Featherweight OCL
A Proposal for a Machine-Checked Formal Semantics for OCL 2.5

Achim D. Brucker∗ Frédéric Tuong‡ Burkhart Wolff†

May 28, 2015

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

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

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

The Unified Modeling Language (UML) is one of the few modeling languages that is widely used in industry. While UML is mostly known as diagrammatic modeling language (e.g., visualizing class models), it is complemented by a textual language, called Object Constraint Language (OCL). OCL is a textual annotation language, based on a three-valued logic, that turns UML into a formal language. Unfortunately the semantics of this specification language, captured in the “Annex A” of the OCL standard, leads to different interpretations of corner cases. Many of these corner cases had been subject to formal analysis since more than ten years.

The situation complicated when with version 2.3 the OCL was aligned with the latest version of UML: this led to the extension of the three-valued logic by a second exception element, called null. While the first exception element invalid has a strict semantics, null has a non strict semantic interpretation. These semantic difficulties lead to remarkable confusion for implementors of OCL compilers and interpreters.

In this paper, we provide a formalization of the core of OCL in HOL. It provides denotational definitions, a logical calculus and operational rules that allow for the execution of OCL expressions by a mixture of term rewriting and code compilation. Our formalization reveals several inconsistencies and contradictions in the current version of the OCL standard. They reflect a challenge to define and implement OCL tools in a uniform manner. Overall, this document is intended to provide the basis for a machine-checked text “Annex A” of the OCL standard targeting at tool implementors.
3.4. Logical Connectives and their Universal Properties .... 47
3.5. A Standard Logical Calculus for OCL .................. 53
  3.5.1. Global vs. Local Judgements .................. 54
  3.5.2. Local Validity and Meta-logic ................. 54
  3.5.3. Local Judgements and Strong Equality .......... 58
  3.5.4. Laws to Establish Definedness (δ-closure) .... 60
3.6. Miscellaneous ...................................... 60
  3.6.1. OCL’s if then else endif .................... 60
  3.6.2. A Side-calculus for (Boolean) Constant Terms ... 61

4. Formalization II: Library Definitions .............. 65
  4.1. Basic Types: Void and Integer ................. 65
    4.1.1. The Construction of the Void Type ........ 65
    4.1.2. The Construction of the Integer Type .... 65
    4.1.3. Validity and Definedness Properties ... 66
    4.1.4. Arithmetical Operations on Integer .... 67
  4.2. Fundamental Predicates on Basic Types: Strict Equality .... 69
    4.2.1. Definition ................................ 69
    4.2.2. Logic and Algebraic Layer on Basic Types ... 69
    4.2.3. Test Statements on Basic Types ........ 72
  4.3. Complex Types: The Set-Collection Type (I) Core ... 73
    4.3.1. The Construction of the Set Type ....... 73
    4.3.2. Validity and Definedness Properties ... 74
    4.3.3. Constants on Sets ........................ 75
  4.4. Complex Types: The Set-Collection Type (II) Library ... 76
    4.4.1. Computational Operations on Set .......... 76
    4.4.2. Validity and Definedness Properties ... 79
    4.4.3. Execution with Invalid or Null or Infinite Set as Argument ... 85
    4.4.4. Context Passing ........................... 88
    4.4.5. Const .................................... 90
  4.5. Fundamental Predicates on Set: Strict Equality .... 91
    4.5.1. Definition ................................ 91
    4.5.2. Logic and Algebraic Layer on Set .......... 91
  4.6. Execution on Set’s Operators (with mtSet and recursive case as arguments) 92
    4.6.1. OclIncluding ............................... 92
    4.6.2. OclExcluding ............................... 94
    4.6.3. OclIncludes ................................ 99
    4.6.4. OclExcludes ................................ 102
    4.6.5. OclSize .................................... 102
    4.6.6. OclIsEmpty ................................ 104
    4.6.7. OclNotEmpty ................................ 105
    4.6.8. OclANY ..................................... 106
    4.6.9. OclForall ................................... 106
    4.6.10. OclExists .................................. 109
4.6.11. OclIterate ................................. 109
4.6.12. OclSelect ................................. 111
4.6.13. OclReject ................................. 116
4.7. Execution on Set’s Operators (higher composition) .................. 116
4.7.1. OclIncludes ............................... 116
4.7.2. OclSize ................................... 118
4.7.3. OclForall ................................. 120
4.7.4. Strict Equality ............................. 124
4.8. Test Statements ............................... 126

5. Formalization III: State Operations and Objects .......................... 129
5.1. Introduction: States over Typed Object Universes ................... 129
5.1.1. Recall: The Generic Structure of States .......................... 129
5.2. Fundamental Predicates on Object: Strict Equality .................. 130
5.2.1. Logic and Algebraic Layer on Object ............................. 130
5.3. Operations on Object ................................ 132
5.3.1. Initial States (for testing and code generation) ................. 132
5.3.2. OclAllInstances ............................. 132
5.3.3. OclIsNew, OclIsDeleted, OclIsMaintained, OclIsAbsent ........ 143
5.3.4. OclIsModifiedOnly ............................ 144
5.3.5. OclSelf .................................... 145
5.3.6. Framing Theorem ................................ 145
5.3.7. Miscellaneous ................................ 148

III. Examples ........................................ 151

6. The Employee Analysis Model ........................................... 153
6.1. The Employee Analysis Model (UML) ................................ 153
6.1.1. Introduction ................................... 153
6.1.2. Example Data-Universe and its Infrastructure .................. 154
6.1.3. Instantiation of the Generic Strict Equality .................... 155
6.1.4. OclAsType .................................... 156
6.1.5. OclIsTypeOf ................................... 158
6.1.6. OclIsKindOf ................................... 161
6.1.7. OclAllInstances ................................ 164
6.1.8. The Accessors (any, boss, salary) ............................... 167
6.1.9. A Little Infrastructure on Example States ....................... 172
6.2. The Employee Analysis Model (OCL) ................................ 179
6.2.1. Standard State Infrastructure ................................ 180
6.2.2. Invariant ....................................... 180
6.2.3. The Contract of a Recursive Query ............................... 180
6.2.4. The Contract of a Method ................................. 181
Part I.

Introduction
1. Motivation

The Unified Modeling Language (UML) \cite{31, 32} is one of the few modeling languages that is widely used in industry. UML is defined, in an open process, by the Object Management Group (OMG), i.e., an industry consortium. While UML is mostly known as diagrammatic modeling language (e.g., visualizing class models), it also comprises a textual language, called Object Constraint Language (OCL) \cite{33}. OCL is a textual annotation language, originally conceived as a three-valued logic, that turns substantial parts of UML into a formal language. Unfortunately the semantics of this specification language, captured in the “Annex A” (originally, based on the work of Richters \cite{35}) of the OCL standard leads to different interpretations of corner cases. Many of these corner cases had been subject to formal analysis since more than nearly fifteen years (see, e.g., \cite{5, 11, 19, 22, 26}).

At its origins \cite{28, 35}, OCL was conceived as a strict semantics for undefinedness (e.g., denoted by the element invalid\footnote{In earlier versions of the OCL standard, this element was called OclUndefined.}), with the exception of the logical connectives of type Boolean that constitute a three-valued propositional logic. At its core, OCL comprises four layers:

1. Operators (e.g., \_ and \_, \_ + \_) on built-in data structures such as Boolean, Integer, or typed sets (Set(_)).
2. Operators on the user-defined data model (e.g., defined as part of a UML class model) such as accessors, type casts and tests.
3. Arbitrary, user-defined, side-effect-free methods,
4. Specification for invariants on states and contracts for operations to be specified via pre- and post-conditions.

Motivated by the need for aligning OCL closer with UML, recent versions of the OCL standard \cite{30, 33} added a second exception element. While the first exception element invalid has a strict semantics, null has a non strict semantic interpretation. Unfortunately, this extension results in several inconsistencies and contradictions. These problems are reflected in difficulties to define interpreters, code-generators, specification animators or theorem provers for OCL in a uniform manner and resulting incompatibilities of various tools.

For the OCL community, the semantics of invalid and null as well as many related issues resulted in the challenge to define a consistent version of the OCL standard that is well aligned with the recent developments of the UML. A syntactical and semantical
consistent standard requires a major revision of both the informal and formal parts of
the standard. 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 [15]. During this meeting, a Request for Proposals (RFP)
for OCL 2.5 was finalized and meanwhile proposed. In particular, this RFP requires
that the future OCL 2.5 standard document shall be generated from a machine-checked
source. This will ensure

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

In this document, we present a formalization using Isabelle/HOL [27] of a core
language of OCL. The semantic theory, based on a “shallow embedding”, is called
Featherweight OCL, since it focuses on a formal treatment of the key-elements of the
language (rather than a full treatment of all operators and thus, a “complete” implemen-
tation). In contrast to full OCL, it comprises just the logic captured in Boolean, the basic
data type Integer, the collection type Set, as well as the generic construction principle
of class models, which is instantiated and demonstrated for two examples (an automated
support for this type-safe construction is again out of the scope of Featherweight OCL).
This formal semantics definition is intended to be a proposal for the standardization
process of OCL 2.5, which should ultimately replace parts of the mandatory part of the
standard document [33] as well as replace completely its informative “Annex A.”
2. Background

2.1. A Guided Tour Through UML/OCL

The Unified Modeling Language (UML) \cite{31,32} 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., \texttt{Hearer}, \texttt{Speaker}, or \texttt{Chair}) using an inheritance relation (also called generalization). In particular, inheritance establishes a subtyping relationship, i.e., every \texttt{Speaker} (subclass) is also a \texttt{Hearer} (superclass).

A class does not only describe a set of instances (called \textit{objects}), i.e., record-like data consisting of \textit{attributes} such as \texttt{name} of class \texttt{Session}, but also \textit{operations} defined over them. For example, for the class \texttt{Session}, representing a conference session, we model an operation \texttt{findRole(p:Person):Role} that should return the role of a \texttt{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.

![Class Diagram](image)

Figure 2.1.: A simple UML class model representing a conference system for organizing conference sessions: persons can participate, in different roles, in a session.
Relations between classes (called associations in UML) can be represented in a class diagram by connecting lines, e.g., Participant and Session or Person and Role. Associations may be labeled by a particular constraint called multiplicity, e.g., 0..* or 0..1, which means that in a relation between participants and sessions, each Participant object is associated to at most one Session object, while each Session object may be associated to arbitrarily many Participant objects. Furthermore, associations may be labeled by projection functions like person and role; these implicit function definitions allow for OCL-expressions like self.person, where self is a variable of the class Role. The expression self.person denotes persons being related to the specific object self of type role. A particular feature of the UML are association classes (Participant in our example) which represent a concrete tuple of the relation within a system state as an object; i.e., associations classes allow also for defining attributes and operations for such tuples. In a class diagram, association classes are represented by a dotted line connecting the class with the association. Associations classes can take part in other associations. Moreover, UML supports also n-ary associations (not shown in our example).

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

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

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

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

where p.role.oclIsTypeOf(Chair) evaluates to true, if p.role is of dynamic type Chair. Besides the usual static types (i.e., the types inferred by a static type inference), objects in UML and other object-oriented languages have a second dynamic type concept. This is a consequence of a family of casting functions (written 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 \texttt{@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 \cite{12} is to translate the diagrammatic UML features into a combination of more elementary features of UML and OCL expressions \cite{21}. 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.

\section{2.2. Formal Foundation}

\subsection{2.2.1. Isabelle}

Isabelle \cite{27} is a \textit{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 \texttt{⇒} allowing to form constructs like \( A_1 \implies \cdots \implies A_n \implies A_{n+1} \), which are viewed as a \textit{rule} of the form “from assumptions \( A_1 \) to \( A_n \), infer conclusion \( A_{n+1} \)” and which is written in Isabelle as

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

or, in mathematical notation,

\[
\frac{A_1 \quad \cdots \quad A_n}{A_{n+1}} \quad (2.1)
\]

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

\[
\forall x_1, \ldots, x_m. [A_1; \ldots; A_n] \implies A_{n+1} \quad (2.2)
\]

Isabelle supports forward- and backward reasoning on rules. For backward-reasoning, a \textit{proof-state} can be initialized and further transformed into others. For example, a proof of \( \phi \), using the Isar \cite{38} language, will look as follows in Isabelle:

\[
\text{lemma label: } \phi
\text{ apply(case_tac)}
\text{ apply(simp_all)}
\text{ done}
\quad (2.3)
\]

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

\[
\begin{align*}
\text{label} & : \phi \\
1. & \phi_1 \\
\vdots & \\
n. & \phi_n
\end{align*}
\]

(2.4)

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

### 2.2.2. Higher-order Logic (HOL)

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

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

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

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

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

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

Isabelle’s syntax engine is instructed to accept the notation \{x | P\} for Collect \(\lambda x. P\) and the notation \(s \in S\) for member \(s S\). As can be inferred from the example, constant definitions are axioms that introduce a fresh constant symbol by some closed, non-recursive expressions; this type of axiom is logically safe since it works like an abbreviation. The syntactic side conditions of this axiom are mechanically checked, of course. It is straightforward to express the usual operations on sets like \(\cup, \cap\) as set comprehensions, too, while the rules of typed set theory were derived by proofs from these definitions.

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

\[
\begin{align*}
datatype & \quad \text{option } = \text{None} | \text{Some } \alpha \\
datatype & \quad \alpha \text{ list } = \text{Nil} | \text{Cons } a \hspace{1mm} l
\end{align*}
\]

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

\[
\text{case } x \text{ of } \text{None } \Rightarrow F \mid \text{Some } a \Rightarrow G a
\]

respectively

\[
\text{case } x \text{ of } [\_] \Rightarrow F \mid \text{Cons } a \Rightarrow G a \hspace{1mm} r.
\]

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

\[
\begin{align*}
\text{(case } [\_] \text{ of } [\_] & \Rightarrow F \mid (a\#r) \Rightarrow G a \hspace{1mm} r) = F \\
\text{(case } b\#t \text{ of } [\_] & \Rightarrow F \mid (a\#r) \Rightarrow G a \hspace{1mm} r) = G b \hspace{1mm} t \\
[a = [\_] \Rightarrow P; \exists \ x \hspace{1mm} t. \ a = x\#t \Rightarrow P] & \Rightarrow P \quad \text{— distinctness} \\
[P[:]; \forall \ at. \ P\hspace{1mm} t \Rightarrow P(a\#t)] & \Rightarrow P x \quad \text{— exhaust} \\
\end{align*}
\]

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

\[
\begin{align*}
\text{fun} & \quad \text{ins} \quad : [\alpha :: \text{linorder}, \alpha \text{ list}] \Rightarrow \alpha \text{ list} \\
\text{where} & \quad \text{ins } x [\_] \hspace{1mm} = [x] \\
& \quad \text{ins } x (y\#ys) \hspace{1mm} = \text{if } x < y \text{ then } x\#y\#ys \text{ else } y\#(\text{ins } x \hspace{1mm} y \hspace{1mm} ys)
\end{align*}
\]

\(\text{To increase readability, we use a slightly simplified presentation.}\)
fun sort :: (α :: linorder) list ⇒ α list
where sort [ ] = [ ]
    sort (x # xs) = ins x (sort xs)

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. Featherweight OCL: Design Goals

Featherweight OCL is a formalization of the core of OCL aiming at formally investigating the relationship between the various concepts. At present, it does not attempt to define the complete OCL library. Instead, it concentrates on the core concepts of OCL as well as the types Boolean, Integer, and typed sets (Set(T)). Following the tradition of HOL-OCL [6, 8], Featherweight OCL is based on the following principles:

1. It is an embedding into a powerful semantic meta-language and environment, namely Isabelle/HOL [27].

2. It is a shallow embedding in HOL; types in OCL were injectively mapped to types in Featherweight OCL. Ill-typed OCL specifications cannot be represented in Featherweight OCL and a type in Featherweight OCL contains exactly the values that are possible in OCL. Thus, sets may contain null (Set{null} is a defined set) but not invalid (Set{invalid} is just invalid).

3. Any Featherweight OCL type contains at least invalid and null (the type Void contains only these instances). The logic is consequently four-valued, and there is a null-element in the type Set(A).

4. It is a strongly typed language in the Hindley-Milner tradition. We assume that a pre-process eliminates all implicit conversions due to subtyping by introducing explicit casts (e.g., oclAsType()). The details of such a pre-processing are described in [4]. Casts are semantic functions, typically injections, that may convert data between the different Featherweight OCL types.
5. All objects are represented in an object universe in the HOL-OCL tradition. The universe construction also gives semantics to type casts, dynamic type tests, as well as functions such as \texttt{oclAllInstances()}, or \texttt{oclIsNew()}.

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

7. For demonstration purposes, the set type in Featherweight OCL may be infinite, allowing infinite quantification and a constant that contains the set of all Integers. Arithmetic laws like commutativity may therefore be expressed in OCL itself. The iterator is only defined on finite sets.

8. It supports equational reasoning and congruence reasoning, but this requires a differentiation of the different equalities like strict equality, strong equality, meta-equality (HOL). Strict equality and strong equality require a subcalculus, “cp” (a detailed discussion of the different equalities as well as the subcalculus “cp”—for three-valued OCL 2.0—is given in [10]), which is nasty but can be hidden from the user inside tools.

2.4. The Theory Organization

The semantic theory is organized in a quite conventional manner in three layers. The first layer, called the \textit{denotational semantics} comprises a set of definitions of the operators of the language. Presented as \textit{definitional axioms} inside Isabelle/HOL, this part assures the logically consistency of the overall construction. The second layer, called \textit{logical layer}, is derived from the former and centered around the notion of validity of an OCL formula \( P \) for a state-transition from pre-state \( \sigma \) to post-state \( \sigma' \), validity statements were written \( (\sigma, \sigma') \models P \). The third layer, called \textit{algebraic layer}, also derived from the former layers, tries to establish algebraic laws of the form \( P = P' \); such laws are amenable to equational reasoning and also help for automated reasoning and code-generation.

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

2.4.1. Denotational Semantics

OCL is composed of

1. operators on built-in data structures such as \texttt{Boolean}, \texttt{Integer}, or \texttt{Set(A)},

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

3. user-defined, side-effect-free methods.
Conceptually, an OCL expression in general and Boolean expressions in particular (i.e., formulae) depends on the pair \((\sigma, \sigma')\) of pre- and post-state. The precise form of states is irrelevant for this paper (compare [13]) and will be left abstract in this presentation. We construct in Isabelle a type-class \texttt{null} that contains two distinguishable elements \texttt{bot} and \texttt{null}. Any type of the form \((\alpha \perp \perp)\) is an instance of this type-class with \texttt{bot} \equiv \perp and \texttt{null} \equiv ⌊\perp⌋. Now, any OCL type can be represented by an HOL type of the form:

\[
V(\alpha) := \text{state} \times \text{state} \to \alpha :: \text{null}.
\]

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

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

\[
I[\text{true} :: \text{Boolean}]\tau = \mid\mid\text{true}\mid\mid \quad I[\text{false}]\tau = \mid\mid\text{false}\mid\mid
\]

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

\[
I[X\.oclIsInvalid()]\tau = (\text{if } I[X] \tau = \text{bot} \text{ then } I[\text{true}]\tau \text{ else } I[\text{false}]\tau)
\]

where \(I[E]\) is the semantic interpretation function commonly used in mathematical textbooks and \(\tau\) stands for pairs of pre- and post state \((\sigma, \sigma').\) For reasons of conciseness, we will write \(\delta X\) for \(\text{not } X\.oclIsUndefined()\) and \(\nu X\) for \(\text{not } X\.oclIsInvalid()\) throughout this paper.

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

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

we can therefore write:

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

In Isabelle theories, this particular presentation of definitions paves the way for an automatic check that the underlying equation has the form of an axiomatic definition and is therefore logically safe. Since all operators of the assertion language depend on the context \(\tau = (\sigma, \sigma')\) and result in values that can be \(\perp\), all expressions can be viewed as evaluations from \((\sigma, \sigma')\) to a type \(\alpha\) which must possess a \(\perp\) and a null-element. Given that such constraints can be expressed in Isabelle/HOL via type classes (written: \(\alpha :: \kappa\)), all types for OCL-expressions are of a form captured by

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

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

The current OCL semantics [29, Annex A] uses different interpretation functions for invariants and pre-conditions; we achieve their semantic effect by a syntactic transformation \_\texttt{pre} which replaces, for example, all accessor functions \_\texttt{a} by their counterparts \_\texttt{a@pre}. For example, \((\text{self}. a > 5)\text{pre}\) is just \((\text{self}. a@\text{pre} > 5)\). This way, also invariants
and pre-conditions can be interpreted by the same interpretation function and have the same type of an evaluation $V(\alpha)$.

On this basis, one can define the core logical operators $\text{not}$ and $\text{and}$ as follows:

$$I[\text{not } X] = (\text{case } I[X] \text{ of}$$

$$\bot \to \bot,$$

$$||\bot|| \to ||\bot||,$$

$$||x|| \to ||\neg x||)$$

$$I[X \text{ and } Y] = (\text{case } I[X] \text{ of}$$

$$\bot \to (\text{case } I[Y] \text{ of}$$

$$\bot \to \bot,$$

$$||\bot|| \to ||\bot||,$$

$$||\text{true}|| \to \bot,$$

$$||\text{false}|| \to ||\text{false}||)$$

$$||\text{true}|| \to (\text{case } I[Y] \text{ of}$$

$$\bot \to \bot,$$

$$||\bot|| \to ||\bot||,$$

$$||\text{true}|| \to ||\bot||,$$

$$||\text{false}|| \to ||\text{false}||)$$

$$||\text{false}|| \to ||\text{false}||)$$

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

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

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

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

2.4.2. Logical Layer

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

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

$$\tau \models P \equiv (I[P]\tau = [[true]]) .$$

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

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

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

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

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

It enjoys nearly the laws of a congruence:

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

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

22
The logical layer of the Featherweight OCL rules gives also a means to convert an OCL formula living in its four-valued world into a representation that is classically two-valued and can be processed by standard SMT solvers such as CVC3 [2] or Z3 [20]. δ-closure rules for all logical connectives have the following format, e.g.:

\[
\tau \models \delta \, x \implies (\tau \models \text{not } x) = (\neg (\tau \models x))
\]

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

\[
\tau \models \delta \, x \implies (\tau \models (x \implies y)) = ((\tau \models x) \implies (\tau \models y))
\]

Together with the general case-distinction

\[
\tau \models \delta \, x \lor \tau \models x \triangleq \text{invalid } \lor \tau \models x \triangleq \text{null},
\]

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 \(\tau \models x = y + 3\) that we have \(\tau \models x \models y - 3 \land \tau \models \delta \, x \land \tau \models \delta \, y\). We call the latter formula the δ-closure of the former. Now, we can convert a formula like \(\tau \models x > 0 \lor 3 \cdot y > x \times x\) into the equivalent formula \(\tau \models x > 0 \lor \tau \models 3 \cdot y > x \times x\) 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.

### 2.4.3. Algebraic Layer

Based on the logical layer, we build a system with simpler rules which are amenable to automated reasoning. We restrict ourselves to pure equations on OCL expressions, where the used equality is the meta-(HOL-)equality.

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

<table>
<thead>
<tr>
<th>(v) invalid = false</th>
<th>(v) null = true</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v) true = true</td>
<td>(v) false = false</td>
</tr>
<tr>
<td>(\delta) invalid = false</td>
<td>(\delta) null = false</td>
</tr>
<tr>
<td>(\delta) true = true</td>
<td>(\delta) false = true</td>
</tr>
<tr>
<td>not invalid = invalid</td>
<td>not null = null</td>
</tr>
<tr>
<td>not true = false</td>
<td>not false = true</td>
</tr>
<tr>
<td>(null and true) = null</td>
<td>(null and false) = false</td>
</tr>
<tr>
<td>(null and null) = null</td>
<td>(null and invalid) = invalid</td>
</tr>
<tr>
<td>(false and true) = false</td>
<td>(false and false) = false</td>
</tr>
<tr>
<td>(false and null) = false</td>
<td>(false and invalid) = false</td>
</tr>
</tbody>
</table>
(true and true) = true \quad (true and false) = false

(true and null) = null \quad (true and invalid) = invalid

(invalid and true) = invalid

(invalid and false) = false

(invalid and null) = invalid

(invalid and invalid) = invalid

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

\[
X \land X = X \\
X \land Y = Y \land X \\
false \land X = false \\
X \land false = false \\
true \land X = X \\
X \land true = X \\
X \land (Y \land Z) = X \land Y \land Z
\]
as well as the dual equalities for \_ or \_ and the De Morgan rules. This wealth of algebraic properties makes the understanding of the logic easier as well as automated analysis possible: it allows for, for example, computing a DNF of invariant systems (by clever term-rewriting techniques) which are a prerequisite for \( \delta \)-closures.

The above equations explain the behavior for the most-important non-strict operations. The clarification of the exceptional behaviors is of key-importance for a semantic definition the standard and the major deviation point from HOL-OCL \[6, 8\], to Featherweight OCL as presented here. The standard expresses at many places that most operations are strict, i.e., enjoy the properties (exemplary for \_ + \_):

\[
invalid + X = invalid \\
X + invalid = invalid \\
X + null = invalid \\
null + X = invalid \\
null.oclAsType(X) = invalid
\]

besides “classical” exceptional behavior:

\[
1 / 0 = invalid \\
1 / null = invalid \\
null->isEmpty() = true
\]

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

\[
invalid + X = invalid \\
X + invalid = invalid \\
X + null = null \\
null + X = null \\
null.oclAsType(X) = null \\
1 / 0 = invalid \\
1 / null = null \\
null->isEmpty() = null
\]

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

\[
\begin{align*}
\text{invalid} + X &= \text{invalid} & X + \text{invalid} &= \text{invalid} \\
X + \text{null} &= X & \text{null} + X &= X \\
\text{null}.oclAsType(X) &= \text{invalid} \\
1 / 0 &= \text{invalid} & 1 / \text{null} &= \text{invalid} \\
\text{null}->\text{isEmpty}() &= \text{true}
\end{align*}
\]

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

\[
\begin{align*}
\delta \text{Set}() &= \text{true} \\
\delta (X->\text{including}(x)) &= \delta X \text{ and } \delta x \\
\text{Set}()->\text{includes}(x) &= (\text{if } \upsilon x \text{ then false} \\
& \quad \text{ else invalid endif}) \\
(X->\text{including}(x)->\text{includes}(y)) &= \\
& \quad (\text{if } \delta X \\
& \quad \text{ then if } x \neq y \\
& \quad \text{ then } \text{true} \\
& \quad \text{ else } X->\text{includes}(y) \\
& \quad \text{ endif} \\
& \quad \text{ else invalid} \\
& \quad \text{ endif})
\end{align*}
\]

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\}->includes(null)} becomes decidable in Featherweight OCL by a combination of rewriting and code-generation and execution. The generated documentation from the theory files can thus be enriched by numerous “test-statements” like:

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

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

\(^2\)In spreadsheet programs the interpretation of null varies from operation to operation; e.g., the average function treats null as non-existing value and not as 0.
2.5. **Object-oriented Datatype Theories**

As mentioned earlier, the OCL is composed of

1. operators on built-in data structures such as Boolean, Integer or Set(⊥), and
2. operators of the user-defined data model such as accessors, type casts and tests.

In the following, we will refine the concepts of a user-defined data-model (implied by a *class-model*, visualized by a class-diagram) as well as the notion of state used in the previous section to much more detail. In contrast to wide-spread opinions, UML class diagrams represent in a compact and visual manner quite complex, object-oriented data-types with a surprisingly rich theory. It is part of our endeavor here to make this theory explicit and to point out corner cases. A UML class diagram—underlying a given OCL formula—produces several implicit operations which become accessible via appropriate OCL syntax:

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

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

### 2.5.1. Object Universes

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

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

(2.12)
where \( \text{inv}_\sigma \) is a to be discussed invariant on states.

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

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

1. an object universe can be constructed for a given class model, leading to closed world semantics, and

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

For the sake of simplicity, we chose the first option for Featherweight OCL, while HOL-OCL [7] used an involved construction allowing the latter.

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

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

(2.13)

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

\[
\text{X.oclIsTypeOf}(C_k) \implies \text{X.oclAsType}(C_i).\text{oclAsType}(C_k) = X 
\]

whenever \( C_k < C_i \) and \( X \) is valid. \hspace{1cm} (2.14)

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

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

- Then the class type extension \( C_{\text{ext}} \) associated to \( C_i \) is \( A_{i_1} \times \cdots \times A_{i_n} \times (C_{j_1\text{ext}} + \cdots + C_{j_m\text{ext}}) \) where \( A_{i_k} \) ranges over the local attribute types of \( C_i \) and \( C_{j_{k\text{ext}}} \) ranges over all class type extensions of the subclass \( C_j \) of \( C_i \).
• Then the class type for $C_i$ is $\text{oid} \times A_{i1} \times \cdots \times A_{in} \times (C_{j1}\text{ext} + \cdots + C_{j\text{ext}})_\perp$ where $A_{ik}$ ranges over the inherited and local attribute types of $C_i$ and $C_{j\text{ext}}$ ranges over all class type extensions of the subclass $C_j$ of $C_i$.

Example instances of this scheme—outlining a compiler—can be found in Section 6.1 and Section 7.1.

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

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

• there is a generic theory of states, which must be formulated independently from a concrete object universe,

• there is a principle of translation (captured by the inductive scheme for class type extensions and class types above) that converts a given class model into an concrete object universe,

• there are fixed principles that allow to derive the semantic theory of any concrete object universe, called the object-oriented datatype theory.

We will work out concrete examples for the construction of the object-universes in Section 6.1 and Section 7.1 and the derivation of the respective datatype theories. While an automatization is clearly possible and desirable for concrete applications of Featherweight OCL, we consider this out of the scope of this paper which has a focus on the semantic construction and its presentation.

2.5.2. Accessors on Objects and Associations

Our choice to use a shallow embedding of OCL in HOL and, thus having an injective mapping from OCL types to HOL types, results in type-safety of Featherweight OCL. Arguments and results of accessors are based on type-safe object representations and not oid’s. This implies the following scheme for an accessor:

• The evaluation and extraction phase. If the argument evaluation results in an object representation, the oid is extracted, if not, exceptional cases like invalid are reported.

• The dereferentiation phase. The oid is interpreted in the pre- or post-state, the resulting object is casted to the expected format. The exceptional case of non-existence in this state must be treated.

• The selection phase. The corresponding attribute is extracted from the object representation.
• The re-construction phase. The resulting value has to be embedded in the adequate HOL type. If an attribute has the type of an object (not value), it is represented by an optional (set of) oid, which must be converted via dereferentiation in one of the states to produce an object representation again. The exceptional case of nonexistence in this state must be treated.

The first phase directly translates into the following formalization:

\[
\text{definition eval} = (\lambda \tau. \text{case } X \tau \text{ of } \perp \Rightarrow \text{invalid } \tau \quad \text{exception} \\
| \_\perp \Rightarrow \text{invalid } \tau \quad \text{deref. null} \\
| \_\_\text{obj}_j \Rightarrow f (\text{oid of } \text{obj}_j) \tau)
\]  

(2.16)

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

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

(2.17)

The operation yields undefined if the oid is uninterpretable in the state or referencing an object representation not conforming to the expected type.

We turn to the selection phase: for each class \( C \) in the class model with at least one attribute, and each attribute \( a \) in this class, we introduce the selection phase of this form:

\[
\text{definition select}_a = (\lambda \text{mk}_C \text{ oid } \cdots \perp \cdots \text{C}_{X\text{ext}} \Rightarrow \text{null} \\
| \text{mk}_C \text{ oid } \cdots \text{a}_j \cdots \text{C}_{X\text{ext}} \Rightarrow f (\lambda x. \_\_\_\_x_a) \text{a})
\]  

(2.18)

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:

\[
\text{definition in}_\text{pre}_\text{state} = \text{fst} \quad \text{first component} \\
\text{definition in}_\text{post}_\text{state} = \text{snd} \quad \text{second component} \\
\text{definition reconst}_\text{basetype} = \text{id} \quad \text{identity function}
\]  

(2.19)

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

\[
\text{definition \_\.getBase :: } C \Rightarrow A_{\text{base}} \\
\text{where } X.\text{getBase} = \text{eval}_{\text{extract}} X (\text{deref}_\text{oid}_C \text{ in}_\text{post}_\text{state} \\
(\text{select}_{\_\text{.getBase} \text{ reconst}_\text{basetype}}))
\]  

(2.20)

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

\[
\text{definition \_\.getObject :: } C \Rightarrow A_{\text{object}} \\
\text{where } X.\text{getObject} = \text{eval}_{\text{extract}} X (\text{deref}_\text{oid}_C \text{ in}_\text{post}_\text{state} \\
(\text{select}_{\_\text{.getObject} \text{ (deref}_\text{oid}_C \text{ in}_\text{post}_\text{state}})))
\]  

(2.21)
The variant for an accessor yielding a collection is omitted here; its construction follows by the application of the principles of the former two. The respective variants \(< a @pre\) were produced when in\_post\_state is replaced by in\_pre\_state.

Examples for the construction of accessors via associations can be found in Section 6.1.8, the construction of accessors via attributes in Section 7.1.8. 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. In case a multiplicity is specified for an attribute, i.e., a lower and an upper bound are provided, we require any collection the attribute evaluates to not contain null. This allows for a straightforward interpretation of the multiplicity constraint. If bounds are not provided for an attribute, we consider the attribute values to not be restricted in any way. Because in particular the cardinality of the attribute’s values is not bounded, the result of an evaluation of the attribute is of collection type. As the range of values that the attribute can assume is not restricted, the attribute can evaluate to a collection containing null. The attribute can also evaluate to invalid. Allowing multiplicities to be optional in this way gives the modeler the freedom to define attributes that can assume the full ranges of values provided by their types. However, we do not permit the omission of multiplicities for association ends, since the values of association ends are not only restricted by multiplicities, but also by other constraints enforcing the semantics of associations. Hence, the values of association ends cannot be completely unrestricted.

The Precise Meaning of Multiplicity Constraints

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

\[
\begin{align*}
\text{context } C & \text{ inv lowerBound : } a->\text{size}() \geq m \\
& \text{ inv upperBound : } a->\text{size}() \leq n \\
& \text{ inv notNull : not } a->\text{includes(null)}
\end{align*}
\]

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 2.5.2. If \( n \leq 1 \), the attribute \( a \) evaluates to a single value, which is then converted to a Set on which the \text{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.

2.5.3. Other Operations on States

Defining \( T . \text{allInstances}() \) is straightforward; the only difference is the property \( T . \text{allInstances}() \to \text{excludes(null)} \) which is a consequence of the fact that null’s are values and do not “live” in the state. In our semantics which admits states with

\footnote{We are however aware that a well-formedness rule of the UML standard does define a default bound of one in case a lower or upper bound is not specified.}
"dangling references," it is possible to define a counterpart to \_\_oclIsNew() called \_\_oclIsDeleted() which asks if an object id (represented by an object representation) is contained in the pre-state, but not the post-state.

OCL does not guarantee that an operation only modifies the path-expressions mentioned in the postcondition, i.e., it allows arbitrary relations from pre-states to post-states. This framing problem is well-known (one of the suggested solutions is \[23\]). 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:

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

where \(X' = I[X](\sigma, \sigma')\) and \(M = (\text{dom } \sigma \cap \text{dom } \sigma') - \{\text{OidOf } x \mid x \in \text{\lfloor }X\text{\rfloor}\}\). Thus, if we require in a postcondition \(\text{Set}() \rightarrow \text{oclIsModifiedOnly}()\) 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 \text{isQuery} property is true. So, whenever we have \(\tau \models X \rightarrow \text{excluding}(s.a) \rightarrow \text{oclIsModifiedOnly}()\) and \(\tau \models X \rightarrow \forall x (\not\equiv (x \equiv s.a))\), we can infer that \(\tau \models s.a \triangleq s.a @\text{pre}\).

2.6. A Machine-checked Annex A

Isabelle, as a framework for building formal tools \[37\], 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., \(\text{not (not x)} = x\).

Figure 2.2 illustrates this approach. Figure 2.2a shows the jEdit-based development environment of Isabelle with an excerpt of one of the core theories of Featherweight OCL. Figure 2.2b shows the generated PDF document where all antiquotations are replaced. Moreover, the document generation tools allows for defining syntactic sugar as well as skipping technical details of the formalization.
Thus, applying the Featherweight OCL approach to writing an updated Annex A that provides a formal semantics of the most fundamental concepts of OCL would ensure

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

Overall, this would contribute to one of the main goals of the OCL 2.5 RFP, as discussed at the OCL meeting in Aachen [15].
Part II.

A Proposal for Formal Semantics of OCL 2.5
3. Formalization I: Core Definitions

theory
  OCL-core
imports
  Main
begin

3.1. Preliminaries

3.1.1. 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:

notification Some ([(-)])
notification None (⊥)

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 ([(-)])
where drop-lift[simp]: [[v]] = v

3.1.2. Minimal Notions of State and State Transitions

Next we will introduce the foundational concept of an object id (oid), which is just some infinite set.

In order to assure executability of as much as possible formulas, we fixed the type of object id’s to just natural numbers.

type-synonym oid = nat

We refrained from the alternative:

type-synonym oid = ind

which is slightly more abstract but non-executable.

States are just a partial map from oid’s to elements of an object universe ′A, and state transitions pairs of states . . .

record ′A.state =
  heap :: oid ⇒ ′A
  assoc2 :: oid ⇒ (oid × oid) list
  assoc3 :: oid ⇒ (oid × oid × oid) list
3.1.3. Prerequisite: An Abstract Interface for OCL Types

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

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

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

```plaintext
class bot =
    fixes bot :: 'a
    assumes nonEmpty : ∃ x. x ≠ bot

class null = bot +
    fixes null :: 'a
    assumes null-is-valid : null ≠ bot
```

3.1.4. 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`, ...).

```plaintext
instantiation option :: (type)bot begin
definition bot-option-def: (bot::'a option) ≡ (None::'a option)
instance proof show ∃ x::'a option. x ≠ bot
    by (rule-tac x=Some x in exI, simp add:bot-option-def)
  qed
end
```
instantiation option :: (bot)null
begin
  definition null-option-def: (null::'a::bot option) ≡ [ bot ]
  instance proof show (null::'a::bot option) ≠ bot
    by simp add: null-option-def bot-option-def
  qed
end

instantiation fun :: (type,bot) bot
begin
  definition bot-fun-def: bot ≡ (λ x. bot)
  instance proof show ∃(x::'a ⇒ 'b). x ≠ bot
    apply (rule_tac x=λ - in exI, auto)
    apply (erule contrapos-pp, simp)
  done
  qed
end

instantiation fun :: (type,null) null
begin
  definition null-fun-def: (null::'a ⇒ 'b::null) ≡ (λ x. null)
  instance proof show (null::'a ⇒ 'b::null) ≠ bot
    apply (auto simp: null-fun-def bot-fun-def)
    apply (erule contrapos-pp, simp add: null-is-valid)
  done
  qed
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).

3.1.5. The Semantic Space of OCL Types: Valuations

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

type-synonym (A,α) val = A st ⇒ α::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:

**definition** Sem :: 'a ⇒ 'a (I[-])
where I[x] ≡ x

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 :: ('A,'α::bot) val
where invalid ≡ λ τ. bot

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

**lemma** textbook-invalid: I[invalid]τ = bot
by(simp add: invalid-def Sem-def)

Note that the definition :

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

is not necessary since we defined the entire function space over null types again as null-types; the crucial definition is null ≡ λx. null. Thus, the polymorphic constant *null* is simply the result of a general type class construction. Nevertheless, we can derive the semantic textbook definition for the OCL null constant based on the abstract null:

**lemma** textbook-null-fun: I[null:('A,'α::null) val] τ = (null::'α::null)
by(simp add: null-fun-def Sem-def)

### 3.2. Definition of the Boolean Type

The semantic domain of the (basic) boolean type is now defined as the Standard: the space of valuation to *bool option option*:

**type-synonym** ('A)Boolean = ('A,bool option option) val

#### 3.2.1. Basic Constants

**lemma** bot-Boolean-def : (bot::('A)Boolean) = (λ τ. ⊥)
by(simp add: bot-fun-def bot-option-def)

**lemma** null-Boolean-def : (null::('A)Boolean) = (λ τ. [⊥])
by(simp add: null-fun-def null-option-def bot-option-def)

**definition** true :: ('A)Boolean
where true ≡ λ τ. [[True]]
**Definition**

\[\text{false} :: (\forall \tau . \text{Boolean})\]

**Lemma**

\[\text{bool-split}: X \tau = \text{invalid} \tau \lor X \tau = \text{null} \tau \lor X \tau = \text{true} \tau \lor X \tau = \text{false} \tau\]

**Apply**

- (simp add: invalid-def null-def true-def false-def)
- (case-tac X \tau, simp-all add: null-fun-def null-option-def bot-option-def)
- (case-tac a, simp)
- (case-tac aa, simp)
- auto
- done

**Lemma**

\[\text{false} (a, b) = \text{false} \]

**By**

(simp add: false-def)

**Lemma**

\[\text{true} (a, b) = \text{true} \]

**By**

(simp add: true-def)

**Lemma**

\[\text{textbook-true}: I[true] \tau = \text{true} \]

**By**

(simp add: Sem-def true-def)

**Lemma**

\[\text{textbook-false}: I[false] \tau = \text{false} \]

**By**

(simp add: Sem-def false-def)

<table>
<thead>
<tr>
<th>Name</th>
<th>Theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>textbook-invalid</td>
<td>[\text{invalid} ] \tau = \text{false} \tau</td>
</tr>
<tr>
<td>textbook-null-fun</td>
<td>[\text{null} ] \tau = \text{null} \tau</td>
</tr>
<tr>
<td>textbook-true</td>
<td>[\text{true} ] \tau = \text{true} \ tau</td>
</tr>
<tr>
<td>textbook-false</td>
<td>[\text{false} ] \tau = \text{false} \tau</td>
</tr>
</tbody>
</table>

Table 3.1.: Basic semantic constant definitions of the logic (except null)

### 3.2.2. 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**

\[\text{valid} :: (\forall \tau . \text{val}) \Rightarrow (\forall \tau . \text{Boolean}) (v - [100])\]

**Where**

\[v X \equiv \lambda \tau . \text{if } X \tau = \text{bot} \tau \text{ then false } \tau \text{ else true } \tau\]

**Lemma**

\[v \text{invalid} = \text{false}\]

**By**

(rule ext, simp add: valid-def bot-fun-def bot-option-def invalid-def true-def false-def)
lemma valid2[simp]: \( \nu \text{ null} = \text{true} \)
by (rule ext, simp add: valid-def bot-fun-def bot-option-def null-is-valid
null-fun-def invalid-def true-def false-def)

lemma valid3[simp]: \( \nu \text{ true} = \text{true} \)
by (rule ext, simp add: valid-def bot-fun-def bot-option-def null-is-valid
null-fun-def invalid-def true-def false-def)

lemma valid4[simp]: \( \nu \text{ false} = \text{true} \)
by (rule ext, simp add: valid-def bot-fun-def bot-option-def null-is-valid
null-fun-def invalid-def true-def false-def)

lemma cp-valid: (\( \nu \ X \) \( \tau \)) = (\( \nu \ (\lambda \cdot X \) \( \tau \)) \( \tau \))
by (simp add: valid-def)

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

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

lemma defined1[simp]: \( \delta \ invalid = \text{false} \)
by (rule ext, simp add: defined-def bot-fun-def bot-option-def
null-def invalid-def true-def false-def)

lemma defined2[simp]: \( \delta \ null = \text{false} \)
by (rule ext, simp add: defined-def bot-fun-def bot-option-def
null-def null-option-def null-fun-def invalid-def true-def false-def)

lemma defined3[simp]: \( \delta \ true = \text{true} \)
by (rule ext, simp add: defined-def bot-fun-def bot-option-def
null-def null-option-def null-fun-def invalid-def true-def false-def)

lemma defined4[simp]: \( \delta \ false = \text{true} \)
by (rule ext, simp add: defined-def bot-fun-def bot-option-def
null-def null-option-def null-fun-def invalid-def true-def false-def)

lemma defined5[simp]: \( \delta \delta \ X = \text{true} \)
by (rule ext,
    auto simp: defined-def true-def false-def
    bot-fun-def bot-option-def null-option-def null-fun-def)

lemma defined6[simp]: \( \delta \nu \ X = \text{true} \)
by (rule ext,
    auto simp: valid-def defined-def true-def false-def)
lemma valid5[simp]: \( \nu \nu X = \text{true} \)
\hspace{1cm} by (rule ext, 
\hspace{1cm} auto simp: valid-def true-def false-def
\hspace{1cm} bot-fun-def bot-option-def null-option-def null-fun-def)

lemma valid6[simp]: \( \nu \delta X = \text{true} \)
\hspace{1cm} by (rule ext, 
\hspace{1cm} auto simp: valid-def defined-def true-def false-def
\hspace{1cm} bot-fun-def bot-option-def null-option-def null-fun-def)

lemma cp-defined: \((\delta X)\tau = (\lambda X. \tau)\)
\hspace{1cm} by (simp add: defined-def)

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

lemma textbook-defined: \( I J \delta(X) \tau = (if I J X \tau = I J \text{bot} \tau \lor I J X \tau = I J \text{null} \tau \
then I J \text{false} \tau 
else I J \text{true} \tau) \)
\hspace{1cm} by (simp add: Sem-def defined-def)

lemma textbook-valid: \( I J \nu(X) \tau = (if I J X \tau = I J \text{bot} \tau \then I J \text{false} \tau 
else I J \text{true} \tau) \)
\hspace{1cm} by (simp add: Sem-def valid-def)

Table 3.2 and Table 3.3 summarize the results of this section.

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

Table 3.2.: Basic predicate definitions of the logic.

### 3.3. The Equalities of OCL

The OCL contains a particular version of equality, written in Standard documents \_ = \_ and \_ \_\_ \_ as its negation, which is referred as weak referential equality hereafter and for which we use the symbol \_ \_\_ \_ 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 \( \equiv \), which is not present in the current standard but was discussed in prior texts on OCL like the Amsterdam Manifesto [19] and was identified as desirable extension of OCL in the Aachen Meeting [15] 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 \( \text{a.boss} \equiv \text{b.boss@pre} \) instead of
   \( \text{a.boss} = \text{b.boss@pre} \) and (\( \ast \text{just the pointers are equal!} \ast \))
   \( \text{a.boss.name} \equiv \text{b.boss@pre.name@pre} \) and
   \( \text{a.boss.age} \equiv \text{b.boss@pre.age@pre} \)

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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>defined1</td>
<td>( \delta \ invalid = false )</td>
</tr>
<tr>
<td>defined2</td>
<td>( \delta \ null = false )</td>
</tr>
<tr>
<td>defined3</td>
<td>( \delta \ true = true )</td>
</tr>
<tr>
<td>defined4</td>
<td>( \delta \ false = true )</td>
</tr>
<tr>
<td>defined5</td>
<td>( \delta \ ?X = true )</td>
</tr>
<tr>
<td>defined6</td>
<td>( \delta \upsilon \ ?X = true )</td>
</tr>
</tbody>
</table>

Table 3.3.: Laws of the basic predicates of the logic.
that equality is defined in HOL itself. We can express Leibniz principle as one logical rule of surprising simplicity and beauty:

\[
    s = t \implies P(s) = P(t)
\]  

(3.1)

“When 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.

3.3.1. Definition

The strict equality on basic types (actually on all types) must be exceptionally defined on \( \text{null} \)—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 \( \text{null} \) or \( \bot \) element. Strong equality is simply polymorphic in Featherweight OCL, i.e., is defined identical for all types in OCL and HOL.

\[
\text{defn} \text{StrongEq}: [\forall \alpha \Rightarrow \forall \alpha \Rightarrow (\forall \alpha)\text{Boolean} \ (\text{infixl} \triangleq 30)
\]

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

From this follow already elementary properties like:

\[
\text{lemma [simp, code-unfold]}: (\text{true} \triangleq \text{false}) = \text{false}
\]
\text{by}(\text{rule ext, auto simp: StrongEq-def})

\[
\text{lemma [simp, code-unfold]}: (\text{false} \triangleq \text{true}) = \text{false}
\]
\text{by}(\text{rule ext, auto simp: StrongEq-def})

In contrast, referential equality behaves differently for all types—on value types, it is basically strong equality for defined values, but on object types it will compare references—we introduce it as an \textit{overloaded} concept and will handle it for each type instance individually.

\[
\text{consts} \text{StrictRefEq} :: [(\forall \alpha)\text{val},(\forall \alpha)\text{val}] \Rightarrow (\forall \alpha)\text{Boolean} \ (\text{infixl} \triangleq 30)
\]

Here is a first instance of a definition of weak equality—for the special case of the type \( \forall \text{Boolean} \), it is just the strict extension of the logical equality:

\[
\text{defs (overloaded)} \quad \text{StrictRefEq\_Boolean}[\text{code-unfold}] :
\]
\[
(x:(\forall \text{Boolean}) \triangleq y \equiv \lambda \tau. \text{if } (\nu x) \tau = \text{true } \tau \wedge (\nu y) \tau = \text{true } \tau \text{ then } (x \triangleq y)\tau)
\]
which implies elementary properties like:

\[
\text{lemma } [\text{simp, code-unfold}] : (\text{true} \equiv \text{false}) = \text{false}
\]
by (simp add: StrictRefEq\_Boolean)

\[
\text{lemma } [\text{simp, code-unfold}] : (\text{false} \equiv \text{true}) = \text{false}
\]
by (simp add: StrictRefEq\_Boolean)

\[
\text{lemma } [\text{simp, code-unfold}] : (\text{invalid} \equiv \text{false}) = \text{invalid}
\]
by (simp add: StrictRefEq\_Boolean false-def true-def)

\[
\text{lemma } [\text{simp, code-unfold}] : (\text{invalid} \equiv \text{true}) = \text{invalid}
\]
by (simp add: StrictRefEq\_Boolean false-def true-def)

\[
\text{lemma } [\text{simp, code-unfold}] : (\text{false} \equiv \text{invalid}) = \text{invalid}
\]
by (simp add: StrictRefEq\_Boolean false-def true-def)

\[
\text{lemma } [\text{simp, code-unfold}] : (\text{true} \equiv \text{invalid}) = \text{invalid}
\]
by (simp add: StrictRefEq\_Boolean false-def true-def)

\[
\text{lemma } [\text{simp, code-unfold}] : ((\text{invalid} :: (\exists) \text{Boolean}) \equiv \text{invalid}) = \text{invalid}
\]
by (simp add: StrictRefEq\_Boolean false-def true-def)

Thus, the weak equality is not reflexive.

\[
\text{lemma null-non-false } [\text{simp, code-unfold}] : (\text{null} \equiv \text{false}) = \text{false}
\]
apply (rule ext, simp add: StrictRefEq\_Boolean StrongEq-def false-def)
by (metis OCL-core.drop.simps cp-valid false-def is-none-code(2) is-none-def valid4
    bot-option-def null-fun-def null-option-def)

\[
\text{lemma null-non-true } [\text{simp, code-unfold}] : (\text{null} \equiv \text{true}) = \text{false}
\]
apply (rule ext, simp add: StrictRefEq\_Boolean StrongEq-def false-def)
by (simp add: true-def bot-option-def null-fun-def null-option-def)

\[
\text{lemma false-non-null } [\text{simp, code-unfold}] : (\text{false} \equiv \text{null}) = \text{false}
\]
apply (rule ext, simp add: StrictRefEq\_Boolean StrongEq-def false-def)
by (metis OCL-core.drop.simps cp-valid false-def is-none-code(2) is-none-def valid4
    bot-option-def null-fun-def null-option-def)

\[
\text{lemma true-non-null } [\text{simp, code-unfold}] : (\text{true} \equiv \text{null}) = \text{false}
\]
apply (rule ext, simp add: StrictRefEq\_Boolean StrongEq-def false-def)
by (simp add: true-def bot-option-def null-fun-def null-option-def)

3.3.2. Fundamental Predicates on Strong Equality

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

\[
\text{lemma StrongEq-refl } [\text{simp}] : (X \triangleq X) = \text{true}
\]
by (rule ext, simp add: null-def invalid-def true-def false-def StrongEq-def)

\[
\text{lemma StrongEq-sym} : (X \triangleq Y) = (Y \triangleq X)
\]
by (rule ext, simp add: eq-sym-conv invalid-def true-def false-def StrongEq-def)

46
lemma StrongEq-trans-strong [simp]:
assumes A: (X ≜ Y) = true
and B: (Y ≜ Z) = true
shows (X ≜ Z) = true
apply(insert A B) apply(rule ext)
apply(simp add: null-def invalid-def true-def false-def StrongEq-def)
apply(drule-tac x=x in fun-cong)+
by auto

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: ∀X. P(X)τ = P(λ -. X τ)τ
and eq: (X ≜ Y)τ = true τ
shows (P X ≜ P Y)τ = true τ
apply(insert cp eq)
apply(simp add: null-def invalid-def true-def false-def StrongEq-def)
apply(subst cp[of X])
apply(subst cp[of Y])
by simp

lemma defined7 [simp]: δ (X ≜ Y) = true
by(rule ext,
  auto simp: defined-def true-def false-def StrongEq-def
  bot-fun-def bot-option-def null-option-def null-fun-def)

lemma valid7 [simp]: υ (X ≜ Y) = true
by(rule ext,
  auto simp: valid-def true-def false-def StrongEq-def
  bot-fun-def bot-option-def null-option-def null-fun-def)

lemma cp-StrongEq: (X ≜ Y) τ = ((λ -. X τ) ≜ (λ -. Y τ)) τ
by(simp add: StrongEq-def)

3.4. Logical Connectives and their Universal Properties

It is a design goal to give OCL a semantics that is as closely as possible to a “logical
system” in a known sense; a specification logic where the logical connectives can not be
understood other that having the truth-table aside when reading fails its purpose in our
view.
Practically, this means that we want to give a definition to the core operations to be as close as possible to the lattice laws; this makes also powerful symbolic normalization of OCL specifications possible as a pre-requisite for automated theorem provers. For example, it is still possible to compute without any definedness and validity reasoning the DNF of an OCL specification; be it for test-case generations or for a smooth transition to a two-valued representation of the specification amenable to fast standard SMT-solvers, for example.

