny}1*:
 $\exists \tau. (\tau \models (\text{OclAny} .\text{allInstances}@pre() \rightarrow \text{forAll}(X|X .\text{oclIsTypeOf}(\text{OclAny}))))$
 $\langle \text{proof} \rangle$

lemma *OclAny-allInstances-generic-oclIsTypeOf_{OclAny}2*:
assumes $[simp]: \bigwedge x. \text{pre-post } (x, x) = x$
shows $\exists \tau. (\tau \models \text{not } ((\text{OclAllInstances-generic } \text{pre-post } \text{OclAny}) \rightarrow \text{forAll}(X|X .\text{oclIsTypeOf}(\text{OclAny}))))$
 $\langle \text{proof} \rangle$

lemma *OclAny-allInstances-at-post-oclIsTypeOf_{OclAny}2*:
 $\exists \tau. (\tau \models \text{not } (\text{OclAny} .\text{allInstances}() \rightarrow \text{forAll}(X|X .\text{oclIsTypeOf}(\text{OclAny}))))$
 $\langle \text{proof} \rangle$

lemma *OclAny-allInstances-at-pre-oclIsTypeOf_{OclAny}2*:
 $\exists \tau. (\tau \models \text{not } (\text{OclAny} .\text{allInstances}@pre() \rightarrow \text{forAll}(X|X .\text{oclIsTypeOf}(\text{OclAny}))))$
 $\langle \text{proof} \rangle$

lemma *Person-allInstances-generic-oclIsTypeOf_{Person}*:
 $\tau \models ((\text{OclAllInstances-generic } \text{pre-post } \text{Person}) \rightarrow \text{forAll}(X|X .\text{oclIsTypeOf}(\text{Person})))$
 $\langle \text{proof} \rangle$

lemma *Person-allInstances-at-post-oclIsTypeOf_{Person}*:
 $\tau \models (\text{Person} .\text{allInstances}() \rightarrow \text{forAll}(X|X .\text{oclIsTypeOf}(\text{Person})))$
 $\langle \text{proof} \rangle$

lemma *Person-allInstances-at-pre-oclIsTypeOf_{Person}*:
 $\tau \models (\text{Person} .\text{allInstances}@pre() \rightarrow \text{forAll}(X|X .\text{oclIsTypeOf}(\text{Person})))$
 $\langle \text{proof} \rangle$

8.7.2. OclIsKindOf

lemma *OclAny-allInstances-generic-oclIsKindOf_{OclAny}*:
 $\tau \models ((\text{OclAllInstances-generic } \text{pre-post } \text{OclAny}) \rightarrow \text{forAll}(X|X .\text{oclIsKindOf}(\text{OclAny})))$
 $\langle \text{proof} \rangle$

lemma *OclAny-allInstances-at-post-oclIsKindOf_{OclAny}*:
 $\tau \models (\text{OclAny} .\text{allInstances}() \rightarrow \text{forAll}(X|X .\text{oclIsKindOf}(\text{OclAny})))$
 $\langle \text{proof} \rangle$

lemma *OclAny-allInstances-at-pre-oclIsKindOf_{OclAny}*:

$\tau \models (\text{OclAny} . \text{allInstances}@pre() \rightarrow \text{forAll}(X|X . \text{oclIsKindOf}(\text{OclAny})))$
 $\langle \text{proof} \rangle$

lemma *Person-allInstances-generic-oclIsKindOf_{OclAny}*:
 $\tau \models ((\text{OclAllInstances-generic pre-post Person}) \rightarrow \text{forAll}(X|X . \text{oclIsKindOf}(\text{OclAny})))$
 $\langle \text{proof} \rangle$

lemma *Person-allInstances-at-post-oclIsKindOf_{OclAny}*:
 $\tau \models (\text{Person} . \text{allInstances}() \rightarrow \text{forAll}(X|X . \text{oclIsKindOf}(\text{OclAny})))$
 $\langle \text{proof} \rangle$

lemma *Person-allInstances-at-pre-oclIsKindOf_{OclAny}*:
 $\tau \models (\text{Person} . \text{allInstances}@pre() \rightarrow \text{forAll}(X|X . \text{oclIsKindOf}(\text{OclAny})))$
 $\langle \text{proof} \rangle$

lemma *Person-allInstances-generic-oclIsKindOf_{Person}*:
 $\tau \models ((\text{OclAllInstances-generic pre-post Person}) \rightarrow \text{forAll}(X|X . \text{oclIsKindOf}(\text{Person})))$
 $\langle \text{proof} \rangle$

lemma *Person-allInstances-at-post-oclIsKindOf_{Person}*:
 $\tau \models (\text{Person} . \text{allInstances}() \rightarrow \text{forAll}(X|X . \text{oclIsKindOf}(\text{Person})))$
 $\langle \text{proof} \rangle$

lemma *Person-allInstances-at-pre-oclIsKindOf_{Person}*:
 $\tau \models (\text{Person} . \text{allInstances}@pre() \rightarrow \text{forAll}(X|X . \text{oclIsKindOf}(\text{Person})))$
 $\langle \text{proof} \rangle$

8.8. The Accessors (any, boss, salary)

Should be generated entirely from a class-diagram.

8.8.1. Definition

definition *eval-extract* :: ($\mathfrak{A}, ('a::\text{object}) \text{ option option} \text{ val}$
 $\Rightarrow (\text{oid} \Rightarrow (\mathfrak{A}, 'c::\text{null}) \text{ val})$
 $\Rightarrow (\mathfrak{A}, 'c::\text{null}) \text{ val}$
where *eval-extract* $X f = (\lambda \tau . \text{case } X \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau \quad (* \text{exception propagation} *)$
 $\quad | \lfloor \perp \rfloor \Rightarrow \text{invalid } \tau \quad (* \text{dereferencing null pointer} *)$
 $\quad | \lfloor \lfloor \text{obj} \rfloor \rfloor \Rightarrow f (\text{oid-of obj}) \tau)$

definition *deref-oid_{Person}* :: ($\mathfrak{A} \text{ state} \times \mathfrak{A} \text{ state} \Rightarrow \mathfrak{A} \text{ state}$)
 $\Rightarrow (\text{type}_{\text{Person}} \Rightarrow (\mathfrak{A}, 'c::\text{null}) \text{ val})$
 $\Rightarrow \text{oid}$
 $\Rightarrow (\mathfrak{A}, 'c::\text{null}) \text{ val}$
where *deref-oid_{Person}* $\text{fst-snd } f \text{ oid} = (\lambda \tau . \text{case } (\text{heap } (\text{fst-snd } \tau)) \text{ oid of}$

$$\begin{array}{l} \lfloor \text{in}_{\text{Person}} \text{ obj} \rfloor \Rightarrow f \text{ obj } \tau \\ | - \quad \Rightarrow \text{invalid } \tau \end{array}$$

definition $\text{deref-oid}_{\text{OclAny}} :: (\mathfrak{A} \text{ state} \times \mathfrak{A} \text{ state} \Rightarrow \mathfrak{A} \text{ state})$
 $\Rightarrow (\text{type}_{\text{OclAny}} \Rightarrow (\mathfrak{A}, 'c::\text{null})\text{val})$
 $\Rightarrow \text{oid}$
 $\Rightarrow (\mathfrak{A}, 'c::\text{null})\text{val}$

where $\text{deref-oid}_{\text{OclAny}} \text{ fst-snd } f \text{ oid} = (\lambda \tau. \text{case } (\text{heap } (\text{fst-snd } \tau)) \text{ oid of}$
 $\lfloor \text{in}_{\text{OclAny}} \text{ obj} \rfloor \Rightarrow f \text{ obj } \tau$
 $| - \quad \Rightarrow \text{invalid } \tau)$

