Extended Version

Featherweight OCL

A Study for a Consistent Semantics of UML/OCL 2.3 in HOL

Achim D. Brucker Delphine Longuet Frédéric Tuong
Burkhart Wolff

November 14, 2013

Abstract

UML/OCL is one of the few modeling languages that is widely used in industry. Besides numerous diagrams describing various aspects of models, the core of the UML, the language OCL, is a textual annotation language that turns it 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 and had been subject to formal analysis earlier. The situation complicated when with version 2.3 the OCL was aligned with the UML; this led to the extension of the three-valued logic by a second exception element, called `null`. While the first exception element `undefined` 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 higher-order logic (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. This document is intended to provide the basis for a machine-checked text "Annex A" of the UML standard targeting at tool implementors.

Further readings: This theory extends the paper "Featherweight OCL: A study for the consistent semantics of OCL 2.3 in HOL" [12] that is published as part of the proceedings of the OCL workshop 2012.

Contents

Part I.

Introduction

1. Motivation

At its origins [18, 22], OCL was conceived as a strict semantics for undefinedness, with the exception of the logical connectives of type `Boolean` that constitute a three-valued propositional logic. Recent versions of the OCL standard [20, 21] added a second exception element, which is given a non-strict semantics. 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, this results in the challenge to define a new formal semantics definition OCL that could replace the "Annex A" of the OCL standard [21].

In the paper "Extending OCL with Null-References" [6] we explored—based on mathematical arguments and paper and pencil proofs—a consistent formal semantics that comprises two exception elements: `invalid` ("bottom" in semantics terminology) and `null` (for "non-existing element").

This short paper is based on a formalization of [6], called "Featherweight OCL," in Isabelle/HOL [17]. This formalization is in its present form merely a semantical study and a proof of technology than a real tool. It focuses on the formalization of the key semantical constructions, i. e., the type `Boolean` and the logic, the type `Integer` and a standard strict operator library, and the collection type `Set(A)` with quantifiers, iterators and key operators.

2. Background

2.1. Formal Foundation

2.1.1. Isabelle

Isabelle [17] is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and natural deduction inference rules. Among other logics, Isabelle supports first-order logic, Zermelo-Fraenkel set theory and the instance for Church's higher-order logic HOL, which we choose as basis for HOL-TestGen and which is introduced in the subsequent section.

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

$$[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1} \qquad \text{or, in mathematical notation,} \qquad \frac{A_1 \qquad \cdots \qquad A_n}{A_{n+1}} \,. \tag{2.1}$$

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

$$\bigwedge x_1, \ldots, x_m.\ [\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1} \,. \tag{2.2}$$

Isabelle supports forward- and backward reasoning on rules. For backward-reasoning, a *proof-state* can be initialized and further transformed into others. For example, a proof of ϕ, using the Isar [25] language, will look as follows in Isabelle:

```
lemma label:   φ
  apply(case_tac)
  apply(simp_all)
done
```
(2.3)

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

$$
\begin{array}{rl}
\text{label}: & \phi \\
1. & \phi_1 \\
& \vdots \\
n. & \phi_n
\end{array}
\tag{2.4}
$$

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

2.1.2. Higher-order logic

Higher-order logic (HOL) [1, 13] is a classical logic based on a simple type system. It provides the usual logical connectives like $\_ \wedge \_$, $\_ \to \_$, $\neg \_$ as well as the object-logical quantifiers $\forall x.\ P\ x$ and $\exists x.\ P\ x$; in contrast to first-order logic, quantifiers may range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centered around extensional equality $\_ = \_ :: \alpha \Rightarrow \alpha \Rightarrow \text{bool}$. HOL is more expressive than first-order logic, since, e. g., induction schemes can be expressed inside the logic. Being based on some polymorphically typed λ-calculus, HOL can be viewed as a combination of a programming language like SML or Haskell and a specification language providing powerful logical quantifiers ranging over elementary and function types.

Isabelle/HOL is a logical embedding of HOL into Isabelle. The (original) simple-type system underlying HOL has been extended by Hindley-Milner style polymorphism with type-classes similar to Haskell. While Isabelle/HOL is usually seen as proof assistant, we use it as symbolic computation environment. Implementations on top of Isabelle/HOL can re-use existing powerful deduction mechanisms such as higher-order resolution, tableaux-based reasoners, rewriting procedures, Presburger arithmetic, and via various integration mechanisms, also external provers such as Vampire and the SMT-solver 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 $\tau_\perp := \perp \mid \lfloor \_ \rfloor : \alpha$ that assigns to each type τ a type $\tau_\perp$ *disjointly extended* by the exceptional element $\perp$. The function $\ulcorner \_ \urcorner : \alpha_\perp \Rightarrow \alpha$ is the inverse of $\lfloor \_ \rfloor$ (unspecified for $\perp$). Partial functions $\alpha \rightharpoonup \beta$ are defined as functions $\alpha \Rightarrow \beta_\perp$ supporting the usual concepts of domain (dom $\_$) and range (ran $\_$).

As another example of a conservative extension, typed sets were built in the Isabelle libraries conservatively on top of the kernel of HOL as functions to bool; consequently, the constant definitions for membership is as follows:[1]

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

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

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

$$
\begin{array}{lrl}
\textbf{datatype} & \text{option} & = \text{None} \mid \text{Some}\,\alpha \\
\textbf{datatype} & \alpha\,\text{list} & = \text{Nil} \mid \text{Cons}\ a\ l
\end{array}
\tag{2.6}
$$

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

$$\text{case}\ x\ \text{of}\ \ \text{None} \Rightarrow F\ \mid \text{Some}\, a \Rightarrow G\ a \tag{2.7}$$

respectively

$$\text{case}\ x\ \text{of}\ \ [] \Rightarrow F\ \mid \text{Cons}\ a\ r \Rightarrow G\ a\ r\,. \tag{2.8}$$

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

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

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

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

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

$$
\begin{array}{lll}
\textbf{fun} & \mathrm{sort} & ::(\alpha :: \mathrm{linorder})\ \mathrm{list} \Rightarrow \alpha\ \mathrm{list} \\
\textbf{where} & \mathrm{sort}\ [\,] & = [\,] \\
 & \mathrm{sort}(x\#xs) & = \mathrm{ins}\ x\ (\mathrm{sort}\ xs)
\end{array}
\tag{2.11}
$$

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

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

2.1.3. Specification Constructs in Isabelle/HOL

2.2. Featherweight OCL: Design Goals

Featherweight OCL is a formalization of the core of OCL aiming at formally investigating the relationship between the different notions of "undefinedness," i. e., `invalid` and `null`. As such, 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 [7, 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 [17].
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 [9]. The universe construction also gives semantics to type casts, dynamic type tests, as well as functions such as `oclAllInstances()`, or `oclIsNew()`.
6. Featherweight OCL types may be arbitrarily nested: `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 [11]), which is nasty but can be hidden from the user inside tools.

2.3. The Theory Organization

The semantic theory is organized in a quite conventional manner in three layers. The first layer, called the *denotational semantics* comprises a set of definitions of the operators of the language. Presented as *definitional axioms* inside Isabelle/HOL, this part assures the logically consistency of the overall construction. The second layer, called *logical layer*, is derived from the former and centered around the notion of validity of an OCL formula P for a state-transition from pre-state σ to post-state σ', validity statements were written $(\sigma, \sigma') \models P$. The third layer, called *algebraic layer*, also derived from the former layers, tries to establish a number of 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 in order 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.

2.3.1. Denotational Semantics

OCL is composed of 1) operators on built-in data structures such as Boolean, Integer or 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*) that depends on the pair (σ, σ') of pre-and post-state. The precise form of states is irrelevant for this paper (compare [6]) and will be left abstract in this presentation. We construct in Isabelle a type-class null that contains two distinguishable elements bot and null. Any type of the form $(\alpha_\bot)_\bot$ is an instance of this type-class with bot $\equiv \bot$ and null $\equiv \llcorner\bot\lrcorner$. Now, any OCL type can be represented by an HOL type of the form:

$$V(\alpha) := \mathrm{state} \times \mathrm{state} \Rightarrow \alpha :: \mathrm{null}\ .$$

On this basis, we define $V((\mathrm{bool}_\bot)_\bot)$ as the HOL type for the OCL type `Boolean` by and define:

$$I[\![\mathtt{invalid} :: V(\alpha)]\!]\tau \equiv \mathrm{bot} \qquad I[\![\mathtt{null} :: V(\alpha)]\!]\tau \equiv \mathrm{null}$$
$$I[\![\mathtt{true} :: \mathtt{Boolean}]\!]\tau = \lfloor\lfloor \mathrm{true}\rfloor\rfloor \qquad I[\![\mathtt{false}]\!]\tau = \lfloor\lfloor \mathrm{false}\rfloor\rfloor$$

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

$$I[\![X\mathtt{.oclIsInvalid()}]\!]\tau = \\ (\mathrm{if}\ I[\![X]\!]\tau = \mathrm{bot}\ \mathrm{then}\ I[\![\mathtt{true}]\!]\tau\ \mathrm{else}\ I[\![\mathtt{false}]\!]\tau)$$

where $I[\![E]\!]$ is the semantic interpretation function commonly used in mathematical textbooks and τ stands for pairs of pre- and post state (σ, σ'). Due to the used style of semantic representation (a shallow embedding) I is in fact superfluous and defined semantically as the identity; in Isabelle theories, it is usually left out in definitions to pave the way for Isabelle to checks that the underlying equations are axiomatic definitions and therefore logically safe. For reasons of conciseness, we will write $\delta\ X$ for `not` X`.oclIsUndefined()` and $\upsilon\ X$ for `not` X`.oclIsInvalid()` throughout this paper.

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

$$
\begin{array}{lll}
I[\![\texttt{not}\ X]\!]\tau & = (\text{case}\, I[\![X]\!]\tau\ \text{of} & \\
& \quad \perp \quad\quad \Rightarrow \perp & \\
& \quad |\lfloor\perp\rfloor \quad \Rightarrow \lfloor\perp\rfloor & \\
& \quad |\lfloor\lfloor x\rfloor\rfloor \quad \Rightarrow \lfloor\lfloor\neg x\rfloor\rfloor) & \\
I[\![X\ \texttt{and}\ Y]\!]\tau & = (\text{case}\, I[\![X]\!]\tau\ \text{of} & \\
& \quad \perp & \Rightarrow (\text{case}\, I[\![Y]\!]\tau\ \text{of} \\
& & \quad \perp \quad \Rightarrow \perp \\
& & \quad |\lfloor\perp\rfloor \quad \Rightarrow \perp \\
& & \quad |\lfloor\lfloor\text{true}\rfloor\rfloor \quad \Rightarrow \perp \\
& & \quad |\lfloor\lfloor\text{false}\rfloor\rfloor \quad \Rightarrow \lfloor\lfloor\text{false}\rfloor\rfloor) \\
& \quad |\lfloor\perp\rfloor & \Rightarrow (\text{case}\, I[\![Y]\!]\tau\ \text{of} \\
& & \quad \perp \quad \Rightarrow \perp \\
& & \quad |\lfloor\perp\rfloor \quad \Rightarrow \lfloor\perp\rfloor \\
& & \quad |\lfloor\lfloor\text{true}\rfloor\rfloor \quad \Rightarrow \lfloor\perp\rfloor \\
& & \quad |\lfloor\lfloor\text{false}\rfloor\rfloor \quad \Rightarrow \lfloor\lfloor\text{false}\rfloor\rfloor) \\
& \quad |\lfloor\lfloor\text{true}\rfloor\rfloor & \Rightarrow (\text{case}\, I[\![Y]\!]\tau\ \text{of} \\
& & \quad \perp \quad \Rightarrow \perp \\
& & \quad |\lfloor\perp\rfloor \quad \Rightarrow \lfloor\perp\rfloor \\
& & \quad |\lfloor\lfloor y\rfloor\rfloor \quad \Rightarrow \lfloor\lfloor y\rfloor\rfloor) \\
& \quad |\lfloor\lfloor\text{false}\rfloor\rfloor & \Rightarrow \lfloor\lfloor\text{false}\rfloor\rfloor)
\end{array}
$$

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

$$
\begin{array}{ll}
I[\![x\texttt{+}y]\!]\tau = & \text{if}\, I[\![\delta\ x]\!]\tau = \lfloor\lfloor\text{true}\rfloor\rfloor \wedge I[\![\delta\ y]\!]\tau = \lfloor\lfloor\text{true}\rfloor\rfloor \\
& \text{then}\lfloor\lfloor\lceil\lceil I[\![x]\!]\tau\rceil\rceil + \lceil\lceil I[\![y]\!]\tau\rceil\rceil\rfloor\rfloor \\
& \text{else}\perp
\end{array}
$$

where the operator "`+`" on the left-hand side of the equation denotes the OCL addition of type $[V((\text{int}_\perp)_\perp), V((\text{int}_\perp)_\perp)] \Rightarrow V((\text{int}_\perp)_\perp)$ 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.

2.3.2. Logical Layer

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

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

where σ is the pre-state and σ' the post-state of the underlying system and P is a formula. Informally, a formula P is valid if and only if its evaluation in (σ, σ') (i. e., τ

for short) yields true. Formally this means:

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

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:

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

By the latter two properties it can be inferred that any valid property P (so for example: a valid invariant) is actually 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 $\_ \doteq \_$), 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 simply `_ = _` 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 simply comparing the references to the objects. In contrast, strong logical equality is a polymorphic concept which is defined once and for all by:

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

It enjoys nearly the laws of a congruence:

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

where the predicate cp stands for *context-passing*, a property that is characterized by $P(X)$ equals $\lambda\,\tau.\ P(\lambda\,\_.\ X\tau)\tau$. 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 cp can be fully automated.

The logical layer of the Featherweight OCL rules gives also a means to convert an OCL formula living in its for-valued world into a representation that is classically two-valued and can be processed by standard SMT solvers such as **cvc3!** [**?**] or Z3 [14]. δ-closure rules for all logical connectives have the following format, e. g.:

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

Together with the general case-distinction

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

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

2.3.3. 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:

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

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

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

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

The above equations explain the behavior for the most-important non-strict operations. The clarification of the exceptional behaviors is of key-importance for a semantic definition the standard and the major deviation point from HOL-OCL [7, 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 `_ + _`):

$$\texttt{invalid + } x = \texttt{invalid} \qquad \texttt{x + invalid} = \texttt{invalid}$$
$$x\ \texttt{+ null} = \texttt{invalid} \qquad \texttt{null + } x = \texttt{invalid}$$
$$\texttt{null.asType(}X\texttt{)} = \texttt{invalid}$$

besides “classical” exceptional behavior:

$$\texttt{1 / 0} = \texttt{invalid} \qquad \texttt{1 / null} = \texttt{invalid}$$
$$\texttt{null->isEmpty()} = \texttt{true}$$

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:

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

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.[2] This semantic alternative (this is *not* Featherweight OCL at present) would yield:

$$\begin{aligned}
\texttt{invalid + }x &= \texttt{invalid} & x\texttt{ + invalid} &= \texttt{invalid}\\
x\texttt{ + null} &= x & \texttt{null + }x &= x\\
1/0 &= \texttt{invalid} & 1/\texttt{null} &= \texttt{invalid}\\
\texttt{null->isEmpty()} &= \texttt{true} & \texttt{null.asType(}X\texttt{)} &= \texttt{invalid}
\end{aligned}$$

Algebraic rules are also the key for execution and compilation of Featherweight OCL

[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`.

expressions. We derived, e. g.:

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

As `Set{1,2}` is only syntactic sugar for

```
Set{}->including(1)->including(2)
```

an expression like `Set{1,2}->includes(null)` becomes automatically 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 ”$\tau \models (\texttt{Set}\{\texttt{Set}\{2, \texttt{null}\}\} \doteq \texttt{Set}\{\texttt{Set}\{\texttt{null}, 2\}\})$”

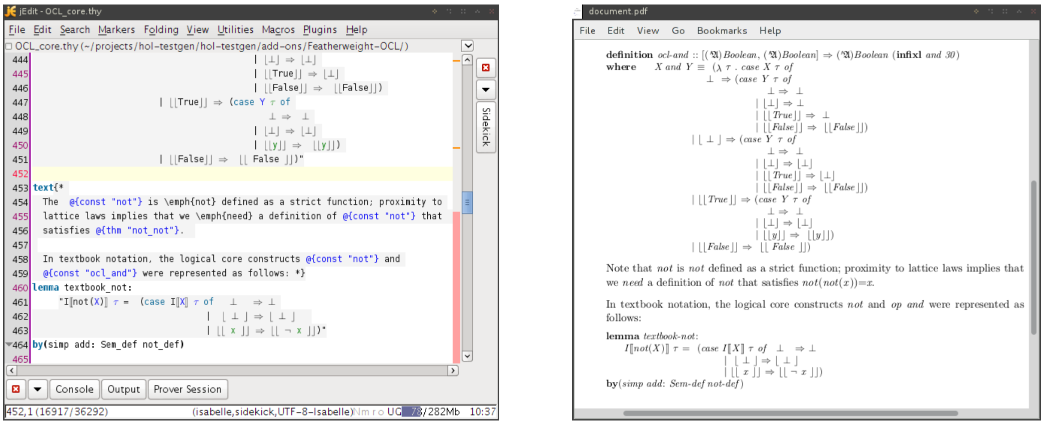
which have been machine-checked and which present a high-level and in our opinion fairly readable information for OCL tool manufactures and users.

2.4. A Machine-checked Annex A

Isabelle, as a framework for building formal tools [24], provides the means for generating *formal documents*. With formal documents we refer to documents that are machine-generated and ensure certain formal guarantees. In particular, all formal content (e. g., definitions, formulae, types) are checked for consistency during the document generation.

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

Figure 2.1 illustrates this approach: 2.1a shows the jEdit-based development environment of Isabelle with an excerpt of one of the core theories of Featherweight OCL. 2.1b



(a) The Isabelle jEdit environment.

(b) The generated formal document.

Figure 2.1.: Generating documents with guaranteed syntactical and semantical consistency.

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.

Part II.

A Formal Semantics of OCL 2.3 in Isabelle/HOL

2.5. Formal and Technical Background

2.5.1. Validity and Evaluations

The topmost goal of the formal semantics is to define the *validity statement*:

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

where σ is the pre-state and σ' the post-state of the underlying system and P is a Boolean expression (a *formula*). The assertion language of P is composed of 1) operators on built-in data structures such as Boolean or set, 2) operators of the user-defined data-model such as accessors, type-casts and tests, and 3) user-defined, side-effect-free methods. Informally, a formula P is valid if and only if its evaluation in the context (σ, σ') yields true. As all types in HOL-OCL are extended by the special element $\perp$ denoting undefinedness, we define formally:

$$(\sigma, \sigma') \vDash P \equiv \big(P(\sigma, \sigma') = \llcorner\mathrm{true}\lrcorner\big)\,.$$

Since all operators of the assertion language depend on the context (σ, σ') and result in values that can be $\perp$, all expressions can be viewed as *evaluations* from (σ, σ') to a type $\tau_\perp$. All types of expressions are of a form captured by

$$V(\alpha) := \mathrm{state} \times \mathrm{state} \Rightarrow \alpha_\perp\,,$$

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

The OCL semantics [19, Annex A] uses different interpretation functions for invariants and pre-conditions; we achieve their semantic effect by a syntactic transformation $\_{}_{\mathrm{pre}}$ which replaces all accessor functions $\_ .\,\mathrm{a}$ by their counterparts $\_ .\,\mathrm{a}\,\texttt{@pre}$. For example, $(\mathit{self}\,.\,\mathrm{a} > 5)_{\mathrm{pre}}$ is just $(\mathit{self}\,.\,\mathrm{a}\,\texttt{@pre} > 5)$.

2.5.2. Strict Operations

An operation is called strict if it returns $\perp$ if one of its arguments is $\perp$. Most OCL operations are strict, e. g., the Boolean negation is formally presented as:

$$I[\![\texttt{not}\ X]\!]\tau \equiv \begin{cases} \llcorner\neg\ulcorner I[\![X]\!]\tau\urcorner\lrcorner & \text{if } I[\![X]\!]\tau \neq \perp, \\ \perp & \text{otherwise}\,. \end{cases}$$

where $\tau = (\sigma, \sigma')$ and $I[\![\_]\!]$ is a notation marking the HOL-OCL constructs to be defined. This notation is motivated by the definitions in the OCL standard [19]. In our case, $I[\![\_]\!]$ is just the identity, i. e., $I[\![X]\!] \equiv X$. These constructs, i. e., $\texttt{not}\ \_$ are HOL functions (in this case of HOL type $V(\mathrm{bool}) \Rightarrow V(\mathrm{bool})$) that can be viewed as *transformers on evaluations*.

The binary case of the integer addition is analogous:

$$I[\![X + Y]\!]\,\tau \equiv \begin{cases} \llcorner\ulcorner I[\![X]\!]\,\tau\urcorner + \ulcorner I[\![Y]\!]\,\tau\urcorner\lrcorner & \text{if } I[\![X]\!]\,\tau \neq \perp \text{ and } I[\![Y]\!]\,\tau \neq \perp, \\ \perp & \text{otherwise}\,. \end{cases}$$

Here, the operator $\_ + \_$ on the right refers to the integer HOL operation with type $[\mathrm{int}, \mathrm{int}] \Rightarrow \mathrm{int}$. The type of the corresponding strict HOL-OCL operator $\_+\_$ is $[V(\mathrm{int}), V(\mathrm{int})] \Rightarrow V(\mathrm{int})$. A slight variation of this definition scheme is used for the operators on collection types such as HOL-OCL sets or sequences:

$$I[\![X \texttt{->union}(Y)]\!]\tau \equiv \begin{cases} S_{\llcorner}\ulcorner I[\![X]\!]\tau\urcorner \cup \ulcorner I[\![Y]\!]\tau\urcorner{}_{\lrcorner} & \text{if } I[\![X]\!]\tau \neq \bot \text{ and } I[\![Y]\!]\tau \neq \bot, \\ \bot & \text{otherwise.} \end{cases}$$

Here, S ("smash") is a function that maps a lifted set $\llcorner X \lrcorner$ to $\bot$ if and only if $\bot \in X$ and to the identity otherwise. Smashedness of collection types is the natural extension of the strictness principle for data structures.

Intuitively, the type expression $V(\tau)$ is a representation of the type that corresponds to the HOL-OCL type τ. We introduce the following type abbreviations:

$$\texttt{Boolean} := V(\mathrm{bool}), \qquad \alpha\,\texttt{Set} := V(\alpha\,\mathrm{set}),$$
$$\texttt{Integer} := V(\mathrm{int}), \text{ and} \qquad \alpha\,\texttt{Sequence} := V(\alpha\,\mathrm{list}).$$

The mapping of an expression `E` of HOL-OCL type `T` to a HOL expression E of HOL type T is injective and preserves well-typedness.

2.5.3. Boolean Operators

There is a small number of explicitly stated exceptions from the general rule that HOL-OCL operators are strict: the strong equality, the definedness operator and the logical connectives. As a prerequisite, we define the logical constants for truth, absurdity and undefinedness. We write these definitions as follows:

$$I[\![\texttt{true}]\!]\tau \equiv \llcorner\mathrm{true}\lrcorner, \qquad I[\![\texttt{false}]\!]\tau \equiv \llcorner\mathrm{false}\lrcorner, \text{ and} \qquad I[\![\texttt{invalid}]\!]\tau \equiv \bot.$$

HOL-OCL has a *strict equality* $\_ \doteq \_$. On the primitive types, it is defined similarly to the integer addition; the case for objects is discussed later. For logical purposes, we introduce also a *strong equality* $\_ \triangleq \_$ which is defined as follows:

$$I[\![X \triangleq Y]\!]\,\tau \equiv (I[\![X]\!]\,\tau = I[\![Y]\!]\,\tau),$$

where the $\_ = \_$ operator on the right denotes the logical equality of HOL. The undefinedness test is defined by $X\,\texttt{.oclIsInvalid()} \equiv (X \triangleq \texttt{invalid})$. The strong equality can be used to state reduction rules like: $\tau \vDash (\texttt{invalid} \doteq X) \triangleq \texttt{invalid}$. The OCL standard requires a Strong Kleene Logic. In particular:

$$I[\![X \texttt{ and } Y]\!]\tau \equiv \begin{cases} \llcorner\ulcorner x\urcorner \wedge \ulcorner y\urcorner\lrcorner & \text{if } x \neq \bot \text{ and } y \neq \bot, \\ \llcorner\mathrm{false}\lrcorner & \text{if } x = \llcorner\mathrm{false}\lrcorner \text{ or } y = \llcorner\mathrm{false}\lrcorner, \\ \bot & \text{otherwise}. \end{cases}$$

where $x = I[\![X]\!]\tau$ and $y = I[\![Y]\!]\tau$. The other Boolean connectives were just shortcuts: $X \texttt{ or } Y \equiv \texttt{not}\,(\texttt{not } X \texttt{ and } \texttt{not } Y)$ and $X \texttt{ implies } Y \equiv \texttt{not } X \texttt{ or } Y$.

2.5.4. Object-oriented Data Structures

Now we turn to several families of operations that the user implicitly defines when stating a class model as logical context of a specification. This is the part of the language where object-oriented features such as type casts, accessor functions, and tests for dynamic types come into play. Syntactically, a class model provides a collection of classes ($C_1, \ldots,$ C_n), an inheritance relation $\_ < \_$ on classes and a collection of attributes A_{C_i} associated to classes. Semantically, a class model means a collection of accessor functions (denoted $\_.\mathrm{a} :: C_i \to B$ and $\_.\mathrm{a}$ `@pre` $:: C_i \to B$ for $\mathrm{a} \in A_{C_i}$ and $B \in \{V(\ldots_\bot), C_1, \ldots, C_n\}$), type casts that can change the static type of an object of a class (denoted $\_{}_{[C_i]}$ of type $C_j \to C_i$) and two dynamic type tests (denoted $\mathrm{isType}_{C_i}\,\_$ and $\mathrm{isKind}_{C_i}\,\_$). A precise formal definition can be found in [11].

Class models: A simplified semantics.

In this section, we will have to clarify the notions of *object identifiers*, *object representations*, *class types* and *state*. We will give a formal model for this, that will satisfy all properties discussed in the subsequent section except one (see [9] for the complete model).

First, object identifiers are captured by an abstract type oid comprising countably many elements and a special element `nullid`. Second, 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). Third, the class type C will be the type of such an object representation: $C := (\mathrm{oid} \times \mathrm{C_t} \times A_1 \times \cdots \times A_k)$ where a unique tag-type $\mathrm{C_t}$ (ensuring type-safety) is created for each class type, where the types $A_1, \ldots, A_k$ are the attribute types (including inherited attributes) with class types substituted by oid. The function OidOf projects the first component, the oid, out of an object representation. Fourth, for a class model M with the classes $C_1, \ldots, C_n$, we define states as partial functions from oid's to object representations satisfying a *state invariant* inv_σ:

$$\mathrm{state} := \{f :: \mathrm{oid} \rightharpoonup (C_1 + \ldots + C_n) \mid \mathrm{inv}_\sigma(f)\}$$

where $\mathrm{inv}_\sigma(f)$ states two conditions: 1) there is no object representation for `nullid`: `nullid` $\notin (\mathrm{dom}\, f)$. 2) there is a"one-to-one" correspondence between object representations and oid's: $\forall oid \in \mathrm{dom}\, f.\ oid = \mathrm{OidOf}\ \ulcorner f(oid) \urcorner$. The latter condition is also mentioned in [19, Annex A] and goes back to Mark Richters [22].

2.5.5. The Accessors

On states built over object universes, we can now define accessors, casts, and type tests of an object model. We consider the case of an attribute a of class C which has the simple class type D (not a primitive type, not a collection):

$$I[\![\mathit{self}\,.\,\mathrm{a}]\!](\sigma,\sigma') \equiv \begin{cases} \bot & \text{if } O = \bot \lor \mathrm{OidOf}\ulcorner O \urcorner \notin \mathrm{dom}\ \sigma' \\ \mathrm{get_D}\ u & \text{if } \sigma'(\mathrm{get_C}\ulcorner \sigma'(\mathrm{OidOf}\ulcorner O \urcorner)\urcorner.\,\mathrm{a}^{(0)}) = \llcorner u \lrcorner, \\ \bot & \text{otherwise.} \end{cases}$$

$$I[\![\mathit{self}\,.\,\mathrm{a@pre}]\!](\sigma,\sigma') \equiv \begin{cases} \bot & \text{if } O = \bot \lor \mathrm{OidOf}\ulcorner O \urcorner \notin \mathrm{dom}\ \sigma \\ \mathrm{get_D}\ u & \text{if } \sigma(\mathrm{get_C}\ulcorner \sigma(\mathrm{OidOf}\ulcorner O \urcorner)\urcorner.\,\mathrm{a}) = \llcorner u \lrcorner, \\ \bot & \text{otherwise.} \end{cases}$$

where $O = I[\![\mathit{self}]\!](\sigma,\sigma')$. Here, $\mathrm{get_D}$ is the projection function from the object universe to $D_\bot$, and $x.\,\mathrm{a}$ is the projection of the attribute from the class type (the Cartesian product). For simple class types, we have to evaluate expression *self*, get an object representation (or undefined), project the attribute, de-reference it in the pre or post state and project the class object from the object universe ($\mathrm{get_D}$ may yield $\bot$ if the element in the universe does not correspond to a D object representation.) In the case for a primitive type attribute, the de-referentiation step is left out, and in the case of a collection over class types, the elements of the collection have to be point-wise de-referenced and smashed.

In our model accessors always yield (type-safe) object representations; not oid's. Thus, a dangling reference, i. e., one that is *not* in dom σ, results in `invalid` (this is a subtle difference to [19, Annex A] where the undefinedness is detected one de-referentiation step later). The strict equality $\_ \doteq \_$ must be defined via OidOf when applied to objects. It satisfies $(\texttt{invalid} \doteq X) \triangleq \texttt{invalid}$.

The definitions of casts and type tests can be found in [9], together with other details of the construction above and its automation in HOL-OCL.

2.6. A Proposal for an OCL 2.1 Semantics

In this section, we describe our OCL 2.1 semantics proposal as an increment to the OCL 2.0 semantics (underlying HOL-OCL and essentially formalizing [19, Annex A]). In later versions of the standard [20] the formal semantics appendix reappears although being incompatible with the normative parts of the standard. Not all rules shown here are formally proven; technically, these are informal proofs "with a glance" on the formal proofs shown in the previous section.

2.6.1. Revised Operations on Primitive Types

In UML, and since [20] in OCL, all primitive types comprise the `null`-element, modeling the possibility to be non-existent. From a functional language perspective, this corresponds to the view that each basic value is a type like `int option` as in SML. Technically, this results in lifting any primitive type twice:

$$\texttt{Integer} := V(\text{int}_\bot)\,,\ \text{etc.}$$

and basic operations have to take the null elements into account. The distinguishable undefined and null-elements were defined as follows:

$$I[\![\texttt{invalid}]\!]\tau \equiv \bot \quad\text{and}\quad I[\![\texttt{null}_{\texttt{Integer}}]\!]\tau \equiv \llcorner\bot\lrcorner\,.$$

An interpretation (consistent with [20]) is that $\texttt{null}_{\texttt{Integer}} + 3 = \texttt{invalid}$, and due to commutativity, we postulate $3+\texttt{null}_{\texttt{Integer}} = \texttt{invalid}$, too. The necessary modification of the semantic interpretation looks as follows:

$$I[\![X + Y]\!]\,\tau \equiv \begin{cases} \llcorner\llcorner\ulcorner\ulcorner x\urcorner\urcorner + \ulcorner\ulcorner y\urcorner\urcorner\lrcorner\lrcorner & \text{if } x{\neq}\bot,\ y{\neq}\bot,\ \ulcorner x\urcorner{\neq}\bot \text{ and } \ulcorner y\urcorner{\neq}\bot \\ \bot & \text{otherwise}\,. \end{cases}$$

where $x = I[\![X]\!]\,\tau$ and $y = I[\![Y]\!]\,\tau$. The resulting principle here is that operations on the primitive types Boolean, Integer, Real, and String treat null as invalid (except $\_ \doteq \_$, $\_$`.oclIsInvalid()`, $\_$`.oclIsUndefined()`, casts between the different representations of `null`, and type-tests).

This principle is motivated by our intuition that invalid represents known errors, and null-arguments of operations for Boolean, Integer, Real, and String belong to this class. Thus, we must also modify the logical operators such that $\texttt{null}_{\texttt{Boolean}}$ `and false` $\triangleq$ `false` and $\texttt{null}_{\texttt{Boolean}}$ `and true` $\triangleq \bot$.

With respect to definedness reasoning, there is a price to pay. For most basic operations we have the rule:

$$\texttt{not}\ (X + Y)\,\texttt{.oclIsInvalid()} \triangleq (\texttt{not}\ X\,\texttt{.oclIsUndefined()})\ \texttt{and}\ (\texttt{not}\ Y\,\texttt{.oclIsUndefined()})$$

where the test x`.oclIsUndefined()` covers two cases: x`.oclIsInvalid()` and $x \doteq$ `null` (i. e., x is `invalid` or `null`). As a consequence, for the inverse case $(X{+}Y)$`.oclIsInvalid()`[3] there are four possible cases for the failure instead of two in the semantics described in [19]: each expression can be an erroneous `null`, or report an error. However, since all built-in OCL operations yield non-null elements (e. g., we have the rule `not` $(X + Y \doteq \texttt{null}_{\texttt{Integer}})$), a pre-computation can drastically reduce the number of cases occurring in expressions except for the base case of variables (e. g., parameters of operations and *self* in invariants). For these cases, it is desirable that implicit pre-conditions were generated as default, ruling out the `null` case. A convenient place for this are the multiplicities, which can be set to `1` (i. e., `1..1`) and will be interpreted as being non-null (see discussion in section 2.7 for more details).

Besides, the case for collection types is analogous: in addition to the invalid collection, there is a $\texttt{null}_{\text{Set(T)}}$ collection as well as collections that contain null values (such as $\text{Set}\{\texttt{null}_{\text{T}}\}$) but never `invalid`.

[3]The same holds for $(X + Y)$`.oclIsUndefined()`.

2.6.2. Null in Class Types

It is a viable option to rule out undefinedness in object-graphs *as such*. The essential source for such undefinedness are oid's which do not occur in the state, i. e., which represent "dangling references." Ruling out undefinedness as result of object accessors would correspond to a world where an accessor is always set explicitly to `null` or to a defined object; in a programming language without explicit deletion and where constructors always initialize their arguments (e. g., Spec# [2]), this may suffice. Semantically, this can be modeled by strengthening the state invariant inv_σ by adding clauses that state that in each object representation all oid's are either `nullid` or element of the domain of the state.

We deliberately decided against this option for the following reasons:

1. *methodologically* we do not like to constrain the semantics of OCL without clear reason; in particular, "dangling references" exist in C and C++ programs and it might be necessary to write contracts for them, and
2. *semantically*, the condition "no dangling references" can only be formulated with the complete knowledge of all classes and their layout in form of object representations. This restricts the OCL semantics to a closed world model.[4]

We can model `null`-elements as object-representations with `nullid` as their oid:

1 (Representation of `null`-Elements) *Let C_i be a class type with the attributes $A_1, \ldots, A_n$. Then we define its null object representation by:*

$$I[\![\mathtt{null}_{Ci}]\!]\tau \equiv {}_\llcorner(\mathtt{nullid}, \mathrm{arb}_t, a_1, \ldots, a_n)_\lrcorner$$

where the a_i are $\perp$ for primitive types and collection types, and `nullid` *for simple class types.* arb_t *is an arbitrary underspecified constant of the tag-type.*

Due to the outermost lifting, the null object representation is a defined value, and due to its special reference `nullid` and the state invariant, it is a typed value not "living" in the state. The $\mathtt{null}_\mathrm{T}$-elements are not equal, but isomorphic: Each type, has its own unique $\mathtt{null}_\mathrm{T}$-element; they can be mapped, i. e., casted, isomorphic to each other. In HOL-OCL, we can overload constants by parametrized polymorphism which allows us to drop the index in this environment.