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

definition OclNot :: (A)Boolean ⇒ (A)Boolean (not)
where
  not X ≡ λ τ . case X τ of
  ⊥ ⇒ ⊥
  | [ ⊥ ] ⇒ [ ⊥ ]
  | [ [ x ] ] ⇒ [ [ ¬ x ] ]

with term ”not” we can express the notation:

syntax
notequal :: (A)Boolean ⇒ (A)Boolean ⇒ (A)Boolean (infix <> 40)
translations
a <> b == CONST OclNot( a ≠ b)

lemma cp-OclNot: (not X)τ = (not (λ - . X τ)) τ
by(simp add: OclNot-def)

lemma OclNot1[simp]: not invalid = invalid
by(rule ext,simp add: OclNot-def null-def invalid-def true-def false-def bot-option-def )

lemma OclNot2[simp]: not null = null
by(rule ext,simp add: OclNot-def null-def invalid-def true-def false-def
    bot-option-def null-fun-def null-option-def )

lemma OclNot3[simp]: not true = false
by(rule ext,simp add: OclNot-def null-def invalid-def true-def false-def)

lemma OclNot4[simp]: not false = true
by(rule ext,simp add: OclNot-def null-def invalid-def true-def false-def)

lemma OclNot-not[simp]: not (not X) = X
  apply(rule ext,simp add: OclNot-def null-def invalid-def true-def false-def)
  apply(case-tac X x, simp-all)
  apply(case-tac a, simp-all)
  done

lemma OclNot-inject: ∧ x y. not x = not y ⇒ x = y
by(subst OclNot-not[THEN sym], simp)
definition OclAnd :: \((\mathbb{A})\)Boolean, \((\mathbb{A})\)Boolean \rightarrow (\mathbb{A})\)Boolean \ ((\text{infixl and } 30)\)
where \(X \text{ and } Y \equiv \lambda \tau \cdot \text{case } X \; \tau \text{ of}
\begin{align*}
| [[\text{False}]] \Rightarrow & [[\text{False}]] \\
| \bot \Rightarrow & (\text{case } Y \; \tau \text{ of} \\
& [[\text{False}]] \Rightarrow [[\text{False}]] \\
& | \bot \Rightarrow \bot \\
& | \bot \Rightarrow [[[\bot]]]) \\
| [[\text{True}]] \Rightarrow & (\text{case } Y \; \tau \text{ of} \\
& \bot \Rightarrow \bot \\
& | \bot \Rightarrow [[[\bot]]]) \\
& | [[[\bot]]]) \Rightarrow [[[\bot]]]) \\
& \text{by} (\text{simp add: } \text{OclNot-def})
\end{align*}

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

In textbook notation, the logical core constructs \textit{not} and \textit{op and} were represented as follows:

\begin{align*}
\text{lemma textbook-OclNot:} \\
I\{\text{not}(X)\} \; \tau = (\text{case } I\{X\} \; \tau \text{ of} \\
& \bot \Rightarrow \bot \\
& | \bot \Rightarrow [[[\bot]]]) \\
& | [[[\text{True}]]]) \Rightarrow [[[\text{False}]]]) \\
& | [[[\text{False}]]}) \Rightarrow [[[\text{False}]]]) \\
& \text{by (simp add: } \text{OclNot-def})
\end{align*}

\begin{align*}
\text{definition OclImplies :: } \((\mathbb{A})\)Boolean, \((\mathbb{A})\)Boolean \Rightarrow (\mathbb{A})\)Boolean \ ((\text{infixl implies } 25)\)
\end{align*}

\begin{align*}
\text{definition OclOr :: } \((\mathbb{A})\)Boolean, \((\mathbb{A})\)Boolean \Rightarrow (\mathbb{A})\)Boolean \ ((\text{infixl or } 30)\)
\end{align*}
by(simp add: OclAnd-def)

lemma cp-OclOr:((X::(α)Boolean) or Y) τ = ((λ -. X τ) or (λ -. Y τ)) τ
apply(simp add: OclOr-def)
apply(subst cp-OclNot[of not (λ -. X τ) and not (λ -. Y τ)])
apply(subst cp-OclAnd[of not (λ -. X τ) not (λ -. Y τ)])
by(simp add: cp-OclNot[symmetric] cp-OclAnd[symmetric] )

lemma cp-OclImplies:(X implies Y) τ = ((λ -. X τ) implies (λ -. Y τ)) τ
apply(simp add: OclImplies-def)
apply(subst cp-OclOr[of not (λ -. X τ) (λ -. Y τ)])
by(simp add: cp-OclNot[symmetric] cp-OclOr[symmetric] )

lemma OclAnd1[simp]: (invalid and true) = invalid
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def)
lemma OclAnd2[simp]: (invalid and false) = false
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def)
lemma OclAnd3[simp]: (invalid and null) = invalid
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def
      null-fun-def null-option-def)
lemma OclAnd4[simp]: (invalid and invalid) = invalid
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def)

lemma OclAnd5[simp]: (null and true) = null
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def
      null-fun-def null-option-def)
lemma OclAnd6[simp]: (null and false) = false
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def
      null-fun-def null-option-def)
lemma OclAnd7[simp]: (null and null) = null
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def
      null-fun-def null-option-def)
lemma OclAnd8[simp]: (null and invalid) = invalid
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def
      null-fun-def null-option-def)
lemma OclAnd9[simp]: (false and true) = false
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def)
lemma OclAnd10[simp]: (false and false) = false
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def)
lemma OclAnd11[simp]: (false and null) = false
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def)
lemma OclAnd12[simp]: (false and invalid) = false
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def)
lemma OclAnd13[simp]: (true and true) = true
  by(rule ext,simp add: OclAnd-def null-def invalid-def true-def false-def)
lemma OclAnd14[simp]: (true and false) = false
by (rule ext, simp add: OclAnd-def null-def invalid-def true-def false-def)

lemma OclAnd15 [simp]: (true and null) = null
  by (rule ext, simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def
      null-fun-def null-option-def)

lemma OclAnd16 [simp]: (true and invalid) = invalid
  by (rule ext, simp add: OclAnd-def null-def invalid-def true-def false-def bot-option-def
      null-fun-def null-option-def)

lemma OclAnd-idem [simp]: (X and X) = X
  apply (rule ext, simp add: OclAnd-def)
  apply (case-tac X x)
  apply (simp-all)
  apply (case-tac a)
  apply (simp-all)
  apply (case-tac aa)
  apply (simp-all)
  done

lemma OclAnd-commute: (X and Y) = (Y and X)
  by (rule ext, auto simp: true-def false-def OclAnd-def invalid-def
      split: option.split option.split-asm
      bool.split bool.split-asm)

lemma OclAnd-false1 [simp]: (false and X) = false
  apply (rule ext, simp add: OclAnd-def)
  apply (auto simp: true-def false-def invalid-def
      split: option.split option.split-asm)
  done

lemma OclAnd-false2 [simp]: (X and false) = false
  by (simp add: OclAnd-commute)

lemma OclAnd-true1 [simp]: (true and X) = X
  apply (rule ext, simp add: OclAnd-def)
  apply (auto simp: true-def false-def invalid-def
      split: option.split option.split-asm)
  done

lemma OclAnd-true2 [simp]: (X and true) = X
  by (simp add: OclAnd-commute)

lemma OclAnd-bot1 [simp]: \( \forall \tau. \; \tau \neq \text{false} \Rightarrow (\text{bot and } X) \; \tau = \text{bot} \; \tau \)
  apply (simp add: OclAnd-def)
  apply (auto simp: true-def false-def bot-fun-def bot-option-def
      split: option.split option.split-asm)
  done

lemma OclAnd-bot2 [simp]: \( \forall \tau. \; \tau \neq \text{false} \Rightarrow (X \text{ and } \text{bot}) \; \tau = \text{bot} \; \tau \)
  by (simp add: OclAnd-commute)
lemma OclAnd-null1 [simp]: \( \forall \tau. \ X \tau \neq false \implies X \tau \neq bot \implies (null \ and X) \tau = null \tau \)
apply (simp add: OclAnd-def)
apply (auto simp: true-def false-def bot-fun-def bot-option-def null-fun-def null-option-def
split: option.split option.split-asm)
done

lemma OclAnd-null2 [simp]: \( \forall \tau. \ X \tau \neq false \implies X \tau \neq bot \implies (null \ and X) \tau = null \tau \)
by (simp add: OclAnd-commute)

lemma OclAnd-assoc: \((X \ and \ (Y \ and Z)) = (X \ and Y \ and Z)\)
apply (rule ext, simp add: OclAnd-def)
apply (auto simp: true-def false-def null-def invalid-def
split: option.split option.split-asm
bool.split bool.split-asm)
done

lemma OclOr1 [simp]: \((invalid \ or \ true) = true\)
by (rule ext, simp add: OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def
bot-option-def)

lemma OclOr2 [simp]: \((invalid \ or \ false) = invalid\)
by (rule ext, simp add: OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def
bot-option-def)

lemma OclOr3 [simp]: \((invalid \ or \ null) = invalid\)
by (rule ext, simp add: OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def
bot-option-def null-fun-def null-option-def)

lemma OclOr4 [simp]: \((invalid \ or \ invalid) = invalid\)
by (rule ext, simp add: OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def
bot-option-def)

lemma OclOr5 [simp]: \((null \ or \ true) = true\)
by (rule ext, simp add: OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def
bot-option-def null-fun-def null-option-def)

lemma OclOr6 [simp]: \((null \ or \ false) = null\)
by (rule ext, simp add: OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def
bot-option-def null-fun-def null-option-def)

lemma OclOr7 [simp]: \((null \ or \ null) = null\)
by (rule ext, simp add: OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def
bot-option-def null-fun-def null-option-def)

lemma OclOr8 [simp]: \((null \ or \ invalid) = invalid\)
by (rule ext, simp add: OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def
bot-option-def null-fun-def null-option-def)

lemma OclOr-idem [simp]: \((X \ or \ X) = X\)
by (simp add: OclOr-def)

lemma OclOr-commute: \((X \ or \ Y) = (Y \ or \ X)\)
by (simp add: OclOr-def OclAnd-commute)
lemma OclOr-false1 [simp]: (false or Y) = Y
  by (simp add: OclOr-def)

lemma OclOr-false2 [simp]: (Y or false) = Y
  by (simp add: OclOr-def)

lemma OclOr-true1 [simp]: (true or Y) = true
  by (simp add: OclOr-def)

lemma OclOr-true2: (Y or true) = true
  by (simp add: OclOr-def)

lemma OclOr-bot1 [simp]: \( \forall \tau. \ X \tau \neq true \forall \tau \Rightarrow (\bot or X) \tau = \bot \tau \)
  apply (simp add: OclOr-def OclAnd-def OclNot-def
             split: option.split option.split-asm)
  done

lemma OclOr-bot2 [simp]: \( \forall \tau. \ X \tau \neq true \forall \tau \Rightarrow (X or \bot) \tau = \bot \tau \)
  by (simp add: OclOr-commute)

lemma OclOr-null1 [simp]: \( \forall \tau. \ X \tau \neq true \forall \tau \Rightarrow X \tau \neq \bot \forall \tau \Rightarrow (null or X) \tau = \null \tau \)
  apply (auto simp:true-def false-def bot-fun-def bot-option-def
         split: option.split option.split-asm)
  apply (metis (full-types) bool.simps(3) bot-option-def null-is-valid null-option-def)
  by (metis (full-types) bool.simps(3) option.distinct(1) option.sel)

lemma OclOr-null2 [simp]: \( \forall \tau. \ X \tau \neq true \forall \tau \Rightarrow X \tau \neq \bot \forall \tau \Rightarrow (X or \null) \tau = \null \tau \)
  by (simp add: OclOr-commute)

lemma OclOr-assoc: (X or (Y or Z)) = (X or Y or Z)
  by (simp add: OclOr-def OclAnd-assoc)

lemma OclImplies-true: (X implies true) = true
  by (simp add: OclImplies-def OclOr-true2)

lemma deMorgan1: not(X and Y) = ((not X) or (not Y))
  by (simp add: OclOr-def)

lemma deMorgan2: not(X or Y) = ((not X) and (not Y))
  by (simp add: OclOr-def)

3.5. A Standard Logical Calculus for OCL

definition OclValid :: [('A)st, ('A)Boolean] ⇒ bool ((1(·)/ |= (·)) 50)
where  \( \tau \models P \equiv ((P \tau) = true \tau) \)

value \( \tau \models true <> false \)
value $\tau \models false \leftrightarrow true$

3.5.1. Global vs. Local Judgements

lemma transform1: $P = true \implies \tau \models P$
by(simp add: OclValid-def)

lemma transform1-rev: $\forall \tau. \tau \models P \implies P = true$
by(rule ext, auto simp: OclValid-def true-def)

lemma transform2: ($P = Q$) $\implies$ (($\tau \models P$) $\implies$ (\tau \models Q))
by(auto simp: OclValid-def)

lemma transform2-rev: $\forall \tau. (\tau \models \delta P) \land (\tau \models \delta Q) \land (\tau \models P) \implies (\tau \models Q) \implies P = Q$
apply(rule ext, auto simp: OclValid-def true-def defined-def)
apply(erule-tac x=\a in allE)
apply(erule-tac x=\b in allE)
apply(auto simp: false-def true-def defined-def bot-Boolean-def null-Boolean-def
split: option.split-asn HOL.split-if-asm)
done

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

lemma
assumes H : $P = true \implies Q = true$
shows $\tau \models P \implies \tau \models Q$
apply(simp add: OclValid-def)
apply(rule H[THEN fun-cong])
apply(rule ext)
oops

3.5.2. Local Validity and Meta-logic

lemma foundation1[simp]: $\tau \models true$
by(auto simp: OclValid-def)

lemma foundation2[simp]: $\neg(\tau \models false)$
by(auto simp: OclValid-def true-def false-def)

lemma foundation3[simp]: $\neg(\tau \models invalid)$
by(auto simp: OclValid-def true-def false-def invalid-def bot-option-def)

lemma foundation4[simp]: $\neg(\tau \models null)$
by(auto simp: OclValid-def true-def false-def invalid-def bot-option-def
null-def null-fun-def null-option-def)

lemma bool-split-local[simp]: ($\tau \models (x \triangleq invalid)) \lor (\tau \models (x \triangleq null)) \lor (\tau \models (x \triangleq true)) \lor (\tau \models (x \triangleq false))$
apply(insert bool-split[of \tau], auto)
apply(simp-all add: OclValid-def StrongEq-def true-def null-def invalid-def)
done

lemma def-split-local:
(\(\tau \mid \delta \ x\)) = ((\neg (\tau \mid (x \triangleq invalid))) \land (\neg (\tau \mid (x \triangleq null))))
by(simp add: defined-def true-def false-def invalid-def null-def
StrongEq-def OclValid-def bot-fun-def null-fun-def)

lemma foundation5:
\(\tau \mid P \land Q \implies (\tau \mid P) \land (\tau \mid Q)\)
by(simp add: OclAnd-def OclValid-def true-def false-def defined-def
split: option.split option.split-asn bool.split bool.split-asn)

lemma foundation6:
\(\tau \mid \delta \ x \implies \tau \mid \delta P\)
by(simp add: OclNot-def OclValid-def true-def false-def defined-def
null-option-def null-fun-def bot-option-def bot-fun-def
split: option.split option.split-asn)

lemma foundation7[simp]:
(\(\tau \mid not \ (\delta \ x)\)) = (\neg (\tau \mid \delta \ x))
by(simp add: OclNot-def OclValid-def true-def false-def defined-def
split: option.split option.split-asn)

lemma foundation7'[simp]:
(\(\tau \mid not \ (\nu \ x)\)) = (\neg (\tau \mid \nu \ x))
by(simp add: OclNot-def OclValid-def true-def false-def valid-def
split: option.split option.split-asn)

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

lemma foundation8:
(\(\tau \mid \delta \ x\) \lor (\tau \mid (x \triangleq invalid)) \lor (\tau \mid (x \triangleq null)))
proof
  have 1: (\(\tau \mid \delta \ x\) \lor (\neg (\tau \mid \delta \ x))) by auto
  have 2: (\neg (\tau \mid \delta \ x)) = ((\tau \mid (x \triangleq invalid)) \lor (\tau \mid (x \triangleq null)))
    by(simp only: def-split-local, simp)
  show \(?thesis\) by(insert 1, simp add:2)
qed

lemma foundation9:
\(\tau \mid \delta \ x \implies (\tau \mid not \ x) = (\neg (\tau \mid x))\)
apply(simp add: def-split-local )
by(auto simp: OclNot-def null-fun-def null-option-def bot-option-def
OclValid-def invalid-def true-def null-def StrongEq-def)

55
lemma foundation10:
\[ \tau \models \delta \; x \implies \tau \models \delta \; y \implies (\tau \models (x \text{ and } y)) = ((\tau \models x) \land (\tau \models y)) \]
apply(simp add: def-split-local)
by(auto simp: OclAnd-def OclValid-def invalid-def true-def null-def StrongEq-def null-option-def bot-option-def split:bool.split-asm)

lemma foundation11:
\[ \tau \models \delta \; x \implies \tau \models \delta \; y \implies (\tau \models (x \text{ or } y)) = ((\tau \models x) \lor (\tau \models y)) \]
apply(simp add: def-split-local)
by(auto simp: OclNot-def OclOr-def OclAnd-def OclValid-def invalid-def true-def null-def StrongEq-def null-fun-def null-option-def bot-option-def split:bool.split-asm bool.split)

lemma foundation12:
\[ \tau \models \delta \; x \implies \tau \models \delta \; y \implies (\tau \models (x \implies y)) = ((\tau \models x) \rightarrow (\tau \models y)) \]
apply(simp add: def-split-local)
by(auto simp: OclNot-def OclOr-def OclAnd-def OclImplies-def bot-option-def OclValid-def invalid-def true-def null-def StrongEq-def null-fun-def null-option-def bot-option-def split:bool.split-asm bool.split)

lemma foundation13:
\[ (\tau \models A \triangleq \text{true}) = (\tau \models A) \]
by(auto simp: OclNot-def OclValid-def invalid-def true-def null-def StrongEq-def split:bool.split-asm bool.split)

lemma foundation14:
\[ (\tau \models A \triangleq \text{false}) = (\tau \models \text{not } A) \]
by(auto simp: OclNot-def OclValid-def invalid-def false-def true-def null-def split:bool.split-asm bool.split option.split)

lemma foundation15:
\[ (\tau \models A \triangleq \text{invalid}) = (\tau \not\models \text{not}(\forall A)) \]

lemma foundation16:
\[ (X \tau \not\models \text{bot} \land X \tau \not\models \text{null}) \]
by(auto simp: OclValid-def defined-def false-def true-def bot-fun-def null-fun-def split:split-if-asm)

lemma foundation16’:
\[ (X \tau \not\models \text{invalid} \land X \tau \not\models \text{null} \tau) \]
apply(simp add:invalid-def null-def null-fun-def)
by(auto simp: OclValid-def defined-def false-def true-def bot-fun-def null-fun-def split:split-if-asm)
lemmas foundation17 = foundation16[THEN iffD1]

lemmas foundation17' = foundation16'[THEN iffD1]

lemma foundation18: \( \tau \models (v \ X) = (X \ \tau \neq \text{invalid}) \)
by(auto simp: OclValid-def valid-def false-def true-def bot-fun-def invalid-def
split:split-if-asm)

lemma foundation18': \( \tau \models (v \ X) = (X \ \tau \neq \text{bot}) \)
by(auto simp: OclValid-def valid-def false-def true-def bot-fun-def
split:split-if-asm)

lemmas foundation19 = foundation18[THEN iffD1]

lemma foundation20: \( \tau \models (\delta \ X) \implies \tau \models v \ X \)
by(simp add: foundation18 foundation16 invalid-def)

lemma foundation21: \( \text{not} \ A \triangleq \text{not} \ B = (A \triangleq B) \)
by(rule ext, auto simp: OclNot-def StrongEq-def
split:bool.split-asm HOL.split-if-asm option.split)

lemma foundation22: \( \tau \models (X \triangleq Y) = (X \ \tau = Y \ \tau) \)
by(auto simp: OclValid-def true-def)

lemma foundation23: \( \tau \models P = (\tau \models (\lambda-\cdot P)) \)
by(auto simp: OclValid-def true-def)

lemmas cp-validity=foundation23

lemma foundation24:(\( \tau \models \text{not} (X \triangleq Y) \)) = (\( X \ \tau \neq Y \ \tau \))
by(simp add: StrongEq-def OclValid-def OclNot-def true-def)

lemma defined-not-I : \( \tau \models \delta \ (x) \implies \tau \models \delta \ (\text{not} \ x) \)
by(auto simp: OclNot-def null-def invalid-def defined-def valid-def OclValid-def
true-def false-def bot-option-def null-option-def bot-fun-def
split:option.split-asm HOL.split-if-asm)

lemma valid-not-I : \( \tau \models v \ (x) \implies \tau \models v \ (\text{not} \ x) \)
by(auto simp: OclValid-def null-def invalid-def defined-def valid-def OclValid-def
true-def false-def bot-option-def null-option-def bot-fun-def
split:option.split-asm option.split HOL.split-if-asm)

lemma defined-and-I : \( \tau \models \delta \ (x) \implies \tau \models \delta \ (y) \implies \tau \models \delta \ (x \ and \ y) \)
apply(simp add: OclAnd-def null-def invalid-def defined-def valid-def OclValid-def
true-def false-def bot-option-def null-option-def bot-fun-def
split:option.split-asm HOL.split-if-asm)
apply (auto simp: null-option-def split: bool.split)
by (case_tac ya, simp-all)

lemma valid-and-I : τ |= v (x) ⇒ τ |= v (y) ⇒ τ |= v (x and y)
apply (simp add: OclAnd-def null-def invalid-def defined-def valid-def OclValid-def
true-def false-def bot-option-def null-option-def null-fun-def bot-fun-def
split: option.split-asm HOL.split-if-asm)
by (auto simp: null-option-def split: option.split bool.split)

3.5.3. Local Judgements and Strong Equality

lemma StrongEq-L-refl: τ |= (x ≜ x)
by (simp add: OclValid-def StrongEq-def)

lemma StrongEq-L-sym: τ |= (x ≜ y) ⇒ τ |= (y ≜ x)
by (simp add: StrongEq-sym)

lemma StrongEq-L-trans: τ |= (x ≜ y) ⇒ τ |= (y ≜ z) ⇒ τ |= (x ≜ z)
by (simp add: OclValid-def StrongEq-def true-def)

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 :: ((A,α) val ⇒ (A,β) val) ⇒ bool
where
  cp P ≡ (∃ f. ∀ X τ. P X τ = f (X τ) τ)

The rule of substitutivity in Featherweight OCL holds only for context-passing expres-
sions, 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: (∀τ. cp P ⇒ τ |= (x ≜ y) ⇒ τ |= (P x ≜ P y))
by (auto simp: OclValid-def StrongEq-def)

lemma StrongEq-L-subst2:
∀τ. cp P ⇒ τ |= (x ≜ y) ⇒ τ |= (P x) ⇒ τ |= (P y)
by (auto simp: OclValid-def StrongEq-def)

lemma StrongEq-L-subst2-rev: τ |= y ≜ x ⇒ cp P ⇒ τ |= P x ⇒ τ |= P y
apply (erule StrongEq-L-subst2)
apply (erule StrongEq-L-sym)
by assumption

lemma StrongEq-L-subst3:
assumes cp: cp P
and eq: τ |= x ≜ y
shows (τ |= P x) = (τ |= P y)
apply (rule iffI)
apply (rule OCL-core.StrongEq-L-subst2[OF cp,OF eq],simp)
apply (rule OCL-core.StrongEq-L-subst2[OF cp,OF eq[THEN StrongEq-L-sym]],simp)
lemma cpI1:
\((\forall X \tau. f X \tau = f(\lambda -. X \tau) \tau) \Longrightarrow cp P \Longrightarrow cp(\lambda X. f (P X))\)
apply(auto simp: true-def cp-def)
apply(rule exI, (rule allI)+)
by(erule-tac x=P X in allE, auto)

lemma cpI2:
\((\forall X Y \tau. f X Y \tau = f(\lambda -. X \tau)(\lambda -. Y \tau) \tau) \Longrightarrow cp P \Longrightarrow cp Q \Longrightarrow cp(\lambda X. f (P X) (Q X))\)
apply(auto simp: true-def cp-def)
apply(rule exI, (rule allI)+)
by(erule-tac x=P X in allE, auto)

lemma cpI3:
\((\forall X Y Z \tau. f X Y Z \tau = f(\lambda -. X \tau)(\lambda -. Y \tau)(\lambda -. Z \tau) \tau) \Longrightarrow cp P \Longrightarrow cp Q \Longrightarrow cp R \Longrightarrow cp(\lambda X. f (P X) (Q X) (R X))\)
apply(auto simp: cp-def)
apply(rule exI, (rule allI)+)
by(erule-tac x=P X in allE, auto)

lemma cpI4:
\((\forall W X Y Z \tau. f W X Y Z \tau = f(\lambda -. W \tau)(\lambda -. X \tau)(\lambda -. Y \tau)(\lambda -. Z \tau) \tau) \Longrightarrow cp P \Longrightarrow cp Q \Longrightarrow cp R \Longrightarrow cp S \Longrightarrow cp(\lambda X. f (P X) (Q X) (R X) (S X))\)
apply(auto simp: cp-def)
apply(rule exI, (rule allI)+)
by(erule-tac x=P X in allE, auto)

lemma cp-const : cp(\lambda -. c)
by (simp add: cp-def, fast)

lemma cp-id : cp(\lambda X. X)
by (simp add: cp-def, fast)

lemmas cp-intro[introl,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-OcINot[THEN allI[THEN allI[THEN cpI1, of not]]]
    cp-OcIAnd[THEN allI[THEN allI[THEN allI[THEN cpI2, of op and]]]
    cp-OcIOr[THEN allI[THEN allI[THEN allI[THEN cpI2, of op or]]
    cp-OcImplies[THEN allI[THEN allI[THEN allI[THEN cpI2, of op implies]]}]
    cp-StrongEq[THEN allI[THEN allI[THEN allI[THEN cpI2, of StrongEq]]]}

59
3.5.4. Laws to Establish Definedness ($\delta$-closure)

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

Lemma $OclNot-defargs$:

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

by (auto simp: OclNot-def OclValid-def true-def invalid-def defined-def false-def
bot-fun-def bot-option-def null-fun-def null-option-def
split: bool.split-asm HOL.split-if-asm option.split option.split-asm)

Lemma $OclNot-contrapos-nn$:

assumes $\tau \models \delta A$

assumes $\tau \models \neg B$

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

shows $\tau \models \neg A$

proof

have change-not : $\forall a\ b\ (\neg a\ \tau = b\ \tau) = (a\ \tau = \neg b\ \tau)$

by (metis OclNot-not cp-OclNot)

show $\neg \thesis$

apply (insert assms, simp add: OclValid-def, subst change-not, subst (asm) change-not)

apply (simp add: OclNot-def true-def)

by (metis OclValid-def bool-split defined-def false-def
foundation2 true-def bot-fun-def invalid-def)

qed

So far, we have only one strict Boolean predicate (-family): the strict equality.

3.6. Miscellaneous

3.6.1. OCL's if then else endif

Definition $OclIf :: ([\'A]'\tau. Boolean \ , \ ('\alpha',\'\alpha::null)\ val\ , \ ('\alpha',\'\alpha)\ val] \Rightarrow ('\alpha',\'\alpha)\ val$

if (-) then (-) else (-) endif $[10,10,10][50]$

Where (if C then B$_1$ else B$_2$ endif) = ($\lambda\ \tau.\ \{\ \begin{array}{ll}
\text{if } (\delta C) \ \tau = \text{true} \\
\text{then } B_1 \ \tau \\
\text{else } B_2 \ \tau
\end{array}\}$)

Lemma $cp-OclIf$:

((if C then B$_1$ else B$_2$ endif) $\tau =$

(if ($\lambda\ \tau.\ C \ \tau$) then ($\lambda\ \tau.\ B_1 \ \tau$) else ($\lambda\ \tau.\ B_2 \ \tau$) endif) $\tau$)

by (simp only: OclIf-def, subst cp-defined, rule refl)

Lemmas $cp-intro$[$intro!,simp,code-unfold$] =

$\quad cp-intro$

$\quad cp-OclIf[THEN allI[THEN allI[THEN allI[THEN allI[THEN cpI?]], of OclIf]]]$

Lemma $OclIf-invalid$ [simp]: (if invalid then B$_1$ else B$_2$ endif) = invalid
by (rule ext, auto simp: OclIf-def)

lemma OclIf-null [simp]: (if null then $B_1$ else $B_2$ endif) = invalid
by (rule ext, auto simp: OclIf-def)

lemma OclIf-true [simp]: (if true then $B_1$ else $B_2$ endif) = $B_1$
by (rule ext, auto simp: OclIf-def)

lemma OclIf-true¹ [simp]: $\tau \models \tau \models (\text{if } P \text{ then } B_1 \text{ else } B_2 \text{ endif}) \tau = B_1 \tau$
apply (subst cp-OclIf, auto simp: OclValid-def)
by (simp add: cp-OclIf[symmetric])

lemma OclIf-false [simp]: (if false then $B_1$ else $B_2$ endif) = $B_2$
by (rule ext, auto simp: OclIf-def)

lemma OclIf-false¹ [simp]: $\tau \models \tau \models (\text{if } P \text{ then } B_1 \text{ else } B_2 \text{ endif}) \tau = B_2 \tau$
apply (subst cp-OclIf)
apply (auto simp: foundation14[symmetric] foundation22)
by (auto simp: cp-OclIf[symmetric])

lemma OclIf-idem1[simp]: (if $\delta X$ then $A$ else $A$ endif) = $A$
by (rule ext, auto simp: OclIf-def)

lemma OclIf-idem2[simp]: (if $\upsilon X$ then $A$ else $A$ endif) = $A$
by (rule ext, auto simp: OclIf-def)

lemma OclNot-if [simp]: not (if $P$ then $C$ else $E$ endif) = (if $P$ then not $C$ else not $E$ endif)
apply (rule OclNot-inject, simp)
apply (rule ext)
apply (subst cp-OclNot, simp add: OclIf-def)
apply (subst cp-OclNot[symmetric])
by simp

3.6.2. A Side-calculus for (Boolean) Constant Terms

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

lemma const-charn: $\text{const } X \implies X \tau = X \tau'$
by (auto simp: const-def)

lemma const-subst:
assumes const-X: const $X$
and const-Y: const $Y$
and eq : $X \tau = Y \tau$
and cp-P: $cp P$
and pp : $P Y \tau = P Y \tau$

by (rule ext, auto simp: OclIf-def)
shows $P \ X \ \tau = P \ X \ \tau'$

proof –

have $A$: $\forall Y. \ P Y \ \tau = P (\lambda Y \ \tau) \ \tau$
  apply (insert cp-P, unfold cp-def)
  apply (elim e, erule-tac $x=Y$ in allE', erule-tac $x=\tau$ in allE)
  apply (erule-tac $x=(\lambda Y \ \tau) \ \tau$ in allE)
  by simp

have $B$: $\forall Y. \ P Y \ \tau' = P (\lambda Y \ \tau') \ \tau'$
  apply (insert cp-P, unfold cp-def)
  apply (elim e, erule-tac $x=Y$ in allE', erule-tac $x=\tau'$ in allE)
  apply (erule-tac $x=(\lambda Y \ \tau) \ \tau'$ in allE)
  by simp

have $C$: $X \ \tau' = Y \ \tau'$
  apply (rule trans, subst const-charn[OF const-X], rule eq)
  by (rule const-charn[OF const-Y])

show ?thesis
  apply (subst $A$, subst $B$, simp add: eq $C$)
  apply (subst $A$[symmetric], subst $B$[symmetric])
  by (simp add: pp)

qed

lemma const-imply2:
  assumes $\forall \tau. \ \tau \ \tau' = P \ \tau \ \tau' \Longrightarrow Q \ \tau = Q \ \tau$
  shows $\Longrightarrow const Q$
  by (simp add: const-def, insert assms, blast)

lemma const-imply3:
  assumes $\forall \tau. \ \tau \ \tau' = P \ \tau \ \tau' \Longrightarrow Q \ \tau = Q \ \tau \Longrightarrow R \ \tau = R \ \tau$
  shows $\Longrightarrow const Q \Longrightarrow const R$
  by (simp add: const-def, insert assms, blast)

lemma const-imply4:
  assumes $\forall \tau. \ \tau \ \tau' = P \ \tau \ \tau' \Longrightarrow Q \ \tau = Q \ \tau \Longrightarrow R \ \tau = R \ \tau \Longrightarrow S \ \tau = S \ \tau$
  shows $\Longrightarrow const Q \Longrightarrow const R \Longrightarrow const S$
  by (simp add: const-def, insert assms, blast)

lemma const-lam: $const \ (\lambda e)$
  by (simp add: const-def)

lemma const-true: $const \ true$
  by (simp add: const-def true-def)

lemma const-false: $const \ false$
  by (simp add: const-def false-def)

lemma const-null: $const \ null$
  by (simp add: const-def null-fun-def)


lemma const-invalid : const invalid
by(simp add: const-def invalid-def)

lemma const-bot : const bot
by(simp add: const-def bot-fun-def)

lemma const-defined :
  assumes const X
  shows const (∆ X)
by(rule const-imply2[OF - assms],
   simp add: defined-def false-def true-def bot-fun-def bot-option-def null-fun-def null-option-def)

lemma const-valid :
  assumes const X
  shows const (∆ X)
by(rule const-imply2[OF - assms],
   simp add: valid-def false-def true-def bot-fun-def null-fun-def null-option-def assms)

lemma const-OclValid1:
  assumes const x
  shows (τ |= ∆ x) = (τ′ |= ∆ x)
apply(simp add: OclValid-def)
apply(subst const-defined[OF assms, THEN const-charn])
by(simp add: true-def)

lemma const-OclValid2:
  assumes const x
  shows (τ |= v x) = (τ′ |= v x)
apply(simp add: OclValid-def)
apply(subst const-valid[OF assms, THEN const-charn])
by(simp add: true-def)

lemma const-OclAnd :
  assumes const X
  assumes const X'
  shows const (X and X')
by(rule const-imply3[OF - assms], subst (1 2) cp-OclAnd, simp add: assms OclAnd-def)

lemma const-OclNot :
  assumes const X
  shows const (not X)
by(rule const-imply2[OF - assms], subst cp-OclNot, simp add: assms OclNot-def)
lemma const-OclOr :  
assumes const X  
assumes const X'  
shows const (X or X')  
by(simp add: assms OclOr-def const-OclNot const-OclAnd)

lemma const-OclImplies :  
assumes const X  
assumes const X'  
shows const (X implies X')  
by(simp add: assms OclImplies-def const-OclNot const-OclOr)

lemma const-StrongEq:  
assumes const X  
assumes const X'  
shows const (X ≜ X')  
apply(simp only: StrongEq-def const-def, intro allI)
apply(subst assms(1)[THEN const-charn])
apply(subst assms(2)[THEN const-charn])
by simp

lemma const-OclIf :  
assumes const B  
and const C1  
and const C2  
shows const (if B then C1 else C2 endif)
apply(rule const-imply4[OF - assms],  
subst (1 2) cp-OclIf, simp only: OclIf-def cp-defined[symmetric])
apply(simp add: const-defined[OF assms(1), simplified const-def, THEN spec, THEN spec]
const-true[simplified const-def, THEN spec, THEN spec]
assms[simplified const-def, THEN spec, THEN spec]
const-invalid[simplified const-def, THEN spec, THEN spec])
by (metis (no-types) OCL-core.bot-fun-def OclValid-def const-true defined-def foundation17
null-fun-def)

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
end
4. Formalization II: Library Definitions

theory OCL-lib
imports OCL-core
begin

The structure of this chapter roughly follows the structure of Chapter 10 of the OCL standard [33], which introduces the OCL Library.

4.1. Basic Types: Void and Integer

4.1.1. The Construction of the Void Type

type-synonym $(\forall A)\text{Void} = (\forall A, \text{unit option})\ val$

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.

4.1.2. The Construction of the Integer Type

Since Integer is again a basic type, we define its semantic domain as the valuations over int option option.

type-synonym $(\forall A)\text{Integer} = (\forall A, \text{int option option})\ val$

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

definition OclInt0 ::$(\forall A)\text{Integer} (0)$
where 0 = ($\lambda \cdot \cdot \lfloor 0 :: \text{int} \rfloor$)

definition OclInt1 ::$(\forall A)\text{Integer} (1)$
where 1 = ($\lambda \cdot \cdot \lfloor 1 :: \text{int} \rfloor$)

definition OclInt2 ::$(\forall A)\text{Integer} (2)$
where 2 = ($\lambda \cdot \cdot \lfloor 2 :: \text{int} \rfloor$)

definition OclInt3 ::$(\forall A)\text{Integer} (3)$
where 3 = ($\lambda \cdot \cdot \lfloor 3 :: \text{int} \rfloor$)

definition OclInt4 ::$(\forall A)\text{Integer} (4)$
where 4 = ($\lambda \cdot \cdot \lfloor 4 :: \text{int} \rfloor$)
definition \textit{OclInt5} :: (\forall)\textit{Integer} (5)
where \hspace{1em} 5 = (\lambda - . \lfloor \lfloor 5 :: \text{int} \rfloor \rfloor)

definition \textit{OclInt6} :: (\forall)\textit{Integer} (6)
where \hspace{1em} 6 = (\lambda - . \lfloor \lfloor 6 :: \text{int} \rfloor \rfloor)

definition \textit{OclInt7} :: (\forall)\textit{Integer} (7)
where \hspace{1em} 7 = (\lambda - . \lfloor \lfloor 7 :: \text{int} \rfloor \rfloor)

definition \textit{OclInt8} :: (\forall)\textit{Integer} (8)
where \hspace{1em} 8 = (\lambda - . \lfloor \lfloor 8 :: \text{int} \rfloor \rfloor)

definition \textit{OclInt9} :: (\forall)\textit{Integer} (9)
where \hspace{1em} 9 = (\lambda - . \lfloor \lfloor 9 :: \text{int} \rfloor \rfloor)

definition \textit{OclInt10} :: (\forall)\textit{Integer} (10)
where \hspace{1em} 10 = (\lambda - . \lfloor \lfloor 10 :: \text{int} \rfloor \rfloor)

#### 4.1.3. Validity and Definedness Properties

lemma \(\delta (\text{null} :: (\forall)\textit{Integer}) = \text{false}\) by simp
lemma \(\nu (\text{null} :: (\forall)\textit{Integer}) = \text{true}\) by simp

lemma [simp, code-unfold]: \(\delta (\lambda - . \lfloor \lfloor n \rfloor \rfloor) = \text{true}\)
by (simp add: defined-def true-def

bot-fun-def bot-option-def null-fun-def null-option-def)

lemma [simp, code-unfold]: \(\nu (\lambda - . \lfloor \lfloor n \rfloor \rfloor) = \text{true}\)
by (simp add: valid-def true-def

bot-fun-def bot-option-def)

lemma [simp, code-unfold]: \(\delta 0 = \text{true}\) by (simp add: OclInt0-def)
lemma [simp, code-unfold]: \(\nu 0 = \text{true}\) by (simp add: OclInt0-def)
lemma [simp, code-unfold]: \(\delta 1 = \text{true}\) by (simp add: OclInt1-def)
lemma [simp, code-unfold]: \(\nu 1 = \text{true}\) by (simp add: OclInt1-def)
lemma [simp, code-unfold]: \(\delta 2 = \text{true}\) by (simp add: OclInt2-def)
lemma [simp, code-unfold]: \(\nu 2 = \text{true}\) by (simp add: OclInt2-def)
lemma [simp, code-unfold]: \(\delta 6 = \text{true}\) by (simp add: OclInt6-def)
lemma [simp, code-unfold]: \(\nu 6 = \text{true}\) by (simp add: OclInt6-def)
lemma [simp, code-unfold]: \(\delta 8 = \text{true}\) by (simp add: OclInt8-def)
lemma [simp, code-unfold]: \(\nu 8 = \text{true}\) by (simp add: OclInt8-def)
lemma [simp, code-unfold]: \(\delta 9 = \text{true}\) by (simp add: OclInt9-def)
lemma [simp, code-unfold]: \(\nu 9 = \text{true}\) by (simp add: OclInt9-def)
4.1.4. Arithmetical Operations on Integer

Definition

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

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

definition OclAdd_integer :: (\texttt{\textit{\texttt{A}}} Integer ⇒ (\texttt{\textit{\texttt{A}}} Integer ⇒ (\texttt{\textit{\texttt{A}}} Integer ⇒ Integer)) (infix ‘+’ 40)
where \( x \uplus y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \land (\delta y) \tau = \text{true } \tau \text{ then } \lceil \lceil x \tau \rceil \rceil \ + \lceil y \tau \rceil \rceil \text{ else invalid } \tau \)

definition OclLess_integer :: (\texttt{\textit{\texttt{A}}} Integer ⇒ (\texttt{\textit{\texttt{A}}} Integer ⇒ (\texttt{\textit{\texttt{A}}} Boolean ⇒ Boolean)) (infix ‘<’ 40)
where \( x \prec y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \land (\delta y) \tau = \text{true } \tau \text{ then } \lceil \lceil x \tau \rceil \rceil \text{<} \lceil y \tau \rceil \rceil \text{ else invalid } \tau \)

definition OclLe_integer :: (\texttt{\textit{\texttt{A}}} Integer ⇒ (\texttt{\textit{\texttt{A}}} Integer ⇒ (\texttt{\textit{\texttt{A}}} Boolean ⇒ Boolean)) (infix ‘\le’ 40)
where \( x \leq y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \land (\delta y) \tau = \text{true } \tau \text{ then } \lceil \lceil x \tau \rceil \rceil \leq \lceil y \tau \rceil \rceil \text{ else invalid } \tau \)

Basic Properties

lemma OclAdd_integer-commute: \( (X \uplus Y) = (Y \uplus X) \)
by (rule ext, auto simp:true-def false-def OclAdd_integer-def invalid-def
  split: option.split option.split-asm
  bool.split bool.split-asm)

Execution with Invalid or Null or Zero as Argument

lemma OclAdd_integer-strict1[simp,code-unfold]: \( (x \uplus \text{invalid}) = \text{invalid} \)
by (rule ext, simp add: OclAdd_integer-def true-def false-def)

lemma OclAdd_integer-strict2[simp,code-unfold]: \( (\text{invalid} \uplus x) = \text{invalid} \)
by (rule ext, simp add: OclAdd_integer-def true-def false-def)

lemma [simp,code-unfold]: \( (x \uplus \text{null}) = \text{invalid} \)
by (rule ext, simp add: OclAdd_integer-def true-def false-def)

lemma [simp,code-unfold]: \( (\text{null} \uplus x) = \text{invalid} \)
by (rule ext, simp add: OclAdd_integer-def true-def false-def)

lemma OclAdd_integer-zero1[simp,code-unfold]:
\( (x \uplus 0) = (\text{if } v x \text{ and not } (\delta x) \text{ then invalid else } x \text{ endif}) \)
proof (rule ext, rename-tac \( \tau \), case-tac \( v x \text{ and not } (\delta x) \) \( \tau = \text{true } \tau \))
  fix \( \tau \) show \( (v x \text{ and not } (\delta x)) \tau = \text{true } \tau \Rightarrow \)

67
(x + 0) \tau = (if \ v \ x \ and \ not \ (\delta \ x) \ then \ invalid \ else \ x \ endif) \tau

apply(subst OclIf-true', simp add: OclValid-def)
by (metis OclAddInteger-def OclNot-defargs OclValid-def foundation5 foundation9)
apply-end assumption

next fix \tau
have A: \{\tau. (\tau \models (v \ x \ and \ not \ (\delta \ x))) = (x \tau = invalid \tau \lor \tau \models \delta \ x)\}
by (metis OclNot-not OclOr-def defined5 defined6 defined-not-I foundation11 foundation18'
foundation6 foundation7 foundation9 invalid-def)

have B: \tau \models \delta \ x \Rightarrow \lfloor\lfloor\lfloor x \tau \rfloor\rfloor\rfloor \tau = x \tau
by(simp)

show \tau \models (v \ x \ and \ not \ (\delta \ x)) \Rightarrow
(x + 0) \tau = (if \ v \ x \ and \ not \ (\delta \ x) \ then \ invalid \ else \ x \ endif) \tau
apply(subst OclIf-false', simp, simp add: A, auto simp: OclAddInteger-def OclInt0-def)

apply (metis OclValid-def foundation19 foundation20)
apply (simp add: B)
by(simp add: OclValid-def)
apply-end(metis OclValid-def defined5 defined6 defined-and-I defined-not-I foundation9)

qed

lemma OclAddInteger-zero2[simp, code-unfold]:
(0 + x) = (if \ v \ x \ and \ not \ (\delta \ x) \ then \ invalid \ else \ x \ endif)
by(subst OclAddInteger-commute, simp)

Context Passing

lemma cp-OclAddInteger:(X + Y) \tau = ((\lambda \ - \ X \tau) \ + \ (\lambda \ - \ Y \tau)) \tau
by(simp add: OclAddInteger-def cp-defined[symmetric])

lemma cp-OclLessInteger:(X < Y) \tau = ((\lambda \ - \ X \tau) \ < \ (\lambda \ - \ Y \tau)) \tau
by(simp add: OclLessInteger-def cp-defined[symmetric])

lemma cp-OclLeInteger:(X \leq Y) \tau = ((\lambda \ - \ X \tau) \ \leq \ (\lambda \ - \ Y \tau)) \tau
by(simp add: OclLeInteger-def cp-defined[symmetric])

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.

value \tau \models (9 \leq 10)
value \tau \models ((4 + 4) \leq 10)
value \neg(\tau \models ((4 + (4 + 4)) < 10))
value \tau \models not (v (null \ + 1))
4.2. Fundamental Predicates on Basic Types: Strict Equality

4.2.1. 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 `A Boolean`-case as strict extension of the strong equality:

```plaintext
defs (overloaded) StrictRefEqInteger[code-unfold] :  
  (x::('A)Integer) ≠ y ≡ λ τ. if (υ x) τ = true τ ∧ (υ y) τ = true τ  
  then (x ≜ y) τ  
  else invalid τ
```

value τ ⊨ 1 <> 2
value τ ⊨ 2 <> 1
value τ ⊨ 2 ≠ 2

4.2.2. Logic and Algebraic Layer on Basic Types

Validity and Definedness Properties (I)

lemma StrictRefEqBoolean-defined-args-valid:
(τ ⊨ δ((x::('A)Boolean) ≠ y)) = (τ ⊨(v x)) ∧ (τ ⊨(v y)))
by(auto simp: StrictRefEqBoolean OclValid-def true-def valid-def false-def StrongEq-def 
      defined-def invalid-def null-fun-def bot-fun-def null-option-def bot-option-def 
      split: bool.split-asm HOL.split-if-asm option.split)

lemma StrictRefEqInteger-defined-args-valid:
(τ ⊨ δ((x::('A)Integer) ≠ y)) = (τ ⊨(v x)) ∧ (τ ⊨(v y)))
by(auto simp: StrictRefEqInteger OclValid-def true-def valid-def false-def StrongEq-def 
      defined-def invalid-def null-fun-def bot-fun-def null-option-def bot-option-def 
      split: bool.split-asm HOL.split-if-asm option.split)

Validity and Definedness Properties (II)

lemma StrictRefEqBoolean-defargs:
τ ⊨ ((x::('A)Boolean) ≠ y) ⇒ (τ ⊨(v x)) ∧ (τ ⊨(v y))
by(simp add: StrictRefEqBoolean OclValid-def true-def invalid-def 
      bot-option-def 
      split: bool.split-asm HOL.split-if-asm)

lemma StrictRefEqInteger-defargs:
τ ⊨ ((x::('A)Integer) ≠ y) ⇒ (τ ⊨(v x)) ∧ (τ ⊨(v y))
by(simp add: StrictRefEqInteger OclValid-def true-def invalid-def valid-def bot-option-def 
      split: bool.split-asm HOL.split-if-asm)

Validity and Definedness Properties (III) Miscellaneous

lemma StrictRefEqBoolean-strict'' : δ ((x::('A)Boolean) ≠ y) = (v(x) and v(y))
by(auto intro!: transform2-rev defined-1 simp:foundation10 
      StrictRefEqBoolean-defined-args-valid)
lemma StrictRefEq\_Integer\_strict'' : \( \delta ((x :: (\mathbb{A}) \text{Integer}) \nvdash y) = (\nu(x) \text{ and } \nu(y)) \)
by(auto intro! : transform2-rev defined-and-I simp foundation10 StrictRefEq\_Integer\_defined-args-valid)

lemma StrictRefEq\_Integer\_strict'' :
assumes A : \( \nu (x :: (\mathbb{A}) \text{Integer}) = true \)
and B : \( \nu y = true \)
shows \( \nu (x \nvdash y) = true \)
apply(insert A B)
apply(rule ext, simp add: StrongEq-def StrictRefEq\_Integer true-def valid-def defined-def bot-fun-def bot-option-def)
done

lemma StrictRefEq\_Integer\_strict'' :
assumes A : \( \nu ((x :: (\mathbb{A}) \text{Integer})) \nvdash y) = true \)
shows \( \nu x = true \land \nu y = true \)
apply(insert A, rule conjI)
apply(rule ext, rename-tac \( \tau \), drule-tac x \( \equiv \tau \) in fun-cong)
pref 2
apply(rule ext, rename-tac \( \tau \), drule-tac x \( \equiv \tau \) in fun-cong)
apply(simp-all add: StrongEq-def StrictRefEq\_Integer false-def true-def valid-def defined-def)
apply(case-tac y \( \tau \), auto)
apply(simp-all add: true-def invalid-def bot-fun-def)
done

Reflexivity

lemma StrictRefEq\_Boolean\_refl[simp, code-unfold] :
(\( (x :: (\mathbb{A}) \text{Boolean}) \nvdash x) = (\text{if } \nu x \text{ then true else invalid endif} \))
by(rule ext, simp add: StrictRefEq\_Boolean OclIf-def)

lemma StrictRefEq\_Integer\_refl[simp, code-unfold] :
(\( (x :: (\mathbb{A}) \text{Integer}) \nvdash x) = (\text{if } \nu x \text{ then true else invalid endif} \))
by(rule ext, simp add: StrictRefEq\_Integer OclIf-def)

Execution with Invalid or Null as Argument

lemma StrictRefEq\_Boolean\_strict1[simp, code-unfold] : ((x :: (\mathbb{A}) \text{Boolean}) \nvdash \text{invalid}) = \text{invalid}
by(rule ext, simp add: StrictRefEq\_Boolean true-def false-def)

lemma StrictRefEq\_Boolean\_strict2[simp, code-unfold] : (invalid \( \nvdash (x :: (\mathbb{A}) \text{Boolean}) \)) = invalid
by(rule ext, simp add: StrictRefEq\_Boolean true-def false-def)

lemma StrictRefEq\_Integer\_strict1[simp, code-unfold] : ((x :: (\mathbb{A}) \text{Integer}) \nvdash \text{invalid}) = \text{invalid}
by(rule ext, simp add: StrictRefEq\_Integer true-def false-def)

70
lemma StrictRefEq\_Integer\_strict2[simp, code-unfold] : (invalid \equiv (x::\texttt{\langle\mathbb{A}\rangle Integer})) = invalid
by (rule ext, simp add: StrictRefEq\_Integer true-def false-def)

lemma integer\_non\_null [simp]: ((\lambda n\cdot \lfloor n \rfloor) \equiv (null::(\texttt{\langle\mathbb{A}\rangle Integer})) = false
by (rule ext, auto simp: StrictRefEq\_Integer valid-def
\quad bot-fun-def bot-option-def null-fun-def null-option-def StrictEq-def)

lemma null\_non\_integer [simp]: ((null::(\texttt{\langle\mathbb{A}\rangle Integer}) \equiv (\lambda n\cdot \lfloor n \rfloor)) = false
by (rule ext, auto simp: StrictRefEq\_Integer valid-def
\quad bot-fun-def bot-option-def null-fun-def null-option-def StrictEq-def)

lemma OclInt0\_non\_null [simp, code-unfold]: (0 \equiv null) = false by (simp add: OclInt0-def)
lemma null\_non\_OclInt0 [simp, code-unfold]: (null \equiv 0) = false by (simp add: OclInt0-def)
lemma OclInt1\_non\_null [simp, code-unfold]: (1 \equiv null) = false by (simp add: OclInt1-def)
lemma null\_non\_OclInt1 [simp, code-unfold]: (null \equiv 1) = false by (simp add: OclInt1-def)
lemma OclInt2\_non\_null [simp, code-unfold]: (2 \equiv null) = false by (simp add: OclInt2-def)
lemma null\_non\_OclInt2 [simp, code-unfold]: (null \equiv 2) = false by (simp add: OclInt2-def)
lemma OclInt6\_non\_null [simp, code-unfold]: (6 \equiv null) = false by (simp add: OclInt6-def)
lemma null\_non\_OclInt6 [simp, code-unfold]: (null \equiv 6) = false by (simp add: OclInt6-def)
lemma OclInt8\_non\_null [simp, code-unfold]: (8 \equiv null) = false by (simp add: OclInt8-def)
lemma null\_non\_OclInt8 [simp, code-unfold]: (null \equiv 8) = false by (simp add: OclInt8-def)
lemma OclInt9\_non\_null [simp, code-unfold]: (9 \equiv null) = false by (simp add: OclInt9-def)
lemma null\_non\_OclInt9 [simp, code-unfold]: (null \equiv 9) = false by (simp add: OclInt9-def)

Const

lemma [simp, code-unfold]: const(0) by (simp add: const-ss OclInt0-def)
lemma [simp, code-unfold]: const(1) by (simp add: const-ss OclInt1-def)
lemma [simp, code-unfold]: const(2) by (simp add: const-ss OclInt2-def)
lemma [simp, code-unfold]: const(6) by (simp add: const-ss OclInt6-def)
lemma [simp, code-unfold]: const(8) by (simp add: const-ss OclInt8-def)
lemma [simp, code-unfold]: const(9) by (simp add: const-ss OclInt9-def)

Behavior vs StrictEq

lemma StrictRefEq\_Boolean\_vs\_StrongEq:
\quad \tau \models (v \; x) \implies \tau \models (v \; y) \implies (\tau \models ((x::\texttt{\langle\mathbb{A}\rangle Boolean}) \equiv y) \equiv (x \equiv y))
apply (simp add: StrictRefEq\_Boolean OclValid-def)
apply (subst cp-StrongEq[of - (x \equiv y)])
by simp

lemma StrictRefEq\_Integer\_vs\_StrongEq:
\quad \tau \models (v \; x) \implies \tau \models (v \; y) \implies (\tau \models ((x::\texttt{\langle\mathbb{A}\rangle Integer}) \equiv y) \equiv (x \equiv y))
apply (simp add: StrictRefEq\_Integer OclValid-def)
apply (subst cp-StrongEq[of - (x \equiv y)])
by simp
Context Passing

lemma cp-StrictRefEqBoolean:
\[(X::(\exists)\text{Boolean}) \equiv Y) \tau = ((\lambda \cdot. X \tau) \equiv (\lambda \cdot. Y \tau)) \tau\]
by(auto simp: StrictRefEqBoolean StrongEq-def defined-def valid-def cp-defined[ symmetric])

lemma cp-StrictRefEqInteger:
\[(X::(\exists)\text{Integer}) \equiv Y) \tau = ((\lambda \cdot. X \tau) \equiv (\lambda \cdot. Y \tau)) \tau\]
by(auto simp: StrictRefEqInteger StrongEq-def valid-def cp-defined[ symmetric])

lemmas cp-intro[ intro, simp, code-unfold] =
cp-intro'
  cp-StrictRefEqBoolean[ THEN allI THEN allI THEN allI THEN allI THEN cpI2], of StrictRefEq
  cp-StrictRefEqInteger[ THEN allI THEN allI THEN allI THEN allI THEN cpI2], of StrictRefEq
  cp-OclAddInteger[ THEN allI THEN allI THEN allI THEN allI THEN cpI2], of OclAddInteger
  cp-OclLessInteger[ THEN allI THEN allI THEN allI THEN allI THEN cpI2], of OclLessInteger
  cp-OclLeInteger[ THEN allI THEN allI THEN allI THEN allI THEN cpI2], of OclLeInteger]

4.2.3. Test Statements on Basic Types.

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 Booleans

value \(\tau\) \equiv \nu(\text{true})
value \(\tau\) \equiv \delta(\text{false})
value \(\neg(\tau\equiv\delta(\text{null}))\)
value \(\neg(\tau\equiv\delta(\text{invalid}))\)
value \(\tau\) \equiv \nu((\text{null}::(\exists)\text{Boolean}))
value \(\neg(\tau\equiv\nu(\text{invalid}))\)
value \(\tau\) \equiv (\text{true and true} \equiv \text{true})
value \(\tau\) \equiv ((\text{null or null}) \equiv \text{null})
value \(\tau\) \equiv ((\text{null or null}) \equiv \text{null})
value \(\tau\) \equiv ((\text{true} \equiv \text{false}) \equiv \text{false})
value \(\tau\) \equiv ((\text{invalid} \equiv \text{false}) \equiv \text{false})
value \(\tau\) \equiv ((\text{invalid} \equiv \text{false}) \equiv \text{invalid})

Elementary computations on Integer

value \(\tau\) \equiv \nu \ 4
value \(\tau\) \equiv \delta \ 4
value \(\tau\) \equiv \nu (\text{null}::(\exists)\text{Integer})
value \(\tau\) \equiv (\text{invalid} \equiv \text{invalid})
value \(\tau\) \equiv (\text{null} \equiv \text{null})
value \(\tau\) \equiv (4 \equiv 4)
value \(\neg(\tau\equiv(9 \equiv 10))\)
value \(\neg(\tau\equiv(\text{invalid} \equiv 10))\)
value \(\neg(\tau\equiv(\text{null} \equiv 10))\)
value \(\neg(\tau\equiv(\text{invalid} \equiv (\text{invalid}::(\exists)\text{Integer})))\)
4.3. Complex Types: The Set-Collection Type (I) Core

4.3.1. The Construction of the Set Type

no-notation None (⊥)
notation bot (⊥)

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(Set(Sequences(Pairs(X,Y)))))).

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

The core of an own type construction is done via a type definition which provides the raw-type \( \alpha \) Set-0. It is shown that this type “fits” indeed into the abstract type interface discussed in the previous section.

typedef \( \alpha \) Set-0 = \{X:('\alpha::null) set option option.
  X = bot \lor X = null \lor (\forall x\in[X]. x \neq bot)\}
  by (rule-tac x=bot in exI, simp)

instantiation Set-0 :: (null)bot
begin

definition bot-Set-0-def: (bot::('a::null) Set-0) \equiv Abs-Set-0 None

instance proof show \( \exists x::'a \) Set-0. \( x \neq bot \)
  apply(rule-tac x=Abs-Set-0 [None] in exI)
apply (simp add bot-Set-0-def)
apply (subst Abs-Set-0-inject)
apply (simp-all add bot-Set-0-def
       null-option-def bot-option-def)
done
qed

end

instantiation Set-0 :: (null)null
begin

  definition null-Set-0-def: (null::('a::null) Set-0) ≡ Abs-Set-0 ⌊None⌋
instance proof show (null::('a::null) Set-0) ≠ bot
  apply (simp add null-Set-0-def bot-Set-0-def)
  apply (subst Abs-Set-0-inject)
  apply (simp-all add bot-Set-0-def
         null-option-def bot-option-def)
done
qed

end

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

  type-synonym ('A,'a) Set = ('A,'a Set-0) val

4.3.2. Validity and Definedness Properties

Every element in a defined set is valid.

  lemma Set-inv-lemma: τ |= (δ X) ⇒ ∀x∀τ[Rep-Set-0 (X τ)]. x ≠ bot
apply (insert Rep-Set-0 [of X τ], simp)
apply (auto simp: OclValid-def defined-def false-def true-def cp-def
       bot-fun-def bot-Set-0-def null-Set-0-def null-fun-def
       split:split-if-asm)
apply (erule contrapos-pp [of Rep-Set-0 (X τ) = bot])
apply (subst Abs-Set-0-inject[symmetric], rule Rep-Set-0, simp)
apply (simp add: Rep-Set-0-inverse bot-Set-0-def bot-option-def)
apply (erule contrapos-pp [of Rep-Set-0 (X τ) = null])
apply (subst Abs-Set-0-inject[symmetric], rule Rep-Set-0, simp)
apply (simp add: Rep-Set-0-inverse null-option-def)
by (simp add: bot-option-def)

  lemma Set-inv-lemma':
  assumes x-def: τ |= δ X
      and e-mem: e ∈ [[Rep-Set-0 (X τ)]]
  shows τ |= v (λ_. e)
apply (rule Set-inv-lemma[OF x-def, THEN ballE[where x = e]])
apply (simp add: foundation18)
by (simp add: e-mem)
lemma abs-rep-simp':
assumes S-all-def : \tau \vdash \delta S
shows Abs-Set-0 \[[\text{Rep-Set-0} (S \tau)]] = S \tau
proof
  have discr-eq-false-true : \forall \tau. (false \tau = true \tau) = False 
by (simp add: false-def true-def)
  show \?thesis
  apply (insert S-all-def, simp add: OclValid-def defined-def)
  apply (rule mp[OF Abs-Set-0-induct[where P = \lambda S. (if S = \bot \tau \lor S = \text{null} \tau 
then false \tau else true \tau) = true \tau \rightarrow
Abs-Set-0 \[[[\text{Rep-Set-0} S]]]] = S]],
    rename-tac S')
  apply (simp add: Abs-Set-0-inverse discr-eq-false-true)
  apply (case-tac S') apply (simp add: bot-fun-def bot-Set-0-def)
   apply (simp add: Abs-Set-0-inverse discr-eq-false-true)
  apply (case-tac S') apply (simp add: null-fun-def null-Set-0-def)
  done
qed

lemma S-lift' :
assumes S-all-def : (\tau :: 'A st) \vdash \delta S
shows \exists S'. (\lambda a :: 'A st. a) \cdot \[[\text{Rep-Set-0} (S \tau)]] = (\lambda a :: 'A st. [a]) \cdot S'
apply (rule-tac x = (\lambda a. [a]) \cdot [[\text{Rep-Set-0} (S \tau)]] in exf)
apply (simp only: image-comp)
apply (simp add: comp-def)
apply (rule image-cong, fast)
apply (erule Set-inv-lemma[OF S-all-def])
by (case-tac x, simp add: bot-option-def foundation18+)

lemma invalid-set-OclNot-defined [simp, code-unfold]:
\delta(\text{invalid} :: ('A, 'a :: null) Set) = false 
by simp
lemma null-set-OclNot-defined [simp, code-unfold]:
\delta(\text{null} :: ('A, 'a :: null) Set) = false
by (simp add: defined-def null-fun-def)
lemma invalid-set-valid [simp, code-unfold]:
\nu(\text{invalid} :: ('A, 'a :: null) Set) = false
by simp
lemma null-set-valid [simp, code-unfold]:
\nu(\text{null} :: ('A, 'a :: null) Set) = true
apply (simp add: valid-def null-fun-def bot-fun-def bot-Set-0-def null-Set-0-def)
apply (subst Abs-Set-0-inject, simp-all add: null-option-def bot-option-def)
done

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

4.3.3. Constants on Sets
definition mtSet::('A, 'a :: null) Set {Set{}}
where \( \text{Set}\{\} \equiv (\lambda \tau. \text{Abs-Set-0} [\{\}::\alpha \text{ set}]) \)

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

**Apply** (rule ext, auto simp: \(\text{mtSet-def} \text{ defined-def} \text{ null-Set-0-def} \text{ bot-Set-0-def} \text{ bot-fun-def} \text{ null-fun-def} \))

**By** (simp-all add: \(\text{Abs-Set-0-inject} \text{ bot-option-def} \text{ null-Set-0-def} \text{ null-option-def} \))

**Lemma** \(\text{mtSet-valid}[\text{simp, code-unfold}]: \nu(\text{Set}\{\}) = \text{true} \)

**Apply** (rule ext, auto simp: \(\text{mtSet-def} \text{ valid-def} \text{ null-Set-0-def} \text{ bot-Set-0-def} \text{ bot-fun-def} \text{ null-fun-def} \))

**By** (simp-all add: \(\text{Abs-Set-0-inject} \text{ bot-option-def} \text{ null-Set-0-def} \text{ null-option-def} \))

**Lemma** \(\text{mtSet-rep-set}: \lceil \lceil \text{Rep-Set-0} (\text{Set}\{\}) \rceil \rceil = \{\} \)

**Apply** (simp add: \(\text{mtSet-def} \text{ subst Abs-Set-0-inverse} \))

**By** (simp add: \(\text{bot-option-def} \))

**Lemma** \([\text{simp, code-unfold}]: \text{const Set}\{\} \)

**By** (simp add: \(\text{const-def} \text{ mtSet-def} \))

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

### 4.4. Complex Types: The Set-Collection Type (II) Library

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

#### 4.4.1. Computational Operations on Set

**Definition**

**Definition** \(\text{OclIncluding} :: [(\mathcal{A},\alpha::null) \text{ Set},(\mathcal{A},\alpha) \text{ val}] \rightarrow (\mathcal{A},\alpha) \text{ Set} \)

**Where** \(\text{OclIncluding} \; x \; y = (\lambda \tau. \text{ if } (\delta \; x) \; \tau = \text{true} \; \tau \land (\nu \; y) \; \tau = \text{true} \; \tau \)

\hspace{1cm} \text{then } \text{Abs-Set-0} [\lceil \lceil \text{Rep-Set-0} (\; x \; \tau) \rceil \rceil \cup \{ y \; \tau \} \rceil \rceil \)

\hspace{1cm} \text{else } \bot \)

**Notation** \(\text{OclIncluding} \; (-\rightarrow \text{including}'(-') \)

**Syntax**

\(-\text{OclFinset} :: \text{ args => } (\mathcal{A},\alpha::null) \text{ Set} \; (\text{Set}\{\})\)

**Translations**

\(\text{Set}\{x, \; xs\} \equiv \text{CONST} \; \text{OclIncluding} \; (\text{Set}\{xs\}) \; x \)

\(\text{Set}\{x\} \equiv \text{CONST} \; \text{OclIncluding} \; (\text{Set}\{\}) \; x \)

**Definition** \(\text{OclExcluding} :: [(\mathcal{A},\alpha::null) \text{ Set},(\mathcal{A},\alpha) \text{ val}] \rightarrow (\mathcal{A},\alpha) \text{ Set} \)

**Where** \(\text{OclExcluding} \; x \; y = (\lambda \tau. \text{ if } (\delta \; x) \; \tau = \text{true} \; \tau \land (\nu \; y) \; \tau = \text{true} \; \tau \)

\hspace{1cm} \text{then } \text{Abs-Set-0} [\lceil \lceil \text{Rep-Set-0} (\; x \; \tau) \rceil \rceil - \{ y \; \tau \} \rceil \rceil \)

\hspace{1cm} \text{else } \bot \)

**Notation** \(\text{OclExcluding} \; (-\rightarrow \text{excluding}'(-') \)
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.