pointer undefined in state or not referencing a type conform object representation

definition $\text{select}_{\text{OclAny}} \mathcal{ANY} f = (\lambda X. \text{case } X \text{ of}$
 $(\text{mk}_{\text{OclAny}} - \perp) \Rightarrow \text{null}$
 $| (\text{mk}_{\text{OclAny}} - \lfloor \text{any} \rfloor) \Rightarrow f (\lambda x -. \lfloor \lfloor x \rfloor \rfloor) \text{ any})$

definition $\text{select}_{\text{Person}} \mathcal{BOSS} f = (\lambda X. \text{case } X \text{ of}$
 $(\text{mk}_{\text{Person}} - - \perp) \Rightarrow \text{null } (* \text{ object contains null pointer } *)$
 $| (\text{mk}_{\text{Person}} - - \lfloor \text{boss} \rfloor) \Rightarrow f (\lambda x -. \lfloor \lfloor x \rfloor \rfloor) \text{ boss})$

definition $\text{select}_{\text{Person}} \mathcal{SALARY} f = (\lambda X. \text{case } X \text{ of}$
 $(\text{mk}_{\text{Person}} - \perp -) \Rightarrow \text{null}$
 $| (\text{mk}_{\text{Person}} - \lfloor \text{salary} \rfloor -) \Rightarrow f (\lambda x -. \lfloor \lfloor x \rfloor \rfloor) \text{ salary})$

definition $\text{in-pre-state} = \text{fst}$

definition $\text{in-post-state} = \text{snd}$

definition $\text{reconst-basetype} = (\lambda \text{convert } x. \text{convert } x)$

definition $\text{dot}_{\text{OclAny}} \mathcal{ANY} :: \text{OclAny} \Rightarrow - \ ((1(-).\text{any}) \ 50)$
where $(X).\text{any} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{OclAny}} \text{ in-post-state}$
 $(\text{select}_{\text{OclAny}} \mathcal{ANY}$
 $\text{reconst-basetype}))$

definition $\text{dot}_{\text{Person}} \mathcal{BOSS} :: \text{Person} \Rightarrow \text{Person} \ ((1(-).\text{boss}) \ 50)$
where $(X).\text{boss} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{Person}} \text{ in-post-state}$
 $(\text{select}_{\text{Person}} \mathcal{BOSS}$
 $(\text{deref-oid}_{\text{Person}} \text{ in-post-state})))$

definition $\text{dot}_{\text{Person}} \mathcal{SALARY} :: \text{Person} \Rightarrow \text{Integer} \ ((1(-).\text{salary}) \ 50)$
where $(X).\text{salary} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{Person}} \text{ in-post-state})$

(*select*_{Person}*SALARY*
reconst-basetype))

definition *dot*_{OclAny}*ANY-at-pre* :: *OclAny* \Rightarrow - ((1(-).any@pre) 50)
where (X).any@pre = eval-extract X
 (deref-oid_{OclAny} in-pre-state
 (select_{OclAny}*ANY*
 reconst-basetype))

definition *dot*_{Person}*BOSS-at-pre*:: *Person* \Rightarrow *Person* ((1(-).boss@pre) 50)
where (X).boss@pre = eval-extract X
 (deref-oid_{Person} in-pre-state
 (select_{Person}*BOSS*
 (deref-oid_{Person} in-pre-state)))

definition *dot*_{Person}*SALARY-at-pre*:: *Person* \Rightarrow *Integer* ((1(-).salary@pre) 50)
where (X).salary@pre = eval-extract X
 (deref-oid_{Person} in-pre-state
 (select_{Person}*SALARY*
 reconst-basetype))

lemmas [simp] =
*dot*_{OclAny}*ANY-def*
*dot*_{Person}*BOSS-def*
*dot*_{Person}*SALARY-def*
*dot*_{OclAny}*ANY-at-pre-def*
*dot*_{Person}*BOSS-at-pre-def*
*dot*_{Person}*SALARY-at-pre-def*

8.8.2. Context Passing

lemmas [simp] = eval-extract-def

lemma *cp-dot*_{OclAny}*ANY*: ((X).any) τ = ((λ -. X τ).any) τ <proof>

lemma *cp-dot*_{Person}*BOSS*: ((X).boss) τ = ((λ -. X τ).boss) τ <proof>

lemma *cp-dot*_{Person}*SALARY*: ((X).salary) τ = ((λ -. X τ).salary) τ <proof>

lemma *cp-dot*_{OclAny}*ANY-at-pre*: ((X).any@pre) τ = ((λ -. X τ).any@pre) τ <proof>

lemma *cp-dot*_{Person}*BOSS-at-pre*: ((X).boss@pre) τ = ((λ -. X τ).boss@pre) τ <proof>

lemma *cp-dot*_{Person}*SALARY-at-pre*: ((X).salary@pre) τ = ((λ -. X τ).salary@pre) τ <proof>

lemmas *cp-dot*_{OclAny}*ANY-I* [simp, intro!]=
*cp-dot*_{OclAny}*ANY*[*THEN* allI[*THEN* allI],
 of λ X -. X λ - τ . τ , *THEN* cpI1]

lemmas *cp-dot*_{OclAny}*ANY-at-pre-I* [simp, intro!]=
*cp-dot*_{OclAny}*ANY-at-pre*[*THEN* allI[*THEN* allI],
 of λ X -. X λ - τ . τ , *THEN* cpI1]

lemmas *cp-dot*_{Person}*BOSS-I* [simp, intro!]=

$cp\text{-}dot_{Person}BOSS[THEN\ allI[THEN\ allI],$
 $of\ \lambda\ X\ \neg.\ X\ \lambda\ \neg.\ \tau.\ \tau,\ THEN\ cpI1]$
lemmas $cp\text{-}dot_{Person}BOSS\text{-}at\text{-}pre\text{-}I\ [simp,\ intro!]=$
 $cp\text{-}dot_{Person}BOSS\text{-}at\text{-}pre[THEN\ allI[THEN\ allI],$
 $of\ \lambda\ X\ \neg.\ X\ \lambda\ \neg.\ \tau.\ \tau,\ THEN\ cpI1]$

lemmas $cp\text{-}dot_{Person}SALARY\text{-}I\ [simp,\ intro!]=$
 $cp\text{-}dot_{Person}SALARY[THEN\ allI[THEN\ allI],$
 $of\ \lambda\ X\ \neg.\ X\ \lambda\ \neg.\ \tau.\ \tau,\ THEN\ cpI1]$
lemmas $cp\text{-}dot_{Person}SALARY\text{-}at\text{-}pre\text{-}I\ [simp,\ intro!]=$
 $cp\text{-}dot_{Person}SALARY\text{-}at\text{-}pre[THEN\ allI[THEN\ allI],$
 $of\ \lambda\ X\ \neg.\ X\ \lambda\ \neg.\ \tau.\ \tau,\ THEN\ cpI1]$