The referential strict equality allows us to write *self* $\doteq$ `null` in OCL. Recall that $\_ \doteq \_$ is based on the projection OidOf from object-representations.

[4]In our presentation, the definition of `state` in ?? assumes a closed world. This limitation can be easily overcome by leaving "polymorphic holes" in our object representation universe, i. e., by extending the type sum in the state definition to $C_1 + \cdots + C_n + \alpha$. The details of the management of universe extensions are involved, but implemented in HOL-OCL (see [9] for details). However, these constructions exclude knowing the set of sub-oid's in advance.

2.6.3. Revised Accessors

The modification of the accessor functions is now straight-forward:

$$I[\![obj\,.\,\mathrm{a}]\!](\sigma,\sigma') \equiv \begin{cases} \bot & \text{if } I[\![obj]\!](\sigma,\sigma') = \bot \vee \mathrm{OidOf}\ulcorner I[\![obj]\!](\sigma,\sigma')\urcorner \notin \operatorname{dom}\sigma' \\ \mathtt{null}_{\mathrm{D}} & \text{if } \mathrm{get}_{\mathrm{C}}\ulcorner\sigma'(\mathrm{OidOf}\ulcorner I[\![obj]\!](\sigma,\sigma')\urcorner)\urcorner.\,\mathrm{a}^{(0)} = \mathtt{nullid} \\ \mathrm{get}_{\mathrm{D}}\, u & \text{if } \sigma'(\mathrm{get}_{\mathrm{C}}\ulcorner\sigma'(\mathrm{OidOf}\ulcorner I[\![obj]\!](\sigma,\sigma')\urcorner)\urcorner.\,\mathrm{a}^{(0)}) = \llcorner u \lrcorner, \\ \bot & \text{otherwise.} \end{cases}$$

The definitions for type-cast and dynamic type test—which are not explicitly shown in this paper, see [9] for details—can be generalized accordingly. In the sequel, we will discuss the resulting properties of these modified accessors.

All functions of the induced signature are strict. This means that this holds for accessors, casts and tests, too:

$$\texttt{invalid}.\,\mathrm{a} \triangleq \texttt{invalid} \qquad \texttt{invalid}_{[\mathrm{C}]} \triangleq \texttt{invalid} \qquad \mathrm{isType}_{\mathrm{C}}\,\texttt{invalid} \triangleq \texttt{invalid}$$

Casts on `null` are always valid, since they have an individual dynamic type and can be casted to any other null-element due to their isomorphism.

$$\texttt{null}_{\mathrm{A}}.\,\mathrm{a} \triangleq \texttt{invalid} \qquad \texttt{null}_{\mathrm{A}[\mathrm{B}]} \triangleq \texttt{null}_{\mathrm{B}} \qquad \mathrm{isType}_{\mathrm{A}}\,\texttt{null}_{\mathrm{A}} \triangleq \texttt{true}$$

for all attributes a and classes A, B, C where $C < B < A$. These rules are further exceptions from the standard's general rule that `null` may never be passed as first ("*self*") argument.

2.6.4. Other Operations on States

Defining `_.allInstances()` is straight-forward; the only difference is the property T `.allInstances()->excludes(null)` which is a consequence of the fact that `null`'s are values and do not "live" in the state. In our semantics which admits states with "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 [15]). We define

```
(S:Set(OclAny))->modifiedOnly():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:

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

where $X' = I[\![X]\!](\sigma,\sigma')$ and $M = (\mathrm{dom}\ \sigma \cap \mathrm{dom}\ \sigma') - \{\mathrm{OidOf}\ x |\ x \in \ulcorner X' \urcorner\}$. Thus, if we require in a postcondition `Set{}->modifiedOnly()` and exclude via `_.oclIsNew()` and `_.oclIsDeleted()` the existence of new or deleted objects, the operation is a query in the sense of the OCL standard, i. e., the `isQuery` property is true. So, whenever we have $\tau \models X$`->modifiedOnly()` and $\tau \models X$`->excludes`$(s.a)$, we can infer that $\tau \models s.a = s.a$ `@pre` (if they are valid).

2.7. Attribute Values

Depending on the specified multiplicity, the evaluation of an attribute can yield a value or a collection of values. A multiplicity defines a lower bound as well as a possibly infinite upper bound on the cardinality of the attribute's values.

2.7.1. Single-Valued Attributes

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

To facilitate accessing attributes with multiplicity `0..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
```

2.7.2. Collection-Valued Attributes

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

We propose to resolve this dilemma by regarding multiplicities as optional. This point of view complies with the UML standard, that does not require lower and upper bounds to be defined for multiplicities.[5] 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 to not contain `null`. This allows for a straightforward interpretation of the multiplicity

[5] 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.

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.

2.7.3. The Precise Meaning of Multiplicity Constraints

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

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

If the upper bound n is infinite, the second invariant is omitted. For the definition of these invariants we are making use of the conversion of single values to sets described in subsection 2.7.1. If $n \leq 1$, the attribute `a` evaluates to a single value, which is then converted to a `Set` on which the `size` operation is called.

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

3. Part 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 "textbook"-like:

notation *Some* $(\lfloor(\text{-})\rfloor)$
notation *None* $(\bot)$

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

fun $\quad$ *drop* :: $'\alpha$ *option* $\Rightarrow$ $'\alpha$ $(\lceil(\text{-})\rceil)$
where *drop-lift*[*simp*]: $\lceil\lfloor v\rfloor\rceil = v$

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 $'\mathfrak{A}$, and state transitions pairs of states...

record $('\mathfrak{A})$*state* =
  heap :: *oid* $\rightharpoonup$ $'\mathfrak{A}$
  $assocs_2$ :: *oid* $\rightharpoonup$ (*oid* $\times$ *oid*) *list*

$assocs_3 :: oid \rightharpoonup (oid \times oid \times oid)\ list$

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

3.1.3. Prerequisite: An Abstract Interface for OCL Types

In order to have the possibility to nest collection types, such that we can give semantics to expressions like *Set*{*Set*{**2**},*null*}, it is necessary to introduce a uniform interface for types having the *invalid* (= bottom) element. The reason is that we impose a data-invariant on raw-collection `types_code` which assures that the *invalid* element is not allowed inside the collection; all raw-collections of this form were identified with the *invalid* element itself. The construction requires that the new collection type is uncomparable 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 (which is defined by $\lfloor \perp \rfloor$ on $'a$ *option option* to a null - element, which may have an abritrary semantic structure, and an undefinedness element $\perp$ to an abstract undefinedness element *bot* (also written $\perp$ whenever no confusion arises). As a consequence, it is necessary to redefine the notions of invalid, defined, valuation etc. on top of this interface.

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

instance *option* :: (*plus*) *plus* **by** *intro-classes*
instance *fun* :: (*type*, *plus*) *plus* **by** *intro-classes*

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

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

3.1.4. Accomodation of Basic Types to the Abstract Interface

In the following it is shown that the option-option type 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, Reals, ...).

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 -. (SOME y. y ≠ bot) in exI, auto)
                 apply(drule-tac x=x in fun-cong,auto simp:bot-fun-def)
                 apply(erule contrapos-pp, simp)
                 apply(rule some-eq-ex[THEN iffD2])
                 apply(simp add: nonEmpty)
                 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(drule-tac x=x in fun-cong)
           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 $'\mathfrak{A}$) to an arbitrary null-type (i.e. containing at least a destinguished *null* and *invalid* element.

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

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

definition *Sem* :: $'a \Rightarrow 'a$ $(I[\![\text{-}]\!])$
where $I[\![x]\!] \equiv x$

As a consequence of semantic domain definition, any OCL type will have the two semantic constants *invalid* (for exceptional, aborted computation) and *null*; the latter, however is either defined

definition *invalid* :: $('\mathfrak{A},'\alpha::bot)$ *val*
where *invalid* $\equiv \lambda\ \tau.\ bot$

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

lemma *textbook-invalid*: $I[\![invalid]\!]\tau = bot$
by(*simp add*: *invalid-def Sem-def*)

Note that the definition :

```
definition null    :: "('\<AA>,'\<alpha>::null) val"
where     "null    \<equiv> \<lambda> \<tau>. null"
```

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

lemma *textbook-null-fun*: $I[\![null::('\mathfrak{A},'\alpha::null)\ val]\!]\ \tau = (null::'\alpha::null)$
by(*simp add*: *null-fun-def Sem-def*)

3.2. Boolean Type and Logic

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

type-synonym $('\mathfrak{A})$*Boolean* $=$ $('\mathfrak{A}$,*bool option option*$)$ *val*

3.2.1. Basic Constants

lemma *bot-Boolean-def* : $(bot{::}('\mathfrak{A})Boolean) = (\lambda\ \tau.\ \bot)$
by(*simp add*: *bot-fun-def bot-option-def*)

lemma *null-Boolean-def* : $(null{::}('\mathfrak{A})Boolean) = (\lambda\ \tau.\ \lfloor\bot\rfloor)$
by(*simp add*: *null-fun-def null-option-def bot-option-def*)

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

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

lemma *bool-split*: $X\ \tau = invalid\ \tau \vee X\ \tau = null\ \tau \vee$
$X\ \tau = true\ \tau\quad \vee X\ \tau = false\ \tau$
apply(*simp add*: *invalid-def null-def true-def false-def*)
apply(*case-tac X* τ,*simp-all add*: *null-fun-def null-option-def bot-option-def*)
apply(*case-tac a*,*simp*)
apply(*case-tac aa*,*simp*)
apply *auto*
done

lemma [*simp*]: $false\ (a,\ b) = \lfloor\lfloor False\rfloor\rfloor$
by(*simp add*:*false-def*)

lemma [*simp*]: $true\ (a,\ b) = \lfloor\lfloor True\rfloor\rfloor$
by(*simp add*:*true-def*)

lemma *textbook-true*: $I[\![true]\!]\ \tau = \lfloor\lfloor True\rfloor\rfloor$
by(*simp add*: *Sem-def true-def*)

lemma *textbook-false*: $I[\![false]\!]\ \tau = \lfloor\lfloor False\rfloor\rfloor$
by(*simp add*: *Sem-def false-def*)

Summary:

3.2.2. Fundamental Predicates I: Validity and Definedness

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

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

lemma *valid1*[*simp*]: $\upsilon\ invalid = false$
by(*rule ext*,*simp add*: *valid-def bot-fun-def bot-option-def*

Name	Theorem
textbook-invalid	$I[\![invalid]\!]\ ?\tau = OCL\text{-}core.bot\text{-}class.bot$
textbook-null-fun	$I[\![null]\!]\ ?\tau = null$
textbook-true	$I[\![true]\!]\ ?\tau = \lfloor\lfloor True\rfloor\rfloor$
textbook-false	$I[\![false]\!]\ ?\tau = \lfloor\lfloor False\rfloor\rfloor$

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

invalid-def true-def false-def)

lemma *valid2*[*simp*]: υ *null* = *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*]: υ *true* = *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*]: υ *false* = *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*: $(\upsilon\ X)\ \tau = (\upsilon\ (\lambda\ \text{-}.\ X\ \tau))\ \tau$
by(*simp add*: *valid-def*)

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

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

lemma *defined1*[*simp*]: δ *invalid* = *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*]: δ *null* = *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*]: δ *true* = *true*
by(*rule ext*,*simp add*: *defined-def bot-fun-def bot-option-def null-is-valid null-option-def*

null-fun-def invalid-def true-def false-def)

lemma *defined4* [*simp*]: δ *false* = *true*
by(*rule ext*,*simp add*: *defined-def bot-fun-def bot-option-def null-is-valid null-option-def null-fun-def invalid-def true-def false-def*)

lemma *defined5* [*simp*]: δ δ X = *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*]: δ υ X = *true*
by(*rule ext*,
auto simp: *valid-def defined-def true-def false-def bot-fun-def bot-option-def null-option-def null-fun-def*)

lemma *valid5* [*simp*]: υ υ X = *true*
by(*rule ext*,
auto simp: *valid-def true-def false-def bot-fun-def bot-option-def null-option-def null-fun-def*)

lemma *valid6* [*simp*]: υ δ X = *true*
by(*rule ext*,
auto simp: *valid-def defined-def true-def false-def bot-fun-def bot-option-def null-option-def null-fun-def*)

lemma *cp-defined*:$(\delta\ X)\tau = (\delta\ (\lambda\ \text{-}.\ X\ \tau))\ \tau$
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*: $I[\![\delta(X)]\!]\ \tau = (\mathit{if}\ I[\![X]\!]\ \tau = I[\![\mathit{bot}]\!]\ \tau\ \ \vee I[\![X]\!]\ \tau = I[\![\mathit{null}]\!]\ \tau$
$\mathit{then}\ I[\![\mathit{false}]\!]\ \tau$
$\mathit{else}\ I[\![\mathit{true}]\!]\ \tau)$
by(*simp add*: *Sem-def defined-def*)

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

Summary: These definitions lead quite directly to the algebraic laws on these predicates:

Name	Theorem
textbook-defined	$I[\![\delta\ X]\!]\ \tau = (\mathit{if}\ I[\![X]\!]\ \tau = I[\![OCL\text{-}core.bot\text{-}class.bot]\!]\ \tau \vee I[\![X]\!]\ \tau = I[\![null]\!]\ \tau\ \mathit{th}$
textbook-valid	$I[\![\upsilon\ X]\!]\ \tau = (\mathit{if}\ I[\![X]\!]\ \tau = I[\![OCL\text{-}core.bot\text{-}class.bot]\!]\ \tau\ \mathit{then}\ I[\![false]\!]\ \tau\ \mathit{else}\ I[\![tru$

Table 3.2.: Basic predicate definitions of the logic.)

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

Table 3.3.: Laws of the basic predicates of the logic.)

3.2.3. Fundamental Predicates II: Logical (Strong) Equality

Note that we define strong equality extremely generic, even for types that contain an *null* or $\perp$ element:

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

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

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

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*)

lemma *StrongEq-trans-strong* [*simp*]:
  assumes *A*: $(X \triangleq Y) = true$
  and *B*: $(Y \triangleq Z) = true$
  shows $(X \triangleq Z) = true$
  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 Essential OCL expressions. A semantic — not syntactic — characterization of OCL-expressions is that they are *context-passing* or *context-invariant*, i.e. the context of an entire OCL expression, i.e. the pre-and post-state it referes to, is passed constantly and unmodified to the sub-expressions, i.e. all sub-expressions inside an OCL expression refer to the same context. Expressed formally, this boils down to:

lemma *StrongEq-subst* :
  assumes *cp*: $\bigwedge X.\ P(X)\tau = P(\lambda\ \text{-}.\ X\ \tau)\tau$
  and *eq*: $(X \triangleq Y)\tau = \mathit{true}\ \tau$
  shows $(P\ X \triangleq P\ Y)\tau = \mathit{true}\ \tau$
  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*

3.2.4. Fundamental Predicates III

And, last but not least,

lemma *defined7*[*simp*]: $\delta\ (X \triangleq Y) = \mathit{true}$
  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*]: $\upsilon\ (X \triangleq Y) = \mathit{true}$
  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*: $(X \triangleq Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau) \triangleq (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
by(*simp add*: *StrongEq-def*)

The semantics of strict equality of OCL is constructed by overloading: for each base type, there is an equality.

find-theorems (*120*) *name*: *commute*

3.2.5. 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 normalizations of OCL specifications possible as a pre-requisite for automated theorem provers. For example, it is still possible to compute without any definedness- and validity reasoning the DNF of an OCL specification; be it for test-case generations or for a smooth transition to a two-valued representation of the specification amenable to fast standard SMT-solvers, for example.

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

definition *OclNot* :: $('\mathfrak{A})$*Boolean* $\Rightarrow$ $('\mathfrak{A})$*Boolean* (*not*)
where $\quad$ $not\ X \equiv\ \lambda\ \tau\ .\ case\ X\ \tau\ of$

$$\begin{array}{ll} \quad\ \bot & \Rightarrow \bot \\ \mid \lfloor \bot \rfloor & \Rightarrow \lfloor \bot \rfloor \\ \mid \lfloor\lfloor x \rfloor\rfloor & \Rightarrow \lfloor\lfloor \neg\ x \rfloor\rfloor \end{array}$$

lemma *cp-OclNot*: $(not\ X)\tau = (not\ (\lambda\ \text{-}.\ X\ \tau))\ \tau$
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*: $\bigwedge x\ y.\ not\ x = not\ y \Longrightarrow x = y$
  by(*subst OclNot-not*[*THEN sym*], *simp*)

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

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

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

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

lemma *textbook-OclNot*:

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

by(*simp add*: *Sem-def OclNot-def*)

lemma *textbook-OclAnd*:

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

by(*simp add*: *OclAnd-def Sem-def split*: *option.split bool.split*)

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

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

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

lemma *cp-OclOr*:$((X::('\mathfrak{A})Boolean)\ or\ Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau)\ or\ (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
apply(*simp add*: *OclOr-def*)
apply(*subst cp-OclNot*[*of not* $(\lambda\text{-}.\ X\ \tau)$ *and not* $(\lambda\text{-}.\ Y\ \tau)$])
apply(*subst cp-OclAnd*[*of not* $(\lambda\text{-}.\ X\ \tau)$ *not* $(\lambda\text{-}.\ Y\ \tau)$])
by(*simp add*: *cp-OclNot*[*symmetric*] *cp-OclAnd*[*symmetric*])

lemma *cp-OclImplies*:$(X\ implies\ Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau)\ implies\ (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
apply(*simp add*: *OclImplies-def*)
apply(*subst cp-OclOr*[*of not* $(\lambda\text{-}.\ X\ \tau)$ $(\lambda\text{-}.\ Y\ \tau)$])

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 null-def invalid-def true-def false-def*)
  apply(*case-tac X x*, *simp-all*)
  apply(*case-tac a*, *simp-all*)

apply(*case-tac aa*, *simp-all*)
done

lemma *OclAnd-commute*: $(X \textit{ and } Y) = (Y \textit{ and } X)$
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*]: $(\textit{false and } X) = \textit{false}$
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*]: $(X \textit{ and false}) = \textit{false}$
by(*simp add*: *OclAnd-commute*)

lemma *OclAnd-true1*[*simp*]: $(\textit{true and } X) = X$
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*]: $(X \textit{ and true}) = X$
by(*simp add*: *OclAnd-commute*)

lemma *OclAnd-bot1*[*simp*]: $\bigwedge\tau.\ X\ \tau \neq \textit{false}\ \tau \Longrightarrow (\textit{bot and } X)\ \tau = \textit{bot}\ \tau$
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*]: $\bigwedge\tau.\ X\ \tau \neq \textit{false}\ \tau \Longrightarrow (X \textit{ and bot})\ \tau = \textit{bot}\ \tau$
by(*simp add*: *OclAnd-commute*)

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

lemma *OclAnd-assoc*: $(X \textit{ and } (Y \textit{ and } Z)) = (X \textit{ and } Y \textit{ and } Z)$
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*]: $\bigwedge\tau.\ X\ \tau \neq true\ \tau \Longrightarrow (bot\ or\ X)\ \tau = bot\ \tau$
apply(*simp add*: *OclOr-def OclAnd-def OclNot-def*)
apply(*auto simp*:*true-def false-def bot-fun-def bot-option-def*
split: *option.split option.split-asm*)
done

lemma *OclOr-bot2*[*simp*]: $\bigwedge\tau.\ X\ \tau \neq true\ \tau \Longrightarrow (X\ or\ bot)\ \tau = bot\ \tau$
by(*simp add*: *OclOr-commute*)

lemma *OclOr-null1*[*simp*]: $\bigwedge\tau.\ X\ \tau \neq true\ \tau \Longrightarrow X\ \tau \neq bot\ \tau \Longrightarrow (null\ or\ X)\ \tau = null\ \tau$
apply(*simp add*: *OclOr-def OclAnd-def OclNot-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*)
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*) *the.simps*)

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

3.3. A Standard Logical Calculus for OCL

Besides the need for algebraic laws for OCL in order to normalize

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

3.3.1. Global vs. Local Judgements

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

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

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

lemma *transform2-rev*: $\forall\ \tau.\ (\tau \models \delta\ P) \wedge (\tau \models \delta\ Q) \wedge (\tau \models P) = (\tau \models Q) \Longrightarrow P = Q$
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-asm 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 *transform3*:
assumes $H : P = true \Longrightarrow Q = true$
shows $\tau \models P \Longrightarrow \tau \models Q$
apply(*simp add*: *OclValid-def*)
apply(*rule H*[*THEN fun-cong*])
apply(*rule ext*)
oops

3.3.2. 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 null-def null-fun-def null-option-def bot-option-def*)

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

lemma *def-split-local*:
$(\tau \models \delta\ x) = ((\neg(\tau \models (x \triangleq invalid))) \wedge (\neg\ (\tau \models (x \triangleq null))))$
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*:
$\tau \models (P \; and \; Q) \Longrightarrow (\tau \models P) \wedge (\tau \models Q)$
by(*simp add*: *OclAnd-def OclValid-def true-def false-def defined-def*
split: *option.split option.split-asm bool.split bool.split-asm*)

lemma *foundation6*:
$\tau \models P \Longrightarrow \tau \models \delta \; P$
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-asm*)

lemma *foundation7*[*simp*]:
$(\tau \models not \; (\delta \; x)) = (\neg \; (\tau \models \delta \; x))$
by(*simp add*: *OclNot-def OclValid-def true-def false-def defined-def*
split: *option.split option.split-asm*)

lemma *foundation7'*[*simp*]:
$(\tau \models not \; (\upsilon \; x)) = (\neg \; (\tau \models \upsilon \; x))$
by(*simp add*: *OclNot-def OclValid-def true-def false-def valid-def*
split: *option.split option.split-asm*)

Key theorem for the δ-closure: either an expression is defined, or it can be replaced (substituted via `StrongEq_L_subst2`; see below) by invalid or null. Strictness-reduction rules will usually reduce these substituted terms drastically.

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

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

lemma *foundation10*:
$\tau \models \delta \; x \Longrightarrow \tau \models \delta \; y \Longrightarrow (\tau \models (x \; and \; y)) = (\; (\tau \models x) \wedge (\tau \models y))$
apply(*simp add*: *def-split-local*)
by(*auto simp*: *OclAnd-def OclValid-def invalid-def*
true-def null-def StrongEq-def null-fun-def null-option-def bot-option-def
split:*bool.split-asm*)

lemma *foundation11*:
$\tau \models \delta\ x \Longrightarrow\ \ \tau \models \delta\ y \Longrightarrow (\tau \models (x\ or\ y)) = (\ (\tau \models x) \vee (\tau \models y))$
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*:
$\tau \models \delta\ x \Longrightarrow\ \ \tau \models \delta\ y \Longrightarrow (\tau \models (x\ implies\ y)) = (\ (\tau \models x) \longrightarrow (\tau \models y))$
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 split*:*bool.split-asm bool.split*)

lemma *foundation13*:$(\tau \models A \triangleq true)\ \ \ \ = (\tau \models A)$
by(*auto simp*: *OclNot-def OclValid-def invalid-def true-def null-def StrongEq-def split*:*bool.split-asm bool.split*)

lemma *foundation14*:$(\tau \models A \triangleq false)\ \ \ = (\tau \models not\ A)$
by(*auto simp*: *OclNot-def OclValid-def invalid-def false-def true-def null-def StrongEq-def split*:*bool.split-asm bool.split option.split*)

lemma *foundation15*:$(\tau \models A \triangleq invalid) = (\tau \models not(\upsilon\ A))$
by(*auto simp*: *OclNot-def OclValid-def valid-def invalid-def false-def true-def null-def StrongEq-def bot-option-def null-fun-def null-option-def bot-option-def bot-fun-def split*:*bool.split-asm bool.split option.split*)

lemma *foundation16*: $\tau \models (\delta\ X) = (X\ \tau \neq bot \wedge X\ \tau \neq null)$
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*,*standard*]

lemma *foundation18*: $\tau \models (\upsilon\ X) = (X\ \tau \neq invalid\ \tau)$
by(*auto simp*: *OclValid-def valid-def false-def true-def bot-fun-def invalid-def split*:*split-if-asm*)

lemma *foundation18'*: $\tau \models (\upsilon\ X) = (X\ \tau \neq bot)$
by(*auto simp*: *OclValid-def valid-def false-def true-def bot-fun-def split*:*split-if-asm*)

lemmas *foundation19* = *foundation18*[*THEN iffD1*,*standard*]

lemma *foundation20* : $\tau \models (\delta\ X) \Longrightarrow \tau \models \upsilon\ X$
by(*simp add*: *foundation18 foundation16 invalid-def*)

lemma *foundation21*: $(not\ A \triangleq not\ B) = (A \triangleq B)$
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*: *StrongEq-def OclValid-def true-def*)

lemma *foundation23*: $(\tau \models P) = (\tau \models (\lambda\ \text{-}\ .\ P\ \tau))$
by(*auto simp*: *OclValid-def true-def*)

lemmas *cp-validity=foundation23*

lemma *defined-not-I* : $\tau \models \delta\ (x) \Longrightarrow \tau \models \delta\ (not\ x)$
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 null-fun-def bot-fun-def
split: *option.split-asm HOL.split-if-asm*)

lemma *valid-not-I* : $\tau \models \upsilon\ (x) \Longrightarrow \tau \models \upsilon\ (not\ x)$
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 null-fun-def bot-fun-def
split: *option.split-asm option.split HOL.split-if-asm*)

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

3.3.3. Local Judgements and Strong Equality

lemma *StrongEq-L-refl*: $\tau \models (x \triangleq x)$
by(*simp add*: *OclValid-def StrongEq-def*)

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

lemma *StrongEq-L-trans*: $\tau \models (x \triangleq y) \Longrightarrow \tau \models (y \triangleq z) \Longrightarrow \tau \models (x \triangleq z)$
by(*simp add*: *OclValid-def StrongEq-def true-def*)

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

lemma [*simp*, *code-unfold*]: $(false \triangleq true) = false$
by(*rule ext*, *auto simp*: *StrongEq-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 :: $(('\mathfrak{A},'\alpha)\ val \Rightarrow ('\mathfrak{A},'\beta)\ val) \Rightarrow bool$
where $cp\ P \equiv (\exists\ f.\ \forall\ X\ \tau.\ P\ X\ \tau = f\ (X\ \tau)\ \tau)$

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

lemma *StrongEq-L-subst1*: $\bigwedge \tau.\ cp\ P \Longrightarrow \tau \models (x \triangleq y) \Longrightarrow \tau \models (P\ x \triangleq P\ y)$
by(*auto simp*: *OclValid-def StrongEq-def true-def cp-def*)

lemma *StrongEq-L-subst2*:
$\bigwedge \tau.\ \ cp\ P \Longrightarrow \tau \models (x \triangleq y) \Longrightarrow \tau \models (P\ x) \Longrightarrow \tau \models (P\ y)$
by(*auto simp*: *OclValid-def StrongEq-def true-def cp-def*)

lemma *StrongEq-L-subst2-rev*: $\tau \models y \triangleq x \Longrightarrow cp\ P \Longrightarrow \tau \models P\ x \Longrightarrow \tau \models P\ y$
apply(*erule StrongEq-L-subst2*)
apply(*erule StrongEq-L-sym*)
by *assumption*

```
ML⟪ (* just a fist sketch *)
fun ocl-subst-tac subst =
        let val foundation22-THEN-iffD1 = @{thm foundation22} RS @{thm iffD1}
            val StrongEq-L-subst2-rev- = @{thm StrongEq-L-subst2-rev}
            val the-context = @{context} (* Hack of bu : will not work in general *)
        in  EVERY[rtac foundation22-THEN-iffD1 1,
                  eres-inst-tac the-context [((P,0),subst)] StrongEq-L-subst2-rev- 1,
                  simp-tac the-context 1,
                  simp-tac the-context 1]
        end

⟫
```

lemma *cpI1*:
$(\forall\ X\ \tau.\ f\ X\ \tau = f(\lambda\text{-}.\ X\ \tau)\ \tau) \Longrightarrow cp\ P \Longrightarrow cp(\lambda X.\ f\ (P\ X))$
apply(*auto simp*: *true-def cp-def*)
apply(*rule exI*, (*rule allI*)+)
by(*erule-tac x=P X* **in** *allE*, *auto*)

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

apply(*auto simp*: *true-def cp-def*)
apply(*rule exI*, (*rule allI*)+)
by(*erule-tac x=P X* **in** *allE*, *auto*)

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

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

lemma *cp-const* : $cp(\lambda\text{-}.\ c)$
by (*simp add*: *cp-def*, *fast*)

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

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

3.3.4. Laws to Establish Definedness (δ-closure)

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

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

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

3.4. Miscellaneous: OCL's if then else endif

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

lemma *cp-OclIf*:$((if\ C\ then\ B_1\ else\ B_2\ endif)\ \tau =$
$(if\ (\lambda\ \text{-}.\ C\ \tau)\ then\ (\lambda\ \text{-}.\ B_1\ \tau)\ else\ (\lambda\ \text{-}.\ B_2\ \tau)\ endif)\ \tau)$
by(*simp only*: *OclIf-def*, *subst cp-defined*, *rule refl*)

lemmas *cp-intro'*[*simp*,*intro*!] =
cp-intro
cp-OclIf[*THEN allI*[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI3*]]], *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-true'* [*simp*]: $\tau \models P \Longrightarrow (if\ P\ then\ B_1\ else\ B_2\ endif)\tau = B_1\ \tau$
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-false'* [*simp*]: $\tau \models not\ P \Longrightarrow (if\ P\ then\ B_1\ else\ B_2\ endif)\tau = B_2\ \tau$
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*

end

4. Part II: Library Definitions

theory *OCL-lib*
imports *OCL-core*
begin

4.1. Basic Types: Void, Integer, UnlimitedNatural

4.1.1. The construction of the Void Type

type-synonym $('\mathfrak{A})\,\mathit{Void} = ('\mathfrak{A},\mathit{unit\ option})\ \mathit{val}$

This *minimal* OCL type contains only two elements: *undefined* 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 $('\mathfrak{A})\mathit{Integer} = ('\mathfrak{A},\mathit{int\ option\ option})\ \mathit{val}$

Although the remaining part of this library reasons about integers abstractly, we provide here some shortcuts to some usual integers.

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

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

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

definition $\mathit{OclInt3}$ $::('\mathfrak{A})\mathit{Integer}$ $(\mathbf{3})$
where $\mathbf{3} = (\lambda\ \text{-}\ .\ \lfloor\lfloor \mathit{3}{::}\mathit{int}\rfloor\rfloor)$

definition $\mathit{OclInt4}$ $::('\mathfrak{A})\mathit{Integer}$ $(\mathbf{4})$
where $\mathbf{4} = (\lambda\ \text{-}\ .\ \lfloor\lfloor \mathit{4}{::}\mathit{int}\rfloor\rfloor)$

definition $\mathit{OclInt5}$ $::('\mathfrak{A})\mathit{Integer}$ $(\mathbf{5})$
where $\mathbf{5} = (\lambda\ \text{-}\ .\ \lfloor\lfloor \mathit{5}{::}\mathit{int}\rfloor\rfloor)$

definition $\mathit{OclInt6}$ ::($'\mathfrak{A}$)$\mathit{Integer}$ (**6**)
where **6** = (λ - . $\lfloor\lfloor \mathit{6}::\mathit{int}\rfloor\rfloor$)

definition $\mathit{OclInt7}$ ::($'\mathfrak{A}$)$\mathit{Integer}$ (**7**)
where **7** = (λ - . $\lfloor\lfloor \mathit{7}::\mathit{int}\rfloor\rfloor$)

definition $\mathit{OclInt8}$ ::($'\mathfrak{A}$)$\mathit{Integer}$ (**8**)
where **8** = (λ - . $\lfloor\lfloor \mathit{8}::\mathit{int}\rfloor\rfloor$)

definition $\mathit{OclInt9}$ ::($'\mathfrak{A}$)$\mathit{Integer}$ (**9**)
where **9** = (λ - . $\lfloor\lfloor \mathit{9}::\mathit{int}\rfloor\rfloor$)

definition $\mathit{OclInt10}$ ::($'\mathfrak{A}$)$\mathit{Integer}$ (**10**)
where **10** = (λ - . $\lfloor\lfloor \mathit{10}::\mathit{int}\rfloor\rfloor$)

4.1.3. Validity and Definedness Properties

lemma $\delta(\mathit{null}::('\mathfrak{A})\mathit{Integer}) = \mathit{false}$ **by** *simp*
lemma $\upsilon(\mathit{null}::('\mathfrak{A})\mathit{Integer}) = \mathit{true}$ **by** *simp*

lemma [*simp*,*code-unfold*]: δ (λ-. $\lfloor\lfloor n\rfloor\rfloor$) = *true*
by(*simp add*:*defined-def true-def*
bot-fun-def bot-option-def null-fun-def null-option-def)

lemma [*simp*,*code-unfold*]: υ (λ-. $\lfloor\lfloor n\rfloor\rfloor$) = *true*
by(*simp add*:*valid-def true-def*
bot-fun-def bot-option-def)

lemma [*simp*,*code-unfold*]:δ **0** = *true* **by**(*simp add*:*OclInt0-def*)
lemma [*simp*,*code-unfold*]:υ **0** = *true* **by**(*simp add*:*OclInt0-def*)
lemma [*simp*,*code-unfold*]:δ **1** = *true* **by**(*simp add*:*OclInt1-def*)
lemma [*simp*,*code-unfold*]:υ **1** = *true* **by**(*simp add*:*OclInt1-def*)
lemma [*simp*,*code-unfold*]:δ **2** = *true* **by**(*simp add*:*OclInt2-def*)
lemma [*simp*,*code-unfold*]:υ **2** = *true* **by**(*simp add*:*OclInt2-def*)
lemma [*simp*,*code-unfold*]: δ **6** = *true* **by**(*simp add*:*OclInt6-def*)
lemma [*simp*,*code-unfold*]: υ **6** = *true* **by**(*simp add*:*OclInt6-def*)
lemma [*simp*,*code-unfold*]: δ **8** = *true* **by**(*simp add*:*OclInt8-def*)
lemma [*simp*,*code-unfold*]: υ **8** = *true* **by**(*simp add*:*OclInt8-def*)
lemma [*simp*,*code-unfold*]: δ **9** = *true* **by**(*simp add*:*OclInt9-def*)
lemma [*simp*,*code-unfold*]: υ **9** = *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 standard OCL for Isabelle- technical reasons; these operators are heavily overloaded in the library that a further overloading would lead to heavy technical buzz in this document...

definition $OclAdd_{Integer}$ ::($'\mathfrak{A}$)*Integer* $\Rightarrow$ ($'\mathfrak{A}$)*Integer* $\Rightarrow$ ($'\mathfrak{A}$)*Integer* (**infix** $+_{ocl}$ *40*)
where $x +_{ocl} y \equiv \lambda\, \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$\quad then\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil + \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\quad else\ invalid\ \tau$

definition $OclLess_{Integer}$ ::($'\mathfrak{A}$)*Integer* $\Rightarrow$ ($'\mathfrak{A}$)*Integer* $\Rightarrow$ ($'\mathfrak{A}$)*Boolean* (**infix** $<_{ocl}$ *40*)
where $x <_{ocl} y \equiv \lambda\, \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$\quad then\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil < \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\quad else\ invalid\ \tau$

definition $OclLe_{Integer}$ ::($'\mathfrak{A}$)*Integer* $\Rightarrow$ ($'\mathfrak{A}$)*Integer* $\Rightarrow$ ($'\mathfrak{A}$)*Boolean* (**infix** $\le_{ocl}$ *40*)
where $x \le_{ocl} y \equiv \lambda\, \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$\quad then\ \lfloor\lfloor\lceil\lceil x\ \tau\rceil\rceil \le \lceil\lceil y\ \tau\rceil\rceil\rfloor\rfloor$
$\quad else\ invalid\ \tau$

abbreviation $OclAdd\text{-}_{Integer}$ (**infix** $+_I$ *40*) **where** $x +_I y \equiv x +_{ocl} y$
abbreviation $OclLess\text{-}_{Integer}$ (**infix** $<_I$ *40*) **where** $x <_I y \equiv x <_{ocl} y$
abbreviation $OclLe\text{-}_{Integer}$ (**infix** $\le_I$ *40*) **where** $x \le_I y \equiv x \le_{ocl} y$