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

**definition OclForall :: \[(\mathcal{A}, \alpha::null) Set, (\mathcal{A}, \alpha) val \Rightarrow \mathcal{A} Boolean\]**

**where OclForall S P = (\lambda \cdot \cdot \cdot) \Rightarrow (\mathcal{A}) Boolean\]**

**where OclForall S P = (\lambda \cdot \cdot \cdot) \Rightarrow (\mathcal{A}) Boolean\]**

**where OclForall S P = (\lambda \cdot \cdot \cdot) \Rightarrow (\mathcal{A}) Boolean\]**
else true $\tau$

\begin{verbatim}
syntax
-OclForall :: [(A,'a::null) Set.id,(A)Boolean] $\Rightarrow$ 'A Boolean  ((-) $\Rightarrow$ forAll(' -'))
translations
X $\Rightarrow$ forAll($\setminus$ | P) $\Rightarrow$ CONST OclForAll X ($\%$ x. P)

Like OclForall, OclExists is also not strict.

definition OclExists :: [(A,'a::null) Set.(A,'a)val$\Rightarrow$(A)Boolean] $\Rightarrow$ 'A Boolean
where OclExists S P = NOT(OclForall S (\lambda X. NOT(P X)))

syntax
-OclExists :: [(A,'a::null) Set.id,(A)Boolean] $\Rightarrow$ 'A Boolean  ((-) $\Rightarrow$ exists(' -'))
translations
X $\Rightarrow$ exists($\setminus$ | P) $\Rightarrow$ CONST OclExists X ($\%$ x. P)

definition OclIterate :: [(A,'a::null) Set,(A,'\beta::null)val, (A,'\alpha)val$\Rightarrow$(A,'\beta)val$\Rightarrow$(A,'\alpha)val] $\Rightarrow$ (A,'\alpha)val
where OclIterate S A P = (\lambda \tau. if (S A) $\tau$ THEN true $\tau$ \&\& finite([(Rep-Set-0 (S $\tau$))])
then (Finite-Set.fold (F) (A) ((\lambda \alpha. $\alpha$) ' [Rep-Set-0 (S $\tau$)]))$\tau$
else $\bot$

translations
X $\Rightarrow$ iterate(a; x = A | P) $\Rightarrow$ CONST OclIterate X A (%a. (% x. P))

definition OclSelect :: [(A,'a::null)Set,(A,'a)val$\Rightarrow$(A)Boolean] $\Rightarrow$ (A,'a)Set
where OclSelect S P = (\lambda \tau. if (S A) $\tau$ THEN true $\tau$
then \bot
else Abs-Set-0 [[((x\in[[Rep-Set-0 (S $\tau$)])]. P (\lambda.- x) $\tau$ \neq false $\tau$)]
else $\bot$

syntax
-OclSelect :: [(A,'a::null) Set.id,(A)Boolean] $\Rightarrow$ 'A Boolean  ((-) $\Rightarrow$ select(' -'))
translations
X $\Rightarrow$ select($\setminus$ | P) $\Rightarrow$ CONST OclSelect X ($\%$ x. P)

definition OclReject :: [(A,'a::null)Set,(A,'a)val$\Rightarrow$(A)Boolean] $\Rightarrow$ (A,'a::null)Set
where OclReject S P = OclSelect S (NOT P)

syntax
-OclReject :: [(A,'a::null) Set.id,(A)Boolean] $\Rightarrow$ 'A Boolean  ((-) $\Rightarrow$ reject(' -'))
translations
X $\Rightarrow$ reject($\setminus$ | P) $\Rightarrow$ CONST OclReject X ($\%$ x. P)

Definition (futur operators)

consts
OclCount :: [(A,'a::null) Set,(A,'a)Set] $\Rightarrow$ 'A Integer
\end{verbatim}
notation
  \(OclCount\) \((-\rightarrow count\,(-)\)
notation
  \(OclSum\) \((-\rightarrow sum\,(-)\)
notation
  \(OclIncludesAll\) \((-\rightarrow includesAll\,(-)\)
notation
  \(OclExcludesAll\) \((-\rightarrow excludesAll\,(-)\)
notation
  \(OclComplement\) \((-\rightarrow complement\,(-)\)
notation
  \(OclUnion\) \((-\rightarrow union\,(-)\)
notation
  \(OclIntersection\) \((-\rightarrow intersection\,(-)\)

4.4.2. Validity and Definedness Properties

OclIncluding

lemma \(OclIncluding-defined-args-valid\):
\[
(\tau \models \delta(X \rightarrow including(x))) = ((\tau \models (\delta \; X)) \land (\tau \models (v \; x)))
\]
proof –
  have \(A: \bot \in \{X. \; X = bot \vee X = null \vee (\forall x \in [[X]]. \; x \neq bot)\}\) by(simp add: bot-option-def)
  have \(B: \bot \in \{X. \; X = bot \vee X = null \vee (\forall x \in [[X]]. \; x \neq bot)\}\) by(simp add: null-option-def bot-option-def)
  have \(C: (\tau \models (\delta X)) \Rightarrow (\tau \models (v \; x)) \Rightarrow [[insert \; (x \; \tau) \; [[Rep-Set-0 \; (X \; \tau)]]]] \in \{X. \; X = bot \vee X = null \vee (\forall x \in [[X]]. \; x \neq bot)\}\)
  by(frule Set-inv-lemma, simp add: foundation18 invalid-def)
  have \(D: (\tau \models (\delta(X \rightarrow including(x)))) \Rightarrow ((\tau \models (\delta \; X)) \land (\tau \models (v \; x)))\)
  by(auto simp: OclIncluding-def OclValid-def true-def valid-def false-def StrongEq-def
  defined-def invalid-def bot-fun-def null-fun-def
  split: bool.split-asm HOL.split-if-asm option.split)
  have \(E: (\tau \models (\delta X)) \Rightarrow (\tau \models (v \; x)) \Rightarrow (\tau \models (X \rightarrow including(x)))\)
  apply(subst OclIncluding-def, subst OclValid-def, subst defined-def)
  apply(auto simp: OclValid-def null-Set-0-def bot-Set-0-def null-fun-def bot-fun-def)
  apply(frule Abs-Set-0-inject[OF C A, simplified OclValid-def, THEN iffD1],
  simp-all add: bot-option-def)
  apply(frule Abs-Set-0-inject[OF C B, simplified OclValid-def, THEN iffD1],
  simp-all add: option-option)
  done
show \(\text{thesis}\) by(auto dest:D intro:E)
qed
lemma OclIncluding-valid-args-valid:
\((\tau \models v(X\to including(x))) = ((\tau \models(\delta X)) \land (\tau \models(v x)))\)
proof –
  have D: \((\tau \models v(X\to including(x))) \Longrightarrow ((\tau \models(\delta X)) \land (\tau \models(v x)))\)
    by(auto simp: OclIncluding-def OclValid-def true-def valid-def false-def StrongEq-def
defined-def invalid-def bot-fun-def null-fun-def
split: bool.split-asm HOL.split-if-asm option.split)
  have E: \((\tau \models(\delta X)) \Longrightarrow (\tau \models(v x)) \Longrightarrow (\tau \models v(X\to including(x)))\)
    by(simp add: foundation20 OclIncluding-defined-args-valid)
show \(?thesis\) by(auto dest:D intro:E)
qed

lemma OclIncluding-defined-args-valid:\[simp,code-unfold]:
\(\delta(X\to including(x)) = ((\delta X) \text{ and } (v x))\)
by(auto intro!: transform2-rev simp:OclIncluding-defined-args-valid foundation10 defined-and-I)

lemma OclIncluding-valid-args-valid'\[simp,code-unfold]:
\(v(X\to including(x)) = ((\delta X) \text{ and } (v x))\)
by(auto intro!: transform2-rev simp:OclIncluding-valid-args-valid foundation10 defined-and-I)

OclExcluding

lemma OclExcluding-defined-args-valid:
\((\tau \models(\delta X\to excluding(x))) = ((\tau \models(\delta X)) \land (\tau \models(v x)))\)
proof –
  have A : \(\bot \in \{X. X = \text{bot} \lor X = \text{null} \lor (\forall x\in\llbracket X\rrbracket, x \neq \text{bot})\}\) by(simp add: bot-option-def)
  have B : \(\llbracket \bot \rrbracket \in \{X. X = \text{bot} \lor X = \text{null} \lor (\forall x\in\llbracket X\rrbracket, x \neq \text{bot})\}\)
    by(simp add: null-option-def)
  have C : \((\tau \models(\delta X)) \Longrightarrow (\tau \models(v x)) \Longrightarrow\)
    \(\llbracket\llbracket \text{Rep-Set-0} (X \tau)\rrbracket - \{x \tau\}\rrbracket \in \{X. X = \text{bot} \lor X = \text{null} \lor (\forall x\in\llbracket X\rrbracket, x \neq \text{bot})\}\)
    by(frule Set-inv-lemma, simp add: foundation18 invalid-def)
  have D: \((\tau \models(\delta X\to excluding(x))) \Longrightarrow ((\tau \models(\delta X)) \land (\tau \models(v x)))\)
    by(auto simp: OclExcluding-def OclValid-def true-def valid-def false-def StrongEq-def
defined-def invalid-def bot-fun-def null-fun-def
split: bool.split-asm HOL.split-if-asm option.split)
  have E: \((\tau \models(\delta X)) \Longrightarrow (\tau \models(\delta X\to excluding(x)))\)
    apply(subst OclExcluding-def, subst OclValid-def, subst defined-def)
    apply(auto simp: OclValid-def null-Set-0-def bot-Set-0-def null-fun-def bot-fun-def)
    apply(frule Abs-Set-0-inject[OF C A, simplified OclValid-def, THEN iffD1],
      simp-all add: bot-option-def)
    apply(frule Abs-Set-0-inject[OF C B, simplified OclValid-def, THEN iffD1],
      simp-all add: bot-option-def)
    done
show \(?thesis\) by(auto dest:D intro:E)
qed
**Lemma** **OclExcluding-valid-args-valid**: 
\( (\tau \models v(X \rightarrow \text{excluding}(x))) = ((\tau \models (\delta X)) \land (\tau \models (v x))) \)

**proof** –

**have** \( D: (\tau \models v(X \rightarrow \text{excluding}(x))) \implies ((\tau \models (\delta X)) \land (\tau \models (v x))) \)

(by \( \text{auto simp: OclExcluding-def OclValid-def true-def valid-def false-def StrongEq-def} \)
  \( \text{defined-def invalid-def bot-fun-def null-fun-def} \)
  \( \text{split: bool.split-asn HOL.split-if-asn option.split} \))

**have** \( E: (\tau \models (\delta X)) \implies (\tau \models (v x)) \implies (\tau \models v(X \rightarrow \text{excluding}(x))) \)

(by \( \text{simp add: foundation20 OclExcluding-defined-args-valid} \))

**show** \( \text{thesis} \) **by** (\( \text{auto dest:D intro:E} \))

**qed**

**Lemma** **OclExcluding-valid-args-valid' [simp, code-unfold]**: 
\( \delta(X \rightarrow \text{excluding}(x)) = ((\delta X) \text{ and } (v x)) \)

by (\( \text{auto intro!: transform2-rev simp:OclExcluding-defined-args-valid foundation10 defined-and-I} \))

**Lemma** **OclExcluding-valid-args-valid''[simp, code-unfold]**: 
\( v(X \rightarrow \text{excluding}(x)) = ((\delta X) \text{ and } (v x)) \)

by (\( \text{auto intro!: transform2-rev simp:OclExcluding-valid-args-valid foundation10 defined-and-I} \))

**OclIncludes**

**Lemma** **OclIncludes-defined-args-valid**: 
\( (\tau \models \delta(X \rightarrow \text{includes}(x))) = ((\tau \models (\delta X)) \land (\tau \models (v x))) \)

**proof** –

**have** \( A: (\tau \models \delta(X \rightarrow \text{includes}(x))) \implies ((\tau \models (\delta X)) \land (\tau \models (v x))) \)

(by \( \text{auto simp: OclIncludes-def OclValid-def true-def valid-def false-def StrongEq-def} \)
  \( \text{defined-def invalid-def bot-fun-def null-fun-def} \)
  \( \text{split: bool.split-asn HOL.split-if-asn option.split} \))

**have** \( B: (\tau \models (\delta X)) \implies (\tau \models (v x)) \implies (\tau \models \delta(X \rightarrow \text{includes}(x))) \)

(by \( \text{auto simp: OclIncludes-def OclValid-def true-def false-def StrongEq-def} \)
  \( \text{defined-def invalid-def valid-def bot-fun-def null-fun-def} \)
  \( \text{bot-option-def null-option-def} \)
  \( \text{split: bool.split-asn HOL.split-if-asn option.split} \))

**show** \( \text{thesis} \) **by** (\( \text{auto dest:A intro:B} \))

**qed**

**Lemma** **OclIncludes-valid-args-valid**: 
\( (\tau \models v(X \rightarrow \text{includes}(x))) = ((\tau \models (\delta X)) \land (\tau \models (v x))) \)

**proof** –

**have** \( A: (\tau \models v(X \rightarrow \text{includes}(x))) \implies ((\tau \models (\delta X)) \land (\tau \models (v x))) \)

(by \( \text{auto simp: OclIncludes-def OclValid-def true-def valid-def false-def StrongEq-def} \)
  \( \text{defined-def invalid-def bot-fun-def null-fun-def} \)
  \( \text{split: bool.split-asn HOL.split-if-asn option.split} \))

**have** \( B: (\tau \models (\delta X)) \implies (\tau \models (v x)) \implies (\tau \models v(X \rightarrow \text{includes}(x))) \)

(by \( \text{auto simp: OclIncludes-def OclValid-def true-def false-def StrongEq-def} \)

81
show \textit{thesis} by (auto dest: A intro: B)

\textbf{OclIncludes}

\textbf{lemma OclIncludes-valid-args-valid}:[simp, code-unfold]:
\(\delta(X \to \text{includes}(x)) = ((\delta X) \land (v \ x))\)
by (auto intro!: transform2-rev simp: OclIncludes-defined-args-valid foundation10 defined-and-I)

\textbf{lemma OclIncludes-valid-args-valid}′[simp, code-unfold]:
\(v(X \to \text{includes}(x)) = ((\delta X) \land (v \ x))\)
by (auto intro!: transform2-rev simp: OclIncludes-defined-args-valid foundation10 defined-and-I)

\textbf{OclExcludes}

\textbf{lemma OclExcludes-defined-args-valid}:
\((\tau \mid = \delta(X \to \text{excludes}(x))) = ((\tau \mid = \delta X) \land (\tau \mid = v \ x))\)
by (metis (hide-lams, no-types) OclExcludes-def OclAnd-idem OclOr-def OclOr-idem defined-not-I OclIncludes-defined-args-valid)

\textbf{lemma OclExcludes-defined-args-valid}′[simp, code-unfold]:
\(\delta(X \to \text{excludes}(x)) = ((\delta X) \land (v \ x))\)
by (auto intro!: transform2-rev simp: OclExcludes-defined-args-valid foundation10 defined-and-I)

\textbf{lemma OclExcludes-defined-args-valid}′′[simp, code-unfold]:
\(v(X \to \text{excludes}(x)) = ((\delta X) \land (v \ x))\)
by (auto intro!: transform2-rev simp: OclExcludes-defined-args-valid foundation10 defined-and-I)

\textbf{OclSize}

\textbf{lemma OclSize-defined-args-valid}:
\(\tau \mid = \delta(X \to \text{size}()) \Rightarrow \tau \mid = \delta X\)
by(auto simp: OclSize-def OclValid-def true-def valid-def false-def StrongEq-def defined-def invalid-def bot-fun-def null-fun-def
    split: bool.split-asn HOL.split-if-asn option.split)

\textbf{lemma OclSize-infinite}:
assumes non-finite: \(\tau \mid = \text{not}(\delta(S \to \text{size}()))\)
shows \((\tau \mid = \text{not}(\delta(S))) \lor \neg \text{finite} \ [[\text{Rep-Set-0} \ (S \ \tau)]]\)
apply(insert non-finite, simp)
apply(rule simpI)
apply(simp add: OclSize-def OclValid-def defined-def)
apply(case-tac finite [[\text{Rep-Set-0} \ (S \ \tau)]]
    simp-all add: null-fun-def null-option-def bot-fun-def bot-option-def)

82
done

lemma τ ⊨ δ X ⇒ ¬ finite [[Rep-Set-0 (X τ)]] ⇒ ¬ τ ⊨ δ (X → size())
by(simp add: OclSize-def OclValid-def defined-def bot-fun-def false-def true-def)

lemma size-defined:
assumes X-finite: ∀τ. finite [[Rep-Set-0 (X τ)]]
shows δ (X → size()) = δ X
apply(rule ext, simp add: cp-defined[of X → size()] OclSize-def)
apply(simp add: defined-def bot-option-def bot-fun-def null-option-def null-fun-def X-finite)
done

lemma size-defined'
assumes X-finite: finite [[Rep-Set-0 (X τ)]]
shows (τ | = δ (X → size())) = (τ | = δ X)
apply(simp add: cp-defined[of X → size()] OclSize-def OclValid-def)
apply(simp add: defined-def bot-option-def bot-fun-def null-option-def null-fun-def X-finite)
done

OclIsEmpty

lemma OclIsEmpty-defined-args-valid: τ ⊨ δ (X → isEmpty()) ⇒ τ ⊨ v X
apply(auto simp: OclIsEmpty-def OclValid-def defined-def valid-def false-def true-def
bot-fun-def null-fun-def OclAnd-def OclOr-def OclNot-def
split: split-if-asm)
apply(case-tac (X → size()) = 0) τ, simp add: bot-option-def, simp, rename-tac x)
apply(case-tac x, simp add: null-option-def bot-option-def, simp)
apply(simp add: OclSize-def StrictRefEq Integer valid-def)
by (metis (hide-lams, no-types)
    OCL-core.bot-fun-def OclValid-def defined-def foundation2 invalid-def)

lemma τ ⊨ δ (null → isEmpty())
by(auto simp: OclIsEmpty-def OclValid-def defined-def valid-def false-def true-def
    bot-fun-def null-fun-def OclAnd-def OclOr-def OclNot-def
    null-is-valid
    split: split-if-asm)

lemma OclIsEmpty-infinite: τ ⊨ δ X ⇒ ¬ finite [[Rep-Set-0 (X τ)]] ⇒ ¬ τ ⊨ δ (X → isEmpty())
apply(auto simp: OclIsEmpty-def OclValid-def defined-def valid-def false-def true-def
    bot-fun-def null-fun-def OclAnd-def OclOr-def OclNot-def
    split: split-if-asm)
apply(case-tac (X → size()) = 0) τ, simp add: bot-option-def, simp, rename-tac x)
apply(case-tac x, simp add: null-option-def bot-option-def, simp)
by(simp add: OclSize-def StrictRefEq Integer valid-def bot-fun-def false-def true-def invalid-def)

OclNotEmpty

lemma OclNotEmpty-defined-args-valid: τ ⊨ δ (X → notEmpty()) ⇒ τ ⊨ v X
by (metis (hide-lams, no-types) OclNotEmpty-def OclNot-defargs OclNot-not foundation6 foundation9)
**OclIsEmpty-defined-args-valid**

**Lemma** \( \tau \models \delta (\text{null} \Rightarrow \text{notEmpty}) \)

**By** (metis (hide-lams, no-types) OclNotEmpty-def OclAnd-false1 OclAnd-iden OclIsEmpty-def OclNot3 OclNot4 OclOr-def defined2 defined4 transform1 valid2)

**Lemma** OclNotEmpty-infinite: \( \tau \models \delta X \iff \neg \text{finite } [[\text{Rep-Set-0} (X \tau)]] \iff \neg \tau \models \delta (X \Rightarrow \text{notEmpty}) \)

**Apply** (simp add: OclNotEmpty-def)

**Apply** (drule OclIsEmpty-infinite, simp)

**By** (metis OclNot-def args OclNot-not foundation6 foundation9)

**Lemma** OclNotEmpty-has-elt:

\( \tau \models \delta X \Rightarrow \tau \models \delta X \Rightarrow \text{notEmpty}() \)

**Apply** (simp add: OclNotEmpty-def OclIsEmpty-def deMorgan1 deMorgan2, drule foundation5)

**Apply** (subst (asm) (2) OclNot-def, simp add: OclValid-def StrictRefEqInteger StrongEq-def split: split-if-asm)

**Prefer** 2

**Apply** (simp add: invalid-def bot-option-def true-def)

**Apply** (simp add: OclSize-def valid-def split: split-if-asm, simp-all add: false-def true-def bot-option-def bot-fun-def OclInt0-def)

**By** (metis equals0I)

**OclANY**

**Lemma** OclANY-defined-args-valid: \( \tau \models \delta (X \Rightarrow \text{any}()) \Rightarrow \tau \models \delta X \)

**By** (auto simp: OclANY-def OclValid-def false-def true-def StrongEq-def defined-def invalid-def bot-fun-def null-fun-def OclAnd-def split: bool.split-asm HOL.split-if-asm option.split)

**Lemma** \( \tau \models \delta X \Rightarrow \tau \models X \Rightarrow \text{isEmpty}() \Rightarrow \neg \tau \models \delta (X \Rightarrow \text{any}()) \)

**Apply** (simp add: OclANY-def OclValid-def)

**Apply** (subst cp-defined, subst cp-OclAnd, simp add: OclNotEmpty-def, subst (1 2) cp-OclNot, simp add: cp-OclNot[symmetric] cp-OclAnd[symmetric] cp-defined[symmetric], simp add: false-def true-def)

**By** (drule foundation20 [simplified OclValid-def true-def], simp)

**Lemma** OclANY-valid-args-valid:

\( (\tau \models v (X \Rightarrow \text{any}())) \Rightarrow (\tau \models v X) \)

**Proof**

**Have** A: \( (\tau \models v (X \Rightarrow \text{any}())) \Rightarrow ((\tau \models v X)) \)

**By** (auto simp: OclANY-def OclValid-def true-def valid-def false-def StrongEq-def defined-def invalid-def bot-fun-def null-fun-def split: bool.split-asm HOL.split-if-asm option.split)

**Have** B: \( (\tau \models v X) \Rightarrow (\tau \models v (X \Rightarrow \text{any}())) \)

**Apply** (auto simp: OclANY-def OclValid-def true-def false-def StrongEq-def defined-def invalid-def valid-def bot-fun-def null-fun-def)

84
4.4.3. Execution with Invalid or Null or Infinite Set as Argument

**OclIncluding**

**lemma OclIncluding-invalid**[simp,code-unfold]:\( \text{invalid} \rightarrow \text{including}(x) = \text{invalid} \)

by(simp add: bot-fun-def OclIncluding-def invalid-def defined-def valid-def false-def true-def)

**lemma OclIncluding-invalid-args**[simp,code-unfold]:\( X \rightarrow \text{including}(\text{invalid}) = \text{invalid} \)

by(simp add: OclIncluding-def invalid-def bot-fun-def defined-def valid-def false-def true-def)

**lemma OclIncluding-null**[simp,code-unfold]:\( \text{null} \rightarrow \text{including}(x) = \text{invalid} \)

by(simp add: OclIncluding-def invalid-def bot-fun-def defined-def valid-def false-def true-def)

**OclExcluding**

**lemma OclExcluding-invalid**[simp,code-unfold]:\( \text{invalid} \rightarrow \text{excluding}(x) = \text{invalid} \)

by(simp add: bot-fun-def OclExcluding-def invalid-def defined-def valid-def false-def true-def)

**lemma OclExcluding-invalid-args**[simp,code-unfold]:\( X \rightarrow \text{excluding}(\text{invalid}) = \text{invalid} \)

by(simp add: OclExcluding-def invalid-def bot-fun-def defined-def valid-def false-def true-def)

**lemma OclExcluding-null**[simp,code-unfold]:\( \text{null} \rightarrow \text{excluding}(x) = \text{invalid} \)

by(simp add: OclExcluding-def invalid-def bot-fun-def defined-def valid-def false-def true-def)

**OclIncludes**

**lemma OclIncludes-invalid**[simp,code-unfold]:\( \text{invalid} \rightarrow \text{includes}(x) = \text{invalid} \)

by(simp add: bot-fun-def OclIncludes-def invalid-def defined-def valid-def false-def true-def)

**lemma OclIncludes-invalid-args**[simp,code-unfold]:\( X \rightarrow \text{includes}(\text{invalid}) = \text{invalid} \)

by(simp add: OclIncludes-def invalid-def bot-fun-def defined-def valid-def false-def true-def)
lemma OclIncludes-null(simp,code-unfold): (null -> includes(x)) = invalid 
by(simp add: OclIncludes-def invalid-def bot-fun-def defined-def valid-def false-def true-def)

OclExcludes

lemma OclExcludes-invalid(simp,code-unfold): (invalid -> excludes(x)) = invalid 
by(simp add: OclExcludes-def OclNot-def, simp add: invalid-def bot-option-def)

lemma OclExcludes-invalid-args(simp,code-unfold): (X -> excludes(invalid)) = invalid 
by(simp add: OclExcludes-def OclNot-def, simp add: invalid-def bot-option-def)

lemma OclExcludes-null(simp,code-unfold): (null -> excludes(x)) = invalid 
by(simp add: OclExcludes-def OclNot-def, simp add: invalid-def bot-option-def)

OclSize

lemma OclSize-invalid(simp,code-unfold): (invalid -> size()) = invalid 
by(simp add: bot-fun-def OclSize-def invalid-def defined-def valid-def false-def true-def)

lemma OclSize-null(simp,code-unfold): (null -> size()) = invalid 
by(rule ext,
  simp add: bot-fun-def null-fun-def null-is-valid OclSize-def
  invalid-def defined-def valid-def false-def true-def)

OclIsEmpty

lemma OclIsEmpty-invalid(simp,code-unfold): (invalid -> isEmpty()) = invalid 
by(simp add: OclIsEmpty-def)

lemma OclIsEmpty-null(simp,code-unfold): (null -> isEmpty()) = true 
by(simp add: OclIsEmpty-def)

OclNotEmpty

lemma OclNotEmpty-invalid(simp,code-unfold): (invalid -> notEmpty()) = invalid 
by(simp add: OclNotEmpty-def)

lemma OclNotEmpty-null(simp,code-unfold): (null -> notEmpty()) = false 
by(simp add: OclNotEmpty-def)

OclANY

lemma OclANY-invalid(simp,code-unfold): (invalid -> any()) = invalid 
by(simp add: bot-fun-def OclANY-def invalid-def defined-def valid-def false-def true-def)

lemma OclANY-null(simp,code-unfold): (null -> any()) = null 
by(simp add: OclANY-def false-def true-def)

86
OclForall

lemma OclForall-invalid[simp,code-unfold];invalid→forAll(a | P a) = invalid
by(simp add: bot-fun-def invalid-def OclForall-def defined-def valid-def false-def true-def)

lemma OclForall-null[simp,code-unfold];null→forAll(a | P a) = invalid
by(simp add: bot-fun-def invalid-def OclForall-def defined-def valid-def false-def true-def)

OclExists

lemma OclExists-invalid[simp,code-unfold];invalid→exists(a | P a) = invalid
by(simp add: OclExists-def)

lemma OclExists-null[simp,code-unfold];null→exists(a | P a) = invalid
by(simp add: OclExists-def)

OclIterate

lemma OclIterate-invalid[simp,code-unfold];invalid→iterate(a; x = A | P a x) = invalid
by(simp add: bot-fun-def invalid-def OclIterate-def defined-def valid-def false-def true-def)

lemma OclIterate-null[simp,code-unfold];null→iterate(a; x = A | P a x) = invalid
by(simp add: bot-fun-def invalid-def OclIterate-def defined-def valid-def false-def true-def)

lemma OclIterate-invalid-args[simp,code-unfold];S→iterate(a; x = invalid | P a x) = invalid
by(simp add: bot-fun-def invalid-def OclIterate-def defined-def valid-def false-def true-def)

An open question is this ...

lemma S→iterate(a; x = null | P a x) = invalid
oops

lemma OclIterate-infinite:
assumes non-finite: τ |= not(δ(S→size(()))
shows (OclIterate S A F) τ = invalid τ
apply(insert non-finite [THEN OclSize-infinite])
apply(subst (asm) foundation9, simp)
by(metis OclIterate-def OclValid-def invalid-def)

OclSelect

lemma OclSelect-invalid[simp,code-unfold];invalid→select(a | P a) = invalid
by(simp add: bot-fun-def invalid-def OclSelect-def defined-def valid-def false-def true-def)

lemma OclSelect-null[simp,code-unfold];null→select(a | P a) = invalid
by(simp add: bot-fun-def invalid-def OclSelect-def defined-def valid-def false-def true-def)

OclReject

lemma OclReject-invalid[simp,code-unfold];invalid→reject(a | P a) = invalid
lemma OclReject-null[simp,code-unfold]:null→reject(a | P a) = invalid
by(simp add: OclReject-def)

4.4.4. Context Passing

lemma cp-OclIncluding:
(X -->including(x)) τ = ((λ . X τ) -->including(λ . x τ)) τ
by(auto simp: OclIncluding-def StrongEq-def invalid-def
     cp-defined[symmetric] cp-valid[symmetric])

lemma cp-OclExcluding:
(X -->excluding(x)) τ = ((λ . X τ) -->excluding(λ . x τ)) τ
by(auto simp: OclExcluding-def StrongEq-def invalid-def
     cp-defined[symmetric] cp-valid[symmetric])

lemma cp-OclIncludes:
(X -->includes(x)) τ = (X -->includes(λ . x τ)) τ
by(auto simp: OclIncludes-def StrongEq-def invalid-def
     cp-defined[symmetric] cp-valid[symmetric])

lemma cp-OclIncludes1:
(X -->includes(x)) τ = (X -->includes(λ . x τ)) τ
by(auto simp: OclIncludes-def StrongEq-def invalid-def
     cp-defined[symmetric] cp-valid[symmetric])

lemma cp-OclExcludes:
(X -->excludes(x)) τ = ((λ . X τ) -->excludes(λ . x τ)) τ
by(simp add: OclExcludes-def OclNot-def, subst cp-OclIncludes, simp)

lemma cp-OclSize: X -->size(τ) = ((λ . X τ) -->size(τ)) τ
by(simp add: OclSize-def cp-defined[symmetric])

lemma cp-OclIsEmpty: X -->isEmpty(τ) = ((λ . X τ) -->isEmpty(τ)) τ
apply(simp only: OclIsEmpty-def)
apply(subst (2) cp-OclOr,
      subst cp-OclAnd,
      subst cp-OclNot,
      subst cp-StrictRefEq[Integer]
     by(simp add: cp-defined[symmetric] cp-valid[symmetric] cp-StrictRefEq[Integer][symmetric]
           cp-OclSize[symmetric] cp-OclNot[symmetric] cp-OclAnd[symmetric]
           cp-OclOr[symmetric])

lemma cp-OclNotEmpty: X -->notEmpty(τ) = ((λ . X τ) -->notEmpty(τ)) τ
apply(simp only: OclNotEmpty-def)
apply(subst (2) cp-OclNot)
by(simp add: cp-OclNot[symmetric] cp-OclIsEmpty[symmetric])

88
lemma \( cp-OclANY \): \( X \rightarrow \text{any}() \) \( \tau = ((\lambda \cdot X \; \tau) \rightarrow \text{any}()) \) \( \tau \)
apply(simp only: OclANY-def)
apply(subst (2) cp-OclAnd)
by(simp only: cp-OclAnd[symmetric] cp-defined[symmetric] cp-valid[symmetric]
cp-OclNotEmpty[symmetric])

lemma \( cp-OclForall \):
\((S \rightarrow \forall (x \mid P \; x)) \) \( \tau = ((\lambda \cdot S \; \tau) \rightarrow \forall (x \mid P \; (\lambda \cdot x \; \tau))) \) \( \tau \)
by(simp add: OclForall-def cp-defined[symmetric])

lemma \( cp-OclForall1 \) [simp, intro!]:
\( cp \; S \Rightarrow cp \; (\lambda \; X \cdotp ((S \; X) \rightarrow \forall (x \mid P \; x))) \)
apply(simp add: cp-def)
apply(erule exE, rule exI, intro allI)
apply(erule-tac x=X in allE)
by(subst cp-OclForall, simp)

lemma "cp-OclExists":
\((S \rightarrow \exists (x \mid P \; x)) \) \( \tau = ((\lambda \cdot S \; \tau) \rightarrow \exists (x \mid P \; (\lambda \cdot x \; \tau))) \) \( \tau \)
by(simp add: OclExists-def OclNot-def, subst cp-OclForall, simp)

lemma \( cp-OclExists1 \) [simp, intro!]:
\( cp \; S \Rightarrow cp \; (\lambda \; X \cdotp ((S \; X) \rightarrow \exists (x \mid P \; x))) \)
apply(simp add: cp-def)
apply(erule exE, rule exI, intro allI)
apply(erule-tac x=X in allE)
by(subst cp-OclExists, simp)

lemma \( cp-OclIterate \):
\((X \rightarrow \text{iterate}(a; \; x = A \mid P \; a \; x)) \) \( \tau = ((\lambda \cdot X \; \tau) \rightarrow \text{iterate}(a; \; x = A \mid P \; a \; x)) \) \( \tau \)
by(simp add: OclIterate-def cp-defined[symmetric])

lemma \( cp-OclSelect \):
\((X \rightarrow \text{select}(a \mid P \; a)) \) \( \tau = ((\lambda \cdot X \; \tau) \rightarrow \text{select}(a \mid P \; a)) \) \( \tau \)
by (simp add: OclSelect-def cp-defined[symmetric])

lemma cp-OclReject: \((X \to reject(a \mid P a)) \tau =
((\lambda - X \tau) \to reject(a \mid P a)) \tau\)
by (simp add: OclReject-def, subst cp-OclSelect, simp)

lemmas cp-intro"[intro!, simp, code-unfold] =
cp-intro'
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]]

4.4.5. Const

lemma const-OclIncluding[simp, code-unfold] :
assumes const-x : const x
and const-S : const S
shows const (S \to including(x))
proof -
have A: \(\forall \tau \tau'. \neg (\tau | v \ x) \to (S \to including(x) \tau) = (S \to including(x) \tau')\)
  apply (simp add: foundation18)
  apply (erule const-subst[OF const-x const-invalid], simp-all)
  by (rule const-charn[OF const-invalid])
have B: \(\forall \tau \tau'. \neg (\tau | \delta \ S) \to (S \to including(x) \tau) = (S \to including(x) \tau')\)
  apply (simp add: foundation16', elim disjE)
  apply (erule const-subst[OF const-S const-invalid], simp-all)
  apply (rule const-charn[OF const-invalid])
  apply (erule const-subst[OF const-S const-null], simp-all)
  by (rule const-charn[OF const-invalid])
show ?thesis
  apply (simp only: const-def,intro allI, rename-tac \tau \tau')
  apply (case-tac \(\tau | v \ x\), simp add: A)
  apply (case-tac \(\tau | \delta \ S\), simp-all add: B)
  apply (frule-tac \tau' \tau'1 = \tau' in const-OclValid2[OF const-x, THEN iffD1])
  apply (frule-tac \tau' \tau'1 = \tau' in const-OclValid1[OF const-S, THEN iffD1])
  apply (simp add: OclIncluding-def OclValid-def)
  apply (subst const-charn[OF const-x])
  apply (subst const-charn[OF const-S])
  by simp

qed
4.5. Fundamental Predicates on Set: Strict Equality

4.5.1. 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:

```plaintext
defs (overloaded) StrictRefEqSet :
  (x:(A, α::null)Set) = y = λ τ. if (υ x) τ = true τ ∧ (υ y) τ = true τ 
  then (x = y)τ 
  else invalid τ
```

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

4.5.2. Logic and Algebraic Layer on Set

Reflexivity

To become operational, we derive:

**Lemma** StrictRefEqSet-refl[simp, code-unfold]:

```plaintext
((x:(A, α::null)Set) = x) = (if (υ x) then true else invalid endif)
```

**Proof** (rule ext, simp add: StrictRefEqSet OclIf-def)

Symmetry

**Lemma** StrictRefEqSet-sym:

```plaintext
((x:(A, α::null)Set) = y) = (y = x)
```

**Proof** (simp add: StrictRefEqSet, subst StrongEq-sym, rule ext, simp)

Execution with Invalid or Null as Argument

**Lemma** StrictRefEqSet-strict1[simp, code-unfold]: ((x:(A, α::null)Set) = invalid) = invalid

**Proof** (simp add: StrictRefEqSet false-def true-def)

**Lemma** StrictRefEqSet-strict2[simp, code-unfold]: (invalid = y:(A, α::null)Set)) = invalid

**Proof** (simp add: StrictRefEqSet false-def true-def)

**Lemma** StrictRefEqSet-strictEq-args-valid:

```plaintext
(τ |= δ ((x:(A, α::null)Set) = y)) = ((τ |= (υ x)) ∧ (τ |= υ y))
```

**Proof**

- **have** A: τ |= δ (x = y) --- τ |= υ x ∧ τ |= υ y
  - apply (simp add: StrictRefEqSet valid-def OclValid-def defined-def)
apply (simp add: invalid-def bot-fun-def split split-if-asm)
done

have B: \(\tau \mid v x \land \tau \mid v y \Rightarrow \tau \mid \delta (x \triangleq y)\)
  apply (simp add: StrictRefEqSet, elim conjE)
  apply (drule foundation13[THEN iffD2], drule foundation13[THEN iffD2])
  apply (rule cp-validity[THEN iffD2])
  apply (subst cp-defined, simp add: foundation22)
  apply (simp add: cp-defined[symmetric] cp-validity[symmetric])
done

show \(?\text{thesis}\) by (auto intro!: A B)
qed

Behavior vs StrongEq

lemma StrictRefEqSet-vs-StrongEq:
\(\tau \mid v x \Rightarrow \tau \mid v y \Rightarrow (\tau \mid ((x::(\mathbb{A}',\alpha::\text{null})\text{Set}) \triangleq y) \triangleq (x \triangleq y))\)
apply (drule foundation13[THEN iffD2], drule foundation13[THEN iffD2])
by (simp add: StrictRefEqSet foundation22)

Context Passing

lemma cp-StrictRefEqSet: \((X::(\mathbb{A}',\alpha::\text{null})\text{Set}) \triangleq Y) \tau = ((\lambda\cdot X \tau) \triangleq (\lambda\cdot Y \tau)) \tau
by (simp add: StrictRefEqSet cp-StrongEq[symmetric] cp-valid[symmetric])

Const

lemma const-StrictRefEqSet:
  assumes const X-def: \(\tau \mid \delta \ X\)
  assumes const X': \(\tau \mid \delta \ X'\)
  shows const X \triangleq X'
apply (simp only: const-def, intro allI)
proof -
fix \(\tau_1\) \(\tau_2\) show (X \triangleq X') \(\tau_1 = (X \triangleq X') \tau_2\)
apply (simp only: StrictRefEqSet)
by (simp add: const-valid[OF assms(1), simplified const-def, THEN spec, THEN spec, of \(\tau_1\) \(\tau_2\)])

const-valid[OF assms(2), simplified const-def, THEN spec, THEN spec, of \(\tau_1\) \(\tau_2\)]
const-true[OF assms(2), simplified const-def, THEN spec, THEN spec, of \(\tau_1\) \(\tau_2\)]
const-invalid[OF assms(2), simplified const-def, THEN spec, THEN spec, of \(\tau_1\) \(\tau_2\)]
const-StrongEq[OF assms, simplified const-def, THEN spec, THEN spec]
qed

4.6. Execution on Set’s Operators (with mtSet and recursive case as arguments)

4.6.1. OclIncluding

lemma OclIncluding-finite-rep-set:
  assumes X-def: \(\tau \mid \delta X\)
and \(x\text{-val} : \tau \models v \ x\)

shows finite \([\text{Rep-Set-0} (X \rightarrow \text{including}(x) \ \tau)]\) = finite \([\text{Rep-Set-0} (X \ \tau)]\)

proof –
have \(C : \{\{\text{insert } (x \ \tau) [\text{Rep-Set-0} (X \ \tau)]\}\in \{X. X = \text{bot} \lor X = \text{null} \lor (\forall x \in [X]. x \neq \text{bot})\}\}

by(insert \text{X-def} \ x\text{-val}, \text{frule Set-inv-lemma}, \text{simp add: foundation18 invalid-def})

show \(\text{thesis}\)
by(insert \text{X-def} \ x\text{-val},
auto simp: \text{OclIncluding-def} \text{Abs-Set-0-inverse}| OF \ C
dest: foundation13|THEN iffD2, \text{THEN foundation22|THEN iffD1}|)

qed

lemma \text{OclIncluding-rep-set}:
assumes \(S\text{-def} : \tau \models \delta S\)
shows \([\text{Rep-Set-0} (S \rightarrow \text{including}(\lambda. \ [\{x\}. \tau])]) = \text{insert} \ [\{x\}] [\text{Rep-Set-0} (S \ \tau)]\)
apply(simp add: \text{OclIncluding-def} \text{S-def|simplified OclValid-def})
apply(subst \text{Abs-Set-0-inverse}, simp add: \text{bot-option-def null-option-def})
apply(insert \text{Set-inv-lemma}|OF \ S\text{-def}, \text{metis bot-option-def not-Some-eq})
by(simp)

lemma \text{OclIncluding-notempty-rep-set}:
assumes \(X\text{-def} : \tau \models \delta X\)
and \(a\text{-val} : \tau \models v \ a\)
shows \([\text{Rep-Set-0} (X \rightarrow \text{including}(a) \ \tau)]\) \neq \{
apply(simp add: \text{OclIncluding-def} \text{X-def|simplified OclValid-def} \text{a-val|simplified OclValid-def})
apply(subst \text{Abs-Set-0-inverse}, simp add: \text{bot-option-def null-option-def})
apply(insert \text{Set-inv-lemma}|OF \ X\text{-def}, \text{metis a-val foundation18}')
by(simp)

lemma \text{OclIncluding-includes}:
assumes \(\tau \models X \rightarrow \text{including}(x)\)
shows \(X \rightarrow \text{including}(x) \ \tau = X \ \tau\)

proof –
have \(\text{includes-def} : \tau \models X \rightarrow \text{including}(x) \implies \tau \models \delta X\)
by (metis \text{OCL-core|bot-fun-def OclIncludes-def OclValid-def defined3 foundation16})

have \(\text{includes-val} : \tau \models X \rightarrow \text{including}(x) \implies \tau \models v \ x\)
by (metis \text{hide-lams, no-types} foundation6
OclIncludes-valid-args-valid OclIncluding-valid-args-valid OclIncluding-valid-args-valid'O)

show \(\text{thesis}\)
apply(insert \text{includes-def}|OF \text{assms} \text{ includes-val}|OF \text{assms} \text{ assms},
\text{simp add: OclIncluding-def OclIncludes-def OclValid-def true-def})
apply(\text{drule insert-absorb, simp, subst abs-rep-simp}')
by(\text{simp-all add: OclValid-def true-def})

qed
4.6.2. OclExcluding

**Lemma** OclExcluding-charn0[simp]:
assumes `val-x:τ |= (v x)` 
shows `τ |= ((Set{}→excluding(x)) ≠ Set{})` 
proof –
  have A: [None] ∈ {X. X = bot ∨ X = null ∨ (∀x∈[[X]]. x ≠ bot)} 
  by(simp add: null-option-def bot-option-def)
  have B: [{} ] ∈ {X. X = bot ∨ X = null ∨ (∀x∈[[X]]. x ≠ bot)} by(simp add: mtSet-def)
  show ?thesis using val-x 
    apply(auto simp: OclValid-def OclIncludes-def OclNot-def false-def true-def StrongEq-def 
      OclExcluding-def mtSet-def defined-def bot-fun-def null-fun-def null-Set-0-def)
  apply(auto simp: OCL-lib Set-0.Abs-Set-0-injective OF B A)
  done
qed

**Lemma** OclExcluding-charn0-exec[simp,code-unfold]:
(Set{}→excluding(x)) = (if (v x) then Set{} else invalid endif)
proof –
  have A: ∀ τ. (Set{}→excluding(invalid)) τ = (if (v invalid) then Set{} else invalid endif) τ 
    by simp
  have B: ∀ τ x. τ |= (v x) ==> (Set{}→excluding(x)) τ = (if (v x) then Set{} else invalid endif) τ 
    by(simp add: OclExcluding-charn0[THEN foundation22[THEN iffD1]])
  show ?thesis 
    apply(rule ext, rename-tac τ)
    apply(case-tac τ |= (v x))
    apply(simp add: B)
    apply(simp add: foundation18)
    apply(subst cp-OclExcluding, simp)
    done
qed

**Lemma** OclExcluding-charn1:
assumes `def-X:τ |= (δ X)` 
and `val-x:τ |= (v x)` 
and `val-y:τ |= (v y)` 
and `neq:τ |= (v x ≠ y)` 
shows `τ |= ((X→including(x))→excluding(y)) = ((X→excluding(y))→including(x))` 
proof –
  have C: [[insert (x τ) [[Rep-Set-0 (X τ)]]]] ∈ {X. X = bot ∨ X = null ∨ (∀x∈[[X]]. x ≠ bot)} 
    by(insert def-X val-x, fraule Set-inv-lemma, simp add: foundation18 invalid-def)
  have D: [[[Rep-Set-0 (X τ)]]→{y τ}]] ∈ {X. X = bot ∨ X = null ∨ (∀x∈[[X]]. x ≠ bot)}
  done

94
by (insert def-X val-x, frule Set-inv-lemma, simp add: foundation18 invalid-def)

have E : x τ ≠ y τ
  (by (insert neq, auto simp: OclValid-def bot-fun-def OclIncluding-def OclExcluding-def false-def true-def defined-def bot-def Set-0-def null-def bot-def null-set-0-def StrongEq-def OclNot-def))

have G1 : Abs-Set-0 [[insert (x τ) [[Rep-Set-0 (X τ)]]]] ≠ Abs-Set-0 None
  by (insert C, simp add: Abs-Set-0-inject bot-option-def null-option-def)

have G2 : Abs-Set-0 [[insert (x τ) [[Rep-Set-0 (X τ)]]]] ≠ Abs-Set-0 [None]
  by (insert C, simp add: Abs-Set-0-inject bot-option-def null-option-def)

have G : (δ (λ-. Abs-Set-0 [[[insert (x τ) [[Rep-Set-0 (X τ)]]]]]) τ = true τ
  by (auto simp: OclValid-def false-def true-def defined-def bot-fun-def bot-def null-set-0-def null-set-0-def G1 G2)

have H1 : Abs-Set-0 [[[Rep-Set-0 (X τ)]] − {y τ}] ≠ Abs-Set-0 None
  by (insert D, simp add: Abs-Set-0-inject bot-option-def null-option-def)

have H2 : Abs-Set-0 [[[Rep-Set-0 (X τ)]] − {y τ}] ≠ Abs-Set-0 [None]
  by (insert D, simp add: Abs-Set-0-inject bot-option-def null-option-def)

have H : (δ (λ-. Abs-Set-0 [[[Rep-Set-0 (X τ)]] − {y τ}]) τ = true τ
  by (auto simp: OclValid-def false-def true-def defined-def bot-fun-def bot-def null-set-0-def null-set-0-def G1 G2)

have Z : insert (x τ) [[Rep-Set-0 (X τ)]] − {y τ} = insert (x τ) ([[Rep-Set-0 (X τ)]] − {y τ})
  (by (auto simp: E))

show ?thesis
apply (insert def-X[THEN foundation13[THEN iffD2]] val-x[THEN foundation13[THEN iffD2]]
  val-y[THEN foundation13[THEN iffD2]])
apply (simp add: foundation22 OclIncluding-def OclExcluding-def X[THEN foundation17])
apply (subst cp-defined, simp)+
apply (simp add: G H Abs-Set-0-inverse[OF C] Abs-Set-0-inverse[OF D] Z)
done

qed

lemma OclExcluding-charn2:
assumes def-X·τ |− (δ X)
and val·x·τ |− (v x)
shows τ |− ((X→including(x))→excluding(x)) ≡ (X→excluding(x))
proof −
  have C : [[insert (x τ) [[Rep-Set-0 (X τ)]]]] ∈ {X. X = bot ∨ X = null ∨ (∀ x∈ [[X]]. x ≠ bot)}
    by (insert def-X val-x, frule Set-inv-lemma, simp add: foundation18 invalid-def)
  (have G1 : Abs-Set-0 [[[insert (x τ) [[Rep-Set-0 (X τ)]]]]] ≠ Abs-Set-0 None
    by (insert C, simp add: Abs-Set-0-inject bot-option-def null-option-def)
    have G2 : Abs-Set-0 [[[insert (x τ) [[Rep-Set-0 (X τ)]]]]] ≠ Abs-Set-0 [None]
      by (insert C, simp add: Abs-Set-0-inject bot-option-def null-option-def)
    show ?thesis
definition valid-def bot-def null-def null-set-0-def StrongEq-def OclNot-def

95
apply(insert def-X [THEN foundation17] val-x [THEN foundation19])
apply(auto simp: OclValid-def bot-fun-def OclIncluding-def OclIncludes-def false-def true-def
invalid-def defined-def valid-def bot-Set-0-def null-fun-def null-Set-0-def
StrongEq-def)
apply(subst cp-OclExcluding)
apply(auto simp: OclExcluding-def)
apply(simp add: Abs-Set-0-inverse OF C)
apply(simp-all add: false-def true-def defined-def valid-def null-fun-def bot-fun-def null-Set-0-def
split: bool.split-asm HOL.split-if-asm option.split)
apply(auto simp: G1 G2)
done
qed

One would like a generic theorem of the form:

**lemma** OclExcluding_charn_exec:

"(X −> including(x::(′A,′a::null)val) −> excluding(y)) =
  (if δ X then if x = y
  then X −> excluding(y)
  else X −> excluding(y) −> including(x)
  endif
  else invalid endif)"

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

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

**lemma** OclExcluding-charn-exec:

assumes strict1: (x = invalid) = invalid
and strict2: (invalid = y) = invalid
and StrictRefEq-valid-args-valid: (x::(′A,′a::null)val) y τ.
  (τ ⊳ δ (x ⊳ y)) = ((τ ⊳ (v x)) ∧ (τ ⊳ v y))
and cp-StrictRefEq: (X::(′A,′a::null)val) Y τ. (X ⊳ Y) τ = ((λx. X τ) ⊳ (λy. Y τ)) τ
and StrictRefEq-vs-StrongEq: ((x::(′A,′a::null)val) y τ.
  τ ⊳ v x ⇒ τ ⊳ v y ⇒ (τ ⊳ ((x ⊳ y) = (x = y))))

shows (X −> including(x::(′A,′a::null)val) −> excluding(y)) =
  (if δ X then if x = y
  then X −> excluding(y)
  else X −> excluding(y) −> including(x)
  endif
  else invalid endif)

proof −

have A1: ∀τ. τ ⊳ (X ⊳ invalid) =⇒
have $B_1$: $\forall \tau. \tau \vdash (X \triangleq \text{null}) \iff (X \rightarrow \text{including}(x) \rightarrow \text{includes}(y)) \tau = \text{invalid} \ \tau$

apply (rule foundation22[THEN iffD1])
by (erule StrongEq-L-subst2-rev, simp, simp)

have $A_2$: $\forall \tau. \tau \vdash (X \triangleq \text{invalid}) \iff X \rightarrow \text{including}(x) \rightarrow \text{excluding}(y) \tau = \text{invalid} \ \tau$

apply (rule foundation22[THEN iffD1])
by (erule StrongEq-L-subst2-rev, simp, simp)

have $B_2$: $\forall \tau. \tau \vdash (X \triangleq \text{null}) \iff X \rightarrow \text{including}(x) \rightarrow \text{excluding}(y) \tau = \text{invalid} \ \tau$

apply (rule foundation22[THEN iffD1])
by (erule StrongEq-L-subst2-rev, simp, simp)

note [simp] = cp-StrictRefEq [THEN allI[THEN allI[THEN allI[THEN cpI2]], of StrictRefEq]]

have $C$: $\forall \tau. \tau \vdash (x \triangleq \text{invalid}) \implies (X \rightarrow \text{including}(x) \rightarrow \text{excluding}(y)) \tau =$

(if $x \triangleq y$ then $X \rightarrow \text{excluding}(y)$ else $X \rightarrow \text{excluding}(y) \rightarrow \text{including}(x)$ endif) $\tau$

apply (rule foundation22[THEN iffD1])
apply (erule StrongEq-L-subst2-rev, simp, simp)
by (simp add: strict2)

have $D$: $\forall \tau. \tau \vdash (y \triangleq \text{invalid}) \implies (X \rightarrow \text{including}(x) \rightarrow \text{excluding}(y)) \tau =$

(if $x \triangleq y$ then $X \rightarrow \text{excluding}(y)$ else $X \rightarrow \text{excluding}(y) \rightarrow \text{including}(x)$ endif) $\tau$

apply (rule foundation22[THEN iffD1])
apply (erule StrongEq-L-subst2-rev, simp, simp)
by (simp add: strict1)

have $E$: $\forall \tau. \tau \vdash v \ x \iff \tau \vdash v \ y \iff$

(if $x \triangleq y$ then $X \rightarrow \text{excluding}(y)$ else $X \rightarrow \text{excluding}(y) \rightarrow \text{including}(x)$ endif) $\tau =$

(if $x \triangleq y$ then $X \rightarrow \text{excluding}(y)$ else $X \rightarrow \text{excluding}(y) \rightarrow \text{including}(x)$ endif) $\tau$

apply (subst cp-OclIf)
apply (subst StrictRefEq-vs-StrongEq[THEN foundation22[THEN iffD1]])
by (simp-all add: cp-OclIf[symmetric])

have $F$: $\forall \tau. \tau \vdash \delta \ X \rightarrow \tau \vdash \nu \ x \iff \tau \vdash (x \triangleq y) \iff (X \rightarrow \text{including}(x) \rightarrow \text{excluding}(y) \tau) = (X \rightarrow \text{excluding}(y) \tau)$

apply (drule StrongEq-L-sym)
apply (rule foundation22[THEN iffD1])
apply (erule StrongEq-L-subst2-rev, simp)
by (simp add: OclExcluding-charn2)

show thesis

apply (rule ext, rename-tac $\tau$)
apply (case-tac \( \tau \models (\delta X) \)), simp add: def-split-local, elim disjE A1 B1 A2 B2
apply (case-tac \( \tau \models (v x) \)),
simp add: foundation18 foundation22[symmetric],
drule StrongEq-L-sym
apply (simp add: foundation22 C)
apply (case-tac \( \tau \models (v y) \)),
simp add: foundation18 foundation22[symmetric],
drule StrongEq-L-sym, simp add: foundation22 D, simp
apply (subst E, simp-all)
apply (case-tac \( \tau \models \text{not} (x \triangleq y) \))
apply (simp add: OclExcluding-charn1[simplified foundation22]
OclExcluding-charn2[simplified foundation22])
apply (simp add: foundation9 F)
done
qed

schematic-lemma OclExcluding-charn-execInteger [simp, code-unfold]: ?X
by (rule OclExcluding-charn-exec [OF StrictRefEqInteger-strict1 StrictRefEqInteger-strict2
StrictRefEqInteger-defined-args-valid
cp-StrictRefEqInteger StrictRefEqInteger-vs-StrongEq, simp-all])
schematic-lemma OclExcluding-charn-execBoolean [simp, code-unfold]: ?X
by (rule OclExcluding-charn-exec [OF StrictRefEqBoolean-strict1 StrictRefEqBoolean-strict2
StrictRefEqBoolean-defined-args-valid
cp-StrictRefEqBoolean StrictRefEqBoolean-vs-StrongEq], simp-all)
schematic-lemma OclExcluding-charn-execSet [simp, code-unfold]: ?X
by (rule OclExcluding-charn-exec [OF StrictRefEqSet-strict1 StrictRefEqSet-strict2
StrictRefEqSet-defined-args-valid
cp-StrictRefEqSet StrictRefEqSet-vs-StrongEq], simp-all)

lemma OclExcluding-finite-rep-set :
assumes X-def : \( \tau \models \delta X \)
and x-val : \( \tau \models v x \)
shows finite [\([\text{Rep-Set-0} (X \rightarrow \text{excluding}(x) \ \tau)]\)] = finite [\([\text{Rep-Set-0} (X \ \tau)]\)]
proof —
have C : [\([\text{Rep-Set-0} (X \ \tau)] - \{x \ \tau\}\)] \( \in \{X. \ X = \text{bot} \lor X = \text{null} \lor (\forall x \in [\{X\}]. \ x \neq \text{bot})\}\)
apply (insert X-def x-val, frule Set-inv-lemma)
apply (simp add: foundation18 invalid-def)
done
show \(?thesis
by (insert X-def x-val,
auto simp: OclExcluding-def Abs-Set-0-inverse [OF C]
dest: foundation13 [THEN iffD2, THEN foundation22 [THEN iffD1]])
qed

98
lemma OclExcluding-rep-set:
  assumes $S$-def: $\tau \models \delta S$
  shows $[[\text{Rep-Set-0} (S \rightarrow \text{excluding}(\lambda x. [[x]]) \, \tau)]] = [[\text{Rep-Set-0} (S \, \tau)]] - \{[[x]]\}$
apply(simp add: OclExcluding-def $S$-def[simplified OclValid-def])
apply(subst Abs-Set-0-inverse, simp add: bot-option-def null-option-def)
apply(insert Set-inv-lemma[OF $S$-def], metis Diff-iff bot-option-def not-None-eq)
by(simp)