8.8.3. Execution with Invalid or Null as Argument

lemma $dot_{OclAny}ANY\text{-}nullstrict\ [simp]: (null).any = invalid$
 $\langle proof \rangle$
lemma $dot_{OclAny}ANY\text{-}at\text{-}pre\text{-}nullstrict\ [simp] : (null).any@pre = invalid$
 $\langle proof \rangle$
lemma $dot_{OclAny}ANY\text{-}strict\ [simp] : (invalid).any = invalid$
 $\langle proof \rangle$
lemma $dot_{OclAny}ANY\text{-}at\text{-}pre\text{-}strict\ [simp] : (invalid).any@pre = invalid$
 $\langle proof \rangle$

lemma $dot_{Person}BOSS\text{-}nullstrict\ [simp]: (null).boss = invalid$
 $\langle proof \rangle$
lemma $dot_{Person}BOSS\text{-}at\text{-}pre\text{-}nullstrict\ [simp] : (null).boss@pre = invalid$
 $\langle proof \rangle$
lemma $dot_{Person}BOSS\text{-}strict\ [simp] : (invalid).boss = invalid$
 $\langle proof \rangle$
lemma $dot_{Person}BOSS\text{-}at\text{-}pre\text{-}strict\ [simp] : (invalid).boss@pre = invalid$
 $\langle proof \rangle$

lemma $dot_{Person}SALARY\text{-}nullstrict\ [simp]: (null).salary = invalid$
 $\langle proof \rangle$
lemma $dot_{Person}SALARY\text{-}at\text{-}pre\text{-}nullstrict\ [simp] : (null).salary@pre = invalid$
 $\langle proof \rangle$
lemma $dot_{Person}SALARY\text{-}strict\ [simp] : (invalid).salary = invalid$
 $\langle proof \rangle$
lemma $dot_{Person}SALARY\text{-}at\text{-}pre\text{-}strict\ [simp] : (invalid).salary@pre = invalid$
 $\langle proof \rangle$

8.9. A Little Infra-structure on Example States

The example we are defining in this section comes from the figure 8.2.

definition $OclInt1000\ (1000)\ \text{where}\ OclInt1000 = (\lambda\ \neg.\ \llbracket 1000 \rrbracket)$

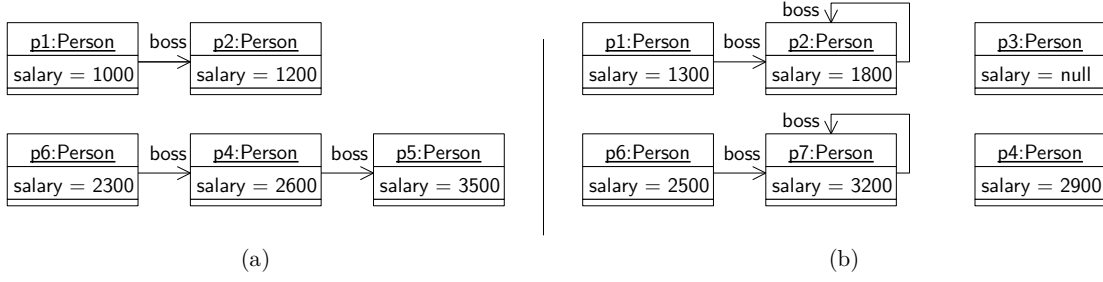


Figure 8.2.: (a) pre-state σ_1 and (b) post-state σ'_1 .

```

definition OclInt1200 (1200) where OclInt1200 = ( $\lambda$  . .  $\llbracket 1200 \rrbracket$ )
definition OclInt1300 (1300) where OclInt1300 = ( $\lambda$  . .  $\llbracket 1300 \rrbracket$ )
definition OclInt1800 (1800) where OclInt1800 = ( $\lambda$  . .  $\llbracket 1800 \rrbracket$ )
definition OclInt2600 (2600) where OclInt2600 = ( $\lambda$  . .  $\llbracket 2600 \rrbracket$ )
definition OclInt2900 (2900) where OclInt2900 = ( $\lambda$  . .  $\llbracket 2900 \rrbracket$ )
definition OclInt3200 (3200) where OclInt3200 = ( $\lambda$  . .  $\llbracket 3200 \rrbracket$ )
definition OclInt3500 (3500) where OclInt3500 = ( $\lambda$  . .  $\llbracket 3500 \rrbracket$ )

```

```

definition oid0  $\equiv$  0
definition oid1  $\equiv$  1
definition oid2  $\equiv$  2
definition oid3  $\equiv$  3
definition oid4  $\equiv$  4
definition oid5  $\equiv$  5
definition oid6  $\equiv$  6
definition oid7  $\equiv$  7
definition oid8  $\equiv$  8

```

```

definition person1  $\equiv$  mkPerson oid0  $\llbracket 1300 \rrbracket$   $\llbracket \text{oid1} \rrbracket$ 
definition person2  $\equiv$  mkPerson oid1  $\llbracket 1800 \rrbracket$   $\llbracket \text{oid1} \rrbracket$ 
definition person3  $\equiv$  mkPerson oid2 None None
definition person4  $\equiv$  mkPerson oid3  $\llbracket 2900 \rrbracket$  None
definition person5  $\equiv$  mkPerson oid4  $\llbracket 3500 \rrbracket$  None
definition person6  $\equiv$  mkPerson oid5  $\llbracket 2500 \rrbracket$   $\llbracket \text{oid6} \rrbracket$ 
definition person7  $\equiv$  mkOclAny oid6  $\llbracket (\llbracket 3200 \rrbracket, \llbracket \text{oid6} \rrbracket) \rrbracket$ 
definition person8  $\equiv$  mkOclAny oid7 None
definition person9  $\equiv$  mkPerson oid8  $\llbracket 0 \rrbracket$  None

```

definition

```

 $\sigma_1 \equiv$  ( $\llbracket$  heap = empty(oid0  $\mapsto$  inPerson (mkPerson oid0  $\llbracket 1000 \rrbracket$   $\llbracket \text{oid1} \rrbracket$ ))
  (oid1  $\mapsto$  inPerson (mkPerson oid1  $\llbracket 1200 \rrbracket$  None))
  (*oid2*)
  (oid3  $\mapsto$  inPerson (mkPerson oid3  $\llbracket 2600 \rrbracket$   $\llbracket \text{oid4} \rrbracket$ ))
  (oid4  $\mapsto$  inPerson person5)
  (oid5  $\mapsto$  inPerson (mkPerson oid5  $\llbracket 2300 \rrbracket$   $\llbracket \text{oid3} \rrbracket$ ))
  (*oid6*))

```

$(*oid7*)$
 $(oid8 \mapsto in_{Person} person9),$
 $assocs = empty \]$

definition

$\sigma_1' \equiv \llbracket heap = empty(oid0 \mapsto in_{Person} person1)$
 $(oid1 \mapsto in_{Person} person2)$
 $(oid2 \mapsto in_{Person} person3)$
 $(oid3 \mapsto in_{Person} person4)$
 $(*oid4*)$
 $(oid5 \mapsto in_{Person} person6)$
 $(oid6 \mapsto in_{OclAny} person7)$
 $(oid7 \mapsto in_{OclAny} person8)$
 $(oid8 \mapsto in_{Person} person9),$
 $assocs = empty \ \rrbracket$

definition $\sigma_0 \equiv \llbracket heap = empty, assocs = empty \ \rrbracket$

lemma *basic- τ -wff*: $WFF(\sigma_1, \sigma_1')$
 $\langle proof \rangle$

lemma [*simp, code-unfold*]: $dom(heap \ \sigma_1) = \{oid0, oid1, (*, oid2*)oid3, oid4, oid5(*, oid6, oid7*), oid8\}$
 $\langle proof \rangle$

lemma [*simp, code-unfold*]: $dom(heap \ \sigma_1') = \{oid0, oid1, oid2, oid3, (*, oid4*)oid5, oid6, oid7, oid8\}$
 $\langle proof \rangle$

definition $X_{Person1} :: Person \equiv \lambda - . \llbracket person1 \rrbracket$

definition $X_{Person2} :: Person \equiv \lambda - . \llbracket person2 \rrbracket$

definition $X_{Person3} :: Person \equiv \lambda - . \llbracket person3 \rrbracket$

definition $X_{Person4} :: Person \equiv \lambda - . \llbracket person4 \rrbracket$

definition $X_{Person5} :: Person \equiv \lambda - . \llbracket person5 \rrbracket$

definition $X_{Person6} :: Person \equiv \lambda - . \llbracket person6 \rrbracket$

definition $X_{Person7} :: OclAny \equiv \lambda - . \llbracket person7 \rrbracket$

definition $X_{Person8} :: OclAny \equiv \lambda - . \llbracket person8 \rrbracket$

definition $X_{Person9} :: Person \equiv \lambda - . \llbracket person9 \rrbracket$

lemma [*code-unfold*]: $((x :: Person) \doteq y) = StrictRefEq_{Object} \ x \ y \ \langle proof \rangle$

lemma [*code-unfold*]: $((x :: OclAny) \doteq y) = StrictRefEq_{Object} \ x \ y \ \langle proof \rangle$

lemmas [*simp, code-unfold*] =
 $OclAsType_{OclAny} - OclAny$
 $OclAsType_{OclAny} - Person$
 $OclAsType_{Person} - OclAny$
 $OclAsType_{Person} - Person$