Basic properties

lemma $OclAdd_{Integer}$*-commute*: $(X +_{ocl} Y) = (Y +_{ocl} X)$
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*] : $(x +_{ocl} invalid) = invalid$
by(*rule ext*, *simp add*: $OclAdd_{Integer}$*-def true-def false-def*)

lemma $OclAdd_{Integer}$*-strict2*[*simp*,*code-unfold*] : $(invalid +_{ocl} x) = invalid$
by(*rule ext*, *simp add*: $OclAdd_{Integer}$*-def true-def false-def*)

lemma $OclAdd_{Integer}$*-zero1*[*simp*,*code-unfold*] : $(x +_{ocl} \mathbf{0}) = (if\ \upsilon\ x\ and\ not\ (\delta\ x)\ then\ invalid\ else\ x\ endif)$
apply(*rule ext*, *rename-tac* τ)
proof − **fix** τ **show** $(x +_I \mathbf{0})\ \tau = (if\ \upsilon\ x\ and\ not\ (\delta\ x)\ then\ invalid\ else\ x\ endif)\ \tau$
apply(*case-tac* $(\upsilon\ x\ and\ not\ (\delta\ x))\ \tau = true\ \tau$)
apply(*subst OclIf-true′*, *simp add*: *OclValid-def*)
apply (*metis* $OclAdd_{Integer}$*-def OclNot-defargs OclValid-def foundation5 foundation9*)
apply(*subst OclIf-false′*)
apply (*metis OclValid-def defined5 defined6 defined-and-I defined-not-I foundation9*)

apply(*simp add*: $OclAdd_{Integer}$*-def OclInt0-def*)

apply(*rule conjI*)
apply(*case-tac x* τ)
apply (*metis OCL-core.bot-fun-def OCL-core.drop.simps bot-option-def defined-def false-def true-def*)
apply(*simp*)
apply(*case-tac a*)
apply(*simp*)
apply (*metis OclValid-def bot-option-def foundation17 null-option-def*)
apply(*simp*)
by (*metis OclValid-def foundation10 foundation18′ foundation6 foundation7 invalid-def*)
qed

lemma $OclAdd_{Integer}$-*zero2*[*simp*,*code-unfold*] : ($\mathbf{0} +_{ocl} x$) = (*if* υ *x and not* (δ *x*) *then invalid else x endif*)
by(*subst* $OclAdd_{Integer}$-*commute*, *simp*)

Context Passing

lemma *cp*-$OclAdd_{Integer}$:$(X +_{ocl} Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau) +_{ocl} (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
by(*simp add*: $OclAdd_{Integer}$-*def cp-defined*[*symmetric*])

lemma *cp*-$OclLess_{Integer}$:$(X <_{ocl} Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau) <_{ocl} (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
by(*simp add*: $OclLess_{Integer}$-*def cp-defined*[*symmetric*])

lemma *cp*-$OclLe_{Integer}$:$(X \leq_{ocl} Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau) \leq_{ocl} (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
by(*simp add*: $OclLe_{Integer}$-*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 $\tau_0 \models (\ \mathbf{9} \leq_{ocl} \mathbf{10}\ )$
value $\tau_0 \models ((\ \mathbf{4} +_{ocl} \mathbf{4}\ ) \leq_{ocl} \mathbf{10}\ )$
value $\neg(\tau_0 \models ((\ \mathbf{4} +_{ocl} (\ \mathbf{4} +_{ocl} \mathbf{4}\ ))\ <_{ocl} \mathbf{10}\ ))$
value $\tau_0 \models$ *not* (υ (*null* $+_{ocl}$ **1**))

4.1.5. The construction of the UnlimitedNatural Type

Unlike *Integer*, we should also include the infinity value besides *undefined* and *null*.

class *infinity* = *null* +
fixes *infinity* :: ′*a*
assumes *infinity-is-valid* : *infinity* ≠ *bot*
assumes *infinity-is-defined* : *infinity* ≠ *null*

instantiation *option* :: (*null*)*infinity*
begin
definition *infinity-option-def*: (*infinity*::′*a*::*null option*) ≡ ⌊ *null* ⌋
instance proof show (*infinity*::′*a*::*null option*) ≠ *null*

by(*simp add*:*infinity-option-def null-is-valid null-option-def bot-option-def*)
show (*infinity*::$'a$::*null option*) $\neq$ *bot*
by(*simp add*:*infinity-option-def null-option-def bot-option-def*)
qed
end

instantiation *fun* :: (*type*,*infinity*) *infinity*
begin
definition *infinity-fun-def*: (*infinity*::$'a \Rightarrow 'b$::*infinity*) $\equiv$ (λ *x*. *infinity*)

instance proof
show (*infinity*::$'a \Rightarrow 'b$::*infinity*) $\neq$ *bot*
apply(*auto simp*: *infinity-fun-def bot-fun-def*)
apply(*drule-tac x*=*x* **in** *fun-cong*)
apply(*erule contrapos-pp*, *simp add*: *infinity-is-valid*)
done
show (*infinity*::$'a \Rightarrow 'b$::*infinity*) $\neq$ *null*
apply(*auto simp*: *infinity-fun-def null-fun-def*)
apply(*drule-tac x*=*x* **in** *fun-cong*)
apply(*erule contrapos-pp*, *simp add*: *infinity-is-defined*)
done
qed
end

type-synonym $('\mathfrak{A},'\alpha)$ *val'* = $'\mathfrak{A}$ *st* $\Rightarrow$ $'\alpha$::*infinity*

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

lemma [*simp*]: υ *infinity* = *true*
by(*rule ext*, *simp add*: *bot-fun-def infinity-fun-def infinity-is-valid valid-def*)

lemma [*simp*]: δ *infinity* = *true*
by(*rule ext*, *simp add*: *bot-fun-def defined-def infinity-fun-def infinity-is-defined infinity-is-valid null-fun-def*)

lemma [*simp*]: μ *invalid* = *false*
by(*rule ext*, *simp add*: *bot-fun-def invalid-def limitedNatural-def*)

lemma [*simp*]: μ *null* = *false*
by(*rule ext*, *simp add*: *limitedNatural-def*)

lemma [*simp*]: μ *infinity* = *false*
by(*rule ext*, *simp add*: *limitedNatural-def*)

type-synonym $('\mathfrak{A})$*UnlimitedNatural* = $('\mathfrak{A}$, *nat option option option*) *val'*
locale *OclUnlimitedNatural*

definition $OclNat0$ ::($'\mathfrak{A}$) $UnlimitedNatural$
where $OclNat0(*\mathbf{0}*) = (\lambda\ \text{-}\ .\ \lfloor\lfloor\lfloor 0::nat\rfloor\rfloor\rfloor)$

definition $OclNat1$ ::($'\mathfrak{A}$) $UnlimitedNatural$
where $OclNat1(*\mathbf{1}*) = (\lambda\ \text{-}\ .\ \lfloor\lfloor\lfloor 1::nat\rfloor\rfloor\rfloor)$

definition $OclNat2$ ::($'\mathfrak{A}$) $UnlimitedNatural$
where $OclNat2(*\mathbf{2}*) = (\lambda\ \text{-}\ .\ \lfloor\lfloor\lfloor 2::nat\rfloor\rfloor\rfloor)$

definition $OclNat3$ ::($'\mathfrak{A}$) $UnlimitedNatural$
where $OclNat3(*\mathbf{3}*) = (\lambda\ \text{-}\ .\ \lfloor\lfloor\lfloor 3::nat\rfloor\rfloor\rfloor)$

definition $OclNat4$ ::($'\mathfrak{A}$) $UnlimitedNatural$
where $OclNat4(*\mathbf{4}*) = (\lambda\ \text{-}\ .\ \lfloor\lfloor\lfloor 4::nat\rfloor\rfloor\rfloor)$

definition $OclNat5$ ::($'\mathfrak{A}$) $UnlimitedNatural$
where $OclNat5(*\mathbf{5}*) = (\lambda\ \text{-}\ .\ \lfloor\lfloor\lfloor 5::nat\rfloor\rfloor\rfloor)$

definition $OclNat6$ ::($'\mathfrak{A}$) $UnlimitedNatural$
where $OclNat6(*\mathbf{6}*) = (\lambda\ \text{-}\ .\ \lfloor\lfloor\lfloor 6::nat\rfloor\rfloor\rfloor)$

definition $OclNat7$ ::($'\mathfrak{A}$) $UnlimitedNatural$
where $OclNat7(*\mathbf{7}*) = (\lambda\ \text{-}\ .\ \lfloor\lfloor\lfloor 7::nat\rfloor\rfloor\rfloor)$

definition $OclNat8$ ::($'\mathfrak{A}$) $UnlimitedNatural$
where $OclNat8(*\mathbf{8}*) = (\lambda\ \text{-}\ .\ \lfloor\lfloor\lfloor 8::nat\rfloor\rfloor\rfloor)$

definition $OclNat9$ ::($'\mathfrak{A}$) $UnlimitedNatural$
where $OclNat9(*\mathbf{9}*) = (\lambda\ \text{-}\ .\ \lfloor\lfloor\lfloor 9::nat\rfloor\rfloor\rfloor)$

definition $OclNat10$ ::($'\mathfrak{A}$) $UnlimitedNatural$
where $OclNat10(*\mathbf{10}*) = (\lambda\ \text{-}\ .\ \lfloor\lfloor\lfloor 10::nat\rfloor\rfloor\rfloor)$

context *OclUnlimitedNatural*
begin

abbreviation *OclNat-0* (**0**) **where** **0** $\equiv$ *OclNat0*
abbreviation *OclNat-1* (**1**) **where** **1** $\equiv$ *OclNat1*
abbreviation *OclNat-2* (**2**) **where** **2** $\equiv$ *OclNat2*
abbreviation *OclNat-3* (**3**) **where** **3** $\equiv$ *OclNat3*
abbreviation *OclNat-4* (**4**) **where** **4** $\equiv$ *OclNat4*
abbreviation *OclNat-5* (**5**) **where** **5** $\equiv$ *OclNat5*
abbreviation *OclNat-6* (**6**) **where** **6** $\equiv$ *OclNat6*
abbreviation *OclNat-7* (**7**) **where** **7** $\equiv$ *OclNat7*
abbreviation *OclNat-8* (**8**) **where** **8** $\equiv$ *OclNat8*
abbreviation *OclNat-9* (**9**) **where** **9** $\equiv$ *OclNat9*
abbreviation *OclNat-10* (**10**) **where** **10** $\equiv$ *OclNat10*

end

definition *OclNat-infinity* :: $('\mathfrak{A})$ *UnlimitedNatural* (∞)
where $\infty = (\lambda\text{-}.\ \lfloor\lfloor None\rfloor\rfloor)$

4.1.6. Validity and Definedness Properties

lemma $\delta(null::('\mathfrak{A})\,UnlimitedNatural) = false$ **by** *simp*
lemma $\upsilon(null::('\mathfrak{A})\,UnlimitedNatural) = true$ **by** *simp*

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

lemma [*simp*,*code-unfold*]: $\upsilon\ (\lambda\text{-}.\ \lfloor\lfloor\lfloor n\rfloor\rfloor\rfloor) = true$
by(*simp*)

lemma [*simp*,*code-unfold*]: $\mu\ (\lambda\text{-}.\ \lfloor\lfloor\lfloor n\rfloor\rfloor\rfloor) = true$
by(*simp add*: *limitedNatural-def true-def*
bot-fun-def bot-option-def null-fun-def null-option-def infinity-fun-def infinity-option-def)

4.1.7. Arithmetical Operations on UnlimitedNatural

Definition

definition $OclAdd_{UnlimitedNatural}$ ::$('\mathfrak{A})$ *UnlimitedNatural* $\Rightarrow$ $('\mathfrak{A})$ *UnlimitedNatural* $\Rightarrow$ $('\mathfrak{A})$ *UnlimitedNatural*
(**infix** $+_{ocl}$ *40*)
where $x +_{ocl} y \equiv \lambda\ \tau.\ if\ (\mu\ x)\ \tau = true\ \tau \wedge (\mu\ y)\ \tau = true\ \tau$
$then\ \lfloor\lfloor\lfloor\lceil\lceil\lceil x\ \tau\rceil\rceil\rceil + \lceil\lceil\lceil y\ \tau\rceil\rceil\rceil\rfloor\rfloor\rfloor$
$else\ invalid\ \tau$

definition $OclLess_{UnlimitedNatural}$ ::$('\mathfrak{A})$ *UnlimitedNatural* $\Rightarrow$ $('\mathfrak{A})$ *UnlimitedNatural* $\Rightarrow$ $('\mathfrak{A})$*Boolean*
(**infix** $<_{ocl}$ *40*)
where $x <_{ocl} y \equiv \lambda\ \tau.\ if\ (\mu\ x)\ \tau = true\ \tau \wedge (\mu\ y)\ \tau = true\ \tau$
$then\ \lfloor\lfloor\lceil\lceil\lceil x\ \tau\rceil\rceil\rceil < \lceil\lceil\lceil y\ \tau\rceil\rceil\rceil\rfloor\rfloor$
$else\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$then\ (\mu\ x)\ \tau$
$else\ invalid\ \tau$

definition $OclLe_{UnlimitedNatural}$ ::$('\mathfrak{A})$ *UnlimitedNatural* $\Rightarrow$ $('\mathfrak{A})$ *UnlimitedNatural* $\Rightarrow$ $('\mathfrak{A})$*Boolean*
(**infix** $\leq_{ocl}$ *40*)
where $x \leq_{ocl} y \equiv \lambda\ \tau.\ if\ (\mu\ x)\ \tau = true\ \tau \wedge (\mu\ y)\ \tau = true\ \tau$
$then\ \lfloor\lfloor\lceil\lceil\lceil x\ \tau\rceil\rceil\rceil \leq \lceil\lceil\lceil y\ \tau\rceil\rceil\rceil\rfloor\rfloor$
$else\ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\delta\ y)\ \tau = true\ \tau$
$then\ not\ (\mu\ y)\ \tau$
$else\ invalid\ \tau$

abbreviation $OclAdd\text{-}_{UnlimitedNatural}$ (**infix** $+_{UN}$ *40*) **where** $x +_{UN} y \equiv OclAdd_{UnlimitedNatural}$
$x\ y$
abbreviation $OclLess\text{-}_{UnlimitedNatural}$ (**infix** $<_{UN}$ *40*) **where** $x <_{UN} y \equiv OclLess_{UnlimitedNatural}$
$x\ y$

abbreviation $OclLe\text{-}_{UnlimitedNatural}$ (**infix** $\leq_{UN}$ *40*) **where** $x \leq_{UN} y \equiv OclLe_{UnlimitedNatural}$ $x\ y$

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*.

context *OclUnlimitedNatural*
begin
value $\tau_0 \models (\ \mathbf{9} \leq_{UN} \mathbf{10}\ )$
value $\tau_0 \models ((\ \mathbf{4} +_{UN} \mathbf{4}\ ) \leq_{UN} \mathbf{10}\ )$
value $\neg(\tau_0 \models ((\ \mathbf{4} +_{UN} (\ \mathbf{4} +_{UN} \mathbf{4}\ )) <_{UN} \mathbf{10}\ ))$
value $\tau_0 \models (\mathbf{0} \leq_{ocl} \infty)$
value $\tau_0 \models not\ (\upsilon\ (null +_{UN} \mathbf{1}))$
value $\tau_0 \models not\ (\upsilon\ (\infty +_{ocl} \mathbf{0}))$
value $\tau_0 \models \quad \mu\ \mathbf{1}$
end
value $\tau_0 \models not\ (\upsilon\ (null +_{ocl} \infty))$
value $\tau_0 \models not\ (\infty <_{ocl} \infty)$
value $\tau_0 \models not\ (\upsilon\ (invalid \leq_{ocl} \infty))$
value $\tau_0 \models not\ (\upsilon\ (null \leq_{ocl} \infty))$
value $\tau_0 \models \quad \upsilon\ \infty$
value $\tau_0 \models \quad \delta\ \infty$
value $\tau_0 \models not\ (\mu\ \infty)$

4.2. Fundamental Predicates on Boolean and Integer: Strict Equality

4.2.1. Definition

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

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

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

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

defs $StrictRefEq_{Integer}$[*code-unfold*] :

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

lemma [*simp*, *code-unfold*] : $(\mathit{true} \doteq \mathit{false}) = \mathit{false}$
by(*simp add*:$\mathit{StrictRefEq}_{\mathit{Boolean}}$)
lemma [*simp*, *code-unfold*] : $(\mathit{false} \doteq \mathit{true}) = \mathit{false}$
by(*simp add*:$\mathit{StrictRefEq}_{\mathit{Boolean}}$)

value $\tau \models \mathbf{1} <> \mathbf{2}$
value $\tau \models \mathbf{2} <> \mathbf{1}$
value $\tau \models \mathbf{2} \doteq \mathbf{2}$
value $\tau \models \mathit{true} <> \mathit{false}$
value $\tau \models \mathit{false} <> \mathit{true}$

4.2.2. Logic and Algebraic Layer on Basic Types

Validity and Definedness Properties (I)

lemma $\mathit{StrictRefEq}_{\mathit{Boolean}}$*-defined-args-valid*:
$(\tau \models \delta((x{::}('\mathfrak{A})\mathit{Boolean}) \doteq y)) = ((\tau \models(\upsilon\ x)) \wedge (\tau \models(\upsilon\ y)))$
by(*auto simp*: $\mathit{StrictRefEq}_{\mathit{Boolean}}$ *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 $\mathit{StrictRefEq}_{\mathit{Integer}}$*-defined-args-valid*:
$(\tau \models \delta((x{::}('\mathfrak{A})\mathit{Integer}) \doteq y)) = ((\tau \models(\upsilon\ x)) \wedge (\tau \models(\upsilon\ y)))$
by(*auto simp*: $\mathit{StrictRefEq}_{\mathit{Integer}}$ *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 $\mathit{StrictRefEq}_{\mathit{Boolean}}$*-defargs*:
$\tau \models ((x{::}('\mathfrak{A})\mathit{Boolean}) \doteq y) \Longrightarrow (\tau \models (\upsilon\ x)) \wedge (\tau \models(\upsilon\ y))$
by(*simp add*: $\mathit{StrictRefEq}_{\mathit{Boolean}}$ *OclValid-def true-def invalid-def bot-option-def*
split: *bool.split-asm HOL.split-if-asm*)

lemma $\mathit{StrictRefEq}_{\mathit{Integer}}$*-defargs*:
$\tau \models ((x{::}('\mathfrak{A})\mathit{Integer}) \doteq y) \Longrightarrow (\tau \models (\upsilon\ x)) \wedge (\tau \models (\upsilon\ y))$
by(*simp add*: $\mathit{StrictRefEq}_{\mathit{Integer}}$ *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 $\mathit{StrictRefEq}_{\mathit{Boolean}}$*-strict''* : $\delta\ ((x{::}('\mathfrak{A})\mathit{Boolean}) \doteq y) = (\upsilon(x)\ \mathit{and}\ \upsilon(y))$
by(*auto intro*!: *transform2-rev defined-and-I simp*:*foundation10* $\mathit{StrictRefEq}_{\mathit{Boolean}}$*-defined-args-valid*)

lemma $\mathit{StrictRefEq}_{\mathit{Integer}}$*-strict''* : $\delta\ ((x{::}('\mathfrak{A})\mathit{Integer}) \doteq y) = (\upsilon(x)\ \mathit{and}\ \upsilon(y))$

by(*auto intro*!: *transform2-rev defined-and-I simp*:*foundation10* $StrictRefEq_{Integer}$-*defined-args-valid*)

lemma $StrictRefEq_{Integer}$-*strict* :
  assumes A: υ $(x::(\prime\mathfrak{A})Integer) = true$
  and B: υ $y = true$
  shows υ $(x \doteq y) = true$
  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: υ $(((x::(\prime\mathfrak{A})Integer)) \doteq y) = true$
  shows υ $x = true \wedge \upsilon$ $y = true$
  apply(*insert A*, *rule conjI*)
  apply(*rule ext*, *drule-tac x=xa* **in** *fun-cong*)
  prefer *2*
  apply(*rule ext*, *drule-tac x=xa* **in** *fun-cong*)
  apply(*simp-all add*: *StrongEq-def* $StrictRefEq_{Integer}$ *false-def true-def valid-def defined-def*)
  apply(*case-tac y xa*, *auto*)
  apply(*simp-all add*: *true-def invalid-def bot-fun-def*)
  done

Reflexivity

lemma $StrictRefEq_{Boolean}$-*refl*[*simp*,*code-unfold*] :
$((x::(\prime\mathfrak{A})Boolean) \doteq x) = (if\ (\upsilon\ x)\ then\ true\ else\ invalid\ endif)$
by(*rule ext*, *simp add*: $StrictRefEq_{Boolean}$ *OclIf-def*)

lemma $StrictRefEq_{Integer}$-*refl*[*simp*,*code-unfold*] :
$((x::(\prime\mathfrak{A})Integer) \doteq x) = (if\ (\upsilon\ x)\ then\ true\ else\ invalid\ endif)$
by(*rule ext*, *simp add*: $StrictRefEq_{Integer}$ *OclIf-def*)

Execution with invalid or null as argument

lemma $StrictRefEq_{Boolean}$-*strict1*[*simp*,*code-unfold*] : $((x::(\prime\mathfrak{A})Boolean) \doteq invalid) = invalid$
by(*rule ext*, *simp add*: $StrictRefEq_{Boolean}$ *true-def false-def*)

lemma $StrictRefEq_{Boolean}$-*strict2*[*simp*,*code-unfold*] : $(invalid \doteq (x::(\prime\mathfrak{A})Boolean)) = invalid$
by(*rule ext*, *simp add*: $StrictRefEq_{Boolean}$ *true-def false-def*)

lemma $StrictRefEq_{Integer}$-*strict1*[*simp*,*code-unfold*] : $((x::(\prime\mathfrak{A})Integer) \doteq invalid) = invalid$
by(*rule ext*, *simp add*: $StrictRefEq_{Integer}$ *true-def false-def*)

lemma $StrictRefEq_{Integer}$-*strict2*[*simp*,*code-unfold*] : $(invalid \doteq (x::(\prime\mathfrak{A})Integer)) = invalid$
by(*rule ext*, *simp add*: $StrictRefEq_{Integer}$ *true-def false-def*)

lemma *null-non-true* [*simp*,*code-unfold*]:$(null \doteq true) = false$
apply(*rule ext*, *simp add*: $StrictRefEq_{Boolean}$ *StrongEq-def false-def*)
by (*metis defined3 foundation1 foundation16 null-fun-def*)

lemma *integer-non-null* [*simp*]: $((\lambda\text{-}. \lfloor\lfloor n\rfloor\rfloor) \doteq (null::('\mathfrak{A})Integer)) = false$
by(*rule ext*,*auto simp*: $StrictRefEq_{Integer}$ *valid-def*
bot-fun-def bot-option-def null-fun-def null-option-def StrongEq-def)

lemma *null-non-integer* [*simp*]: $((null::('\mathfrak{A})Integer) \doteq (\lambda\text{-}. \lfloor\lfloor n\rfloor\rfloor)) = false$
by(*rule ext*,*auto simp*: $StrictRefEq_{Integer}$ *valid-def*
bot-fun-def bot-option-def null-fun-def null-option-def StrongEq-def)

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

Behavior vs StrongEq

lemma $StrictRefEq_{Boolean}$*-vs-StrongEq*:
$\tau \models(\upsilon\ x) \Longrightarrow \tau \models(\upsilon\ y) \Longrightarrow (\tau \models (((x::('\mathfrak{A})Boolean) \doteq y) \triangleq (x \triangleq y)))$
apply(*simp add*: $StrictRefEq_{Boolean}$ *OclValid-def*)
apply(*subst cp-StrongEq*)**back**
by *simp*

lemma $StrictRefEq_{Integer}$*-vs-StrongEq*:
$\tau \models(\upsilon\ x) \Longrightarrow \tau \models(\upsilon\ y) \Longrightarrow (\tau \models (((x::('\mathfrak{A})Integer) \doteq y) \triangleq (x \triangleq y)))$
apply(*simp add*: $StrictRefEq_{Integer}$ *OclValid-def*)
apply(*subst cp-StrongEq*)**back**
by *simp*

Context Passing

lemma *cp-*$StrictRefEq_{Boolean}$:
$((X::('\mathfrak{A})Boolean) \doteq Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau) \doteq (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
by(*auto simp*: $StrictRefEq_{Boolean}$ *StrongEq-def defined-def valid-def cp-defined*[*symmetric*])

lemma *cp-*$StrictRefEq_{Integer}$:
$((X::('\mathfrak{A})Integer) \doteq Y)\ \tau = ((\lambda\ \text{-}.\ X\ \tau) \doteq (\lambda\ \text{-}.\ Y\ \tau))\ \tau$
by(*auto simp*: $StrictRefEq_{Integer}$ *StrongEq-def valid-def cp-defined*[*symmetric*])

lemmas *cp-intro'*[*simp*,*intro*!] =
cp-intro'
cp-StrictRefEq$_{Boolean}$[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of StrictRefEq*]]
cp-StrictRefEq$_{Integer}$[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of StrictRefEq*]]
cp-OclAdd$_{Integer}$[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of OclAdd*$_{Integer}$]]
cp-OclLess$_{Integer}$[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of OclLess*$_{Integer}$]]
cp-OclLe$_{Integer}$[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]], *of OclLe*$_{Integer}$]]

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 $\tau_0 \models \upsilon(\mathit{true})$
value $\tau_0 \models \delta(\mathit{false})$
value $\neg(\tau_0 \models \delta(\mathit{null}))$
value $\neg(\tau_0 \models \delta(\mathit{invalid}))$
value $\tau_0 \models \upsilon((\mathit{null}::(\prime\mathfrak{A})\mathit{Boolean}))$
value $\neg(\tau_0 \models \upsilon(\mathit{invalid}))$
value $\tau_0 \models (\mathit{true\ and\ true})$
value $\tau_0 \models (\mathit{true\ and\ true} \triangleq \mathit{true})$
value $\tau_0 \models ((\mathit{null\ or\ null}) \triangleq \mathit{null})$
value $\tau_0 \models ((\mathit{null\ or\ null}) \doteq \mathit{null})$
value $\tau_0 \models ((\mathit{true} \triangleq \mathit{false}) \triangleq \mathit{false})$
value $\tau_0 \models ((\mathit{invalid} \triangleq \mathit{false}) \triangleq \mathit{false})$
value $\tau_0 \models ((\mathit{invalid} \doteq \mathit{false}) \triangleq \mathit{invalid})$

Elementary computations on Integer

value $\tau_0 \models \upsilon(\mathbf{4})$
value $\tau_0 \models \delta(\mathbf{4})$
value $\tau_0 \models \upsilon((\mathit{null}::(\prime\mathfrak{A})\mathit{Integer}))$
value $\tau_0 \models (\mathit{invalid} \triangleq \mathit{invalid}\ )$
value $\tau_0 \models (\mathit{null} \triangleq \mathit{null}\ )$
value $\tau_0 \models (\mathbf{4} \triangleq \mathbf{4})$
value $\neg(\tau_0 \models (\mathbf{9} \triangleq \mathbf{10}\ ))$
value $\neg(\tau_0 \models (\mathit{invalid} \triangleq \mathbf{10}\ ))$
value $\neg(\tau_0 \models (\mathit{null} \triangleq \mathbf{10}\ ))$
value $\neg(\tau_0 \models (\mathit{invalid} \doteq (\mathit{invalid}::(\prime\mathfrak{A})\mathit{Integer})))$
value $\neg(\tau_0 \models \upsilon(\mathit{invalid} \doteq (\mathit{invalid}::(\prime\mathfrak{A})\mathit{Integer})))$
value $\neg(\tau_0 \models (\mathit{invalid} <> (\mathit{invalid}::(\prime\mathfrak{A})\mathit{Integer})))$
value $\neg(\tau_0 \models \upsilon(\mathit{invalid} <> (\mathit{invalid}::(\prime\mathfrak{A})\mathit{Integer})))$
value $\tau_0 \models (\mathit{null} \doteq (\mathit{null}::(\prime\mathfrak{A})\mathit{Integer})\ )$
value $\tau_0 \models (\mathit{null} \doteq (\mathit{null}::(\prime\mathfrak{A})\mathit{Integer})\ )$
value $\tau_0 \models (\mathbf{4} \doteq \mathbf{4})$
value $\neg(\tau_0 \models (\mathbf{4} <> \mathbf{4}))$
value $\neg(\tau_0 \models (\mathbf{4} \doteq \mathbf{10}\ ))$

value $(\tau_0 \models (\mathbf{4} <> \mathbf{10}\ ))$

4.3. Complex Types: The Set-Collection Type (I) Core

4.3.1. The construction of the Set Type

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

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

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

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*.
$\qquad X = bot \vee X = null \vee (\forall x\in\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
by (*rule-tac* x=*bot* **in** *exI*, *simp*)

instantiation *Set-0* :: (*null*)*bot*
begin

definition *bot-Set-0-def*: $(bot::('a::null)\ Set\text{-}0) \equiv Abs\text{-}Set\text{-}0\ None$

instance proof show $\exists x::'a\ Set\text{-}0.\ x \neq bot$
apply(*rule-tac* x=*Abs-Set-0* $\lfloor None\rfloor$ **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*: $(null::('a::null)\ Set\text{-}0) \equiv Abs\text{-}Set\text{-}0\ \lfloor\ None\ \rfloor$

instance proof show $(null::('a::null)\ Set\text{-}0) \neq bot$
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 $('\mathfrak{A},'\alpha)\ Set\ = ('\mathfrak{A},\ '\alpha\ Set\text{-}0)\ val$

4.3.2. Validity and Definedness Properties

Every element in a defined set is valid.

lemma *Set-inv-lemma*: $\tau \models (\delta\ X) \Longrightarrow \forall x{\in}\lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil.\ x \neq bot$
apply(*insert OCL-lib.Set-0.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* $(X\ \tau) = bot$])
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* $(X\ \tau) = null$])
apply(*subst Abs-Set-0-inject*[*symmetric*], *rule Rep-Set-0*, *simp*)
apply(*simp add*: *Rep-Set-0-inverse null-option-def*)
by (*metis bot-option-def null-Set-0-def null-option-def*)

lemma *Set-inv-lemma'* :
assumes *x-def* : $\tau \models \delta\ X$
and *e-mem* : $e \in \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil$
shows $\tau \models \upsilon\ (\lambda\text{-}.\ e)$
apply(*rule Set-inv-lemma*[*OF x-def*, *THEN ballE*[**where** $x = e$]])
apply (*metis foundation18'*)
by (*metis e-mem*)

lemma *abs-rep-simp'* :
assumes *S-all-def* : $\tau \models \delta\ S$
shows $Abs\text{-}Set\text{-}0\ \lfloor\lfloor\lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil\rfloor\rfloor = S\ \tau$
proof −
have *discr-eq-false-true* : $\bigwedge\tau.\ (false\ \tau = true\ \tau) = False$ **by** (*metis OclValid-def foundation2*)
show *?thesis*
apply(*insert S-all-def*, *simp add*: *OclValid-def defined-def*)
apply(*rule mp*[*OF Abs-Set-0-induct*[**where** $P = \lambda S.\ (if\ S = \bot\ \tau \vee S = null\ \tau\ then\ false\ \tau\ else\ true\ \tau) = true\ \tau \longrightarrow Abs\text{-}Set\text{-}0\ \lfloor\lfloor\lceil\lceil Rep\text{-}Set\text{-}0\ S\rceil\rceil\rfloor\rfloor = S$]])
apply(*simp add*: *Abs-Set-0-inverse discr-eq-false-true*)

apply(*case-tac y*) **apply**(*simp add*: *bot-fun-def bot-Set-0-def*)+
apply(*case-tac a*) **apply**(*simp add*: *null-fun-def null-Set-0-def*)+
done
qed

lemma *S-lift'* :
assumes *S-all-def* : $(\tau :: '\mathfrak{A}\ st) \models \delta\ S$
shows $\exists\, S'.\ (\lambda a\ (\text{-}::'\mathfrak{A}\ st).\ a)\ `\ \lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil = (\lambda a\ (\text{-}::'\mathfrak{A}\ st).\ \lfloor a\rfloor)\ `\ S'$
apply(*rule-tac* $x = (\lambda a.\ \lceil a\rceil)\ `\ \lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil$ **in** *exI*)
apply(*simp only*: *image-comp*[*symmetric*])
apply(*simp add*: *comp-def*)
apply(*subgoal-tac* $\forall\, x \in \lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil.\ \lfloor\lceil x\rceil\rfloor = x$)
apply(*rule equalityI*)

apply(*rule subsetI*)
apply(*drule imageE*) **prefer** *2* **apply** *assumption*
apply(*drule-tac* $x = a$ **in** *ballE*) **prefer** *3* **apply** *assumption*
apply(*drule-tac* $f = \lambda x\ \tau.\ \lfloor\lceil x\rceil\rfloor$ **in** *imageI*)
apply(*simp*)
apply(*simp*)

apply(*rule subsetI*)
apply(*drule imageE*) **prefer** *2* **apply** *assumption*
apply(*drule-tac* $x = xa$ **in** *ballE*) **prefer** *3* **apply** *assumption*
apply(*drule-tac* $f = \lambda x\ \tau.\ x$ **in** *imageI*)
apply(*simp*)
apply(*simp*)

apply(*rule ballI*)
apply(*drule Set-inv-lemma'*[*OF S-all-def*])
apply(*case-tac x*, *simp add*: *bot-option-def foundation18'*)
apply(*simp*)
done

lemma *invalid-set-OclNot-defined* [*simp*,*code-unfold*]:$\delta(invalid::('\mathfrak{A},'\alpha::null)\ Set) = false$ **by** *simp*
lemma *null-set-OclNot-defined* [*simp*,*code-unfold*]:$\delta(null::('\mathfrak{A},'\alpha::null)\ Set) = false$
by(*simp add*: *defined-def null-fun-def*)
lemma *invalid-set-valid* [*simp*,*code-unfold*]:$\upsilon(invalid::('\mathfrak{A},'\alpha::null)\ Set) = false$
by *simp*
lemma *null-set-valid* [*simp*,*code-unfold*]:$\upsilon(null::('\mathfrak{A},'\alpha::null)\ Set) = true$
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 $('\mathfrak{A},('\mathfrak{A},('\mathfrak{A})\ Integer)\ Set)\ Set$ corresponding exactly to Set(Set(Integer)) in OCL notation. Note that the parameter $\mathfrak{A}$ still refers to the object universe; making the OCL semantics entirely parametric in the object universe makes it possible to study (and prove) its properties independently from a concrete class diagram.