4.6.3. OclIncludes

lemma OclIncludes-charn0[simp]:
  assumes val-x: $\tau \models (\nu x)$
  shows $\tau \models \neg (\text{Set} \emptyset \rightarrow \text{includes}(x))$
using val-x
apply(auto simp: OclValid-def OclIncludes-def OclNot-def false-def true-def)
apply(auto simp: mtSet-def OCL-lib.Abs-Set-0-inverse)
done

lemma OclIncludes-charn0'[simp, code-unfold]:
  $\text{Set} \emptyset \rightarrow \text{includes}(x) = (\text{if } \nu x \text{ then false else invalid endif})$
proof -
  have $A : \forall \tau. (\text{Set} \emptyset \rightarrow \text{includes}(\text{invalid})) \, \tau = (\text{if } \text{invalid} \text{ then false else invalid endif}) \, \tau$
    by simp
  have $B : \forall \tau x. \tau \models (\nu x) \implies (\text{Set} \emptyset \rightarrow \text{includes}(x)) \, \tau = (\text{if } \nu x \text{ then false else invalid endif}) \, \tau$
    apply(frule OclIncludes-charn0, simp add: OclValid-def)
    apply(rule foundation21[THEN fun-cong, simplified StrongEq-def,simplified, THEN iffD1, of - - false])
    by simp
  show ?thesis
    apply(rule ext, rename-tac $\tau$)
    apply(case-tac $\tau \models (\nu x)$)
    apply(subst cp-OclIncludes, simp add: cp-OclIncludes[symmetric] $A$)
done
qued

lemma OclIncludes-charn1:
  assumes def-X: $\tau \models (\delta X)$
  assumes val-x: $\tau \models (\nu x)$
  shows $\tau \models (X \rightarrow \text{including}(x) \rightarrow \text{includes}(x))$
proof -
  have $C : [[\text{insert} (\tau \, x) \, [[\text{Rep-Set-0} (X \, \tau)]]]] \in \{X. X = \text{bot} \lor X = \text{null} \lor (\forall x \in [[X]]. x \neq \text{bot})\}$
    by(insert def-X val-x, frule Set-inv-lemma, simp add: foundation18 invalid-def)
  show ?thesis
    apply(subst OclIncludes-def, simp add: foundation10[simplified OclValid-def] OclValid-def def-X[simplified OclValid-def] val-x[simplified OclValid-def])
apply(simp add: OclIncluding-def def-X[simplified OclValid-def] val-x[simplified OclValid-def]
  Abs-Set-0-inverse[OF C] true-def)
done
qed

lemma OclIncludes-charn2:
assumes def-X::τ ::(∀X. X. (X ≜ (X ≜ X)) τ)
and val-x::τ ::(∀X. X. (X ≜ X)) τ
and val-y::τ ::(∀X. X. (X ≜ X)) τ
and neq ::(∀X. X. (X ≜ X)) τ
shows τ ::(∀X. X. (X ≜ X)) τ
proof –
  have C ::(∀X. X. (X ≜ X)) τ
    by(insert def-X val-x, frule Set-inv-lemma, simp add: foundation18 invalid-def)
  show ?thesis
  apply(subst OclIncludes-def,
    simp add: def-X[simplified OclValid-def] val-x[simplified OclValid-def]
    val-y[simplified OclValid-def] foundation10[simplified OclValid-def]
    OclValid-def StrongEq-def)
apply(simp add: OclIncluding-def OclIncludes-def def-X[simplified OclValid-def]
by(metis foundation22 foundation6 foundation9 neq)
qed

Here is again a generic theorem similar as above.

lemma OclIncludes-execute-generic:
assumes strict1: (x ≜ invalid) = invalid
and strict2: (invalid ≜ y) = invalid
and cp-StrictRefEq: ∀X. (X ≜ Y. (X ≜ Y) τ) = (((λX. X) ≜ Y) τ)
and StrictRefEq-vs-StrongEq: ∀X. (X ≜ Y) τ.
                               τ ≜ (∀X. X ≜ Y τ)
shows (X ≜ Y) τ ≜ (∀X. X ≜ Y τ)
proof –
  have A: ∀τ. τ ::(∀X. X ≜ Y) τ
    by(erule StrongEq-L-subst2-rev,simp)
  have B: ∀τ. τ ::(∀X. X ≜ Y) τ
    by(erule StrongEq-L-subst2-rev,simp)
  note [simp] = cp-StrictRefEq [THEN allI[THEN allI[THEN allI[THEN cpI2]], of

100
have C: \( \forall \tau.\ (x \triangleq \text{invalid}) \implies (X \triangleright \text{including}(x) \triangleright \text{includes}(y))\)  
\(\tau = \begin{cases} \text{true} & \text{if } x \triangleq y \text{ then true else } X \triangleright \text{includes}(y) \end{cases} \)  
apply(rule foundation22[THEN iffD1])  
apply(erule StrongEq-L-subst2-rev,simp,simp)  
by (simp add: strict2)

have D: \( \forall \tau.\ (y \triangleq \text{invalid}) \implies (X \triangleright \text{including}(x) \triangleright \text{includes}(y))\)  
\(\tau = \begin{cases} \text{true} & \text{if } x \triangleq y \text{ then true else } X \triangleright \text{includes}(y) \end{cases} \)  
apply(rule foundation22[THEN iffD1])  
apply(erule StrongEq-L-subst2-rev,simp,simp)  
by (simp add: strict1)

have E: \( \forall \tau.\ (v \triangleright \text{includes}(y)) \implies (v \triangleright \text{includes}(y) \triangleright \text{includes}(y))\)  
\(\tau = \begin{cases} \text{true} & \text{if } x \neq y \text{ then true else } X \triangleright \text{includes}(y) \end{cases} \)  
apply(rule foundation22[THEN iffD1])  
by(erule StrongEq-L-subst2-rev,simp, simp)

show ?thesis
apply(rule ext, rename-tac \(\tau\))
apply(case-tac ~ (\(\tau = (\delta \ X)\)), simp add: def-split-local,elim disjE A B)
apply(case-tac ~ (\(\tau = (v \ x)\)),  
simp add: foundation18 foundation22[symmetric],  
drule StrongEq-L-sym)
apply(simp add: foundation22 C)
apply(case-tac ~ (\(\tau = (v \ y)\)),  
simp add: foundation18 foundation22[symmetric],  
drule StrongEq-L-sym, simp add: foundation22 D, simp)
apply(subst E,simp-all)
apply(case-tac \(\tau = \text{not}(x \triangleq y)\))
apply(simp add: OclIncludes-charn2[simplified foundation22])
apply(simp add: foundation9 F
  
OclIncludes-charn1[THEN foundation13[THEN iffD2],  
  THEN foundation22[THEN iffD1]])

done
qed

schematic-lemma OclIncludes-execute\_Integer[simp, code-unfold]: ?X

by(rule OclIncludes-execute-generic[OF StrictRefEq\_Integer-strict1 StrictRefEq\_Integer-strict2  
cp-StrictRefEq\_Integer  
StrictRefEq\_Integer-vs-StrongEq], simp-all)

101
schematic-lemma OclIncludes-execute\textsubscript{Boolean}[simp, code-unfold]: ?X
by(rule OclIncludes-execute-generic[OF StrictRefEq\textsubscript{Boolean}-strict1 StrictRefEq\textsubscript{Boolean}-strict2 cp-StrictRefEq\textsubscript{Boolean}
StrictRefEq\textsubscript{Boolean}-vs-StrongEq], simp-all)

schematic-lemma OclIncludes-execute\textsubscript{Set}[simp, code-unfold]: ?X
by(rule OclIncludes-execute-generic[OF StrictRefEq\textsubscript{Set}-strict1 StrictRefEq\textsubscript{Set}-strict2 cp-StrictRefEq\textsubscript{Set}
StrictRefEq\textsubscript{Set}-vs-StrongEq], simp-all)

lemma OclIncludes-including-generic :
assumes OclIncludes-execute-generic [simp] : \(\forall X \ x \ y. (X \rightarrow \text{including}(x::(\text{\text{null}})) \rightarrow \text{includes}(y)) =
\) (if \(\delta \ X\) then if \(x \simeq y\) then true else \(X \rightarrow \text{includes}(y)\) endif else invalid endif)
and StrictRefEq-strict'': \(\forall X \ x \ y. \delta ((x::(\text{\text{null}})) \simeq y) = (v(x) \text{ and } v(y))\)
and a-val : \(\forall \tau. \tau \models v a\)
and x-val : \(\forall \tau. \tau \models v x\)
and S-incl : \(\tau \models (S) \rightarrow \text{includes}(x)\)
shows \(\tau \models S \rightarrow \text{including}(a::(\text{\text{null}})) \rightarrow \text{includes}(x)\)
proof –
have discr-eq-bot1-true : \(\forall \tau. (\bot \tau \models \text{true}) = \text{False}\)
by (metis OCL-core.\text{bot-fun-def foundation1 foundation18 valid3})
have discr-eq-bot2-true : \(\forall \tau. (\bot \models \text{true}) = \text{False}\)
by (metis bot-fun-def discr-eq-bot1-true)
have discr-neq-invalid-true : \(\forall \tau. (\text{invalid} \ \tau \neq \text{true}) = \text{True}\)
by (metis discr-eq-bot2-true invalid-def)
have discr-eq-invalid-true : \(\forall \tau. (\text{invalid} \ \tau \models \text{true}) = \text{False}\)
by (metis bot-option-def invalid-def option.simps(2) true-def)
show ?thesis
apply(simp)
apply(subgoal-tac \(\tau \models \delta \ S\))
preferring 2
apply(insert S-incl[simplified OclIncludes-def], simp add : OclValid-def)
apply(metis discr-eq-bot2-true)
apply(simp add : cp-OclIf[\(\delta \ S\) OclValid-def OclIf-def x-val[simplified OclValid-def] discr-neq-invalid-true discr-eq-invalid-true]
by (metis OclValid-def S-incl StrictRefEq\textsubscript{Set}-strict'' a-val foundation10 foundation6 x-val)
qed

lemmas OclIncludes-including\textsubscript{Integer} =
OclIncludes-including-generic[O\ OclIncludes-execute\textsubscript{Integer} StrictRefEq\textsubscript{Integer}-strict'']

4.6.4. OclExcludes

4.6.5. OclSize

lemma [simp, code-unfold]: \(\text{Set}() \rightarrow \text{size()} = 0\)
apply \textit{rule ext}
apply \textit{(simp add: defined-def mSet-def OclSize-def bot-Set-0-def bot-fun-def null-Set-0-def null-fun-def)}
apply \textit{(subst Abs-Set-0-inject, simp-all add: bot-option-def null-option-def)} +
by \textit{(simp add: Abs-Set-0-inverse bot-option-def null-option-def OclInt0-def)}

lemma \textit{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}() \ + \ \text{if } X \rightarrow \text{includes}(x) \text{ then } 0 \text{ else } 1 \text{ endif} \)
\text{else invalid endif)}

proof ~

have \textit{valid-inject-true} \[\land P. (v P) \tau \neq \text{true } \tau \equiv (v P) \tau = \text{false } \tau\]
apply \textit{(simp add: valid-def true-def false-def bot-fun-def bot-option-def null-fun-def null-option-def)}
by \textit{(case-tac P \tau = \bot, simp-all add: true-def)}

have \textit{defined-inject-true} \[\land P. (\delta P) \tau \neq \text{true } \tau \equiv (\delta P) \tau = \text{false } \tau\]
apply \textit{(simp add: defined-def true-def false-def bot-fun-def bot-option-def null-fun-def null-option-def)}
by \textit{(case-tac P \tau = \bot \lor P \tau = \text{null, simp-all add: true-def)}

show \textit{?thesis}
apply \textit{(rule ext, rename-tac \tau)}

proof ~

have \textit{includes-notin} \[\tau \mid X \rightarrow \text{includes}(x) \equiv (\delta X) \tau = \text{true } \tau \wedge (v x) \tau = \text{true } \tau \equiv x \tau \notin \{\text{Rep-Set-0}(X \tau)\}\]
by \textit{(simp add: Ocincludes-def OclValid-def true-def)}

have \textit{includes-def} \[\tau \mid X \rightarrow \text{includes}(x) \equiv \tau \mid \delta X\]
by \textit{(metis OCL-core.bot-fun-def Ocincludes-def OclValid-def defined3 foundation16)}

have \textit{includes-val} \[\tau \mid X \rightarrow \text{includes}(x) \equiv \tau \mid v x\]
by \textit{(metis (hide-lams, no-types) foundation6 Ocincludes-valid-args-valid\prime Ocincluding-valid-args-valid Ocincluding-valid-args-valid\prime\prime)}

have \textit{ins-in-Set-0} \[\tau \mid \delta X \equiv \tau \mid v x \equiv \{\text{insert}(x \tau)\text{[[Rep-Set-0}(X \tau)\text{]]}\} \in \{X. X = \bot \vee X = \text{null} \vee (\forall x \in \{X\}. x \neq \bot)\}\]
apply \textit{(simp add: bot-option-def null-option-def)}
by \textit{(metis (hide-lams, no-types) Set-inv-lemma foundation18\prime foundation5)}

show \[X \rightarrow \text{including}(x) \rightarrow \text{size}() \tau = (\text{if } \delta X \text{ and } v x \text{ then } X \rightarrow \text{size}() \ + \ \text{if } X \rightarrow \text{includes}(x) \text{ then } 0 \text{ else } 1 \text{ endif} \tau \text{ else invalid endif })\]
apply \textit{(case-tac \tau \mid \delta X \text{ and } v x, simp)}
apply \textit{(subst cp-OclAdd_{\text{Integer}})}
apply (case-tac \(\tau\) \(\models X \rightarrow \text{includes}(x)\), simp add: cp-OclAdd\_integer[symmetric])
apply (case-tac \(\tau\) \(\models (v \ (X \rightarrow \text{size}()) \land \neg (\delta \ (X \rightarrow \text{size}()))\), simp)
apply (drule foundation5[where \(P = v \ X \rightarrow \text{size}()\), erule conjE])
apply (drule OclSize\_infinite)
apply (frule includes\_def, drule includes\_val, simp)
apply (metis (hide-lams, no-types) invalid\_def)
apply (subst OclIf\_false', metis (hide-lams, no-types) defined5 defined6 defined\_and\_I defined\_not\_I foundation1 foundation9)
apply (drule foundation5)
apply (rule conjI)
apply (simp add: OclIncluding\_def)
apply (subtr Abs\_Set\_0\_inverse OF ins\_in\_Set\_0, fast+)
apply (drule Finite\_Set.card.insert[where \(x = x \ \tau\)], simp)
apply (rule includes\_notin, simp, simp)
apply (rule conjI)
apply (simp add: OclIncluding\_def)
apply (subtr Abs\_Set\_0\_inverse OF ins\_in\_Set\_0, fast+)
apply (subtr (asm) (2 3) OclValid\_def, simp add: OclAdd\_integer\_def OclInt1\_def)
apply (rule conjI)
apply (drule Finite\_Set.card.insert[where \(x = x \ \tau\)], simp)
apply (rule includes\_notin, simp, simp)
apply (rule conjI)
apply (simp add: OclIncluding\_def)
apply (subtr Abs\_Set\_0\_inverse OF ins\_in\_Set\_0, fast+)
apply (subtr OclSize\_def, subst OclIncluding\_finite\_rep\_set, assumption+)
apply (metis (hide-lams, no-types) invalid\_def)
apply (subst OclIf\_false', subst foundation9, metis (hide-lams, no-types) OclIncludes\_valid\_args\_valid', simp, simp add: OclSize\_def)
apply (drule foundation5)
apply (rule conjI)
apply (drule OclIncluding\_finite\_rep\_set, fast+)
apply (rule OclIncluding\_finite\_rep\_set, fast+)
apply (drule OclIncluding\_finite\_rep\_set, fast+,, simp add: OclValid\_def)
apply (drule OclIncluding\_finite\_rep\_set, fast+)
apply (drule OclIncluding\_finite\_rep\_set, fast+)
apply (auto)
apply (subtr OclSize\_def, subst OclIncluding\_finite\_rep\_set, assumption+)
apply (metis (hide-lams, no-types) invalid\_def OclSize\_invalid)
by (metis OclIsEmpty foundation18' OclIncluding\_valid\_args\_valid'' invalid\_def OclSize\_invalid)
qed

4.6.6. OclIsEmpty

lemma [simp, code-unfold]: \(\text{Set}\{} \rightarrow \text{isEmpty}() = \text{true}\)
by (simp add: OclIsEmpty\_def)

lemma OclIsEmpty\_including [simp]:
assumes X-def: \(\tau\) \(\models \delta \ X\)
and X-finite: \(\text{finite} \ [[\text{Rep\_Set\_0} \ (X \ \tau)]]\)
and a-val: \(\tau\) \(\models v \ a\)
shows \(X \rightarrow \text{including}(a) \rightarrow \text{isEmpty}() \ \tau = \text{false} \ \tau\)
proof -
have A1 : \(\forall X. \ X \ \tau = \text{true} \ \tau \lor X \ \tau = \text{false} \ \tau \Rightarrow (X \ \text{and not} \ X) \ \tau = \text{false} \ \tau\)
by (metis (no-types) OclAnd-false OclAnd-idem OclImplies-def OclNot3 OclNot-not OclOr-false cp-OclAnd cp-OclNot deMorgan1 deMorgan2)

have defined-inject-true : \( \tau \ P \). (\( \delta P \) \( \tau \neq \) true \( \tau = \) false \( \tau = \) \( (\delta P) \) \( \tau = \) false \( \tau = \))
apply(simp add: defined-def true-def false-def bot-fun-def bot-option-def null-fun-def null-option-def)
by (case-tac \( P \) \( \tau = \) \( \bot \lor P \) \( \tau = \) \( \) null, simp-all add: true-def)

have \( B : \forall \tau. \tau \models v X \Rightarrow X \neq 0 \Rightarrow(X \not= 0) \tau = false \tau \)
by (metis OclAnd-true2 OclValid-def Sem-def foundation16 foundation22 valid4 StrictRefEq_Integer StrictRefEq_Integer-strict StrictRefEq_Integer-strict' StrictRefEq_Integer-strict'' StrongEq-sym bool-split invalid-def null-fun-def null-non-OclInt0)

show \( \) \( \) ?thesis \( \)
apply(simp add: OclIsEmpty-def del: OclSize-including-exec)
apply(subst cp-OclOr, subst A1)
apply(metis (hide-lams, no-types) defined-inject-true OclExcluding-valid-args-valid')
apply(simp add: cp-OclOr[symmetric] def: OclSize-including-exec)
apply(rule B,
  rule foundation20,
  metis (hide-lams, no-types) OclIncluding-defined-args-valid OclIncluding-finite-rep-set
  X-def X-finite a-val size-defined')
apply(simp add: OclSize-def OclIncluding-finite-rep-set[OF X-def a-val] X-finite OclInt0-def)
by (metis OclValid-def X-def a-val foundation10 foundation6 OclIncluding-notempty-rep-set[OF X-def a-val])

qed

4.6.7. OclNotEmpty

lemma [simp, code-unfold]: \( \) \( Set()\rightarrow\) notEmpty() = false
by(simp add: OclNotEmpty-def)

lemma OclNotEmpty-including [simp, code-unfold]:
assumes X-def: \( \tau \models \delta X \)
  and X-finite: finite [[Rep-Set-0 (X \( \tau \))]]
  and a-val: \( \tau \models v a \)
shows X \rightarrow including(a) \rightarrow notEmpty() \( \tau = true \tau \)
apply(simp add: OclNotEmpty-def)
apply(subst cp-OclNot, subst OclIsEmpty-including, simp-all add: assms)
by (metis OclNot4 cp-OclNot)

4.6.8. OclANY

lemma [simp, code-unfold]: \( \) \( Set()\rightarrow\) any() = null
by(rule ext, simp add: OclANY-def, simp add: false-def true-def)

lemma OclANY-singleton-exec[simp, code-unfold]:
  \( \) \( (Set() \rightarrow\) including(a) \rightarrow any() = a \)
apply(rule ext, rename-tac \( \tau \), simp add: mtSet-def OclANY-def)
apply (case-tac $\tau \models v \ a$)
apply (simp add: OclValid-def mtSet-defined [simplified mtSet-def]
  mtSet-valid [simplified mtSet-def] mtSet-rep-set [simplified mtSet-def])
apply (subst (1 2) cp-OclAnd,
  subst (1 2) OclNotEmpty-including [where $X = \text{Set\{\}}$, simplified mtSet-def])
apply (metis (hide-lams, no-types) finite.emptyI mtSet-def mtSet-rep-set)
apply (simp add: OclValid-def)
apply (simp add: OclIncluding-def)
apply (rule conjI)
apply (simp)
apply (subst (1 2 3 4) Abs-Set-0-inverse,
  simp add: bot-option-def null-option-def)
apply (simp add: false-def true-def)
by (simp add: cp-OclAnd [symmetric], rule impI)

4.6.9. OclForall

lemma OclForall-mtSet-exec [simp, code-unfold] : 
  $\text{((}\text{Set\{\}}\rightarrow\text{forall}(z \ | \ P(z))) = \text{true} \\rightarrow \text{forall} \ (z \ | \ P(z))$
apply (simp add: OclForall-def)
apply (subgoal-tac a $\tau = \bot$
prefer 2
apply (simp add: OclValid-def valid-def bot-fun-def split: split-if-asm)
apply (simp add: OclValid-def foundation18')
apply (simp)
apply (simp add: mtSet-defined [simplified mtSet-def])
apply (subgoal-tac)

proof –
have $cp : \forall. \ x \ x \tau = P (\lambda. \ x \tau)$
by (insert $cp0$, auto simp: cp-def)

have insert-in-Set-0 : $\forall. (\tau = (\delta \ S)) \rightarrow (\tau = (v \ x)) \rightarrow$
  $\text{[[insert (x \tau) [[Rep-Set-0 (S \tau)]]]] \in \{X. X = \bot \lor X = \text{null} \lor (\forall x \in [[X]]. x \neq \bot)\}$
by (frule Set-inv-lemma, simp add: foundation18 invalid-def)

have forall-including-invert : $\forall. (f \ x \tau = f (\lambda. \ x \tau) \tau) \rightarrow$
  $\tau = (\delta \ S and v \ x) \rightarrow$
  $(\forall x \in [[Rep-Set-0 (S \rightarrow \text{including}(x) \tau)]]. f (\lambda. \ x \tau) =$
(f x τ ∧ (∀x∈[[Rep-Set-0 (S τ)]] f (∤-x) τ))

apply(drule foundation5, simp add: OclIncluding-def)
apply(subst Abs-Set-0-inverse)
apply(rule insert-in-Set-0, fast+)
apply(set_tac [])

by(simp add: OclValid-def)

have exists-including-invert : ∃τ f. (f x τ = f (∤-x) τ) →
  τ ⊨ (δ S and v x) →
  (∃x∈[[Rep-Set-0 (S→including(x) τ)]] f (∤-x) τ) =
  (f x τ ∨ (∃x∈[[Rep-Set-0 (S τ)]] f (∤-x) τ))

apply(subst arg-cong[where f = λx. ¬x], OF forall-including-invert[where f = λx τ. ¬(f x τ)], simplified)

by simp-all

have cp-eq : ∃τ v. (P x τ = v) = (P (∤-x) τ = v) by(subst cp, simp)
have cp-OclNot-eq : ∃τ v. (P x τ ≠ v) = (P (∤-x) τ ≠ v) by(subst cp, simp)

have foundation10: ∃τ x y. (τ ⊨ x) ∧ (τ ⊨ y) → τ ⊨ (x and y)
apply(erule conjE, subst foundation6)
by((rule foundation6)?, simp)+

have contradict-Rep-Set-0: ∃τ S f.
  (∀x∈[[Rep-Set-0 S]] f (∤-x) τ →
  (∃x∈[[Rep-Set-0 S]] ¬(f (∤-x) τ)) = False
by(case-tac (∀x∈[[Rep-Set-0 S]] ¬(f (∤-x) τ)) = True, simp-all)

show thesis
apply(rule ext, rename-tac τ)
apply(simp add: OclIf-def)
apply(simp add: cp-defined[of δ S and v x] cp-defined[THEN sym])
apply(intro conjI impl)

apply(subgoal_tac τ ⊨ δ S)
prefer 2
  apply(drule foundation5[simplified OclValid-def], erule conjE)+ apply(simp add: OclValid-def)

apply(subst OclForall-def)
apply(simp add: cp-OclAnd[THEN sym] OclValid-def
  foundation10[where x = δ S and y = v x, simplified OclValid-def])

apply(subgoal-tac τ ⊨ (δ S and v x))
prefer 2
apply(simp add: OclValid-def)

apply(case-tac ∃x∈[[Rep-Set-0 (S→including(x) τ)]] P (∤-x) τ = false τ, simp-all)
apply(subst contradict-Rep-Set-0[where f1 = λ x τ. P x τ = false τ], simp)+
apply(simp add: exists-including-invert[where f = λ x τ. P x τ = false τ, OF cp-eq])

apply(simp add: cp-OclAnd[of P x])
apply(erule disjE)
apply(simp only: cp-OclAnd[symmetric], simp)
apply(subgoal-tac OclForall S P τ = false τ)
apply(simp only: cp-OclAnd[symmetric], simp)
apply(simp add: OclForall-def)
apply(subgoal-tac OclForall S P τ ≠ false τ)
apply(simp only: cp-OclAnd[symmetric], simp)
apply(simp add: OclForall-def true-def false-def null-fun-def null-option-def bot-fun-def bot-option-def)
apply(subgoal-tac OclForall S P τ = bot τ)
apply(simp only: cp-OclAnd[symmetric], simp)
apply(simp add: OclForall-def true-def false-def null-fun-def null-option-def bot-fun-def bot-option-def)
apply(simp add: forall-including-invert[where f = λ x τ. P x τ ≠ false τ, OF cp-OclNot-eq], erule conjE)

apply(case-tac ∃x∈[[Rep-Set-0 (S→including(x) τ)]]]. P (λ.- x) τ = bot τ, simp-all)
apply(subst contradict-Rep-Set-0[where f1 = λ x τ. P x τ = bot τ], simp)+
apply(simp add: exists-including-invert[where f = λ x τ. P x τ = bot τ, OF cp-eq])
apply(simp add: cp-OclAnd[of P x])
apply(erule disjE)
apply(subgoal-tac OclForall S P τ ≠ false τ ∧ OclForall S P τ ≠ bot τ)
apply(simp only: cp-OclAnd[symmetric], simp)
apply(simp add: OclForall-def true-def false-def
null-fun-def null-option-def bot-fun-def bot-option-def)

apply(subgoal-tac OclForall S P τ = null τ)
apply(simp only: cp-OclAnd[symmetric], simp)
apply(simp add: OclForall-def true-def false-def
null-fun-def null-option-def bot-fun-def bot-option-def)

apply(simp add: forall-including-invert[where f = λ x τ. P x τ ≠ null τ, OF cp-OclNot-eq],
erule conjE)

apply(simp add: cp-OclAnd[of P x] OclForall-def)
apply(subgoal-tac P x τ = true τ, simp)
apply(metis bot-fun-def bool-split foundation18' foundation2 valid1)

by(metis OclForall-def OclIncluding-defined-args-valid' invalid-def)
qed

4.6.10. OclExists

lemma OclExists-mtSet-exec[simp, code-unfold] : 
((Set{}) −> exists(z | P(z))) = false
by(simp add: OclExists-def)

lemma OclExists-including-exec[simp, code-unfold] :
assumes cp: cp P
shows ((S −> including(x)) −> exists(z | P(z))) = (if δ S and υ x
then P x or (S −> exists(z | P(z)))
else invalid
endif)
by(simp add: OclExists-def OclOr-def OclForall-including-exec cp OclNot-inject)

4.6.11. OclIterate

lemma OclIterate-empty[simp, code-unfold] : ((Set{}) −> iterate(a; x = A | P a x)) = A
proof —
have C : ∃ τ. (δ (λτ. Abs-Set-0 [[{}]]) τ = true τ
by (metis (no-types) defined-def mtSet-def mtSet-defined null-fun-def)
show ?thesis
  apply(simp add: OclIterate-def mtSet-def Abs-Set-0-inverse valid-def C)
  apply(rule ext, rename-tac τ)
  apply(case-tac A τ = ⊥ τ, simp-all, simp add:true-def false-def bot-fun-def)
  apply(simp add: Abs-Set-0-inverse)
done
qed

In particular, this does hold for A = null.
lemma OclIterate-including:
assumes S-finite: \( \tau \models \delta(S \rightarrow \text{size}()) \)
and F-valid-arg: \((v A) \tau = (v (F \ a) A) \ tau \)
and F-commute: \( \text{comp-fun-commute} F \)
and F-cp: \( \forall x \ y \ \tau. \ F \ x \ y \ \tau = F \ (\lambda \cdot. x \ \tau) \ y \ \tau \)
shows \((S \rightarrow \text{including}(a)) \rightarrow \text{iterate}(a; x = A | F \ a \ x) \tau =
((S \rightarrow \text{excluding}(a)) \rightarrow \text{iterate}(a; x = F \ a \ A | F \ a \ x)) \tau \)
proof
have insert-in-Set-0 : \( \forall x. (\tau \modelsto \delta(S)) \Rightarrow (\tau \modelsto (v \ a)) \Rightarrow
[[\text{insert} \ (a \ \tau) \ [[\text{Rep-Set-0} \ (S \ \tau)]]]] \in \{X. X = \text{bot} \lor X = \text{null} \lor (\forall x \in [[[X]]], x \neq \text{bot})\} \)
apply(subst defined-def)
apply(simp add: bot-fun-def)
by(subst Abs-Set-0-inject,
rule insert-in-Set-0, simp-all add: null-option-def bot-option-def)+

have remove-finite : finite \( [[\text{Rep-Set-0} \ (S \ \tau)]] \Rightarrow
finite ((\lambda \ a. \ \tau. \ a) \ (([[\text{Rep-Set-0} \ (S \ \tau)]] - \{a \ \tau\})) \)
by(simp)

have remove-in-Set-0 : \( \forall x. (\tau \modelsto \delta(S)) \Rightarrow (\tau \modelsto (v \ a)) \Rightarrow
[[[[\text{Rep-Set-0} \ (S \ \tau)]] - \{a \ \tau\}]] \in \{X. X = \text{bot} \lor X = \text{null} \lor (\forall x \in [[[X]]], x \neq \text{bot})\} \)
by(frule Set-inv-lemma, simp add: foundation18 invalid-def)

have remove-defined : \( \forall x. (\tau \modelsto \delta(S)) \Rightarrow (\tau \modelsto (v \ a)) \Rightarrow
(\delta (\lambda-x. \ Abs-Set-0 [[[\text{Rep-Set-0} \ (S \ \tau)]] - \{a \ \tau\}]]) \tau = \text{true} \)
apply(subst defined-def)
apply(simp add: bot-fun-def)
by(subst Abs-Set-0-inject,
rule remove-in-Set-0, simp-all add: null-option-def bot-option-def)+

have abs-rep: \( \forall x. [[[x]]] \in \{X. X = \text{bot} \lor X = \text{null} \lor (\forall x \in [[[X]]], x \neq \text{bot})\} \Rightarrow
[[[\text{Rep-Set-0} \ (Abs-Set-0 [[[x]]])]]] = x \)
by(subst Abs-Set-0-inverse, simp-all)

have inject : inj \( (\lambda\ a. \ \tau. \ a) \)
by(rule inj-fun, simp)

show \( ?\text{thesis} \)
apply(subst (1 2) cp-OclIterate, subst OclIncluding-def, subst OclExcluding-def)
apply(case-tac \((\delta S) \ \tau = \text{true}\ \land (v \ a) \ \tau = \text{true} \ \tau), \ \text{simp})
apply(subgoal-tac OclIterate \((\lambda-x. \ \bot) A F \ \tau = OclIterate \ (\lambda-x. \ \bot) (F \ a \ A) F \ \tau, \ \text{simp})
apply(rule conjI, blast+)
apply(simp add: OclIterate-def defined-def bot-option-def bot-fun-def false-def true-def)
apply(simp add: OclIterate-def)
apply((subst abs-rep)[OF insert-in-Set-0[simplified OclValid-def], of τ], simp-all)+,
(subst abs-rep)[OF remove-in-Set-0[simplified OclValid-def], of τ], simp-all)+,
(subst insert-defined, simp-all add: OclValid-def)+,
(subst remove-defined, simp-all add: OclValid-def)+)

apply(case-tac ¬ ((v A) τ = true τ), (simp add: F-valid-arg)+)
apply(rule impl, subst Finite-Set.comp-fun-commute.fold-fun-left-comm[ symmetric, OF F-commute],
rule remove-finite, simp)
apply(subst image-set-diff[OF inject], simp)
apply(subgoal-tac Finite-set.fold F A (insert (λτ'. a τ) ((λa τ. a) ' [[Rep-Set-0 (S τ)]])) τ
= F (λτ'. a τ) (Finite-set.fold F A ((λa τ. a) ' [[Rep-Set-0 (S τ)]] − {λτ'. a τ})) τ)
apply(subst F-cp, simp)

by(subst Finite-set.comp-fun-commute.fold-insert-remove[OF F-commute], simp+)
qed

4.6.12. OclSelect

lemma OclSelect-mtSet-exec[simp,code-unfold]: OclSelect mtSet P = mtSet
apply(rule ext, rename-tac τ)
apply(simp add: OclSelect-def mtSet-def defined-def false-def true-def
bot-Set-0-def bot-fun-def null-Set-0-def null-fun-def)
by(( subst (1 2 3 4 5) Abs-Set-0-inverse
| subst Abs-Set-0-inject), (simp add: null-option-def bot-option-def)+)

definition OclSelect-body :: - ⇒ - ⇒ (ʼA, 'a option option) Set
≡ (λP x acc. if P x = false then acc else acc−>including(x) endif)

lemma OclSelect-including-exec[simp,code-unfold]:
assumes P-cp : cp P
shows OclSelect (X−>including(y)) P = OclSelect-body P y (OclSelect (X−>excluding(y)) P)
(is _ = ?select)
proof –
have P-cp: ∃x τ. P x τ = P (λ_. x τ) τ
  by(insert P-cp, auto simp: cp-def)

have ex-including : ∃f X y τ. τ = δ X −> τ = v y −>
(∃x∈[[Rep-Set-0 (X−>including(y) τ)]]; f (P (λ_. x)) τ) =
(f (P (λ_. y τ)) τ ∨ (∃x∈[[Rep-Set-0 (X τ)]]; f (P (λ_. x)) τ))
apply(simp add: OclIncluding-def OclValid-def)
apply(subst Abs-Set-0-inverse, simp, (rule disjI2)+)
apply(metis (hide-lams, no-types) OclValid-def Set-inv-lemma foundation18)
by(simp)

have al-including : ∃f X y τ. τ = δ X −> τ = v y −>

111
(∀ x∈[[Rep-Set-0 (X→including(y)) τ]]]. f (P (λ· x)) τ) ⇒
(f (P (λ· y) τ) τ ≡ (∀ x∈[[Rep-Set-0 (X τ)]]]. f (P (λ· x) τ))
apply(simp add: OclIncluding-def OclValid-def)
apply(subst Abs-Set-0-inverse, simp, (rule disjI2)+)
  apply (metis (hide-lams, no-types) OclValid-def Set-inv-lemma foundation18′)
by(simp)

have ex-excluding1 : (∃ x∈[[Rep-Set-0 (X → excluding1 y) τ]]]. f (P (λ· x) τ) τ =
(f (P (λ· y) τ) τ ≡ (∀ x∈[[Rep-Set-0 (X τ)]]]. f (P (λ· x) τ)) τ)
apply(simp add: OclIncluding-def OclValid-def)
apply(subst Abs-Set-0-inverse, simp, (rule disjI2)+)
  apply (metis (no-types) Diff-iff OclValid-def Set-inv-lemma)
by(auto)

have al-excluding1 : (∀ x∈[[Rep-Set-0 (X → excluding1 y) τ]]]. f (P (λ· x) τ) τ =
(∀ x∈[[Rep-Set-0 (X τ)]]]. f (P (λ· x) τ)) τ)
apply(simp add: OclIncluding-def OclValid-def)
apply(subst Abs-Set-0-inverse, simp, (rule disjI2)+)
  apply (metis (no-types) Diff-iff OclValid-def Set-inv-lemma)
by(auto)

have in-including : (∀ x∈[[Rep-Set-0 (X → including(y)) τ]]]. f (P (λ· x) τ) τ =
{x∈[[Rep-Set-0 (X → including(y)) τ]]]. f (P (λ· x) τ)}
(let s = {x∈[[Rep-Set-0 (X τ)]]]. f (P (λ· x) τ) in
if f (P (λ· y) τ) then insert (y τ) s else s)
apply(simp add: OclIncluding-def OclValid-def)
apply(subst Abs-Set-0-inverse, simp, (rule disjI2)+)
  apply (metis (hide-lams, no-types) OclValid-def Set-inv-lemma foundation18′)
by(simp add: Let-def, auto)

let ?OclSet = λS. [[S]] ∈ \{X. X = ⊥ ∨ X = null ∨ (∀ x∈[[X]]). x ≠ ⊥\}
have diff-in-Set-0 : (∀ τ. (δ X) τ = true τ =⇒
?OclSet ([[[Rep-Set-0 (X τ)]]] \ {y τ})
apply(simp, (rule disjI2)+)

by (metis (mono-tags) Diff-iff OclValid-def Set-inv-lemma)

have ins-in-Set-0 : (∀ τ. (δ X) τ = true τ =⇒ (v y) τ = true τ =⇒
?OclSet (insert (y τ) \{x∈[[Rep-Set-0 (X τ)]]]. P (λ· x) τ ≠ false τ})
apply(simp, (rule disjI2)+)

by (metis (hide-lams, no-types) OclValid-def Set-inv-lemma foundation18′)

have ins-in-Set-0′ : (∀ τ. (δ X) τ = true τ =⇒ (v y) τ = true τ =⇒
?OclSet (insert (y τ) \{x∈[[Rep-Set-0 (X τ)]]]. x ≠ y τ ∧ P (λ· x) τ ≠ false τ})
apply(simp, (rule disjI2)+)

by (metis (hide-lams, no-types) OclValid-def Set-inv-lemma foundation18′)

have ins-in-Set-0″ : (∀ τ. (δ X) τ = true τ =⇒
?OclSet \{x∈[[Rep-Set-0 (X τ)]]]. P (λ· x) τ ≠ false τ}
apply(simp, (rule disjI2)+)

by (metis (hide-lams, no-types) OclValid-def Set-inv-lemma foundation18′)

have ins-in-Set-0″′ : (∀ τ. (δ X) τ = true τ =⇒
?OclSet \{x∈[[Rep-Set-0 (X τ)]]]. x ≠ y τ ∧ P (λ· x) τ ≠ false τ}
apply(simp, (rule disjI2)+)

112
by (metis (hide-lams, no-types) OclValid-def Set-inv-lemma foundation18')

have if-same : \( \forall a \ b \ c \ d \ t. \ t = \delta \ a \Rightarrow b \ t = d \ t \Rightarrow c \ t = d \ t \Rightarrow \\
(\text{if} \ a \ \text{then} \ b \ \text{else} \ c \ \text{endif}) \ t = d \ t \\
\text{by}\{\text{simp add: OclIf-def OclValid-def}\}

have invert-including : \( \forall P \ y \ t. \ P \ t = \bot \Rightarrow P \rightarrow \text{including}(y) \ t = \bot \\
\text{by}\{\text{metis (hide-lams, no-types) foundation17 foundation18'} \text{ OclIncluding-valid-args-valid}\}

have exclude-defined : \( \forall \tau. \ t = \delta \ X \Rightarrow \\
(\delta \ (\lambda. \ Abs-Set-0[[[[[\text{Rep-Set-0 (X \ t)]}, x \neq y \ t \wedge P (\lambda \cdot x) \ t \neq false \ t]]]])) \ t = true \tau \\
\text{apply}(\text{subst defined-def},
\text{simp add: false-def true-def bot-Set-0-def bot-fun-def null-Set-0-def null-fun-def})
\text{by}(\text{subst Abs-Set-0-inject}[\text{OF ins-in-Set-0''[simplified false-def]},
(\text{simp add: OclValid-def bot-option-def null-option-def})]

have if-eq : \( \forall A \ B \ t. \ t = \nu x \Rightarrow \tau \Rightarrow (\text{if} \ x \ \text{then} \ A \ \text{else} \ B \ \text{endif}) \ t \\
\text{by}(\text{subst cp-OclIf, simp add: cp-OclIf[symmetric] true-def})

have OclSelect-body-bot : \( \forall \tau. \ t = \delta \ X \Rightarrow t = \nu y \Rightarrow P \ y \ t \neq \bot \Rightarrow \\
\exists x \in [[\text{Rep-Set-0 (X \ t)]]. P (\lambda \cdot x) \ t = \bot \Rightarrow \bot = \text{?select} \ t \\
\text{apply}(\text{drule ex-excluding1}[\text{where} X2 = X \ \text{and} \ y2 = y \ \text{and} \ f2 = \lambda x \ t. \ x \ t = \bot],
(\text{simp add: P-cp[ symmetric]+})
\text{apply}(\text{subgoal-tac t = (bot = ?select), simp add: OclValid-def StrongEq-def true-def bot-fun-def})
\text{apply}(\text{simp add: OclSelect-body-def})
\text{apply}(\text{subst StrongEq-def subst3[OF if-eq], simp, metis foundation18'})
\text{apply}(\text{simp add: OclValid-def, subst StrongEq-def, subst true-def, simp})
\text{apply}(\text{subgoal-tac } \exists x \in [[\text{Rep-Set-0 (X \rightarrow \text{excluding}(y) \ t)]]. P (\lambda \cdot x) \ t = \bot \ t)
\text{prefer 2}
\text{apply (metis OCL-core.bot-fun-def foundation18')}
\text{apply (subst if-same[where d5 = \bot])}
\text{apply (metis defined7 transform1)}
\text{apply (simp add: OclSelect-def bot-option-def bot-fun-def})
\text{apply (subst invert-including)}
\text{by}(\text{simp add: OclSelect-def bot-option-def bot-fun-def})

have d-and-v-inject : \( \forall t X y. (\delta X \ \text{and} \ v \ y) \ t \neq true \ t \Rightarrow (\delta X \ \text{and} \ v \ y) \ t = false \ t \\
\text{by}\{\text{metis bool-split defined5 defined6 defined-and-I foundation16 transform1}
\text{invalid-def null-fun-def})

have OclSelect-body-bot' : \( \forall \tau. (\delta X \ \text{and} \ v \ y) \ t \neq true \ t \Rightarrow \bot = \text{?select} \ t \\
\text{apply(drude d-and-v-inject)}
\text{apply(simp add: OclSelect-def OclSelect-body-def})
\text{apply(subst cp-OclIf, subst cp-OclIncluding, simp add: false-def true-def})
\text{apply(subst cp-OclIf[symmetric], subst cp-OclIncluding[symmetric])}
by (metis (lifting, no-types) OclIf-def foundation18 foundation18' invert-including)

have conj-split2 : \(a \land b \land c \tau\). ((\(a \triangleq \text{false}\) \(\tau = \text{false} \rightarrow b\)) \land ((\(a \triangleq \text{false}\) \(\tau = \text{true} \rightarrow c\))) \rightarrow ((\(a \neq \text{false}\) \(\tau \rightarrow b\)) \land (a \neq \text{false} \rightarrow c))
by (metis OclValid-def defined7 foundation14 foundation22 foundation9)

have defined-inject-true : \(\forall \tau. \delta \ P \neq \text{true} \rightarrow \tau = \text{false} \rightarrow (\delta \ P = \text{false} \rightarrow b) \land ((\delta \ P = \text{true} \rightarrow c)) = \Rightarrow (((\delta \ P = \text{false} \rightarrow b) \land (a \neq \text{false} \rightarrow c)))
by (case-tac \(P \tau = \bot \lor P \tau = \text{null}\), simp-all add: true-def)

have cp-OclSelect-body : \(\forall \tau. ?\ select \tau = \text{OclSelect-body} P y \ (\\lambda. \text{OclSelect} X \rightarrow \text{excluding}(y) \ P \tau)))
apply(simp add: OclSelect-body-def)
by(rule ext, simp add: OclSelect-body-def OclIf-def)

have bool-invalid: \(\forall (x::(\forall X \rightarrow \text{Boolean}) \ y \ \tau. \neg (\tau |\!|= v x) \rightarrow \tau |\!|= (x \triangleq y) \triangleq \text{invalid}\)
by(simp add: StrictRefEq Boolean OclValid-def StrongEq-def true-def)

have conj-comm : \(\forall p q r. (p \land q \land r) = ((p \land q) \land r)\)
by blast

show \(?\ thesis\)
apply(rule ext, rename-tac \(\tau\))
apply(subst OclSelect-def)
apply(case-tac (\(\delta X \rightarrow \text{excluding}(y)\)) \(\tau = \text{true} \rightarrow \), simp)
apply((subst ex-including
| subst in-including),
metis OclValid-def foundation5,
metis OclValid-def foundation5+)
apply(simp add: Let-def)
apply(subst (4) false-def, subst (4) bot-fun-def, simp add: bot-option-def P-cp[ symmetric])
apply(case-tac \(\neg (\tau |\!|= (v P y))\))
apply(subgoal-tac \(P y \tau \neq \text{false} \rightarrow \))
prefer 2
apply (metis (hide-lams, no-types) foundation1 foundation18' valid4)
apply(simp)
apply(subst conj-comm, rule conjI)
apply(drule-tac y11 = false in bool-invalid)
apply(simp only: OclSelect-body-def,
metis OclIf-def OclValid-def defined-def foundation2 foundation22
bot-fun-def invalid-def)

114
apply (drule foundation5 [simplified OclValid-def], subst al-including [simplified OclValid-def], simp, simp)
apply (simp add: P-cp [symmetric])
apply (metis OCL-core.bot-fun-def foundation18')
apply (simp add: foundation18' bot-fun-def OclSelect-body-bot OclSelect-body-bot')
apply (subst (1 2 3 4) cp-OclIf, subst (1 2 3 4) foundation18' [THEN iffD2, simplified OclValid-def], simp, simp only: cp-OclIf[ symmetric] refl if-True)
apply (subgoal-tac P (λ x. y τ) τ = false τ)
prefer 2
apply (drule foundation5 [simplified OclValid-def], simp)
apply (simp add: foundation10 [simplified OclValid-def])
apply (rule equalityI)
apply (rule subsetI, simp, metis)
apply (rule subsetI, simp)
apply (drule defined-inject-true)
apply (subgoal-tac \tau (\tau = \delta X) \lor \neg (\tau = \nu y))
prefer 2
apply (metis bot-fun-def OclValid-def foundation18' OclIncluding-defined-args-valid valid-def)
apply (simp add: OclValid-def false-def true-def, rule conjI, blast)
apply (simp add: OclSelect-invalid[simplified invalid-def], rule OclSelect-including-exec)
done
qed

4.6.13. OclReject

lemma OclReject-mtSet-exec[simp, code-unfold]: OclReject mtSet P = mtSet
by (simp add: OclReject-def)

lemma OclReject-including-exec[simp, code-unfold]:
assumes P: cp P
shows OclReject (X >\rightarrow including(y)) P = OclSelect-body (not o P) y (OclReject (X >\rightarrow excluding(y)) P)
apply (simp add: OclReject-def comp-def, rule OclSelect-including-exec)
by (metis assms cp-intro '5')

4.7. Execution on Set’s Operators (higher composition)

4.7.1. OclIncludes

lemma OclIncludes-any[simp, code-unfold]:
  X >\rightarrow includes(X >\rightarrow any()) = (if \delta X then
  if \delta (X >\rightarrow size()) then not(X >\rightarrow isEmpty())
  else X >\rightarrow includes(null) endif
  else invalid endif)
proof
  have defined-inject-true: \forall \tau P. (\delta P) \tau \neq true \tau \Longrightarrow (\delta P) \tau = false \tau
  apply (simp add: defined-def true-def false-def bot-fun-def bot-option-def
  null-fun-def null-option-def)
  by (case-tac P \tau = \bot \lor P \tau = null, simp-all add: true-def)

  have valid-inject-true: \forall \tau P. (\nu P) \tau \neq true \tau \Longrightarrow (\nu P) \tau = false \tau
  apply (simp add: valid-def true-def false-def bot-fun-def bot-option-def
  null-fun-def null-option-def)
  by (case-tac P \tau = \bot, simp-all add: true-def)

  have notempty': \forall X. \tau \vdash \delta X \Longrightarrow finite [[Rep-Set-0 (X \tau)]] \Longrightarrow not (X >\rightarrow isEmpty()) \tau
  \neq true \tau \Longrightarrow X \tau = Set{\} \tau
  apply (case-tac X \tau, rename-tac X', simp add: mtSet-def Abs-Set-0-inject)
  apply (erule disjE, metis (hide-lams, no-types) bot-Set-0-def bot-option-def foundation17)
  apply (erule disjE, metis (hide-lams, no-types) bot-option-def foundation17)}
null-Set-0-def null-option-def foundation17)
apply(case-tac X', simp, metis (hide-lams, no-types) bot-Set-0-def foundation17)
apply(rename-tac X'', case-tac X''', simp)
apply (metis (hide-lams, no-types) foundation17 null-Set-0-def)
apply(simp add: OclIsEmpty-def OclSize-def)
apply(subst (asm) cp-OclNot, subst (asm) cp-OclOr, subst (asm) cp-StrictRefEqInteger,
  subst (asm) cp-OclAnd, subst (asm) cp-OclNot)
apply(simp only: OclValid-def foundation20[simplified OclValid-def]
  cp-OclNot[symmetric] cp-OclAnd[symmetric] cp-OclOr[symmetric])
apply(simp add: Abs-Set-0-inverse split: split-if-asm)
by(simp add: true-def OclInt0-def OclNot-def StrictRefEq-def)

have B: \( \forall X \tau. \neg \text{finite} \left[ [[\text{Rep-Set-0} (X \tau)]] \right] \implies \left( \delta \left( X \rightarrow \text{size}(\tau) \right) \right) \tau = \text{false} \tau \)
apply(subst cp-defined)
apply(simp add: OclSize-def)
by (metis OCL-core.bot-fun-def defined-def)

show thesis
apply(rule ext, rename-tac \( \tau \), simp only: OclIncludes-def OclANY-def)
apply(subst cp-OclIf, subst (2) cp-valid)
apply(case-tac \( \delta X \tau = \text{true} \tau \),
  simp only: foundation20[simplified OclValid-def] cp-OclIf[symmetric], simp,
  subst (1 2) cp-OclAnd, simp add: cp-OclAnd[symmetric])
apply(case-tac finite \( \left[ [[\text{Rep-Set-0} (X \tau)]] \right] \))
apply(frule size-defined THEN iffD2, simplified OclValid-def, assumption)
apply(subst (1 2 3 4) cp-OclIf, simp)
apply(subst (1 2 3 4) cp-OclIf[symmetric], simp)
apply(case-tac \( X \rightarrow \text{notEmpty}(\tau) \) \( \tau = \text{true} \tau \), simp)
apply(frule OclNotEmpty-has-elt[simplified OclValid-def], simp)
apply(simp add: OclNotEmpty-def cp-OclIf[symmetric])
apply(subgoal-tac (SOME y. y \in [[\text{Rep-Set-0} (X \tau)]])) \( \in [[\text{Rep-Set-0} (X \tau)]] \), simp add: true-def)
apply(metis OclValid-def Set-inv-lemma foundation18' null-option-def true-def)
apply(rule somel-ex, simp)
apply(simp add: OclNotEmpty-def cp-valid[symmetric])
apply(subgoal-tac \( \neg (\tau \in \left[ [[\text{Rep-Set-0} (X \tau)]] \right) \), simp)
apply(subst OclIsEmpty-def, simp add: OclSize-def)
apply(subst cp-OclNot, subst cp-OclOr, subst cp-StrictRefEqInteger, subst cp-OclAnd,
  subst cp-OclNot, simp add: OclValid-def foundation20[simplified OclValid-def]
  cp-OclNot[symmetric] cp-OclAnd[symmetric] cp-OclOr[symmetric])
apply(frule notempty[simplified OclValid-def],
  (simp add: mtSet-def Abs-Set-0-inverse OclInt0-def false-def)+)
apply(drule notempty[simplified OclValid-def], simp, simp)
apply (metis (hide-lams, no-types) empty-iff mtSet-rep-set)
apply(frule B)
apply(subst (1 2 3 4) cp-OclIf, simp)
apply(subst (1 2 3 4) cp-OclIf[symmetric], simp)
apply(case-tac \( X \rightarrow \text{notEmpty}(\tau) \) \( \tau = \text{true} \tau \), simp)
apply (frule OclNotEmpty-has-elt[simplified OclValid-def], simp)
apply (simp add: OclNotEmpty-def OclIsEmpty-def)
apply (subgoal-tac X -->size() τ = ⊥)
prefer 2
apply (metis (hide-lams, no-types) OclSize-def)
apply (subt (asm) cp-OclNot, subst (asm) cp-OclOr, subst (asm) cp-StrictRefEq\textsubscript{Integer},
    subst (asm) cp-OclAnd, subst (asm) cp-OclNot)
apply (simp add: OclValid-def foundation20[simplified OclValid-def]
    cp-OclNot[symmetric] cp-OclAnd[symmetric] cp-OclOr[symmetric])
apply (simp add: OclNot-def StrongEq-def StrictRefEq\textsubscript{Integer} valid-def false-def true-def
    bot-option-def bot-fun-def invalid-def)

apply (metis OCL-core.bot-fun-def null-fun-def null-is-valid valid-def)
by (drule defined-inject-true,
    simp add: false-def true-def OclIf-false[simplified false-def] invalid-def)

qed

4.7.2. OclSize

lemma [simp,code-unfold]: δ (Set{} -->size()) = true
by simp

lemma [simp,code-unfold]: δ ((X -->including(x)) -->size()) = (δ(X -->size())) and v(x))
proof –
have defined-inject-true : ∀τ P. (δ P) τ ≠ true τ ⟷ (δ P) τ = false τ
    apply (simp add: defined-def true-def false-def bot-fun-def bot-option-def
    null-fun-def null-option-def)
    by (case-tac P τ = ⊥ ∨ P τ = null, simp-all add: true-def)

have valid-inject-true : ∀τ P. (v P) τ ≠ true τ ⟷ (v P) τ = false τ
    apply (simp add: valid-def true-def false-def bot-fun-def bot-option-def
    null-fun-def null-option-def)
    by (case-tac P τ = ⊥, simp-all add: true-def)

have OclIncluding-finite-rep-set : ∀τ. (δ X and v x) τ = true τ ⟷
    finite [[Rep-Set-0 (X -->including(x) τ)]] = finite [[Rep-Set-0 (X τ)]]
apply (rule OclIncluding-finite-rep-set)
by (metis OclValid-def foundation5+)

have card-including-exec : ∃τ. (δ (λx. ⌈⌈int (card [[Rep-Set-0 (X -->including(x) τ)]]))⌉⌉) τ =
    (δ (λx. ⌈⌈int (card [[Rep-Set-0 (X τ)]]))⌉⌉) τ
by (simp add: defined-def bot-fun-def bot-option-def null-fun-def null-option-def)

show ?thesis
apply (rule ext, rename-tac τ)
apply (case-tac (δ (X -->including(x) -->size()))) τ = true τ, simp del: OclSize-including-exec)
apply (subt cp-OclAnd, subst cp-defined, simp only: cp-defined[of
X \rightarrow \text{including}(x) \rightarrow \text{size}()]

\text{simp add: OclSize-def}
\text{apply(case-tac (δ X and v x) τ = true τ ∧ finite [[\text{Rep-Set-0} (X \rightarrow \text{including}(x) τ)]]), simp}
\text{apply(erule conjE},
\text{simp add: OclIncluding-finite-rep-set[simplified OclValid-def] card-including-exec}
cp-OclAnd[of δ X v x]
cp-OclAnd[of true, THEN sym])
\text{apply(subgoal-tac (δ X) τ = true τ ∧ (v x) τ = true τ, simp)}
\text{apply(rule foundation5[of : δ X v x, simplified OclValid-def],}
simp only: cp-OclAnd[THEN sym])
\text{apply(simp, simp add: defined-def true-def false-def bot-fun-def bot-option-def)}
\text{apply(drule defined-inject-true[of X \rightarrow \text{including}(x) \rightarrow \text{size}()],}
simp del: OclSize-including-exec,
simp only: cp-OclAnd[of δ (X \rightarrow \text{size}()) v x],
simp add: cp-defined[of X \rightarrow \text{including}(x) \rightarrow \text{size}()] cp-defined[of X \rightarrow \text{size}()]
del: OclSize-including-exec,
simp add: OclSize-def card-including-exec
del: OclSize-including-exec)
\text{apply(case-tac (δ X and v x) τ = true τ ∧ finite [[\text{Rep-Set-0} (X τ)]]],}
simp add: OclIncluding-finite-rep-set[simplified OclValid-def] card-including-exec,
simp only: cp-OclAnd[THEN sym],
simp add: defined-def bot-fun-def)
\text{apply(split split-if_asm)
apply(simp add: OclIncluding-finite-rep-set[simplified OclValid-def] card-including-exec) +
apply(simp only: cp-OclAnd[THEN sym], simp, rule implI, erule conjE)
apply(case-tac (v x) τ = true τ, simp add: cp-OclAnd[of δ X v x])
by(drule valid-inject-true[of x], simp add: cp-OclAnd[of - v x])
qed

\text{lemma [simp, code-unfold]: δ ((X \rightarrow \text{excluding}(x)) \rightarrow \text{size}()) = (δ(X \rightarrow \text{size}()) and v(x))

\text{proof –
have defined-inject-true : \(\bigwedge P. (δ P) τ \neq true τ \implies (δ P) τ = false τ\)
apply(simp add: defined-def true-def false-def bot-fun-def bot-option-def
null-fun-def null-option-def)
by (case-tac P τ = ⊥ ∨ P τ = null, simp-all add: true-def)

have valid-inject-true : \(\bigwedge P. (v P) τ \neq true τ \implies (v P) τ = false τ\)
apply(simp add: valid-def true-def false-def bot-fun-def bot-option-def
null-fun-def null-option-def)
by (case-tac P τ = ⊥, simp-all add: true-def)

have OclExcluding-finite-rep-set : \(\bigwedge τ. (δ X and v x) τ = true τ \implies finite [[\text{Rep-Set-0} (X \rightarrow \text{excluding}(x) τ)]] = finite [[\text{Rep-Set-0} (X τ)]]\)
apply(rule OclExcluding-finite-rep-set)
by(metis OclValid-def foundation5) +