$OclIsTypeOf_{OclAny} - OclAny$

$OclIsTypeOf_{OclAny} - Person$

OclIsTypeOf_{Person}-OclAny
OclIsTypeOf_{Person}-Person

OclIsKindOf_{OclAny}-OclAny
OclIsKindOf_{OclAny}-Person
OclIsKindOf_{Person}-OclAny
OclIsKindOf_{Person}-Person

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .salary <> 1000)$
Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .salary \doteq 1300)$
Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .salary@pre \doteq 1000)$
Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .salary@pre <> 1300)$
Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .boss <> X_{Person1})$
Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .boss .salary \doteq 1800)$
Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .boss .boss <> X_{Person1})$
Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .boss .boss \doteq X_{Person2})$
Assert $(\sigma_1, \sigma_1') \models (X_{Person1} .boss@pre .salary \doteq 1800)$
Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .boss@pre .salary@pre \doteq 1200)$
Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .boss@pre .salary@pre <> 1800)$
Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .boss@pre \doteq X_{Person2})$
Assert $(\sigma_1, \sigma_1') \models (X_{Person1} .boss@pre .boss \doteq X_{Person2})$
Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .boss@pre .boss@pre \doteq null)$
Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models not(v(X_{Person1} .boss@pre .boss@pre .boss@pre))$

lemma $(\sigma_1, \sigma_1') \models (X_{Person1} .oclIsMaintained())$
<proof>

lemma $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models ((X_{Person1} .oclAsType(OclAny) .oclAsType(Person)) \doteq X_{Person1})$
<proof>

Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person1} .oclIsTypeOf(Person))$
Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models not(X_{Person1} .oclIsTypeOf(OclAny))$
Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person1} .oclIsKindOf(Person))$
Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person1} .oclIsKindOf(OclAny))$
Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models not(X_{Person1} .oclAsType(OclAny) .oclIsTypeOf(OclAny))$

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person2} .salary \doteq 1800)$
Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person2} .salary@pre \doteq 1200)$
Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person2} .boss \doteq X_{Person2})$
Assert $(\sigma_1, \sigma_1') \models (X_{Person2} .boss .salary@pre \doteq 1200)$
Assert $(\sigma_1, \sigma_1') \models (X_{Person2} .boss .boss@pre \doteq null)$
Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person2} .boss@pre \doteq null)$
Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person2} .boss@pre <> X_{Person2})$
Assert $(\sigma_1, \sigma_1') \models (X_{Person2} .boss@pre <> (X_{Person2} .boss))$
Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models not(v(X_{Person2} .boss@pre .boss))$
Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models not(v(X_{Person2} .boss@pre .salary@pre))$

lemma $(\sigma_1, \sigma_1') \models (X_{Person2} .oclIsMaintained())$
 $\langle proof \rangle$

Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models (X_{Person3} .salary \doteq null)$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models not(v(X_{Person3} .salary@pre))$
Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models (X_{Person3} .boss \doteq null)$
Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models not(v(X_{Person3} .boss .salary))$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models not(v(X_{Person3} .boss@pre))$
lemma $(\sigma_1, \sigma_1') \models (X_{Person3} .oclIsNew())$
 $\langle proof \rangle$

Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models (X_{Person4} .boss@pre \doteq X_{Person5})$
Assert $(\sigma_1, \sigma_1') \models not(v(X_{Person4} .boss@pre .salary))$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models (X_{Person4} .boss@pre .salary@pre \doteq 3500)$
lemma $(\sigma_1, \sigma_1') \models (X_{Person4} .oclIsMaintained())$
 $\langle proof \rangle$

Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models not(v(X_{Person5} .salary))$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models (X_{Person5} .salary@pre \doteq 3500)$
Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models not(v(X_{Person5} .boss))$
lemma $(\sigma_1, \sigma_1') \models (X_{Person5} .oclIsDeleted())$
 $\langle proof \rangle$

Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models not(v(X_{Person6} .boss .salary@pre))$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models (X_{Person6} .boss@pre \doteq X_{Person4})$
Assert $(\sigma_1, \sigma_1') \models (X_{Person6} .boss@pre .salary \doteq 2900)$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models (X_{Person6} .boss@pre .salary@pre \doteq 2600)$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models (X_{Person6} .boss@pre .boss@pre \doteq X_{Person5})$
lemma $(\sigma_1, \sigma_1') \models (X_{Person6} .oclIsMaintained())$
 $\langle proof \rangle$

Assert $\bigwedge_{s_{pre} s_{post}} . (s_{pre}, s_{post}) \models v(X_{Person7} .oclAsType(Person))$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models not(v(X_{Person7} .oclAsType(Person) .boss@pre))$
lemma $\bigwedge_{s_{pre} s_{post}} . (s_{pre}, s_{post}) \models ((X_{Person7} .oclAsType(Person) .oclAsType(OclAny) .oclAsType(Person)) \doteq (X_{Person7} .oclAsType(Person)))$
 $\langle proof \rangle$
lemma $(\sigma_1, \sigma_1') \models (X_{Person7} .oclIsNew())$
 $\langle proof \rangle$

Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models (X_{Person8} <> X_{Person7})$
Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models \text{not}(v(X_{Person8} .oclAsType(Person)))$
Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models (X_{Person8} .oclIsTypeOf(OclAny))$
Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models \text{not}(X_{Person8} .oclIsTypeOf(Person))$
Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models \text{not}(X_{Person8} .oclIsKindOf(Person))$
Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models (X_{Person8} .oclIsKindOf(OclAny))$

lemma $\sigma\text{-modifiedonly}: (\sigma_1, \sigma_1') \models (\text{Set}\{ X_{Person1} .oclAsType(OclAny)$
 $, X_{Person2} .oclAsType(OclAny)$
 $(*, X_{Person3} .oclAsType(OclAny)*)$
 $, X_{Person4} .oclAsType(OclAny)$
 $(*, X_{Person5} .oclAsType(OclAny)*)$
 $, X_{Person6} .oclAsType(OclAny)$
 $(*, X_{Person7} .oclAsType(OclAny)*)$
 $(*, X_{Person8} .oclAsType(OclAny)*)$
 $(*, X_{Person9} .oclAsType(OclAny)*)\} \rightarrow \text{oclIsModifiedOnly}())$
 $\langle \text{proof} \rangle$

lemma $(\sigma_1, \sigma_1') \models ((X_{Person9} @pre (\lambda x. \lfloor \text{OclAsType}_{Person} \mathfrak{A} x \rfloor)) \triangleq X_{Person9})$
 $\langle \text{proof} \rangle$