4.3.3. Constants on Sets

definition $mtSet$::($'\mathfrak{A}$,$'\alpha$::$null$) Set ($Set\{\}$)
where $Set\{\} \equiv (\lambda\ \tau.\ \ Abs\text{-}Set\text{-}0\ \lfloor\lfloor\{\}::'\alpha\ set\rfloor\rfloor\ )$

lemma $mtSet\text{-}defined[simp,code\text{-}unfold]{:}\delta(Set\{\}) = true$
apply(*rule ext, auto simp: mtSet-def defined-def null-Set-0-def bot-Set-0-def bot-fun-def null-fun-def*)
apply(*simp-all add: Abs-Set-0-inject bot-option-def null-Set-0-def null-option-def*)
done

lemma $mtSet\text{-}valid[simp,code\text{-}unfold]{:}\upsilon(Set\{\}) = true$
apply(*rule ext,auto simp: mtSet-def valid-def null-Set-0-def bot-Set-0-def bot-fun-def null-fun-def*)
apply(*simp-all add: Abs-Set-0-inject bot-option-def null-Set-0-def null-option-def*)
done

lemma $mtSet\text{-}rep\text{-}set{:}\ \lceil\lceil Rep\text{-}Set\text{-}0\ \ (Set\{\}\ \tau)\rceil\rceil = \{\}$
apply(*simp add: mtSet-def, subst Abs-Set-0-inverse*)
by(*simp add: bot-option-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 $OclIncluding$ :: $[('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ val] \Rightarrow ('\mathfrak{A},'\alpha)\ Set$
where $OclIncluding\ x\ y = (\lambda\ \tau.\ if\ \ (\delta\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau$
$then\ Abs\text{-}Set\text{-}0\ \lfloor\lfloor\ \lceil\lceil Rep\text{-}Set\text{-}0\ \ (x\ \tau)\rceil\rceil\ \cup \{y\ \tau\}\ \rfloor\rfloor$
$else\ \bot\ )$
notation $OclIncluding$ (-–>*including*'(-'))

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

definition $OclExcluding$ :: $[('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ val] \Rightarrow ('\mathfrak{A},'\alpha)\ Set$
where $OclExcluding\ x\ y = (\lambda\ \tau.\ \ if\ \ (\delta\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau$
$then\ Abs\text{-}Set\text{-}0\ \lfloor\lfloor\ \lceil\lceil Rep\text{-}Set\text{-}0\ \ (x\ \tau)\rceil\rceil - \{y\ \tau\}\ \rfloor\rfloor$
$else\ \bot\ )$

notation *OclExcluding* (-->*excluding*'(-'))

definition *OclIncludes* :: $[('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ val] \Rightarrow '\mathfrak{A}\ Boolean$
where $OclIncludes\ x\ y = (\lambda\ \tau.\ \ if\ (\delta\ x)\ \tau = true\ \tau \wedge (\upsilon\ y)\ \tau = true\ \tau$
$then\ \lfloor\lfloor(y\ \tau) \in \lceil\lceil Rep\text{-}Set\text{-}0\ (x\ \tau)\rceil\rceil\ \rfloor\rfloor$
$else\ \perp\ )$
notation *OclIncludes* (-->*includes*'(-') [66,65]65)

definition *OclExcludes* :: $[('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ val] \Rightarrow '\mathfrak{A}\ Boolean$
where $OclExcludes\ x\ y = (not(OclIncludes\ x\ y))$
notation *OclExcludes* (-->*excludes*'(-') [66,65]65)

The case of the size definition is somewhat special, we admit explicitly in Featherweight OCL the possibility of infinite sets. For the size definition, this requires an extra condition that assures that the cardinality of the set is actually a defined integer.

definition *OclSize* :: $('\mathfrak{A},'\alpha::null)Set \Rightarrow '\mathfrak{A}\ Integer$
where $OclSize\ x = (\lambda\ \tau.\ if\ (\delta\ x)\ \tau = true\ \tau \wedge finite(\lceil\lceil Rep\text{-}Set\text{-}0\ (x\ \tau)\rceil\rceil)$
$then\ \lfloor\lfloor\ int(card\ \lceil\lceil Rep\text{-}Set\text{-}0\ (x\ \tau)\rceil\rceil)\ \rfloor\rfloor$
$else\ \perp\ )$
notation
OclSize (-->*size*'(') [66])

The following definition follows the requirement of the standard to treat null as neutral element of sets. It is a well-documented exception from the general strictness rule and the rule that the distinguished argument self should be non-null.

definition *OclIsEmpty* :: $('\mathfrak{A},'\alpha::null)\ Set \Rightarrow '\mathfrak{A}\ Boolean$
where $OclIsEmpty\ x = \ ((\upsilon\ x\ and\ not\ (\delta\ x))\ or\ ((OclSize\ x) \doteq \mathbf{0}))$
notation *OclIsEmpty* (-->*isEmpty*'(') [66])

definition *OclNotEmpty* :: $('\mathfrak{A},'\alpha::null)\ Set \Rightarrow '\mathfrak{A}\ Boolean$
where $OclNotEmpty\ x = \ not(OclIsEmpty\ x)$
notation *OclNotEmpty* (-->*notEmpty*'(') [66])

definition *Ocl-Any* :: $[('\mathfrak{A},'\alpha::null)\ Set] \Rightarrow ('\mathfrak{A},'\alpha)\ val$
where $Ocl\text{-}Any\ x = (\lambda\ \tau.\ if\ (\upsilon\ x)\ \tau = true\ \tau$
$then\ if\ (\delta\ x\ and\ OclNotEmpty\ x)\ \tau = true\ \tau\ then\ SOME\ y.\ y \in \lceil\lceil Rep\text{-}Set\text{-}0$
$(x\ \tau)\rceil\rceil$
$else\ null\ \tau$
$else\ \perp\ )$
notation *Ocl-Any* (-->*any*'('))

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

definition *OclForall* :: $[('\mathfrak{A},'\alpha::null)Set,('\mathfrak{A},'\alpha)val \Rightarrow ('\mathfrak{A})Boolean] \Rightarrow '\mathfrak{A}\ Boolean$
where $OclForall\ S\ P = (\lambda\ \tau.\ if\ (\delta\ S)\ \tau = true\ \tau$
$then\ if\ (\exists\ x\in\lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil.\ P(\lambda\ \text{-}.\ x)\ \tau = false\ \tau)$
$then\ false\ \tau$
$else\ if\ (\exists\ x\in\lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil.\ P(\lambda\ \text{-}.\ x)\ \tau = \perp\ \tau)$

$then \perp \tau$
$else\ if\ (\exists x \in \lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil.\ P(\lambda\ \text{-}.\ x)\ \tau = null\ \tau)$
$then\ null\ \tau$
$else\ true\ \tau$
$else \perp)$

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

Like OclForall, OclExists is also not strict.

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

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

definition $OclIterate_{Set} :: [('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\beta::null)val,$
$('\mathfrak{A},'\alpha)val \Rightarrow ('\mathfrak{A},'\beta)val \Rightarrow ('\mathfrak{A},'\beta)val] \Rightarrow ('\mathfrak{A},'\beta)val$
where $OclIterate_{Set}\ S\ A\ F = (\lambda\ \tau.\ if\ (\delta\ S)\ \tau = true\ \tau \wedge (\upsilon\ A)\ \tau = true\ \tau \wedge finite\lceil\lceil Rep\text{-}Set\text{-}0$
$(S\ \tau)\rceil\rceil$
$then\ (Finite\text{-}Set.fold\ (F)\ (A)\ ((\lambda a\ \tau.\ a)\ `\ \lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil))\tau$
$else \perp)$

syntax
$\text{-}OclIterate :: [('\mathfrak{A},'\alpha::null)\ Set,\ idt,\ idt,\ '\alpha,\ '\beta] => ('\mathfrak{A},'\gamma)val$
$(\text{-}\ \text{--}{>}iterate'(\text{-};\text{-}=\text{-} \mid \text{-}')\ [71,100,70]50)$
translations
$X\text{--}{>}iterate(a;\ x = A \mid P) == CONST\ OclIterate_{Set}\ X\ A\ (\%a.\ (\%\ x.\ P))$

definition $OclSelect_{set} :: [('\mathfrak{A},'\alpha::null)Set,('\mathfrak{A},'\alpha)val \Rightarrow ('\mathfrak{A})Boolean] \Rightarrow ('\mathfrak{A},'\alpha)Set$
where $OclSelect_{set}\ S\ P = (\lambda\tau.\ if\ (\delta\ S)\ \tau = true\ \tau$
$then\ if\ (\exists x \in \lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil.\ P(\lambda\ \text{-}.\ x)\ \tau = \perp\ \tau)$
$then \perp$
$else\ Abs\text{-}Set\text{-}0\ \lfloor\lfloor\ \{\ x \in \lceil\lceil\ Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil.\ P\ (\lambda\text{-}.\ x)\ \tau \neq false\ \tau\ \}$
$\rfloor\rfloor$
$else \perp)$

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

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

Definition (futur operators)

consts

$OclUnion$ :: $[('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow ('\mathfrak{A},'\alpha)\ Set$
$OclIntersection$:: $[('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow ('\mathfrak{A},'\alpha)\ Set$
$OclIncludesAll$ :: $[('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow '\mathfrak{A}\ Boolean$
$OclExcludesAll$ :: $[('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow '\mathfrak{A}\ Boolean$
$OclComplement$ :: $('\mathfrak{A},'\alpha::null)\ Set \Rightarrow ('\mathfrak{A},'\alpha)\ Set$
$OclSum$ :: $('\mathfrak{A},'\alpha::null)\ Set \Rightarrow '\mathfrak{A}\ Integer$
$OclCount$ :: $[('\mathfrak{A},'\alpha::null)\ Set,('\mathfrak{A},'\alpha)\ Set] \Rightarrow '\mathfrak{A}\ Integer$

notation
OclCount (-->count'(-') [66,65]65)
notation
OclSum (-->sum'(') [66])
notation
OclIncludesAll (-->includesAll'(-') [66,65]65)
notation
OclExcludesAll (-->excludesAll'(-') [66,65]65)
notation
OclComplement (-->complement'('))
notation
OclUnion (-->union'(-') [66,65]65)
notation
OclIntersection(-->intersection'(-') [71,70]70)

4.4.2. Validity and Definedness Properties

OclIncluding

lemma *including-defined-args-valid*:
$(\tau \models \delta(X{-}{>}including(x))) = ((\tau \models(\delta\ X)) \land (\tau \models(\upsilon\ x)))$
proof −
have $A : \bot \in \{X.\ X = bot \lor X = null \lor (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$ **by**(*simp add*: *bot-option-def*)
have $B : \lfloor\bot\rfloor \in \{X.\ X = bot \lor X = null \lor (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$ **by**(*simp add*: *null-option-def bot-option-def*)
have $C : (\tau \models(\delta\ X)) \Longrightarrow (\tau \models(\upsilon\ x)) \Longrightarrow \lfloor\lfloor insert\ (x\ \tau)\ \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil\rfloor\rfloor \in \{X.\ X = bot \lor X = null \lor (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
apply(*frule Set-inv-lemma*)
apply(*simp add*: *foundation18 invalid-def*)
done
have D: $(\tau \models \delta(X{-}{>}including(x))) \Longrightarrow ((\tau \models(\delta\ X)) \land (\tau \models(\upsilon\ x)))$
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: $(\tau \models (\delta\ X)) \Longrightarrow (\tau \models (\upsilon\ x)) \Longrightarrow (\tau \models \delta(X{-}{>}including(x)))$
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*: *bot-option-def*)
done
show *?thesis* **by**(*auto dest*:*D intro*:*E*)
qed

lemma *including-valid-args-valid*:
$(\tau \models \upsilon(X{-}{>}including(x))) = ((\tau \models (\delta\ X)) \wedge (\tau \models (\upsilon\ x)))$
proof −
have D: $(\tau \models \upsilon(X{-}{>}including(x))) \Longrightarrow ((\tau \models (\delta\ X)) \wedge (\tau \models (\upsilon\ x)))$
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: $(\tau \models (\delta\ X)) \Longrightarrow (\tau \models (\upsilon\ x)) \Longrightarrow (\tau \models \upsilon(X{-}{>}including(x)))$
by(*simp add*: *foundation20 including-defined-args-valid*)
show *?thesis* **by**(*auto dest*:*D intro*:*E*)
qed

lemma *including-defined-args-valid′*[*simp*,*code-unfold*]:
$\delta(X{-}{>}including(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
by(*auto intro*!: *transform2-rev simp*:*including-defined-args-valid foundation10 defined-and-I*)

lemma *including-valid-args-valid″*[*simp*,*code-unfold*]:
$\upsilon(X{-}{>}including(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
by(*auto intro*!: *transform2-rev simp*:*including-valid-args-valid foundation10 defined-and-I*)

OclExcluding

lemma *excluding-defined-args-valid*:
$(\tau \models \delta(X{-}{>}excluding(x))) = ((\tau \models (\delta\ X)) \wedge (\tau \models (\upsilon\ x)))$
proof −
have A : $\bot \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$ **by**(*simp add*: *bot-option-def*)
have B : $\lfloor\bot\rfloor \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$ **by**(*simp add*: *null-option-def bot-option-def*)
have C : $(\tau \models (\delta\ X)) \Longrightarrow (\tau \models (\upsilon\ x)) \Longrightarrow \lfloor\lfloor\lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil - \{x\ \tau\}\rfloor\rfloor \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
apply(*frule Set-inv-lemma*)
apply(*simp add*: *foundation18 invalid-def*)
done
have D: $(\tau \models \delta(X{-}{>}excluding(x))) \Longrightarrow ((\tau \models (\delta\ X)) \wedge (\tau \models (\upsilon\ x)))$

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: $(\tau \models(\delta\ X)) \Longrightarrow (\tau \models(\upsilon\ x)) \Longrightarrow (\tau \models \delta(X{-}{>}excluding(x)))$
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 *excluding-valid-args-valid*:
$(\tau \models \upsilon(X{-}{>}excluding(x))) = ((\tau \models(\delta\ X)) \land (\tau \models(\upsilon\ x)))$
proof −
have D: $(\tau \models \upsilon(X{-}{>}excluding(x))) \Longrightarrow ((\tau \models(\delta\ X)) \land (\tau \models(\upsilon\ x)))$
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: $(\tau \models(\delta\ X)) \Longrightarrow (\tau \models(\upsilon\ x)) \Longrightarrow (\tau \models \upsilon(X{-}{>}excluding(x)))$
by(*simp add*: *foundation20 excluding-defined-args-valid*)
show *?thesis* **by**(*auto dest*:*D intro*:*E*)
qed

lemma *excluding-valid-args-valid'*[*simp*,*code-unfold*]:
$\delta(X{-}{>}excluding(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
by(*auto intro*!: *transform2-rev simp*:*excluding-defined-args-valid foundation10 defined-and-I*)

lemma *excluding-valid-args-valid''*[*simp*,*code-unfold*]:
$\upsilon(X{-}{>}excluding(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
by(*auto intro*!: *transform2-rev simp*:*excluding-valid-args-valid foundation10 defined-and-I*)

OclIncludes

lemma *includes-defined-args-valid*:
$(\tau \models \delta(X{-}{>}includes(x))) = ((\tau \models(\delta\ X)) \land (\tau \models(\upsilon\ x)))$
proof −
have A: $(\tau \models \delta(X{-}{>}includes(x))) \Longrightarrow ((\tau \models(\delta\ X)) \land (\tau \models(\upsilon\ x)))$
by(*auto simp*: *OclIncludes-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: $(\tau \models(\delta\ X)) \Longrightarrow (\tau \models(\upsilon\ x)) \Longrightarrow (\tau \models \delta(X{-}{>}includes(x)))$
by(*auto simp*: *OclIncludes-def OclValid-def true-def false-def StrongEq-def defined-def invalid-def valid-def bot-fun-def null-fun-def*

bot-option-def null-option-def
split: *bool.split-asm HOL.split-if-asm option.split*)
show *?thesis* **by**(*auto dest*:*A intro*:*B*)
qed

lemma *includes-valid-args-valid*:
$(\tau \models \upsilon(X->includes(x))) = ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
proof −
have *A*: $(\tau \models \upsilon(X->includes(x))) \Longrightarrow ((\tau \models(\delta\ X)) \wedge (\tau \models(\upsilon\ x)))$
by(*auto simp*: *OclIncludes-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*: $(\tau \models(\delta\ X)) \Longrightarrow (\tau \models(\upsilon\ x)) \Longrightarrow (\tau \models \upsilon(X->includes(x)))$
by(*auto simp*: *OclIncludes-def OclValid-def true-def false-def StrongEq-def*
defined-def invalid-def valid-def bot-fun-def null-fun-def
bot-option-def null-option-def
split: *bool.split-asm HOL.split-if-asm option.split*)
show *?thesis* **by**(*auto dest*:*A intro*:*B*)
qed

lemma *includes-valid-args-valid′*[*simp*,*code-unfold*]:
$\delta(X->includes(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
by(*auto intro*!: *transform2-rev simp*:*includes-defined-args-valid foundation10 defined-and-I*)

lemma *includes-valid-args-valid″*[*simp*,*code-unfold*]:
$\upsilon(X->includes(x)) = ((\delta\ X)\ and\ (\upsilon\ x))$
by(*auto intro*!: *transform2-rev simp*:*includes-valid-args-valid foundation10 defined-and-I*)

OclNotEmpty

lemma *notempty-has-elt* : $\tau \models \delta\ X \Longrightarrow$
$\tau \models X->notEmpty() \Longrightarrow$
$\exists\, e.\ e \in \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil$
apply(*simp add*: *OclNotEmpty-def OclIsEmpty-def deMorgan1 deMorgan2*, *drule foundation5*)
apply(*subst* (*asm*) (*2*) *OclNot-def*,
simp add: *OclValid-def* $StrictRefEq_{Integer}$ *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*)

Ocl Any

lemma *any-valid-args-valid*[*simp*,*code-unfold*]:
$(\tau \models \upsilon(X->any())) = (\tau \models \upsilon\ X)$
proof −
have *A*: $(\tau \models \upsilon(X->any())) \Longrightarrow ((\tau \models(\upsilon\ X)))$
by(*auto simp*: *Ocl-Any-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: $(\tau \models (\upsilon\ X)) \Longrightarrow (\tau \models \upsilon(X{-}{>}any()))$
apply(*auto simp*: *Ocl-Any-def OclValid-def true-def false-def StrongEq-def*
defined-def invalid-def valid-def bot-fun-def null-fun-def
bot-option-def null-option-def null-is-valid
OclAnd-def
split: *bool.split-asm HOL.split-if-asm option.split*)
apply(*frule Set-inv-lemma*[*OF foundation16*[*THEN iffD2*], *OF conjI*], *simp*)
apply(*subgoal-tac* $(\delta\ X)\ \tau = true\ \tau$)
prefer *2*
apply (*metis* (*hide-lams*, *no-types*) *OclValid-def foundation16*)
apply(*simp add*: *true-def*)
apply(*drule notempty-has-elt*[*simplified OclValid-def true-def*], *simp*)
apply(*drule exE*) **prefer** *2* **apply**(*simp*)
proof − **fix** e **show** $X\ \tau \neq null \Longrightarrow$
$(SOME\ y.\ y \in \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil) = \bot \Longrightarrow$
$\forall x{\in}\lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil.\ x \neq \bot \Longrightarrow e \in \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil \Longrightarrow$
False
apply(*subgoal-tac* $(SOME\ x.\ x \in \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil) \neq \bot$)
prefer *2*
apply(*rule someI2*[**where** $Q = \lambda x.\ x \neq \bot$ **and** $P = \lambda y.\ y \in \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil$
and $a = e$], *simp*)
apply(*simp*)
apply(*simp*)
done
apply-end(*simp*)+
qed
show *?thesis* **by**(*auto dest*:*A intro*:*B*)
qed

lemma *any-valid-args-valid''*[*simp*,*code-unfold*]:
$\upsilon(X{-}{>}any()) = (\upsilon\ X)$
by(*auto intro*!: *transform2-rev*)

4.4.3. Execution with invalid or null as argument

OclIncluding

lemma *including-strict1*[*simp*,*code-unfold*]:$(invalid{-}{>}including(x)) = invalid$
by(*simp add*: *bot-fun-def OclIncluding-def invalid-def defined-def valid-def false-def true-def*)

lemma *including-strict2*[*simp*,*code-unfold*]:$(X{-}{>}including(invalid)) = invalid$
by(*simp add*: *OclIncluding-def invalid-def bot-fun-def defined-def valid-def false-def true-def*)

lemma *including-strict3*[*simp*,*code-unfold*]:$(null{-}{>}including(x)) = invalid$
by(*simp add*: *OclIncluding-def invalid-def bot-fun-def defined-def valid-def false-def true-def*)

OclExcluding

lemma *excluding-strict1* [*simp*,*code-unfold*]:(*invalid*−>*excluding*(*x*)) = *invalid*
by(*simp add*: *bot-fun-def OclExcluding-def invalid-def defined-def valid-def false-def true-def*)

lemma *excluding-strict2* [*simp*,*code-unfold*]:(*X*−>*excluding*(*invalid*)) = *invalid*
by(*simp add*: *OclExcluding-def invalid-def bot-fun-def defined-def valid-def false-def true-def*)

lemma *excluding-strict3* [*simp*,*code-unfold*]:(*null*−>*excluding*(*x*)) = *invalid*
by(*simp add*: *OclExcluding-def invalid-def bot-fun-def defined-def valid-def false-def true-def*)

OclIncludes

lemma *includes-strict1* [*simp*,*code-unfold*]:(*invalid*−>*includes*(*x*)) = *invalid*
by(*simp add*: *bot-fun-def OclIncludes-def invalid-def defined-def valid-def false-def true-def*)

lemma *includes-strict2* [*simp*,*code-unfold*]:(*X*−>*includes*(*invalid*)) = *invalid*
by(*simp add*: *OclIncludes-def invalid-def bot-fun-def defined-def valid-def false-def true-def*)

lemma *includes-strict3* [*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*)

OclSize

lemma *size-strict1* [*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 *size-strict3* [*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 *isEmpty-strict1* [*simp*,*code-unfold*]:(*invalid*−>*isEmpty*()) = *invalid*
by(*simp add*: *OclIsEmpty-def*)

lemma *isEmpty-strict3* [*simp*,*code-unfold*]:(*null*−>*isEmpty*()) = *true*
by(*simp add*: *OclIsEmpty-def*)

OclNotEmpty

lemma *notEmpty-strict1* [*simp*,*code-unfold*]:(*invalid*−>*notEmpty*()) = *invalid*
by(*simp add*: *OclNotEmpty-def*)

lemma *notEmpty-strict3* [*simp*,*code-unfold*]:(*null*−>*notEmpty*()) = *false*
by(*simp add*: *OclNotEmpty-def*)

Ocl Any

lemma *any-strict1* [*simp*,*code-unfold*]:(*invalid*−>*any*()) = *invalid*

by(*simp add*: *bot-fun-def Ocl-Any-def invalid-def defined-def valid-def false-def true-def*)

lemma *any-strict3*[*simp*,*code-unfold*]:(*null*$->$*any*()) = *null*
by(*simp add*: *Ocl-Any-def false-def true-def*)

OclIterate

lemma $OclIterate_{Set}$*-strict1*[*simp*,*code-unfold*]:*invalid*$->$*iterate*(a; x = A | $P\ a\ x$) = *invalid*
by(*simp add*: *bot-fun-def invalid-def* $OclIterate_{Set}$*-def defined-def valid-def false-def true-def*)

lemma $OclIterate_{Set}$*-null1*[*simp*,*code-unfold*]:*null*$->$*iterate*(a; x = A | $P\ a\ x$) = *invalid*
by(*simp add*: *bot-fun-def invalid-def* $OclIterate_{Set}$*-def defined-def valid-def false-def true-def*)

lemma $OclIterate_{Set}$*-strict2*[*simp*,*code-unfold*]:$S$$->$*iterate*($a$; x = *invalid* | $P\ a\ x$) = *invalid*
by(*simp add*: *bot-fun-def invalid-def* $OclIterate_{Set}$*-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

4.4.4. Context Passing

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

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

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

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

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

lemma *cp-OclIsEmpty*: $X$$->$*isEmpty*() τ = (λ-. $X\ \tau$)$->$*isEmpty*() τ
apply(*simp add*: *OclIsEmpty-def*)
apply(*subst* (*2*) *cp-OclOr*)

apply(*subst cp-OclAnd*)
apply(*subst cp-OclNot*)
apply(*subst* $cp\text{-}StrictRefEq_{Integer}$)
apply(*simp add*: *cp-defined*[*symmetric*] *cp-valid*[*symmetric*]
$cp\text{-}StrictRefEq_{Integer}$[*symmetric*] *cp-OclSize*[*symmetric*] *cp-OclNot*[*symmetric*] *cp-OclAnd*[*symmetric*]
cp-OclOr[*symmetric*])
done

lemma *cp-OclNotEmpty*: $X{-}{>}notEmpty()\ \tau = (\lambda\text{-}.\ X\ \tau){-}{>}notEmpty()\ \tau$
apply(*simp add*: *OclNotEmpty-def*)
apply(*subst* (*2*) *cp-OclNot*)
apply(*simp add*: *cp-OclNot*[*symmetric*] *cp-OclIsEmpty*[*symmetric*])
done

lemma *cp-Ocl-Any*: $X{-}{>}any()\ \tau = (\lambda\text{-}.\ X\ \tau){-}{>}any()\ \tau$
apply(*simp only*: *Ocl-Any-def*)
apply(*subst* (*2*) *cp-OclAnd*)
apply(*simp only*: *cp-OclAnd*[*symmetric*] *cp-defined*[*symmetric*] *cp-valid*[*symmetric*] *cp-OclNotEmpty*[*symmet*
done

lemma *cp-OclForall*:
$(S{-}{>}forAll(x \mid P\ x))\ \tau = ((\lambda\ \text{-}.\ S\ \tau){-}{>}forAll(x \mid P\ (\lambda\ \text{-}.\ x\ \tau)))\ \tau$
by(*simp add*: *OclForall-def cp-defined*[*symmetric*])

lemma *cp-OclForall1* [*simp*,*intro*!]:
$cp\ S \Longrightarrow cp\ (\lambda X.\ ((S\ X){-}{>}forAll(x \mid P\ x)))$
apply(*simp add*: *cp-def*)
apply(*erule exE*, *rule exI*, *rule allI*, *rule allI*, *rule allI*)
apply(*erule-tac x*=*X* **in** *allE*)
apply(*subst cp-OclForall*)
apply(*simp*)
done

lemma *cp-OclForall2* [*simp*,*intro*!]:
$cp\ (\lambda X\ St\ x.\ P\ (\lambda\tau.\ x)\ X\ St) \Longrightarrow cp\ S \Longrightarrow cp\ (\lambda X.\ (S\ X){-}{>}forAll(x|P\ x\ X))$
apply(*simp only*: *cp-def*)
oops

lemma *cp-OclForall*:
$cp\ S \Longrightarrow$
$(\bigwedge\ x.\ cp(P\ x)) \Longrightarrow$
$cp(\lambda X.\ ((S\ X){-}{>}forAll(x \mid P\ x\ X)))$
oops

lemma $cp\text{-}OclIterate_{Set}$: $(X{-}{>}iterate(a;\ x = A \mid P\ a\ x))\ \tau =$

$((\lambda\ \text{-}.\ X\ \tau) -\!\!> \mathit{iterate}(a;\ x = A \mid P\ a\ x))\ \tau$
by(*simp add*: $\mathit{OclIterate}_{Set}$*-def cp-defined*[*symmetric*])

lemmas *cp-intro''*[*simp*,*intro*!] =
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-OclSize [*THEN allI*[*THEN allI*[*THEN cpI1*], *of OclSize*]]
cp-Ocl-Any [*THEN allI*[*THEN allI*[*THEN cpI1*], *of Ocl-Any*]]

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:

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

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 id 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 $\mathit{StrictRefEq}_{Set}$*-refl*[*simp*,*code-unfold*]:
$((x{::}('\mathfrak{A},'\alpha{::}null)Set) \doteq x) = (\mathit{if}\ (\upsilon\ x)\ \mathit{then}\ \mathit{true}\ \mathit{else}\ \mathit{invalid}\ \mathit{endif})$
by(*rule ext*, *simp add*: $\mathit{StrictRefEq}_{Set}$ *OclIf-def*)

Symmetry

lemma $\mathit{StrictRefEq}_{Set}$*-sym*:
$((x{::}('\mathfrak{A},'\alpha{::}null)Set) \doteq y) = (y \doteq x)$
by(*simp add*: $\mathit{StrictRefEq}_{Set}$, *subst StrongEq-sym*, *rule ext*, *simp*)

Execution with invalid or null as argument

lemma *StrictRefEq*$_{Set}$*-strict1*: $((x{::}('\mathfrak{A},'\alpha{::}null)Set) \doteq invalid) = invalid$
by(*simp add*:*StrictRefEq*$_{Set}$ *false-def true-def*)

lemma *StrictRefEq*$_{Set}$*-strict2*: $(invalid \doteq (y{::}('\mathfrak{A},'\alpha{::}null)Set)) = invalid$
by(*simp add*:*StrictRefEq*$_{Set}$ *false-def true-def*)

lemma *StrictRefEq*$_{Set}$*-strictEq-valid-args-valid*:
$(\tau \models \delta\ ((x{::}('\mathfrak{A},'\alpha{::}null)Set) \doteq y)) = ((\tau \models (\upsilon\ x)) \wedge (\tau \models \upsilon\ y))$
proof −
  have *A*: $\tau \models \delta\ (x \doteq y) \Longrightarrow \tau \models \upsilon\ x \wedge \tau \models \upsilon\ y$
    apply(*simp add*: *StrictRefEq*$_{Set}$ *valid-def OclValid-def defined-def*)
    apply(*simp add*: *invalid-def bot-fun-def split*: *split-if-asm*)
    done
  have *B*: $(\tau \models \upsilon\ x) \wedge (\tau \models \upsilon\ y) \Longrightarrow \tau \models \delta\ (x \doteq y)$
    apply(*simp add*: *StrictRefEq*$_{Set}$, *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 *?thesis* **by**(*auto intro*!: *A B*)
qed

Behavior vs StrongEq

lemma *StrictRefEq*$_{Set}$*-vs-StrongEq*:
$\tau \models \upsilon\ x \Longrightarrow \tau \models \upsilon\ y \Longrightarrow (\tau \models (((x{::}('\mathfrak{A},'\alpha{::}null)Set) \doteq y) \triangleq (x \triangleq y)))$
apply(*drule foundation13*[*THEN iffD2*],*drule foundation13*[*THEN iffD2*])
by(*simp add*:*StrictRefEq*$_{Set}$ *foundation22*)

Context Passing

lemma *cp-StrictRefEq*$_{Set}$:$((X{::}('\mathfrak{A},'\alpha{::}null)Set) \doteq Y)\ \tau = ((\lambda\text{-}.\ X\ \tau) \doteq (\lambda\text{-}.\ Y\ \tau))\ \tau$
by(*simp add*:*StrictRefEq*$_{Set}$ *cp-StrongEq*[*symmetric*] *cp-valid*[*symmetric*])

4.6. Execution on Set's Operators

4.6.1. OclIncluding

lemma *including-charn0*[*simp*]:
assumes *val-x*:$\tau \models (\upsilon\ x)$
shows $\tau \models not(Set\{\}{-}{>}includes(x))$
using *val-x*
apply(*auto simp*: *OclValid-def OclIncludes-def OclNot-def false-def true-def*)
apply(*auto simp*: *mtSet-def OCL-lib.Set-0.Abs-Set-0-inverse*)
done

lemma *including-charn0'*[*simp*,*code-unfold*]:
Set{}−>*includes*(x) = (*if* υ *x then false else invalid endif*)
proof −
  have *A*: $\bigwedge \tau$. (*Set*{}−>*includes*(*invalid*)) τ = (*if* (υ *invalid*) *then false else invalid endif*) τ
      by *simp*
  have *B*: $\bigwedge \tau\ x.\ \tau \models (\upsilon\ x) \Longrightarrow$ (*Set*{}−>*includes*(x)) τ = (*if* υ *x then false else invalid endif*) τ
      apply(*frule including-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* τ)
    apply(*case-tac* $\tau \models (\upsilon\ x)$)
    apply(*simp-all add*: *B foundation18*)
    apply(*subst cp-OclIncludes*, *simp add*: *cp-OclIncludes*[*symmetric*] *A*)
  done
qed

lemma *including-charn1*:
assumes *def-X*:$\tau \models (\delta\ X)$
assumes *val-x*:$\tau \models (\upsilon\ x)$
shows $\tau \models$ (X−>*including*(x)−>*includes*(x))
proof −
 have *C* : $\lfloor\lfloor$*insert* ($x\ \tau$) $\lceil\lceil$*Rep-Set-0* ($X\ \tau$)$\rceil\rceil\rfloor\rfloor \in \{X.\ X = bot \vee X = null \vee (\forall x \in \lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
      apply(*insert val-x Set-inv-lemma*[*OF def-X*])
      apply(*simp add*: *foundation18 invalid-def*)
      done
 show *?thesis*
 apply(*subst OclIncludes-def*, *simp add*: *def-X*[*simplified OclValid-def*] *val-x*[*simplified OclValid-def*] *foundation10*[*simplified OclValid-def*] *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 *including-charn2*:
assumes *def-X*:$\tau \models (\delta\ X)$
and *val-x*:$\tau \models (\upsilon\ x)$
and *val-y*:$\tau \models (\upsilon\ y)$
and *neq* :$\tau \models not(x \triangleq y)$
shows $\tau \models$ (X−>*including*(x)−>*includes*(y)) $\triangleq$ (X−>*includes*(y))
proof −
 have *C* : $\lfloor\lfloor$*insert* ($x\ \tau$) $\lceil\lceil$*Rep-Set-0* ($X\ \tau$)$\rceil\rceil\rfloor\rfloor \in \{X.\ X = bot \vee X = null \vee (\forall x \in \lceil\lceil X\rceil\rceil.\ x \neq bot)\}$

apply(*insert val-x Set-inv-lemma*[*OF def-X*])
apply(*simp add*: *foundation18 invalid-def*)
done
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*] *val-x*[*simplified OclValid-def*] *val-y*[*simplified OclValid-def*] *Abs-Set-0-inverse*[*OF C*] *true-def*)
by(*metis foundation22 foundation6 foundation9 neq*)
qed