119
have \( \text{card-excluding-exec} : \forall \tau. (\delta (\lambda . [\lfloor \text{int (card [[\text{Rep-Set-0} (X \rightarrow \text{excluding}(x) \tau]])]]) \, \tau) = \\
(\delta (\lambda . [\lfloor \text{int (card [[\text{Rep-Set-0} (X \tau)])]]) \, \tau) \) \\
by (\text{simp add: defined-def bot-fun-def bot-option-def null-fun-def null-option-def})

show \( \text{thesis} \) \\
apply (\text{rule ext, rename-tac \( \tau \)}) \\
apply (\text{case-tac} (\delta (X \rightarrow \text{excluding}(x) \rightarrow \text{size}()) \, \tau) = \text{true} \, \tau, \text{simp}) \\
\quad apply (\text{subt cop-OclAnd}, \text{subst \text{cp-defined, simp only: \text{cp-defined}}[\text{of X \rightarrow \text{excluding}(x) \rightarrow \text{size}()]}, \\
\quad \text{simp add: OclSize-def}) \\
apply (\text{case-tac} ((\delta X \text{ and } v \, x) \, \tau = \text{true} \, \tau \land \text{finite} [[\text{Rep-Set-0} (X \rightarrow \text{excluding}(x) \tau)], \\
\text{simp}) \\
\quad apply (\text{erule conjE}, \\
\quad \text{simp add: OclExcluding-finite-rep-set[simplified OclValid-def] card-excluding-exec} \\
\quad \text{cp-OclAnd}[\text{of \delta X \, v \, x}] \\
\quad \text{cp-OclAnd}[\text{of true, THEN sym}]) \\
apply (\text{subgoal-tac} (\delta X) \, \tau = \text{true} \, \tau \land (v \, x) \, \tau = \text{true} \, \tau, \text{simp}) \\
apply (\text{rule foundations5}[\text{of - \delta X v x, simplified OclValid-def}], \\
\text{simp only: \text{cp-OclAnd}[THEN sym}}) \\
apply (\text{simp, simp add: defined-def true-def false-def bot-fun-def bot-option-def}) \\
apply (\text{drule defined-inject-true}[\text{of X \rightarrow \text{excluding}(x) \rightarrow \text{size}()]), \\
\text{simp,} \\
\text{simp only: cp-OclAnd}[\text{of \delta (X \rightarrow \text{size}()) \, v \, x}], \\
\text{simp add: cp-defined}[\text{of X \rightarrow \text{excluding}(x) \rightarrow \text{size}() \, \text{of X \rightarrow \text{size}()}], \\
\text{simp add: OclSize-def card-excluding-exec}) \\
apply (\text{case-tac} (\delta X \text{ and } v \, x) \, \tau = \text{true} \, \tau \land \text{finite} [[\text{Rep-Set-0} (X \tau)]], \\
\text{simp add: OclExcluding-finite-rep-set[simplified OclValid-def] card-excluding-exec,} \\
\text{simp only: cp-OclAnd}[\text{THEN sym}], \\
\text{simp add: defined-def bot-fun-def}) \\
apply (\text{split split-if-asn}) \\
apply (\text{simp add: OclExcluding-finite-rep-set[simplified OclValid-def] card-excluding-exec}+) \\
apply (\text{simp only: cp-OclAnd}[\text{THEN sym}], \text{simp impI, erule conjE}) \\
apply (\text{case-tac} (v \, x) \, \tau = \text{true} \, \tau, \text{simp add: cp-OclAnd}[\text{of \delta X \, v \, x}]) \\
by (\text{drule valid-inject-true}[\text{of x}], \text{simp add: cp-OclAnd}[\text{of - v \, x}]) \\
\text{qed}

lemma [simp]: \\
\text{assumes X-finite: } \forall \tau. \text{finite} [[\text{Rep-Set-0} (X \tau)]] \\
\text{shows } \delta ((X \rightarrow \text{including}(x) \rightarrow \text{size}()) = (\delta(X) \text{ and } v(x)) \\
by (\text{simp add: size-defined[OF X-finite del: OclSize-including-exec})

4.7.3. OclForall

lemma OclForall-rep-set-false: \\
\text{assumes } \tau \models \delta \, X
shows \((\text{OclForall } X \ P \ \tau = \text{false } \tau) = (\exists x \in [[[\text{Rep-Set-0 } (X \ \tau)]]]. \ P (\lambda x. \ \tau) = \text{false } \tau)\)

by(insert assms, simp add: OclForall-def OclValid-def false-def true-def
          bot-fun-def bot-option-def null-fun-def null-option-def)

lemma OclForall-rep-set-true:

assumes \(\tau \models \delta X\)

shows \(\tau \models OclForall X P\) = (\(\forall x \in [[[\text{Rep-Set-0 } (X \ \tau)]]]. \ \tau \models \ P (\lambda x. \ \tau)\))

proof –

have destroy-ocl: \(\forall x. \ \tau \models x = \text{true } \tau \lor x = \text{false } \tau \lor x = \bot \tau\)

apply(case-tac x) apply (metis bot-Boolean-def)

apply(rename-tac x', case-tac x'') apply (metis null-Boolean-def)

apply(rename-tac x'', case-tac x'') apply (metis (full-types) true-def)

by (metis (full-types) false-def)

have disjE4 : \(\forall P1 P2 P3 P4 R.\)

\((P1 \lor P2 \lor P3 \lor P4) \Longrightarrow (P1 \Longrightarrow R) \Longrightarrow (P2 \Longrightarrow R) \Longrightarrow (P3 \Longrightarrow R) \Longrightarrow (P4 \Longrightarrow R)\) 

\(\Longrightarrow R\)

by metis

show \(?\text{thesis}\)

apply(simp add: OclForall-def OclValid-def true-def false-def
          bot-fun-def bot-option-def null-fun-def null-option-def split: split-if-asm)

apply(rule conjI, rule implI) apply (metis OCL-core.drop.simps option.distinct(1))

apply(rule implI, rule conjI, rule implI) apply (metis option.distinct(1))

apply(rule implI, rule conjI, rule implI) apply (metis OCL-core.drop.simps)

apply(intro conjI impl ballI)

proof –

fix x show \(\forall \ x \ \exists x \in [[[\text{Rep-Set-0 } (X \ \tau)]]]. \ P (\lambda x. \ \tau) \neq \text{None}\) \(\Longrightarrow\)

\(\forall \ x \ \exists x \in [[[\text{Rep-Set-0 } (X \ \tau)]]]. \ \exists y. \ P (\lambda x. \ \tau) = \text{[y]}\) \(\Longrightarrow\)

\(\forall \ x \ \exists x \in [[[\text{Rep-Set-0 } (X \ \tau)]]]. \ P (\lambda x. \ \tau) \neq [\text{False}]\) \(\Longrightarrow\)

\(x \in [[[\text{Rep-Set-0 } (X \ \tau)]]] \Longrightarrow P (\lambda x. \ \tau) = [\text{True}]\)

apply(erule-tac x = x in ballE)+

by(rule disjE4[OF destroy-ocl[of P (\lambda x. \ \tau)]],

(simp add: true-def false-def null-fun-def null-option-def bot-fun-def bot-option-def)+)

apply-end(simp add: assms[simplified OclValid-def true-def])+

qed

qed

lemma OclForall-includes :

assumes \(\delta \ x : \tau \models \delta \ x\)

and \(\gamma \ x : \tau \models \delta \ y\)

shows \(\tau \models \text{OclForall } x (\text{OclIncludes } y) = [[[\text{Rep-Set-0 } (x \ \tau)]]] \subseteq [[[\text{Rep-Set-0 } (y \ \tau)]]]\)

apply(simp add: OclForall-rep-set-true[OF \(\delta \ x\), OF \(\gamma \ x\)],

simp add: OclIncludes-def OclValid-def \(\gamma \ x\)[simplified OclValid-def])

apply(insert Set-inv-lemma[OF \(\delta \ x\), simp add: valid-def false-def true-def bot-fun-def]

by(rule iffI, simp add: subsetI, simp add: subsetD)

lemma OclForall-not-includes :

assumes \(\delta \ x : \tau \models \delta \ x\)

and \(\gamma \ x : \tau \models \delta \ y\)

shows \(\text{OclForall } x (\text{OclIncludes } y) \ \tau = \text{false } \tau) = (\neg [[[\text{Rep-Set-0 } (x \ \tau)]]] \subseteq [[[\text{Rep-Set-0 } (y \ \tau)]]] \models \text{false } \tau)\)

by rule
lemma OclForall-iterate:
assumes S-finite: finite [[Rep-Set-0 (S τ)]]
shows S->forall(x | P x) τ = (S->iterate(x; acc = true | acc and P x)) τ
proof –
have and-comm : comp-fun-commute (λx acc. acc and P x)
apply(simp add: comp-fun-commute-def comp-def)
by (metis OclAnd-assoc OclAnd-commute)

have ex-insert : \(\forall x F. (\exists x\in insert x F. P x) = (P x \vee (\exists x\in F. P x))\)
by (metis insert_iff)

have destruct-ocl : \(\forall x. x = \text{true} \vee x = \text{false} \vee x = \text{null} \vee x = \bot \)
apply(case-tac x) apply (metis bot-Boolean-def)
apply(rename-tac x', case-tac x') apply (metis null-Boolean-def)
apply(rename-tac x'', case-tac x''') apply (metis (full-types) true-def)
by (metis (full-types) false-def)

have disjE4 : \(\forall P1 P2 P3 P4 R. (P1 \vee P2 \vee P3 \vee P4) \implies (P1 \implies R) \implies (P2 \implies R) \implies (P3 \implies R) \implies (P4 \implies R) \implies R\)
by metis

let ?P-eq = \(\lambda x b \tau. P (\lambda x. \tau) = b \tau\)
let ?P = \(\lambda set b \tau. \exists x\in set. ?P-eq x b \tau\)
let ?if = \(\lambda f b c. \text{if } f b \tau \text{ then } b \tau \text{ else } c\)
let ?forall = \(\lambda P. ?if P \text{ false } (\text{?if } P \bot (\text{?if } P \text{ null } (\text{true } \tau)))\)
show ?thesis
apply(simp only: OclForall-def OclIterate-def)
apply(case-tac τ = δ S, simp only: OclValid-def)
apply(subgoal-tac let set = \([\text{Rep-Set-0} (S τ)]\) in
?forall (?P set) =
Finite-Set.fold (λx acc. acc and P x) true ((\lambda a τ. a) \ ' set) τ,
simp only: Let-def, simp add: S-finite, simp only: Let-def)
apply(case-tac \([\text{Rep-Set-0} (S τ)]\) = \{}, simp)
apply(rule finite-ne-induct[OF S-finite], simp)
apply(simp only: image-insert)
apply(subst comp-fun-commute.fold-insert[OF and-comm], simp)
apply(metis empty-if image-empty)
apply(simp)
apply (metis OCL-core.bot-fun-def destruct-ocl null-fun-def)
apply(simp only: image-insert)

122
apply\(\text{subst comp-fun-commute.fold-insert[OF and-comm], simp}\)
apply (metis (mono-tags) imageE)

apply (subst cp-OclAnd) apply (drule sym, drule sym, simp only:, drule sym, simp only:)
apply (simp only: ex-insert)
apply (subgoal-tac \(\exists x. x \in F\)) prefer 2
apply (metis mono-tags imageE)
apply (subst cp-OclAnd)
apply (drule sym, drule sym, simp only:)
apply (simp only:)
apply (simp only:)
apply (subgoal-tac \(\exists x. x \in F\))
apply (metis all-not-in-conv)
proof −
fix \(x F\) show \(\delta S\) \(\tau = \text{true} \tau \implies \exists x. x \in F \implies \tau\)
apply (rule disjE4 [OF destruct-ocl [where \(x1 = P (\lambda \tau. x)\)]])
apply (simp-all add: true-def false-def OclAnd-def
null-fun-def null-option-def bot-fun-def bot-option-def)
by (metis (lifting) option.distinct(1))
qed

lemma OclForall-cong:
assumes \(\forall x. x \in \{\{\text{Rep-Set-0} (X \tau)\}\} \implies \tau \models P (\lambda \tau. x) \implies \tau \models Q (\lambda \tau. x)\)
assumes \(P: \tau \models \text{OclForall} X P\)
shows \(\tau \models \text{OclForall} X Q\)
proof −
have def-X: \(\tau \models \delta X\)
by (insert P, simp add: OclForall-def OclValid-def bot-option-def true-def split: split-if-asm)
show \(?\)thesis
apply (insert P)
apply (subst (asm) OclForall-rep-set-true[Ocl-def-X], subst OclForall-rep-set-true[Ocl-def-X])
by (simp add: assms)
qed

lemma OclForall-cong’:
assumes \(\forall x. x \in \{\{\text{Rep-Set-0} (X \tau)\}\} \implies \tau \models P (\lambda \tau. x) \implies \tau \models Q (\lambda \tau. x) \implies \tau \models R (\lambda \tau. x)\)
assumes \(P: \tau \models \text{OclForall} X P\)
assumes \(Q: \tau \models \text{OclForall} X Q\)
shows \(\tau \models \text{OclForall} X R\)
proof −
have def-X: \(\tau \models \delta X\)
by (insert P, simp add: OclForall-def OclValid-def bot-option-def true-def split: split-if-asm)
show \(?\)thesis
apply (insert P Q)
apply (subst (asm) \(1 2\) OclForall-rep-set-true[Ocl-def-X], subst OclForall-rep-set-true[Ocl-def-X])
by (simp add: assms)
qed
4.7.4. Strict Equality

lemma StrictRefEqSet-defined :
assumes x-def: \( \tau \models \delta \ x \)
assumes y-def: \( \tau \models \delta \ y \)
shows \( ((x:\( A, x::nul\) Set) \models y) \) \( \tau = \\
(\tau \models \forall z. y \rightarrow \in\mathrm{cludes}(z)) \) and \( (\tau \models \forall z. x \rightarrow \in\mathrm{cludes}(z)) \) \( \tau \)
proof 
  have rep-set-inj : \( \forall \tau. (\delta \ x) \tau = \text{true} \tau \rightarrow \\
(\delta \ y) \tau = \text{true} \tau \rightarrow \\
x \tau \neq y \tau \rightarrow \\
\lceil\lceil \text{Rep-Set-0} (y \tau) \rceil \rceil \neq \\
\lceil\lceil \text{Rep-Set-0} (x \tau) \rceil \rceil \\
apply(simp add: defined-def)
apply(split split-if-asn, simp add: false-def true-def)+
apply(simp add: null-fun-def null-Set-0-def bot-fun-def bot-Set-0-def)
apply(case-tac x \tau, rename-tac x'')
apply(case-tac y \tau, rename-tac y'')
apply(simp add: Abs-Set-0-inverse)
by(blast)

show ?thesis
apply(simp add: StrictRefEqSet StrongEq-def
  foundation20[OF x-def, simplified OclValid-def]
  foundation20[OF y-def, simplified OclValid-def])
apply(subgoal-tac \( \lceil x \tau = y \tau \rceil = \text{true} \tau \lor \\
\lceil x \tau = y \tau \rceil = \text{false} \tau \)
prefer 2
apply(simp add: false-def true-def)
apply(crule disjE)
apply(simp add: true-def)
apply(subgoal-tac \( \tau \models \OclForall x (\OclIncludes y)) \land \\
(\tau \models \OclForall y (\OclIncludes x))\)
apply(subst cp-OclAnd, simp add: true-def OclValid-def)
apply(simp add: OclForall-includes[OF x-def y-def]
  OclForall-includes[OF y-def x-def])

apply(simp)
apply(subgoal-tac OclForall x (OclIncludes y) \( \tau = \text{false} \tau \lor \\
\OclForall y (\OclIncludes x) \tau = \text{false} \tau \)
apply(subst cp-OclAnd, metis OclAnd-false1 OclAnd-false2 cp-OclAnd)
apply(simp only: OclForall-not-includes[OF x-def y-def, simplified OclValid-def]

124
lemma StrictRefEqSet-exec[simp,code-unfold] :
((x::(A',α::null)Set) = y)
(if δ x then (if δ y
  then ((x→forAll(z| y→includes(z)) and (y→forAll(z| x→includes(z)))))
  else if δ y
    then false (* x'→includes = null *)
    else invalid
  endif
else if v x (* null = ??? *)
  then if v y then not(δ y) else invalid endif
else invalid
endif)

proof –
have defined-inject-true : (∀ τ P. ¬ (τ |= δ P)) = ((δ P) τ = false τ)
by (metis bot-fun-def OclValid-def defined-def foundation16 null-fun-def)

have valid-inject-true : (∀ τ P. ¬ (τ |= v P)) = ((v P) τ = false τ)
by (metis bot-fun-def OclIf-true' OclIncludes-charn0 OclIncludes-charn0' OclValid-def valid-def
foundation6 foundation9)

show ?thesis
apply (rule ext, rename-tac τ)

apply (simp add: OclIf-def
  defined-inject-true[simplified OclValid-def]
  valid-inject-true[simplified OclValid-def],
  subst false-def, subst true-def, simp)
apply (subst (1 2) cp-OclNot, simp, simp add: cp-OclNot[symmetric])
apply (simp add: StrictRefEqSet-defined[simplified OclValid-def])
by (simp add: StrictRefEqSet StrongEq-def false-def true-def valid-def defined-def)

qed

lemma StrictRefEqSet-L-subst1 : cp P τ |= v x τ |= v y τ |= v P x τ |= v P y
τ |= (x::(A',α::null)Set) = y τ |= (P x ::(A',α::null)Set) = P y

apply (simp only: StrictRefEqSet OclValid-def)
apply (split split-if-asm)
apply (simp add: StrongEq-L-subst1[simplified OclValid-def])
by (simp add: invalid-def bot-option-def true-def)

lemma OclIncluding-cong' :
shows τ |= δ s τ |= δ t τ |= v x
τ |= (s::(A',α::null)Set) τ |= (t->including(x))
proof -
  have cp: cp (λs. (s→including(x)))
  apply(simp add: cp-def, subst cp-OclIncluding)
  by (rule-tac x = (λxab ab. ((λ-, xab)→including(λ-, x ab)) ab) in exI, simp)

  show τ |= δ s ⇒ τ |= δ t ⇒ τ |= v x ⇒ τ |= (s = t) ⇒ ?thesis
  apply(rule-tac P = λs. (s→including(x)) in StrictRefEqSet-L-subst1)
   apply(simp add: foundation20)
   apply(simp add: foundation20)
   apply(simp add: foundation10 foundation6+)
  done
done

lemma OclIncluding-cong : ⋀(s::(′A,′a::null)Set) t x y τ. τ |\= δ t ⇒ τ |\= υ y ⇒ τ |\= s t⇒ \(x = y \Rightarrow \tau |\= s→including(x) \= (t→including(y))
apply(simp only:)
apply(rule OclIncluding-cong', simp-all only:)
by(auto simp: OclValid-def OclIf-def invalid-def bot-option-def OclNot-def split : split-if-asm)

lemma const-StrictRefEq-including : const a ⇒ const S ⇒ const X ⇒
const (X \= S→including(a))
apply(rule const-StrictRefEqSet-including : const a ⇒ const S ⇒ const X ⇒
const (X \= S→including(a))
by(rule constant-OclIncluding)

4.8. Test Statements

lemma syntax-test: Set{2,1} = (Set{}→including(1)→including(2))
by (rule refl)

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} \= null) = (false::(∃)Boolean)
  shows (τ::(∃)st) |\= (Set{Set{2},null}→includes(null))
by(simp add: OclIncludes-executeSet H)

lemma short-cut'[simp,code-unfold]: (8 \= 6) = false
apply(rule ext)
apply(simp add: StrictRefEqIntegral StrongEq-def OclInt8-def OclInt6-def
true-def false-def invalid-def bot-option-def)
done

lemma short-cut''[simp,code-unfold]: (2 \= 1) = false
apply(rule ext)
apply(simp add: StrictRefEqIntegral StrongEq-def OclInt2-def OclInt1-def
true-def false-def invalid-def bot-option-def)
done

126
lemma short-cut'"[simp,code-unfold]: \((1 \vdash 2) = \text{false}\)
apply (rule ext)
apply (simp add: StrictRefEq\_integer StrongEq\_def Ocl\_Int2\_def Ocl\_Int1\_def
true\_def false\_def invalid\_def bot\_option\_def)
done

Elementary computations on Sets.

declare Ocl\_Select\_body\_def [simp]

value \(\neg (\tau \models v(\text{invalid}::(\alpha::\text{null}) \text{Set}))\)
value \(\tau \models v(\text{null}::(\alpha::\text{null}) \text{Set})\)
value \(\neg (\tau \models \delta(\text{null}::(\alpha::\text{null}) \text{Set}))\)
value \(\tau \models v(\text{Set}())\)
value \(\tau \models v(\text{Set}\{\text{null},2\})\)
value \(\tau \models \delta(\text{Set}\{\text{null},2\})\)
value \(\tau \models (\text{Set}\{2,1\} \rightarrow \text{includes}(1))\)
value \(\neg (\tau \models (\text{Set}\{2\} \rightarrow \text{includes}(1)))\)
value \(\neg (\tau \models (\text{Set}\{2,1\} \rightarrow \text{includes}(\text{null})))\)
value \(\tau \models (\text{Set}\{2,\text{null}\} \rightarrow \text{includes}(\text{null}))\)
value \(\tau \models (\text{Set}\{\text{null},2\} \rightarrow \text{includes}(\text{null}))\)

value \(\tau \models ((\text{Set}()) \rightarrow \text{forall}(z \mid 0' < z))\)
value \(\tau \models ((\text{Set}\{2,1\}) \rightarrow \text{forall}(z \mid 0' < z))\)
value \(\neg (\tau \models ((\text{Set}\{2,1\}) \rightarrow \text{exists}(z \mid z' < 0)))\)
value \(\neg (\tau \models \delta(\text{Set}\{2,\text{null}\}) \rightarrow \text{forall}(z \mid 0' < z))\)
value \(\neg (\tau \models ((\text{Set}\{2,\text{null}\}) \rightarrow \text{forall}(z \mid 0' < z)))\)
value \(\tau \models ((\text{Set}\{\text{null},2\}) \rightarrow \text{exists}(z \mid 0' < z))\)
value \(\neg (\tau \models (\text{Set}\{\text{null}::\alpha \text{ Boolean} \} \models \text{Set}())\)
value \(\tau \models (\text{Set}\{\text{null}::\alpha \text{ Integer} \} \models \text{Set}())\)

value \((\tau \models (\text{Set}\{\lambda\cdot [[x]] \} \models \text{Set}\{\lambda\cdot [[x]]\}))\)
value \((\tau \models (\text{Set}\{\lambda\cdot [x] \} \models \text{Set}\{\lambda\cdot [x]]\}))\)

lemma \(\neg (\tau \models (\text{Set}\{\text{true} \} \models \text{Set}\{\text{false}\}))\) by simp
lemma \(\neg (\tau \models (\text{Set}\{\text{true}, \text{true} \} \models \text{Set}\{\text{false}\}))\) by simp
lemma \(\neg (\tau \models (\text{Set}\{2\} \models \text{Set}\{1\}))\) by simp
lemma \(\tau \models (\text{Set}\{2, \text{null},2\} \models \text{Set}\{\text{null},2\})\) by simp
lemma \(\tau \models (\text{Set}\{1, \text{null},2\} \rightarrow \text{Set}\{\text{null},2\})\) by simp
lemma \(\tau \models (\text{Set}\{\text{null}, \text{null}\}) \models \text{Set}\{\text{null},2\})\) by simp
lemma \(\tau \models (\text{Set}\{\text{null},2\} \rightarrow \text{select}(x \mid \text{not} x) \models \text{Set}\{\text{null}\})\) by simp
lemma \(\tau \models (\text{Set}\{\text{null}\} \rightarrow \text{reject}(x \mid \text{not} x) \models \text{Set}\{\text{null}\})\) by simp

lemma \(\text{const} (\text{Set}\{\text{null}, \text{null}, \text{invalid}\})\) by (simp add: const-ss)

127
end
5. Formalization III: State Operations and Objects

theory OCL-state
imports OCL-lib
begin

5.1. 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.

5.1.1. Recall: The Generic Structure of States

Recall the foundational concept of an object id (oid), which is just some infinite set.

type-synonym oid = nat

Further, recall that states are pair of a partial map from oid’s to elements of an object universe \( \mathfrak{A} \)—the heap—and a map to relations of objects. The relations were encoded as lists of pairs to leave the possibility to have Bags, OrderedSets or Sequences as association ends.

This leads to the definitions:

record \( ('\mathfrak{A})\text{state} = \)
\begin{align*}
\text{heap} & :: "oid \rightarrow \mathfrak{A} " \\
\text{assocs}_2 & :: "oid \rightarrow (oid \times oid) \text{ list } " \\
\text{assocs}_3 & :: "oid \rightarrow (oid \times oid \times oid) \text{ list } "
\end{align*}

type-synonym \( ('\mathfrak{A})\text{st} = "\mathfrak{A} \text{ state} \times \mathfrak{A} \text{ state}" \)

Now we refine our state-interface. In certain contexts, we will require that the elements of the object universe have a particular structure; more precisely, we will require that there is a function that reconstructs the oid of an object in the state (we will settle the question how to define this function later).

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:

\[ \text{typ } \mathcal{A} :: \text{object} \]

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

\[ \text{instantiation } \text{option} :: (\text{object})\text{object} \]

begin
\[ \text{definition } \text{oid-of-option-def}: \text{oid-of } x = \text{oid-of } (\text{the } x) \]
\[ \text{instance } .. \]
end

## 5.2. Fundamental Predicates on Object: Strict Equality

### Definition

Generic referential equality - to be used for instantiations with concrete object types ...

\[ \text{definition } \text{StrictRefEqObject} :: (\mathcal{A},'a::\{\text{object},\text{null}\})\text{val} \Rightarrow (\mathcal{A},'a)\text{val} \Rightarrow (\mathcal{A})\text{Boolean} \]

where
\[ \text{StrictRefEqObject} \ x \ y \equiv \lambda \tau. \; \begin{cases} \text{if } (v \ x) \tau = \text{true} \tau \land (v \ y) \tau = \text{true} \tau & \text{then if } x \tau = \text{null} \lor y \tau = \text{null} \text{ then } [[x \tau = \text{null} \land y \tau = \text{null}]] \\ \text{else } [[(\text{oid-of } (x \tau)) = (\text{oid-of } (y \tau)) ]] \\ \text{else invalid } \tau \end{cases} \]

### 5.2.1. Logic and Algebraic Layer on Object

### Validity and Definedness Properties

We derive the usual laws on definedness for (generic) object equality:

\[ \text{lemma } \text{StrictRefEqObject-defargs}: \]
\[ \tau \models (\text{StrictRefEqObject} \ x \ y : (\mathcal{A},'a::\{\text{null},\text{object}\})\text{val}) \Rightarrow (\tau \models (v \ x)) \land (\tau \models (v \ y)) \]
\[ \text{by}(\text{simp add: StrictRefEqObject-def OclValid-def true-def invalid-def bot-option-def split: bool.split-asm HOL.split-if-asm}) \]

### Symmetry

\[ \text{lemma } \text{StrictRefEqObject-sym} : \]
\[ \text{assumes } x\text{-val} : \tau \models v \ x \]
\[ \text{shows } \tau \models \text{StrictRefEqObject} \ x \ x \]
\[ \text{by}(\text{simp add: StrictRefEqObject-def true-def OclValid-def x-val[simplified OclValid-def])} \]

### Execution with Invalid or Null as Argument

\[ \text{lemma } \text{StrictRefEqObject-strict1 [simp,code-unfold]} : \]
\[ (\text{StrictRefEqObject} \ x \ invalid) = \text{invalid} \]
\[ \text{by}(\text{rule ext, simp add: StrictRefEqObject-def true-def false-def}) \]
lemma StrictRefEqObject\_strict2[simp, code-unfold]:
(StrictRefEqObject invalid x) = invalid
by (rule ext, simp add: StrictRefEqObject-def true-def false-def)

Context Passing

lemma cp-StrictRefEqObject:
(StrictRefEqObject x y τ) = (StrictRefEqObject (λ- x τ) (λ- y τ)) τ
by (auto simp: StrictRefEqObject-def cp-valid [symmetric])

lemmas cp-intro' '[intro!, simp, code-unfold] =
cp-intro''
    cp-StrictRefEqObject[THEN allI[THEN allI[THEN allI[THEN cpE2]],
of StrictRefEqObject]]

Behavior vs StrongEq

It remains to clarify the role of the state invariant \( \text{inv}_\sigma(\sigma) \) mentioned above that states the condition that there is a “one-to-one” correspondence between object representations and oid's: \( \forall \text{oid} \in \text{dom} \sigma. \text{oid} = \text{OidOf} \left< \text{oid} \right> \). This condition is also mentioned in [33, Annex A] and goes back to Richters [35]; 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:

\[
\begin{align*}
\text{WFF } \tau &= (\forall x \in \text{ran}(\text{heap}(\text{fst } \tau)). \left< \text{heap}(\text{fst } \tau) \left< \text{oid-of } x \right> \right> = x) \\
&\quad \land \\
&\quad (\forall x \in \text{ran}(\text{heap}(\text{snd } \tau)). \left< \text{heap}(\text{snd } \tau) \left< \text{oid-of } x \right> \right> = x)
\end{align*}
\]

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 [6, 8], 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.

\[
\begin{align*}
\text{theorem } \text{StrictRefEqObject-vs-StrongEq}: \\
\text{assumes } \text{WFF}: \text{WFF } \tau \\
\text{and } \text{valid-x}: \tau \models (\forall x)
\end{align*}
\]
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 \( (\tau \models (\text{StrictRefEq}_{\text{Object}} x y)) = (\tau \models (x \triangleq y)) \)
apply (insert WFF valid-x valid-y x-present-pre y-present-pre x-present-post y-present-post)
apply (auto simp: StrictRefEq_{Object-def} OclValid-def WFF-def StrongEq-def true-def Ball-def)
apply (erule-tac x = x \( \tau \) in allE', simp-all)
done

theorem StrictRefEq_{Object}-vs-StrongEq'::
  assumes WFF: \( \text{WFF} \ \tau \)
  and valid-x: \( \tau \models (v (x :: (\mathcal{A}::\text{object},\alpha::\{\text{null},\text{object}\})\text{val})) \)
  and valid-y: \( \tau \models (v y) \)
  and oid-preserve: \( \forall x. \ x \in \text{ran} (\text{heap}(\text{fst} \ \tau)) \lor x \in \text{ran} (\text{heap}(\text{snd} \ \tau)) \implies H x \neq \bot \implies \text{oid-of} (H x) = \text{oid-of} x \)
  and xy-together: \( x \ \tau \in H \ \forall \ \text{ran} (\text{heap}(\text{fst} \ \tau)) \land y \ \tau \in H \ \forall \ \text{ran} (\text{heap}(\text{snd} \ \tau)) \lor \)
  \( x \ \tau \in H \ \forall \ \text{ran} (\text{heap}(\text{snd} \ \tau)) \land y \ \tau \in H \ \forall \ \text{ran} (\text{heap}(\text{snd} \ \tau)) \)
  shows \( (\tau \models (\text{StrictRefEq}_{\text{Object}} x y)) = (\tau \models (x \triangleq y)) \)
apply (insert WFF valid-x valid-y xy-together)
apply (simp add: WFF-def)
apply (auto simp: StrictRefEq_{Object-def} OclValid-def WFF-def StrongEq-def true-def Ball-def)
by (metis foundation18' oid-preserve valid-x valid-y)+

So, if two object descriptions live in the same state (both pre or post), the referential equality on objects implies in a WFF state the logical equality.

5.3. Operations on Object

5.3.1. Initial States (for testing and code generation)
definition \( \tau_0 :: (\mathcal{A})\text{st} \)
where \( \tau_0 \equiv (\langle \text{heap} = \text{Map.empty}, \text{assocs}_2 = \text{Map.empty}, \text{assocs}_3 = \text{Map.empty} \rangle) \)
  
  (\langle \text{heap} = \text{Map.empty}, \text{assocs}_2 = \text{Map.empty}, \text{assocs}_3 = \text{Map.empty} \rangle)

5.3.2. 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 :: ((\mathcal{A}::\text{object}) \text{st} \Rightarrow \mathcal{A} \text{ state}) \Rightarrow (\mathcal{A}::\text{object} \rightarrow 'a) \Rightarrow (\mathcal{A}, 'a \text{ option option}) \text{Set}
where OclAllInstances-generic \( \text{fst-snd} \ H = \)
  \( (\lambda\tau. \text{Abs-Set-0} [[\text{Some } ((H \ \forall \ \text{ran} (\text{heap } (\text{fst-snd} \ \tau)))) - \{ \text{None } \} ]]) \)

132
lemma OclAllInstances-generic-defined: \( \tau \models \delta \) (OclAllInstances-generic pre-post \( H \))
apply(simp add: defined-def OclValid-def OclAllInstances-generic-def false-def def true-def bot-fun-def bot-Set-0-def null-fun-def null-Set-0-def)
apply(rule conjI)
apply(rule notI, subst (asm) Abs-Set-0-inject, simp, (rule disjI2)+)
apply(asm Abs-Set-0-inject, simp, (rule disjI2)+)
apply(erule contrapos-pn)
apply(subst OCL-lib.Set-0.Abs-Set-0-inverse, auto simp: null-fun-def null-option-def bot-option-def)
done

lemma OclAllInstances-generic-init-empty:
assumes \( \forall x. \text{pre-post}(x, x) = x \)
shows \( \tau_0 \models \text{OclAllInstances-generic pre-post } H \triangleq \text{Set}\{\} \)
by(simp add: StrongEq-def OclAllInstances-generic-def OclValid-def \( \tau_0 \)-def mtSet-def)

lemma represented-generic-objects-nonnull:
assumes \( A: \tau \models ((\text{OclAllInstances-generic pre-post } H::(\forall::\text{object} \rightarrow 'a)) \rightarrow \text{includes}(x)) \)
shows \( \tau \models \neg \text{not}(x \triangleq \text{null}) \)
proof –
  have \( B: \tau \models \delta \) (OclAllInstances-generic pre-post \( H \))
    by(insert A[THEN OCL-core.foundation6, simplified OCL-lib.OclIncludes-defined-args-valid], auto)
  have \( C: \tau \models \nu x \)
    by(insert A[THEN OCL-core.foundation6, simplified OCL-lib.OclIncludes-defined-args-valid], auto)
  show \( ?\text{thesis} \)
  apply(insert A)
apply(simp add: OclValid-def OclNot-def def true-def OclIncludes-def B[simplified OclValid-def]
  C[simplified OclValid-def])
apply(simp add: OclAllInstances-generic-def)
apply(erule contrapos-pm)
apply(subst OCL-lib.Set-0.Abs-Set-0-inverse, auto simp: null-fun-def null-option-def bot-option-def)
done
qed

lemma represented-generic-objects-defined:
assumes \( A: \tau \models ((\text{OclAllInstances-generic pre-post } H::(\forall::\text{object} \rightarrow 'a)) \rightarrow \text{includes}(x)) \)
shows \( \tau \models \delta \) (OclAllInstances-generic pre-post \( H \)) \( \land \tau \models \delta \) \( \nu x \)
apply(insert A[THEN OCL-core.foundation6, simplified OCL-lib.OclIncludes-defined-args-valid])
apply(simp add: OCL-core.foundation16 OCL-core.foundation18 invalid-def, erule conjE)
apply(insert A[THEN represented-generic-objects-nonnull])
by(simp add: foundation24 null-fun-def)

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

lemma represented-generic-objects-in-state:
assumes $A \vdash (\text{OclAllInstances-generic pre-post } H) \rightarrow \text{includes}(x)$

shows $x \pi \in (\text{Some o } H) \cdot \text{ran}(\text{heap}(\text{pre-post } \pi))$

proof –

have $B$: $(\delta (\text{OclAllInstances-generic pre-post } H)) \pi = \text{true}$

by (simp add: OCL-core.OclValid-def[symmetric] OclAllInstances-generic-defined)

have $C$: $(\forall x \pi \tau = \text{true})$

by (insert $A[\text{THEN OCL-core.foundation6, simplified OCL-lib.OclIncludes-defined-args-valid}]$

auto simp: OclValid-def)

have $F$: $\text{Rep-Set-0} (\text{Abs-Set-0} [[\text{Some } \cdot (H \cdot \text{ran}(\text{heap}(\text{pre-post } \pi)) - \{\text{None}\})]]) = [[\text{Some } \cdot (H \cdot \text{ran}(\text{heap}(\text{pre-post } \pi)) - \{\text{None}\})]]$

by (subst OCL-lib.Set-0.Abs-Set-0-inverse, simp add: bot-option-def)

show $\text{thesis}$

apply simp

apply (simp add: OclIncludes-def OclValid-def ran-def B C image-def true-def)

apply (simp add: OclAllInstances-generic-def)

apply (simp add: F)

apply (simp add: ran-def)

by (fastforce)

qed

**Lemma** state-update-vs-allInstances-generic-empty:

assumes $[\text{simp}]: \forall a. \text{pre-post (mk a)} = a$

assumes $\forall x. \sigma' \cdot \text{oid} = \text{Some } x \Rightarrow x = \text{Object}$ and $\text{Type Object \neq None}$

shows $(\text{mk (heap=empty, assocs}_2=A, assocs}_3=B)) \vdash \text{OclAllInstances-generic pre-post Type} \leq \text{Set}\{\}$

proof –

have state-update-vs-allInstances-empty:

$(\text{OclAllInstances-generic pre-post Type}) (\text{mk (heap=empty, assocs}_2=A, assocs}_3=B)) = \text{Set}\{\}$

by (simp add: OclAllInstances-generic-def mtSet-def)

show $\text{thesis}$

apply (simp only: OclValid-def, subst cp-StrictRefEqSet, simp add: state-update-vs-allInstances-empty)

apply (simp add: OclIf-def valid-def mtSet-def defined-def)

by (subst Abs-Set-0-inject, simp add: bot-option-def true-def+)

qed

Here comes a couple of operational rules that allow to infer the value of oclAllInstances from the context $\pi$. These rules are a special-case in the sense that they are the only rules that relate statements with different $\pi$’s. For that reason, new concepts like “constant contexts $P$” are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

**Lemma** state-update-vs-allInstances-generic-including’:

assumes $[\text{simp}]: \forall a. \text{pre-post (mk a)} = a$

assumes $\forall x. \sigma' \cdot \text{oid} = \text{Some } x \Rightarrow x = \text{Object}$ and $\text{Type Object \neq None}$

134
shows \(\text{OclAllInstances-generic pre-post Type}\)
\[
(mk \ (\\heap = \sigma'(\oid \rightarrow \text{Object}), \ \text{assocs}_2 = A, \ \text{assocs}_3 = B)) =
((\text{OclAllInstances-generic pre-post Type}) \rightarrow \text{including}(\lambda -. \ \text{[drop (Type Object)]}))
\]
\[
(mk \ (\\heap = \sigma', \ \text{assocs}_2 = A, \ \text{assocs}_3 = B))
\]
proof

have \(\text{drop-none} : \forall x. \ x \neq \text{None} \implies \text{[}x\text{]} = x\)
by (case-tac \(x\), simp+)

have \(\text{insert-diff} : \forall x S. \ \text{insert} \ [x] \ (S - \{\text{None}\}) = \text{insert} \ [x] \ S - \{\text{None}\}\)
by (metis insert-Diff-if option.distinct(1) singletonE)

show \(?\text{thesis}\)
apply(simp add: OclIncluding-def OclAllInstances-generic-defined[simplified OclValid-def],
simp add: OclAllInstances-generic-def)
apply(subst Abs-Set-0-inverse, simp add: bot-option-def, simp add: comp-def,
subst image-insert[symmetric],
subst drop-none, simp add: assms)
apply(case-tac Type Object, simp add: assms, simp only:,
subgoal-tac \(\sigma\)' oid = \some \ x\),
apply(subgoal-tac \(\sigma\)' oid = \Object\),
apply(rule ran-map-upd, simp)
apply(simp, erule exE, frule assms, simp)
apply(subgoal-tac \Object \in \text{ran} \ \sigma'\) prefer 2
apply(rule ranI, simp)
by (subgoal-tac \forall x. \ x = \Object\)

by (subst insert-absorb, simp, metis fun-apd-apply)
qed

lemma state-update-vs-allInstances-generic-including:
assumes \(\forall a. \text{pre-post} \ (mk \ a) = a\)
assumes \(\forall x. \ \sigma' \ oid = \text{Some} \ x \implies x = \Object\)
and Type Object \neq \text{None}
shows \(\text{OclAllInstances-generic pre-post Type}\)
\[
(mk \ (\\heap = \sigma'(\oid \rightarrow \text{Object}), \ \text{assocs}_2 = A, \ \text{assocs}_3 = B)) =
((\lambda -. \ (\text{OclAllInstances-generic pre-post Type})
\ \text{[drop (Type Object)]}))
\]
\[
(mk \ (\\heap = \sigma', \ \text{assocs}_2 = A, \ \text{assocs}_3 = B))
\]
apply(subst state-update-vs-allInstances-generic-including', (simp add: assms)+,
subst cp-OclIncluding,
simp add: OclIncluding-def)
apply(subst (1 3) cp-defined[symmetric],
simp add: OclAllInstances-generic-defined[simplified OclValid-def])
apply(simp add: defined-def OclValid-def OclAllInstances-generic-def
bot-fun-def null-fun-def bot-Set-0-def null-Set-0-def)
apply (subst (1 3) Abs-Set-0-inject)
by (simp add: bot-option-def null-option-def+)

lemma state-update-vs-allInstances-generic-noincluding':
assumes [simp]: ∃a. pre-post (mk a) = a
assumes \( \forall x. \sigma'(oid = Some x \implies x = Object) \)
and Type Object = None
shows \( OclAllInstances-generic \ pre-post \ Type \)
\( (mk \ (\{heap=\sigma'(oid\rightarrow Object), assocs_2=A, assocs_3=B\})) \)
= \( OclAllInstances-generic \ pre-post \ Type \)
\( (mk \ (\{heap=\sigma', assocs_2=A, assocs_3=B\})) \)
proof –
have drop-none: \( \forall x. x \neq None \implies \lfloor \lceil x \rceil \rfloor = x \)
by (case-tac x, simp+)
have insert-diff: \( \forall x S. insert \lfloor x \rfloor (S - \{None\}) = (insert \lfloor x \rfloor S) - \{None\} \)
by (metis insert-Diff-if option.distinct (1) singletonE)
show ?thesis
apply (simp add: OclIncluding-def OclAllInstances-generic-defined[ simplified OclValid-def]
OclAllInstances-generic-def)
apply (subgoal-tac ran (\sigma'(oid \rightarrow Object))) = insert Object (ran \sigma', simp add: assms)
apply (case-tac \( \exists x. \sigma' oid = Some x \))
apply (rule ran-map-upd, simp)
apply (simp, erule exE, frule assms, simp)
apply (subgoal-tac Object \in ran \sigma') prefer 2
apply (rule ranI, simp)
apply (subst insert-absorb, simp)
by (metis fun-upd-apply)
qed

theorem state-update-vs-allInstances-generic-ntc:
assumes [simp]: \( \exists a. \ pre-post \ (mk a) = a \)
assumes oid-def: \( \exists \sigma' \in \text{dom} \sigma \)
and non-type-conform: Type Object = None
and cp-ctxt: \( cp P \)
and const-ctxt: \( \forall X. \text{const} X \implies \text{const} (P X) \)
shows \( (mk \ (\{heap=\sigma'(oid\rightarrow Object), assocs_2=A, assocs_3=B\}) \models P \ (OclAllInstances-generic \ pre-post \ Type)) \)
\( (mk \ (\{heap=\sigma', assocs_2=A, assocs_3=B\}) \models P \ (OclAllInstances-generic \ pre-post \ Type)) \)
(is \( (?\tau \models P \ ?\varphi) = (?\tau' \models P \ ?\varphi) \))
proof –
have P-cp: \( \forall x \tau. P x \tau = P (\lambda - x \tau) \)
by (metis (full-types) cp-ctxt cp-def)
have A: \( \text{const} (P (\lambda - ?\varphi ?\tau)) \)
shows (and const-ctxt and assumes − Type pre-post Type type-conform assumes state-update-vs-allInstances-generic-tc theorem qed

finally have X: (?τ | P ?φ) = (?τ′ | P (λ·. (?φ ?τ) ?τ′)) by simp

show ?thesis
apply(subst X) apply(subst OCL-core.cp-validity[ symmetric])
yapply(rule StrongEq-L-subst[OF cp-ctxt])
yapply(simp add: OclValid-def StrongEq-def true-def)
yapply(rule state-update-vs-allInstances-generic-noincluding')
by(insert oid-def, auto simp: non-type-conform)

theorem state-update-vs-allInstances-generic-tc:
assumes [simp]: ∀a. pre-post (mk a) = a
assumes oid-def: oid≠ dom σ′
and type-conform: Type Object ≠ None
and cp-ctxt: cp P
and const-ctxt: ∃X. const X ⇒ const (P X)
shows (mk {heap=σ'(oid:Object), assocS2=A, assocS3=B} | P (OclAllInstances-generic pre-post Type)) = 
(mk {heap=σ′, assocS2=A, assocS3=B} | P ((OclAllInstances-generic pre-post Type) −> including(λ·. [(Type Object)]))))

(is (?τ | P ?φ) = (?τ′ | P ?φ′))

proof –

have_P-cp : ∃τ. P x τ = P (λ·. x τ) τ
by (metis (full_types) cp-ctxt cp-def)

have A : const (P (λ·. (?φ ?τ)))
by(simp add: const-ctxt const-ss)

have (?τ | P ?φ) = (?τ | P (?φ ?τ))
by(subst OCL-core.cp-validity, rule refl)

also have ... = (?τ | P (λ·. (?φ ?τ) ?τ′))
by(subst P-cp, rule refl)

also have ... = (?τ′ | P (λ·. (?φ ?τ) ?τ′))
apply(simp add: OclValid-def)
by(subst A[simplified const-def], subst const-true[simplified const-def], simp)

finally have X: (?τ | P ?φ) = (?τ′ | P (λ·. (?φ ?τ) ?τ′))
by simp

let ?allInstances = OclAllInstances-generic pre-post Type

have ?allInstances (?τ = λ·. ?allInstances (?τ′ → including(λ·. [(Type Object)]))) ?τ
apply(rule state-update-vs-allInstances-generic-including)
by(insert oid-def, auto simp: type-conform)

also have ... = ((λ·. ?allInstances (?τ′)) → including(λ·. (λ·. [(Type Object)])) ?τ′) ?τ′)

137
by (subst const-OclIncluding[simplified const-def], simp+)
also have ... = (?allInstances -> including(λ - [Type Object]) ?τ)
apply (subst OCL-lib.cp-OclIncluding[symmetric])
by (insert type-conform, auto)
finally have Y : ?allInstances ?τ = (?allInstances -> including(λ - [Type Object]) ?τ)
by auto
show ?thesis
  apply (subst X) apply (subst OCL-core.cp-validity[symmetric])
  apply (rule StrongEq-L-subst3[OF cp-ctxt])
  apply (simp add: OclValid-def StrongEq-def Y true-def)
done
qed
declare OclAllInstances-generic-def [simp]

OclAllInstances @post

definition OclAllInstances-at-post :: (A :: object ↦ α) ⇒ (A, 'α option option) Set
where OclAllInstances-at-post = OclAllInstances-generic snd

lemma OclAllInstances-at-post-defined: τ \models δ (H . allInstances())
unfolding OclAllInstances-at-post-def
by (rule OclAllInstances-generic-defined)

lemma τ₀ \models H . allInstances() ≡ Set{}
unfolding OclAllInstances-at-post-def
by (rule OclAllInstances-generic-init-empty, simp)

lemma represented-at-post-objects-nonnul:
assumes A: τ \models (((H::(A::object ↦ 'α)).allInstances()) -> includes(x))
shows τ \models not(x \triangleq null)
by (rule represented-generic-objects-nonnul[OF A[simplified OclAllInstances-at-post-def]])

lemma represented-at-post-objects-defined:
assumes A: τ \models (((H::(A::object ↦ 'α)).allInstances()) -> includes(x))
shows τ \models δ (H . allInstances()) ∧ τ \models δ x
unfolding OclAllInstances-at-post-def
by (rule represented-generic-objects-defined[OF A[simplified OclAllInstances-at-post-def]])

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

lemma assumes A: τ \models H . allInstances() -> includes(x)
shows x \in (Some o H) ' run (heap(snd τ))
by (rule represented-generic-objects-in-state[OF A[simplified OclAllInstances-at-post-def]])

lemma state-update-vs-allInstances-at-post-empty:
shows (σ, ([heap=empty, assocs₂=A, assocs₃=B]) \models Type . allInstances() ≡ Set{}}
Here comes a couple of operational rules that allow to infer the value of oclAllInstances from the context \( \tau \). These rules are a special-case in the sense that they are the only rules that relate statements with different \( \tau \)'s. For that reason, new concepts like “constant contexts P” are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

**Lemma state-update-vs-allInstances-at-post-including':**

**Assumes** \( \forall x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object} \)

**And** \( \text{Type Object} \neq \text{None} \)

**Shows** \( (\text{Type . allInstances}()) \)

\[
(\sigma, (\text{heap}=\sigma'(\text{oid} \rightarrow \text{Object}), \text{assoc}_2 = A, \text{assoc}_3 = B))
\]

\[
= 
((\text{Type . allInstances}()) \rightarrow \text{including}(\lambda -. [[\text{drop (Type Object) }]]))
\]

\[
(\sigma, (\text{heap}=\sigma', \text{assoc}_2 = A, \text{assoc}_3 = B))
\]

**Unfolding** \( \text{OclAllInstances-at-post-def} \)

**By** \( (\text{rule state-update-vs-allInstances-generic-including}[OF \text{snd-conv}], \text{insert assms}) \)

**Lemma state-update-vs-allInstances-at-post-including:**

**Assumes** \( \forall x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object} \)

**And** \( \text{Type Object} \neq \text{None} \)

**Shows** \( (\text{Type . allInstances}()) \)

\[
(\sigma, (\text{heap}=\sigma'(\text{oid} \rightarrow \text{Object}), \text{assoc}_2 = A, \text{assoc}_3 = B))
\]

\[
= 
((\text{Type . allInstances}()) \rightarrow \text{including}(\lambda -. [[\text{drop (Type Object) }]]))
\]

\[
(\sigma, (\text{heap}=\sigma', \text{assoc}_2 = A, \text{assoc}_3 = B))
\]

**Unfolding** \( \text{OclAllInstances-at-post-def} \)

**By** \( (\text{rule state-update-vs-allInstances-generic-including}[OF \text{snd-conv}], \text{insert assms}) \)

**Theorem state-update-vs-allInstances-at-post-ntc:**

**Assumes** \( \text{oid-def} \): \( \text{oid} \in \text{dom} \sigma' \)
and non-type-conform: Type Object = None
and cp-ctxt: cp P
and const-ctxt: \( \bigwedge X, \text{const } X \Rightarrow \text{const } (P X) \)
shows \( ((\sigma, \{\text{heap}=\sigma.'(\text{oid}\rightarrow\text{Object}), \text{assoc}_2=A, \text{assoc}_3=B\}) = (P(\text{Type}.\text{allInstances}()).)) = ((\sigma, \{\text{heap}=\sigma', \text{assoc}_2=A, \text{assoc}_3=B\}) = (P(\text{Type}.\text{allInstances}()))) \)

unfolding OclAllInstances-at-post-def
by (rule state-update-vs-allInstances-generic-ntc[OF snd-conv], insert assms)

theorem state-update-vs-allInstances-at-post-tc:
assumes oid-def: oid \( \notin \text{dom } \sigma' \)
and type-conform: Type Object \( \neq \) None
and cp-ctxt: cp P
and const-ctxt: \( \bigwedge X, \text{const } X \Rightarrow \text{const } (P X) \)
shows \( ((\sigma, \{\text{heap}=\sigma.'(\text{oid}\rightarrow\text{Object}), \text{assoc}_2=A, \text{assoc}_3=B\}) = (P(\text{Type}.\text{allInstances}()).)) = ((\sigma, \{\text{heap}=\sigma', \text{assoc}_2=A, \text{assoc}_3=B\}) = (P(\text{Type}.\text{allInstances}()))) \)

unfolding OclAllInstances-at-post-def
by (rule state-update-vs-allInstances-generic-tc[OF snd-conv], insert assms)

OclAllInstances (\( \Theta \text{pre} \))
definition OclAllInstances-at-pre :: (\( \forall \alpha::\text{object } \Rightarrow ' \alpha \) ) \Rightarrow (\( \forall \alpha, \alpha \text{ option option} , \text{Set} \text{allInstances}@pre(') \))
where OclAllInstances-at-pre = OclAllInstances-generic fst

lemma OclAllInstances-at-pre-defined: \( \tau \models \delta (H.\text{allInstances}@pre()) \)
unfolding OclAllInstances-at-pre-def
by (rule OclAllInstances-generic-defined)

lemma \( \tau_0 \models H.\text{allInstances}@pre() \equiv \text{Set}{} \)
unfolding OclAllInstances-at-pre-def
by (rule OclAllInstances-generic-init-empty, simp)

lemma represented-at-pre-objects-nonnull:
assumes A: \( \tau \models ((H::(\forall \alpha::\text{object } \Rightarrow ' \alpha)) . \text{allInstances}@pre()) \rightarrow \text{includes}(x) \)
shows \( \tau \models \text{not}(x \equiv \text{null}) \)
by (rule represented-generic-objects-nonnull[OF A[simplified OclAllInstances-at-pre-def]])

lemma represented-at-pre-objects-defined:
assumes A: \( \tau \models ((H::(\forall \alpha::\text{object } \Rightarrow ' \alpha)) . \text{allInstances}@pre()) \rightarrow \text{includes}(x) \)
shows \( \tau \models \delta (H.\text{allInstances}@pre()) \land \tau \models \delta x \)
unfolding OclAllInstances-at-pre-def
by (rule represented-generic-objects-defined[OF A[simplified OclAllInstances-at-pre-def]])

One way to establish the actual presence of an object representation in a state is:
lemma
assumes A: \( \tau \models H.\text{allInstances}@pre() \rightarrow \text{includes}(x) \)
shows \( x \tau \in (\text{Some } o H) \cdot \text{ran}(\text{heap}(\text{fst } \tau)) \)
by (rule represented-generic-objects-in-state[OF A[simplified OclAllInstances-at-pre-def]])
lemma state-update-vs-allInstances-at-pre-empty:
shows \(((heap=\text{empty},\, \text{assocs}_2=A,\, \text{assocs}_3=B),\, \sigma) \models \text{Type} \, . \, \text{allInstances}\@\, \text{pre}() = \text{Set}\{\} \)
unfolding OclAllInstances-at-pre-def
by(rule state-update-vs-allInstances-generic-empty[OF \, fst-conv])

Here comes a couple of operational rules that allow to infer the value of oclAllInstances@pre from the context \(\tau\). These rules are a special-case in the sense that they are the only rules that relate statements with different \(\tau\)'s. For that reason, new concepts like “constant contexts П” are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straightforward by hand).

lemma state-update-vs-allInstances-at-pre-including':
assumes \(\forall x.\, \sigma'\, \text{oid} = \text{Some} \, x \implies x = \text{Object} \quad \text{and} \quad \text{Type} \, \text{Object} \neq \text{None} \)
shows \(((\text{Type} \, . \, \text{allInstances}\@\, \text{pre}()))

\(((heap=\sigma'(\text{oid}\mapsto\text{Object}),\, \text{assocs}_2=A,\, \text{assocs}_3=B),\, \sigma)\)

= \((\text{Type} \, . \, \text{allInstances}\@\, \text{pre}()) \rightarrow \text{including}(\lambda \, \text{.} \, \text{[}\text{drop} \, (\text{Type} \, \text{Object}) \, ]))\)

\(((heap=\sigma',\, \text{assocs}_2=A,\, \text{assocs}_3=B),\, \sigma)\)
unfolding OclAllInstances-at-pre-def
by(rule state-update-vs-allInstances-generic-including'[OF \, fst-conv], \, \text{insert} \, \text{assms})

lemma state-update-vs-allInstances-at-pre-including:
assumes \(\forall x.\, \sigma'\, \text{oid} = \text{Some} \, x \implies x = \text{Object} \quad \text{and} \quad \text{Type} \, \text{Object} \neq \text{None} \)
shows \(((\text{Type} \, . \, \text{allInstances}\@\, \text{pre}()))

\(((heap=\sigma'(\text{oid}\mapsto\text{Object}),\, \text{assocs}_2=A,\, \text{assocs}_3=B),\, \sigma)\)

= \((\lambda \, \text{.} \, (\text{Type} \, . \, \text{allInstances}\@\, \text{pre}())

\(((heap=\sigma',\, \text{assocs}_2=A,\, \text{assocs}_3=B),\, \sigma)) \rightarrow \text{including}(\lambda \, \text{.} \, \text{[}\text{drop} \, (\text{Type} \, \text{Object}) \, ]))\)

\(((heap=\sigma',\, \text{assocs}_2=A,\, \text{assocs}_3=B),\, \sigma)\)
unfolding OclAllInstances-at-pre-def
by(rule state-update-vs-allInstances-generic-including[OF \, fst-conv], \, \text{insert} \, \text{assms})

lemma state-update-vs-allInstances-at-pre-noincluding':
assumes \(\forall x.\, \sigma'\, \text{oid} = \text{Some} \, x \implies x = \text{Object} \quad \text{and} \quad \text{Type} \, \text{Object} = \text{None} \)
shows \(((\text{Type} \, . \, \text{allInstances}\@\, \text{pre}()))

\(((heap=\sigma'(\text{oid}\mapsto\text{Object}),\, \text{assocs}_2=A,\, \text{assocs}_3=B),\, \sigma)\)

= \((\text{Type} \, . \, \text{allInstances}\@\, \text{pre}())

\(((heap=\sigma',\, \text{assocs}_2=A,\, \text{assocs}_3=B),\, \sigma)\)
unfolding OclAllInstances-at-pre-def
by (rule state-update-vs-allInstances-generic-noincluding \((\text{OF \ fst-conv})\), insert assms)

\textbf{theorem} state-update-vs-allInstances-at-pre-ntc:
\begin{itemize}
\item assumes oid-def: \(\text{oid} \notin \text{dom } \sigma'\)
\item and non-type-conform: Type Object \(\neq \text{None}\)
\item and cp-ctxt: \(\text{cp } P\)
\item and const-ctxt: \(\forall X. \text{const } X \implies \text{const } (P X)\)
\item shows \((((\text{heap }= \sigma' (\text{oid} \rightarrow \text{Object}), \text{assocs}_2 = A, \text{assocs}_3 = B), \sigma) \models (P (\text{Type}. \text{allInstances} @ \text{pre} ())))\)
\end{itemize}
unfolding \text{OclAllInstances-at-pre-def}
by (rule state-update-vs-allInstances-generic-ntc \((\text{OF \ fst-conv})\), insert assms)

\textbf{theorem} state-update-vs-allInstances-at-pre-tc:
\begin{itemize}
\item assumes oid-def: \(\text{oid} \notin \text{dom } \sigma'\)
\item and type-conform: Type Object \(\neq \text{None}\)
\item and cp-ctxt: \(\text{cp } P\)
\item and const-ctxt: \(\forall X. \text{const } X \implies \text{const } (P X)\)
\item shows \((((\text{heap }= \sigma' (\text{oid} \rightarrow \text{Object}), \text{assocs}_2 = A, \text{assocs}_3 = B), \sigma) \models (P (\text{Type}. \text{allInstances} @ \text{pre} ())))\)
\end{itemize}
unfolding \text{OclAllInstances-at-pre-def}
by (rule state-update-vs-allInstances-generic-tc \((\text{OF \ fst-conv})\), insert assms)

\texttt{\#post or \#pre}

\textbf{theorem} StrictRefEqObject-vs-StrongEq'':
\begin{itemize}
\item assumes WFF: \(\text{WFF } \tau\)
\item and valid-x: \(\tau \models (v \ (x :: (\text{\'A} \cdot \text{object}, \text{\'C} \cdot \text{object} \ \text{option} \ \text{option}) \ \text{val}))\)
\item and valid-y: \(\tau \models (v \ y)\)
\item and oid-preserve: \(\forall x. \ x \in \text{ran } (\text{heap } (\text{fst } \tau)) \vee x \in \text{ran } (\text{heap } (\text{snd } \tau)) \implies \text{oid-of } (H x) = \text{oid-of } x\)
\item and xy-together: \(\tau \models ((H . \text{allInstances}() \rightarrow \text{includes}(x)) \text{ and } H . \text{allInstances}() \rightarrow \text{includes}(y)) \text{ or } (H . \text{allInstances} @ \text{pre}() \rightarrow \text{includes}(x) \text{ and } H . \text{allInstances} @ \text{pre}() \rightarrow \text{includes}(y))\)
\item shows \((\tau \models (\text{StrictRefEqObject } x \ y)) = (\tau \models (x \ \triangleq y))\)
\item proof
\begin{itemize}
\item have at-post-def : \(\forall x. \ x \models v \ x \implies \tau \models \delta (H . \text{allInstances}() \rightarrow \text{includes}(x))\)
\item apply (simp add: OclIncludes-def OclValid-def)
\item unfolding OclAllInstances-at-post-defined[simplified OclValid-def]
\item by (subst cp-defined, simp)
\item have at-pre-def : \(\forall x. \ x \models v \ x \implies \tau \models \delta (H . \text{allInstances} @ \text{pre}() \rightarrow \text{includes}(x))\)
\item apply (simp add: OclIncludes-def OclValid-def)
\item unfolding OclAllInstances-at-pre-defined[simplified OclValid-def]
\item by (subst cp-defined, simp)
\item have \(P: \text{Rep-Set-0} \ (\text{Abs-Set-0} [[(\text{Some} \ (H \ \text{\'r} \ \text{ran } (\text{heap \ (\text{fst} \ \tau)}) = \text{\{None\}})]]) = [[\text{Some} \ (H \ \text{\'r} \ \text{ran } (\text{heap \ (\text{fst} \ \tau)}) = \text{\{None\}})]])\)
\end{itemize}
\end{itemize}