lemma $(\sigma_1, \sigma_1') \models ((X_{Person9} @post (\lambda x. \lfloor \text{OclAsType}_{Person} \mathfrak{A} x \rfloor)) \triangleq X_{Person9})$
 $\langle \text{proof} \rangle$

lemma $(\sigma_1, \sigma_1') \models (((X_{Person9} .oclAsType(OclAny)) @pre (\lambda x. \lfloor \text{OclAsType}_{OclAny} \mathfrak{A} x \rfloor)) \triangleq$
 $((X_{Person9} .oclAsType(OclAny)) @post (\lambda x. \lfloor \text{OclAsType}_{OclAny} \mathfrak{A} x \rfloor)))$
 $\langle \text{proof} \rangle$

lemma $\text{perm-}\sigma_1' : \sigma_1' = \lfloor \text{heap} = \text{empty}$
 $(oid8 \mapsto in_{Person} person9)$
 $(oid7 \mapsto in_{OclAny} person8)$
 $(oid6 \mapsto in_{OclAny} person7)$
 $(oid5 \mapsto in_{Person} person6)$
 $(*oid4*)$
 $(oid3 \mapsto in_{Person} person4)$
 $(oid2 \mapsto in_{Person} person3)$
 $(oid1 \mapsto in_{Person} person2)$
 $(oid0 \mapsto in_{Person} person1)$
 $, \text{assocs} = \text{assocs } \sigma_1' \rfloor$
 $\langle \text{proof} \rangle$

declare $\text{const-ss} [\text{simp}]$

lemma $\bigwedge_{\sigma_1.}$
 $(\sigma_1, \sigma_1') \models (Person .allInstances() \doteq \text{Set}\{ X_{Person1}, X_{Person2}, X_{Person3}, X_{Person4}(*,$
 $X_{Person5}*), X_{Person6},$
 $X_{Person7} .oclAsType(Person)(*, X_{Person8}*), X_{Person9} \})$

$\langle proof \rangle$

lemma $\bigwedge \sigma_1.$

$(\sigma_1, \sigma_1') \models (OclAny.allInstances() \doteq Set\{ X_{Person1}.oclAsType(OclAny), X_{Person2}.oclAsType(OclAny),$

$X_{Person3}.oclAsType(OclAny), X_{Person4}.oclAsType(OclAny)$
 $(*, X_{Person5}*), X_{Person6}.oclAsType(OclAny),$
 $X_{Person7}, X_{Person8}, X_{Person9}.oclAsType(OclAny) \})$

$\langle proof \rangle$

end

theory

Design-OCL

imports

Design-UML

begin

8.10. OCL Part: Standard State Infrastructure

Ideally, these definitions are automatically generated from the class model.

8.11. Invariant

These recursive predicates can be defined conservatively by greatest fix-point constructions—automatically. See [4, 5] for details. For the purpose of this example, we state them as axioms here.

```
context Person
  inv label : self .boss <> null implies (self .salary \<le>
((self .boss) .salary))
```

definition $Person\text{-}label_{inv} :: Person \Rightarrow Boolean$

where $Person\text{-}label_{inv}(self) \equiv$
 $(self.boss \neq null \text{ implies } (self.salary \leq_{int} ((self.boss).salary)))$

definition $Person\text{-}label_{invATpre} :: Person \Rightarrow Boolean$

where $Person\text{-}label_{invATpre}(self) \equiv$
 $(self.boss@pre \neq null \text{ implies } (self.salary@pre \leq_{int} ((self.boss@pre).salary@pre)))$

definition $Person\text{-}label_{globalinv} :: Boolean$

where $Person\text{-}label_{global\text{-}inv} \equiv (Person .allInstances() \rightarrow forAll(x \mid Person\text{-}label_{inv}(x)) \text{ and } (Person .allInstances@pre() \rightarrow forAll(x \mid Person\text{-}label_{invATpre}(x))))$

lemma $\tau \models \delta(X .boss) \implies \tau \models Person .allInstances() \rightarrow includes(X .boss) \wedge \tau \models Person .allInstances() \rightarrow includes(X)$
 $\langle proof \rangle$

lemma $REC\text{-}pre : \tau \models Person\text{-}label_{global\text{-}inv} \implies \tau \models Person .allInstances() \rightarrow includes(X) \text{ (* } X \text{ represented object in state *)}$
 $\implies \exists REC. \tau \models REC(X) \triangleq (Person\text{-}label_{inv}(X) \text{ and } (X .boss \neq null \text{ implies } REC(X .boss)))$
 $\langle proof \rangle$

This allows to state a predicate:

axiomatization $inv_{Person\text{-}label} :: Person \Rightarrow Boolean$
where $inv_{Person\text{-}label}\text{-}def:$
 $(\tau \models Person .allInstances() \rightarrow includes(self)) \implies$
 $(\tau \models (inv_{Person\text{-}label}(self) \triangleq (self .boss \neq null \text{ implies } (self .salary \leq_{int} ((self .boss) .salary)) \text{ and } inv_{Person\text{-}label}(self .boss))))$

axiomatization $inv_{Person\text{-}labelATpre} :: Person \Rightarrow Boolean$
where $inv_{Person\text{-}labelATpre}\text{-}def:$
 $(\tau \models Person .allInstances@pre() \rightarrow includes(self)) \implies$
 $(\tau \models (inv_{Person\text{-}labelATpre}(self) \triangleq (self .boss@pre \neq null \text{ implies } (self .salary@pre \leq_{int} ((self .boss@pre) .salary@pre)) \text{ and } inv_{Person\text{-}labelATpre}(self .boss@pre))))$

lemma $inv\text{-}1 :$
 $(\tau \models Person .allInstances() \rightarrow includes(self)) \implies$
 $(\tau \models inv_{Person\text{-}label}(self) = ((\tau \models (self .boss \neq null)) \vee (\tau \models (self .boss \neq null) \wedge \tau \models ((self .salary) \leq_{int} (self .boss .salary)) \wedge \tau \models (inv_{Person\text{-}label}(self .boss))))$
 $\langle proof \rangle$

lemma $inv\text{-}2 :$
 $(\tau \models Person .allInstances@pre() \rightarrow includes(self)) \implies$
 $(\tau \models inv_{Person\text{-}labelATpre}(self) = ((\tau \models (self .boss@pre \neq null)) \vee (\tau \models (self .boss@pre \neq null) \wedge (\tau \models (self .boss@pre .salary@pre \leq_{int} self .salary@pre)) \wedge (\tau \models (inv_{Person\text{-}labelATpre}(self .boss@pre))))$
 $\langle proof \rangle$

A very first attempt to characterize the axiomatization by an inductive definition -

this can not be the last word since too weak (should be equality!)

```

coinductive inv :: Person  $\Rightarrow$  ( $\mathfrak{A}$ )st  $\Rightarrow$  bool where
  ( $\tau \models (\delta \text{ self})$ )  $\implies$  (( $\tau \models (\text{self} . \text{boss} \doteq \text{null})$ )  $\vee$ 
    ( $\tau \models (\text{self} . \text{boss} <> \text{null}) \wedge (\tau \models (\text{self} . \text{boss} . \text{salary} \leq_{int} \text{self} . \text{salary})) \wedge$ 
    ( $\text{inv}(\text{self} . \text{boss})\tau$ )))
   $\implies$  (inv self  $\tau$ )

```

8.12. The Contract of a Recursive Query

This part is analogous to the Analysis Model and skipped here.

end

Part III.