One would like a generic theorem of the form:

```
lemma includes_execute[code_unfold]:
"(X->including(x)->includes(y)) = (if \<delta> X then if x \<doteq> y
                                                     then true
                                                     else X->includes(y)
                                                     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 `includes_execute` becomes generic since it uses strict equality which in itself is generic. It is possible to prove the following generic theorem and instantiate it if a number of properties that link the polymorphic logical, Strong Equality with the concrete instance of strict quality.

lemma *includes-execute-generic*:
assumes *strict1*: $(x \doteq invalid) = invalid$
and *strict2*: $(invalid \doteq y) = invalid$
and *cp-StrictRefEq*: $\bigwedge (X::('\mathfrak{A},'a::null)val)\ Y\ \tau.\ (X \doteq Y)\ \tau = ((\lambda\text{-}.\ X\ \tau) \doteq (\lambda\text{-}.\ Y\ \tau))\ \tau$
and *StrictRefEq-vs-StrongEq*: $\bigwedge (x::('\mathfrak{A},'a::null)val)\ y\ \tau.$
$\tau \models \upsilon\ x \Longrightarrow \tau \models \upsilon\ y \Longrightarrow (\tau \models ((x \doteq y) \triangleq (x \triangleq y)))$
shows
$(X{-}{>}including(x::('\mathfrak{A},'a::null)val){-}{>}includes(y)) =$
$(if\ \delta\ X\ then\ if\ x \doteq y\ then\ true\ else\ X{-}{>}includes(y)\ endif\ else\ invalid\ endif)$
proof −
have *A*: $\bigwedge\tau.\ \tau \models (X \triangleq invalid) \Longrightarrow$
$(X{-}{>}including(x){-}{>}includes(y))\ \tau = invalid\ \tau$
apply(*rule foundation22*[*THEN iffD1*])
by(*erule StrongEq-L-subst2-rev*,*simp*,*simp*)
have *B*: $\bigwedge\tau.\ \tau \models (X \triangleq null) \Longrightarrow$
$(X{-}{>}including(x){-}{>}includes(y))\ \tau = invalid\ \tau$
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*: $\bigwedge\tau.\ \tau \models (x \triangleq \mathit{invalid}) \Longrightarrow$
$(X{-}{>}\mathit{including}(x){-}{>}\mathit{includes}(y))\ \tau =$
$(\mathit{if}\ x \doteq y\ \mathit{then}\ \mathit{true}\ \mathit{else}\ X{-}{>}\mathit{includes}(y)\ \mathit{endif})\ \tau$
apply(*rule foundation22*[*THEN iffD1*])
apply(*erule StrongEq-L-subst2-rev*,*simp*,*simp*)
by (*simp add*: *strict2*)
have *D*: $\bigwedge\tau.\ \tau \models (y \triangleq \mathit{invalid}) \Longrightarrow$
$(X{-}{>}\mathit{including}(x){-}{>}\mathit{includes}(y))\ \tau =$
$(\mathit{if}\ x \doteq y\ \mathit{then}\ \mathit{true}\ \mathit{else}\ X{-}{>}\mathit{includes}(y)\ \mathit{endif})\ \tau$
apply(*rule foundation22*[*THEN iffD1*])
apply(*erule StrongEq-L-subst2-rev*,*simp*,*simp*)
by (*simp add*: *strict1*)
have *E*: $\bigwedge\tau.\ \tau \models \upsilon\ x \Longrightarrow \tau \models \upsilon\ y \Longrightarrow$
$(\mathit{if}\ x \doteq y\ \mathit{then}\ \mathit{true}\ \mathit{else}\ X{-}{>}\mathit{includes}(y)\ \mathit{endif})\ \tau =$
$(\mathit{if}\ x \triangleq y\ \mathit{then}\ \mathit{true}\ \mathit{else}\ X{-}{>}\mathit{includes}(y)\ \mathit{endif})\ \tau$
apply(*subst cp-OclIf*)
apply(*subst StrictRefEq-vs-StrongEq*[*THEN foundation22*[*THEN iffD1*]])
by(*simp-all add*: *cp-OclIf*[*symmetric*])
have *F*: $\bigwedge\tau.\ \tau \models (x \triangleq y) \Longrightarrow$
$(X{-}{>}\mathit{including}(x){-}{>}\mathit{includes}(y))\ \tau = (X{-}{>}\mathit{including}(x){-}{>}\mathit{includes}(x))\ \tau$
apply(*rule foundation22*[*THEN iffD1*])
by(*erule StrongEq-L-subst2-rev*,*simp*, *simp*)
show *?thesis*
apply(*rule ext*, *rename-tac* τ)
apply(*case-tac* $\neg\ (\tau \models (\delta\ X))$, *simp add*:*def-split-local*,*elim disjE A B*)
apply(*case-tac* $\neg\ (\tau \models (\upsilon\ x))$,
simp add:*foundation18 foundation22*[*symmetric*],
drule StrongEq-L-sym)
apply(*simp add*: *foundation22 C*)
apply(*case-tac* $\neg\ (\tau \models (\upsilon\ y))$,
simp add:*foundation18 foundation22*[*symmetric*],
drule StrongEq-L-sym, *simp add*: *foundation22 D*, *simp*)
apply(*subst E*,*simp-all*)
apply(*case-tac* $\tau \models \mathit{not}(x \triangleq y)$)
apply(*simp add*: *including-charn2*[*simplified foundation22*])
apply(*simp add*: *foundation9 F*
including-charn1[*THEN foundation13*[*THEN iffD2*],
THEN foundation22[*THEN iffD1*]])
done
qed

schematic-lemma *includes-execute-int*[*simp*,*code-unfold*]: *?X*
by(*rule includes-execute-generic*[*OF StrictRefEq*$_{Integer}$*-strict1 StrictRefEq*$_{Integer}$*-strict2*
cp-StrictRefEq$_{Integer}$

StrictRefEq$_{Integer}$*-vs-StrongEq*], *simp-all*)

schematic-lemma *includes-execute-bool*[*simp*,*code-unfold*]: *?X*
by(*rule includes-execute-generic*[*OF StrictRefEq*$_{Boolean}$*-strict1 StrictRefEq*$_{Boolean}$*-strict2*
cp-StrictRefEq$_{Boolean}$
StrictRefEq$_{Boolean}$*-vs-StrongEq*], *simp-all*)

schematic-lemma *includes-execute-set*[*simp*,*code-unfold*]: *?X*
by(*rule includes-execute-generic*[*OF StrictRefEq*$_{Set}$*-strict1 StrictRefEq*$_{Set}$*-strict2*
cp-StrictRefEq$_{Set}$
StrictRefEq$_{Set}$*-vs-StrongEq*], *simp-all*)

lemma *finite-including-rep-set* :
assumes *X-def* : $\tau \models \delta\ X$
and *x-val* : $\tau \models \upsilon\ x$
shows *finite* $\lceil\lceil$*Rep-Set-0* (*X*$->$*including*(*x*) τ)$\rceil\rceil$ = *finite* $\lceil\lceil$*Rep-Set-0* (*X* τ)$\rceil\rceil$
proof −
have *C* : $\lfloor\lfloor$*insert* (*x* τ) $\lceil\lceil$*Rep-Set-0* (*X* τ)$\rceil\rceil\rfloor\rfloor \in \{X.\ X = bot \lor X = null \lor (\forall x \in \lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
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: *OclIncluding-def Abs-Set-0-inverse*[*OF C*]
dest: *foundation13*[*THEN iffD2*, *THEN foundation22*[*THEN iffD1*]])
qed

lemma *including-includes* :
assumes *a-val* : $\tau \models \upsilon\ a$
and *x-val* : $\tau \models \upsilon\ x$
and *S-incl* : $\tau \models$ (*S* :: ('$\mathfrak{A}$, *int option option*) *Set*)$->$*includes*(*x*)
shows $\tau \models$ *S*$->$*including*(*a*)$->$*includes*(*x*)
proof −
have *discr-eq-bot1-true* : $\bigwedge\tau.\ (\bot\ \tau = true\ \tau) = False$ **by** (*metis OCL-core.bot-fun-def foundation1 foundation18' valid3*)
have *discr-eq-bot2-true* : $\bigwedge\tau.\ (\bot = true\ \tau) = False$ **by** (*metis bot-fun-def discr-eq-bot1-true*)
have *discr-neq-invalid-true* : $\bigwedge\tau.\ (invalid\ \tau \neq true\ \tau) = True$ **by** (*metis discr-eq-bot2-true invalid-def*)
have *discr-eq-invalid-true* : $\bigwedge\tau.\ (invalid\ \tau = true\ \tau) = False$ **by** (*metis bot-option-def invalid-def option.simps*(*2*) *true-def*)
show *?thesis*
apply(*simp*)
apply(*subgoal-tac* $\tau \models \delta\ S$)
prefer *2*
apply(*insert S-incl*[*simplified OclIncludes-def*], *simp add*: *OclValid-def*)
apply(*metis discr-eq-bot2-true*)

apply(*simp add*: *cp-OclIf*[*of* δ *S*] *OclValid-def OclIf-def discr-neq-invalid-true discr-eq-invalid-true x-val*[*simplified OclValid-def*])
by (*metis OclValid-def S-incl* $StrictRefEq_{Integer}$*-strict''* *a-val foundation10 foundation6 x-val*)
qed

lemma *including-rep-set*:
assumes *S-def*: $\tau \models \delta\ S$
shows $\lceil\lceil Rep\text{-}Set\text{-}0\ (S{-}{>}including(\lambda\text{-}.\ \lfloor\lfloor x\rfloor\rfloor)\ \tau)\rceil\rceil = insert\ \lfloor\lfloor x\rfloor\rfloor\ \lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil$
apply(*simp add*: *OclIncluding-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 bot-option-def not-Some-eq*)
by(*simp*)

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

lemma *including-includes-simp*:
assumes $\tau \models X{-}{>}includes(x)$
shows $X{-}{>}including(x)\ \tau = X\ \tau$
proof –
have *includes-def*: $\tau \models X{-}{>}includes(x) \Longrightarrow \tau \models \delta\ X$
by (*metis OCL-core.bot-fun-def OclIncludes-def OclValid-def defined3 foundation16*)

have *includes-val*: $\tau \models X{-}{>}includes(x) \Longrightarrow \tau \models \upsilon\ x$
by (*metis* (*hide-lams*, *no-types*) *foundation6 includes-valid-args-valid' including-valid-args-valid including-valid-args-valid''*)

show *?thesis*
apply(*insert includes-def*[*OF assms*] *includes-val*[*OF assms*] *assms*, *simp add*: *OclIncluding-def OclIncludes-def OclValid-def true-def*)
apply(*drule insert-absorb*, *simp*, *subst abs-rep-simp'*)
by(*simp-all add*: *OclValid-def true-def*)
qed

4.6.2. OclExcluding

lemma *excluding-charn0*[*simp*]:
assumes *val-x*: $\tau \models (\upsilon\ x)$
shows $\tau \models ((Set\{\}{-}{>}excluding(x)) \triangleq Set\{\})$
proof –
have *A* : $\lfloor None\rfloor \in \{X.\ X = bot \lor X = null \lor (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$ **by**(*simp add*: *null-option-def bot-option-def*)
have *B* : $\lfloor\lfloor\{\}\rfloor\rfloor \in \{X.\ X = bot \lor X = null \lor (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$ **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*: *mtSet-def OCL-lib.Set-0.Abs-Set-0-inverse OCL-lib.Set-0.Abs-Set-0-inject*[*OF B A*])
done
qed

lemma *excluding-charn0-exec*[*code-unfold*]:
$(Set\{\}{-}{>}excluding(x)) = (if\ (\upsilon\ x)\ then\ Set\{\}\ else\ invalid\ endif)$
proof −
have *A*: $\bigwedge \tau.\ (Set\{\}{-}{>}excluding(invalid))\ \tau = (if\ (\upsilon\ invalid)\ then\ Set\{\}\ else\ invalid\ endif)\ \tau$
by *simp*
have *B*: $\bigwedge \tau\ x.\ \tau \models (\upsilon\ x) \Longrightarrow (Set\{\}{-}{>}excluding(x))\ \tau = (if\ (\upsilon\ x)\ then\ Set\{\}\ else\ invalid\ endif)\ \tau$
by(*simp add*: *excluding-charn0*[*THEN foundation22*[*THEN iffD1*]])
show *?thesis*
apply(*rule ext*, *rename-tac* τ)
apply(*case-tac* $\tau \models (\upsilon\ x)$)
apply(*simp add*: *B*)
apply(*simp add*: *foundation18*)
apply(*subst cp-OclExcluding*, *simp*)
apply(*simp add*: *cp-OclIf*[*symmetric*] *cp-OclExcluding*[*symmetric*] *cp-valid*[*symmetric*] *A*)
done
qed

lemma *excluding-charn1*:
assumes *def-X*:$\tau \models (\delta\ X)$
and *val-x*:$\tau \models (\upsilon\ x)$
and *val-y*:$\tau \models (\upsilon\ y)$
and *neq* :$\tau \models not(x \triangleq y)$
shows $\tau \models ((X{-}{>}including(x)){-}{>}excluding(y)) \triangleq ((X{-}{>}excluding(y)){-}{>}including(x))$
proof −
have *A* : $\bot \in \{X.\ X = bot \lor X = null \lor (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$ **by**(*simp add*: *bot-option-def*)
have *B* : $\lfloor\bot\rfloor \in \{X.\ X = bot \lor X = null \lor (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$ **by**(*simp add*: *null-option-def bot-option-def*)
have *C* : $\lfloor\lfloor insert\ (x\ \tau)\ \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil\rfloor\rfloor \in \{X.\ X = bot \lor X = null \lor (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
apply(*insert def-X val-x*, *frule Set-inv-lemma*)
apply(*simp add*: *foundation18 invalid-def*)
done
have *D* : $\lfloor\lfloor\lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil - \{y\ \tau\}\rfloor\rfloor \in \{X.\ X = bot \lor X = null \lor (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
apply(*insert def-X val-x*, *frule Set-inv-lemma*)
apply(*simp add*: *foundation18 invalid-def*)
done

have $E : x\ \tau \neq y\ \tau$
  apply(*insert neq*)
  by(*auto simp*: *OclValid-def bot-fun-def OclIncluding-def OclIncludes-def false-def true-def defined-def valid-def bot-Set-0-def null-fun-def null-Set-0-def StrongEq-def OclNot-def*)

have *G1* : $Abs\text{-}Set\text{-}0\ \lfloor\lfloor insert\ (x\ \tau)\ \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil\rfloor\rfloor \neq Abs\text{-}Set\text{-}0\ None$
  apply(*insert C*, *simp*)
  apply(*simp add*: *def-X val-x A Abs-Set-0-inject B C OclValid-def Rep-Set-0-cases Rep-Set-0-inverse bot-Set-0-def bot-option-def insert-compr insert-def not-Some-eq null-Set-0-def null-option-def*)
done
have *G2* : $Abs\text{-}Set\text{-}0\ \lfloor\lfloor insert\ (x\ \tau)\ \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil\rfloor\rfloor \neq Abs\text{-}Set\text{-}0\ \lfloor None\rfloor$
  apply(*insert C*, *simp*)
  apply(*simp add*: *def-X val-x A Abs-Set-0-inject B C OclValid-def Rep-Set-0-cases Rep-Set-0-inverse bot-Set-0-def bot-option-def insert-compr insert-def not-Some-eq null-Set-0-def null-option-def*)
done

have *G* : $(\delta\ (\lambda\text{-}.\ Abs\text{-}Set\text{-}0\ \lfloor\lfloor insert\ (x\ \tau)\ \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil\rfloor\rfloor))\ \tau = true\ \tau$
  apply(*auto simp*: *OclValid-def false-def true-def defined-def bot-fun-def bot-Set-0-def null-fun-def null-Set-0-def G1 G2*)
done

have *H1* : $Abs\text{-}Set\text{-}0\ \lfloor\lfloor\lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil - \{y\ \tau\}\rfloor\rfloor \neq Abs\text{-}Set\text{-}0\ None$
  apply(*insert D*, *simp*)
  apply(*simp add*: *A Abs-Set-0-inject Abs-Set-0-inverse B C OclExcluding-def OclValid-def Option.set.simps(2) Rep-Set-0-inverse bot-Set-0-def bot-option-def null-Set-0-def null-option-def option.distinct(1)*)
done
have *H2* : $Abs\text{-}Set\text{-}0\ \lfloor\lfloor\lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil - \{y\ \tau\}\rfloor\rfloor \neq Abs\text{-}Set\text{-}0\ \lfloor None\rfloor$
  apply(*insert D*, *simp*)
  apply(*simp add*: *A Abs-Set-0-inject Abs-Set-0-inverse B C OclExcluding-def OclValid-def Option.set.simps(2) Rep-Set-0-inverse bot-Set-0-def bot-option-def null-Set-0-def null-option-def option.distinct(1)*)
done
have *H* : $(\delta\ (\lambda\text{-}.\ Abs\text{-}Set\text{-}0\ \lfloor\lfloor\lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil - \{y\ \tau\}\rfloor\rfloor))\ \tau = true\ \tau$
  apply(*auto simp*: *OclValid-def false-def true-def defined-def bot-fun-def bot-Set-0-def null-fun-def null-Set-0-def H1 H2*)
done

have $Z{:}insert\ (x\ \tau)\ \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil - \{y\ \tau\} = insert\ (x\ \tau)\ (\lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil - \{y\ \tau\})$
  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 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 *excluding-charn2*:
assumes *def-X*:$\tau \models (\delta\ X)$
and *val-x*:$\tau \models (\upsilon\ x)$
shows $\tau \models (((X{-}{>}\mathit{including}(x)){-}{>}\mathit{excluding}(x)) \triangleq (X{-}{>}\mathit{excluding}(x)))$
proof −
have $A : \bot \in \{X.\ X = \mathit{bot} \vee X = \mathit{null} \vee (\forall x \in \lceil\lceil X\rceil\rceil.\ x \neq \mathit{bot})\}$ **by**(*simp add*: *bot-option-def*)
have $B : \lfloor\bot\rfloor \in \{X.\ X = \mathit{bot} \vee X = \mathit{null} \vee (\forall x \in \lceil\lceil X\rceil\rceil.\ x \neq \mathit{bot})\}$ **by**(*simp add*: *null-option-def bot-option-def*)
have $C : \lfloor\lfloor \mathit{insert}\ (x\ \tau)\ \lceil\lceil \mathit{Rep\text{-}Set\text{-}0}\ (X\ \tau)\rceil\rceil\rfloor\rfloor \in \{X.\ X = \mathit{bot} \vee X = \mathit{null} \vee (\forall x \in \lceil\lceil X\rceil\rceil.\ x \neq \mathit{bot})\}$
apply(*insert def-X val-x*, *frule Set-inv-lemma*)
apply(*simp add*: *foundation18 invalid-def*)
done
have $G1 : \mathit{Abs\text{-}Set\text{-}0}\ \lfloor\lfloor \mathit{insert}\ (x\ \tau)\ \lceil\lceil \mathit{Rep\text{-}Set\text{-}0}\ (X\ \tau)\rceil\rceil\rfloor\rfloor \neq \mathit{Abs\text{-}Set\text{-}0}\ \mathit{None}$
apply(*insert C*, *simp*)
apply(*simp add*: *def-X val-x A Abs-Set-0-inject B C OclValid-def Rep-Set-0-cases Rep-Set-0-inverse bot-Set-0-def bot-option-def insert-compr insert-def not-Some-eq null-Set-0-def null-option-def*)
done
have $G2 : \mathit{Abs\text{-}Set\text{-}0}\ \lfloor\lfloor \mathit{insert}\ (x\ \tau)\ \lceil\lceil \mathit{Rep\text{-}Set\text{-}0}\ (X\ \tau)\rceil\rceil\rfloor\rfloor \neq \mathit{Abs\text{-}Set\text{-}0}\ \lfloor \mathit{None}\rfloor$
apply(*insert C*, *simp*)
apply(*simp add*: *def-X val-x A Abs-Set-0-inject B C OclValid-def Rep-Set-0-cases Rep-Set-0-inverse bot-Set-0-def bot-option-def insert-compr insert-def not-Some-eq null-Set-0-def null-option-def*)
done
show *?thesis*
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*) **back**
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 bot-Set-0-def split*: *bool.split-asm HOL.split-if-asm option.split*)
apply(*auto simp*: *G1 G2*)
done
qed

lemma *excluding-charn-exec*:
assumes *strict1*: $(x \doteq \mathit{invalid}) = \mathit{invalid}$
and *strict2*: $(\mathit{invalid} \doteq y) = \mathit{invalid}$
and *StrictRefEq-valid-args-valid*: $\bigwedge\ (x{::}('\mathfrak{A},'a{::}\mathit{null})\mathit{val})\ y\ \tau.$

$$(\tau \models \delta\ (x \doteq y)) = ((\tau \models (\upsilon\ x)) \wedge (\tau \models \upsilon\ y))$$

and *cp-StrictRefEq*: $\bigwedge (X{::}('\mathfrak{A},'a{::}null)val)\ Y\ \tau.\ (X \doteq Y)\ \tau = ((\lambda\text{-}.\ X\ \tau) \doteq (\lambda\text{-}.\ Y\ \tau))\ \tau$

and *StrictRefEq-vs-StrongEq*: $\bigwedge (x{::}('\mathfrak{A},'a{::}null)val)\ y\ \tau.$

$$\tau \models \upsilon\ x \Longrightarrow \tau \models \upsilon\ y \Longrightarrow (\tau \models ((x \doteq y) \triangleq (x \triangleq y)))$$

shows $(X{-}{>}including(x{::}('\mathfrak{A},'a{::}null)val){-}{>}excluding(y)) =$

$(if\ \delta\ X\ then\ if\ x \doteq y$
$then\ X{-}{>}excluding(y)$
$else\ X{-}{>}excluding(y){-}{>}including(x)$
$endif$
$else\ invalid\ endif)$

proof −

have *A1*: $\bigwedge\tau.\ \tau \models (X \triangleq invalid) \Longrightarrow$
$(X{-}{>}including(x){-}{>}includes(y))\ \tau = invalid\ \tau$
apply(*rule foundation22*[*THEN iffD1*])
by(*erule StrongEq-L-subst2-rev*, *simp*,*simp*)

have *B1*: $\bigwedge\tau.\ \tau \models (X \triangleq null) \Longrightarrow$
$(X{-}{>}including(x){-}{>}includes(y))\ \tau = invalid\ \ \tau$
apply(*rule foundation22*[*THEN iffD1*])
by(*erule StrongEq-L-subst2-rev*, *simp*,*simp*)

have *A2*: $\bigwedge\tau.\ \tau \models (X \triangleq invalid) \Longrightarrow X{-}{>}including(x){-}{>}excluding(y)\ \tau = invalid\ \tau$
apply(*rule foundation22*[*THEN iffD1*])
by(*erule StrongEq-L-subst2-rev*, *simp*,*simp*)

have *B2*: $\bigwedge\tau.\ \tau \models (X \triangleq null) \Longrightarrow X{-}{>}including(x){-}{>}excluding(y)\ \tau = invalid\ \tau$
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*: $\bigwedge\tau.\ \tau \models (x \triangleq invalid) \Longrightarrow$
$(X{-}{>}including(x){-}{>}excluding(y))\ \tau =$
$(if\ x \doteq y\ then\ X{-}{>}excluding(y)\ \ else\ X{-}{>}excluding(y){-}{>}including(x)\ \ endif)\ \tau$
apply(*rule foundation22*[*THEN iffD1*])
apply(*erule StrongEq-L-subst2-rev*,*simp*,*simp*)
by(*simp add*: *strict2*)

have *D*: $\bigwedge\tau.\ \tau \models (y \triangleq invalid) \Longrightarrow$
$(X{-}{>}including(x){-}{>}excluding(y))\ \tau =$
$(if\ x \doteq y\ then\ X{-}{>}excluding(y)\ \ else\ X{-}{>}excluding(y){-}{>}including(x)\ \ endif)\ \tau$
apply(*rule foundation22*[*THEN iffD1*])
apply(*erule StrongEq-L-subst2-rev*,*simp*,*simp*)
by (*simp add*: *strict1*)

have *E*: $\bigwedge\tau.\ \tau \models \upsilon\ x \Longrightarrow \tau \models \upsilon\ y \Longrightarrow$
$(if\ x \doteq y\ then\ X{-}{>}excluding(y)\ \ else\ X{-}{>}excluding(y){-}{>}including(x)\ \ endif)\ \tau =$
$(if\ x \triangleq y\ then\ X{-}{>}excluding(y)\ \ else\ X{-}{>}excluding(y){-}{>}including(x)\ \ endif)\ \tau$

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

have *F*: $\bigwedge \tau.\ \tau \models \delta\ X \Longrightarrow \tau \models \upsilon\ x \Longrightarrow \tau \models (x \triangleq y) \Longrightarrow$
$(X\text{->}including(x)\text{->}excluding(y)\ \tau) = (X\text{->}excluding(y)\ \tau)$
apply(*drule StrongEq-L-sym*)
apply(*rule foundation22*[*THEN iffD1*])
apply(*erule StrongEq-L-subst2-rev*,*simp*)
by(*simp add*: *excluding-charn2*)

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

schematic-lemma *excluding-charn-exec-int*[*simp*,*code-unfold*]: *?X*
by(*rule excluding-charn-exec*[*OF* $StrictRefEq_{Integer}$*-strict1* $StrictRefEq_{Integer}$*-strict2*
$StrictRefEq_{Integer}$*-defined-args-valid*
cp-$StrictRefEq_{Integer}$ $StrictRefEq_{Integer}$*-vs-StrongEq*], *simp-all*)

schematic-lemma *excluding-charn-exec-bool*[*simp*,*code-unfold*]: *?X*
by(*rule excluding-charn-exec*[*OF* $StrictRefEq_{Boolean}$*-strict1* $StrictRefEq_{Boolean}$*-strict2*
$StrictRefEq_{Boolean}$*-defined-args-valid*
cp-$StrictRefEq_{Boolean}$ $StrictRefEq_{Boolean}$*-vs-StrongEq*], *simp-all*)

schematic-lemma *excluding-charn-exec-set*[*simp*,*code-unfold*]: *?X*
by(*rule excluding-charn-exec*[*OF* $StrictRefEq_{Set}$*-strict1* $StrictRefEq_{Set}$*-strict2*
$StrictRefEq_{Set}$*-strictEq-valid-args-valid*
cp-$StrictRefEq_{Set}$ $StrictRefEq_{Set}$*-vs-StrongEq*], *simp-all*)

lemma *finite-excluding-rep-set* :
assumes *X-def* : $\tau \models \delta\ X$
and *x-val* : $\tau \models \upsilon\ x$

shows *finite* $\lceil\lceil$*Rep-Set-0* $(X{-}{>}excluding(x)\ \tau)\rceil\rceil$ = *finite* $\lceil\lceil$*Rep-Set-0* $(X\ \tau)\rceil\rceil$
proof −
have C : $\lfloor\lfloor\lceil\lceil$*Rep-Set-0* $(X\ \tau)\rceil\rceil - \{x\ \tau\}\rfloor\rfloor \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
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

lemma *excluding-rep-set*:
assumes *S-def*: $\tau \models \delta\ S$
shows $\lceil\lceil$*Rep-Set-0* $(S{-}{>}excluding(\lambda\text{-}.\ \lfloor\lfloor x\rfloor\rfloor)\ \tau)\rceil\rceil = \lceil\lceil$*Rep-Set-0* $(S\ \tau)\rceil\rceil - \{\lfloor\lfloor x\rfloor\rfloor\}$
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. OclSize

lemma *OclSize-infinite*:
assumes *non-finite*:$\tau \models not(\delta(S{-}{>}size()))$
shows $(\tau \models not(\delta(S))) \vee \neg$ *finite* $\lceil\lceil$*Rep-Set-0* $(S\ \tau)\rceil\rceil$
apply(*insert non-finite*, *simp*)
apply(*rule impI*)
apply(*simp add*: *OclSize-def OclValid-def defined-def*)
apply(*case-tac finite* $\lceil\lceil$*Rep-Set-0* $(S\ \tau)\rceil\rceil$,
simp-all add:*null-fun-def null-option-def bot-fun-def bot-option-def*)
done

lemma [*simp*,*code-unfold*]: $Set\{\}\ {-}{>}size() = \mathbf{0}$
proof −
have $A1$: $\lfloor\lfloor\{\}\rfloor\rfloor \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$ **by**(*simp add*: *mtSet-def*)
have $A2$: $None \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$ **by**(*simp add*: *bot-option-def*)
have $A3$: $\lfloor None\rfloor \in \{X.\ X = bot \vee X = null \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$ **by**(*simp add*: *bot-option-def null-option-def*)
show *?thesis*
apply(*rule ext*)
apply(*simp add*: *defined-def mtSet-def OclSize-def*
bot-Set-0-def bot-fun-def
null-Set-0-def null-fun-def)
apply(*subst Abs-Set-0-inject*, *simp-all add*: *A1 A2 A3 bot-option-def null-option-def*) +
by(*simp add*: *A1 Abs-Set-0-inverse bot-fun-def bot-option-def null-fun-def null-option-def OclInt0-def*)
qed

lemma [*simp*,*code-unfold*]: δ (*Set*{} −>*size*()) = *true*
by *simp*

lemma *including-size-defined*[*simp*,*code-unfold*]: δ ((*X* −>*including*(*x*)) −>*size*()) = (δ(*X*−>*size*()) *and* υ(*x*))
proof −

have *defined-inject-true* : $\bigwedge\tau$ *P*. (δ *P*) $\tau \neq$ *true* $\tau \Longrightarrow$ (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* $\tau = \perp \vee$ *P* τ = *null*, *simp-all add*: *true-def*)

have *valid-inject-true* : $\bigwedge\tau$ *P*. (υ *P*) $\tau \neq$ *true* $\tau \Longrightarrow$ (υ *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* $\tau = \perp$, *simp-all add*: *true-def*)

have *finite-including-rep-set* : $\bigwedge\tau$. (δ *X and* υ *x*) τ = *true* $\tau \Longrightarrow$
finite ⌈⌈*Rep-Set-0* (*X*−>*including*(*x*) τ)⌉⌉ = *finite* ⌈⌈*Rep-Set-0* (*X* τ)⌉⌉
apply(*rule finite-including-rep-set*)
apply(*metis OclValid-def foundation5*)+
done

have *card-including-exec* : $\bigwedge\tau$. (δ (λ-. ⌊⌊*int* (*card* ⌈⌈*Rep-Set-0* (*X*−>*including*(*x*) τ)⌉⌉)⌋⌋)) τ = (δ (λ-. ⌊⌊*int* (*card* ⌈⌈*Rep-Set-0* (*X* τ)⌉⌉)⌋⌋)) τ
apply(*simp add*: *defined-def bot-fun-def bot-option-def null-fun-def null-option-def*)
done

show *?thesis*

apply(*rule ext*, *rename-tac* τ)
apply(*case-tac* (δ (*X*−>*including*(*x*)−>*size*())) τ = *true* τ, *simp*)
apply(*subst cp-OclAnd*)
apply(*subst cp-defined*)
apply(*simp only*: *cp-defined*[*of X*−>*including*(*x*)−>*size*()])
apply(*simp add*: *OclSize-def*)
apply(*case-tac* ((δ *X and* υ *x*) τ = *true* $\tau \wedge$ *finite* ⌈⌈*Rep-Set-0* (*X*−>*including*(*x*) τ)⌉⌉), *simp*)
prefer *2*
apply(*simp*)
apply(*simp add*: *defined-def true-def false-def bot-fun-def bot-option-def*)
apply(*erule conjE*)
apply(*simp add*: *finite-including-rep-set*[*simplified OclValid-def*] *card-including-exec*
cp-OclAnd[*of* δ *X* υ *x*]
cp-OclAnd[*of true*, *THEN sym*])
apply(*subgoal-tac* (δ *X*) τ = *true* $\tau \wedge$ (υ *x*) τ = *true* τ, *simp*)
apply(*rule foundation5*[*of* - δ *X* υ *x*, *simplified OclValid-def*], *simp only*: *cp-OclAnd*[*THEN sym*])

apply(*drule defined-inject-true*[*of* X−>*including*(x)−>*size*()], *simp*)
apply(*simp only*: *cp-OclAnd*[*of* δ (X−>*size*()) υ x])
apply(*simp add*: *cp-defined*[*of* X−>*including*(x)−>*size*()] *cp-defined*[*of* X−>*size*()])
apply(*simp add*: *OclSize-def card-including-exec*)
apply(*case-tac* (δ X *and* υ x) τ = *true* τ $\wedge$ *finite* $\lceil\lceil$*Rep-Set-0* (X τ)$\rceil\rceil$,
simp add: *finite-including-rep-set*[*simplified OclValid-def*] *card-including-exec*)
apply(*simp only*: *cp-OclAnd*[*THEN sym*])
apply(*simp add*: *defined-def bot-fun-def*)

apply(*split split-if-asm*)
apply(*simp add*: *finite-including-rep-set*[*simplified OclValid-def*])
apply(*simp add*: *finite-including-rep-set*[*simplified OclValid-def*] *card-including-exec*)
apply(*simp only*: *cp-OclAnd*[*THEN sym*])
apply(*simp*)
apply(*rule impI*)
apply(*erule conjE*)
apply(*case-tac* (υ x) τ = *true* τ, *simp add*: *cp-OclAnd*[*of* δ X υ x])
apply(*drule valid-inject-true*[*of* x], *simp add*: *cp-OclAnd*[*of* - υ x])
done
qed

lemma *including-size-exec*[*code-unfold*]:
((X −>*including*(x)) −>*size*()) = (*if* δ X *and* υ x *then*
X −>*size*() $+_{ocl}$ *if* X −>*includes*(x) *then* **0** *else* **1** *endif*
else
invalid
endif)
proof −

have *valid-inject-true* : $\bigwedge\tau$ P. (υ P) τ $\neq$ *true* τ $\Longrightarrow$ (υ 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 = $\bot$, *simp-all add*: *true-def*)
have *defined-inject-true* : $\bigwedge\tau$ P. (δ P) τ $\neq$ *true* τ $\Longrightarrow$ (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 = $\bot$ $\vee$ P τ = *null*, *simp-all add*: *true-def*)

show *?thesis*
apply(*rule ext*, *rename-tac* τ)
proof −
fix τ
have *includes-notin*: $\neg$ τ $\models$ X−>*includes*(x) $\Longrightarrow$ (δ X) τ = *true* τ $\wedge$ (υ x) τ = *true* τ $\Longrightarrow$
x τ $\notin$ $\lceil\lceil$*Rep-Set-0* (X τ)$\rceil\rceil$
by(*simp add*: *OclIncludes-def OclValid-def true-def*)

have *includes-def*: τ $\models$ X−>*includes*(x) $\Longrightarrow$ τ $\models$ δ X
by (*metis OCL-core.bot-fun-def OclIncludes-def OclValid-def defined3 foundation16*)

have *includes-val*: $\tau \models X{-}{>}includes(x) \Longrightarrow \tau \models \upsilon\ x$
by (*metis* (*hide-lams*, *no-types*) *foundation6 includes-valid-args-valid′ including-valid-args-valid including-valid-args-valid′′*)

show $X{-}{>}including(x){-}{>}size()\ \tau = (if\ \delta\ X\ and\ \upsilon\ x\ then\ X{-}{>}size() +_I\ if\ X{-}{>}includes(x)\ then\ \mathbf{0}\ else\ \mathbf{1}\ endif\ else\ invalid\ endif)\ \tau$
apply(*case-tac* $\tau \models \delta\ X\ and\ \upsilon\ x$)
apply(*simp*)
apply(*subst* $cp\text{-}OclAdd_{Integer}$)