142
5.3.3. OclIsNew, OclIsDeleted, OclIsMaintained, OclIsAbsent

definition OclIsNew:: (\forall \alpha::\{null,object\})val \Rightarrow (\forall)Boolean \quad ((\cdot).oclIsNew(''))

where X .oclIsNew() ≡ (λτ . if (δ X) τ = true τ
then [oid-of (X τ) \notin dom(heap(fst τ)) \land
oid-of (X τ) \in dom(heap(snd τ))] 
else invalid τ)

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:: (\forall \alpha::\{null,object\})val \Rightarrow (\forall)Boolean \quad ((\cdot).oclIsDeleted(''))

where X .oclIsDeleted() ≡ (λτ . if (δ X) τ = true τ
then [oid-of (X τ) \in dom(heap(fst τ)) \land
oid-of (X τ) \notin dom(heap(snd τ))] 
else invalid τ)

definition OclIsMaintained:: (\forall \alpha::\{null,object\})val \Rightarrow (\forall)Boolean((\cdot).oclIsMaintained(''))

where X .oclIsMaintained() ≡ (λτ . if (δ X) τ = true τ
then [oid-of (X τ) \in dom(heap(fst τ)) \land
oid-of (X τ) \notin dom(heap(snd τ))] 
else invalid τ)

definition OclIsAbsent:: (\forall \alpha::\{null,object\})val \Rightarrow (\forall)Boolean (\cdot).oclIsAbsent(''))

where X .oclIsAbsent() ≡ (λτ . if (δ X) τ = true τ
then [oid-of (X τ) \notin dom(heap(fst τ)) \land
oid-of (X τ) \in dom(heap(snd τ))] 
else invalid τ)

lemma state-split : τ \models δ X \Rightarrow
\[ \tau \models (X.oclIsNew()) \lor \tau \models (X.oclIsDeleted()) \lor \\
\tau \models (X.oclIsMaintained()) \lor \tau \models (X.oclIsAbsent()) \]

by (simp add: OclIsDeleted-def OclIsNew-def OclIsMaintained-def OclIsAbsent-def OclValid-def true-def, blast)

**Lemma notNew-vs-others: ** \[ \tau |\tau = \delta \ X \Rightarrow \\
(\neg \tau \models (X.oclIsNew())) = (\tau \models (X.oclIsDeleted()) \lor \\
\tau \models (X.oclIsMaintained()) \lor \tau \models (X.oclIsAbsent())) \]

by (simp add: OclIsDeleted-def OclIsNew-def OclIsMaintained-def OclIsAbsent-def OclNot-def OclValid-def true-def, blast)

### 5.3.4. 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** \( \text{OclIsModifiedOnly} ::= (\forall \sigma, \sigma' : \{null, object\} \rightarrow (\forall X. oclIsModifiedOnly)) \)

where \( X \rightarrow oclIsModifiedOnly() \equiv (\lambda (\sigma, \sigma'). \ [
\text{Rep-Set-0}(X(\sigma, \sigma'))]) \)

let \( X' = (\text{oid-of} \ [\text{Rep-Set-0}(X(\sigma, \sigma'))])); \)

\( S = ((\text{dom} \ (\text{heap} \ \sigma)) \cap (\text{dom} \ (\text{heap} \ \sigma'))) - X') \)

in if \( (\delta X) (\sigma, \sigma') = \text{true} \) (\( \forall x \in [\text{Rep-Set-0}(X(\sigma, \sigma'))], x \neq \text{null} \))
then \( [\forall x \in S. (\text{heap} \ \sigma) x = (\text{heap} \ \sigma') x] \)
else invalid \( (\sigma, \sigma') \)

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

**Lemma invalid→oclIsModifiedOnly() = invalid**
by (simp add: OclIsModifiedOnly-def)

**Lemma null→oclIsModifiedOnly() = invalid**
by (simp add: OclIsModifiedOnly-def)

**Lemma**

assumes \( X-null : \tau \models X \rightarrow \text{includes}(\text{null}) \)
shows \( \tau \models X \rightarrow oclIsModifiedOnly() \triangleq \text{invalid} \)
apply (insert X-null, simp add: OclIncludes-def OclIsModifiedOnly-def StrongEq-def OclValid-def true-def)
apply (case_tac \( \tau \), simp)
apply (simp split: split-if-asm)
by (simp add: null-fun-def, blast)
Context Passing

**Lemma** \(cp-OclIsModifiedOnly : X \rightarrow oclIsModifiedOnly() \tau = (\lambda. X \tau) \rightarrow oclIsModifiedOnly() \tau\)

by(simp only: OclIsModifiedOnly-def, case-tac \tau, simp only:, subst cp-defined, simp)

### 5.3.5. 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]: \(OclSelf x H \text{ fst-snd} = (\lambda \tau . \text{ if } (\delta x) \tau = \text{true} \tau \text{ then } \text{H} \left[\text{heap(fst-snd }\tau)\text{)}(\text{oid-of } (x \tau))\right] \text{ else } \text{invalid } \tau \text{ else } \text{invalid } \tau\)

**Definition** \(OclSelf-at-pre :: (\forall::\text{object},'\alpha::\{null,\text{object}\})\text{val }\Rightarrow (\forall \Rightarrow '\alpha)\Rightarrow (\forall::\text{object},'\alpha::\{null,\text{object}\})\text{val }((-)@\text{pre}(\cdot))\)

**Definition** \(OclSelf-at-post :: (\forall::\text{object},'\alpha::\{null,\text{object}\})\text{val }\Rightarrow (\forall \Rightarrow '\alpha)\Rightarrow (\forall::\text{object},'\alpha::\{null,\text{object}\})\text{val }((-)@\text{post}(\cdot))\)

### 5.3.6. Framing Theorem

**Lemma** \(all-oid-diff::\)

- assumes \(def-x : \tau \models \delta x\)
- assumes \(def-X : \tau \models \delta X\)
- assumes \(def-X' : \forall x . x \in [[\text{Rep-Set-0 } (X \tau)]] \implies x \neq \text{null}\)

defines \(P \equiv (\lambda a. \text{not} (\text{StrictRefEqObject } x a))\)

shows \((\tau \models X \rightarrow \text{forAll}(a | P a)) = (\text{oid-of } (x \tau) \notin \text{oid-of }' [[\text{Rep-Set-0 } (X \tau)]]))\)

proof –

have \(P\text{-null-bot}: \forall a . b = \text{null } \lor b = \bot \implies \text{\neg } (\exists x \in [[\text{Rep-Set-0 } (X \tau)]]). P (\lambda(x::'a \text{ state } \times 'a \text{ state}), x) \tau = b \tau)\)

apply(erule disjE)

apply(simp, rule ballI,

  simp add: P-def StrictRefEqObject-def, rename-tac x',
  subst cp-OclNot, simp,
  subgoal-tac x \tau \neq \text{null } \land x' \neq \text{null}, simp,
  simp add: OclNot-def null-fun-def null-option-def bot-option-def bot-fun-def invalid-def,
  (metis def-X' def-x foundation17
  | (metis OCL-core.bot-fun-def OclValid-def Set-inv-lemma def-X def-x defined-def valid-def,
  metis def-X' def-x foundation17)))+

done
have not-inj : \( \forall x y. ((\text{not } x) \tau = (\text{not } y) \tau) = (x \tau = y \tau) \)
by (metis foundation21 foundation22)

have \( P \)-false : \( \exists x \in [[\text{Rep-Set-0} (X \tau)]] \). \( P (\lambda-. x) \tau = \text{false} \implies \)
oid-of \( (x \tau) \in \text{oid-of }' [[\text{Rep-Set-0} (X \tau)]] \)
apply (erule bexE, rename-tac \( x' \))
apply (simp add: P-def)
apply (simp only: OclNot3[symmetric], simp only: not-inj)
apply (simp add: StrictRefE{Object-def split: split-if-asm})
apply (subgoal-tac \( \forall x \in [[\text{Rep-Set-0} (X \tau)]] . \text{oid-of } x' \neq \text{oid-of } (x \tau) \))
apply (metis (mono-tacs) OCL-core.drop.simps def-x foundation17 true-def)
by (simp add: invalid-def bot-option-def true-def)+

have \( P \)-true : \( \forall x \in [[\text{Rep-Set-0} (X \tau)]] \). \( P (\lambda-. x) \tau = \text{true} \implies \)
oid-of \( (x \tau) \notin \text{oid-of }' [[\text{Rep-Set-0} (X \tau)]] \)
apply (subgoal-tac \( \forall x \in [[\text{Rep-Set-0} (X \tau)]] . \text{oid-of } x' \neq \text{oid-of } (x \tau) \))
apply (metis imageE)
apply (rule ballI, drule-tac \( x = x' \in \text{ballE} \)) prefer 3 apply assumption
apply (simp add: P-def)
apply (simp only: OclNot4[symmetric], simp only: not-inj)
apply (simp add: StrictRefE{Object-def split: split-if-asm})
apply (subgoal-tac \( \forall x \in [[\text{Rep-Set-0} (X \tau)]] . \text{oid-of } x' \neq \text{oid-of } (x \tau) \))
apply (metis def-X' def-x foundation17)
by (simp add: invalid-def bot-option-def false-def)+

have bool-split : \( \forall x \in [[\text{Rep-Set-0} (X \tau)]] . P (\lambda-. x) \tau \neq \text{null} \implies \)
\( \forall x \in [[\text{Rep-Set-0} (X \tau)]] . P (\lambda-. x) \tau \neq \bot \implies \)
\( \forall x \in [[\text{Rep-Set-0} (X \tau)]] . P (\lambda-. x) \tau \neq \text{false} \implies \)
\( \forall x \in [[\text{Rep-Set-0} (X \tau)]] . P (\lambda-. x) \tau = \text{true} \)
apply (rule ballI)
apply (drule-tac \( x = x \in \text{ballE} \)) prefer 3 apply assumption
apply (drule-tac \( x = x \in \text{ballE} \)) prefer 3 apply assumption
apply (drule-tac \( x = x \in \text{ballE} \)) prefer 3 apply assumption
apply (metis (full-types) OCL-core.bot-fun-def OclNot4 OclValid-def foundation16 foundation18'
foundation9 not-inj null-fun-def)
by (fast+)

show \( ? \)thesis
apply (subst OclForall-rep-set-true[OCL def-X], simp add: OclValid-def)
apply (rule iffI, simp add: P-true)
by (metis P-false P-null-bot bool-split)
qed

theorem framing:
 assumes modifiesclause: \( \tau \models (X \rightarrow \text{excluding}(x)) \rightarrow \text{oclIsModifiedOnly}() \)
and oid-is-typeopr : \( \tau \models X \rightarrow \forall a (\text{Not } (\text{StrictRefEqObject } x \ a)) \)
shows \( \tau \models (x @ \text{pre } P \models (x @ \text{post } P)) \)
apply (case-tac \( \tau \models \delta \ x \))
proof show \( \sigma \models \delta x \implies \text{thesis} \)
proof assume \( \delta x \models \sigma \) show \( \text{thesis} \)

have \( \sigma \models \delta \)
proof

apply (insert oid-is-typerepr , simp add: OclForall-def OclValid-def split: split-if-asm)
by (simp add: bot-option-def true-def)

have \( \sigma \models \delta \)
proof

apply (case-tac \( \sigma \), simp split: split-if-asm)
apply (simp add: OclIsModifiedOnly-def OclValid-def split: split-if-asm)
apply (case-tac \( \tau \), subst (asm) (2) Abs-Set-0-inverse)
apply (simp, (rule disjI2)+)
apply (metis (hide-lams, mono-tags) Diff-iff Set-inv-lemma def-X)
apply (simp)
apply (rule ballE[where \( \lambda x. x \neq \text{null} \)] simp add: OclSelf-at-pre-def OclSelf-at-post-def)
apply (rule conjI, rule impI)
apply (rule_tac \( \lambda x. P \) \( \in \) arg-cong)
apply (insert modifiesclause[ simplified OclIsModifiedOnly-def OclValid-def])
apply (case-tac \( \sigma \), rename-tac \( \sigma \) \( \sigma' \), simp split: split-if-asm)
apply (simp add: OclExcluding-def)
apply (drule foundation5[ simplified OclValid-def true-def], simp)
apply (simp add: Abs-Set-0-inverse, simp)
apply (rule disjI2)+
apply (metis (hide-lams, no-types) DiffD1 OclValid-def Set-inv-lemma def-x

foundation16 foundation18)
apply (simp)
apply (erule_tac \( \lambda x. \text{oid-of} (x \sigma \) \neq \text{oid-of} \) \( \in \) ballE simp+)
apply (metis (hide-lams, no-types) DiffD1 image-jff image-insert insert-Diff-single insert-absorb oid-is-typerepr)
apply (simp add: invalid-def bot-option-def)+
by blast
qed

As corollary, the framing property can be expressed with only the strong equality as comparison operator.
**Theorem framing':**

**Assumes** \( wff : WFF \ \tau \)

**Assumes** modifiesclause: \( \exists (X \rightarrow \text{excluding}(x)) \rightarrow \text{oclIsModifiedOnly}() \)

**And** oid-is-typerepr: \( \exists (X \rightarrow \text{forall}(a) \not (x \equiv a)) \)

**And** oid-preserve: \( \forall x. x \in \text{run} (\text{heap}(\text{fst} \ \tau)) \vee x \in \text{run} (\text{heap}(\text{snd} \ \tau)) \rightarrow \text{oid-of} (H x) = \text{oid-of} x \)

**And** xy-together:

\( \exists (x @ \text{pre P} \equiv (x @ \text{post P})) \)

**Proof**

- **Have** def-X: \( \exists (x @ \text{pre P} \equiv (x @ \text{post P})) \)

**Apply** \( \text{insert oid-is-typerepr, simp add: Oclforall-def OclValid-def split: split-if-asm} \)

**Show** ?thesis

**Apply** \( \text{case-tac } \tau \equiv \delta \ X \)

**Apply** \( \text{rule framing} \ [\text{OF modifiesclause}] \)

**Apply** \( \text{rule Oclforall-cong'[OF - oid-is-typerepr xy-together], rename-tac y} \)

**Apply** \( \text{cut-tac Set-inv-lemma'[OF def-X]} \) prefer 2 **Apply** assumption

**Apply** \( \text{rule OclNot-contrapos-nn, simp add: StrictRefEqObject-def, subst cp-defined, simp, assumption} \)

**Apply** \( \text{rule StrictRefEqObject-def-SameEq}[\text{THEN iffD1, OF wff - - oid-preserve}, assumption+] \)

**By** \( \text{simp add: OclSelf-at-post-def OclSelf-at-pre-def OclValid-def StrongEq-def true-def} \)+

**QED**

### 5.3.7. Miscellaneous

**Lemma pre-post-new** \( \exists (x . \text{oclIsNew()}() \implies \neg (\tau \equiv v(x @ \text{pre H1}) \wedge \neg (\tau \equiv v(x @ \text{post H2})) \)

**By** \( \text{simp add: OclIsNew-def OclSelf-at-pre-def OclSelf-at-post-def OclValid-def StrongEq-def true-def bot-option-def invalid-def bot-fun-def valid-def split: split-if-asm} \)

**Lemma pre-post-old** \( \exists (x . \text{oclIsDeleted()}() \implies \neg (\tau \equiv v(x @ \text{pre H1}) \wedge \neg (\tau \equiv v(x @ \text{post H2})) \)

**By** \( \text{simp add: OclIsDeleted-def OclSelf-at-pre-def OclSelf-at-post-def OclValid-def StrongEq-def true-def bot-option-def invalid-def bot-fun-def valid-def split: split-if-asm} \)

**Lemma pre-post-absent** \( \exists (x . \text{oclIsAbsent()}() \implies \neg (\tau \equiv v(x @ \text{pre H1}) \wedge \neg (\tau \equiv v(x @ \text{post H2})) \)

**By** \( \text{simp add: OclIsAbsent-def OclSelf-at-pre-def OclSelf-at-post-def OclValid-def StrongEq-def true-def bot-option-def invalid-def bot-fun-def valid-def split: split-if-asm} \)
lemma pre-post-maintained: \( (\tau \models v(x @\text{pre } H1) \lor \tau \models v(x @\text{post } H2)) \implies \tau \models (x .\text{oclIsMaintained}) \)
by (simp add: OclIsMaintained-def OclSelf-at-pre-def OclSelf-at-post-def OclValid-def StrongEq-def true-def false-def bot-option-def invalid-def bot-fun-def valid-def split: split-if-asm)

lemma pre-post-maintained':
\( \tau \models (x .\text{oclIsMaintained}) \implies (\tau \models v(x @\text{pre } (\text{Some } o H1)) \land \tau \models v(x @\text{post } (\text{Some } o H2))) \)
by (simp add: OclIsMaintained-def OclSelf-at-pre-def OclSelf-at-post-def OclValid-def StrongEq-def true-def false-def bot-option-def invalid-def bot-fun-def valid-def split: split-if-asm)

lemma framing-same-state: \( (\sigma, \sigma) @\text{pre } H \equiv (x @\text{post } H) \)
by (simp add: OclSelf-at-pre-def OclSelf-at-post-def OclValid-def StrongEq-def)

end

decision
theory OCL-tools
imports OCL-core
begin

end

theory OCL-main
imports OCL-lib OCL-state OCL-tools
begin

end
Part III.

Examples
6. The Employee Analysis Model

6.1. The Employee Analysis Model (UML)

theory
	Employee-AnalysisModel-UMLPart
imports
	../OCL-main
begin

6.1.1. 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, 7]. In this paper, we follow another approach for UML 2.4: we define the concepts of the compilation informally, and present a concrete example which is verified in Isabelle/HOL.

Outlining the Example

We are presenting here an “analysis-model” of the (slightly modified) example Figure 7.3, page 20 of the OCL standard [33]. Here, analysis model means that associations were really represented as relation on objects on the state—as is intended by the standard—rather than by pointers between objects as is done in our “design model” (see Section 7.1). To be precise, this theory contains the formalization of the data-part covered by the UML class model (see Figure 6.1):

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).
6.1.2. Example Data-Universe and its Infrastructure

Ideally, the following is generated automatically from a UML class model.

Our data universe consists in the concrete class diagram just of node’s, and implicitly of the class object. Each class implies the existence of a class type defined for the corresponding object representations as follows:

\[
\text{datatype typePerson} = \text{mkPerson oid int option}
\]

\[
\text{datatype typeOclAny} = \text{mkOclAny oid (int option) option}
\]

Now, we construct a concrete “universe of OclAny types” by injection into a sum type containing the class types. This type of OclAny will be used as instance for all respective type-variables.

\[
\text{datatype A = inPerson typePerson | inOclAny typeOclAny}
\]

Having fixed the object universe, we can introduce type synonyms that exactly correspond to OCL types. Again, we exploit that our representation of OCL is a “shallow embedding” with a one-to-one correspondance of OCL-types to types of the meta-language HOL.

\[
\begin{align*}
\text{type-synonym Boolean} & = \text{A Boolean} \\
\text{type-synonym Integer} & = \text{A Integer} \\
\text{type-synonym Void} & = \text{A Void} \\
\text{type-synonym OclAny} & = (\text{A}, \text{typeOclAny option option}) \text{ val} \\
\text{type-synonym Person} & = (\text{A}, \text{typePerson option option}) \text{ val} \\
\text{type-synonym Set-Integer} & = (\text{A}, \text{int option option}) \text{ Set} \\
\text{type-synonym Set-Person} & = (\text{A}, \text{typePerson option option}) \text{ Set}
\end{align*}
\]

Just a little check:

\[
\text{typ Boolean}
\]

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 \textit{oid-of} yielding the object id (oid) of the object.

\textbf{instantiation} \texttt{typePerson :: object} \\
begin \\
\textbf{definition} \texttt{oid-of-typePerson-def:} \textit{oid-of} \textit{x} = \textit{(case x of mkPerson oid - ⇒ oid)} \\
\textbf{instance} .. \\
end

\textbf{instantiation} \texttt{typeOclAny :: object} \\
begin \\
\textbf{definition} \texttt{oid-of-typeOclAny-def:} \textit{oid-of} \textit{x} = \textit{(case x of mkOclAny oid - ⇒ oid)} \\
\textbf{instance} .. \\
end

\textbf{instantiation} \texttt{A :: object} \\
begin \\
\textbf{definition} \texttt{oid-of-A-def:} \textit{oid-of} \textit{x} = \textit{(case x of inPerson person ⇒ oid-of person | inOclAny oclany ⇒ oid-of oclany)} \\
\textbf{instance} .. \\
end

\section*{6.1.3. Instantiation of the Generic Strict Equality}

We instantiate the referential equality on \texttt{Person} and \texttt{OclAny}

\textbf{defs} \texttt{(overloaded)} \texttt{StrictRefEqObject-Person} : \texttt{(x::Person) ≡ y \equiv \texttt{StrictRefEqObject} x y} \\
\textbf{defs} \texttt{(overloaded)} \texttt{StrictRefEqObject-OclAny} : \texttt{(x::OclAny) ≡ y \equiv \texttt{StrictRefEqObject} x y}

\textbf{lemmas} \\
\texttt{cp-StrictRefEqObject} \texttt{[of x::Person y::Person τ,} \\
simplified \texttt{StrictRefEqObject-Person[symmetric]} \texttt{]} \\
\texttt{cp-intro(9) \quad \texttt{[of P::Person ⇒ PersonQ::Person ⇒ Person,} \\
simplified \texttt{StrictRefEqObject-Person[symmetric] \]} \\
\texttt{StrictRefEqObject-def \quad \texttt{[of x::Person y::Person,} \\
simplified \texttt{StrictRefEqObject-Person[symmetric]} \texttt{]} \\
\texttt{StrictRefEqObject-defargs \quad \texttt{[of - x::Person y::Person,} \\
simplified \texttt{StrictRefEqObject-Person[symmetric]} \texttt{]} \\
\texttt{StrictRefEqObject-strict1 \quad \texttt{[of x::Person,} \\
simplified \texttt{StrictRefEqObject-Person[symmetric]} \texttt{]} \\
\texttt{StrictRefEqObject-strict2 \quad \texttt{[of x::Person,} \\
simplified \texttt{StrictRefEqObject-Person[symmetric]} \texttt{]}}

For each Class \texttt{C}, we will have a casting operation \texttt{oclAsType(C)}, a test on the actual type \texttt{oclIsTypeOf(C)} as well as its relaxed form \texttt{oclIsKindOf(C)} (corresponding exactly to Java’s \textit{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.

6.1.4. OclAsType

Definition

definition OclAsTypeOclAny #: α ⇒ OclAny ((-) .oclAsType '(OclAny'))
definition OclAsTypePerson #: α ⇒ Person ((-) .oclAsType '(Person'))

definition OclAsTypeOclAny-∀ = (λα. [case u of inOclAny a ⇒ a |
inPerson (mkPerson oid a) ⇒ mkOclAny oid [α]])

lemma OclAsTypeOclAny-∀-some: OclAsTypeOclAny-∀ x ≠ None
by(simp add: OclAsTypeOclAny-∀-def)

defs (overloaded) OclAsTypeOclAny-OclAny:
(X::OclAny) .oclAsType(OclAny) ≡ X

defs (overloaded) OclAsTypePerson-OclAny:
(X::Person) .oclAsType(OclAny) ≡
(λτ. case X τ of |
⊥ ⇒ invalid τ |
[⊥] ⇒ null τ |
[mkPerson oid a] ⇒ [mkOclAny oid [α]])

definition OclAsTypePerson-∀ = (λα. case u of inPerson p ⇒ [p] |
inOclAny (mkOclAny oid [α]) ⇒ [mkPerson oid a] |
⊥ ⇒ None)

defs (overloaded) OclAsTypePerson-OclAny:
(X::OclAny) .oclAsType(Person) ≡
(λτ. case X τ of |
⊥ ⇒ invalid τ |
[⊥] ⇒ null τ |
[mkOclAny oid ⊥] ⇒ invalid τ (* down–cast exception *) |
[mkOclAny oid [α]] ⇒ [mkPerson oid a]])

defs (overloaded) OclAsTypePerson-Person:
(X::Person) .oclAsType(Person) ≡ X

lemmas [simp] =
  OclAsTypeOclAny-∀-OclAny
  OclAsTypePerson-Person

Context Passing

lemma cp-OclAsTypeOclAny-Person-Person: cp P ⇒ cp(λX. (P X::Person)::Person) .oclAsType(OclAny))
by(rule cpII , simp-all add: OclAsTypeOclAny-Person)
lemma cp-OclAsType_{OclAny\rightarrow OclAny-OclAny}: \forall P. (P \Rightarrow cp(\lambda X. (P (X::OclAny)::OclAny)))
  by (rule cpI1, simp-all add: OclAsType_{OclAny\rightarrow OclAny})

lemma cp-OclAsType_{Person\rightarrow Person-Person}: \forall P. (P \Rightarrow cp(\lambda X. (P (X::Person)::Person)))
  by (rule cpI1, simp-all add: OclAsType_{Person\rightarrow Person})

lemma cp-OclAsType_{Person\rightarrow OclAny-OclAny}: \forall P. (P \Rightarrow cp(\lambda X. (P (X::OclAny)::OclAny)))
  by (rule cpI1, simp-all add: OclAsType_{Person\rightarrow OclAny})

lemma cp-OclAsType_{OclAny\rightarrow Person-OclAny}: \forall P. (P \Rightarrow cp(\lambda X. (P (X::OclAny)::Person)))
  by (rule cpI1, simp-all add: OclAsType_{OclAny\rightarrow Person})

lemmas[ simp ] =
  cp-OclAsType_{OclAny\rightarrow Person-Person}
  cp-OclAsType_{OclAny\rightarrow OclAny-OclAny}
  cp-OclAsType_{Person\rightarrow Person-Person}
  cp-OclAsType_{Person\rightarrow OclAny-OclAny}

Execution with Invalid or Null as Argument

lemma OclAsType_{OclAny\rightarrow OclAny-strict} : (invalid::OclAny).OclAsType(OclAny) = invalid
  by (simp)

lemma OclAsType_{OclAny\rightarrow OclAny-nullstrict} : (null::OclAny).OclAsType(OclAny) = null
  by (simp)

lemma OclAsType_{OclAny\rightarrow Person-strict[simp]} : (invalid::Person).OclAsType(OclAny) = invalid
  by (rule ext, simp add: bot-option-def invalid-def OclAsType_{OclAny\rightarrow Person})

lemma OclAsType_{OclAny\rightarrow Person-nullstrict[simp]} : (null::Person).OclAsType(OclAny) = null
  by (rule ext, simp add: null-fun-def null-option-def bot-option-def OclAsType_{OclAny\rightarrow Person})
lemma $OclAsType_{\text{Person}}$-OclAny-strict [simp] : $(\text{invalid}::\text{OclAny})\ .\ oclAsType(\text{Person}) = \text{invalid}$
by (rule ext, simp add: bot-option-def invalid-def
OclAsType$_{\text{Person}}$-OclAny)

lemma $OclAsType_{\text{Person}}$-OclAny-nullstrict [simp] : $(\text{null}::\text{OclAny})\ .\ oclAsType(\text{Person}) = \text{null}$
by (rule ext, simp add: null-fun-def null-option-def bot-option-def
OclAsType$_{\text{Person}}$-OclAny)

lemma $OclAsType_{\text{Person}}$-Person-strict : $(\text{invalid}::\text{Person})\ .\ oclAsType(\text{Person}) = \text{invalid}$
by (simp)

lemma $OclAsType_{\text{Person}}$-Person-nullstrict : $(\text{null}::\text{Person})\ .\ oclAsType(\text{Person}) = \text{null}$
by (simp)

6.1.5. OclIsTypeOf
Definition

corrales $OclIsTypeOf_{\text{OclAny}} :: \alpha \Rightarrow \text{Boolean}$

corrales $OclIsTypeOf_{\text{Person}} :: \alpha \Rightarrow \text{Boolean}$

defs (overloaded) $OclIsTypeOf_{\text{OclAny}}$-OclAny:
$(X::\text{OclAny})\ .\ oclIsTypeOf(\text{OclAny}) \equiv$
$(\lambda \tau. \text{case } X \tau \text{ of}
\hspace{1em} \bot \Rightarrow \text{invalid } \tau
\hspace{1em} | [\bot] \Rightarrow \text{true } \tau \hspace{1em} (* \text{invalid } ?? *)
\hspace{1em} | \hspace{1em} \text{mkOclAny oid } \bot \hspace{1em} \Rightarrow \text{true } \tau
\hspace{1em} | \hspace{1em} \text{mkOclAny oid } [\bot] \hspace{1em} \Rightarrow \text{false } \tau)$

defs (overloaded) $OclIsTypeOf_{\text{OclAny}}$-Person:
$(X::\text{Person})\ .\ oclIsTypeOf(\text{OclAny}) \equiv$
$(\lambda \tau. \text{case } X \tau \text{ of}
\hspace{1em} \bot \Rightarrow \text{invalid } \tau
\hspace{1em} | [\bot] \Rightarrow \text{true } \tau \hspace{1em} (* \text{invalid } ?? *)
\hspace{1em} | [\ ] \Rightarrow \text{false } \tau)$

defs (overloaded) $OclIsTypeOf_{\text{Person}}$-OclAny:
$(X::\text{OclAny})\ .\ oclIsTypeOf(\text{Person}) \equiv$
$(\lambda \tau. \text{case } X \tau \text{ of}
\hspace{1em} \bot \Rightarrow \text{invalid } \tau
\hspace{1em} | [\bot] \Rightarrow \text{true } \tau
\hspace{1em} | \hspace{1em} \text{mkOclAny oid } \bot \hspace{1em} \Rightarrow \text{false } \tau
\hspace{1em} | \hspace{1em} \text{mkOclAny oid } [\bot] \hspace{1em} \Rightarrow \text{true } \tau)$

defs (overloaded) $OclIsTypeOf_{\text{Person}}$-Person:
$(X::\text{Person})\ .\ oclIsTypeOf(\text{Person}) \equiv$
$(\lambda \tau. \text{case } X \tau \text{ of}
\hspace{1em} \bot \Rightarrow \text{invalid } \tau
\hspace{1em} | [\ ] \Rightarrow \text{true } \tau)$

158
Context Passing

\begin{align*}
\text{lemma} \quad & \text{cp-OclIsTypeOf}_{\text{OclAny}}\text{-Person-Person:} \\
& \text{cp} (\lambda X. (P(X::\text{Person})::\text{Person}), \text{oclIsTypeOf}(\text{OclAny})) \\
& \text{by (rule cpI1, simp-all add: OclIsTypeOf}_{\text{OclAny}}\text{-Person)} \\
\text{lemma} \quad & \text{cp-OclIsTypeOf}_{\text{OclAny}}\text{-OclAny-OclAny:} \\
& \text{cp} (\lambda X. (P(X::\text{OclAny})::\text{OclAny}), \text{oclIsTypeOf}(\text{OclAny})) \\
& \text{by (rule cpI1, simp-all add: OclIsTypeOf}_{\text{OclAny}}\text{-OclAny-OclAny)} \\
\text{lemma} \quad & \text{cp-OclIsTypeOf}_{\text{Person}}\text{-Person-Person:} \\
& \text{cp} (\lambda X. (P(X::\text{Person})::\text{Person}), \text{oclIsTypeOf}(\text{Person})) \\
& \text{by (rule cpI1, simp-all add: OclIsTypeOf}_{\text{Person}}\text{-Person-Person)} \\
\text{lemma} \quad & \text{cp-OclIsTypeOf}_{\text{Person}}\text{-OclAny-OclAny:} \\
& \text{cp} (\lambda X. (P(X::\text{OclAny})::\text{OclAny}), \text{oclIsTypeOf}(\text{Person})) \\
& \text{by (rule cpI1, simp-all add: OclIsTypeOf}_{\text{Person}}\text{-OclAny-OclAny)} \\
\text{lemmas [simp]} = \\
& \text{cp-OclIsTypeOf}_{\text{OclAny}}\text{-Person-Person} \\
& \text{cp-OclIsTypeOf}_{\text{OclAny}}\text{-OclAny-OclAny} \\
& \text{cp-OclIsTypeOf}_{\text{Person}}\text{-Person-Person} \\
& \text{cp-OclIsTypeOf}_{\text{Person}}\text{-OclAny-OclAny} \\
& \text{cp-OclIsTypeOf}_{\text{OclAny}}\text{-Person-Person} \\
& \text{cp-OclIsTypeOf}_{\text{OclAny}}\text{-OclAny-Person} \\
& \text{cp-OclIsTypeOf}_{\text{Person}}\text{-Person-OclAny} \\
& \text{cp-OclIsTypeOf}_{\text{OclAny}}\text{-Person-Person-Person} \\
\end{align*}

Execution with Invalid or Null as Argument

\begin{align*}
\text{lemma} \quad & \text{OclIsTypeOf}_{\text{OclAny}}\text{-OclAny-strict1 [simp]:} \\
& \text{(invalid::OclAny) . oclIsTypeOf(OclAny) = invalid} \\
& \text{by (rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def \text{OclIsTypeOf}_{\text{OclAny}}\text{-OclAny})} \\
\text{lemma} \quad & \text{OclIsTypeOf}_{\text{OclAny}}\text{-OclAny-strict2 [simp]:} \\
& \text{(null::OclAny) . oclIsTypeOf(OclAny) = true} \\
& \text{by (rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def \text{OclIsTypeOf}_{\text{OclAny}}\text{-OclAny})} \\
\end{align*}
**Up Down Casting**

**lemma** actualType-larger-staticType:

assumes isdef: \( \tau \models (\delta \ X) \)

shows \( \tau \models (X :: \text{Person}) .oclIsTypeOf(OclAny) \triangleq \text{false} \)

using isdef

by(auto simp : null-option-def bot-option-def OclIsTypeOf OclAny-Person foundation22 foundation16)

**lemma** down-cast-type:

assumes isOclAny: \( \tau \models (X :: \text{OclAny}) .oclIsTypeOf(OclAny) \)

and non-null: \( \tau \models (\delta \ X) \)

shows \( \tau \models (X .oclAsType(Person)) \triangleq \text{invalid} \)

using isOclAny non-null

apply(auto simp : bot-fun-def null-fun-def null-option-def bot-option-def null-def invalid-def OclAsTypeOclAny-Person OclAsTypePerson-OclAny foundation22 foundation16 split: option.split typeOclAny-split typePerson.split)

by(simp add: OclIsTypeOfOclAny-OclAny OclValid-def false-def true-def)

**lemma** down-cast-type':

assumes isOclAny: \( \tau \models (X :: \text{OclAny}) .oclIsTypeOf(OclAny) \)

and non-null: \( \tau \models (\delta \ X) \)

shows \( \tau \models \text{not} (\nu (X .oclAsType(Person))) \)

by(rule foundation15[THEN iffD1], simp add: down-cast-type[OF assms])
lemma up-down-cast :
assumes isdef: \( \tau \models (\delta \ X) \)
shows \( \tau \models ((X :: Person) .oclAsType(OclAny) .oclAsType(Person) \triangleq X) \)
using isdef
by(auto simp : null-fun-def null-option-def bot-option-def null-def invalid-def
\begin{align*}
&\text{OclAsType}_{\text{OclAny}} \cdot \text{Person} \\
&\text{OclAsType}_{\text{Person}} \cdot \text{OclAny} \text{ foundation22 foundation16} \\
&\text{split: option.split type}_{\text{Person}.split}
\end{align*}
)

lemma up-down-cast-Person-OclAny-Person [simp]:
shows \((X :: Person) .oclAsType(OclAny) .oclAsType(Person) = X) \)
apply(rule ext, rename-tac \( \tau \))
apply(rule foundation22 [THEN iffD1])
apply(case-tac \( \tau \models (\delta \ X), \text{simp add: up-down-cast} \))
apply(simp add: def-split-local, elim disjE)
apply(erule StrongEq-L-subst2-rev, simp, simp+)
done

lemma up-down-cast-Person-OclAny-Person':
assumes \( \tau \models v \ X \)
shows \((((X :: \text{Person}) .oclAsType(OclAny) .oclAsType(Person)) \triangleq X) \)
apply(simp only: up-down-cast-Person-OclAny-Person StrictRefEqObject-Person)
apply(rule StrictRefEqObject-sym, simp add: assms)

lemma up-down-cast-Person-OclAny-Person'':
assumes \( \tau \models v \ (X :: \text{Person}) \)
shows \((X .oclIsTypeOf(Person) \text{ implies } (X .oclAsType(OclAny) .oclAsType(Person)) \triangleq X) \)
apply(simp add: OclValid-def)
apply(subst cp-OclImplies)
apply(simp add: StrictRefEqObject-Person StrictRefEqObject-sym[OF assms, simplified OclValid-def])
apply(subst cp-OclImplies[symmetric])
by (simp add: OclImplies-true)

6.1.6. OclIsKindOf

Definition

consts OclIsKindOf_{OclAny} :: 'a \Rightarrow Boolean ((_).oclIsKindOf'(_OclAny'))
consts OclIsKindOf_{Person} :: 'a \Rightarrow Boolean ((_).oclIsKindOf'(_Person'))

defs (overloaded) OclIsKindOf_{OclAny} : OclIsKindOf_{OclAny}:
\begin{align*}
(X :: \text{OclAny}) .oclIsKindOf(OclAny) &\equiv \\
(\lambda \tau. \text{ case } X \tau \text{ of} \\
\bot \Rightarrow \text{invalid } \tau \\
\_ \Rightarrow \text{true } \tau)
\end{align*}

defs (overloaded) OclIsKindOf_{Person} : OclIsKindOf_{Person}:
\begin{align*}
(X :: \text{Person}) .oclIsKindOf(OclAny) &\equiv \\
(\lambda \tau. \text{ case } X \tau \text{ of} \\
\end{align*}
\[\bot \Rightarrow \text{invalid } \tau\]
\[\ast \Rightarrow \text{true } \tau\]

**defs (overloaded)** \(\text{OclIsKindOf}\_\text{Person}-\text{OclAny}:\)

\[(X::\text{OclAny})\_\text{oclIsKindOf}(\text{Person}) \equiv\]
\[(\lambda \tau. \text{case } X \tau \text{ of}\]
\[\bot \Rightarrow \text{invalid } \tau\]
\[\ast \Rightarrow \text{true } \tau\]
\[\text{mkOclAny oid } \bot \Rightarrow \text{false } \tau\]
\[\text{mkOclAny oid } \ast \Rightarrow \text{true } \tau\]

**defs (overloaded)** \(\text{OclIsKindOf}\_\text{Person}-\text{Person}:\)

\[(X::\text{Person})\_\text{oclIsKindOf}(\text{Person}) \equiv\]
\[(\lambda \tau. \text{case } X \tau \text{ of}\]
\[\bot \Rightarrow \text{invalid } \tau\]
\[\ast \Rightarrow \text{true } \tau\]

**Context Passing**

**lemma**  \(\text{cp-OclIsKindOf}\_\text{OclAny-Person-Person}:\)

\(\text{cp}(\lambda X.(P(X::\text{Person})::\text{Person}).\text{oclIsKindOf}(\text{OclAny}))\)

by (rule cpII, simp-all add: OclIsKindOf\_\text{OclAny-}\text{Person})

**lemma**  \(\text{cp-OclIsKindOf}\_\text{OclAny-}\text{OclAny-}\text{OclAny}:\)

\(\text{cp}(\lambda X.(P(X::\text{OclAny})::\text{OclAny}).\text{oclIsKindOf}(\text{OclAny}))\)

by (rule cpII, simp-all add: OclIsKindOf\_\text{OclAny-}\text{OclAny})

**lemma**  \(\text{cp-OclIsKindOf}\_\text{Person-Person-}\text{Person-Person}:\)

\(\text{cp}(\lambda X.(P(X::\text{Person})::\text{Person}).\text{oclIsKindOf}(\text{Person}))\)

by (rule cpII, simp-all add: OclIsKindOf\_\text{Person-}\text{OclAny})

**lemma**  \(\text{cp-OclIsKindOf}\_\text{Person-}\text{Person-}\text{OclAny}:\)

\(\text{cp}(\lambda X.(P(X::\text{Person})::\text{OclAny}).\text{oclIsKindOf}(\text{Person}))\)

by (rule cpII, simp-all add: OclIsKindOf\_\text{Person-}\text{OclAny})

**lemma**  \(\text{cp-OclIsKindOf}\_\text{Person-}\text{OclAny-}\text{Person}:\)

\(\text{cp}(\lambda X.(P(X::\text{OclAny})::\text{Person}).\text{oclIsKindOf}(\text{Person}))\)

by (rule cpII, simp-all add: OclIsKindOf\_\text{Person-}\text{OclAny})

**lemma**  \(\text{cp-OclIsKindOf}\_\text{Person-}\text{OclAny-Person}:\)

\(\text{cp}(\lambda X.(P(X::\text{OclAny})::\text{Person}).\text{oclIsKindOf}(\text{Person}))\)

by (rule cpII, simp-all add: OclIsKindOf\_\text{Person-}\text{OclAny})

**lemmas** [simp] =

\(\text{cp-OclIsKindOf}\_\text{OclAny-}\text{Person-Person}\)
\(\text{cp-OclIsKindOf}\_\text{OclAny-}\text{OclAny-}\text{OclAny}\)
\(\text{cp-OclIsKindOf}\_\text{Person-Person-}\text{Person}\)
Execution with Invalid or Null as Argument

lemma OclIsKindOf\(\text{OclAny}\)\(-\text{OclAny}\)\(-\text{OclAny}\)-strict1\[simp\] : (invalid::\(\text{OclAny}\)) . oclIsKindOf(\(\text{OclAny}\)) = invalid
by(rule ext, simp add: invalid-def bot-option-def
OclIsKindOf\(\text{OclAny}\)\(-\text{OclAny}\)-OclAny)

lemma OclIsKindOf\(\text{OclAny}\)\(-\text{OclAny}\)-strict2\[simp\] : (null::\(\text{OclAny}\)) . oclIsKindOf(\(\text{OclAny}\)) = true
by(rule ext, simp add: null-fun-def null-option-def bot-option-def
OclIsKindOf\(\text{OclAny}\)\(-\text{OclAny}\)-OclAny)

lemma OclIsKindOf\(\text{OclAny}\)\(-\text{Person}\)-strict1\[simp\] : (invalid::\(\text{Person}\)) . oclIsKindOf(\(\text{OclAny}\)) = invalid
by(rule ext, simp add: bot-option-def invalid-def
OclIsKindOf\(\text{OclAny}\)\(-\text{OclAny}\)-Person)

lemma OclIsKindOf\(\text{OclAny}\)\(-\text{Person}\)-strict2\[simp\] : (null::\(\text{Person}\)) . oclIsKindOf(\(\text{OclAny}\)) = true
by(rule ext, simp add: null-fun-def null-option-def bot-option-def bot-option-def
OclIsKindOf\(\text{OclAny}\)\(-\text{OclAny}\)-Person)

lemma OclIsKindOf\(\text{Person}\)\(-\text{OclAny}\)-strict1\[simp\] : (invalid::\(\text{OclAny}\)) . oclIsKindOf(\(\text{Person}\)) = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
OclIsKindOf\(\text{Person}\)\(-\text{OclAny}\)-\(\text{OclAny}\))

lemma OclIsKindOf\(\text{Person}\)\(-\text{OclAny}\)-strict2\[simp\] : (null::\(\text{OclAny}\)) . oclIsKindOf(\(\text{Person}\)) = true
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
null-fun-def null-option-def bot-option-def null-def invalid-def
OclIsKindOf\(\text{Person}\)\(-\text{OclAny}\)-\(\text{OclAny}\))

lemma OclIsKindOf\(\text{Person}\)\(-\text{Person}\)-strict1\[simp\] : (invalid::\(\text{Person}\)) . oclIsKindOf(\(\text{Person}\)) = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
null-fun-def null-option-def bot-option-def null-def invalid-def
OclIsKindOf\(\text{Person}\)\(-\text{Person}\))

lemma OclIsKindOf\(\text{Person}\)\(-\text{Person}\)-strict2\[simp\] : (null::\(\text{Person}\)) . oclIsKindOf(\(\text{Person}\)) = true
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
null-fun-def null-option-def bot-option-def null-def invalid-def
OclIsKindOf\(\text{Person}\)\(-\text{Person}\))

Up Down Casting

lemma actualKind-larger-staticKind:
assumes isdef: \(\tau \models (\delta X)\)
shows \( \tau \models (X::\text{Person}) . \text{oclIsKindOf}(\text{OclAny}) \triangleq \text{true} \)
using \text{isdef}
by (auto simp : bot-option-def
\text{OclIsKindOf}_\text{OclAny}-\text{Person} foundation22 foundation16)

\begin{verbatim}

\begin{lemma} down-cast-kind: 
\begin{assumptions}
\text{isOclAny}: \neg \tau \models (X::\text{OclAny}) . \text{oclIsKindOf}(\text{Person})
and non-null: \tau \models (\delta X)
\end{assumptions}
\begin{shows}
\tau \models (X . \text{oclAsType}(\text{Person})) \triangleq \text{invalid}
\end{shows}
using \text{isOclAny} non-null
apply (auto simp : bot-fun-def null-fun-def null-option-def bot-option-def null-def invalid-def
\text{OclAsType}_\text{OclAny}-\text{Person} \text{OclAsType}_{\text{Person}}-\text{OclAny} foundation22 foundation16
split: option.split type\text{OclAny}.split type\text{Person}.split)
by (simp add: \text{OclIsKindOf}_\text{Person}-\text{OclAny} \text{OclValid-def} false-def true-def)
\end{verbatim}

6.1.7. \text{OclAllInstances}

To denote OCL-types occuring in OCL expressions syntactically—as, for example, as ‘argument’ of \text{oclAllInstances}()—we use the inverses of the injection functions into the object universes; we show that this is sufficient “characterization.”

\textbf{definition} \text{Person} \equiv \text{OclAsType}_{\text{Person}}-\mathbb{A}

\textbf{definition} \text{OclAny} \equiv \text{OclAsType}_{\text{OclAny}}-\mathbb{A}

\begin{lemmas} simp \end{lemmas} = \text{Person-def} \text{OclAny-def}

\begin{verbatim}

\begin{lemma} \text{OclAllInstances-generic}_{\text{OclAny}}-exec: \text{OclAllInstances-generic pre-post} \text{OclAny} = 
(\lambda \tau. \text{Abs-Set-0} \, [[ \text{Some } \tau \text{OclAny } \text{ran} (\text{heap (pre-post } \tau)) ]] )
\end{lemma}

\begin{proof}
let \(?S1 = \lambda \tau. \text{OclAny } \text{ran} (\text{heap (pre-post } \tau))
let \(?S2 = \lambda \tau. (?S1 \tau \setminus \{\text{None}\})
have B : \forall \tau. (?S2 \tau \subseteq (?S1 \tau) \text{ by auto}
have C : \forall \tau. (?S1 \tau \subseteq (?S2 \tau) \text{ by (auto simp: \text{OclAsType}_{\text{OclAny}}-\mathbb{A}-some)}
show ?thesis by (insert equalityI [OF B C], simp)
qed

\begin{lemma} \text{OclAllInstances-at-post}_{\text{OclAny}}-exec: \text{OclAny . allInstances()} = 
(\lambda \tau. \text{Abs-Set-0} \, [[ \text{Some } \tau \text{OclAny } \text{ran} (\text{heap (snd } \tau)) ]] )
\end{lemma}

unfolding \text{OclAllInstances-at-post-def}
by (rule \text{OclAllInstances-generic}_{\text{OclAny}}-exec)

\begin{lemma} \text{OclAllInstances-at-pre}_{\text{OclAny}}-exec: \text{OclAny . allInstances@pre()} = 
(\lambda \tau. \text{Abs-Set-0} \, [[ \text{Some } \tau \text{OclAny } \text{ran} (\text{heap (fst } \tau)) ]] )
\end{lemma}

unfolding \text{OclAllInstances-at-pre-def}
by (rule \text{OclAllInstances-generic}_{\text{OclAny}}-exec)
\end{verbatim}

\text{OclIsTypeOf}

\begin{verbatim}

\begin{lemma} \text{OclAny-allInstances-generic-oclIsTypeOf}_{\text{OclAny}}1:
\begin{assumptions}
\text{simp} : \forall x. \text{pre-post} \, (x, x) = x
\end{assumptions}
\end{lemma}
\end{verbatim}

164
shows \( \exists \tau. \ (\tau \models ((\text{OclAllInstances-generic pre-post OclAny}) \rightarrow \forall x. x.oclIsTypeOf(OclAny))) \)  
apply(rule-tac x = \( ?t_0 \) in exI, simp add: \( ?t_0 \)-def OclValid-def del: OclAllInstances-generic-def)  
apply(simp only: assms OclForall-def refl if-True  
OclAllInstances-generic-defined[simplified OclValid-def])  
apply(simp only: OclAllInstances-generic-def)  
apply(subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)  
by(simp add: OclIsTypeOfOclAny-OclAny)

lemma OclAny-allInstances-at-post-oclIsTypeOfOclAny1:  
\( \exists \tau. \ (\tau \models (\text{OclAny . allInstances()}) \rightarrow \forall x. x.oclIsTypeOf(OclAny))) \)  
unfolding OclAllInstances-at-post-def  
by(rule OclAny-allInstances-generic-oclIsTypeOfOclAny1, simp)

lemma OclAny-allInstances-at-pre-oclIsTypeOfOclAny1:  
\( \exists \tau. \ (\tau \models (\text{OclAny . allInstances@pre()} \rightarrow \forall x. x.oclIsTypeOf(OclAny))) \)  
unfolding OclAllInstances-at-pre-def  
by(rule OclAny-allInstances-generic-oclIsTypeOfOclAny1, simp)

lemma OclAny-allInstances-generic-oclIsTypeOfOclAny2:  
asumes [simp]: \( \forall x. x.oclIsTypeOf(OclAny)) \)  
shows \( \exists \tau. \ (\tau \models \forall x. x.oclIsTypeOf(OclAny)) \)  
proof - \( \exists \tau. \ (\tau \models \forall x. x.oclIsTypeOf(OclAny)) \)  
by simp

lemma OclAny-allInstances-at-post-oclIsTypeOfOclAny2:  
\( \exists \tau. \ (\tau \models \forall x. x.oclIsTypeOf(OclAny)) \)  
unfolding OclAllInstances-at-post-def  
by(rule OclAny-allInstances-generic-oclIsTypeOfOclAny2, simp)

lemma OclAny-allInstances-at-pre-oclIsTypeOfOclAny2:  
\( \exists \tau. \ (\tau \models \forall x. x.oclIsTypeOf(OclAny)) \)  
unfolding OclAllInstances-at-pre-def  
by(rule OclAny-allInstances-generic-oclIsTypeOfOclAny2, simp)

lemma Person-allInstances-generic-oclIsTypeOfPerson:  
\( \tau \models ((\text{OclAllInstances-generic pre-post Person}) \rightarrow \forall x. x.oclIsTypeOf(Person)) \)  
by(simp add: OclValid-def del: OclAllInstances-generic-def)

qed
apply (subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)
by (simp add: OclIsTypeOf Person)

lemma Person-allInstances-at-post-oclIsTypeOf Person:
\[ \tau \models (\text{Person} . \text{allInstances}(\cdot) \rightarrow \forall X (X . \text{oclIsTypeOf}(\text{Person}))) \]
unfolding OclAllInstances-at-post-def
by (rule Person-allInstances-generic-oclIsTypeOf Person)

lemma Person-allInstances-at-pre-oclIsTypeOf Person:
\[ \tau \models (\text{Person} . \text{allInstances}(\cdot)@\text{pre}(\cdot) \rightarrow \forall X (X . \text{oclIsTypeOf}(\text{Person}))) \]
unfolding OclAllInstances-at-pre-def
by (rule Person-allInstances-generic-oclIsTypeOf Person)

OclIsKindOf

lemma OclAny-allInstances-generic-oclIsKindOf OclAny:
\[ \tau \models (\text{(OclAllInstances-generic pre-post OclAny)} \rightarrow \forall X (X . \text{oclIsKindOf}(\text{OclAny}))) \]
apply (simp add: OclValid-def del: OclAllInstances-generic-def)
apply (simp only: OclForall-def refl if-True OclAllInstances-generic-defined [simplified OclValid-def])
apply (simp only: OclAllInstances-generic-def)
apply (subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)
by (simp add: OclIsKindOf OclAny)

lemma OclAny-allInstances-at-post-oclIsKindOf OclAny:
\[ \tau \models (\text{OclAny} . \text{allInstances}(\cdot) \rightarrow \forall X (X . \text{oclIsKindOf}(\text{OclAny}))) \]
unfolding OclAllInstances-at-post-def
by (rule OclAny-allInstances-generic-oclIsKindOf OclAny)

lemma OclAny-allInstances-at-pre-oclIsKindOf OclAny:
\[ \tau \models (\text{OclAny} . \text{allInstances}(\cdot)@\text{pre}(\cdot) \rightarrow \forall X (X . \text{oclIsKindOf}(\text{OclAny}))) \]
unfolding OclAllInstances-at-pre-def
by (rule OclAny-allInstances-generic-oclIsKindOf OclAny)

lemma Person-allInstances-generic-oclIsKindOf OclAny:
\[ \tau \models (\text{(OclAllInstances-generic pre-post Person)} \rightarrow \forall X (X . \text{oclIsKindOf}(\text{OclAny}))) \]
apply (simp add: OclValid-def del: OclAllInstances-generic-def)
apply (simp only: OclForall-def refl if-True OclAllInstances-generic-defined [simplified OclValid-def])
apply (simp only: OclAllInstances-generic-def)
apply (subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)
by (simp add: OclIsKindOf OclAny)

lemma Person-allInstances-at-post-oclIsKindOf OclAny:
\[ \tau \models (\text{Person} . \text{allInstances}(\cdot) \rightarrow \forall X (X . \text{oclIsKindOf}(\text{OclAny}))) \]
unfolding OclAllInstances-at-post-def
by (rule Person-allInstances-generic-oclIsKindOf OclAny)

lemma Person-allInstances-at-pre-oclIsKindOf OclAny:
\[ \tau \models (\text{Person} . \text{allInstances}(\cdot)@\text{pre}(\cdot) \rightarrow \forall X (X . \text{oclIsKindOf}(\text{OclAny}))) \]
unfolding OclAllInstances-at-pre-def
by (rule Person-allInstances-generic-oclIsKindOf OclAny)

lemma Person-allInstances-at-post-oclIsKindOf OclAny:
\[ \tau \models (\text{Person} . \text{allInstances}(\cdot) \rightarrow \forall X (X . \text{oclIsKindOf}(\text{OclAny}))) \]
unfolding OclAllInstances-at-post-def
by (rule Person-allInstances-generic-oclIsKindOf OclAny)

lemma Person-allInstances-at-pre-oclIsKindOf OclAny:
\[ \tau \models (\text{Person} . \text{allInstances}(\cdot)@\text{pre}(\cdot) \rightarrow \forall X (X . \text{oclIsKindOf}(\text{OclAny}))) \]
unfolding OclAllInstances-at-pre-def
by (rule Person-allInstances-generic-oclIsKindOf OclAny)

lemma Person-allInstances-at-post-oclIsKindOf OclAny:
\[ \tau \models (\text{Person} . \text{allInstances}(\cdot) \rightarrow \forall X (X . \text{oclIsKindOf}(\text{OclAny}))) \]
unfolding OclAllInstances-at-post-def
by (rule Person-allInstances-generic-oclIsKindOf OclAny)
\[\tau \models (\text{Person} \text{.allInstances}@\text{pre}() \rightarrow \text{forAll}(X|X \text{.oclIsKindOf}(\text{OclAny})))\] unfolding \text{OclAllInstances-at-pre-def}
by(rule Person-allInstances-generic-oclIsKindOf\text{OclAny})

lemma Person-allInstances-generic-oclIsKindOf\text{Person}:
\[\tau \models ((\text{OclAllInstances-generic pre-post Person}) \rightarrow \text{forAll}(X|X \text{.oclIsKindOf}(\text{Person})))\]
apply(simp add: OclValid-def del: OclAllInstances-generic-def)
apply(simp only: OclForall-def refl if-True OclAllInstances-generic-defined[simplified OclValid-def])
by(add: OclIsKindOf\text{Person}-\text{Person})

lemma Person-allInstances-at-post-oclIsKindOf\text{Person}:
\[\tau \models (\text{Person} \text{.allInstances}() \rightarrow \text{forAll}(X|X \text{.oclIsKindOf}(\text{Person})))\]
unfolding \text{OclAllInstances-at-post-def}
by(rule Person-allInstances-generic-oclIsKindOf\text{Person})

lemma Person-allInstances-at-pre-oclIsKindOf\text{Person}:
\[\tau \models (\text{Person} \text{.allInstances}@\text{pre}() \rightarrow \text{forAll}(X|X \text{.oclIsKindOf}(\text{Person})))\]
unfolding \text{OclAllInstances-at-pre-def}
by(rule Person-allInstances-generic-oclIsKindOf\text{Person})

6.1.8. The Accessors \text{(any, boss, salary)}

Should be generated entirely from a class-diagram.