Conclusion

9. Conclusion

9.1. Lessons Learned and Contributions

We provided a typed and type-safe shallow embedding of the core of UML [26, 27] and OCL [28]. Shallow embedding means that types of OCL were injectively, i.e., mapped by the embedding one-to-one to types in Isabelle/HOL [25]. We followed the usual methodology to build up the theory uniquely by conservative extensions of all operators in a denotational style and to derive logical and algebraic (execution) rules from them; thus, we can guarantee the logical consistency of the library and instances of the class model construction, i.e., closed-world object-oriented datatype theories, as long as it follows the described methodology.¹ Moreover, all derived execution rules are by construction type-safe (which would be an issue, if we had chosen to use an object universe construction in Zermelo-Fraenkel set theory as an alternative approach to subtyping.). In more detail, our theory gives answers and concrete solutions to a number of open major issues for the UML/OCL standardization:

1. the role of the two exception elements `invalid` and `null`, the former usually assuming strict evaluation while the latter ruled by non-strict evaluation.
2. the functioning of the resulting four-valued logic, together with safe rules (for example `foundation9` – `foundation12` in Section 5.1.5) that allow a reduction to two-valued reasoning as required for many automated provers. The resulting logic still enjoys the rules of a strong Kleene Logic in the spirit of the Amsterdam Manifesto [17].
3. the complicated life resulting from the two necessary equalities: the standard’s “strict weak referential equality” as default (written \doteq throughout this document) and the strong equality (written \triangleq), which follows the logical Leibniz principle that “equals can be replaced by equals.” Which is not necessarily the case if `invalid` or objects of different states are involved.
4. a type-safe representation of objects and a clarification of the old idea of a one-to-one correspondence between object representations and object-id’s, which became a state invariant.
5. a simple concept of state-framing via the novel operator `_->oclIsModifiedOnly()` and its consequences for strong and weak equality.

¹Our two examples of `Employee_AnalysisModel` and `Employee_DesignModel` (see Chapter 7 and Figure II as well as Chapter 8 and Figure II) sketch how this construction can be captured by an automated process.

6. a semantic view on subtyping clarifying the role of static and dynamic type (aka *apparent* and *actual* type in Java terminology), and its consequences for casts, dynamic type-tests, and static types.
7. a semantic view on path expressions, that clarify the role of invalid and null as well as the tricky issues related to de-referentiation in pre- and post state.
8. an optional extension of the OCL semantics by *infinite* sets that provide means to represent “the set of potential objects or values” to state properties over them (this will be an important feature if OCL is intended to become a full-blown code annotation language in the spirit of JML [23] for semi-automated code verification, and has been considered desirable in the Aachen Meeting [13]).

Moreover, we managed to make our theory in large parts executable, which allowed us to include mechanically checked value-statements that capture numerous corner-cases relevant for OCL implementors. Among many minor issues, we thus pin-pointed the behavior of `null` in collections as well as in casts and the desired `isKindOf`-semantics of `allInstances()`.

9.2. Lessons Learned

While our paper and pencil arguments, given in [11], turned out to be essentially correct, there had also been a lesson to be learned: If the logic is not defined as a Kleene-Logic, having a structure similar to a complete partial order (CPO), reasoning becomes complicated: several important algebraic laws break down which makes reasoning in OCL inherent messy and a semantically clean compilation of OCL formulae to a two-valued presentation, that is amenable to animators like KodKod [31] or SMT-solvers like Z3 [18] completely impractical. Concretely, if the expression `not(null)` is defined `invalid` (as is the case in the present standard [28]), then standard involution does not hold, i. e., `not(not(A)) = A` does not hold universally. Similarly, if `null and null` is `invalid`, then not even idempotence `X and X = X` holds. We strongly argue in favor of a lattice-like organization, where `null` represents “more information” than `invalid` and the logical operators are monotone with respect to this semantical “information ordering.”

A similar experience with prior paper and pencil arguments was our investigation of the object-oriented data-models, in particular path-expressions [14]. The final presentation is again essentially correct, but the technical details concerning exception handling lead finally to a continuation-passing style of the (in future generated) definitions for accessors, casts and tests. Apparently, OCL semantics (as many other “real” programming and specification languages) is meanwhile too complex to be treated by informal arguments solely.

Featherweight OCL makes several minor deviations from the standard and showed how the previous constructions can be made correct and consistent, and the DNF-normalization as well as δ -closure laws (necessary for a transition into a two-valued

presentation of OCL specifications ready for interpretation in SMT solvers (see [12] for details)) are valid in Featherweight OCL.

9.3. Conclusion and Future Work

Featherweight OCL concentrates on formalizing the semantics of a core subset of OCL in general and in particular on formalizing the consequences of a four-valued logic (i. e., OCL versions that support, besides the truth values `true` and `false` also the two exception values `invalid` and `null`).

In the following, we outline the necessary steps for turning Featherweight OCL into a fully fledged tool for OCL, e. g., similar to HOL-OCL as well as for supporting test case generation similar to HOL-TestGen [8]. There are essentially five extensions necessary:

- extension of the library to support all OCL data types, e. g., `OrderedSet(T)` or `Sequence(T)`. This formalization of the OCL standard library can be used for checking the consistency of the formal semantics (known as “Annex A”) with the informal and semi-formal requirements in the normative part of the OCL standard.
- development of a compiler that compiles a textual or CASE tool representation (e. g., using XMI or the textual syntax of the USE tool [30]) of class models. Such compiler could also generate the necessary casts when converting standard OCL to Featherweight OCL as well as providing “normalizations” such as converting multiplicities of class attributes to into OCL class invariants.
- a setup for translating Featherweight OCL into a two-valued representation as described in [12]. As, in real-world scenarios, large parts of UML/OCL specifications are defined (e. g., from the default multiplicity 1 of an attributes `x`, we can directly infer that for all valid states `x` is neither `invalid` nor `null`), such a translation enables an efficient test case generation approach.
- a setup in Featherweight OCL of the Nitpick animator [3]. It remains to be shown that the standard, Kodkod [31] based animator in Isabelle can give a similar quality of animation as the OCLexec Tool [22]
- a code-generator setup for Featherweight OCL for Isabelle’s code generator. For example, the Isabelle code generator supports the generation of F#, which would allow to use OCL specifications for testing arbitrary .net-based applications.

The first two extensions are sufficient to provide a formal proof environment for OCL 2.5 similar to HOL-OCL while the remaining extensions are geared towards increasing the degree of proof automation and usability as well as providing a tool-supported test methodology for UML/OCL.

Our work shows that developing a machine-checked formal semantics of recent OCL standards still reveals significant inconsistencies—even though this type of research is not new. In fact, we started our work already with the 1.x series of OCL. The reasons for this ongoing consistency problems of OCL standard are manifold. For example, the

consequences of adding an additional exception value to OCL 2.2 are widespread across the whole language and many of them are also quite subtle. Here, a machine-checked formal semantics is of great value, as one is forced to formalize all details and subtleties. Moreover, the standardization process of the OMG, in which standards (e. g., the UML infrastructure and the OCL standard) that need to be aligned closely are developed quite independently, are prone to ad-hoc changes that attempt to align these standards. And, even worse, updating a standard document by voting on the acceptance (or rejection) of isolated text changes does not help either. Here, a tool for the editor of the standard that helps to check the consistency of the whole standard after each and every modifications can be of great value as well.