apply(*case-tac* $\tau \models X{-}{>}includes(x)$, *simp*)

apply(*simp add*: $cp\text{-}OclAdd_{Integer}$[*symmetric*])
apply(*case-tac* $\tau \models ((\upsilon\ (X{-}{>}size()))\ and\ not\ (\delta\ (X{-}{>}size())))$, *simp*)
apply(*drule foundation5*[**where** $P = \upsilon\ X{-}{>}size()$], *erule conjE*)
apply(*drule OclSize-infinite*)
apply(*frule includes-def*, *drule includes-val*)
apply(*simp*)
apply(*subst OclSize-def*, *subst finite-including-rep-set*, *assumption*, *assumption*)
apply (*metis* (*hide-lams*, *no-types*) *invalid-def*)

apply(*subst OclIf-false′*)
apply (*metis* (*hide-lams*, *no-types*) *defined5 defined6 defined-and-I defined-not-I foundation1 foundation9*)
apply(*subst cp-OclSize*)
apply(*simp add*: *including-includes-simp cp-OclSize*[*symmetric*])

apply(*subst OclIf-false′*, *subst foundation9*)
apply (*metis* (*hide-lams*, *no-types*) *includes-valid-args-valid′*, *simp*)
apply(*simp add*: *OclSize-def*)
apply(*subst* (*1 2*) *finite-including-rep-set*)
apply (*metis OclValid-def foundation5*)
apply (*metis OclValid-def foundation5*)
apply(*subst* (*1 2*) *cp-OclAnd*, *subst* (*1 2*) $cp\text{-}OclAdd_{Integer}$)
apply(*simp*)
apply(*rule conjI*)
apply(*simp add*: *OclIncluding-def*)
apply(*subst Abs-Set-0-inverse*, *simp add*: *bot-option-def null-option-def*)
apply (*metis* (*hide-lams*, *no-types*) *Set-inv-lemma foundation18′ foundation5*)
apply(*drule foundation5*)
apply(*subst* (*asm*) (*2 3*) *OclValid-def*)
apply(*simp add*: $OclAdd_{Integer}$*-def OclInt1-def*)
apply(*rule impI*)
apply(*drule Finite-Set.card.insert*[**where** $x = x\ \tau$])
apply(*rule includes-notin*, *simp*)
apply(*simp*)
apply (*metis Suc-eq-plus1 int-1 of-nat-add*)

apply(*subst* (*1 2*) $OclAdd_{Integer}$*-strict2*[*simplified invalid-def*], *simp*)
apply(*subst finite-including-rep-set*)
apply (*metis OclValid-def foundation5*)
apply (*metis OclValid-def foundation5*)
apply (*metis OclValid-def foundation5*)

apply(*subst OclIf-false′*)
apply (*metis* (*hide-lams*, *no-types*) *defined6 excluding-valid-args-valid″ foundation1 foundation9*)
by (*metis cp-OclSize foundation18′ including-valid-args-valid″ invalid-def size-strict1*)
qed
qed

lemma *excluding-size-defined*[*simp*,*code-unfold*]: δ ((X −>*excluding*(x)) −>*size*()) = (δ(X−>*size*()) *and* υ(x))
proof −

have *defined-inject-true* : $\bigwedge\tau$ P. (δ P) $\tau \neq$ *true* $\tau \Longrightarrow$ (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 $\tau = \bot \vee P$ τ = *null*, *simp-all add*: *true-def*)

have *valid-inject-true* : $\bigwedge\tau$ P. (υ P) $\tau \neq$ *true* $\tau \Longrightarrow$ (υ 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 $\tau = \bot$, *simp-all add*: *true-def*)

have *finite-excluding-rep-set* : $\bigwedge\tau$. (δ X *and* υ x) τ = *true* $\tau \Longrightarrow$
finite ⌈⌈*Rep-Set-0* (X−>*excluding*(x) τ)⌉⌉ =
finite ⌈⌈*Rep-Set-0* (X τ)⌉⌉
apply(*rule finite-excluding-rep-set*)
apply(*metis OclValid-def foundation5*)+
done

have *card-excluding-exec* : $\bigwedge\tau$. (δ (λ-. ⌊⌊*int* (*card* ⌈⌈*Rep-Set-0* (X−>*excluding*(x) τ)⌉⌉)⌋⌋)) τ =
(δ (λ-. ⌊⌊*int* (*card* ⌈⌈*Rep-Set-0* (X τ)⌉⌉)⌋⌋)) τ
apply(*simp add*: *defined-def bot-fun-def bot-option-def null-fun-def null-option-def*)
done

show *?thesis*

apply(*rule ext*, *rename-tac* τ)
apply(*case-tac* (δ (X−>*excluding*(x)−>*size*())) τ = *true* τ, *simp*)
apply(*subst cp-OclAnd*)
apply(*subst cp-defined*)
apply(*simp only*: *cp-defined*[*of* X−>*excluding*(x)−>*size*()])
apply(*simp add*: *OclSize-def*)

apply(*case-tac* ((δ *X and* υ *x*) τ = *true* τ ∧ *finite* ⌈⌈*Rep-Set-0* (*X*−>*excluding*(*x*) τ)⌉⌉), *simp*)
prefer *2*
apply(*simp*)
apply(*simp add*: *defined-def true-def false-def bot-fun-def bot-option-def*)
apply(*erule conjE*)
apply(*simp add*: *finite-excluding-rep-set card-excluding-exec*
cp-OclAnd[*of* δ *X* υ *x*]
cp-OclAnd[*of true*, *THEN sym*])
apply(*subgoal-tac* (δ *X*) τ = *true* τ ∧ (υ *x*) τ = *true* τ, *simp*)
apply(*rule foundation5*[*of* - δ *X* υ *x*, *simplified OclValid-def*], *simp only*: *cp-OclAnd*[*THEN sym*])

apply(*drule defined-inject-true*[*of X*−>*excluding*(*x*)−>*size*()], *simp*)
apply(*simp only*: *cp-OclAnd*[*of* δ (*X*−>*size*()) υ *x*])
apply(*simp add*: *cp-defined*[*of X*−>*excluding*(*x*)−>*size*()] *cp-defined*[*of X*−>*size*()])
apply(*simp add*: *OclSize-def finite-excluding-rep-set card-excluding-exec*)
apply(*case-tac* (δ *X and* υ *x*) τ = *true* τ ∧ *finite* ⌈⌈*Rep-Set-0* (*X* τ)⌉⌉,
simp add: *finite-excluding-rep-set card-excluding-exec*)
apply(*simp only*: *cp-OclAnd*[*THEN sym*])
apply(*simp add*: *defined-def bot-fun-def*)

apply(*split split-if-asm*)
apply(*simp add*: *finite-excluding-rep-set*)
apply(*simp add*: *finite-excluding-rep-set card-excluding-exec*)
apply(*simp only*: *cp-OclAnd*[*THEN sym*])
apply(*simp*)
apply(*rule impI*)
apply(*erule conjE*)
apply(*case-tac* (υ *x*) τ = *true* τ, *simp add*: *cp-OclAnd*[*of* δ *X* υ *x*])
apply(*drule valid-inject-true*[*of x*], *simp add*: *cp-OclAnd*[*of* - υ *x*])
done
qed

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

lemma [*simp*]:

assumes *X-finite*: $\bigwedge\tau$. *finite* $\lceil\lceil$*Rep-Set-0* $(X\ \tau)\rceil\rceil$
shows δ $((X \ {-}{>}including(x))\ {-}{>}size()) = (\delta(X)\ and\ \upsilon(x))$
by(*simp add*: *size-defined*[*OF X-finite*])

4.6.4. OclIsEmpty

lemma [*simp*,*code-unfold*]: *Set*{}$-$>*isEmpty*() = *true*
by(*simp add*: *OclIsEmpty-def*)

lemma *including-not-isempty* [*simp*]:
assumes *X-def*: $\tau \models \delta\ X$
and *X-finite*: *finite* $\lceil\lceil$*Rep-Set-0* $(X\ \tau)\rceil\rceil$
and *a-val*: $\tau \models \upsilon\ a$
shows $X{-}{>}including(a){-}{>}isEmpty()\ \tau = false\ \tau$
proof $-$
have *A1* : $\bigwedge\tau\ X.\ X\ \tau = true\ \tau \vee X\ \tau = false\ \tau \Longrightarrow (X\ and\ not\ X)\ \tau = false\ \tau$
by (*metis* (*no-types*) *OclAnd-false1 OclAnd-idem OclImplies-def OclNot3 OclNot-not OclOr-false1 cp-OclAnd cp-OclNot deMorgan1 deMorgan2*)

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

have *B* : $\bigwedge X\ \tau.\ \tau \models \upsilon\ X \Longrightarrow X\ \tau \neq \mathbf{0}\ \tau \Longrightarrow (X \doteq \mathbf{0})\ \tau = false\ \tau$
by (*metis OclAnd-true2 OclValid-def Sem-def* $StrictRefEq_{Integer}$ $StrictRefEq_{Integer}$*-strict*$'$ $StrictRefEq_{Integer}$*-strict*$''$ *StrongEq-sym bool-split foundation16 foundation22 invalid-def null-fun-def null-non-OclInt0 valid4*)

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

4.6.5. OclNotEmpty

lemma [*simp*,*code-unfold*]: *Set*{}$-$>*notEmpty*() = *false*
by(*simp add*: *OclNotEmpty-def*)

lemma *including-notempty-true* [*simp*,*code-unfold*]:
assumes *X-def*: $\tau \models \delta\ X$

and *X-finite*: *finite* $\lceil\lceil$*Rep-Set-0* (*X* τ)$\rceil\rceil$
and *a-val*: $\tau \models \upsilon$ *a*
shows *X*−>*including*(*a*)−>*notEmpty*() τ = *true* τ
apply(*simp add*: *OclNotEmpty-def*)
apply(*subst cp-OclNot*, *subst including-not-isempty*, *simp-all add*: *assms*)
by (*metis OclNot4 cp-OclNot*)

4.6.6. Ocl Any

lemma [*simp*,*code-unfold*]: *Set*{}−>*any*() = *null*
apply(*rule ext*, *simp add*: *Ocl-Any-def*)
apply(*rule impI*)
apply(*simp add*: *false-def true-def*)
done

lemma *any-exec*[*simp*,*code-unfold*]:
(*Set*{}−>*including*(*a*))−>*any*() = *a*
apply(*rule ext*, *rename-tac* τ, *simp add*: *mtSet-def Ocl-Any-def*)
apply(*case-tac* $\tau \models \upsilon$ *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*) *including-notempty-true*[**where** *X* = *Set*{}, *simplified mtSet-def*])
apply(*simp add*: *mtSet-defined*[*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(*subst* (*1 2*) *Abs-Set-0-inverse*, *simp add*: *bot-option-def null-option-def*)
apply(*simp*, *metis OclValid-def foundation18'*)
apply(*simp*)
apply(*simp add*: *mtSet-defined*[*simplified mtSet-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*)
apply(*subst* (*1 2 3 4*) *cp-OclAnd*, *simp add*: *mtSet-defined*[*simplified mtSet-def*] *valid-def bot-fun-def*)
apply(*simp add*: *cp-OclAnd*[*symmetric*], *rule impI*, *simp add*: *false-def true-def*)
done

lemma *any-exec-unfold*[*simp*,*code-unfold*]:
X−>*includes*(*X*−>*any*()) = (*if* δ *X then*
if δ (*X*−>*size*()) *then not*(*X*−>*isEmpty*())
else X−>*includes*(*null*) *endif*
else invalid endif)
proof −
have *defined-inject-true* : $\bigwedge\tau\ P.\ (\delta\ P)\ \tau \neq true\ \tau \Longrightarrow (\delta\ P)\ \tau = false\ \tau$

apply(*simp add*: *defined-def true-def false-def bot-fun-def bot-option-def*
null-fun-def null-option-def)
by (*case-tac* $P\ \tau = \bot \vee P\ \tau = null$, *simp-all add*: *true-def*)

have *valid-inject-true* : $\bigwedge \tau\ P.\ (\upsilon\ P)\ \tau \neq true\ \tau \Longrightarrow (\upsilon\ P)\ \tau = false\ \tau$
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'*: $\bigwedge \tau\ X.\ \tau \models \delta\ X \Longrightarrow finite\ \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil \Longrightarrow not\ (X{-}{>}isEmpty())\ \tau \neq true\ \tau \Longrightarrow X\ \tau = Set\{\}\ \tau$
apply(*case-tac* $X\ \tau$, *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*
null-Set-0-def null-option-def foundation17)
apply(*case-tac y*, *simp*, *metis* (*hide-lams*, *no-types*) *bot-Set-0-def foundation17*)
apply(*case-tac a*, *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-StrictRefEq*$_{Integer}$, *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*$_{Integer}$ *StrongEq-def*)

have *B*: $\bigwedge X\ \tau.\ \neg\ finite\ \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil \Longrightarrow (\delta\ (X{-}{>}size()))\ \tau = false\ \tau$
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* τ, *simp only*: *OclIncludes-def Ocl-Any-def*)
apply(*subst cp-OclIf*, *subst* (*2*) *cp-valid*)
apply(*case-tac* $(\delta\ X)\ \tau = true\ \tau$, *simp only*: *foundation20*[*simplified OclValid-def*] *cp-OclIf*[*symmetric*], *simp*,
subst (*1 2*) *cp-OclAnd*, *simp add*: *cp-OclAnd*[*symmetric*])
apply(*case-tac finite* $\lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil$)
apply(*frule size-defined'*[*THEN iffD2*, *simplified OclValid-def*], *assumption*)
apply(*subst* (*1 2 3 4*) *cp-OclIf*) **apply**(*simp*)
apply(*subst* (*1 2 3 4*) *cp-OclIf*[*symmetric*], *simp*)
apply(*case-tac* $(X{-}{>}notEmpty())\ \tau = true\ \tau$, *simp*)
apply(*frule notempty-has-elt*[*simplified OclValid-def*], *simp*)
apply(*simp add*: *OclNotEmpty-def cp-OclIf*[*symmetric*])
apply(*subgoal-tac* $(SOME\ y.\ y \in \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil) \in \lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil$, *simp add*: *true-def*)
apply(*metis OclValid-def Set-inv-lemma foundation18' null-option-def true-def*)
apply(*rule someI-ex*, *simp*)
apply(*simp add*: *OclNotEmpty-def cp-valid*[*symmetric*])

apply(*subgoal-tac* $\neg$ (*null* $\tau \in \lceil\lceil$*Rep-Set-0* (*X* τ)$\rceil\rceil$), *simp*)
apply(*subst OclIsEmpty-def*, *simp add*: *OclSize-def*)
apply(*subst cp-OclNot*, *subst cp-OclOr*, *subst cp-StrictRefEq*$_{Integer}$, *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*)
apply(*simp*)
apply(*subst* (*1 2 3 4*) *cp-OclIf*[*symmetric*], *simp*)
apply(*case-tac* (*X*−>*notEmpty*()) τ = *true* τ, *simp*)
apply(*frule notempty-has-elt*[*simplified OclValid-def*], *simp*)
apply(*simp add*: *OclNotEmpty-def OclIsEmpty-def*)
apply(*subgoal-tac X*−>*size*() $\tau = \perp$)
prefer *2*
apply (*metis* (*hide-lams*, *no-types*) *OclSize-def*)
apply(*subst* (*asm*) *cp-OclNot*, *subst* (*asm*) *cp-OclOr*, *subst* (*asm*) *cp-StrictRefEq*$_{Integer}$, *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$_{Integer}$ *valid-def bot-option-def bot-fun-def false-def true-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

4.6.7. OclForall

lemma *forall-set-null-exec*[*simp*,*code-unfold*] :
(*null*−>*forAll*(*z*| *P*(*z*))) = *invalid*
by(*simp add*: *OclForall-def invalid-def false-def true-def*)

lemma *forall-set-mt-exec*[*simp*,*code-unfold*] :
((*Set*{})−>*forAll*(*z*| *P*(*z*))) = *true*
apply(*simp add*: *OclForall-def*)
apply(*subst mtSet-def*)+
apply(*subst Abs-Set-0-inverse*, *simp-all add*: *true-def*)+
done

lemma *forall-set-including-exec*[*simp*,*code-unfold*] :
assumes *cp0* : *cp P*

shows $((S{-}{>}including(x)){-}{>}forAll(z \mid P(z))) = (if\ \delta\ S\ and\ \upsilon\ x$
$then\ P\ x\ and\ (S{-}{>}forAll(z \mid P(z)))$
$else\ invalid$
$endif)$

proof −
have cp: $\bigwedge\tau.\ P\ x\ \tau = P\ (\lambda\text{-}.\ x\ \tau)\ \tau$
by($insert\ cp0$, $auto\ simp$: $cp\text{-}def$)

have $insert\text{-}in\text{-}Set\text{-}0$: $\bigwedge\tau.\ (\tau \models (\delta\ S)) \Longrightarrow (\tau \models (\upsilon\ x)) \Longrightarrow \lfloor\lfloor insert\ (x\ \tau)\ \lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil\rfloor\rfloor$
$\in \{X.\ X = bot \lor X = null \lor (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
apply($frule\ Set\text{-}inv\text{-}lemma$)
apply($simp\ add$: $foundation18\ invalid\text{-}def$)
done

have $d\text{-}and\text{-}v\text{-}destruct\text{-}defined$: $\bigwedge\tau\ S\ x.\ \tau \models (\delta\ S\ and\ \upsilon\ x) \Longrightarrow \tau \models \delta\ S$
by ($simp\ add$: $foundation5[THEN\ conjunct1]$)
have $d\text{-}and\text{-}v\text{-}destruct\text{-}valid$:$\bigwedge\tau\ S\ x.\ \tau \models (\delta\ S\ and\ \upsilon\ x) \Longrightarrow \tau \models \upsilon\ x$
by ($simp\ add$: $foundation5[THEN\ conjunct2]$)

have $forall\text{-}including\text{-}invert$: $\bigwedge\tau\ f.\ (f\ x\ \tau = f\ (\lambda\ \text{-}.\ x\ \tau)\ \tau) \Longrightarrow$
$\tau \models (\delta\ S\ and\ \upsilon\ x) \Longrightarrow$
$(\forall x{\in}\lceil\lceil Rep\text{-}Set\text{-}0\ (S{-}{>}including(x)\ \tau)\rceil\rceil.\ f\ (\lambda\text{-}.\ x)\ \tau) =$
$(f\ x\ \tau \land (\forall x{\in}\lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil.\ f\ (\lambda\text{-}.\ x)\ \tau))$
apply($simp\ add$: $OclIncluding\text{-}def$)
apply($subst\ Abs\text{-}Set\text{-}0\text{-}inverse$)
apply($rule\ insert\text{-}in\text{-}Set\text{-}0$)
apply($rule\ d\text{-}and\text{-}v\text{-}destruct\text{-}defined$, $assumption$)
apply($rule\ d\text{-}and\text{-}v\text{-}destruct\text{-}valid$, $assumption$)
apply($simp\ add$: $d\text{-}and\text{-}v\text{-}destruct\text{-}defined\ d\text{-}and\text{-}v\text{-}destruct\text{-}valid$)
apply($frule\ d\text{-}and\text{-}v\text{-}destruct\text{-}defined$, $drule\ d\text{-}and\text{-}v\text{-}destruct\text{-}valid$)
apply($simp\ add$: $OclValid\text{-}def$)
done

have $exists\text{-}including\text{-}invert$: $\bigwedge\tau\ f.\ (f\ x\ \tau = f\ (\lambda\ \text{-}.\ x\ \tau)\ \tau) \Longrightarrow$
$\tau \models (\delta\ S\ and\ \upsilon\ x) \Longrightarrow$
$(\exists x{\in}\lceil\lceil Rep\text{-}Set\text{-}0\ (S{-}{>}including(x)\ \tau)\rceil\rceil.\ f\ (\lambda\text{-}.\ x)\ \tau) =$
$(f\ x\ \tau \lor (\exists x{\in}\lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil.\ f\ (\lambda\text{-}.\ x)\ \tau))$
apply($subst\ arg\text{-}cong$[**where** $f = \lambda x.\ \neg x$,
$OF\ forall\text{-}including\text{-}invert$[**where** $f = \lambda x\ \tau.\ \neg\ (f\ x\ \tau)$],
$simplified$])
by $simp\text{-}all$

have $cp\text{-}eq$: $\bigwedge\tau\ v.\ (P\ x\ \tau = v) = (P\ (\lambda\text{-}.\ x\ \tau)\ \tau = v)$ **by**($subst\ cp$, $simp$)
have $cp\text{-}OclNot\text{-}eq$: $\bigwedge\tau\ v.\ (P\ x\ \tau \neq v) = (P\ (\lambda\text{-}.\ x\ \tau)\ \tau \neq v)$ **by**($subst\ cp$, $simp$)

have $foundation10'$: $\bigwedge\tau\ x\ y.\ (\tau \models x) \land (\tau \models y) \Longrightarrow \tau \models (x\ and\ y)$
apply($erule\ conjE$)
apply($subst\ foundation10$)
apply($rule\ foundation6$, $simp$)

apply(*rule foundation6*, *simp*)
by *simp*

have *contradict-Rep-Set-0*: $\bigwedge\tau$ *S f*.
$\exists\, x\in\lceil\lceil$*Rep-Set-0 S*$\rceil\rceil$. *f* ($\lambda$-. *x*) $\tau \Longrightarrow$
($\forall\, x\in\lceil\lceil$*Rep-Set-0 S*$\rceil\rceil$. $\neg$ (*f* (λ-. *x*) τ)) = *False*
by(*case-tac* ($\forall\, x\in\lceil\lceil$*Rep-Set-0 S*$\rceil\rceil$. $\neg$ (*f* (λ-. *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* υ *x*])
apply(*simp add*: *cp-defined*[*THEN sym*])
apply(*rule conjI*, *rule impI*)

apply(*subgoal-tac* $\tau \models \delta$ *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** $x = \delta$ *S* **and** $y = \upsilon$ *x*, *simplified OclValid-def*])

apply(*subgoal-tac* $\tau \models$ (δ *S and* υ *x*))
prefer *2*
apply(*simp add*: *OclValid-def*)

apply(*case-tac* $\exists\, x\in\lceil\lceil$*Rep-Set-0* (*S*−>*including*(*x*) τ)$\rceil\rceil$. *P* (λ-. *x*) τ = *false* τ, *simp-all*)
apply(*subst contradict-Rep-Set-0*[**where** *f* = 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(*simp add*: *forall-including-invert*[**where** *f* = λ *x* τ. *P x* $\tau \neq$ *false* τ, *OF cp-OclNot-eq*],
erule conjE)

apply(*case-tac* $\exists\, x\in\lceil\lceil$*Rep-Set-0* (*S*−>*including*(*x*) τ)$\rceil\rceil$. *P* (λ-. *x*) τ = *bot* τ, *simp-all*)

apply(*subst contradict-Rep-Set-0*[**where** $f = \lambda\ x\ \tau.\ P\ x\ \tau = bot\ \tau$], *simp*)+
apply(*simp add*: *exists-including-invert*[**where** $f = \lambda\ x\ \tau.\ P\ x\ \tau = bot\ \tau$, *OF cp-eq*])

apply(*simp add*: *cp-OclAnd*[*of P x*])
apply(*erule disjE*)

apply(*subgoal-tac OclForall S P* $\tau \neq$ *false* τ)
apply(*simp only*: *cp-OclAnd*[*symmetric*], *simp*)
apply(*simp add*: *OclForall-def null-fun-def null-option-def bot-fun-def bot-option-def true-def false-def*)

apply(*subgoal-tac OclForall S P* $\tau =$ *bot* τ)
apply(*simp only*: *cp-OclAnd*[*symmetric*], *simp*)
apply(*simp add*: *OclForall-def null-fun-def null-option-def bot-fun-def bot-option-def true-def false-def*)

apply(*simp add*: *forall-including-invert*[**where** $f = \lambda\ x\ \tau.\ P\ x\ \tau \neq bot\ \tau$, *OF cp-OclNot-eq*],
erule conjE)

apply(*case-tac* $\exists x \in \lceil\lceil$*Rep-Set-0* (*S*−>*including*(*x*) τ)$\rceil\rceil$. *P* (λ-. *x*) $\tau =$ *null* τ, *simp-all*)
apply(*subst contradict-Rep-Set-0*[**where** $f = \lambda\ x\ \tau.\ P\ x\ \tau = null\ \tau$], *simp*)+
apply(*simp add*: *exists-including-invert*[**where** $f = \lambda\ x\ \tau.\ P\ x\ \tau = null\ \tau$, *OF cp-eq*])

apply(*simp add*: *cp-OclAnd*[*of P x*])
apply(*erule disjE*)

apply(*subgoal-tac OclForall S P* $\tau \neq$ *false* $\tau \wedge$ *OclForall S P* $\tau \neq$ *bot* τ)
apply(*simp only*: *cp-OclAnd*[*symmetric*], *simp*)
apply(*simp add*: *OclForall-def null-fun-def null-option-def bot-fun-def bot-option-def true-def false-def*)

apply(*subgoal-tac OclForall S P* $\tau =$ *null* τ)
apply(*simp only*: *cp-OclAnd*[*symmetric*], *simp*)
apply(*simp add*: *OclForall-def null-fun-def null-option-def bot-fun-def bot-option-def true-def false-def*)

apply(*simp add*: *forall-including-invert*[**where** $f = \lambda\ x\ \tau.\ P\ x\ \tau \neq null\ \tau$, *OF cp-OclNot-eq*],
erule conjE)

apply(*simp add*: *cp-OclAnd*[*of P x*] *OclForall-def*)
apply(*subgoal-tac P x* $\tau =$ *true* τ, *simp*)
apply(*metis bot-fun-def bool-split foundation18' foundation2 valid1*)

by(*metis OclForall-def including-defined-args-valid′ invalid-def*)
qed

lemma *forall-includes* :
assumes *x-def* : $\tau \models \delta\ x$
and *y-def* : $\tau \models \delta\ y$
shows $(\tau \models \mathit{OclForall}\ x\ (\mathit{OclIncludes}\ y)) = (\lceil\lceil \mathit{Rep\text{-}Set\text{-}0}\ (x\ \tau)\rceil\rceil \subseteq \lceil\lceil \mathit{Rep\text{-}Set\text{-}0}\ (y\ \tau)\rceil\rceil)$
proof −
have *discr-eq-false-true* : $\bigwedge\tau.\ (\mathit{false}\ \tau = \mathit{true}\ \tau) = \mathit{False}$ **by** (*metis OclValid-def foundation2*)
have *discr-eq-bot1-true* : $\bigwedge\tau.\ (\bot\ \tau = \mathit{true}\ \tau) = \mathit{False}$ **by** (*metis defined3 defined-def discr-eq-false-true*)
have *discr-eq-bot2-true* : $\bigwedge\tau.\ (\bot = \mathit{true}\ \tau) = \mathit{False}$ **by** (*metis bot-fun-def discr-eq-bot1-true*)
have *discr-eq-null-true* : $\bigwedge\tau.\ (\mathit{null}\ \tau = \mathit{true}\ \tau) = \mathit{False}$ **by** (*metis OclValid-def foundation4*)
show *?thesis*
apply(*case-tac* $\tau \models \mathit{OclForall}\ x\ (\mathit{OclIncludes}\ y)$)

apply(*simp add*: *OclValid-def OclForall-def*)
apply(*split split-if-asm*, *simp-all add*: *discr-eq-false-true discr-eq-bot1-true discr-eq-null-true discr-eq-bot2-true*)+
apply(*subgoal-tac* $\forall x \in \lceil\lceil \mathit{Rep\text{-}Set\text{-}0}\ (x\ \tau)\rceil\rceil.\ (\tau \models y{-}{>}\mathit{includes}((\lambda\text{-}.\ x)))$)
prefer *2*
apply(*simp add*: *OclValid-def*)
apply (*metis* (*full-types*) *bot-fun-def bool-split invalid-def null-fun-def*)
apply(*rule subsetI*, *rename-tac e*)
apply(*drule-tac* $P = \lambda x.\ \tau \models y{-}{>}\mathit{includes}((\lambda\text{-}.\ x))$ **and** $x = e$ **in** *ballE*) **prefer** *3* **apply** *assumption*
apply(*simp add*: *OclIncludes-def OclValid-def*)
apply (*metis discr-eq-bot2-true option.inject true-def*)
apply(*simp*)

apply(*simp add*: *OclValid-def OclForall-def x-def*[*simplified OclValid-def*])
apply(*subgoal-tac* $(\exists x \in \lceil\lceil \mathit{Rep\text{-}Set\text{-}0}\ (x\ \tau)\rceil\rceil.\ (y{-}{>}\mathit{includes}((\lambda\text{-}.\ x)))\ \tau = \mathit{false}\ \tau$
$\lor (y{-}{>}\mathit{includes}((\lambda\text{-}.\ x)))\ \tau = \bot\ \tau$
$\lor (y{-}{>}\mathit{includes}((\lambda\text{-}.\ x)))\ \tau = \mathit{null}\ \tau)$)
prefer *2*
apply *metis*
apply(*erule bexE*, *rename-tac e*)
apply(*simp add*: *OclIncludes-def y-def*[*simplified OclValid-def*])

apply(*case-tac* $\tau \models \upsilon\ (\lambda\text{-}.\ e)$, *simp add*: *OclValid-def*)
apply(*erule disjE*)
apply(*metis* (*mono-tags*) *discr-eq-false-true set-mp true-def*)
apply(*simp add*: *bot-fun-def bot-option-def null-fun-def null-option-def*)
apply(*erule contrapos-nn*[*OF - Set-inv-lemma′*[*OF x-def*]], *simp*)
done
qed

lemma *forall-not-includes* :
assumes *x-def* : $\tau \models \delta\ x$
and *y-def* : $\tau \models \delta\ y$
shows (*OclForall x* (*OclIncludes y*) τ = *false* τ) = ($\neg$ ⌈⌈*Rep-Set-0* (*x* τ)⌉⌉ $\subseteq$ ⌈⌈*Rep-Set-0* (*y* τ)⌉⌉)
proof –
have *discr-eq-false-true* : $\bigwedge\tau$. (*false* τ = *true* τ) = *False* **by** (*metis OclValid-def foundation2*)
have *discr-eq-null-true* : $\bigwedge\tau$. (*null* τ = *true* τ) = *False* **by** (*metis OclValid-def foundation4*)
have *discr-eq-null-false* : $\bigwedge\tau$. (*null* τ = *false* τ) = *False* **by** (*metis defined4 foundation1 foundation16 null-fun-def*)
have *discr-neq-false-true* : $\bigwedge\tau$. (*false* $\tau \neq$ *true* τ) = *True* **by** (*metis discr-eq-false-true*)
have *discr-neq-true-false* : $\bigwedge\tau$. (*true* $\tau \neq$ *false* τ) = *True* **by** (*metis discr-eq-false-true*)
have *discr-eq-bot1-true* : $\bigwedge\tau$. ($\perp$ τ = *true* τ) = *False* **by** (*metis defined3 defined-def discr-eq-false-true*)
have *discr-eq-bot2-true* : $\bigwedge\tau$. ($\perp$ = *true* τ) = *False* **by** (*metis bot-fun-def discr-eq-bot1-true*)
have *discr-eq-bot1-false* : $\bigwedge\tau$. ($\perp$ τ = *false* τ) = *False* **by** (*metis OCL-core.bot-fun-def defined4 foundation1 foundation16*)
have *discr-eq-bot2-false* : $\bigwedge\tau$. ($\perp$ = *false* τ) = *False* **by** (*metis foundation1 foundation18′ valid4*)
show *?thesis*
apply(*subgoal-tac* $\neg$ (*OclForall x* (*OclIncludes y*) τ = *false* τ) = ($\neg$ ($\neg$ ⌈⌈*Rep-Set-0* (*x* τ)⌉⌉ $\subseteq$ ⌈⌈*Rep-Set-0* (*y* τ)⌉⌉)), *simp*)
apply(*subst forall-includes*[*symmetric*], *simp add*: *x-def*, *simp add*: *y-def*)
apply(*subst OclValid-def*)
apply(*simp add*: *OclForall-def*
discr-neq-false-true
discr-neq-true-false
discr-eq-bot1-false
discr-eq-bot2-false
discr-eq-bot1-true
discr-eq-bot2-true
discr-eq-null-false
discr-eq-null-true)
apply(*simp add*: *x-def*[*simplified OclValid-def*])
apply(*subgoal-tac* ($\forall$ *x*∈⌈⌈*Rep-Set-0* (*x* τ)⌉⌉. ((*y*−>*includes*((λ-. *x*))) τ = *true* τ $\vee$ (*y*−>*includes*((λ-. *x*))) τ = *false* τ)))
apply(*metis bot-fun-def discr-eq-bot2-true discr-eq-null-true null-fun-def*)
apply(*rule ballI*, *rename-tac e*)
apply(*simp add*: *OclIncludes-def*, *rule conjI*)
apply (*metis* (*full-types*) *false-def true-def*)

apply(*simp add*: *y-def*[*simplified OclValid-def*], *rule impI*)
apply(*drule contrapos-nn*[*OF* - *Set-inv-lemma′*[*OF x-def*], *simplified OclValid-def*], *blast* +)
done
qed

lemma *forall-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* : $\bigwedge x$ *F P*. ($\exists x \in$*insert x F*. *P x*) = (*P x* $\vee$ ($\exists x \in F$. *P x*))
 by (*metis insert-iff*)

 have *destruct-ocl* : $\bigwedge x\ \tau$. *x* = *true* τ $\vee$ *x* = *false* τ $\vee$ *x* = *null* τ $\vee$ *x* = $\bot\ \tau$
  apply(*case-tac x*) **apply** (*metis bot-Boolean-def*)
  apply(*case-tac a*) **apply** (*metis null-Boolean-def*)
  apply(*case-tac aa*) **apply** (*metis* (*full-types*) *true-def*)
 by (*metis* (*full-types*) *false-def*)

 have *disjE4* : $\bigwedge$ *P1 P2 P3 P4 R*.
   (*P1* $\vee$ *P2* $\vee$ *P3* $\vee$ *P4*) $\Longrightarrow$ (*P1* $\Longrightarrow$ *R*) $\Longrightarrow$ (*P2* $\Longrightarrow$ *R*) $\Longrightarrow$ (*P3* $\Longrightarrow$ *R*) $\Longrightarrow$ (*P4* $\Longrightarrow$ *R*) $\Longrightarrow$ *R*
 by *metis*

 show *?thesis*
  apply(*simp only*: *OclForall-def OclIterate*$_{Set}$*-def*)
  apply(*case-tac* $\tau \models \delta$ *S*, *simp only*: *OclValid-def*)
  apply(*subgoal-tac* (*if* $\exists x \in \lceil\lceil$*Rep-Set-0* (*S* τ)$\rceil\rceil$. *P* (λ-. *x*) τ = *false* τ *then false* τ
   else if $\exists x \in \lceil\lceil$*Rep-Set-0* (*S* τ)$\rceil\rceil$. *P* (λ-. *x*) τ = $\bot\ \tau$ *then* $\bot\ \tau$
    else if $\exists x \in \lceil\lceil$*Rep-Set-0* (*S* τ)$\rceil\rceil$. *P* (λ-. *x*) τ = *null* τ *then null* τ
     else true τ) = *Finite-Set.fold* (λx *acc. acc and P x*) *true* (($\lambda a\ \tau$. *a*) ‘ $\lceil\lceil$*Rep-Set-0* (*S* τ)$\rceil\rceil$) τ,
   simp add: *S-finite*)
  apply(*case-tac* $\lceil\lceil$*Rep-Set-0* (*S* τ)$\rceil\rceil$ = {}, *simp*)
  apply(*rule finite-ne-induct*[**where** *P* = λ*set*. (*if* $\exists x \in$*set*. *P* (λ-. *x*) τ = *false* τ *then false* τ
   else if $\exists x \in$*set*. *P* (λ-. *x*) τ = $\bot\ \tau$ *then* $\bot\ \tau$
    else if $\exists x \in$*set*. *P* (λ-. *x*) τ = *null* τ *then null* τ *else true* τ) =
   Finite-Set.fold (λx *acc. acc and P x*) *true* (($\lambda a\ \tau$. *a*) ‘ *set*) τ, *OF S-finite*])
  apply(*simp*)

  apply(*simp only*: *image-insert*)
  apply(*subst comp-fun-commute.fold-insert*[*OF and-comm*], *simp*)
  apply (*metis empty-iff image-empty*)
  apply(*simp*)
  apply (*metis OCL-core.bot-fun-def destruct-ocl null-fun-def*)

  apply(*simp only*: *image-insert*)
  apply(*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*:)
  apply(*simp only*: *ex-insert*)
  apply(*subgoal-tac* $\exists x$. $x \in F$) **prefer** *2*

apply(*metis all-not-in-conv*)
proof − **fix** x F **show** $(\delta\ S)\ \tau = true\ \tau \Longrightarrow \exists x.\ x \in F \Longrightarrow$
$(if\ P\ (\lambda\text{-}.\ x)\ \tau = false\ \tau \vee (\exists x{\in}F.\ P\ (\lambda\text{-}.\ x)\ \tau = false\ \tau)\ then\ false\ \tau$