Definition \text{(of the association Employee-Boss)}

We start with a oid for the association; this oid can be used in presence of association classes to represent the association inside an object, pretty much similar to the Employee\_DesignModel\_UMLPart, where we stored an oid inside the class as “pointer.”

definition oid\text{PersonBOSS} :: oid where oid\text{PersonBOSS} = 10

From there on, we can already define an empty state which must contain for oid\text{PersonBOSS} the empty relation (encoded as association list, since there are associations with a Sequence-like structure).

definition eval-extract :: (\forall (\forall w::\text{object} \text{ option option}) \text{ val}
⇒ (oid ⇒ (\forall (\forall c::null) \text{ val})
⇒ (\forall (\forall c::null) \text{ val})
where eval-extract X f = (\lambda.\tau. \text{ case } X \tau \text{ of}
\downarrow ⇒ \text{invalid } \tau \text{ (* exception propagation *)}
| [ ] \downarrow ⇒ \text{invalid } \tau \text{ (* dereferencing null pointer *)}
| [ ] [ ] ⇒ f (oid-of \text{obj}) \tau

definition choose2-1 = \text{fst}
definition choose₂-2 = snd
definition choose₃-1 = fst
definition choose₂-2 = fst o snd
definition choose₃-3 = snd o snd

definition deref-assocs₂ :: (A state × A state ⇒ A state)
⇒ (oid × oid ⇒ oid × oid)
⇒ oid
⇒ (oid list ⇒ oid ⇒ (A, 'f')val)
⇒ oid
⇒ (A, 'f :: null')val

where deref-assocs₂ pre-post to-from assoc-oid f oid =
(λτ. case (assoc₂₂ (pre-post τ)) assoc-oid of
[ S ] ⇒ f (map (choose₂-2 o to-from)
(filter (λ p. choose₂-1 (to-from p)=oid) S))
oid τ
| - ⇒ invalid τ)

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 = id
definition switch₂-2 = (λ(x,y). (y,x))
definition switch₃-1 = id
definition switch₃-2 = (λ(x,y,z). (x,z,y))
definition switch₃-3 = (λ(x,y,z). (y,x,z))
definition switch₃-4 = (λ(x,y,z). (y,z,x))
definition switch₃-5 = (λ(x,y,z). (z,x,y))
definition switch₃-6 = (λ(x,y,z). (z,y,x))

definition select-object :: ((A, 'b :: null')val
⇒ ((A, 'b')val ⇒ (A, 'c')val ⇒ (A, 'b')val)
⇒ ((A, 'b')val ⇒ (A, 'd')val)
⇒ (oid ⇒ (A, 'c :: null')val)
⇒ oid list
⇒ oid
⇒ (A, 'd')val

where select-object mt incl smash deref l oid = smash (foldl incl mt (map deref l))
(* smash returns null with mt in input (in this case, object contains null pointer) *)

The continuation f is usually instantiated with a smashing function which is either
the identity id or, for 0..1 cardinalities of associations, the OclANY-selector which also
handles the null-cases appropriately. A standard use-case for this combinator is for
example:
term (select-object mtSet OclIncluding OclANY f l oid )::(A, 'a :: null')val

definition deref-oid_person :: (A state × A state ⇒ A state)
⇒ (type_person ⇒ (A, 'c :: null')val)
⇒ oid
⇒ (A, 'c :: null')val
where \( \text{deref-oid}_{\text{Person}} \text{fst-snd f oid} = (\lambda \tau. \text{case heap (fst-snd \( \tau \)) oid of}
\begin{align*}
&| \text{in}_{\text{Person}} \text{obj } \Rightarrow f \text{ obj } \tau \\
&| - \Rightarrow \text{invalid } \tau
\end{align*}
)\)

\[
\text{definition deref-oid}_{\text{OclAny}} :: (\text{A state} \times \text{A state} \Rightarrow \text{A state}) \Rightarrow (\text{type}_{\text{OclAny}} \Rightarrow (\text{A}, \text{'c::null}) \text{val}) \Rightarrow (\text{A}, \text{'c::null}) \text{val}
\]

where \( \text{deref-oid}_{\text{OclAny}} \text{fst-snd f oid} = (\lambda \tau. \text{case heap (fst-snd \( \tau \)) oid of}
\begin{align*}
&| \text{in}_{\text{OclAny}} \text{obj } \Rightarrow f \text{ obj } \tau \\
&| - \Rightarrow \text{invalid } \tau
\end{align*}
)\)

pointer undefined in state or not referencing a type confirm object representation

\[
\text{definition select}_{\text{OclAnyANY}} f = (\lambda X. \text{case } X \text{ of}
\begin{align*}
&| \text{mk}_{\text{OclAny}} - \bot \Rightarrow \text{null} \\
&| (\text{mk}_{\text{OclAny}} - [\text{any}]) \Rightarrow f (\lambda x. [[x]]) \text{ any}
\end{align*}
)
\]

\[
\text{definition select}_{\text{PersonBOSS}} f = \text{select-object mtSet OclIncluding OclANY } (f (\lambda x. [[x]]))
\]

\[
\text{definition select}_{\text{PersonSALARY}} f = (\lambda X. \text{case } X \text{ of}
\begin{align*}
&| \text{mk}_{\text{Person}} - \bot \Rightarrow \text{null} \\
&| (\text{mk}_{\text{Person}} - [\text{salary}]) \Rightarrow f (\lambda x. [[x]]) \text{ salary}
\end{align*}
)
\]

\[
\text{definition deref-assocs}_{2} \text{BOSS fst-snd } f = (\lambda \text{mk}_{\text{Person}} \text{ oid } \Rightarrow \text{deref-assocs}_{2} \text{fst-snd switch}_{2} -1 \text{ oid}_{\text{PersonBOSS}} f \text{ oid})
\]

\[
\text{definition in-pre-state } = \text{fst}
\]
\[
\text{definition in-post-state } = \text{snd}
\]

\[
\text{definition reconst-basetype } = (\lambda \text{convert } x. \text{convert } x)
\]

\[
\text{definition dot}_{\text{OclAnyANY}} :: \text{OclAny } \Rightarrow - ((1(-).\text{any}) \text{ 50})
\]

where \( (X).\text{any } = \text{eval-extract } X
\begin{align*}
&| (\text{deref-oid}_{\text{OclAny}} \text{ in-post-state} \\
&| (\text{select}_{\text{OclAnyANY}} \text{reconst-basetype}))
\end{align*}
\]

\[
\text{definition dot}_{\text{PersonBOSS}} :: \text{Person } \Rightarrow \text{Person } ((1(-).\text{boss}) \text{ 50})
\]

where \( (X).\text{boss } = \text{eval-extract } X
\begin{align*}
&| (\text{deref-oid}_{\text{Person}} \text{ in-post-state} \\
&| (\text{deref-assocs}_{2} \text{BOSS \text{ in-post-state} \\
&| (\text{select}_{\text{PersonBOSS}} \text{BOSS} \\
&| (\text{deref-oid}_{\text{Person}} \text{ in-post-state}))))
\end{align*}
\]

169
definition \(\text{dot}_{\text{Person}}\text{SALARY} :: \text{Person} \Rightarrow \text{Integer} \ ((1\text{-}.\text{salary}) 50)\)

where \((X).\text{salary} = \text{eval-extract} X\)

\(\text{(deref-oid}_{\text{Person}}\text{in-post-state}
\text{select}_{\text{Person}}\text{SALARY}
\text{reconst-basetype})\)

\[\text{definition} \quad \text{dot}_{\text{OclAny}}\text{ANY-at-pre} :: \text{OclAny} \Rightarrow - \ ((1\text{-}.\text{any}@\text{pre}) 50)\]

where \((X).\text{any}@\text{pre} = \text{eval-extract} X\)

\(\text{(deref-oid}_{\text{OclAny}}\text{in-pre-state}
\text{select}_{\text{OclAny}}\text{ANY}
\text{reconst-basetype})\)

\[\text{definition} \quad \text{dot}_{\text{Person}}\text{BOSS-at-pre} :: \text{Person} \Rightarrow \text{Person} \ ((1\text{-}.\text{boss}@\text{pre}) 50)\]

where \((X).\text{boss}@\text{pre} = \text{eval-extract} X\)

\(\text{(deref-oid}_{\text{Person}}\text{in-pre-state}
\text{deref-assocs}_{2}\text{BOSS in-pre-state}
\text{select}_{\text{Person}}\text{BOSS}
\text{deref-oid}_{\text{Person}}\text{in-pre-state})\))

\[\text{definition} \quad \text{dot}_{\text{Person}}\text{SALARY-at-pre} :: \text{Person} \Rightarrow \text{Integer} \ ((1\text{-}.\text{salary}@\text{pre}) 50)\]

where \((X).\text{salary}@\text{pre} = \text{eval-extract} X\)

\(\text{(deref-oid}_{\text{Person}}\text{in-pre-state}
\text{select}_{\text{Person}}\text{SALARY}
\text{reconst-basetype})\)

lemmas \[\text{[simp]} = \]

\(\text{dot}_{\text{OclAny}}\text{ANY-def}\)
\(\text{dot}_{\text{Person}}\text{BOSS-def}\)
\(\text{dot}_{\text{Person}}\text{SALARY-def}\)
\(\text{dot}_{\text{OclAny}}\text{ANY-at-pre-def}\)
\(\text{dot}_{\text{Person}}\text{BOSS-at-pre-def}\)
\(\text{dot}_{\text{Person}}\text{SALARY-at-pre-def}\)

Context Passing

lemmas \[\text{[simp]} = \]

\(\text{eval-extract-def}\)

\[\text{lemma} \quad \text{cp-dot}_{\text{OclAny}}\text{ANY} :: ((X).\text{any}) \tau = ((\lambda \cdot X \tau).\text{any}) \tau \text{ by simp}\]

\[\text{lemma} \quad \text{cp-dot}_{\text{Person}}\text{BOSS} :: ((X).\text{boss}) \tau = ((\lambda \cdot X \tau).\text{boss}) \tau \text{ by simp}\]

\[\text{lemma} \quad \text{cp-dot}_{\text{Person}}\text{SALARY} :: ((X).\text{salary}) \tau = ((\lambda \cdot X \tau).\text{salary}) \tau \text{ by simp}\]

\[\text{lemma} \quad \text{cp-dot}_{\text{OclAny}}\text{ANY-at-pre} :: ((X).\text{any}@\text{pre}) \tau = ((\lambda \cdot X \tau).\text{any}@\text{pre}) \tau \text{ by simp}\]

\[\text{lemma} \quad \text{cp-dot}_{\text{Person}}\text{BOSS-at-pre} :: ((X).\text{boss}@\text{pre}) \tau = ((\lambda \cdot X \tau).\text{boss}@\text{pre}) \tau \text{ by simp}\]

\[\text{lemma} \quad \text{cp-dot}_{\text{Person}}\text{SALARY-at-pre} :: ((X).\text{salary}@\text{pre}) \tau = ((\lambda \cdot X \tau).\text{salary}@\text{pre}) \tau \text{ by simp}\]

\[\text{lemmas} \quad \text{cp-dot}_{\text{OclAny}}\text{ANY-at-pre-I} \quad \text{[simp}, \ \text{intro}!]= \]

\[\text{cp-dot}_{\text{OclAny}}\text{ANY}[\text{THEN} \text{all}][\text{THEN} \text{all}],[\lambda \cdot X \cdot X \cdot \tau \cdot \tau, \ \text{THEN} \text{cp}!]\]

\[\text{lemmas} \quad \text{cp-dot}_{\text{OclAny}}\text{ANY-at-pre-I} \quad \text{[simp}, \ \text{intro}!]= \]

170
Lemma dotOclAny\text{ANY}-at-pre\[ \text{THEN allI}[ \text{THEN allI} ],\]
of \( \lambda X \cdot X \lambda - \_ \_ \_ \_ , \text{THEN cpI} ]

Lemmas dotOclPerson\text{BOSS}-I \text{[simp, intro]}=
dotOclPerson\text{BOSS}[ \text{THEN allI}[ \text{THEN allI} ],\]
of \( \lambda X \cdot X \lambda - \_ \_ \_ \_ , \text{THEN cpI} ]

Lemmas dotOclPerson\text{SALARY}-I \text{[simp, intro]}=
dotOclPerson\text{SALARY}[ \text{THEN allI}[ \text{THEN allI} ],\]
of \( \lambda X \cdot X \lambda - \_ \_ \_ \_ , \text{THEN cpI} ]

Lemmas dotOclPerson\text{SALARY}-at-pre-I \text{[simp, intro]}=
dotOclPerson\text{SALARY}-at-pre[ \text{THEN allI}[ \text{THEN allI} ],\]
of \( \lambda X \cdot X \lambda - \_ \_ \_ \_ , \text{THEN cpI} ]

Execution with Invalid or Null as Argument

Lemma dotOclAny\text{ANY}-nullstrict \text{[simp]}: \text{(null).any = invalid}
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)

Lemma dotOclAny\text{ANY}-at-pre-nullstrict \text{[simp]}: \text{(null).any\_pre = invalid}
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)

Lemma dotOclAny\text{ANY}-at-pre-strict \text{[simp]}: \text{(invalid).any = invalid}
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)

Lemma dotOclAny\text{ANY}-at-pre-strict \text{[simp]}: \text{(invalid).any\_pre = invalid}
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)

Lemma dotOclPerson\text{BOSS}-nullstrict \text{[simp]}: \text{(null).boss = invalid}
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)

Lemma dotOclPerson\text{BOSS}-at-pre-nullstrict \text{[simp]}: \text{(null).boss\_pre = invalid}
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)

Lemma dotOclPerson\text{BOSS}-at-pre-strict \text{[simp]}: \text{(invalid).boss = invalid}
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)

Lemma dotOclPerson\text{BOSS}-at-pre-strict \text{[simp]}: \text{(invalid).boss\_pre = invalid}
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)

Lemma dotOclPerson\text{SALARY}-nullstrict \text{[simp]}: \text{(null).salary = invalid}
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)

Lemma dotOclPerson\text{SALARY}-at-pre-nullstrict \text{[simp]}: \text{(null).salary\_pre = invalid}
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)

Lemma dotOclPerson\text{SALARY}-at-pre-strict \text{[simp]}: \text{(invalid).salary = invalid}
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)

Lemma dotOclPerson\text{SALARY}-at-pre-strict \text{[simp]}: \text{(invalid).salary\_pre = invalid}
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)
6.1.9. A Little Infrastructure on Example States

The example we are defining in this section comes from the figure 6.2.

\[ \sigma_1 \equiv \{ \text{heap} = \emptyset \rightarrow \text{inPerson} (\text{mkPerson} \ oid0 [1300]) \]
\[ \quad (\text{oid1} \rightarrow \text{inPerson} (\text{mkPerson} \ oid1 [1200])) \]
\[ \quad (*\text{oid2}*) \]
\[(\text{oid3} \mapsto \text{in\_person}\,(\text{mk\_person}\,\text{oid3} \ [2600]))\] \\
\[(\text{oid4} \mapsto \text{in\_person}\,\text{person5})\] \\
\[(\text{oid5} \mapsto \text{in\_person}\,(\text{mk\_person}\,\text{oid5} \ [2300]))\] \\
\[(\ast\text{oid6}\ast)\] \\
\[(\ast\text{oid7}\ast)\] \\
\[(\text{oid8} \mapsto \text{in\_person}\,\text{person9}),\] \\
\[\text{assoc}\_2 = \text{empty}(\text{oid\_person}\,\text{BOSS} \mapsto [(\text{oid0},\text{oid1}), (\text{oid3},\text{oid4}), (\text{oid5},\text{oid3})]),\] \\
\[\text{assoc}\_3 = \text{empty}()\]

definition \(\sigma_1' \equiv (\lambda \ \text{heap} = \text{empty}(\text{oid0} \mapsto \text{in\_person}\,\text{person1})\] \\
\[(\text{oid1} \mapsto \text{in\_person}\,\text{person2})\] \\
\[(\text{oid2} \mapsto \text{in\_person}\,\text{person3})\] \\
\[(\text{oid3} \mapsto \text{in\_person}\,\text{person4})\] \\
\[(\ast\text{oid4}\ast)\] \\
\[(\text{oid5} \mapsto \text{in\_person}\,\text{person6})\] \\
\[(\text{oid6} \mapsto \text{in\_Ocl\_Any}\,\text{person7})\] \\
\[(\text{oid7} \mapsto \text{in\_Ocl\_Any}\,\text{person8})\] \\
\[(\text{oid8} \mapsto \text{in\_person}\,\text{person9}),\] \\
\[\text{assoc}\_2 = \text{empty}(\text{oid\_person}\,\text{BOSS} \mapsto [(\text{oid0},\text{oid1}), (\text{oid3},\text{oid3}), (\text{oid5},\text{oid3}), (\text{oid6},\text{oid6})]),\] \\
\[\text{assoc}\_3 = \text{empty}()\]

definition \(\sigma_0 \equiv (\lambda \ \text{heap} = \text{empty}, \text{assoc}\_2 = \text{empty}, \text{assoc}\_3 = \text{empty}())\]

definition X\_\text{Person1} :: \text{Person} \equiv \lambda - - [\text{person1}] \\
definition X\_\text{Person2} :: \text{Person} \equiv \lambda - - [\text{person2}] \\
definition X\_\text{Person3} :: \text{Person} \equiv \lambda - - [\text{person3}] \\
definition X\_\text{Person4} :: \text{Person} \equiv \lambda - - [\text{person4}] \\
definition X\_\text{Person5} :: \text{Person} \equiv \lambda - - [\text{person5}] \\
definition X\_\text{Person6} :: \text{Person} \equiv \lambda - - [\text{person6}] \\
definition X\_\text{Person7} :: \text{Ocl\_Any} \equiv \lambda - - [\text{person7}] \\
definition X\_\text{Person8} :: \text{Ocl\_Any} \equiv \lambda - - [\text{person8}] \\
definition X\_\text{Person9} :: \text{Person} \equiv \lambda - - [\text{person9}] \]
**lemma** [code-unfold]: \((x :: \text{Person}) \equiv y\) = \text{StrictRefEq\textunderscore Object } x \ y \ \text{by}(\text{simp only: StrictRefEq\textunderscore Object\textunderscore Person})

**lemma** [code-unfold]: \((x :: \text{OclAny}) \equiv y\) = \text{StrictRefEq\textunderscore Object } x \ y \ \text{by}(\text{simp only: StrictRefEq\textunderscore Object\textunderscore OclAny})

**lemmas** [simp,code-unfold] = 
\begin{align*}
\text{OclAsType\textunderscore OclAny\textunderscore OclAny} \\
\text{OclAsType\textunderscore OclAny\textunderscore Person} \\
\text{OclAsType\textunderscore Person\textunderscore OclAny} \\
\text{OclAsType\textunderscore Person\textunderscore Person} \\
\text{OclIsTypeOf\textunderscore OclAny\textunderscore OclAny} \\
\text{OclIsTypeOf\textunderscore OclAny\textunderscore Person} \\
\text{OclIsTypeOf\textunderscore Person\textunderscore OclAny} \\
\text{OclIsTypeOf\textunderscore Person\textunderscore Person} \\
\text{OclIsKindOf\textunderscore OclAny\textunderscore OclAny} \\
\text{OclIsKindOf\textunderscore OclAny\textunderscore Person} \\
\text{OclIsKindOf\textunderscore Person\textunderscore OclAny} \\
\text{OclIsKindOf\textunderscore Person\textunderscore Person} \\
\end{align*}

**value** \(\bigwedge \_{\text{pre}} \cdot (\text{pre}, \sigma_1') \trianglerighteq (X_{\text{Person}}. \text{salary} < > 1000)\)

**value** \(\bigwedge \_{\text{pre}} \cdot (\text{pre}, \sigma_1') \trianglerighteq (X_{\text{Person}}. \text{salary} = 1300)\)

**value** \(\bigwedge \text{post} \cdot (\sigma_1, \text{post}) \trianglerighteq (X_{\text{Person}}. \text{salary@pre} = 1000)\)

**value** \(\bigwedge \text{post} \cdot (\sigma_1, \text{post}) \trianglerighteq (X_{\text{Person}}. \text{salary@pre} < > 1300)\)

**lemma** \(\bigwedge \text{pre} \cdot \text{post} \cdot (\text{pre}, \text{post}) \trianglerighteq ((X_{\text{Person}}. \text{oclAsType}(\text{OclAny}). \text{oclAsType}(\text{Person})) \trianglerighteq X_{\text{Person}})\)

by(\text{simp add: OclValid-def OclIsMaintained-def})

\begin{align*}
\sigma_1-def \quad \sigma_1'-def \\
X_{\text{Person}}-def \quad \text{person1-def} \\
\text{oid-def} \quad \text{oid1-def} \quad \text{oid2-def} \quad \text{oid3-def} \quad \text{oid4-def} \quad \text{oid5-def} \quad \text{oid6-def} \\
\text{oid-of-option-def} \quad \text{oid-of-typePerson-def}
\end{align*}

**value** \(\bigwedge \text{pre} \cdot \text{post} \cdot (\text{pre}, \text{post}) \trianglerighteq ((X_{\text{Person}}. \text{oclAsType}(\text{OclAny}). \text{oclAsType}(\text{Person})) \trianglerighteq X_{\text{Person}})\)

by(\text{rule up-down-cast-Person-OclAny-Person' , simp add: X_{\text{Person}}-1-def})

**value** \(\bigwedge \text{pre} \cdot \text{post} \cdot (\text{pre}, \text{post}) \trianglerighteq ((X_{\text{Person}}. \text{oclIsTypeOf}(\text{OclAny}))\)

**value** \(\bigwedge \text{pre} \cdot \text{post} \cdot (\text{pre}, \text{post}) \trianglerighteq \neg(X_{\text{Person}}. \text{oclIsTypeOf}(\text{OclAny}))\)

**value** \(\bigwedge \text{pre} \cdot \text{post} \cdot (\text{pre}, \text{post}) \trianglerighteq ((X_{\text{Person}}. \text{oclIsKindOf}(\text{Person}))\)

**value** \(\bigwedge \text{pre} \cdot \text{post} \cdot (\text{pre}, \text{post}) \trianglerighteq \neg(X_{\text{Person}}. \text{oclIsKindOf}(\text{OclAny}))\)

**value** \(\bigwedge \text{pre} \cdot \text{post} \cdot (\text{pre}, \text{post}) \trianglerighteq ((X_{\text{Person}}. \text{oclIsTypeOf}(\text{OclAny}). \text{oclIsTypeOf}(\text{OclAny}))\)

**value** \(\bigwedge \text{pre} \cdot \text{post} \cdot (\text{pre}, \text{post}) \trianglerighteq ((X_{\text{Person}}. \text{oclIsMaintained}())\)

**value** \(\bigwedge \text{pre} \cdot \text{post} \cdot (\text{pre}, \text{post}) \trianglerighteq ((X_{\text{Person}}. \text{salary} = 1800)\)

**value** \(\bigwedge \text{post} \cdot (\sigma_1, \text{post}) \trianglerighteq (X_{\text{Person}}. \text{salary@pre} = 1200)\)

**lemma** \(\bigwedge \text{pre} \cdot \text{post} \cdot (\text{pre}, \text{post}) \trianglerighteq ((X_{\text{Person}}. \text{oclIsMaintained}())\)

**value** \(\bigwedge \text{pre} \cdot \text{post} \cdot (\text{pre}, \text{post}) \trianglerighteq ((X_{\text{Person}}. \text{salary} \equiv 1\text{800})\)

**value** \(\bigwedge \text{post} \cdot (\sigma_1, \text{post}) \trianglerighteq (X_{\text{Person}}. \text{salary@pre} \equiv 1\text{200})\)

**lemma** \(\bigwedge \text{pre} \cdot \text{post} \cdot (\text{pre}, \text{post}) \trianglerighteq ((X_{\text{Person}}. \text{oclIsMaintained}())\)

\(\sigma_1-def \quad \sigma_1'-def\)
\[ X_{\text{Person}}^2 \text{-def} \ person2\text{-def} \]
\[ \text{oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def} \]
\[ \text{oid-of-option-def oid-of-type Person-def} \)

value $\wedge_{s_{\text{pre}}} \ (s_{\text{pre}},\sigma_1) \models (X_{\text{Person}}^3\ .\text{salary} \ \equiv \ \text{null})$
value $\wedge_{s_{\text{post}}} \ (\sigma_1,\sigma_{\text{post}}) \models \neg(v(X_{\text{Person}}^3\ .\text{salary}@\text{pre}))$
lemma $\ (\sigma_1,\sigma_1') \models (X_{\text{Person}}^3\ .\text{oclIsNew}())$
by (simp add: OclValid-def OclIsNew-def)

\[ X_{\text{Person}}^3 \text{-def person3-def} \]
\[ \text{oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid8-def} \]
\[ \text{oid-of-option-def oid-of-type Person-def} \)

lemma $\ (\sigma_1,\sigma_1') \models (X_{\text{Person}}^4\ .\text{oclIsMaintained}())$
by (simp add: OclValid-def OclIsMaintained-def)

\[ X_{\text{Person}}^4 \text{-def person4-def} \]
\[ \text{oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def} \]
\[ \text{oid-of-option-def oid-of-type Person-def} \)

value $\wedge_{s_{\text{pre}}} \ (s_{\text{pre}},\sigma_1) \models \neg(v(X_{\text{Person}}^5\ .\text{salary}))$
value $\wedge_{s_{\text{post}}} \ (\sigma_1,\sigma_{\text{post}}) \models (X_{\text{Person}}^5\ .\text{salary}@\text{pre} \ \equiv \ 3500)$
lemma $\ (\sigma_1,\sigma_1') \models (X_{\text{Person}}^5\ .\text{oclIsDeleted}())$
by (simp add: OclNot-def OclValid-def OclIsDeleted-def)

\[ X_{\text{Person}}^5 \text{-def person5-def} \]
\[ \text{oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def} \]
\[ \text{oid-of-option-def oid-of-type Person-def} \)

lemma $\ (\sigma_1,\sigma_1') \models (X_{\text{Person}}^6\ .\text{oclIsMaintained}())$
by (simp add: OclValid-def OclIsMaintained-def)

\[ X_{\text{Person}}^6 \text{-def person6-def} \]
\[ \text{oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def} \]
\[ \text{oid-of-option-def oid-of-type Person-def} \)

value $\wedge_{s_{\text{pre}}} s_{\text{post}}. \ (s_{\text{pre}},s_{\text{post}}) \models v(X_{\text{Person}}^7\ .\text{oclAsType(Person}))$
lemma $\wedge_{s_{\text{pre}}} s_{\text{post}}. \ (s_{\text{pre}},s_{\text{post}}) \models \ (X_{\text{Person}}^7\ .\text{oclAsType(Person)}\ .\text{oclAsType(OclAny)}\ .\text{oclAsType(Person}))$

\[ \ \equiv \ (X_{\text{Person}}^7\ .\text{oclAsType(Person)}) \)

175
by (rule up-down-cast-Person-OclAny-Person', simp add: \( X_{\text{Person}} \)-def OclValid-def valid-def person7-def)

lemma \( (\sigma_1, \sigma_1') \models (X_{\text{Person}} \_\text{oclIsNew}()) \)

by (simp add: OclValid-def OclIsNew-def
\( \sigma_1 \)-def \( \sigma_1' \)-def
\( X_{\text{Person}} \)7-def person7-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def oid-of-option-def oid-of-typeOclAny-def)

value \( \lambda \text{pre} \, \text{post}. \ (\text{pre}, \text{post}) \models (X_{\text{Person}} \_8 \leftrightarrow X_{\text{Person}} \_7) \)

value \( \lambda \text{pre} \, \text{post}. \ (\text{pre}, \text{post}) \models \text{not}(v(X_{\text{Person}} \_8 \_\text{oclIsType}(\text{Person}))) \)

value \( \lambda \text{pre} \, \text{post}. \ (\text{pre}, \text{post}) \models (X_{\text{Person}} \_8 \_\text{oclIsTypeOf}(\text{OclAny})) \)

value \( \lambda \text{pre} \, \text{post}. \ (\text{pre}, \text{post}) \models \text{not}(X_{\text{Person}} \_8 \_\text{oclIsKindOf}(\text{Person})) \)

value \( \lambda \text{pre} \, \text{post}. \ (\text{pre}, \text{post}) \models (X_{\text{Person}} \_8 \_\text{oclIsKindOf}(\text{OclAny})) \)


lemma \( \sigma \)-modifiedonly: \( (\sigma_1, \sigma_1') \models (\text{Set}\{X_{\text{Person}} \_1 \_\text{oclAsType}(\text{OclAny}) \)
, \( X_{\text{Person}} \_2 \_\text{oclAsType}(\text{OclAny}) \)
, \( *, X_{\text{Person}} \_3 \_\text{oclAsType}(\text{OclAny}) * \)
, \( X_{\text{Person}} \_4 \_\text{oclAsType}(\text{OclAny}) \)
, \( *, X_{\text{Person}} \_5 \_\text{oclAsType}(\text{OclAny}) * \)
, \( X_{\text{Person}} \_6 \_\text{oclAsType}(\text{OclAny}) \)
, \( *, X_{\text{Person}} \_7 \_\text{oclAsType}(\text{OclAny}) * \)
, \( *, X_{\text{Person}} \_8 \_\text{oclAsType}(\text{OclAny}) * \)
, \( *, X_{\text{Person}} \_9 \_\text{oclAsType}(\text{OclAny}) * \}) \rightarrow \text{oclIsModifiedOnly}() \)

apply (simp add: OclIsModifiedOnly-def OclValid-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid7-def oid8-def
\( X_{\text{Person}} \_1 \)-def \( X_{\text{Person}} \_2 \)-def \( X_{\text{Person}} \_3 \)-def \( X_{\text{Person}} \_4 \)-def
\( X_{\text{Person}} \_5 \)-def \( X_{\text{Person}} \_6 \)-def \( X_{\text{Person}} \_7 \)-def \( X_{\text{Person}} \_8 \)-def \( X_{\text{Person}} \_9 \)-def
person1-def person2-def person3-def person4-def
person5-def person6-def person7-def person8-def person9-def
image-def)

apply (simp add: OclIncluding-rep-set mtSet-rep-set null-option-def bot-option-def)

apply (simp add: oid-of-option-def oid-of-typeOclAny-def , clarsimp)

apply (simp add: \( \sigma_1 \)-def \( \sigma_1' \)-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)

done

lemma \( (\sigma_1, \sigma_1') \models ((X_{\text{Person}} \_9 \_\text{oclAsType}(\text{Person}-\text{A} \ x)) \triangleq X_{\text{Person}} \_9) \)

by (simp add: OclSelf-at-pre-def \( \sigma_1 \)-def oid-of-option-def oid-of-type\text{Person}-def
\( X_{\text{Person}} \_9 \)-def person9-def oid8-def OclValid-def StrongEq-def OclAsType\text{Person}-\text{A}-def)

lemma \( (\sigma_1, \sigma_1') \models ((X_{\text{Person}} \_9 \_\text{oclAsType}(\text{OclAny})) \_\text{oclPre} (\lambda x. \_\text{OclAsType}(\text{OclAny}-\text{A} \ x)) \triangleq X_{\text{Person}} \_9) \)

by (simp add: OclSelf-at-post-def \( \sigma_1' \)-def oid-of-option-def oid-of-type\text{Person}-def
\( X_{\text{Person}} \_9 \)-def person9-def oid8-def OclValid-def StrongEq-def OclAsType\text{Person}-\text{A}-def)

lemma \( (\sigma_1, \sigma_1') \models (((X_{\text{Person}} \_9 \_\text{oclAsType}(\text{OclAny})) \_\text{oclPre} (\lambda x. \_\text{OclAsType}(\text{OclAny}-\text{A} \ x)) \triangleq X_{\text{Person}} \_9) \)

176
proof –

have including4 : \( \forall a \ b \ c \ d \ \tau. \)
\( \{ \lambda \tau. \lfloor \lfloor a \rfloor \rfloor, \lambda \tau. \lfloor \lfloor b \rfloor \rfloor, \lambda \tau. \lfloor \lfloor c \rfloor \rfloor, \lambda \tau. \lfloor \lfloor d \rfloor \rfloor \} \ \tau = \Abs-Set-0 \ \lfloor \lfloor \lfloor a \rfloor \rfloor, \lfloor \lfloor b \rfloor \rfloor, \lfloor \lfloor c \rfloor \rfloor, \lfloor \lfloor d \rfloor \rfloor \rfloor \)
apply(subst abs-rep-simp[symmetric], simp)
by(simp add: OclIncluding-rep-set mtSet-rep-set)

have excluding1 : \( \forall S \ a \ b \ c \ d \ e \ \tau. \)
\( (\lambda -. \Abs-Set-0 \ \lfloor \lfloor \lfloor a \rfloor \rfloor, \lfloor \lfloor b \rfloor \rfloor, \lfloor \lfloor c \rfloor \rfloor, \lfloor \lfloor d \rfloor \rfloor \rfloor) \rightarrow excluding(\lambda \tau. \lfloor \lfloor e \rfloor \rfloor) \ \tau = \Abs-Set-0 \ \lfloor \lfloor \lfloor a \rfloor \rfloor, \lfloor \lfloor b \rfloor \rfloor, \lfloor \lfloor c \rfloor \rfloor, \lfloor \lfloor d \rfloor \rfloor \rfloor - \lfloor \lfloor e \rfloor \rfloor \rfloor \)
apply(simp add: OclExcluding-def)
apply(rule conjI, subst (asm) Abs-Set-0-inject)
apply(rule conjI)
apply(rule implI, subst (asm) Abs-Set-0-inject)
apply(rule conjI)
apply(rule conjI)
apply(rule conjI)
apply(subst Abs-Set-0-inverse, simp add: bot-option-def)
apply(subst Abs-Set-0-inverse, simp add: null-option-def)
apply(subst Abs-Set-0-inverse, simp add: bot-option-def)
apply(subst Abs-Set-0-inverse, simp add: null-option-def)
done

show ?thesis
apply(rule framing|where X = \( \Set \{ \ X_{\text{Person}} 1 \ . \oclAsType(\text{OclAny}) \)
, \( X_{\text{Person}} 2 \ . \oclAsType(\text{OclAny}) \)
, \( *, \ X_{\text{Person}} 3 \ . \oclAsType(\text{OclAny}) \)*)
, \( X_{\text{Person}} 4 \ . \oclAsType(\text{OclAny}) \)
, \( *, \ X_{\text{Person}} 5 \ . \oclAsType(\text{OclAny}) \)*)
, \( X_{\text{Person}} 6 \ . \oclAsType(\text{OclAny}) \)
, \( *, \ X_{\text{Person}} 7 \ . \oclAsType(\text{OclAny}) \)*)
, \( *, \ X_{\text{Person}} 8 \ . \oclAsType(\text{OclAny}) \)*)
, \( *, \ X_{\text{Person}} 9 \ . \oclAsType(\text{OclAny}) \)*)
apply(cut-tac \sigma-modifiedonly)
apply(simp only: OclValid-def)
apply(simp only: OclValid-def X_{\text{Person}} 1-def X_{\text{Person}} 2-def X_{\text{Person}} 3-def X_{\text{Person}} 4-def X_{\text{Person}} 5-def X_{\text{Person}} 6-def X_{\text{Person}} 7-def X_{\text{Person}} 8-def X_{\text{Person}} 9-def)
person1-def person2-def person3-def person4-def person5-def person6-def person7-def person8-def person9-def
OclAsType(\text{OclAny}-\text{Person})
apply(subst cp-OclIsModifiedOnly, subst cp-OclExcluding, subst (asm) cp-OclIsModifiedOnly, simp add: including4 excluding1)
apply(simp only: X_{\text{Person}} 1-def X_{\text{Person}} 2-def X_{\text{Person}} 3-def X_{\text{Person}} 4-def X_{\text{Person}} 5-def X_{\text{Person}} 6-def X_{\text{Person}} 7-def X_{\text{Person}} 8-def X_{\text{Person}} 9-def)
person1-def person2-def person3-def person4-def person5-def person6-def person7-def person8-def person9-def
apply(simp add: OclIncluding-rep-set mtSet-rep-set oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)
apply(simp add: StrictRefEqObject-def oid-of-option-def oid-of-typeOclAny-def OclNot-def OclValid-def
null-option-def bot-option-def)
done
qed

lemma perm-σ₁': σ₁' = ⟨ heap = empty
| (oid8 ↦ in_Person person9)
| (oid7 ↦ in_OclAny person8)
| (oid6 ↦ in_OclAny person7)
| (oid5 ↦ in_Person person6)
| (*oid4*)
| (oid3 ↦ in_Person person4)
| (oid2 ↦ in_Person person3)
| (oid1 ↦ in_Person person2)
| (oid0 ↦ in_Person person1)
| , assocs₂ = assocs₂ σ₁'
| , assocs₃ = assocs₃ σ₁' []

proof –

note P = fun-upd-twist

show θthesis

apply(simp add: σ₁'-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)

apply(subst (1) P, simp)

apply(subst (2) P, simp) apply(subst (1) P, simp)

apply(subst (3) P, simp) apply(subst (2) P, simp) apply(subst (1) P, simp)

apply(subst (4) P, simp) apply(subst (3) P, simp) apply(subst (2) P, simp) apply(subst (1) P, simp)

apply(subst (5) P, simp) apply(subst (4) P, simp) apply(subst (3) P, simp) apply(subst (2) P, simp) apply(subst (1) P, simp)


by(simp)

qed

declare const-ss [simp]

lemma (∀σ₁. {
| σ₁, σ₁' |= (Person . allInstances) ∈ Set{ X_Person1, X_Person2, X_Person3, X_Person4(*, X_Person5*), X_Person6, X_Person7 . oclAsType(Person)(*, X_Person8*), X_Person9 } 
apply(subst perm-σ₁')

apply(simp only: oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def
X_Person1-def X_Person2-def X_Person3-def X_Person4-def
X_Person5-def X_Person6-def X_Person7-def X_Person8-def X_Person9-def

person7-def)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsType person-A-def,
6.2. The Employee Analysis Model (OCL)

theory

end
6.2.1. Standard State Infrastructure

Ideally, these definitions are automatically generated from the class model.

6.2.2. Invariant

These recursive predicates can be defined conservatively by greatest fix-point constructions—automatically. See [4, 6] for details. For the purpose of this example, we state them as axioms here.

axiomatization inv-Person :: Person ⇒ Boolean
where A : \( \tau | \delta \text{self} ) \rightarrow
( \tau \models \text{inv-Person}( \text{self} ) ) =
( ( \tau \models ( \text{self}. \text{boss} = \text{null} ) ) \lor
( \tau \models ( \text{self}. \text{boss} <> \text{null} ) \land
( \tau \models ( \text{inv-Person}( \text{self}. \text{boss} ) ) ) )

axiomatization inv-Person-at-pre :: Person ⇒ Boolean
where B : \( \tau | \delta \text{self} ) \rightarrow
( \tau \models \text{inv-Person-at-pre}( \text{self} ) ) =
( ( \tau \models ( \text{self}. \text{boss@pre} \delta \text{null} ) ) \lor
( \tau \models ( \text{self}. \text{boss@pre} <> \text{null} ) \land
( \tau \models ( \text{self}. \text{boss@pre}. \text{salary@pre} \leq \text{self}. \text{salary@pre} ) ) \land
( \tau \models ( \text{inv-Person-at-pre}( \text{self}. \text{boss@pre} ) ) ) )

A very first attempt to characterize the axiomatization by an inductive definition - this can not be the last word since too weak (should be equality!)

coinductive inv :: Person ⇒ (∃)st ⇒ bool where
( \tau \models ( \delta \text{self} ) ) \rightarrow
( ( \tau \models ( \text{self}. \text{boss} = \text{null} ) ) \lor
( \tau \models ( \text{self}. \text{boss} <> \text{null} ) \land
( \tau \models ( \text{self}. \text{boss}. \text{salary} \leq \text{self}. \text{salary} ) ) \land
( \tau \models ( \text{inv}( \text{self}. \text{boss} ) ) ) )
\rightarrow
t \models \text{inv}( \text{self} )

6.2.3. The Contract of a Recursive Query

The original specification of a recursive query:

```
class Person::contents():Set(Integer)
post: result = if self.boss = null then Set{1}
else self.boss.contents()->including(i) endif
```
consts dot-contents :: Person ⇒ Set-Integer ((1-).contents(′)) 50

axiomatization where dot-contents-def:
(τ \models (\text{(self).contents()} \triangleq \text{result})) =
(if (δ self) τ = true τ
then ((τ \models true) \land
(τ \models \text{result} \triangleq \text{if (self . boss \triangleq null)
then (Set{self . salary})
else (self . boss . contents(\rightarrow including(self . salary))
endif)))
else τ \models \text{result} \triangleq invalid)

consts dot-contents-AT-pre :: Person ⇒ Set-Integer ((1-).contents@pre(′)) 50

axiomatization where dot-contents-AT-pre-def:
(τ \models \text{(self).contents@pre()} \triangleq \text{result}) =
(if (δ self) τ = true τ
then τ \models true \land
(τ \models \text{result} \triangleq \text{if (self).boss@pre} \triangleq \text{null} (* post *)
then Set{(self).salary@pre}
else (self).boss@pre . contents@pre() \rightarrow including(self . salary@pre)
endif))
else τ \models \text{result} \triangleq invalid)

These \text{@pre} variants on methods are only available on queries, i. e., operations without side-effect.

6.2.4. The Contract of a Method

The specification in high-level OCL input syntax reads as follows:

\textbf{context} Person :: insert (x: Integer)
\textbf{post:} contents():Set(Integer)
contents() = contents@pre() \rightarrow including(x)

consts dot-insert :: Person ⇒ Integer ⇒ Void ((1-).insert(′)) 50

axiomatization where dot-insert-def:
(τ \models ((self).insert(x) \triangleq \text{result})) =
(if (δ self) τ = true τ \land (v x) τ = true τ
then τ \models true \land
(τ \models \text{(self).contents()} \triangleq \text{(self).contents@pre()} \rightarrow including(x))
else τ \models ((self).insert(x) \triangleq invalid))

end
7. The Employee Design Model

7.1. The Employee Design Model (UML)

theory Employee-DesignModel-UMLPart
imports ../OCL-main
begin

7.1.1. 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, 7]. In this paper, we follow another approach for UML 2.4: we define the concepts of the compilation informally, and present a concrete example which is verified in Isabelle/HOL.

Outlining the Example

We are presenting here a “design-model” of the (slightly modified) example Figure 7.3, page 20 of the OCL standard [33]. 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).

7.1.2. Example Data-Universe and its Infrastructure

Ideally, the following is generated automatically from a UML class model.
Our data universe consists in the concrete class diagram just of node’s, and implicitly of the class object. Each class implies the existence of a class type defined for the corresponding object representations as follows:

```plaintext
datatype type_{Person} = mk_{Person} oid
                   int option
                   oid option
```

```plaintext
datatype type_{OclAny} = mk_{OclAny} oid
                     (int option × oid option) option
```

Now, we construct a concrete “universe of OclAny types” by injection into a sum type containing the class types. This type of OclAny will be used as instance for all respective type-variables.

```plaintext
datatype Α = in_{Person} type_{Person} | 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 correspondence of OCL-types to types of the meta-language HOL.

```plaintext
type-synonym Boolean = Α Boolean

type-synonym Integer = Α Integer

type-synonym Void = Α Void

type-synonym OclAny = (Α, type_{OclAny} option option) val

type-synonym Person = (Α, type_{Person} option option) val

type-synonym Set-Integer = (Α, int option option) Set

type-synonym Set-Person = (Α, type_{Person} option option) Set
```

Just a little check:

```plaintext
typ Boolean
```

To reuse key-elements of the library like referential equality, we have to show that the object universe belongs to the type class “oclany,” i.e., each class type has to provide a function oid-of yielding the object id (oid) of the object.
instantiation type Person :: object
begin
definition oid-of-type Person-def: oid-of x = (case x of mkPerson oid - ⇒ oid)
instance ..
end

instantiation type OclAny :: object
begin
definition oid-of-type OclAny-def: oid-of x = (case x of mkOclAny oid - ⇒ oid)
instance ..
end

instantiation A :: object
begin
definition oid-of-A-def: oid-of x = (case x of in Person person ⇒ oid-of person
| in OclAny oclany ⇒ oid-of oclany)
instance ..
end

7.1.3. Instantiation of the Generic Strict Equality

We instantiate the referential equality on Person and OclAny

defs(overloaded) StrictRefEqObject-Person : (x::Person) ≡ y ≡ StrictRefEqObject x y
defs(overloaded) StrictRefEqObject-OclAny : (x::OclAny) ≡ y ≡ StrictRefEqObject x y

lemmas

cp-StrictRefEqObject[of x::Person y::Person τ, simplified StrictRefEqObject-Person[symmetric]]
cp-intro(9) [of P::Person ⇒ PersonQ::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.
7.1.4. OclAsType

Definition

c咨ts OclAsTypeOclAny :: 'α ⇒ OclAny ((-) .oclAsType('OclAny'))
c咨ts OclAsTypePerson :: 'α ⇒ Person ((-) .oclAsType('Person'))

definition OclAsTypeOclAny-∀ = (λu. [case u of inOclAny a ⇒ a |
                                              inPerson (mkPerson oid a b) ⇒ mkOclAny oid [(a,b)]])

lemma OclAsTypeOclAny-∀-some: OclAsTypeOclAny-∀ x ≠ None
by(simp add: OclAsTypeOclAny-∀-def)

defs (overloaded) OclAsTypeOclAny-OclAny:
  (X::OclAny) .oclAsType(OclAny) ≡ X

defs (overloaded) OclAsTypeOclAny-Person:
  (X::Person) .oclAsType(OclAny) ≡
  (λτ. case X τ of
          ⊥ ⇒ invalid τ
         | [⊥] ⇒ null τ
         | [mkPerson oid a b] ⇒ [mkOclAny oid [(a,b)]])

definition OclAsTypePerson-∀ = (λu. case u of inPerson p ⇒ |p|
                                           | inOclAny (mkOclAny oid [(a,b)]) ⇒ mkPerson oid a b
                                          | ⊥ ⇒ None)

defs (overloaded) OclAsTypePerson-OclAny:
  (X::OclAny) .oclAsType(Person) ≡
  (λτ. case X τ of
         ⊥ ⇒ invalid τ
        | [⊥] ⇒ null τ
        | [mkOclAny oid ⊥] ⇒ invalid τ (* down−cast exception *)
        | [mkOclAny oid [(a,b)]] ⇒ [mkPerson oid a b] )

defs (overloaded) OclAsTypePerson-Person:
  (X::Person) .oclAsType(Person) ≡ X

lemmas [simp] =
OclAsTypeOclAny-OclAny
OclAsTypePerson-Person

Context Passing

lemma cp-OclAsTypeOclAny-Person-Person: cp P ⇒ cp(λX. (P (X::Person)::Person) .oclAsType(OclAny))
by(rule cpII, simp-all add: OclAsTypeOclAny-Person)
lemma cp-OclAsTypeOclAny-OclAny-OclAny: cp P ⇒ cp(λX. (P (X::OclAny)::OclAny) .oclAsType(OclAny))
by (rule cpI1, simp-all add: OclAsType OclAny-OclAny)

**lemma** cp-OclAsType Person-Person-Person: cp P \(\Longrightarrow\) cp(\(\lambda\)X. (P (X::Person)::Person))

by (rule cpI1, simp-all add: OclAsType Person)

**lemma** cp-OclAsType Person-OclAny-OclAny: cp P \(\Longrightarrow\) cp(\(\lambda\)X. (P (X::OclAny)::OclAny))

by (rule cpI1, simp-all add: OclAsType Person-OclAny)

**lemma** cp-OclAsType OclAny-Person-Person: cp P \(\Longrightarrow\) cp(\(\lambda\)X. (P (X::OclAny)::Person))

by (rule cpI1, simp-all add: OclAsType OclAny)

**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

**Execution with Invalid or Null as Argument**

**lemma** OclAsType OclAny-OclAny-strict : (invalid::OclAny) .oclAsType(OclAny) = invalid

by (simp)

**lemma** OclAsType OclAny-OclAny-nullstrict : (null::OclAny) .oclAsType(OclAny) = null

by (simp)

**lemma** OclAsType OclAny-Person-strict[simp] : (invalid::Person) .oclAsType(OclAny) = invalid

by (rule ext, simp add: bot-option-def invalid-def OclAsType OclAny-Person)

**lemma** OclAsType OclAny-Person-nullstrict[simp] : (null::Person) .oclAsType(OclAny) = null

by (rule ext, simp add: null-fun-def null-option-def bot-option-def OclAsType OclAny-Person)

**lemma** OclAsType Person-OclAny-strict[simp] : (invalid::OclAny) .oclAsType(Person) = invalid
by (rule ext, simp add: bot-option-def invalid-def

OclAsType Person::OclAny)

lemma OclAsType Person::OclAny-nullstrict[simp] : (null::OclAny) . oclAsType(Person) = null
by (rule ext, simp add: null-fun-def null-option-def bot-option-def

OclAsType Person::OclAny)

lemma OclAsType Person::Person-strict : (invalid::Person) . oclAsType(Person) = invalid
by (simp)
lemma OclAsType Person::Person-nullstrict : (null::Person) . oclAsType(Person) = null
by (simp)

7.1.5. OclIsTypeOf

Definition

consts OclIsTypeOf OclAny :: 'a ⇒ Boolean ((·).oclIsTypeOf '(OclAny'))
consts OclIsTypeOf Person :: 'a ⇒ Boolean ((·).oclIsTypeOf '(Person'))

defs (overloaded) OclIsTypeOf OclAny-OclAny:
(X::OclAny) . oclIsTypeOf(OclAny) ≡
(λτ. case X τ of
  ⊥ ⇒ invalid τ
  | [⊥] ⇒ true τ (* invalid ?? *)
  | [mkOclAny oid ⊥ ] ⇒ true τ
  | [mkOclAny 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
  | [mkOclAny oid ⊥ ] ⇒ false τ
  | [mkOclAny oid [- ] ] ⇒ true τ)

defs (overloaded) OclIsTypeOf Person-Person:
(X::Person) . oclIsTypeOf(Person) ≡
(λτ. case X τ of
  ⊥ ⇒ invalid τ
  | - ⇒ true τ)
Context Passing

**Lemma** \( cp\text{-}OclIsTypeOf_{\text{OclAny}}\text{-Person-Person} : \)
\[ cp(\lambda X. (P(X::\text{Person})::\text{Person}), \text{OclIsTypeOf}(\text{OclAny})) \]
by (rule \( cpI1 \), simp-all add: \( \text{OclIsTypeOf}_{\text{OclAny}}\text{-Person-Person} \))

**Lemma** \( cp\text{-}OclIsTypeOf_{\text{OclAny}}\text{-OclAny-OclAny} : \)
\[ cp(\lambda X. (P(X::\text{OclAny})::\text{OclAny}), \text{OclIsTypeOf}(\text{OclAny})) \]
by (rule \( cpI1 \), simp-all add: \( \text{OclIsTypeOf}_{\text{OclAny}}\text{-OclAny-OclAny} \))

**Lemma** \( cp\text{-}OclIsTypeOf_{\text{Person}}\text{-Person-Person} : \)
\[ cp(\lambda X. (P(X::\text{Person})::\text{Person}), \text{OclIsTypeOf}(\text{Person})) \]
by (rule \( cpI1 \), simp-all add: \( \text{OclIsTypeOf}_{\text{Person}}\text{-Person-Person} \))

**Lemma** \( cp\text{-}OclIsTypeOf_{\text{Person}}\text{-OclAny-OclAny} : \)
\[ cp(\lambda X. (P(X::\text{OclAny})::\text{OclAny}), \text{OclIsTypeOf}(\text{Person})) \]
by (rule \( cpI1 \), simp-all add: \( \text{OclIsTypeOf}_{\text{Person}}\text{-OclAny-OclAny} \))

**Lemma** \( cp\text{-}OclIsTypeOf_{\text{OclAny}}\text{-Person-OclAny} : \)
\[ cp(\lambda X. (P(X::\text{Person})::\text{OclAny}), \text{OclIsTypeOf}(\text{OclAny})) \]
by (rule \( cpI1 \), simp-all add: \( \text{OclIsTypeOf}_{\text{OclAny}}\text{-Person-OclAny} \))

**Lemma** \( cp\text{-}OclIsTypeOf_{\text{OclAny}}\text{-OclAny-Person} : \)
\[ cp(\lambda X. (P(X::\text{OclAny})::\text{OclAny}), \text{OclIsTypeOf}(\text{OclAny})) \]
by (rule \( cpI1 \), simp-all add: \( \text{OclIsTypeOf}_{\text{OclAny}}\text{-OclAny-Person} \))

**Lemma** \( \text{OclIsTypeOf}_{\text{OclAny}}\text{-OclAny-Person-Person} \)
\[ (\text{invalid}::\text{OclAny})\text{-}\text{OclIsTypeOf}(\text{OclAny}) = \text{invalid} \]
by (rule \( ext \), simp add: \( \text{null-fun-def} \text{-}\text{null-option-def} \text{-}\text{bot-option-def} \text{-}\text{null-def} \text{-}\text{invalid-def} \text{-}\text{OclIsTypeOf}_{\text{OclAny}}\text{-OclAny} \))

**Lemma** \( \text{OclIsTypeOf}_{\text{OclAny}}\text{-OclAny-Person-Person} \)
\[ (\text{null}::\text{OclAny})\text{-}\text{OclIsTypeOf}(\text{OclAny}) = \text{true} \]
by (rule \( ext \), simp add: \( \text{null-fun-def} \text{-}\text{null-option-def} \text{-}\text{bot-option-def} \text{-}\text{null-def} \text{-}\text{invalid-def} \text{-}\text{OclIsTypeOf}_{\text{OclAny}}\text{-OclAny} \))
lemma OclIsTypeOf_OclAny-Person-strict1 [simp]: 
  (invalid::Person) .oclIsTypeOf(OclAny) = invalid 
by (rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def OclIsTypeOf_OclAny-Person)

lemma OclIsTypeOf_OclAny-Person-strict2 [simp]: 
  (null::Person) .oclIsTypeOf(OclAny) = true 
by (rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def OclIsTypeOf_OclAny-Person)

lemma OclIsTypeOf_Person-OclAny-strict1 [simp]: 
  (invalid::OclAny) .oclIsTypeOf(Person) = invalid 
by (rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def OclIsTypeOf_Person-OclAny)

lemma OclIsTypeOf_Person-OclAny-strict2 [simp]: 
  (null::OclAny) .oclIsTypeOf(Person) = true 
by (rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def OclIsTypeOf_Person-OclAny)

lemma actualType-larger-staticType: 
  assumes isdef: \( \tau \models (\delta \ X) \) 
  shows \( \tau \models (X::\text{Person}) .oclIsTypeOf(OclAny) \equiv \text{false} \) 
using isdef 
by (auto simp : null-option-def bot-option-def OclIsTypeOf_OclAny-Person foundation22 foundation16)

lemma down-cast-type: 
  assumes isOclAny: \( \tau \models (X::\text{OclAny}) .oclIsTypeOf(OclAny) \) and 
  non-null: \( \tau \models (\delta \ X) \) 
  shows \( \tau \models (X .oclAsType(Person)) \equiv \text{invalid} \) 
using isOclAny non-null 
apply (auto simp : bot-fun-def null-fun-def null-option-def bot-option-def null-def invalid-def 
  OclAsType_OclAny-Person OclAsType_Person-OclAny foundation22 foundation16 
  split: option.split type_OclAny-split type_Person.split) 
by (simp add: OclIsTypeOf_OclAny-OclAny OclValid-def false-def true-def)

lemma down-cast-type': 
  assumes isOclAny: \( \tau \models (X::\text{OclAny}) .oclIsTypeOf(OclAny) \) and 
  non-null: \( \tau \models (\delta \ X) \) 
  shows \( \tau \models \text{not} (\upsilon (X .oclAsType(Person))) \) 
by (rule foundation15 [THEN iffD1], simp add: down-cast-type[OF assms])

190
lemma up-down-cast :
assumes isdef: \( \tau \models (\delta X) \)
shows \( \tau \models ((X::\text{Person}) .oclAsType(OclAny) .oclAsType(\text{Person}) \triangleq X) \)
using isdef
by(auto simp: null-fun-def null-option-def bot-option-def null-def invalid-def
\text{OclAsType}OclAny\cdot\text{Person} OclAsType\text{Person}\cdot\text{OclAny} foundation22 foundation16
\text{split}: \text{option.split type}\text{Person.split})

lemma up-down-cast-Person-OclAny-Person [simp]:
shows ((X::\text{Person}) .oclAsType(OclAny) .oclAsType(\text{Person}) = X)
apply(rule ext, rename-tac \( \tau \))
apply(rule foundation22[THEN iffD1])
apply(case-tac \( \tau \models (\delta X) \), simp add: up-down-cast)
apply(simp add: def-split-local, elim disjE)
apply(erule StrongEq-L-subst2-rev, simp, simp)+
done

lemma up-down-cast-Person-OclAny-Person': assumes \( \tau \models v X \)
shows \( ((X::\text{Person}) .oclAsType(OclAny) .oclAsType(\text{Person}) \models X) \)
apply(simp only: up-down-cast-Person-OclAny-Person StrictRefEqObject-Person)
by(rule StrictRefEqObject\cdot-sym, simp add: assms)

lemma up-down-cast-Person-OclAny-Person'': assumes \( \tau \models v (X::\text{Person}) \)
shows \( (X .\text{oclIsTypeOf}(\text{Person}) \implies (X .oclAsType(OclAny) .oclAsType(\text{Person}) \models X) \)
apply(simp add: OclValid-def)
apply(subst cp-OclImplies)
apply(simp add: StrictRefEqObject-Person StrictRefEqObject\cdot-sym\{OF assms, simplified OclValid-def\})
apply(subst cp-OclImplies\{symmetric\})
by(simp add: OclImplies-true)

7.1.6. OclIsKindOf

Definition
const OclIsKindOf Ocl\text{Any} :: '\alpha \Rightarrow \text{Boolean} ((\cdot).\text{oclIsKindOf}'(\text{OclAny}'))
const OclIsKindOf\text{Person} :: '\alpha \Rightarrow \text{Boolean} ((\cdot).\text{oclIsKindOf}'(\text{Person}'))

defs (overloaded) OclIsKindOf\text{Ocl\text{Any}}\cdot\text{Ocl\text{Any}}:
(\text{X::Ocl\text{Any}) .oclIsKindOf(\text{Ocl\text{Any}}) \equiv
 (\lambda. \text{case } X \tau \text{ of
 \bot \Rightarrow \text{invalid } \tau
 | - \Rightarrow \text{true } \tau ))
defs (overloaded) OclIsKindOf\text{Ocl\text{Any}}\cdot\text{Person}:
(X::\text{Person} .oclIsKindOf(\text{Ocl\text{Any}}) \equiv
 (\lambda. \text{case } X \tau \text{ of
}
\[ \bot \Rightarrow invalid \ \tau \]
\[ \rightarrow \Rightarrow true \ \tau \]

**defs (overloaded) OclIsKindOf \_ OclAny**:
\[ (X::OclAny).oclIsKindOf(P\_Person) \equiv \]
\[ (\lambda \tau. \text{case } X \ \tau \ of \]
\[ \bot \Rightarrow invalid \ \tau \]
\[ \mid {\downarrow} \Rightarrow true \ \tau \]
\[ \mid {\text{mkOclAny oid } \bot} \Rightarrow false \ \tau \]
\[ \mid {\text{mkOclAny oid } \bot} \Rightarrow true \ \tau \]

**defs (overloaded) OclIsKindOf \_ Person**:
\[ (X::Person).oclIsKindOf(P\_Person) \equiv \]
\[ (\lambda \tau. \text{case } X \ \tau \ of \]
\[ \bot \Rightarrow invalid \ \tau \]
\[ \rightarrow \Rightarrow true \ \tau \]

**Context Passing**

**lemma**
\[ cp-OclIsKindOf(OclAny\_Person-Person-Person): \]
\[ cp((\lambda X. (P(X::Person)::Person).oclIsKindOf(OclAny))) \]
\[ \text{by (rule cp}\_II, \text{ simp-all add: OclIsKindOf(OclAny\_Person)} \]

**lemma**
\[ cp-OclIsKindOf(OclAny\_OclAny-OclAny\_OclAny): \]
\[ cp((\lambda X. (P(X::OclAny)::OclAny).oclIsKindOf(OclAny))) \]
\[ \text{by (rule cp}\_II, \text{ simp-all add: OclIsKindOf(OclAny\_OclAny) } \]

**lemma**
\[ cp-OclIsKindOf(Person\_Person-Person-Person): \]
\[ cp((\lambda X. (P(X::Person)::Person).oclIsKindOf(Person))) \]
\[ \text{by (rule cp}\_II, \text{ simp-all add: OclIsKindOf(Person\_Person)} \]

**lemma**
\[ cp-OclIsKindOf(OclAny\_Person-Person-OclAny): \]
\[ cp((\lambda X. (P(X::Person)::OclAny).oclIsKindOf(OclAny))) \]
\[ \text{by (rule cp}\_II, \text{ simp-all add: OclIsKindOf(OclAny\_OclAny) } \]

**lemma**
\[ cp-OclIsKindOf(Person\_Person-OclAny-OclAny): \]
\[ cp((\lambda X. (P(X::OclAny)::Person).oclIsKindOf(OclAny))) \]
\[ \text{by (rule cp}\_II, \text{ simp-all add: OclIsKindOf(Person\_OclAny) } \]

**lemma**
\[ cp-OclIsKindOf(OclAny\_Person-Person-\_OclAny): \]
\[ cp((\lambda X. (P(X::OclAny)::Person).oclIsKindOf(OclAny))) \]
\[ \text{by (rule cp}\_II, \text{ simp-all add: OclIsKindOf(OclAny\_Person) } \]

**lemmas [simp] =**
\[ cp-OclIsKindOf(OclAny\_Person-Person-Person) \]
\[ cp-OclIsKindOf(OclAny\_OclAny-OclAny-OclAny) \]
\[ cp-OclIsKindOf(Person\_Person-Person-Person) \]
Execution with Invalid or Null as Argument

**Lemma** \( \text{OclIsKindOf\_OclAny\_OclAny\_strict1} \): \( (\text{invalid}::\text{OclAny}) . \text{OclIsKindOf}(\text{OclAny}) = \text{invalid} \)
**By** (rule ext, simp add: invalid-def bot-option-def)

**Lemma** \( \text{OclIsKindOf\_OclAny\_OclAny\_strict2} \): \( (\text{null}::\text{OclAny}) . \text{OclIsKindOf}(\text{OclAny}) = \text{true} \)
**By** (rule ext, simp add: null-fun-def null-option-def)

**Lemma** \( \text{OclIsKindOf\_OclAny\_Person\_strict1} \): \( (\text{invalid}::\text{Person}) . \text{OclIsKindOf}(\text{OclAny}) = \text{invalid} \)
**By** (rule ext, simp add: bot-option-def invalid-def)

**Lemma** \( \text{OclIsKindOf\_OclAny\_Person\_strict2} \): \( (\text{null}::\text{Person}) . \text{OclIsKindOf}(\text{OclAny}) = \text{true} \)
**By** (rule ext, simp add: null-fun-def null-option-def bot-option-def)

**Lemma** \( \text{OclIsKindOf\_Person\_OclAny\_strict1} \): \( (\text{invalid}::\text{OclAny}) . \text{OclIsKindOf}(\text{Person}) = \text{invalid} \)
**By** (rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)

**Lemma** \( \text{OclIsKindOf\_Person\_OclAny\_strict2} \): \( (\text{null}::\text{OclAny}) . \text{OclIsKindOf}(\text{Person}) = \text{true} \)
**By** (rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)

**Lemma** \( \text{OclIsKindOf\_Person\_Person\_strict1} \): \( (\text{invalid}::\text{Person}) . \text{OclIsKindOf}(\text{Person}) = \text{invalid} \)
**By** (rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)

**Lemma** \( \text{OclIsKindOf\_Person\_Person\_strict2} \): \( (\text{null}::\text{Person}) . \text{OclIsKindOf}(\text{Person}) = \text{true} \)
**By** (rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)

Up Down Casting

**Lemma** \( \text{actualKind\_larger\_staticKind} \):
**Assumes** isdef: \( \tau \models (\delta X) \)
\[ \tau \models (X :: \text{Person}) . \text{oclIsKindOf}(\text{OclAny}) \equiv \text{true} \]

using isDef

by(auto simp : bot-option-def
     \text{OclIsKindOf}_{\text{OclAny} \cdot \text{Person}} \text{foundation22 foundation16})

lemma down-cast-kind:
assumes isOclAny: \( \neg \tau \models (X :: \text{OclAny}) . \text{oclIsKindOf}(\text{Person}) \)
and non-null: \( \tau \models (X . \text{oclAsType}(\text{Person})) \equiv \text{invalid} \)
using isOclAny non-null

apply(auto simp : bot-fun-def null-fun-def null-option-def bot-option-def null-def invalid-def
       \text{OclAsType}_{\text{OclAny} \cdot \text{Person}} \text{OclAsType}_{\text{Person} \cdot \text{OclAny}} \text{foundation22 foundation16}
       \text{split: option.split type}\text{OclAny.split type}\text{Person.split})

by(simp add: \text{OclIsKindOf}_{\text{Person} \cdot \text{OclAny}} \text{OclValid-def false-def true-def})

\section{OclAllInstances}

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

definition \text{Person} \equiv \text{OclAsType}_{\text{Person} \cdot \mathbb{A}}

definition \text{OclAny} \equiv \text{OclAsType}_{\text{OclAny} \cdot \mathbb{A}}

lemmas [simp] = \text{Person-def} \text{OclAny-def}

lemma OclAllInstances-generic_{\text{OclAny}}-exec: \text{OclAllInstances-generic pre-post OclAny} =
\begin{align*}
(\lambda \tau. \text{Abs-Set-0} [[[[\text{Some } \circ \text{OclAny } \cdot \text{ran } (\text{heap (pre-post } \tau)) ]]])
\end{align*}

proof –

let \( ?S1 = \lambda \tau. \text{OclAny } \cdot \text{ran } (\text{heap (pre-post } \tau)) \)

let \( ?S2 = \lambda \tau. \text{abs } ?S1 \tau - \{\text{None}\} \)

have \( B : \bigwedge \tau. ?S2 \tau \subseteq ?S1 \tau \) by auto

have \( C : \bigwedge \tau. ?S1 \tau \subseteq ?S2 \tau \) by(auto simp: \text{OclAsType}_{\text{OclAny} \cdot \mathbb{A}-some})

show ?thesis by(insert equalityI[OF B C], simp)

qed

lemma OclAllInstances-at-post_{\text{OclAny}}-exec: \text{OclAllInstances-at-post OclAny} =
\begin{align*}
(\lambda \tau. \text{Abs-Set-0} [[[[\text{Some } \circ \text{OclAny } \cdot \text{ran } (\text{heap (snd } \tau)) ]]])
\end{align*}

unfolding OclAllInstances-at-post-def

by(rule OclAllInstances-generic_{\text{OclAny}}-exec)

lemma OclAllInstances-at-pre_{\text{OclAny}}-exec: \text{OclAllInstances-at-pre OclAny} =
\begin{align*}
(\lambda \tau. \text{Abs-Set-0} [[[[\text{Some } \circ \text{OclAny } \cdot \text{ran } (\text{heap (fst } \tau)) ]]])
\end{align*}

unfolding OclAllInstances-at-pre-def

by(rule OclAllInstances-generic_{\text{OclAny}}-exec)

OclIsTypeOf

lemma OclAny-allInstances-generic-oclIsTypeOf_{\text{OclAny}}1:

assumes [simp]: \( \forall x. \text{pre-post } (x, x) = x \)
shows \( \exists \tau. (\tau \models ((OclAllInstances-generic \preccurlyeq \forall X, X.oclIsTypeOf(OclAny)))) \)

apply(rule-tac \( x = \tau_0 \in exI \), simp add: \( \tau_0 = \forall X, X.oclIsTypeOf(OclAny) \) def: OclAllInstances-generic-def)

apply(simp only: assms OclForall-def refl if-True
OclAllInstances-generic-defined[simplified OclValid-def])

apply(simp only: OclAllInstances-generic-def)

apply(subst \( (1 \ 2 \ 3) \) Abs-Set-0-inverse, simp add: bot-option-def)

by(simp add: OclIsTypeOfOclAny-OclAny)

lemma OclAny-allInstances-at-post-oclIsTypeOfOclAny1:
\( \exists \tau. (\tau \models ((OclAny . allInstances() \rightarrow \forall X, X.oclIsTypeOf(OclAny)))) \)

unfolding OclAllInstances-at-post-def
by(rule OclAny-allInstances-generic-oclIsTypeOfOclAny1, simp)

lemma OclAny-allInstances-at-pre-oclIsTypeOfOclAny1:
\( \exists \tau. (\tau \models ((OclAny . allInstances@pre() \rightarrow \forall X, X.oclIsTypeOf(OclAny)))) \)

unfolding OclAllInstances-at-pre-def
by(rule OclAny-allInstances-generic-oclIsTypeOfOclAny1, simp)

lemma OclAny-allInstances-generic-oclIsTypeOfOclAny2:
assumes [simp]: \( \forall x. \forall \tau \rightarrow (\tau \models (OclAllInstances-generic \preccurlyeq \forall X, X.oclIsTypeOf(OclAny)))) \)

shows \( \exists \tau. (\tau \models \neg ((OclAllInstances-generic \preccurlyeq \forall X, X.oclIsTypeOf(OclAny)))) \)

proof -
fix oid a let ?t0 = \{ \text{heap = empty(oid } \mapsto \text{inOclAny (mkOclAny oid [a]))},
\text{assoc}_2 = \text{empty}, \text{assoc}_3 = \text{empty} \}

show ?thesis

apply(rule-tac \( x = (?t0, ?t0) \) in exI, simp add: OclValid-def def: OclAllInstances-generic-def)

apply(simp only: OclForall-def refl if-True
OclAllInstances-generic-defined[simplified OclValid-def])

apply(simp only: OclAllInstances-generic-def OclAsType)

apply(subst \( (1 \ 2 \ 3) \) Abs-Set-0-inverse, simp add: bot-option-def)

by(simp add: OclIsTypeOfOclAny-OclAny OclNot-def OclAny-def)

qed

lemma OclAny-allInstances-at-post-oclIsTypeOfOclAny2:
\( \exists \tau. (\tau \models \neg ((OclAny . allInstances() \rightarrow \forall X, X.oclIsTypeOf(OclAny)))) \)

unfolding OclAllInstances-at-post-def
by(rule OclAny-allInstances-generic-oclIsTypeOfOclAny2, simp)

lemma OclAny-allInstances-at-pre-oclIsTypeOfOclAny2:
\( \exists \tau. (\tau \models \neg ((OclAny . allInstances@pre() \rightarrow \forall X, X.oclIsTypeOf(OclAny)))) \)

unfolding OclAllInstances-at-pre-def
by(rule OclAny-allInstances-generic-oclIsTypeOfOclAny2, simp)

lemma Person-allInstances-generic-oclIsTypeOfPerson:
\( \tau \models ((OclAllInstances-generic \preccurlyeq \forall X, X.oclIsTypeOf(Person)))) \)

apply(simp add: OclValid-def def: OclAllInstances-generic-def)

apply(simp only: OclForall-def refl if-True
OclAllInstances-generic-defined[simplified OclValid-def])

apply(simp only: OclAllInstances-generic-def)

195
apply (subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)
by (simp add: OclIsTypeOf Person-Person)

lemma Person-allInstances-at-post-oclIsTypeOfPerson:
\( \tau \models (\text{Person.allInstances}() \rightarrow \forall X. X.oclIsTypeOf(\text{Person}))) \)
unfolding OclAllInstances-at-post-def
by (rule Person-allInstances-generic-oclIsTypeOfPerson)

lemma Person-allInstances-at-pre-oclIsTypeOfPerson:
\( \tau \models (\text{Person.allInstances@pre()} \rightarrow \forall X. X.oclIsTypeOf(\text{Person}))) \)
unfolding OclAllInstances-at-pre-def
by (rule Person-allInstances-generic-oclIsTypeOfPerson)

OclIsKindOf

lemma OclAny-allInstances-generic-oclIsKindOfOclAny:
\( \tau \models ((\text{OclAllInstances-generic pre-post OclAny}) \rightarrow \forall X. X.oclIsKindOf(\text{OclAny}))) \)
apply (simp add: OclValid-def del: OclAllInstances-generic-def)
apply (simp only: OclForall-def refl if-True OclAllInstances-generic-defined [simplified OclValid-def])
apply (simp only: OclAllInstances-generic-def)
apply (subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)
by (simp add: OclIsKindOf OclAny-Person)

lemma OclAny-allInstances-at-post-oclIsKindOfOclAny:
\( \tau \models (\text{OclAny.allInstances}() \rightarrow \forall X. X.oclIsKindOf(\text{OclAny}))) \)
unfolding OclAllInstances-at-post-def
by (rule OclAny-allInstances-generic-oclIsKindOfOclAny)

lemma OclAny-allInstances-at-pre-oclIsKindOfOclAny:
\( \tau \models (\text{OclAny.allInstances@pre()} \rightarrow \forall X. X.oclIsKindOf(\text{OclAny}))) \)
unfolding OclAllInstances-at-pre-def
by (rule OclAny-allInstances-generic-oclIsKindOfOclAny)

lemma Person-allInstances-generic-oclIsKindOfOclAny:
\( \tau \models ((\text{OclAllInstances-generic pre-post Person}) \rightarrow \forall X. X.oclIsKindOf(\text{OclAny}))) \)
apply (simp add: OclValid-def del: OclAllInstances-generic-def)
apply (simp only: OclForall-def refl if-True OclAllInstances-generic-defined [simplified OclValid-def])
apply (simp only: OclAllInstances-generic-def)
apply (subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)
by (simp add: OclIsKindOf OclAny-Person)

lemma Person-allInstances-at-post-oclIsKindOfOclAny:
\( \tau \models (\text{Person.allInstances}() \rightarrow \forall X. X.oclIsKindOf(\text{OclAny}))) \)
unfolding OclAllInstances-at-post-def
by (rule Person-allInstances-generic-oclIsKindOfOclAny)

lemma Person-allInstances-at-pre-oclIsKindOfOclAny:

196
\[ \tau \models \text{(Person . allInstances@pre()) -> forAll}(X | X . oclIsKindOf(OclAny))) \]

**unfolding** OclAllInstances-at-pre-def

**by**(rule Person-allInstances-generic-oclIsKindOfOclAny)

**lemma** Person-allInstances-generic-oclIsKindOfPerson:
\[ \tau \models ((\text{OclAllInstances-generic pre-post Person}) -> \text{forAll}(X | X . oclIsKindOf(Person))) \]

**apply**(simp add: OclValid-def del: OclAllInstances-generic-def)

**apply**(simp only: OclForall-def refl if-True OclAllInstances-generic-defined[simplified OclValid-def])

**apply**(simp only: OclAllInstances-generic-def)

**apply**(subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)

**by**(simp add: OclIsKindOfPerson-Person)

**lemma** Person-allInstances-at-post-oclIsKindOfPerson:
\[ \tau \models \text{(Person . allInstances@post() -> forAll}(X | X . oclIsKindOf(Person))) \]

**unfolding** OclAllInstances-at-post-def

**by**(rule Person-allInstances-generic-oclIsKindOfPerson)

**lemma** Person-allInstances-at-pre-oclIsKindOfPerson:
\[ \tau \models \text{(Person . allInstances@pre()) -> forAll}(X | X . oclIsKindOf(Person))) \]

**unfolding** OclAllInstances-at-pre-def

**by**(rule Person-allInstances-generic-oclIsKindOfPerson)

### 7.1.8. The Accessors (any, boss, salary)

Should be generated entirely from a class-diagram.

**Definition**

**definition** eval-extract :: (\(\mathfrak{A}\),('a::object) option option) val

\[ \Rightarrow (\text{oid} \Rightarrow (\mathfrak{A},'c::null) \text{val}) \]

\[ \Rightarrow (\mathfrak{A},'c::null) \text{val} \]

**where** eval-extract \(X\ f\) = (\(\lambda\ \tau. \text{case } X\ \tau\ of\)

\[ \bot \Rightarrow \text{invalid}\ \tau\ \text{(* exception propagation *)} \]

\[ [\ ] \Rightarrow \text{invalid}\ \tau\ \text{(* dereferencing null pointer *)} \]

\[ [[[\ obj \ ]]} \Rightarrow f (\text{oid-of}\ \text{obj})\ \tau] \]

**definition** deref-oid\_Person :: (\(\mathfrak{A}\) state \(\times\) \(\mathfrak{A}\) state \(\Rightarrow\) \(\mathfrak{A}\) state)

\[ \Rightarrow (\text{type}\_\text{Person} \Rightarrow (\mathfrak{A},'c::null)\text{val}) \]

\[ \Rightarrow \text{oid} \]

\[ \Rightarrow (\mathfrak{A},'c::null)\text{val} \]

**where** deref-oid\_Person \textit{fst-snd} \textit{f oid} = (\(\lambda\ \tau. \text{case } \text{heap}(\text{fst-snd}\ \tau)\) \text{oid of}

\[ [\text{in}\_\text{Person}\ \text{obj}] \Rightarrow f\ \text{obj}\ \tau \]

\[ - \Rightarrow \text{invalid}\ \tau\]
**definition** deref-oid\textsubscript{OclAny} :: (\textit{A} state × \textit{A} state ⇒ \textit{A} state) = (\textit{type}_{\textit{OclAny}} ⇒ (\textit{A}, 'c::null)val) ⇒ oid = (\textit{A}, 'c::null)val

where deref-oid\textsubscript{OclAny} \textit{fst-snd} \textit{oid} = (\lambda \tau. \text{case heap}(\textit{fst-snd} \tau)) \textit{oid} of

- \textit{inOclAny} \textit{obj} ⇒ \textit{f obj} \tau
- ⇒ \text{invalid} \tau

pointer undefined in state or not referencing a type conform object representation

**definition** select\textsubscript{OclAny}ANY \textit{f} = (\lambda X. \text{case } X \text{ of }

\textit{mkOclAny} - \bot) ⇒ \text{null}
\textit{mkOclAny} - \{\textit{any}\} \Rightarrow \textit{f} (\lambda x. \{\{x\}\}) \text{any}

**definition** select\textsubscript{Person}BOSS \textit{f} = (\lambda X. \text{case } X \text{ of }

\textit{mkPerson} - - \bot) ⇒ \text{null} (* object contains null pointer *)
\textit{mkPerson} - - \{\textit{boss}\} ⇒ \textit{f} (\lambda x. \{\{x\}\}) \text{boss}

**definition** select\textsubscript{Person}SALARY \textit{f} = (\lambda X. \text{case } X \text{ of }

\textit{mkPerson} - - \bot - \bot) ⇒ \text{null}
\textit{mkPerson} - - \{\textit{salary}\} - \bot) ⇒ \textit{f} (\lambda x. \{\{x\}\}) \text{salary}

**definition** \textit{in-pre-state} = \textit{fst}
**definition** \textit{in-post-state} = \textit{snd}

**definition** reconst-basetype = (\lambda \textit{convert} \textit{x}. \textit{convert} \textit{x})

**definition** dot\textsubscript{OclAny}ANY :: OclAny ⇒ - ((1\textit{-}.\textit{any}) 50)
where \textit{(X).any} = \text{eval-extract} \textit{X}
(deref-oid\textsubscript{OclAny} \textit{in-post-state}
(select\textsubscript{OclAny}ANY\textit{Y}
reconst-basetype))

**definition** dot\textsubscript{Person}BOSS :: Person ⇒ Person ((1\textit{-}.\textit{boss}) 50)
where \textit{(X).boss} = \text{eval-extract} \textit{X}
(deref-oid\textsubscript{Person} in-post-state
(select\textsubscript{Person}BOSS
(deref-oid\textsubscript{Person} in-post-state)))

**definition** dot\textsubscript{Person}SALARY :: Person ⇒ Integer ((1\textit{-}.\textit{salary}) 50)
where \textit{(X).salary} = \text{eval-extract} \textit{X}
(deref-oid\textsubscript{Person} in-post-state
(select\textsubscript{Person}SALARY
reconst-basetype))

**definition** dot\textsubscript{OclAny}ANY-at-pre :: OclAny ⇒ - ((1\textit{-}.\textit{any}@pre) 50)
where \textit{(X).any}@\textit{pre} = \text{eval-extract} \textit{X}

198
(deref-oid_OclAny in-pre-state
(select_OclAny AX
reconst-basetype))

definition dot_Person_BOSS-at-pre:: Person ⇒ Person ((1(-).boss@pre 50))
where (X).boss@pre = eval-extract X
(deref-oid_Person in-pre-state
(select_PersonBOSS
(deref-oid_Person in-pre-state)))

definition dot_Person_SALARY-at-pre:: Person ⇒ Integer ((1(-).salary@pre 50))
where (X).salary@pre = eval-extract X
(deref-oid_Person in-pre-state
(select_PersonSALARY
reconst-basetype))

lemmas [simp] =
dot_OclAnyANY-def
dot_PersonBOSS-def
dot_PersonSALARY-def
dot_OclAnyANY-at-pre-def
dot_PersonBOSS-at-pre-def
dot_PersonSALARY-at-pre-def

Context Passing

lemmas [simp] = eval-extract-def

lemma cp-dot_OclAnyANY: ((X).any) τ = ((λ-. X τ).any) τ by simp
lemma cp-dot_PersonBOSS: ((X).boss) τ = ((λ-. X τ).boss) τ by simp
lemma cp-dot_PersonSALARY: ((X).salary) τ = ((λ-. X τ).salary) τ by simp

lemma cp-dot_OclAnyANY-at-pre: ((X).any@pre) τ = ((λ-. X τ).any@pre) τ by simp
lemma cp-dot_PersonBOSS-at-pre: ((X).boss@pre) τ = ((λ-. X τ).boss@pre) τ by simp
lemma cp-dot_PersonSALARY-at-pre: ((X).salary@pre) τ = ((λ-. X τ).salary@pre) τ by simp

lemmas cp-dot_OclAnyANY-I [simp, intro]=
cp-dot_OclAnyANY[THEN allI[THEN allI],
of λ X -. X λ - τ. τ, THEN cpII]

lemmas cp-dot_OclAnyANY-at-pre-I [simp, intro]=
cp-dot_OclAnyANY-at-pre[THEN allI[THEN allI],
of λ X -. X λ - τ. τ, THEN cpII]

lemmas cp-dot_PersonBOSS-I [simp, intro]=
cp-dot_PersonBOSS[THEN allI[THEN allI],
of λ X -. X λ - τ. τ, THEN cpII]

lemmas cp-dot_PersonBOSS-at-pre-I [simp, intro]=
cp-dot_PersonBOSS-at-pre[THEN allI[THEN allI],
of λ X -. X λ - τ. τ, THEN cpII]

199
lemmas cp-dot_person\textsc{salary}\_I \ [simp, intro!] = \\
cp-dot_person\textsc{salary}[\textsc{THEN} all][\textsc{THEN} all], \\
of \, \lambda \, X \cdot \, X \, \lambda \cdot \, \tau. \, \tau, \, \textsc{THEN} \, \text{cpI]
lemmas cp-dot_person\textsc{salary}\_at-pre-I \ [simp, intro!] = \\
cp-dot_person\textsc{salary}\_at-pre[\textsc{THEN} all][\textsc{THEN} all], \\
of \, \lambda \, X \cdot \, X \, \lambda \cdot \, \tau. \, \tau, \, \textsc{THEN} \, \text{cpI]

Execution with Invalid or Null as Argument

lemma dot_{Ocl\textsc{Any}}\textsc{any}\_nullstrict \ [simp]: \ (null).\textsc{any} = \text{invalid} \\
by \text{(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)}
lemma dot_{Ocl\textsc{Any}}\textsc{any}\_at-pre-nullstrict \ [simp] : \ (null).\textsc{any}@pre = \text{invalid} \\
by \text{(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)}
lemma dot_{Ocl\textsc{Any}}\textsc{any}\_strict \ [simp] : \ (invalid).\textsc{any} = \text{invalid} \\
by \text{(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)}
lemma dot_{Ocl\textsc{Any}}\textsc{any}\_at-pre-strict \ [simp] : \ (invalid).\textsc{any}@pre = \text{invalid} \\
by \text{(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)}

lemma dot_person\textsc{boss}\_nullstrict \ [simp] : \ (null).\textsc{boss} = \text{invalid} \\
by \text{(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)}
lemma dot_person\textsc{boss}\_at-pre-nullstrict \ [simp] : \ (null).\textsc{boss}@pre = \text{invalid} \\
by \text{(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)}
lemma dot_person\textsc{boss}\_strict \ [simp] : \ (invalid).\textsc{boss} = \text{invalid} \\
by \text{(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)}
lemma dot_person\textsc{boss}\_at-pre-strict \ [simp] : \ (invalid).\textsc{boss}@pre = \text{invalid} \\
by \text{(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)}

lemma dot_person\textsc{salary}\_nullstrict \ [simp] : \ (null).\textsc{salary} = \text{invalid} \\
by \text{(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)}
lemma dot_person\textsc{salary}\_at-pre-nullstrict \ [simp] : \ (null).\textsc{salary}@pre = \text{invalid} \\
by \text{(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)}
lemma dot_person\textsc{salary}\_strict \ [simp] : \ (invalid).\textsc{salary} = \text{invalid} \\
by \text{(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)}
lemma dot_person\textsc{salary}\_at-pre-strict \ [simp] : \ (invalid).\textsc{salary}@pre = \text{invalid} \\
by \text{(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def)}

7.1.9. A Little Infra-structure on Example States

The example we are defining in this section comes from the figure 7.2

\begin{align*}
definition \text{OclInt1000} \ (1000) \ & \text{where} \quad \text{OclInt1000} = (\lambda \cdot. \ [(1000)]) \\
definition \text{OclInt1200} \ (1200) \ & \text{where} \quad \text{OclInt1200} = (\lambda \cdot. \ [(1200)]) \\
definition \text{OclInt1300} \ (1300) \ & \text{where} \quad \text{OclInt1300} = (\lambda \cdot. \ [(1300)]) \\
definition \text{OclInt1800} \ (1800) \ & \text{where} \quad \text{OclInt1800} = (\lambda \cdot. \ [(1800)]) \\
definition \text{OclInt2600} \ (2600) \ & \text{where} \quad \text{OclInt2600} = (\lambda \cdot. \ [(2600)]) \\
definition \text{OclInt2900} \ (2900) \ & \text{where} \quad \text{OclInt2900} = (\lambda \cdot. \ [(2900)]) \\
definition \text{OclInt3200} \ (3200) \ & \text{where} \quad \text{OclInt3200} = (\lambda \cdot. \ [(3200)])
\end{align*}
Figure 7.2.: (a) pre-state $\sigma_1$ and (b) post-state $\sigma'_1$.

definition $OclInt3500$ ($3500$) where $OclInt3500 = (\lambda \cdot [3500])$

definition $oid0 \equiv 0$
definition $oid1 \equiv 1$
definition $oid2 \equiv 2$
definition $oid3 \equiv 3$
definition $oid4 \equiv 4$
definition $oid5 \equiv 5$
definition $oid6 \equiv 6$
definition $oid7 \equiv 7$
definition $oid8 \equiv 8$
definition $person1 \equiv mk_Person oid0 [1300] [oid1]$
definition $person2 \equiv mk_Person oid1 [1800] [oid1]$
definition $person3 \equiv mk_Person oid2 None None$
definition $person4 \equiv mk_Person oid3 [2900] [oid4]$
definition $person5 \equiv mk_Person oid4 [3500] None$
definition $person6 \equiv mk_Person oid5 [2500] [oid6]$
definition $person7 \equiv mk_OclAny oid6 [([3200], [oid6])]
definition $person8 \equiv mk_OclAny oid7 None$
definition $person9 \equiv mk_Person oid8 [0] None$
definition $\sigma_1 \equiv () \text{ heap = empty(oid0 } \mapsto in_Person (mk_Person oid0 [1000] [oid1]))$
\hspace{1cm} (oid1 } \mapsto in_Person (mk_Person oid1 [1200] None))
\hspace{1cm} (*oid2*)
\hspace{1cm} (oid3 } \mapsto in_Person (mk_Person oid3 [2600] [oid4]))
\hspace{1cm} (oid4 } \mapsto in_Person person5)
\hspace{1cm} (oid5 } \mapsto in_Person (mk_Person oid5 [2300] [oid3]))
\hspace{1cm} (*oid6*)
\hspace{1cm} (*oid7*)
\hspace{1cm} (oid8 } \mapsto in_Person person9),$
\hspace{1cm} assocx2 = empty,
\hspace{1cm} assocx3 = empty$}$

definition
\[ \sigma_1' \equiv (\langle \text{heap} = \text{empty}(oid0 \mapsto \text{in}_{\text{Person}} \text{person1}) \\
(oid1 \mapsto \text{in}_{\text{Person}} \text{person2}) \\
(oid2 \mapsto \text{in}_{\text{Person}} \text{person3}) \\
(oid3 \mapsto \text{in}_{\text{Person}} \text{person4}) \\
(\ast oid4 \ast) \\
(oid5 \mapsto \text{in}_{\text{Person}} \text{person6}) \\
(oid6 \mapsto \text{in}_{\text{OclAny}} \text{person7}) \\
(oid7 \mapsto \text{in}_{\text{OclAny}} \text{person8}) \\
(oid8 \mapsto \text{in}_{\text{Person}} \text{person9}), \\
\ast assocs_2 = \text{empty}, \\
\ast assocs_3 = \text{empty} \rangle ) \]

**definition** \[ \sigma_0 \equiv (\langle \text{heap} = \text{empty}, \text{assocs}_2 = \text{empty}, \text{assocs}_3 = \text{empty} \rangle ) \]

**lemma** basic-\(\tau\)-wff: \(WFF(\sigma_1, \sigma_1')\)

**by** (auto simp: \(WFF\)-def \(\sigma_1\)-def \(\sigma_1'\)-def

oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def
oid-of-\(\ast\)-def oid-of-type \(\text{Person}\)-def oid-of-type \(\text{OclAny}\)-def
person1-def person2-def person3-def person4-def
person5-def person6-def person7-def person8-def person9-def)

**lemma** [simp, code-unfold]: \(\text{dom}(\text{heap } \sigma_1) = \{oid0, oid1, (\ast, oid2 \ast)oid3, oid4, oid5(\ast, oid6, oid7 \ast), oid8\}\)

**by** (auto simp: \(\sigma_1\)-def)

**lemma** [simp, code-unfold]: \(\text{dom}(\text{heap } \sigma_1') = \{oid0, oid1, oid2, oid3, (\ast, oid4 \ast)oid5, oid6, oid7, oid8\}\)

**by** (auto simp: \(\sigma_1'\)-def)

**definition** \(X_{\text{Person}1} :: \text{Person} \equiv \lambda - .[| \text{person1} |]

**definition** \(X_{\text{Person}2} :: \text{Person} \equiv \lambda - .[| \text{person2} |]

**definition** \(X_{\text{Person}3} :: \text{Person} \equiv \lambda - .[| \text{person3} |]

**definition** \(X_{\text{Person}4} :: \text{Person} \equiv \lambda - .[| \text{person4} |]

**definition** \(X_{\text{Person}5} :: \text{Person} \equiv \lambda - .[| \text{person5} |]

**definition** \(X_{\text{Person}6} :: \text{Person} \equiv \lambda - .[| \text{person6} |]

**definition** \(X_{\text{Person}7} :: \text{OclAny} \equiv \lambda - .[| \text{person7} |]

**definition** \(X_{\text{Person}8} :: \text{OclAny} \equiv \lambda - .[| \text{person8} |]

**definition** \(X_{\text{Person}9} :: \text{Person} \equiv \lambda - .[| \text{person9} |]

**lemma** [code-unfold]: \((x :: \text{Person}) \equiv y) = \text{StrictRefEqObject} x y

**by** (simp only: \text{StrictRefEqObject}-\text{Person})

**lemma** [code-unfold]: \((x :: \text{OclAny}) \equiv y) = \text{StrictRefEqObject} x y

**by** (simp only: \text{StrictRefEqObject}-\text{OclAny})

**lemmas** [simp, code-unfold] =

\(\text{OclAsType}_{\text{OclAny}} - \text{OclAny}\)

\(\text{OclAsType}_{\text{OclAny}} - \text{Person}\)

\(\text{OclAsType}_{\text{Person}} - \text{OclAny}\)

\(\text{OclAsType}_{\text{Person}} - \text{Person}\)
\textit{OclIsTypeOf} \textit{OclAny-Person}\n\textit{OclIsTypeOf} \textit{Person-OclAny}\n\textit{OclIsTypeOf} \textit{Person-Person}\n\textit{OclIsKindOf} \textit{OclAny-Person}\n\textit{OclIsKindOf} \textit{Person-Person}\n\textit{OclIsKindOf} \textit{Person-OclAny}\n\textit{OclIsKindOf} \textit{Person-Person}\n
\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person1}} \text{.salary} \Leftrightarrow 1000)\n\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person1}} \text{.salary} = 1300)\n\text{value } \bigwedge (s_{\text{post}}, (s_{\text{post}}, 1)) = (X_{\text{Person1}} \text{.salary@pre} \geq 1000)\n\text{value } \bigwedge (s_{\text{post}}, (s_{\text{post}}, 1)) = (X_{\text{Person1}} \text{.salary@pre} \Leftrightarrow 1300)\n\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person1}} \text{.boss} \Leftrightarrow X_{\text{Person1}})\n\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person1}} \text{.boss} \text{.salary} \Leftrightarrow 1800)\n\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person1}} \text{.boss} \text{.boss} \Leftrightarrow X_{\text{Person1}})\n\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person1}} \text{.boss} \text{.boss} \Leftrightarrow X_{\text{Person2}})\n\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person1}} \text{.boss} @ \text{pre} \text{.salary} \Leftrightarrow X_{\text{Person2}})\n\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person1}} \text{.boss} @ \text{pre} \text{.boss} \Leftrightarrow X_{\text{Person2}})\n\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person1}} \text{.boss} \text{.boss} \Leftrightarrow X_{\text{Person2}})\n\text{value } \bigwedge (s_{\text{post}}, (s_{\text{post}}, 1)) = (X_{\text{Person1}} \text{.boss} \text{.boss} \Leftrightarrow X_{\text{Person2}})\n\text{value } \bigwedge (s_{\text{post}}, (s_{\text{post}}, 1)) = (X_{\text{Person1}} \text{.boss} \text{.boss} \Leftrightarrow \text{null})\n\text{value } \bigwedge (s_{\text{post}}, (s_{\text{post}}, 1)) = (X_{\text{Person1}} \text{.boss} \text{.boss} \Leftrightarrow \text{null})\n
\text{lemma} \ (s_{\text{pre}}, s_{\text{post}}) = (X_{\text{Person1}} \text{.oclIsMaintained}())\n\text{by (simp add: OclValid-def OclIsMaintained-def} )\n\text{\sigma_1-def \sigma_1'-def} \n\text{X_{\text{Person1-def person1-def} } } \n\text{oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def} \n\text{oid-of-option-def oid-of-type \textit{Person-def} ) } \n\text{lemma} \ (s_{\text{pre}}, s_{\text{post}}) = (X_{\text{Person1}} \text{.oclAsType} (\textit{OclAny}) \text{.oclAsType} (\textit{Person})) \n\text{by (rule up-down-cast-Person-OclAny-Person')} \n\text{simp add: X_{\text{Person1-def} }) } \n\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person1}} \text{.oclIsTypeOf} (\textit{Person})) \n\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person1}} \text{.oclIsTypeOf} (\textit{OclAny})) \n\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person1}} \text{.oclIsKindOf} (\textit{Person})) \n\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person1}} \text{.oclIsKindOf} (\textit{OclAny})) \n\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person1}} \text{.oclIsTypeOf} (\textit{OclAny})) \n\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person2}} \text{.salary} \Leftrightarrow 1800) \n\text{value } \bigwedge (s_{\text{post}}, (s_{\text{post}}, 1)) = (X_{\text{Person2}} \text{.salary@pre} \Leftrightarrow 1200) \n\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person2}} \text{.boss} \Leftrightarrow X_{\text{Person2}}) \n\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person2}} \text{.boss} \text{.salary@pre} \Leftrightarrow 1200) \n\text{value } \bigwedge (s_{\text{pre}}, (s_{\text{pre}}, 1)) = (X_{\text{Person2}} \text{.boss} \text{.boss@pre} \Leftrightarrow \text{null})
\[
\begin{align*}
\text{value} \land & \quad s_{\text{post}}: (\sigma_1,s_{\text{post}}) \models (X_{\text{Person}^2}.\text{boss@pre} \doteq \text{null}) \\
\text{value} \land & \quad s_{\text{post}}: (\sigma_1,s_{\text{post}}) \models (X_{\text{Person}^2}.\text{boss@pre} \leftrightarrow X_{\text{Person}^2}) \\
\text{value} & \quad (\sigma_1,\sigma_1') \models (X_{\text{Person}^2}.\text{boss@pre} \leftrightarrow (X_{\text{Person}^2}.\text{boss})) \\
\text{value} \land & \quad s_{\text{post}}: (\sigma_1,s_{\text{post}}) \models \neg \tau(X_{\text{Person}^2}.\text{boss@pre} \cdot \text{boss}) \\
\text{value} \land & \quad s_{\text{post}}: (\sigma_1,s_{\text{post}}) \models \neg \tau(X_{\text{Person}^2}.\text{boss@pre} \cdot \text{salary@pre}) \\
\text{lemma} & \quad (\sigma_1,\sigma_1') \models (X_{\text{Person}^2}.\text{oclIsMaintained}()) \\
\text{by (simp add: OclValid-def OclIsMaintained-def} & \\
\quad \quad \quad \quad \quad \quad \sigma_1\text{-def } \sigma_1'\text{-def} \\
\quad \quad \quad \quad \quad \quad X_{\text{Person}^2}\text{-def person2-def} \\
\quad \quad \quad \quad \quad \quad \text{oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def} \\
\quad \quad \quad \quad \quad \quad \text{oid-of-option-def oid-of-typePerson-def}) \\
\text{value} \land & \quad s_{\text{pre}} : (s_{\text{pre}},\sigma_1') \models (X_{\text{Person}^3}.\text{salary} \doteq \text{null}) \\
\text{value} \land & \quad s_{\text{post}}: (\sigma_1,s_{\text{post}}) \models \neg \tau(X_{\text{Person}^3}.\text{salary@pre}) \\
\text{value} \land & \quad s_{\text{pre}} : (s_{\text{pre}},\sigma_1') \models (X_{\text{Person}^3}.\text{boss} \doteq \text{null}) \\
\text{value} \land & \quad s_{\text{pre}} : (s_{\text{pre}},\sigma_1') \models \neg \tau(X_{\text{Person}^3}.\text{boss} \cdot \text{salary}) \\
\text{value} \land & \quad s_{\text{post}}: (\sigma_1,s_{\text{post}}) \models \neg \tau(X_{\text{Person}^3}.\text{boss@pre}) \\
\text{lemma} & \quad (\sigma_1,\sigma_1') \models (X_{\text{Person}^3}.\text{oclIsNew}()) \\
\text{by (simp add: OclValid-def OclIsNew-def} & \\
\quad \quad \quad \quad \quad \quad \sigma_1\text{-def } \sigma_1'\text{-def} \\
\quad \quad \quad \quad \quad \quad X_{\text{Person}^3}\text{-def person3-def} \\
\quad \quad \quad \quad \quad \quad \text{oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid8-def} \\
\quad \quad \quad \quad \quad \quad \text{oid-of-option-def oid-of-typePerson-def}) \\
\text{value} \land & \quad s_{\text{post}}: (\sigma_1,s_{\text{post}}) \models (X_{\text{Person}^4}.\text{boss@pre} \doteq X_{\text{Person}^5}) \\
\text{value} & \quad (\sigma_1,\sigma_1') \models \neg \tau(X_{\text{Person}^4}.\text{boss@pre} \cdot \text{salary@pre}) \\
\text{value} \land & \quad s_{\text{post}}: (\sigma_1,s_{\text{post}}) \models (X_{\text{Person}^4}.\text{boss@pre} \cdot \text{salary@pre} \doteq 3500) \\
\text{lemma} & \quad (\sigma_1,\sigma_1') \models (X_{\text{Person}^4}.\text{oclIsMaintained}()) \\
\text{by (simp add: OclValid-def OclIsMaintained-def} & \\
\quad \quad \quad \quad \quad \quad \sigma_1\text{-def } \sigma_1'\text{-def} \\
\quad \quad \quad \quad \quad \quad X_{\text{Person}^4}\text{-def person4-def} \\
\quad \quad \quad \quad \quad \quad \text{oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def} \\
\quad \quad \quad \quad \quad \quad \text{oid-of-option-def oid-of-typePerson-def}) \\
\text{value} \land & \quad s_{\text{pre}} : (s_{\text{pre}},\sigma_1') \models \neg \tau(X_{\text{Person}^5}.\text{salary}) \\
\text{value} \land & \quad s_{\text{post}}: (\sigma_1,s_{\text{post}}) \models (X_{\text{Person}^5}.\text{salary@pre} \doteq 3500) \\
\text{value} \land & \quad s_{\text{pre}} : (s_{\text{pre}},\sigma_1') \models \neg \tau(X_{\text{Person}^5}.\text{boss}) \\
\text{lemma} & \quad (\sigma_1,\sigma_1') \models (X_{\text{Person}^5}.\text{oclIsDeleted}()) \\
\text{by (simp add: OclNot-def OclValid-def OclIsDeleted-def} & \\
\quad \quad \quad \quad \quad \quad \sigma_1\text{-def } \sigma_1'\text{-def} \\
\quad \quad \quad \quad \quad \quad X_{\text{Person}^5}\text{-def person5-def} \\
\quad \quad \quad \quad \quad \quad \text{oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def} \\
\quad \quad \quad \quad \quad \quad \text{oid-of-option-def oid-of-typePerson-def})
\end{align*}
\]
value \( \wedge \sigma_{\text{pre}} \). \( (\sigma_{\text{pre}}, \sigma_1') \models \lnot (\nu(\text{Person}_6. \text{boss} \cdot \text{salary} \neg \text{pre})) \)

value \( \wedge \sigma_{\text{post}} \). \( (\sigma_1, \sigma_{\text{post}}) \models (\text{Person}_6. \text{boss} \cdot \text{pre} \equiv \text{Person}_4) \)

value \( \wedge \sigma_{\text{post}} \). \( (\sigma_1, \sigma_{\text{post}}) \models (\text{Person}_6. \text{boss} \cdot \text{pre} \cdot \text{salary} \equiv 2900) \)

value \( \wedge \sigma_{\text{post}} \). \( (\sigma_1, \sigma_{\text{post}}) \models (\text{Person}_6. \text{boss} \cdot \text{pre} \cdot \text{salary} \equiv 2600) \)

value \( \wedge \sigma_{\text{post}} \). \( (\sigma_1, \sigma_{\text{post}}) \models (\text{Person}_6. \text{boss} \cdot \text{pre} \cdot \text{boss} \cdot \text{pre} \equiv \text{Person}_5) \)

lemma \( \wedge \sigma_{\text{pre}} \). \( (\sigma_1, \sigma_1') \models (\text{Person}_6. \text{OclValid}) \)

by iso_decl add: OclValid-def OclIsMaintained-def
\( \sigma_1\text{-def} \sigma_1'\text{-def} \)
\( \text{Person}_6\text{-def} \text{person}_6\text{-def} \)
\( \text{oid}_0\text{-def} \text{oid}_1\text{-def} \text{oid}_2\text{-def} \text{oid}_3\text{-def} \text{oid}_4\text{-def} \text{oid}_5\text{-def} \text{oid}_6\text{-def} \)
\( \text{oid-of-option-def} \text{oid-of-type Person-def} \)

value \( \wedge \sigma_{\text{pre}} \). \( (\sigma_{\text{pre}}, \sigma_{\text{post}}) \models \nu(\text{Person}_7. \text{oclAsType}(\text{Person})) \)

value \( \wedge \sigma_{\text{post}} \). \( (\sigma_1, \sigma_{\text{post}}) \models \lnot (\nu(\text{Person}_7. \text{oclAsType}(\text{Person}) \cdot \text{boss} \cdot \text{pre})) \)

lemma \( \wedge \sigma_{\text{pre}} \). \( (\sigma_{\text{pre}}, \sigma_{\text{post}}) \models ((\text{Person}_7. \text{oclAsType}(\text{Person}) \cdot \text{oclAsType}(\text{OclAny}) \cdot \text{oclAsType}(\text{Person})) \)

by rule up-down-cast-Person-OclAny-Person', iso_decl add: Person7-def OclValid-def valid-def
\( \text{person}_7\text{-def} \)

lemma \( \wedge \sigma_{\text{pre}} \). \( (\sigma_1, \sigma_1') \models (\text{Person}_7. \text{OclIsNew}) \)

by iso_decl add: OclValid-def OclIsNew-def
\( \sigma_1\text{-def} \sigma_1'\text{-def} \)
\( \text{OclIsNew-def} \text{Person7-def} \)
\( \text{oid}_0\text{-def} \text{oid}_1\text{-def} \text{oid}_2\text{-def} \text{oid}_3\text{-def} \text{oid}_4\text{-def} \text{oid}_5\text{-def} \text{oid}_6\text{-def} \text{oid}_8\text{-def} \)
\( \text{oid-of-option-def} \text{oid-of-type OclAny-def} \)

value \( \wedge \sigma_{\text{pre}} \). \( (\sigma_{\text{pre}}, \sigma_{\text{post}}) \models (\text{Person}_8 \equiv \text{Person}_7) \)

value \( \wedge \sigma_{\text{pre}} \). \( (\sigma_{\text{pre}}, \sigma_{\text{post}}) \models \lnot (\nu(\text{Person}_8. \text{oclIsTypeOf}(\text{OclAny}))) \)

value \( \wedge \sigma_{\text{pre}} \). \( (\sigma_{\text{pre}}, \sigma_{\text{post}}) \models (\text{Person}_8. \text{oclIsTypeOf}(\text{Person})) \)

value \( \wedge \sigma_{\text{pre}} \). \( (\sigma_{\text{pre}}, \sigma_{\text{post}}) \models (\text{Person}_8. \text{oclIsKindOf}(\text{Person})) \)

value \( \wedge \sigma_{\text{pre}} \). \( (\sigma_{\text{pre}}, \sigma_{\text{post}}) \models (\text{Person}_8. \text{oclIsKindOf}(\text{OclAny})) \)

lemma \( \sigma\text{-modifiedonly}: (\sigma_1, \sigma_1') \models (\text{Set}\{\text{Person}_1. \text{oclAsType}(\text{OclAny}) \}
\, \text{Person}_2. \text{oclAsType}(\text{OclAny})
\, \text{Person}_3. \text{oclAsType}(\text{OclAny})\}
\, (\ast, \text{Person}_4. \text{oclAsType}(\text{OclAny})\}
\, (\ast, \text{Person}_5. \text{oclAsType}(\text{OclAny})\}
\, (\ast, \text{Person}_6. \text{oclAsType}(\text{OclAny})\}
\, (\ast, \text{Person}_7. \text{oclAsType}(\text{OclAny})\}
\, (\ast, \text{Person}_8. \text{oclAsType}(\text{OclAny})\}
\, (\ast, \text{Person}_9. \text{oclAsType}(\text{OclAny})\}
\rightarrow \text{OclIsModifiedOnly}()) \)

apply iso_decl add: OclIsModifiedOnly-def OclValid-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def
X_person1-def X_person2-def X_person3-def X_person4-def
X_person5-def X_person6-def X_person7-def X_person8-def X_person9-def
person1-def person2-def person3-def person4-def
person5-def person6-def person7-def person8-def person9-def
image-def)
apply(simp add: OclIncluding-rep-set mtSet-rep-set null-option-def bot-option-def)
apply(simp add: oid-of-option-def oid-of-typeOclAny-def, clarsimp)
apply(simp add: σ1-def σ1′-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)
done

lemma (σ1,σ1′) \models (((X_{\text{Person}9} @pre (\lambda x. \text{OclAsType}_{\text{Person} \cdot \mathcal{A}} x))) \triangleq X_{\text{Person}9})
by(simp add: OclSelf-at-pre-def σ1-def oid-of-option-def oid-of-type_{\text{Person}-def}
X_{\text{Person}9-def person9-def oid8-def OclValid-def StrongEq-def OclAsType_{\text{Person} \cdot \mathcal{A}-def})

lemma (σ1,σ1′) \models (((X_{\text{Person}9} \cdot \text{OclAsType}(\text{OclAny})) @pre (\lambda x. \text{OclAsType}_{\text{OclAny} \cdot \mathcal{A}} x))) \triangleq
((X_{\text{Person}9} \cdot \text{OclAsType}(\text{OclAny})) @post (\lambda x. \text{OclAsType}_{\text{OclAny} \cdot \mathcal{A}} x)))

proof –

have including4 : \forall a b c d \tau.
\qquad \text{Set}(\lambda \tau. [[a]], \lambda \tau. [[b]], \lambda \tau. [[c]], \lambda \tau. [[d]]) \tau = Abs-Set-0 [[[[a]], [[[b]], [[c]], [[d]]]]] \tau = Abs-Set-0 [[a]],[[b]],[[c]],[[d]]]
apply(subst subst-rep-simp'[symmetric], simp)
by(simp add: OclIncluding-rep-set mtSet-rep-set)

have excluding1: \forall S a b c d e \tau.
\qquad (\lambda\tau. Abs-Set-0 [[a],[b],[c],[d]]) \rightarrow excluding(\lambda\tau. [[e]]) \tau = Abs-Set-0 [[a],[b],[c],[d]] - [[e]]
apply(simp add: OclExcluding-def)
apply(simp add: defined-def OclValid-def false-def true-def
bot-fun-def bot-Set-0-def null-fun-def null-Set-0-def)
apply(rule cong) apply(rule impI, subst (asm) Abs-Set-0-inject) apply (simp add: bot-option-def)+
apply(rule cong) apply(rule impI, subst (asm) Abs-Set-0-inject) apply (simp add: bot-option-def
null-option-def)+
apply(subst Abs-Set-0-inverse, simp add: bot-option-def, simp)
done

show ?thesis
apply(rule framing)[where \( X = \text{Set}\{ X_{\text{Person}1} \cdot \text{OclAsType}(\text{OclAny}) \),
, X_{\text{Person}2} \cdot \text{OclAsType}(\text{OclAny})
(*, X_{\text{Person}3} \cdot \text{OclAsType}(\text{OclAny})*)
, X_{\text{Person}4} \cdot \text{OclAsType}(\text{OclAny})]

206
apply\(cut-tac\ \sigma\)-modifiedonly

apply\(simp\ only\): \textit{OclValid-def}
\(\textit{Person_1-def} \textit{Person_2-def} \textit{Person_3-def} \textit{Person_4-def}
\textit{Person_5-def} \textit{Person_6-def} \textit{Person_7-def} \textit{Person_8-def} \textit{Person_9-def}
\textit{OclAsType(OclAny-Person)}

apply\(\textit{subst\ cp-OclIsModifiedOnly, subst\ cp-OclExcluding,}
\textit{subst\ (asm) cp-OclIsModifiedOnly, simp\ add: including4\ excluding1})

apply\(\textit{simp\ only\}: \textit{Person_1-def} \textit{Person_2-def} \textit{Person_3-def} \textit{Person_4-def}
\textit{Person_5-def} \textit{Person_6-def} \textit{Person_7-def} \textit{Person_8-def} \textit{Person_9-def}
\textit{OclValid-def}
\textit{null-option-def\ bot-option-def})

done

qed

\textbf{lemma perm-\(\sigma_1\)' : \(\sigma_1\)' = (\emptyset \ heap = \textit{empty})}
\(\textit{oid8} \mapsto \textit{in\_Person person9})
\(\textit{oid7} \mapsto \textit{in\_OclAny person8})
\(\textit{oid6} \mapsto \textit{in\_OclAny person7})
\(\textit{oid5} \mapsto \textit{in\_Person person6})
\(*\textit{oid4}*\)
\(\textit{oid3} \mapsto \textit{in\_Person person4})
\(\textit{oid2} \mapsto \textit{in\_Person person3})
\(\textit{oid1} \mapsto \textit{in\_Person person2})
\(\textit{oid0} \mapsto \textit{in\_Person person1})
\textit{\textbf{, assocs}_2 = assocs_2 \ \sigma_1'}
\textit{\textbf{, assocs}_3 = assocs_3 \ \sigma_1'}\)

qed

declare const-ss [simp]

lemma \( \sigma_1 \setminus \sigma_1' \)

\[ \{ \text{Person. allInstances()} \} \models (\text{Set} \{ X_{\text{Person}}1, X_{\text{Person}}2, X_{\text{Person}}3, X_{\text{Person}}4(*, X_{\text{Person}}5*), X_{\text{Person}}6, X_{\text{Person}}7.oclAsType(Person)(*, X_{\text{Person}}8*), X_{\text{Person}}9 \}) \]

apply(subst perm-\( \sigma_1' \))
apply(simp only: oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def X_{\text{Person}}1-def X_{\text{Person}}2-def X_{\text{Person}}3-def X_{\text{Person}}4-def X_{\text{Person}}5-def X_{\text{Person}}6-def X_{\text{Person}}7-def X_{\text{Person}}8-def X_{\text{Person}}9-def person7-def)
apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsType_person-\( \mathcal{A} \)-def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule OclIncluding-cong, simp, simp)
apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsType_person-\( \mathcal{A} \)-def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule OclIncluding-cong, simp, simp)
apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsType_person-\( \mathcal{A} \)-def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, simp, rule OclIncluding-cong, simp, simp)
apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsType_person-\( \mathcal{A} \)-def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, simp, rule OclIncluding-cong, simp, simp)
apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsType_person-\( \mathcal{A} \)-def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, simp, rule OclIncluding-cong, simp, simp)
apply(subst state-update-vs-allInstances-at-post-ntc, simp, simp add: OclAsType_person-\( \mathcal{A} \)-def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, simp)
apply(subst state-update-vs-allInstances-at-post-ntc, simp, simp add: OclAsType_person-\( \mathcal{A} \)-def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, simp, rule OclIncluding-cong, simp, simp)
apply(rule state-update-vs-allInstances-at-post-empty)
by(simp-all add: OclAsType_person-\( \mathcal{A} \)-def)

lemma \( \sigma_1 \).
(σ₁, σ₁') |=(OclAny.allInstances()) ⊆ \{ X_{\text{Person}1}.oclAsType(OclAny), X_{\text{Person}2}.oclAsType(OclAny),
X_{\text{Person}3}.oclAsType(OclAny), X_{\text{Person}4}.oclAsType(OclAny),
X_{\text{Person}5}.oclAsType(OclAny), X_{\text{Person}6}.oclAsType(OclAny),
X_{\text{Person}7}, X_{\text{Person}8}, X_{\text{Person}9}.oclAsType(OclAny) \}

apply(subst perm-σ₁) 
apply(simp only: oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def
X_{\text{Person}1}.oclAsType(X_{\text{Person}2}.oclAsType(X_{\text{Person}3}.oclAsType(X_{\text{Person}4}.oclAsType(X_{\text{Person}5}.oclAsType(X_{\text{Person}6}.oclAsType(X_{\text{Person}7}.oclAsType(X_{\text{Person}8}.oclAsType(X_{\text{Person}9}.oclAsType(OclAny))

X_{\text{Person}1}.oclAsType(OclAny), X_{\text{Person}2}.oclAsType(OclAny),
X_{\text{Person}3}.oclAsType(OclAny), X_{\text{Person}4}.oclAsType(OclAny),
X_{\text{Person}5}.oclAsType(OclAny), X_{\text{Person}6}.oclAsType(OclAny),
X_{\text{Person}7}, X_{\text{Person}8}, X_{\text{Person}9}.oclAsType(OclAny) \})

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypeOclAny-X-def, simp, rule const-StrictRefEqSet-including, simp, simp, rule OclIncluding-cong, simp, simp)+
apply(rule state-update-vs-allInstances-at-post-empty)
by(simp-all add: OclAsTypeOclAny-X-def)
end

7.2. The Employee Design Model (OCL)

theory Employee-DesignModel-OCLElement
imports Employee-DesignModel-UMLPart
begin

7.2.1. Standard State Infrastructure

Ideally, these definitions are automatically generated from the class model.

7.2.2. Invariant

These recursive predicates can be defined conservatively by greatest fix-point constructions—automatically. See [4, 6] for details. For the purpose of this example, we state them as axioms here.

axiomatization inv-Person :: Person ⇒ Boolean
where A : (τ |= (δ self)) →
(τ |= inv-Person(self)) =
((τ |= (self . boss = null)) ∨
( τ |= (self . boss <> null) ∧ (τ |= ((self . salary) ‘≤ (self . boss . salary))) ∧
(τ |= (inv-Person(self . boss)))))

axiomatization inv-Person-at-pre :: Person ⇒ Boolean
where B : (τ |= (δ self)) →
(τ |= inv-Person-at-pre(self)) =
((τ |= (self . boss@pre = null)) ∨

(τ |= (self . boss@pre <> null) ∧
(τ |= (self . boss@pre . salary@pre ‘≤ self . salary@pre)) ∧
(τ |= (inv-Person-at-pre(self . boss@pre))))

A very first attempt to characterize the axiomatization by an inductive definition -
this can not be the last word since too weak (should be equality!)

coinductive inv :: Person ⇒ bool where
(τ |= (δ self)) =⇒ ((τ |= (self . boss = null)) ∨
(τ |= (self . boss <> null) ∧ (τ |= (self . boss . salary ‘≤ self . salary)) ∧
((inv(self . boss) τ )))
⇒ (inv self τ)

7.2.3. The Contract of a Recursive Query

The original specification of a recursive query :

context Person :: contents() : Set(Integer)
post: result = if self . boss = null
then Set{i}
else self . boss . contents()-> including(i)
endif

consts dot-contents :: Person ⇒ Set-Integer (((I(.-). contents’)) 50)

axiomatization where dot-contents-def:
(τ |= ((self). contents() ≜ result)) =
(if (δ self) τ = true τ
then ((τ |= true) ∧
(τ |= (result ≜ if (self . boss = null)
then (Set{self . salary})
else (self . boss . contents()-> including(self . salary)))
endif))
else τ |= result ≜ invalid)

consts dot-contents-AT-pre :: Person ⇒ Set-Integer (((I(.-). contents@pre’)) 50)

axiomatization where dot-contents-AT-pre-def:
(τ |= (self). contents@pre() ≜ result) =
(if (δ self) τ = true τ
then τ |= true ∧
τ |= (result ≜ if (self). boss@pre = null (* post *)
then Set{(self). salary@pre}
else (self). boss@pre . contents@pre() -> including(self . salary@pre)
endif)
else τ |= result ≜ invalid)

These @pre variants on methods are only available on queries, i.e., operations without
side-effect.
7.2.4. The Contract of a Method

The specification in high-level OCL input syntax reads as follows:

```ocl
class Person {
  operation insert: Integer -> void
}
operation contents: Set(Integer)

context Person::insert(x: Integer)
  post: contents(): Set(Integer)
  contents() = contents@pre()->including(x)
}

definition dot-insert :: Person ⇒ Integer ⇒ void
  (self.insert(x) = result)

axiomatization where dot-insert-def:
  if (δ self) τ = true ∧ (υ x) τ = true τ
  then τ |= true ∧
    τ |= ((self).contents() = (self).contents@pre()->including(x))
  else τ |= ((self).insert(x) = invalid))
```

end
Part IV.

Conclusion
8. Conclusion

8.1. Lessons Learned and Contributions

We provided a typed and type-safe shallow embedding of the core of UML \[31, 32\] and OCL \[33\]. Shallow embedding means that types of OCL were injectively, i.e., mapped by the embedding one-to-one to types in Isabelle/HOL \[27\]. 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. 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 3.5.2) 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 \[19\].

3. the complicated life resulting from the two necessary equalities: the standard’s “strict weak referential equality” as default (written \(\_ = \_\) 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 \(\_\rightarrow\text{oclIsModifiedOnly}()\) and its consequences for strong and weak equality.

\[1\]Our two examples of Employee\_DesignModel (see Chapter 7) 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 [25] for semi-automated code verification, and has been considered desirable in the Aachen Meeting [15]).

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().

### 8.2. Lessons Learned

While our paper and pencil arguments, given in [13], 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 [36] or SMT-solvers like Z3 [20] completely impractical. Concretely, if the expression not(null) is defined invalid (as is the case in the present standard [33]), than standard involution does not hold, i.e., not(not(A)) ≠ A does not hold universally. Similarly, if null and null is invalid, then not even idempotence X and X = X holds. We strongly argue in favor of a lattice-like organization, where null represents “more information” than invalid and the logical operators are monotone with respect to this semantical “information ordering.”

A similar experience with prior paper and pencil arguments was our investigation of the object-oriented data-models, in particular path-expressions [16]. 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 [14] for details)) are valid in Featherweight OCL.

8.3. 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 [9]. 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 [35]) 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 [14]. 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 [36] based animator in Isabelle can give a similar quality of animation as the OCLexec Tool [24].

- 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.
Bibliography


219