Contents

I. Annex A	1
1. Introduction	3
2. Background	7
2.1. A Running Example for UML/OCL	7
2.2. Formal Foundation	9
2.2.1. Isabelle	9
2.2.2. Higher-order Logic (HOL)	10
2.3. How this Annex A was Generated from Isabelle/HOL Theories	12
3. The Essence of UML-OCL Semantics	15
3.1. The Theory Organization	15
3.1.1. Denotational Semantics of Types	15
3.1.2. Denotational Semantics of Constants and Operations	16
3.1.3. Logical Layer	18
3.1.4. Algebraic Layer	20
3.2. Object-oriented Datatype Theories	22
3.2.1. A Denotational Space for Class-Models: Object Universes	23
3.2.2. Denotational Semantics of Accessors on Objects and Associations	25
3.2.3. Logic Properties of Class-Models	28
3.2.4. Algebraic Properties of the Class-Models	29
3.2.5. Other Operations on States	30
3.3. Data Invariants	30
3.4. Operation Contracts	31
II. Formal Semantics of UML-OCL 2.5	33
4. Formalization I: OCL Types and Core Definitions	35
4.1. Preliminaries	35
4.1.1. Notations for the Option Type	35
4.1.2. Common Infrastructure for all OCL Types	36
4.1.3. Accommodation of Basic Types to the Abstract Interface	36
4.1.4. The Common Infrastructure of Object Types (Class Types) and States.	37

4.1.5.	Common Infrastructure for all OCL Types (II): Valuations as OCL Types	38
4.1.6.	The fundamental constants 'invalid' and 'null' in all OCL Types	39
4.2.	Basic OCL Value Types	39
4.3.	Some OCL Collection Types	40
4.3.1.	The Construction of the Pair Type (Tuples)	40
4.3.2.	The Construction of the Set Type	41
4.3.3.	The Construction of the Sequence Type	41
4.3.4.	Discussion: The Representation of UML/OCL Types in Featherweight OCL	42
5.	Formalization II: OCL Terms and Library Operations	45
5.1.	The Operations of the Boolean Type and the OCL Logic	45
5.1.1.	Basic Constants	45
5.1.2.	Validity and Definedness	46
5.1.3.	The Equalities of OCL	47
5.1.4.	Logical Connectives and their Universal Properties	50
5.1.5.	A Standard Logical Calculus for OCL	55
5.1.6.	OCL's if then else endif	61
5.1.7.	Fundamental Predicates on Basic Types: Strict (Referential) Equality	62
5.1.8.	Laws to Establish Definedness (δ -closure)	62
5.1.9.	A Side-calculus for Constant Terms	63
5.2.	Property Profiles for OCL Operators via Isabelle Locales	66
5.2.1.	mono	66
5.2.2.	single	67
5.2.3.	bin	67
5.2.4.	Fundamental Predicates on Basic Types: Strict (Referential) Equality	70
5.2.5.	Test Statements on Boolean Operations.	71
5.3.	Basic Type Void	72
5.3.1.	Fundamental Properties on Basic Types: Strict Equality	72
5.3.2.	Test Statements	72
5.4.	Basic Type Integer: Operations	73
5.4.1.	Basic Integer Constants	73
5.4.2.	Validity and Definedness Properties	73
5.4.3.	Arithmetical Operations	74
5.4.4.	Fundamental Predicates on Integers: Strict Equality	76
5.4.5.	Test Statements on Basic Integer	76
5.5.	Basic Type Real: Operations	77
5.5.1.	Basic Real Constants	77
5.5.2.	Validity and Definedness Properties	78
5.5.3.	Arithmetical Operations	79
5.5.4.	Fundamental Predicates on Reals: Strict Equality	81

5.5.5.	Test Statements on Basic Real	81
5.6.	Basic Type String: Operations	82
5.6.1.	Basic String Constants	82
5.6.2.	Validity and Definedness Properties	83
5.6.3.	String Operations	83
5.6.4.	Fundamental Properties on Strings: Strict Equality	84
5.6.5.	Test Statements on Basic String	84
5.7.	Collection Type Pairs: Operations	85
5.7.1.	Semantic Properties of the Type Constructor	85
5.7.2.	Strict Equality	85
5.7.3.	Standard Operations	86
5.7.4.	Logical Properties	86
5.7.5.	Execution Properties	86
5.7.6.	Test Statements	87
5.8.	Collection Type Set: Operations	87
5.8.1.	As a Motivation for the (infinite) Type Construction: Type- Extensions as Sets	87
5.8.2.	Validity and Definedness Properties	88
5.8.3.	Constants on Sets	89
5.8.4.	Operations	89
5.8.5.	Strict Equality	100
5.8.6.	Test Statements	110
5.9.	Collection Type Sequence: Operations	111
5.9.1.	Constants: mtSequence	111
5.9.2.	Strict Equality	111
5.9.3.	Standard Operations	112
5.9.4.	Test Statements	113
5.10.	Miscellaneous Stuff	113
5.10.1.	Properties on Collection Types: Strict Equality	113
5.10.2.	MOVE TEXT : Collection Types	113
5.10.3.	MOVE TEXT: Test Statements	114
6.	Formalization III: UML/OCL constructs: State Operations and Objects	117
6.1.	Introduction: States over Typed Object Universes	117
6.1.1.	Fundamental Properties on Objects: Core Referential Equality . .	117
6.1.2.	Logic and Algebraic Layer on Object	118
6.2.	Operations on Object	120
6.2.1.	Initial States (for testing and code generation)	120
6.2.2.	OclAllInstances	120
6.2.3.	OclIsNew, OclIsDeleted, OclIsMaintained, OclIsAbsent	126
6.2.4.	OclIsModifiedOnly	127
6.2.5.	OclSelf	128
6.2.6.	Framing Theorem	128
6.2.7.	Miscellaneous	129

7. Example I : The Employee Analysis Model (UML)	137
7.1. Introduction	137
7.1.1. Outlining the Example	137
7.2. Example Data-Universe and its Infrastructure	138
7.3. Instantiation of the Generic Strict Equality	139
7.4. OclAsType	140
7.4.1. Definition	140
7.4.2. Context Passing	141
7.4.3. Execution with Invalid or Null as Argument	141
7.5. OclIsTypeOf	142
7.5.1. Definition	142
7.5.2. Context Passing	143
7.5.3. Execution with Invalid or Null as Argument	143
7.5.4. Up Down Casting	144
7.6. OclIsKindOf	145
7.6.1. Definition	145
7.6.2. Context Passing	145
7.6.3. Execution with Invalid or Null as Argument	146
7.6.4. Up Down Casting	147
7.7. OclAllInstances	147
7.7.1. OclIsTypeOf	148
7.7.2. OclIsKindOf	148
7.8. The Accessors (any, boss, salary)	149
7.8.1. Definition (of the association Employee-Boss)	149
7.8.2. Context Passing	152
7.8.3. Execution with Invalid or Null as Argument	153
7.9. A Little Infra-structure on Example States	154
7.10. OCL Part: Standard State Infrastructure	159
7.11. Invariant	159
7.12. The Contract of a Recursive Query	160
7.13. The Contract of a User-defined Method	162
8. Example II: The Employee Design Model (UML)	165
8.1. Introduction	165
8.1.1. Outlining the Example	165
8.2. Example Data-Universe and its Infrastructure	165
8.3. Instantiation of the Generic Strict Equality	167
8.4. OclAsType	168
8.4.1. Definition	168
8.4.2. Context Passing	168
8.4.3. Execution with Invalid or Null as Argument	169
8.5. OclIsTypeOf	170
8.5.1. Definition	170
8.5.2. Context Passing	170

8.5.3.	Execution with Invalid or Null as Argument	171
8.5.4.	Up Down Casting	172
8.6.	OclIsKindOf	173
8.6.1.	Definition	173
8.6.2.	Context Passing	173
8.6.3.	Execution with Invalid or Null as Argument	174
8.6.4.	Up Down Casting	175
8.7.	OclAllInstances	175
8.7.1.	OclIsTypeOf	175
8.7.2.	OclIsKindOf	176
8.8.	The Accessors (any, boss, salary)	177
8.8.1.	Definition	177
8.8.2.	Context Passing	179
8.8.3.	Execution with Invalid or Null as Argument	180
8.9.	A Little Infra-structure on Example States	180
8.10.	OCL Part: Standard State Infrastructure	186
8.11.	Invariant	186
8.12.	The Contract of a Recursive Query	188