$else\ if\ P\ (\lambda\text{-}.\ x)\ \tau = \bot\ \tau \vee (\exists x{\in}F.\ P\ (\lambda\text{-}.\ x)\ \tau = \bot\ \tau)\ then\ \bot\ \tau$
$else\ if\ P\ (\lambda\text{-}.\ x)\ \tau = null\ \tau \vee (\exists x{\in}F.\ P\ (\lambda\text{-}.\ x)\ \tau = null\ \tau)\ then\ null\ \tau\ else\ (\delta$
$S)\ \tau) =$
$((\lambda\text{-}.\ if\ \exists x{\in}F.\ P\ (\lambda\text{-}.\ x)\ \tau = false\ \tau\ then\ false\ \tau$
$else\ if\ \exists x{\in}F.\ P\ (\lambda\text{-}.\ x)\ \tau = \bot\ \tau\ then\ \bot\ \tau$
$else\ if\ \exists x{\in}F.\ P\ (\lambda\text{-}.\ x)\ \tau = null\ \tau\ then\ null\ \tau\ else\ (\delta\ S)\ \tau)\ and$
$(\lambda\text{-}.\ P\ (\lambda\tau.\ x)\ \tau))$
τ
apply(*cut-tac destruct-ocl*[**where** $x = P\ (\lambda\tau.\ x)\ \tau$ **and** $\tau = \tau$])
apply(*erule disjE4*)
apply(*simp-all add*: *true-def false-def null-fun-def null-option-def bot-fun-def bot-option-def OclAnd-def*)
by (*metis* (*lifting*) *option.distinct*(*1*))+
apply-end(*simp add*: *OclValid-def*)+
qed
qed

4.6.8. OclExists

lemma *exists-set-null-exec*[*simp*,*code-unfold*] :
$(null{-}{>}exists(z \mid P(z))) = invalid$
by(*simp add*: *OclExists-def*)

lemma *exists-set-mt-exec*[*simp*,*code-unfold*] :
$((Set\{\}){-}{>}exists(z \mid P(z))) = false$
by(*simp add*: *OclExists-def*)

lemma *exists-set-including-exec*[*simp*,*code-unfold*] :
assumes *cp*: *cp P*
shows $((S{-}{>}including(x)){-}{>}exists(z \mid P(z))) = (if\ \delta\ S\ and\ \upsilon\ x$
$then\ P\ x\ or\ (S{-}{>}exists(z \mid P(z)))$
$else\ invalid$
$endif)$
by(*simp add*: *OclExists-def OclOr-def forall-set-including-exec cp OclNot-inject*)

4.6.9. OclIterate

lemma $OclIterate_{Set}$*-infinite*:
assumes *non-finite*: $\tau \models not(\delta(S{-}{>}size()))$
shows $(OclIterate_{Set}\ S\ A\ F)\ \tau = invalid\ \tau$
apply(*insert non-finite* [*THEN OclSize-infinite*])
apply(*erule disjE*)
apply(*simp-all add*: $OclIterate_{Set}$*-def invalid-def*)
apply(*erule contrapos-np*)
apply(*simp add*: *OclValid-def*)
done

lemma $OclIterate_{Set}$-*empty*[*simp*,*code-unfold*]: $((Set\{\})\text{−>}iterate(a;\ x = A \mid P\ a\ x)) = A$
proof −
have *A1* : $\lfloor\lfloor\{\}\rfloor\rfloor \in \{X.\ X = bot \lor X = null \lor (\forall x\in\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$ **by**(*simp add*: *mtSet-def*)
have *C* : $\bigwedge \tau.\ (\delta\ (\lambda\tau.\ Abs\text{-}Set\text{-}0\ \lfloor\lfloor\{\}\rfloor\rfloor))\ \tau = true\ \tau$
by (*metis A1 Abs-Set-0-cases Abs-Set-0-inverse cp-defined defined-def false-def mtSet-def mtSet-defined null-fun-def null-option-def null-set-OclNot-defined true-def*)
show *?thesis*
apply(*simp add*: $OclIterate_{Set}$-*def mtSet-def Abs-Set-0-inverse valid-def C*)
apply(*rule ext*)
apply(*case-tac* $A\ \tau = \bot\ \tau$, *simp-all*, *simp add*:*true-def false-def bot-fun-def*)
apply(*simp add*: *A1 Abs-Set-0-inverse*)
done
qed

In particular, this does hold for A = null.

lemma $OclIterate_{Set}$-*including*:
assumes *S-finite*: $\tau \models \delta(S\text{−>}size())$
and *F-valid-arg*: $(\upsilon\ A)\ \tau = (\upsilon\ (F\ a\ A))\ \tau$
and *F-commute*: *comp-fun-commute F*
and *F-cp*: $\bigwedge\ x\ y\ \tau.\ F\ x\ y\ \tau = F\ (\lambda\ \text{-}.\ x\ \tau)\ y\ \tau$
shows $((S\text{−>}including(a))\text{−>}iterate(a;\ x = A \mid F\ a\ x))\ \tau =$
$((S\text{−>}excluding(a))\text{−>}iterate(a;\ x = F\ a\ A \mid F\ a\ x))\ \tau$
proof −

have *valid-inject-true* : $\bigwedge\tau\ P.\ (\upsilon\ P)\ \tau \neq true\ \tau \Longrightarrow (\upsilon\ P)\ \tau = false\ \tau$
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 *insert-in-Set-0* : $\bigwedge\tau.\ (\tau \models(\delta\ S)) \Longrightarrow (\tau \models(\upsilon\ a)) \Longrightarrow \lfloor\lfloor insert\ (a\ \tau)\ \lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil\rfloor\rfloor$
$\in \{X.\ X = bot \lor X = null \lor (\forall x\in\lceil\lceil X\rceil\rceil.\ x \neq bot)\}$
apply(*frule Set-inv-lemma*)
apply(*simp add*: *foundation18 invalid-def*)
done

have *insert-defined* : $\bigwedge\tau.\ (\tau \models(\delta\ S)) \Longrightarrow (\tau \models(\upsilon\ a)) \Longrightarrow$
$(\delta\ (\lambda\text{-}.\ Abs\text{-}Set\text{-}0\ \lfloor\lfloor insert\ (a\ \tau)\ \lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil\rfloor\rfloor))\ \tau = true\ \tau$
apply(*subst defined-def*)
apply(*simp add*: *bot-fun-def bot-option-def bot-Set-0-def null-Set-0-def null-option-def null-fun-def false-def true-def*)
apply(*subst Abs-Set-0-inject*)
apply(*rule insert-in-Set-0*, *simp-all add*: *bot-option-def*)

apply(*subst Abs-Set-0-inject*)
apply(*rule insert-in-Set-0*, *simp-all add*: *null-option-def bot-option-def*)
done

have *remove-finite* : *finite* $\lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil \Longrightarrow$ *finite* $((\lambda a\ \tau.\ a)\ `\ (\lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil\ -$

$\{a\ \tau\}))$
by(*simp*)

have *remove-in-Set-0* : $\bigwedge\tau.\ (\tau \models(\delta\ S)) \Longrightarrow (\tau \models(\upsilon\ a)) \Longrightarrow \lfloor\lfloor\lceil\lceil \mathit{Rep\text{-}Set\text{-}0}\ (S\ \tau)\rceil\rceil - \{a\ \tau\}\rfloor\rfloor \in \{X.\ X = \mathit{bot} \vee X = \mathit{null} \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq \mathit{bot})\}$
apply(*frule Set-inv-lemma*)
apply(*simp add*: *foundation18 invalid-def*)
done

have *remove-defined* : $\bigwedge\tau.\ (\tau \models(\delta\ S)) \Longrightarrow (\tau \models(\upsilon\ a)) \Longrightarrow$
$(\delta\ (\lambda\text{-}.\ \mathit{Abs\text{-}Set\text{-}0}\ \lfloor\lfloor\lceil\lceil \mathit{Rep\text{-}Set\text{-}0}\ (S\ \tau)\rceil\rceil - \{a\ \tau\}\rfloor\rfloor))\ \tau = \mathit{true}\ \tau$
apply(*subst defined-def*)
apply(*simp add*: *bot-fun-def bot-option-def bot-Set-0-def null-Set-0-def null-option-def null-fun-def false-def true-def*)
apply(*subst Abs-Set-0-inject*)
apply(*rule remove-in-Set-0*, *simp-all add*: *bot-option-def*)

apply(*subst Abs-Set-0-inject*)
apply(*rule remove-in-Set-0*, *simp-all add*: *null-option-def bot-option-def*)
done

have *abs-rep*: $\bigwedge x.\ \lfloor\lfloor x\rfloor\rfloor \in \{X.\ X = \mathit{bot} \vee X = \mathit{null} \vee (\forall x{\in}\lceil\lceil X\rceil\rceil.\ x \neq \mathit{bot})\} \Longrightarrow \lceil\lceil \mathit{Rep\text{-}Set\text{-}0}\ (\mathit{Abs\text{-}Set\text{-}0}\ \lfloor\lfloor x\rfloor\rfloor)\rceil\rceil = x$
by(*subst Abs-Set-0-inverse*, *simp-all*)

have *inject* : *inj* $(\lambda a\ \tau.\ a)$
by(*rule inj-fun*, *simp*)

show *?thesis*
apply(*simp only*: *cp-OclIterate*$_{Set}$[*of S*−>*including*(*a*)] *cp-OclIterate*$_{Set}$[*of S*−>*excluding*(*a*)])
apply(*subst OclIncluding-def*, *subst OclExcluding-def*)
apply(*case-tac* $\neg\ ((\delta\ S)\ \tau = \mathit{true}\ \tau \wedge (\upsilon\ a)\ \tau = \mathit{true}\ \tau)$, *simp*)

apply(*subgoal-tac* $\mathit{OclIterate}_{Set}\ (\lambda\text{-}.\ \bot)\ A\ F\ \tau = \mathit{OclIterate}_{Set}\ (\lambda\text{-}.\ \bot)\ (F\ a\ A)\ F\ \tau$, *simp*)
apply(*rule conjI*)
apply(*blast*)
apply(*blast*)
apply(*auto*)

apply(*simp add*: *OclIterate*$_{Set}$*-def*) **apply**(*auto*)
apply(*simp add*: *defined-def bot-option-def bot-fun-def false-def true-def*)
apply(*simp add*: *defined-def bot-option-def bot-fun-def false-def true-def*)
apply(*simp add*: *defined-def bot-option-def bot-fun-def false-def true-def*)

apply(*simp add*: *OclIterate*$_{Set}$*-def*) **apply**(*auto*)
apply(*simp add*: *defined-def bot-option-def bot-fun-def false-def true-def*)
apply(*simp add*: *defined-def bot-option-def bot-fun-def false-def true-def*)
apply(*simp add*: *defined-def bot-option-def bot-fun-def false-def true-def*)

apply(*simp add*: *OclIterate*$_{Set}$*-def*)

apply(*subst abs-rep*[*OF insert-in-Set-0*[*simplified OclValid-def*], *of* τ], *simp-all*)+
apply(*subst abs-rep*[*OF remove-in-Set-0*[*simplified OclValid-def*], *of* τ], *simp-all*)+
apply(*subst insert-defined*, *simp-all add*: *OclValid-def*)+
apply(*subst remove-defined*, *simp-all add*: *OclValid-def*)+

apply(*case-tac* $\neg$ ((υ *A*) τ = *true* τ), *simp add*: *F-valid-arg*)
apply(*simp add*: *valid-inject-true F-valid-arg*)
apply(*rule impI*)
apply(*subst Finite-Set.comp-fun-commute.fold-fun-left-comm*[**where** $f = F$ **and** $z = A$ **and** $x = a$ **and** $A = ((\lambda a\ \tau.\ a)$ ' $(\lceil\lceil$*Rep-Set-0* $(S\ \tau)\rceil\rceil - \{a\ \tau\}))$, *symmetric*, *OF F-commute*])
apply(*rule remove-finite*, *simp*)

apply(*subst image-set-diff*[*OF inject*], *simp*)
apply(*subgoal-tac Finite-Set.fold F A* (*insert* $(\lambda\tau'.\ a\ \tau)$ $((\lambda a\ \tau.\ a)$ ' $\lceil\lceil$*Rep-Set-0* $(S\ \tau)\rceil\rceil))$ τ =
F $(\lambda\tau'.\ a\ \tau)$ (*Finite-Set.fold F A* $((\lambda a\ \tau.\ a)$ ' $\lceil\lceil$*Rep-Set-0* $(S\ \tau)\rceil\rceil - \{\lambda\tau'.\ a\ \tau\})$) τ)
apply(*subst F-cp*)
apply(*simp*)

apply(*subst Finite-Set.comp-fun-commute.fold-insert-remove*[*OF F-commute*])
apply(*simp*)+
done
qed

4.6.10. OclSelect

lemma *select-set-mt-exec*[*code-unfold*, *simp*]: *OclSelect*$_{set}$ *mtSet P* = *mtSet*
apply(*rule ext*, *rename-tac* τ)
apply(*simp add*: *OclSelect*$_{set}$*-def mtSet-def defined-def false-def true-def bot-Set-0-def null-Set-0-def null-fun-def bot-fun-def*)
apply(*subst* (*1 2 3 4 5*) *Abs-Set-0-inverse*)
apply(*simp add*: *null-option-def bot-option-def*)+
apply(*subst Abs-Set-0-inject*)
apply(*simp add*: *null-option-def bot-option-def*)+
done

lemma *select-set-including-exec*[*simp*,*code-unfold*]:
OclSelect$_{set}$ (*X*−>*including*(*y*)) *P* =
(*if* δ *X* *then*
  if υ *y* *then*
    if δ(*X*−>*size*()) *then*
      if P y *then* (*OclSelect*$_{set}$ *X P*)−>*including*(*y*)
      else (*OclSelect*$_{set}$ *X P*)
      endif
    else invalid
    endif
  else invalid

endif
else invalid
endif)

sorry

definition *select-body* $\equiv$ ($\lambda P\ x\ acc.$ *if* υ ($P\ x$) *then if* $P\ x \triangleq$ *false then acc else acc−>including*(x)
endif else $\bot$ *endif*)

lemma *select-body-commute* : *comp-fun-commute* (*select-body* P)
sorry

lemma *select-iterate*:
assumes *S-finite*: *finite* $\lceil\lceil$*Rep-Set-0* ($S\ \tau$)$\rceil\rceil$
and *P-strict*: $\bigwedge x.\ x\ \tau = \bot \Longrightarrow (P\ x)\ \tau = \bot$
shows $OclSelect_{set}$ $S\ P\ \tau$ = (S−>*iterate*(x; *acc* = *Set*{} | *select-body* $P\ x$ *acc*)) τ
proof −
have *ex-insert* : $\bigwedge x\ F\ P.\ (\exists x{\in}insert\ x\ F.\ P\ x) = (P\ x \vee (\exists x{\in}F.\ P\ x))$
by (*metis insert-iff*)

have *insert-set* : $\bigwedge s\ P\ S.\ \neg\ P\ s \Longrightarrow \{x \in insert\ s\ S.\ P\ x\} = \{x \in S.\ P\ x\}$
by (*metis* (*mono-tags*) *insert-iff*)

have *inj* : $\bigwedge x\ F.\ x \notin F \Longrightarrow (\lambda\tau.\ x) \notin (\lambda a\ \tau.\ a)$ ‘ F
by (*metis image-iff*)

have *valid-inject-true* : $\bigwedge\tau\ P.\ (\upsilon\ P)\ \tau \neq true\ \tau \Longrightarrow (\upsilon\ P)\ \tau = false\ \tau$
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 *defined-inject-true* : $\bigwedge\tau\ P.\ (\delta\ P)\ \tau \neq true\ \tau \Longrightarrow (\delta\ P)\ \tau = false\ \tau$
apply(*simp add*: *defined-def true-def false-def bot-fun-def bot-option-def*
null-fun-def null-option-def)
by (*case-tac* $P\ \tau = \bot \vee P\ \tau = null$, *simp-all add*: *true-def*)

have *not-strongeq* : $\bigwedge P.\ \neg\ \tau \models P \triangleq false \Longrightarrow (P \triangleq false)\ \tau = false\ \tau$
by (*metis OclNot2 OclValid-def StrongEq-sym bool-split cp-OclNot defined7 foundation1 foundation19 foundation9 valid7*)

show *?thesis*
apply(*simp add*: *select-body-def*)
apply(*simp only*: $OclSelect_{set}$*-def* $OclIterate_{Set}$*-def*)
apply(*case-tac* $\tau \models \delta\ S$, *simp only*: *OclValid-def*)
apply(*subgoal-tac* (*if* $\exists x{\in}\lceil\lceil$*Rep-Set-0* ($S\ \tau$)$\rceil\rceil$. P (λ-. x) $\tau = \bot\ \tau$ *then* $\bot$
else Abs-Set-0 $\lfloor\lfloor\{x \in \lceil\lceil$*Rep-Set-0* ($S\ \tau$)$\rceil\rceil$. P (λ-. x) $\tau \neq$ *false* $\tau\}\rfloor\rfloor$) =

$Finite\text{-}Set.fold$ $(\lambda x\ acc.\ if\ \upsilon\ (P\ x)\ then\ if\ P\ x \triangleq false\ then\ acc\ else\ acc{-}{>}including(x)$ $endif\ else\ \bot\ endif)\ Set\{\}$
$((\lambda a\ \tau.\ a)\ `\ \lceil\lceil Rep\text{-}Set\text{-}0\ (S\ \tau)\rceil\rceil)\ \tau,$
simp add: *S-finite*)
apply(*rule finite-induct*[**where** $P = \lambda set.\ (if\ \exists x{\in}set.\ P\ (\lambda\text{-}.\ x)\ \tau = \bot\ \tau\ then\ \bot$
$else\ Abs\text{-}Set\text{-}0\ \lfloor\lfloor\{x \in set.\ P\ (\lambda\text{-}.\ x)\ \tau \neq false\ \tau\}\rfloor\rfloor) =$
$Finite\text{-}Set.fold\ (\lambda x\ acc.\ if\ \upsilon\ (P\ x)\ then\ if\ P\ x \triangleq false\ then\ acc\ else\ acc{-}{>}including(x)\ endif$ $else\ \bot\ endif)\ Set\{\}$
$((\lambda a\ \tau.\ a)\ `\ set)\ \tau,$ *OF S-finite*])
apply(*simp add*: *mtSet-def*)

apply(*simp only*: *image-insert*)
apply(*subst comp-fun-commute.fold-insert*[*OF select-body-commute*[*simplified select-body-def*]], *simp*)
apply(*rule inj*, *fast*)

apply(*simp only*: *ex-insert*)
apply(*subst cp-OclIf*)
apply(*case-tac* $\neg\ ((\upsilon\ (P\ (\lambda\text{-}.\ x)))\ \tau = true\ \tau)$)
apply(*drule valid-inject-true*)
apply(*subgoal-tac* $P\ (\lambda\text{-}.\ x)\ \tau = \bot\ \tau$, *simp add*: *cp-OclIf*[*symmetric*], *simp add*: *bot-fun-def*)
apply (*metis OCL-core.bot-fun-def OclValid-def foundation2 valid-def*)

apply(*subst cp-OclIf*)
apply(*subgoal-tac* $P\ (\lambda\text{-}.\ x)\ \tau \neq \bot\ \tau$)
prefer *2*
apply (*metis OCL-core.bot-fun-def OclValid-def foundation2 valid-def*)

apply(*case-tac* $\tau \models (P\ (\lambda\text{-}.\ x) \triangleq false)$)
apply(*subst insert-set*, *metis foundation22*)

apply(*simp add*: *cp-OclIf*[*symmetric*])

apply(*subst not-strongeq*, *simp*)

apply(*simp add*: *cp-OclIf*[*symmetric*])
apply(*drule sym*, *drule sym*)
apply(*subst* (*1 2*) *cp-OclIncluding*)
apply(*subgoal-tac* $((\lambda\text{-}.\ Finite\text{-}Set.fold\ (\lambda x\ acc.\ if\ \upsilon\ P\ x\ then\ if\ P\ x \triangleq false\ then\ acc\ else$ $acc{-}{>}including(x)\ endif\ else\ \bot\ endif)\ Set\{\}\ ((\lambda a\ \tau.\ a)\ `\ F)\ \tau){-}{>}including(\lambda\tau.\ x))\ \tau$
$=$
$((\lambda\text{-}.\ if\ \exists x{\in}F.\ P\ (\lambda\text{-}.\ x)\ \tau = \bot\ \tau\ then\ \bot\ else\ Abs\text{-}Set\text{-}0\ \lfloor\lfloor\{x \in F.\ P\ (\lambda\text{-}.\ x)$ $\tau \neq false\ \tau\}\rfloor\rfloor){-}{>}including(\lambda\tau.\ x))\ \tau$)
prefer *2*
apply (*metis* (*lifting*))
apply(*simp add*:)

apply(*rule conjI*)
apply (*metis* (*no-types*) *OclIncluding-def OclValid-def foundation16*)

apply(*rule impI*, *subst OclIncluding-def*, *subst Abs-Set-0-inverse*, *simp add*: *bot-option-def null-option-def*)
apply (*metis* (*no-types*) *OCL-core.bot-fun-def P-strict*)
apply(*simp*)

apply(*drule sym*, *simp only*:, *drule sym*, *simp only*:)
apply(*subst* (*1 2*) *defined-def*, *simp add*: *bot-Set-0-def null-Set-0-def false-def true-def null-fun-def bot-fun-def*)

apply(*subgoal-tac* $(\upsilon\ (\lambda\text{-}.\ x))\ \tau = \lfloor\lfloor \mathit{True} \rfloor\rfloor$)
prefer *2*
proof − **fix** x **show** $(\upsilon\ P\ (\lambda\text{-}.\ x))\ \tau = \lfloor\lfloor \mathit{True} \rfloor\rfloor \Longrightarrow (\upsilon\ (\lambda\text{-}.\ x))\ \tau = \lfloor\lfloor \mathit{True} \rfloor\rfloor$
by (*metis OCL-core.bot-fun-def P-strict true-def valid-def*)
apply-end(*simp*)
apply-end(*simp*)
apply-end(*subgoal-tac Abs-Set-0* $\lfloor\lfloor\{x \in F.\ P\ (\lambda\text{-}.\ x)\ \tau \neq \lfloor\lfloor \mathit{False} \rfloor\rfloor\}\rfloor\rfloor \neq$ *Abs-Set-0 None* $\wedge$ *Abs-Set-0* $\lfloor\lfloor\{x \in F.\ P\ (\lambda\text{-}.\ x)\ \tau \neq \lfloor\lfloor \mathit{False} \rfloor\rfloor\}\rfloor\rfloor \neq$ *Abs-Set-0* $\lfloor \mathit{None} \rfloor$, *simp*)
apply-end(*subgoal-tac* $\{xa.\ (xa = x \vee xa \in F) \wedge P\ (\lambda\text{-}.\ xa)\ \tau \neq \lfloor\lfloor \mathit{False} \rfloor\rfloor\} = \mathit{insert}\ x\ \{x \in F.\ P\ (\lambda\text{-}.\ x)\ \tau \neq \lfloor\lfloor \mathit{False} \rfloor\rfloor\}$, *simp*)
apply-end(*rule equalityI*)
apply-end(*rule subsetI*, *simp*)
apply-end(*rule subsetI*, *simp*, *metis foundation22*)

fix F
show $\forall x{\in}F.\ P\ (\lambda\text{-}.\ x)\ \tau \neq \bot \Longrightarrow$ *Abs-Set-0* $\lfloor\lfloor\{x \in F.\ P\ (\lambda\text{-}.\ x)\ \tau \neq \lfloor\lfloor \mathit{False} \rfloor\rfloor\}\rfloor\rfloor \neq$ *Abs-Set-0 None* $\wedge$ *Abs-Set-0* $\lfloor\lfloor\{x \in F.\ P\ (\lambda\text{-}.\ x)\ \tau \neq \lfloor\lfloor \mathit{False} \rfloor\rfloor\}\rfloor\rfloor \neq$ *Abs-Set-0* $\lfloor \mathit{None} \rfloor$
apply(*subst* (*1 2*) *Abs-Set-0-inject*, *simp-all add*: *bot-option-def null-option-def*)
apply(*rule allI*, *rule impI*)
proof − **fix** x **show** $\forall x{\in}F.\ \exists y.\ P\ (\lambda\text{-}.\ x)\ \tau = \lfloor y \rfloor \Longrightarrow x \in F \wedge P\ (\lambda\text{-}.\ x)\ \tau \neq \lfloor\lfloor \mathit{False} \rfloor\rfloor \Longrightarrow x \neq \bot$
apply(*case-tac* $x = \bot$, *drule P-strict*[**where** $x = \lambda\text{-}.\ x$])
apply(*drule-tac* $x = x$ **in** *ballE*) **prefer** *3* **apply** *assumption*
apply(*simp add*: *bot-option-def*)+
done
apply-end(*simp*)+
qed
apply-end(*simp add*: *OclValid-def*)+
qed
qed

4.6.11. Strict Equality

lemma *StrictRefEq*$_{Set}$*-exec*[*simp*,*code-unfold*] :
$((x{::}('\mathfrak{A},'\alpha{::}null)Set) \doteq y) =$
(*if* δ x *then* (*if* δ y
then ((*x*−>*forAll*(*z*| *y*−>*includes*(*z*)) *and* (*y*−>*forAll*(*z*| *x*−>*includes*(*z*)))))
else if υ y
then false (∗ *x*′−>*includes* = *null* ∗)

else invalid
endif
endif)
else if υ *x* (* *null* = ??? *)
then if υ *y then not*(δ *y*) *else invalid endif*
else invalid
endif
endif)

proof −

have *defined-inject-true* : $\bigwedge\tau\ P.\ \neg\ (\tau \models \delta\ P) \Longrightarrow (\delta\ P)\ \tau = \mathit{false}\ \tau$
by(*metis bot-fun-def defined-def foundation16 null-fun-def*)

have *valid-inject-true* : $\bigwedge\tau\ P.\ \neg\ (\tau \models \upsilon\ P) \Longrightarrow (\upsilon\ P)\ \tau = \mathit{false}\ \tau$
by(*metis bot-fun-def foundation18' valid-def*)

have *valid-inject-defined* : $\bigwedge\tau\ P.\ \neg\ (\tau \models \upsilon\ P) \Longrightarrow \neg\ (\tau \models \delta\ P)$
by(*metis foundation20*)

have *null-simp* : $\bigwedge\tau\ y.\ \tau \models \upsilon\ y \Longrightarrow \neg\ (\tau \models \delta\ y) \Longrightarrow y\ \tau = \mathit{null}\ \tau$
by (*simp add*: *foundation16 foundation18' null-fun-def*)

have *discr-eq-false-true* : $\bigwedge\tau.\ (\mathit{false}\ \tau = \mathit{true}\ \tau) = \mathit{False}$ **by** (*metis OclValid-def foundation2*)
have *discr-neq-true-false* : $\bigwedge\tau.\ (\mathit{true}\ \tau \neq \mathit{false}\ \tau) = \mathit{True}$ **by** (*metis discr-eq-false-true*)

have *strongeq-true* : $\bigwedge\ \tau\ x\ y.\ (\lfloor\lfloor x\ \tau = y\ \tau\rfloor\rfloor = \mathit{true}\ \tau) = (x\ \tau = y\ \tau)$
by(*simp add*: *foundation22*[*simplified OclValid-def StrongEq-def*])

have *strongeq-false* : $\bigwedge\ \tau\ x\ y.\ (\lfloor\lfloor x\ \tau = y\ \tau\rfloor\rfloor = \mathit{false}\ \tau) = (x\ \tau \neq y\ \tau)$
apply(*case-tac* $x\ \tau \neq y\ \tau$, *simp add*: *false-def*)
apply(*simp add*: *false-def true-def*)
done

have *rep-set-inj* : $\bigwedge\tau.\ (\delta\ x)\ \tau = \mathit{true}\ \tau \Longrightarrow$
$(\delta\ y)\ \tau = \mathit{true}\ \tau \Longrightarrow$
$x\ \tau \neq y\ \tau \Longrightarrow$
$\lceil\lceil \mathit{Rep\text{-}Set\text{-}0}\ (y\ \tau)\rceil\rceil \neq \lceil\lceil \mathit{Rep\text{-}Set\text{-}0}\ (x\ \tau)\rceil\rceil$
apply(*simp add*: *defined-def*)
apply(*split split-if-asm*, *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* $x\ \tau$)
apply(*case-tac ya*, *simp-all*)
apply(*case-tac a*, *simp-all*)

apply(*case-tac* $y\ \tau$)
apply(*case-tac yaa*, *simp-all*)
apply(*case-tac ab*, *simp-all*)

apply(*simp add*: *Abs-Set-0-inverse*)

apply(*blast*)
done

show *?thesis*
apply(*rule ext*, *rename-tac* τ)

apply(*simp add*: *cp-OclIf*[*of* δ *x*])
apply(*case-tac* $\neg$ ($\tau \models \upsilon$ *x*))
apply(*subgoal-tac* $\neg$ ($\tau \models \delta$ *x*))
prefer *2*
apply(*metis foundation20*)
apply(*simp add*: *defined-inject-true*)
apply(*simp add*: *cp-OclIf*[*symmetric*] *OclValid-def StrictRefEq*$_{Set}$)

apply(*simp*)

apply(*case-tac* $\neg$ ($\tau \models \upsilon$ *y*))
apply(*subgoal-tac* $\neg$ ($\tau \models \delta$ *y*))
prefer *2*
apply(*metis foundation20*)
apply(*simp add*: *defined-inject-true*)
apply(*simp add*: *cp-OclIf*[*symmetric*] *OclValid-def StrictRefEq*$_{Set}$)

apply(*simp*)

apply(*simp add*: *cp-OclIf*[*of* δ *y*])
apply(*simp add*: *cp-OclIf*[*symmetric*])

apply(*simp add*: *cp-OclIf*[*of* δ *x*])
apply(*case-tac* $\neg$ ($\tau \models \delta$ *x*))
apply(*simp add*: *defined-inject-true*)
apply(*simp add*: *cp-OclIf*[*symmetric*])
apply(*simp add*: *cp-OclNot*[*of* δ *y*])
apply(*case-tac* $\neg$ ($\tau \models \delta$ *y*))
apply(*simp add*: *defined-inject-true*)
apply(*simp add*: *cp-OclNot*[*symmetric*])
apply(*metis* (*hide-lams*, *no-types*) *OclValid-def StrongEq-sym foundation22 null-fun-def null-simp StrictRefEq*$_{Set}$*-vs-StrongEq true-def*)
apply(*simp add*: *OclValid-def cp-OclNot*[*symmetric*])

apply(*simp add*: *null-simp*[*simplified OclValid-def*, *of x*] *StrictRefEq*$_{Set}$ *StrongEq-def false-def*)
apply(*simp add*: *defined-def*[*of y*])
apply(*metis discr-neq-true-false*)

apply(*simp*)
apply(*simp add*: *OclValid-def*)

apply(*simp add*: *cp-OclIf*[*of* δ *y*])
apply(*case-tac* $\neg$ ($\tau \models \delta$ *y*))
apply(*simp add*: *defined-inject-true*)
apply(*simp add*: *cp-OclIf*[*symmetric*])

apply(*drule null-simp*[*simplified OclValid-def*, *of y*])
apply(*simp add*: *OclValid-def*)
apply(*simp add*: *cp-StrictRefEq*$_{Set}$[*of x*])
apply(*simp add*: *cp-StrictRefEq*$_{Set}$[*symmetric*])
apply(*simp add*: *null-simp*[*simplified OclValid-def*, *of y*] *StrictRefEq*$_{Set}$ *StrongEq-def false-def*)
apply(*simp add*: *defined-def*[*of x*])
apply (*metis discr-neq-true-false*)

apply(*simp add*: *OclValid-def*)

apply(*simp add*: *StrictRefEq*$_{Set}$ *StrongEq-def*)

apply(*subgoal-tac* $\lfloor\lfloor x\ \tau = y\ \tau\rfloor\rfloor = true\ \tau \vee \lfloor\lfloor x\ \tau = y\ \tau\rfloor\rfloor = false\ \tau$)
prefer *2*
apply(*case-tac* $x\ \tau = y\ \tau$)
apply(*rule disjI1*, *simp add*: *true-def*)
apply(*rule disjI2*, *simp add*: *false-def*)

apply(*erule disjE*)
apply(*simp add*: *strongeq-true*)

apply(*subgoal-tac* ($\tau \models$ *OclForall x* (*OclIncludes y*)) $\wedge$ ($\tau \models$ *OclForall y* (*OclIncludes x*)))
apply(*simp add*: *cp-OclAnd*[*of OclForall x* (*OclIncludes y*)] *true-def OclValid-def*)
apply(*simp add*: *OclValid-def*)
apply(*simp add*: *forall-includes*[*simplified OclValid-def*])

apply(*simp add*: *strongeq-false*)

apply(*subgoal-tac OclForall x* (*OclIncludes y*) τ = *false* τ $\vee$ *OclForall y* (*OclIncludes x*) τ = *false* τ)
apply(*simp add*: *cp-OclAnd*[*of OclForall x* (*OclIncludes y*)] *false-def*)
apply(*erule disjE*)
apply(*simp*)
apply(*subst cp-OclAnd*[*symmetric*])
apply(*simp only*: *OclAnd-false1*[*simplified false-def*])

apply(*simp*)
apply(*subst cp-OclAnd*[*symmetric*])
apply(*simp only*: *OclAnd-false2*[*simplified false-def*])
apply(*simp add*: *forall-not-includes*[*simplified OclValid-def*] *rep-set-inj*)

done
qed

4.7. Test Statements

lemma *syntax-test*: $Set\{\mathbf{2},\mathbf{1}\} = (Set\{\}{-}{>}including(\mathbf{1}){-}{>}including(\mathbf{2}))$
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 $H{:}(Set\{\mathbf{2}\} \doteq null) = (false{::}('\mathfrak{A})Boolean)$
shows $(\tau{::}('\mathfrak{A})st) \models (Set\{Set\{\mathbf{2}\},null\}{-}{>}includes(null))$
by(*simp add*: *includes-execute-set H*)

lemma *short-cut'*[*simp*,*code-unfold*]: $(\mathbf{8} \doteq \mathbf{6}) = false$
apply(*rule ext*)
apply(*simp add*: $StrictRefEq_{Integer}$ *StrongEq-def OclInt8-def OclInt6-def true-def false-def invalid-def bot-option-def*)
done

lemma *short-cut''*[*simp*,*code-unfold*]: $(\mathbf{2} \doteq \mathbf{1}) = false$
apply(*rule ext*)
apply(*simp add*: $StrictRefEq_{Integer}$ *StrongEq-def OclInt2-def OclInt1-def true-def false-def invalid-def bot-option-def*)
done
lemma *short-cut'''*[*simp*,*code-unfold*]: $(\mathbf{1} \doteq \mathbf{2}) = false$
apply(*rule ext*)
apply(*simp add*: $StrictRefEq_{Integer}$ *StrongEq-def OclInt2-def OclInt1-def true-def false-def invalid-def bot-option-def*)
done

Elementary computations on Sets.

value $\neg\ (\tau \models \upsilon(invalid{::}('\mathfrak{A},'\alpha{::}null)\ Set))$
value $\tau \models \upsilon(null{::}('\mathfrak{A},'\alpha{::}null)\ Set)$
value $\neg\ (\tau \models \delta(null{::}('\mathfrak{A},'\alpha{::}null)\ Set))$
value $\tau \models \upsilon(Set\{\})$
value $\tau \models \upsilon(Set\{Set\{\mathbf{2}\},null\})$
value $\tau \models \delta(Set\{Set\{\mathbf{2}\},null\})$
value $\tau \models (Set\{\mathbf{2},\mathbf{1}\}{-}{>}includes(\mathbf{1}))$
value $\neg\ (\tau \models (Set\{\mathbf{2}\}{-}{>}includes(\mathbf{1})))$
value $\neg\ (\tau \models (Set\{\mathbf{2},\mathbf{1}\}{-}{>}includes(null)))$
value $\tau \models (Set\{\mathbf{2},null\}{-}{>}includes(null))$
value $\tau \models (Set\{null,\mathbf{2}\}{-}{>}includes(null))$

value $\tau \models ((Set\{\}){-}{>}forAll(z \mid \mathbf{0} <_{ocl} z))$

value $\tau \models$ *if* $\mathbf{0} <_I \mathbf{2}$ *then if* $\mathbf{0} <_I \mathbf{1}$ *then true else false endif else false endif*

declare *cp-intro''*[*code-unfold*]
value $\tau \models ((Set\{\mathbf{2},\mathbf{1}\}){-}{>}forAll(z \mid \mathbf{0} <_{ocl} z))$
value $\neg (\tau \models ((Set\{\mathbf{2},\mathbf{1}\}){-}{>}exists(z \mid z <_{ocl} \mathbf{0})))$
value $\neg (\tau \models \delta(Set\{\mathbf{2},null\}){-}{>}forAll(z \mid \mathbf{0} <_{ocl} z))$
value $\neg (\tau \models ((Set\{\mathbf{2},null\}){-}{>}forAll(z \mid \mathbf{0} <_{ocl} z)))$
value $\tau \models ((Set\{\mathbf{2},null\}){-}{>}exists(z \mid \mathbf{0} <_{ocl} z))$

value $\neg (\tau \models \mathbf{0} <_{ocl} null)$
value $\tau \models not(\delta(\mathbf{0} <_{ocl} null))$

value $\neg (\tau \models (Set\{null::'a\ Boolean\} \doteq Set\{\}))$
value $\neg (\tau \models (Set\{null::'a\ Integer\} \doteq Set\{\}))$

value $(\tau \models (Set\{\lambda\text{-}.\ \lfloor\lfloor x\rfloor\rfloor\} \doteq Set\{\lambda\text{-}.\ \lfloor\lfloor x\rfloor\rfloor\}))$
value $(\tau \models (Set\{\lambda\text{-}.\ \lfloor x\rfloor\} \doteq Set\{\lambda\text{-}.\ \lfloor x\rfloor\}))$

lemma $\neg (\tau \models (Set\{true\} \doteq Set\{false\}))$ **by** *simp*
lemma $\neg (\tau \models (Set\{true,true\} \doteq Set\{false\}))$ **by** *simp*
lemma $\neg (\tau \models (Set\{\mathbf{2}\} \doteq Set\{\mathbf{1}\}))$ **by** *simp*
lemma $\tau \models (Set\{\mathbf{2},null,\mathbf{2}\} \doteq Set\{null,\mathbf{2}\})$ **by** *simp*
lemma $\tau \models (Set\{\mathbf{1},null,\mathbf{2}\} <> Set\{null,\mathbf{2}\})$ **by** *simp*
lemma $\tau \models (Set\{Set\{\mathbf{2},null\}\} \doteq Set\{Set\{null,\mathbf{2}\}\})$ **by** *simp*
lemma $\tau \models (Set\{Set\{\mathbf{2},null\}\} <> Set\{Set\{null,\mathbf{2}\},null\})$ **by** *simp*