III. Conclusion 189

9. Conclusion 191

9.1.	Lessons Learned and Contributions	191
9.2.	Lessons Learned	192
9.3.	Conclusion and Future Work	193

Bibliography

- [1] P. B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Kluwer Academic Publishers, Dordrecht, 2nd edition, 2002. ISBN 1-402-00763-9.
- [2] C. Barrett and C. Tinelli. Cvc3. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 2007. ISBN 978-3-540-73367-6. doi: 10.1007/978-3-540-73368-3_34.
- [3] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. C. Paulson, editors, *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer-Verlag, 2010. ISBN 978-3-642-14051-8. doi: 10.1007/978-3-642-14052-5_11.
- [4] A. D. Brucker. *An Interactive Proof Environment for Object-oriented Specifications*. PhD thesis, ETH Zurich, Mar. 2007. URL <http://www.brucker.ch/bibliography/abstract/brucker-interactive-2007>. ETH Dissertation No. 17097.
- [5] A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-ocl-book-2006>.
- [6] A. D. Brucker and B. Wolff. An extensible encoding of object-oriented data models in hol. *Journal of Automated Reasoning*, 41:219–249, 2008. ISSN 0168-7433. doi: 10.1007/s10817-008-9108-3. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-extensible-2008-b>.
- [7] A. D. Brucker and B. Wolff. HOL-OCL – A Formal Proof Environment for UML/OCL. In J. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering (FASE08)*, number 4961 in *Lecture Notes in Computer Science*, pages 97–100. Springer-Verlag, Heidelberg, 2008. doi: 10.1007/978-3-540-78743-3_8. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-ocl-2008>.
- [8] A. D. Brucker and B. Wolff. HOL-TestGen: An interactive test-case generation framework. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering (FASE09)*, number 5503 in *Lecture Notes in Computer Science*, pages 417–420. Springer-Verlag, Heidelberg, 2009. doi: 10.1007/978-3-642-00593-0_28. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-testgen-2009>.

- [9] A. D. Brucker, J. Doser, and B. Wolff. Semantic issues of OCL: Past, present, and future. *Electronic Communications of the EASST*, 5, 2006. ISSN 1863-2122. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-semantic-2006-b>.
- [10] A. D. Brucker, J. Doser, and B. Wolff. A model transformation semantics and analysis methodology for SecureUML. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS 2006: Model Driven Engineering Languages and Systems*, number 4199 in Lecture Notes in Computer Science, pages 306–320. Springer-Verlag, Heidelberg, 2006. doi: 10.1007/11880240_22. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-transformation-2006>. An extended version of this paper is available as ETH Technical Report, no. 524.
- [11] A. D. Brucker, M. P. Krieger, and B. Wolff. Extending OCL with null-references. In S. Gosh, editor, *Models in Software Engineering*, number 6002 in Lecture Notes in Computer Science, pages 261–275. Springer-Verlag, Heidelberg, 2009. doi: 10.1007/978-3-642-12261-3_25. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-ocl-null-2009>. Selected best papers from all satellite events of the MoDELS 2009 conference.
- [12] A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff. A specification-based test case generation method for UML/OCL. In J. Dingel and A. Solberg, editors, *MoDELS Workshops*, number 6627 in Lecture Notes in Computer Science, pages 334–348. Springer-Verlag, Heidelberg, 2010. ISBN 978-3-642-21209-3. doi: 10.1007/978-3-642-21210-9_33. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-ocl-testing-2010>. Selected best papers from all satellite events of the MoDELS 2010 conference. Workshop on OCL and Textual Modelling.
- [13] A. D. Brucker, D. Chiorean, T. Clark, B. Demuth, M. Gogolla, D. Plotnikov, B. Rumpe, E. D. Willink, and B. Wolff. Report on the Aachen OCL meeting. In J. Cabot, M. Gogolla, I. Rath, and E. Willink, editors, *Proceedings of the MODELS 2013 OCL Workshop (OCL 2013)*, volume 1092 of *CEUR Workshop Proceedings*, pages 103–111. CEUR-WS.org, 2013. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-summary-aachen-2013>.
- [14] A. D. Brucker, D. Longuet, F. Tuong, and B. Wolff. On the semantics of object-oriented data structures and path expressions. In *OCL@MoDELS*, pages 23–32, 2013.
- [15] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.
- [16] T. Clark and J. Warmer, editors. *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *Lecture Notes in Computer Science*, Heidelberg, 2002. Springer-Verlag. ISBN 3-540-43169-1.

- [17] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The amsterdam manifesto on OCL. In Clark and Warmer [16], pages 115–149. ISBN 3-540-43169-1.
- [18] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-78799-0. doi: 10.1007/978-3-540-78800-3_24.
- [19] M. Gogolla and M. Richters. Expressing UML class diagrams properties with OCL. In Clark and Warmer [16], pages 85–114. ISBN 3-540-43169-1.
- [20] A. Hamie, F. Civello, J. Howse, S. Kent, and R. Mitchell. Reflections on the Object Constraint Language. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language. «UML»'98: Beyond the Notation*, volume 1618 of *Lecture Notes in Computer Science*, pages 162–172, Heidelberg, 1998. Springer-Verlag. ISBN 3-540-66252-9. doi: 10.1007/b72309.
- [21] P. Kosiuczenko. Specification of invariability in OCL. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *Lecture Notes in Computer Science*, pages 676–691, Heidelberg, 2006. Springer-Verlag. ISBN 978-3-540-45772-5. doi: 10.1007/11880240_47.
- [22] M. P. Krieger, A. Knapp, and B. Wolff. Generative programming and component engineering. In E. Visser and J. Järvi, editors, *International Conference on Generative Programming and Component Engineering (GPCE 2010)*, pages 53–62. ACM, Oct. 2010. ISBN 978-1-4503-0154-1.
- [23] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML reference manual (revision 1.2), Feb. 2007. Available from <http://www.jmlspecs.org>.
- [24] L. Mandel and M. V. Cengarle. On the expressive power of OCL. In J. M. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems (FM)*, volume 1708 of *Lecture Notes in Computer Science*, pages 854–874, Heidelberg, 1999. Springer-Verlag. ISBN 3-540-66587-0.
- [25] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2002. doi: 10.1007/3-540-45949-9.
- [26] Object Management Group. UML 2.4.1: Infrastructure specification, Aug. 2011. Available as OMG document formal/2011-08-05.
- [27] Object Management Group. UML 2.4.1: Superstructure specification, Aug. 2011. Available as OMG document formal/2011-08-06.

- [28] Object Management Group. UML 2.3.1 OCL specification, Feb. 2012. Available as OMG document formal/2012-01-01.
- [29] A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *CADE*, volume 1632 of *Lecture Notes in Computer Science*, pages 292–296. Springer-Verlag, 1999. ISBN 3-540-66222-7. doi: 10.1007/3-540-48660-7_26.
- [30] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.
- [31] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-71208-4. doi: 10.1007/978-3-540-71209-1_49.
- [32] M. Wenzel and B. Wolff. Building formal method tools in the Isabelle/Isar framework. In K. Schneider and J. Brandt, editors, *TPHOLs 2007*, number 4732 in *Lecture Notes in Computer Science*, pages 352–367. Springer-Verlag, Heidelberg, 2007. doi: 10.1007/978-3-540-74591-4_26.
- [33] M. M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, TU München, München, Feb. 2002. URL <http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html>.