lemma $\neg (\tau \models (Set\{null\}{-}{>}select(x \mid not\ x) \doteq Set\{null\}))$ **by** *simp*

end

5. Part III: State Operations and Objects

theory *OCL-state*
imports *OCL-lib*
begin

5.1. Complex Types: The Object Type (I) Core

5.1.1. Recall: The generic structure of States

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

type_synonym oid = nat

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 in order to leave the possibility to have Bags, OrderedSets or Sequences as association ends.

Recall:

record ('\<AA>)state =
heap :: "oid ⇀ '\<AA> "
assocs :: "oid ⇀ (oid × oid) list "

type_synonym ('\<AA>)st = "'\<AA> state ×'\<AA> 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:

typ $'\mathfrak{A}$ :: *object*

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

5.2. Fundamental Predicates on Object: Strict Equality

5.2.1. Definition

Generic referential equality - to be used for instantiations with concrete object types ...

definition $\mathit{StrictRefEq}_{Object}$ :: $('\mathfrak{A},'a::\{object,null\})val \Rightarrow ('\mathfrak{A},'a)val \Rightarrow ('\mathfrak{A})Boolean$
where $\mathit{StrictRefEq}_{Object}\ x\ y$
$\equiv \lambda\ \tau.\ \mathit{if}\ (\upsilon\ x)\ \tau = \mathit{true}\ \tau \wedge (\upsilon\ y)\ \tau = \mathit{true}\ \tau$
$\mathit{then}\ \mathit{if}\ x\ \tau = \mathit{null} \vee y\ \tau = \mathit{null}$
$\mathit{then}\ \lfloor\lfloor x\ \tau = \mathit{null} \wedge y\ \tau = \mathit{null}\rfloor\rfloor$
$\mathit{else}\ \lfloor\lfloor(\mathit{oid\text{-}of}\ (x\ \tau)) = (\mathit{oid\text{-}of}\ (y\ \tau))\ \rfloor\rfloor$
$\mathit{else}\ \mathit{invalid}\ \tau$

5.2.2. Logic and Algebraic Layer on Object

Validity and Definedness Properties

We derive the usual laws on definedness for (generic) object equality:

lemma $\mathit{StrictRefEq}_{Object}$*-defargs*:
$\tau \models (\mathit{StrictRefEq}_{Object}\ x\ (y::('\mathfrak{A},'a::\{null,object\})val)) \Longrightarrow (\tau \models(\upsilon\ x)) \wedge (\tau \models(\upsilon\ y))$
by(*simp add*: $\mathit{StrictRefEq}_{Object}$*-def OclValid-def true-def invalid-def bot-option-def*
split: *bool.split-asm HOL.split-if-asm*)

Symmetry

lemma $\mathit{StrictRefEq}_{Object}$*-sym* : **assumes** *x-val* : $\tau \models \upsilon\ x$ **shows** $\tau \models \mathit{StrictRefEq}_{Object}\ x\ x$
by(*simp add*: $\mathit{StrictRefEq}_{Object}$*-def true-def OclValid-def x-val*[*simplified OclValid-def*])

Execution with invalid or null as argument

lemma $\mathit{StrictRefEq}_{Object}$*-strict1*[*simp*] :
$(\mathit{StrictRefEq}_{Object}\ x\ \mathit{invalid}) = \mathit{invalid}$
by(*rule ext*, *simp add*: $\mathit{StrictRefEq}_{Object}$*-def true-def false-def*)

lemma $\mathit{StrictRefEq}_{Object}$*-strict2*[*simp*] :
$(\mathit{StrictRefEq}_{Object}\ \mathit{invalid}\ x) = \mathit{invalid}$
by(*rule ext*, *simp add*: $\mathit{StrictRefEq}_{Object}$*-def true-def false-def*)

Context Passing

lemma *cp-*$\mathit{StrictRefEq}_{Object}$:
$(\mathit{StrictRefEq}_{Object}\ x\ y\ \tau) = (\mathit{StrictRefEq}_{Object}\ (\lambda\text{-}.\ x\ \tau)\ (\lambda\text{-}.\ y\ \tau))\ \tau$
by(*auto simp*: $\mathit{StrictRefEq}_{Object}$*-def cp-valid*[*symmetric*])

lemmas *cp-intro''*[*simp*,*intro*!] =
cp-intro''
cp-$\mathit{StrictRefEq}_{Object}$[*THEN allI*[*THEN allI*[*THEN allI*[*THEN cpI2*]],
of $\mathit{StrictRefEq}_{Object}$]]

Behavior vs StrongEq

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.

definition $WFF :: ('\mathfrak{A}::object)st \Rightarrow bool$
where $WFF\ \tau = ((\forall\ x \in ran(heap(fst\ \tau)).\ \lceil heap(fst\ \tau)\ (oid\text{-}of\ x)\rceil = x) \wedge$
$(\forall\ x \in ran(heap(snd\ \tau)).\ \lceil heap(snd\ \tau)\ (oid\text{-}of\ x)\rceil = x))$

This is a generic definition of referential equality: Equality on objects in a state is reduced to equality on the references to these objects. As in HOL-OCL, 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-correspondance" of objects to their references — and therefore the definition below behaves as we expect.

Generic Referential Equality enjoys the usual properties: (quasi) reflexivity, symmetry, transitivity, substitutivity for defined values. For type-technical reasons, for each concrete object type, the equality $\doteq$ is defined by generic referential equality.

theorem $StrictRefEq_{Object}\text{-}vs\text{-}StrongEq$:
$WFF\ \tau \Longrightarrow \tau \models(\upsilon\ x) \Longrightarrow \tau \models(\upsilon\ y) \Longrightarrow$
$(x\ \tau \in ran\ (heap(fst\ \tau)) \wedge y\ \tau \in ran\ (heap(fst\ \tau))) \wedge$
$(x\ \tau \in ran\ (heap(snd\ \tau)) \wedge y\ \tau \in ran\ (heap(snd\ \tau))) \Longrightarrow$ (* *x and y must be object representations that exist in either the pre or post state* *)
$(\tau \models (StrictRefEq_{Object}\ x\ y)) = (\tau \models (x \triangleq y))$
apply(*auto simp*: *StrictRefEq*$_{Object}$*-def OclValid-def WFF-def StrongEq-def true-def Ball-def*)
apply(*erule-tac x=x τ* **in** *allE′*, *simp-all*)
done

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. Uffz.

5.3. Complex Types: The Object Type (II) Library

5.3.1. Initial States (for Testing and Code Generation)

definition $\tau_0 :: ('\mathfrak{A})st$
where $\tau_0 \equiv ((\!|heap=Map.empty,\ assocs_2= Map.empty,\ assocs_3= Map.empty|\!),$
$(\!|heap=Map.empty,\ assocs_2= Map.empty,\ assocs_3= Map.empty|\!))$

5.3.2. OclAllInstances

In order to denote OCL-types occuring in OCL expressions syntactically — as, for example, as "argument" of allInstances — we use the inverses of the injection functions into the object universes; we show that this is sufficient "characterization".

definition [*simp*]: $OclAllInstances = (\lambda\ fst\text{-}snd\ H\ \tau.$
$Abs\text{-}Set\text{-}0\ \lfloor\lfloor\ Some\ `\ ((H\ `\ ran\ (heap\ (fst\text{-}snd\ \tau))) - \{\ None\ \})\ \rfloor\rfloor)$

definition *OclAllInstances-at-post* :: $('\mathfrak{A} \Rightarrow '\alpha$ *option*$) \Rightarrow ('\mathfrak{A}$ :: *object*, $'\alpha$ *option option*) *Set*
(- *.allInstances*'('))
where *OclAllInstances-at-post H* τ = *OclAllInstances snd H* τ

definition *OclAllInstances-at-pre* :: $('\mathfrak{A} \Rightarrow '\alpha$ *option*$) \Rightarrow ('\mathfrak{A}$ ::*object*, $'\alpha$ *option option*) *Set*
(- *.allInstances@pre*'('))
where *OclAllInstances-at-pre H* τ = *OclAllInstances fst H* τ

lemma *OclAllInstances-defined*: $\tau \models \delta$ (*X .allInstances*())
apply(*simp add*: *defined-def OclValid-def OclAllInstances-at-post-def bot-fun-def bot-Set-0-def null-fun-def null-Set-0-def false-def true-def*)
apply(*rule conjI*)
apply(*rule notI*, *subst* (*asm*) *Abs-Set-0-inject*, *simp*)
apply(*rule disjI2*)+
apply (*metis bot-option-def option.distinct*(*1*))
apply(*simp add*: *bot-option-def*)+

apply(*rule notI*, *subst* (*asm*) *Abs-Set-0-inject*, *simp*)
apply(*rule disjI2*)+
apply (*metis bot-option-def option.distinct*(*1*))
apply(*simp add*: *bot-option-def null-option-def*)+
done

lemma $\tau_0 \models$ *H .allInstances*() $\triangleq$ *Set*{}
by(*simp add*: *StrongEq-def OclAllInstances-at-post-def OclValid-def* τ_0*-def mtSet-def*)

lemma $\tau_0 \models$ *H .allInstances@pre*() $\triangleq$ *Set*{}
by(*simp add*: *StrongEq-def OclAllInstances-at-pre-def OclValid-def* τ_0*-def mtSet-def*)

lemma *state-update-vs-allInstances-empty*:
shows (*Type .allInstances*())
$(\sigma, (\!|heap{=}empty,\ assocs_2{=}A,\ assocs_3{=}B|\!))$
=
Set{}
$(\sigma, (\!|heap{=}empty,\ assocs_2{=}A,\ assocs_3{=}B|\!))$
by(*simp add*: *OclAllInstances-at-post-def mtSet-def*)

lemma *state-update-vs-allInstances-including*':
assumes $\bigwedge x.\ \sigma'$ *oid* = *Some x* $\Longrightarrow$ *x* = *Object*
and *Type Object* $\neq$ *None*
shows (*Type .allInstances*())
$(\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object),\ assocs_2{=}A,\ assocs_3{=}B|\!))$
=
((*Type .allInstances*())->*including*(λ -. $\lfloor\lfloor$ *drop* (*Type Object*) $\rfloor\rfloor$))
$(\sigma, (\!|heap{=}\sigma',assocs_2{=}A,\ assocs_3{=}B|\!))$
proof −
have *allinst-def* : $(\sigma, (\!|heap = \sigma',\ assocs_2{=}A,\ assocs_3{=}B|\!)) \models (\delta$ (*Type .allInstances*()))

apply(*simp add*: *defined-def OclValid-def bot-fun-def null-fun-def bot-Set-0-def null-Set-0-def OclAllInstances-at-post-def*)
apply(*subst* (*1 2*) *Abs-Set-0-inject*)
by(*simp add*: *bot-option-def null-option-def*)+

have *drop-none* : $\bigwedge x.\ x \neq None \Longrightarrow \lfloor\lceil x\rceil\rfloor = x$
by(*case-tac x*, *simp*+)

have *insert-diff* : $\bigwedge x\ S.\ insert\ \lfloor x\rfloor\ (S - \{None\}) = (insert\ \lfloor x\rfloor\ S) - \{None\}$
by (*metis insert-Diff-if option.distinct*(*1*) *singletonE*)

show *?thesis*
apply(*simp add*: *OclIncluding-def allinst-def*[*simplified OclValid-def*] *OclAllInstances-at-post-def*)
apply(*subst Abs-Set-0-inverse*, *simp add*: *bot-option-def*, *simp add*: *comp-def*)
apply(*subst image-insert*[*symmetric*])
apply(*subst drop-none*, *simp add*: *assms*)
apply(*case-tac Type Object*, *simp add*: *assms*, *simp only*:)
apply(*subst insert-diff*, *drule sym*, *simp*)
apply(*subgoal-tac ran* $(\sigma'(oid \mapsto Object)) = insert\ Object\ (ran\ \sigma')$, *simp*)
apply(*case-tac* $\neg\ (\exists\, x.\ \sigma'\ oid = Some\ x)$)
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

lemma *state-update-vs-allInstances-including*:
assumes $\bigwedge x.\ \sigma'\ oid = Some\ x \Longrightarrow x = Object$
and $Type\ Object \neq None$
shows $(Type\ .allInstances())$
$(\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object),\ assocs_2{=}A,\ assocs_3{=}B|\!))$
$=$
$((\lambda\text{-}.\ (Type\ .allInstances())\ (\sigma, (\!|heap{=}\sigma',\ assocs_2{=}A,\ assocs_3{=}B|\!)))\text{->}including(\lambda\ \text{-}.\ \lfloor\lfloor$ $drop\ (Type\ Object)\ \rfloor\rfloor))$
$(\sigma, (\!|heap{=}\sigma'(oid{\mapsto}Object),\ assocs_2{=}A,\ assocs_3{=}B|\!))$
proof −
have *allinst-def* : $(\sigma, (\!|heap = \sigma',\ assocs_2{=}A,\ assocs_3{=}B|\!)) \models (\delta\ (Type\ .allInstances()))$
apply(*simp add*: *defined-def OclValid-def bot-fun-def null-fun-def bot-Set-0-def null-Set-0-def OclAllInstances-at-post-def*)
apply(*subst* (*1 2*) *Abs-Set-0-inject*)
by(*simp add*: *bot-option-def null-option-def*)+

show *?thesis*

apply(*subst state-update-vs-allInstances-including*′, (*simp add*: *assms*)+)
apply(*subst cp-OclIncluding*)

apply(*simp add*: *OclIncluding-def*)
apply(*subst* (*1 3*) *cp-defined*[*symmetric*], *simp add*: *allinst-def*[*simplified OclValid-def*])

apply(*simp add*: *defined-def OclValid-def bot-fun-def null-fun-def bot-Set-0-def null-Set-0-def OclAllInstances-at-post-def*)
apply(*subst* (*1 3*) *Abs-Set-0-inject*)
by(*simp add*: *bot-option-def null-option-def*)+
qed

lemma *state-update-vs-allInstances-noincluding'*:
assumes $\bigwedge x.\ \sigma'\ oid = Some\ x \Longrightarrow x = Object$
and $Type\ Object = None$
shows $(Type\ .allInstances())$
$(\sigma,\ (\!|heap{=}\sigma'(oid{\mapsto}Object),\ assocs_2{=}A,\ assocs_3{=}B|\!))$
$=$
$(Type\ .allInstances())$
$(\sigma,\ (\!|heap{=}\sigma',\ assocs_2{=}A,\ assocs_3{=}B|\!))$
proof −
have *allinst-def* : $(\sigma,\ (\!|heap = \sigma',\ assocs_2{=}A,\ assocs_3{=}B|\!)) \models (\delta\ (Type\ .allInstances()))$
apply(*simp add*: *defined-def OclValid-def bot-fun-def null-fun-def bot-Set-0-def null-Set-0-def OclAllInstances-at-post-def*)
apply(*subst* (*1 2*) *Abs-Set-0-inject*)
by(*simp add*: *bot-option-def null-option-def*)+

have *drop-none* : $\bigwedge x.\ x \neq None \Longrightarrow \lfloor\lceil x\rceil\rfloor = x$
by(*case-tac x*, *simp*+)

have *insert-diff* : $\bigwedge x\ S.\ insert\ \lfloor x\rfloor\ (S - \{None\}) = (insert\ \lfloor x\rfloor\ S) - \{None\}$
by (*metis insert-Diff-if option.distinct*(*1*) *singletonE*)

show *?thesis*
apply(*simp add*: *OclIncluding-def allinst-def*[*simplified OclValid-def*] *OclAllInstances-at-post-def*)
apply(*subgoal-tac* $ran\ (\sigma'(oid \mapsto Object)) = insert\ Object\ (ran\ \sigma')$, *simp add*: *assms*)
apply(*case-tac* $\neg\ (\exists x.\ \sigma'\ oid = Some\ x)$)
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

lemma *state-update-vs-allInstances-noincluding*:
assumes $\bigwedge x.\ \sigma'\ oid = Some\ x \Longrightarrow x = Object$
and $Type\ Object = None$
shows $(Type\ .allInstances())$

$(\sigma,\ (\!|heap{=}\sigma'(oid{\mapsto}Object),\ assocs_2{=}A,\ assocs_3{=}B|\!))$
$=$
$(\lambda\text{-}.\ (Type\ .allInstances())\ (\sigma,\ (\!|heap{=}\sigma',\ assocs_2{=}A,\ assocs_3{=}B|\!)))$
$(\sigma,\ (\!|heap{=}\sigma'(oid{\mapsto}Object),\ assocs_2{=}A,\ assocs_3{=}B|\!))$
by(*subst state-update-vs-allInstances-noincluding'*, (*simp add*: *assms*)+)

theorem *state-update-vs-allInstances*:
assumes $oid \notin dom\ \sigma'$
and $cp\ P$
shows $((\sigma,\ (\!|heap{=}\sigma'(oid{\mapsto}Object),\ assocs_2{=}A,\ assocs_3{=}B|\!)) \models (P(Type\ .allInstances()))) =$
$((\sigma,\ (\!|heap{=}\sigma',\ assocs_2{=}A,\ assocs_3{=}B|\!)) \models (P((Type\ .allInstances()){-}{>}including(\lambda\ \text{-}.$
$\lfloor\lfloor\ drop\ (Type\ Object)\ \rfloor\rfloor)))))$
proof –
 have $P\text{-}cp : \bigwedge x\ \tau.\ P\ x\ \tau = P\ (\lambda\text{-}.\ x\ \tau)\ \tau$
 by (*metis* (*full-types*) *assms*(*2*) *cp-def*)
oops

theorem *state-update-vs-allInstances-at-pre*:
assumes $oid \notin dom\ \sigma$
and $cp\ P$
shows $(((\!|heap{=}\sigma(oid{\mapsto}Object),\ assocs_2{=}A,\ assocs_3{=}B|\!),\ \sigma') \models (P(Type\ .allInstances@pre())))$
$=$
$(((\!|heap{=}\sigma,\ assocs_2{=}A,\ assocs_3{=}B|\!),\ \sigma') \models (P((Type\ .allInstances@pre()){-}{>}including(\lambda$
$\text{-}.\ \lfloor\lfloor drop\ (Type\ Object)\rfloor\rfloor)))))$
oops

5.3.3. OclIsNew

definition *OclIsNew*:: $('\mathfrak{A},\ '\alpha{::}\{null,object\})val \Rightarrow ('\mathfrak{A})Boolean \quad ((\text{-}).oclIsNew'('))$
where $X\ .oclIsNew() \equiv (\lambda\tau\ .\ if\ (\delta\ X)\ \tau = true\ \tau$
$\quad then\ \lfloor\lfloor oid\text{-}of\ (X\ \tau) \notin dom(heap(fst\ \tau)) \wedge$
$\quad\quad oid\text{-}of\ (X\ \tau) \in dom(heap(snd\ \tau))\rfloor\rfloor$
$\quad else\ invalid\ \tau)$

The following predicates — which are not part of the OCL standard descriptions — complete the goal of oclIsNew() by describing where an object belongs.

definition *OclIsOld*:: $('\mathfrak{A},\ '\alpha{::}\{null,object\})val \Rightarrow ('\mathfrak{A})Boolean \quad ((\text{-}).oclIsOld'('))$
where $X\ .oclIsOld() \equiv (\lambda\tau\ .\ if\ (\delta\ X)\ \tau = true\ \tau$
$\quad then\ \lfloor\lfloor oid\text{-}of\ (X\ \tau) \in dom(heap(fst\ \tau)) \wedge$
$\quad\quad oid\text{-}of\ (X\ \tau) \notin dom(heap(snd\ \tau))\rfloor\rfloor$
$\quad else\ invalid\ \tau)$

definition *OclIsEverywhere*:: $('\mathfrak{A},\ '\alpha{::}\{null,object\})val \Rightarrow ('\mathfrak{A})Boolean \quad ((\text{-}).oclIsEverywhere'('))$
where $X\ .oclIsEverywhere() \equiv (\lambda\tau\ .\ if\ (\delta\ X)\ \tau = true\ \tau$
$\quad then\ \lfloor\lfloor oid\text{-}of\ (X\ \tau) \in dom(heap(fst\ \tau)) \wedge$
$\quad\quad oid\text{-}of\ (X\ \tau) \in dom(heap(snd\ \tau))\rfloor\rfloor$
$\quad else\ invalid\ \tau)$

definition *OclIsAbsent*:: $('\mathfrak{A},\ '\alpha{::}\{null,object\})val \Rightarrow ('\mathfrak{A})Boolean \quad ((\text{-}).oclIsAbsent'('))$

where X *.oclIsAbsent*() $\equiv (\lambda\tau\ .\ \mathit{if}\ (\delta\ X)\ \tau = \mathit{true}\ \tau$
$\quad \mathit{then}\ \lfloor\lfloor \mathit{oid\text{-}of}\ (X\ \tau) \notin \mathit{dom}(\mathit{heap}(\mathit{fst}\ \tau)) \wedge$
$\quad\quad \mathit{oid\text{-}of}\ (X\ \tau) \notin \mathit{dom}(\mathit{heap}(\mathit{snd}\ \tau))\rfloor\rfloor$
$\quad \mathit{else\ invalid}\ \tau)$

lemma *state-split* : $\tau \models \delta\ X \Longrightarrow \tau \models (X\ .\mathit{oclIsNew}()) \vee \tau \models (X\ .\mathit{oclIsOld}()) \vee \tau \models (X\ .\mathit{oclIsEverywhere}()) \vee \tau \models (X\ .\mathit{oclIsAbsent}())$
by(*simp add*: *OclIsOld-def OclIsNew-def OclIsEverywhere-def OclIsAbsent-def OclValid-def true-def*, *blast*)

lemma *notNew-vs-others* : $\tau \models \delta\ X \Longrightarrow (\neg\ \tau \models (X\ .\mathit{oclIsNew}())) = (\tau \models (X\ .\mathit{oclIsOld}()) \vee \tau \models (X\ .\mathit{oclIsEverywhere}()) \vee \tau \models (X\ .\mathit{oclIsAbsent}()))$
by(*simp add*: *OclIsOld-def OclIsNew-def OclIsEverywhere-def OclIsAbsent-def OclNot-def OclValid-def true-def*, *blast*)

5.3.4. OclIsModifiedOnly

The following predicate — which is not part of the OCL standard descriptions — 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 transistion that DOES NOT CHANGE is of premordial importance. The following operator establishes the equality between old and new objects in the state (provided that they exist in both states), with the exception of those objects

definition *OclIsModifiedOnly* $::('\mathfrak{A}::\mathit{object},'\alpha::\{\mathit{null},\mathit{object}\})\mathit{Set} \Rightarrow {}'\mathfrak{A}\ \mathit{Boolean}$
$(\text{-}{-}{>}\mathit{oclIsModifiedOnly}'(')$)
where $X{-}{>}\mathit{oclIsModifiedOnly}() \equiv (\lambda(\sigma,\sigma').\ \ \mathit{let}\ \ X' = (\mathit{oid\text{-}of}\ `\ \lceil\lceil \mathit{Rep\text{-}Set\text{-}0}(X(\sigma,\sigma'))\rceil\rceil);$
$\quad S = ((\mathit{dom}\ (\mathit{heap}\ \sigma) \cap \mathit{dom}\ (\mathit{heap}\ \sigma')) - X')$
$\quad \mathit{in\ if}\ (\delta\ X)\ (\sigma,\sigma') = \mathit{true}\ (\sigma,\sigma')$
$\quad\quad \mathit{then}\ \lfloor\lfloor \forall\ x \in S.\ (\mathit{heap}\ \sigma)\ x = (\mathit{heap}\ \sigma')\ x\rfloor\rfloor$
$\quad\quad \mathit{else\ invalid}\ (\sigma,\sigma'))$

lemma *cp-OclIsModifiedOnly* : $X{-}{>}\mathit{oclIsModifiedOnly}()\ \tau = (\lambda\text{-}.\ X\ \tau){-}{>}\mathit{oclIsModifiedOnly}()\ \tau$
by(*simp only*: *OclIsModifiedOnly-def*, *case-tac* τ, *simp only*:, *subst cp-defined*, *simp*)

definition [*simp*]: *OclSelf x H fst-snd* $= (\lambda\tau\ .\ \mathit{if}\ (\delta\ x)\ \tau = \mathit{true}\ \tau$
$\quad \mathit{then\ if}\ \mathit{oid\text{-}of}\ (x\ \tau) \in \mathit{dom}(\mathit{heap}(\mathit{fst}\ \tau)) \wedge \mathit{oid\text{-}of}\ (x\ \tau) \in \mathit{dom}(\mathit{heap}\ (\mathit{snd}\ \tau))$
$\quad\quad \mathit{then}\ \ H\ \lceil(\mathit{heap}(\mathit{fst\text{-}snd}\ \tau))(\mathit{oid\text{-}of}\ (x\ \tau))\rceil$
$\quad\quad \mathit{else\ invalid}\ \tau$
$\quad \mathit{else\ invalid}\ \tau)$

definition *OclSelf-at-pre* $::\ ('\mathfrak{A}::\mathit{object},'\alpha::\{\mathit{null},\mathit{object}\})\mathit{val} \Rightarrow$
$('\mathfrak{A} \Rightarrow {}'\alpha) \Rightarrow$
$('\mathfrak{A}::\mathit{object},'\alpha::\{\mathit{null},\mathit{object}\})\mathit{val}\ ((\text{-})@\mathit{pre}(\text{-}))$
where $x\ @\mathit{pre}\ H = \mathit{OclSelf}\ x\ H\ \mathit{fst}$

definition *OclSelf-at-post* $::\ ('\mathfrak{A}::\mathit{object},'\alpha::\{\mathit{null},\mathit{object}\})\mathit{val} \Rightarrow$

$('\mathfrak{A} \Rightarrow '\alpha) \Rightarrow$
$('\mathfrak{A}::object,'\alpha::\{null,object\})val$ ((-)@*post*(-))
where *x* @*post* *H* = *OclSelf x H snd*

theorem *framing*:
assumes *modifiesclause*:$\tau \models (X->excluding(x :: ('\mathfrak{A}::object,'\alpha::\{null,object\})val))->oclIsModifiedOnly()$
and *represented-x*: $\tau \models \delta(x$ @*pre* $(H::('\mathfrak{A} \Rightarrow '\alpha)))$
and *oid-is-typerepr* : *inj-on* (*oid-of* :: $'\alpha \Rightarrow$ -) (*insert* $(x\ \tau)$ $\lceil\lceil Rep\text{-}Set\text{-}0\ (X\ \tau)\rceil\rceil$)
shows $\tau \models (x$ @*pre* $H \triangleq (x$ @*post* $H))$
proof −
have *def-x* : $\tau \models \delta\ x$
by(*insert represented-x*, *simp add*: *defined-def OclValid-def null-fun-def bot-fun-def false-def true-def OclSelf-at-pre-def invalid-def split*: *split-if-asm*)
show *?thesis*
apply(*simp add*:*StrongEq-def OclValid-def true-def OclSelf-at-pre-def OclSelf-at-post-def def-x*[*simplified OclValid-def*])
apply(*rule conjI*, *rule impI*)
apply(*rule-tac f* = $\lambda x.\ H\ \lceil x\rceil$ **in** *arg-cong*)
apply(*insert modifiesclause*[*simplified OclIsModifiedOnly-def OclValid-def*])
apply(*case-tac* τ, *rename-tac* $\sigma\ \sigma'$, *simp split*: *split-if-asm*)
apply(*simp add*: *OclExcluding-def*)
apply(*drule foundation5*[*simplified OclValid-def true-def*], *simp*)
apply(*subst* (*asm*) *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 x* = *oid-of* $(x\ (\sigma, \sigma'))$ **in** *ballE*) **apply** *simp*
apply(*subst* (*asm*) *inj-on-image-set-diff*[**where** *C* = *insert* $(x\ (\sigma, \sigma'))$ $\lceil\lceil Rep\text{-}Set\text{-}0\ (X\ (\sigma, \sigma'))\rceil\rceil$], *simp add*: *oid-is-typerepr*)
apply (*metis* (*hide-lams*, *no-types*) *inj-on-insert oid-is-typerepr*)
apply (*metis subset-insertI*)
apply(*simp add*: *invalid-def bot-option-def*)+

apply(*blast*)
done
qed

lemma *pre-post-new*: $\tau \models (x\ .oclIsNew()) \Longrightarrow \neg\ (\tau \models \upsilon(x$ @*pre* $H1)) \land \neg\ (\tau \models \upsilon(x$ @*post* $H2))$
by(*simp add*: *OclIsNew-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-old*: $\tau \models (x\ .oclIsOld()) \Longrightarrow \neg\ (\tau \models \upsilon(x$ @*pre* $H1)) \land \neg\ (\tau \models \upsilon(x$ @*post* $H2))$
by(*simp add*: *OclIsOld-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-absent*: $\tau \models (x\ .oclIsAbsent()) \Longrightarrow \neg\ (\tau \models \upsilon(x\ @pre\ H1)) \wedge \neg\ (\tau \models \upsilon(x\ @post\ H2))$
by(*simp add*: *OclIsAbsent-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-everywhere*: $(\tau \models \upsilon(x\ @pre\ H1) \vee \tau \models \upsilon(x\ @post\ H2)) \Longrightarrow \tau \models (x\ .oclIsEverywhere())$
by(*simp add*: *OclIsEverywhere-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-everywhere'*: $\tau \models (x\ .oclIsEverywhere()) \Longrightarrow (\tau \models \upsilon(x\ @pre\ (Some\ o\ H1)) \wedge \tau \models \upsilon(x\ @post\ (Some\ o\ H2)))$
by(*simp add*: *OclIsEverywhere-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*: $(\sigma, \sigma) \models (x\ @pre\ H \triangleq (x\ @post\ H))$
by(*simp add*: *OclSelf-at-pre-def OclSelf-at-post-def OclValid-def StrongEq-def*)

end

theory *OCL-tools*
imports *OCL-core*
begin

end

theory *OCL-main*
imports *OCL-lib OCL-state OCL-tools*
begin

end

Part III.

Conclusion

6. Conclusion

6.1. Lessons Learned

While our paper and pencil arguments, given in [6], 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 [23] or SMT-solvers like Z3 [14] completely impractical. Concretely, if the expression `not(null)` is defined `invalid` (as is the case in the present standard [21]), 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."

Featherweight OCL makes these two deviations from the standard, builds all logical operators on Kleene-`not` and Kleene-`and`, and shows that the entire construction of our paper "Extending OCL with Null-References" [6] is then correct, and the DNF-normaliation 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 [5] for details) are valid in Featherweight OCL.

6.2. 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 [10]. There are essentially five extensions necessary:

- extension of the library to support all OCL data types, e. g., `Sequence(T)`, `OrderedSet(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 [22]) 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 [5]. 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 [23] based animator in Isabelle can give a similar quality of animation as the OCLexec Tool [16]
- 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 2.3 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

[1] P. B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Kluwer Academic Publishers, Dordrecht, 2nd edition, 2002.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. pages 49–69, May 25.

[3] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. C. Paulson, editors, *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer-Verlag, 2010.

[4] A. D. Brucker. *An Interactive Proof Environment for Object-oriented Specifications*. PhD thesis, ETH Zurich, Mar. 2007. ETH Dissertation No. 17097.

[5] A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff. A specification-based test case generation method for UML/OCL. In J. Dingel and A. Solberg, editors, *MoDELS Workshops*, number 6627 in Lecture Notes in Computer Science, pages 334–348. Springer-Verlag, 2010. Selected best papers from all satellite events of the MoDELS 2010 conference. Workshop on OCL and Textual Modelling.

[6] A. D. Brucker, M. P. Krieger, and B. Wolff. Extending OCL with null-references. In S. Gosh, editor, *Models in Software Engineering*, number 6002 in Lecture Notes in Computer Science, pages 261–275. Springer-Verlag, 2009. Selected best papers from all satellite events of the MoDELS 2009 conference.

[7] A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.

[8] A. D. Brucker and B. Wolff. HOL-OCL – A Formal Proof Environment for UML/OCL. In J. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering (FASE08)*, number 4961 in Lecture Notes in Computer Science, pages 97–100. Springer-Verlag, 2008.

[9] A. D. Brucker and B. Wolff. An extensible encoding of object-oriented data models in HOL. *Journal of Automated Reasoning*, 41:219–249, 2008.

[10] A. D. Brucker and B. Wolff. HOL-TestGen: An interactive test-case generation framework. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering (FASE09)*, number 5503 in Lecture Notes in Computer Science, pages 417–420. Springer-Verlag, 2009.

[11] A. D. Brucker and B. Wolff. Semantics, calculi, and analysis for object-oriented specifications. *Acta Informatica*, 46(4):255–284, July 2009.

[12] A. D. Brucker and B. Wolff. Featherweight ocl: A study for the consistent semantics of ocl 2.3 in hol. In *Workshop on OCL and Textual Modelling (OCL 2012)*, 2012.

[13] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[14] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Heidelberg, 2008. Springer-Verlag.

[15] P. Kosiuczenko. Specification of invariability in OCL. pages 676–691.

[16] M. P. Krieger, A. Knapp, and B. Wolff. Generative programming and component engineering. In E. Visser and J. Järvi, editors, *International Conference on Generative Programming and Component Engineering (GPCE 2010)*, pages 53–62. ACM, Oct. 2010.

[17] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2002.

[18] Object constraint language specification (version 1.1), Sept. 1997. Available as OMG document ad/97-08-08.

[19] UML 2.0 OCL specification, Oct. 2003. Available as OMG document ptc/03-10-14.

[20] UML 2.0 OCL specification, Apr. 2006. Available as OMG document formal/06-05-01.

[21] UML 2.3.1 OCL specification, Feb. 2012. Available as OMG document formal/2012-01-01.

[22] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[23] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647, Heidelberg, 2007. Springer-Verlag.

[24] M. Wenzel and B. Wolff. Building formal method tools in the Isabelle/Isar framework. In K. Schneider and J. Brandt, editors, *TPHOLS 2007*, number 4732 in Lecture Notes in Computer Science, pages 352–367. Springer-Verlag, Heidelberg, 2007.

[25] M. M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, TU München, München, Feb. 2002.