

Extended Version

Featherweight OCL

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

Achim D. Brucker

Delphine Longuet
Burkhart Wolff

Frédéric Tuong

December 7, 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

I. Introduction	11
1. Motivation	13
2. Background	15
2.1. Formal Foundation	15
2.1.1. Isabelle	15
2.1.2. Higher-order logic	16
2.1.3. Specification Constructs in Isabelle/HOL	18
2.2. Featherweight OCL: Design Goals	18
2.3. The Theory Organization	19
2.3.1. Denotational Semantics	19
2.3.2. Logical Layer	21
2.3.3. Algebraic Layer	23
2.4. A Machine-checked Annex A	26
II. A Formal Semantics of OCL 2.3 in Isabelle/HOL	29
2.5. Formal and Technical Background	31
2.5.1. Validity and Evaluations	31
2.5.2. Strict Operations	31
2.5.3. Boolean Operators	32
2.5.4. Object-oriented Data Structures	33
2.5.5. The Accessors	33
2.6. A Proposal for an OCL 2.1 Semantics	34
2.6.1. Revised Operations on Primitive Types	34
2.6.2. Null in Class Types	36
2.6.3. Revised Accessors	37
2.6.4. Other Operations on States	38
2.7. Attribute Values	39
2.7.1. Single-Valued Attributes	39
2.7.2. Collection-Valued Attributes	39
2.7.3. The Precise Meaning of Multiplicity Constraints	40

3. Part I: Core Definitions	41
3.1. Preliminaries	41
3.1.1. Notations for the option type	41
3.1.2. Minimal Notions of State and State Transitions	41
3.1.3. Prerequisite: An Abstract Interface for OCL Types	42
3.1.4. Accomodation of Basic Types to the Abstract Interface	42
3.1.5. The Semantic Space of OCL Types: Valuations.	44
3.2. Boolean Type and Logic	44
3.2.1. Basic Constants	45
3.2.2. Fundamental Predicates I: Validity and Definedness	45
3.2.3. Fundamental Predicates II: Logical (Strong) Equality	48
3.2.4. Fundamental Predicates III	49
3.2.5. Logical Connectives and their Universal Properties	49
3.3. A Standard Logical Calculus for OCL	55
3.3.1. Global vs. Local Judgements	55
3.3.2. Local Validity and Meta-logic	56
3.3.3. Local Judgements and Strong Equality	59
3.3.4. Laws to Establish Definedness (δ -closure)	61
3.4. Miscellaneous: OCL's if then else endif	62
4. Part II: Library Definitions	65
4.1. Basic Types: Void, Integer	65
4.1.1. The construction of the Void Type	65
4.1.2. The construction of the Integer Type	65
4.1.3. Validity and Definedness Properties	66
4.1.4. Arithmetical Operations on Integer	66
4.2. Fundamental Predicates on Boolean and Integer: Strict Equality	68
4.2.1. Definition	68
4.2.2. Logic and Algebraic Layer on Basic Types	69
4.2.3. Test Statements on Basic Types.	72
4.3. Complex Types: The Set-Collection Type (I) Core	73
4.3.1. The construction of the Set Type	73
4.3.2. Validity and Definedness Properties	74
4.3.3. Constants on Sets	76
4.4. Complex Types: The Set-Collection Type (II) Library	77
4.4.1. Computational Operations on Set	77
4.4.2. Validity and Definedness Properties	80
4.4.3. Execution with invalid or null as argument	84
4.4.4. Context Passing	86
4.5. Fundamental Predicates on Set: Strict Equality	87
4.5.1. Definition	87
4.5.2. Logic and Algebraic Layer on Set	88
4.6. Execution on Set's Operators	89
4.6.1. OclIncluding	89

4.6.2.	OclExcluding	94
4.6.3.	OclSize	100
4.6.4.	OclIsEmpty	105
4.6.5.	OclNotEmpty	106
4.6.6.	Ocl Any	106
4.6.7.	OclForall	109
4.6.8.	OclExists	116
4.6.9.	OclIterate	116
4.6.10.	OclSelect	119
4.6.11.	Strict Equality	119
4.7.	Test Statements	123
5.	Part III: State Operations and Objects	125
5.1.	Complex Types: The Object Type (I) Core	125
5.1.1.	Recall: The generic structure of States	125
5.2.	Fundamental Predicates on Object: Strict Equality	126
5.2.1.	Definition	126
5.2.2.	Logic and Algebraic Layer on Object	126
5.3.	Complex Types: The Object Type (II) Library	127
5.3.1.	Initial States (for Testing and Code Generation)	127
5.3.2.	OclAllInstances	127
5.3.3.	OclIsNew	131
5.3.4.	OclIsModifiedOnly	132
6.	Part IV: OCL Contracts and an Example	139
6.0.5.	Introduction	139
6.0.6.	Outlining the Example	139
6.1.	Example Data-Universe and its Infrastructure	139
6.2.	Instantiation of the generic strict equality. We instantiate the referential equality on <i>Person</i> and <i>OclAny</i>	141
6.3.	OclAsType	141
6.3.1.	Definition	141
6.3.2.	Context Passing	142
6.3.3.	Execution with invalid or null as argument	143
6.4.	OclIsTypeOf	143
6.4.1.	Definition	143
6.4.2.	Context Passing	144
6.4.3.	Execution with invalid or null as argument	145
6.4.4.	up down cast	146
6.5.	OclIsKindOf	147
6.5.1.	Definition	147
6.5.2.	Context Passing	148
6.5.3.	Execution with invalid or null as argument	148
6.5.4.	up down cast	149

6.6.	OclAllInstances	150
6.6.1.	IsTypeOf	150
6.6.2.	IsKindOf	151
6.7.	dot boss, dot salary	152
6.7.1.	Definition (of the association Employee-Boss)	152
6.7.2.	Context Passing	155
6.7.3.	Execution with invalid or null as argument	155
6.8.	A little infra-structure on example states.	156
7.	Part IV (bis): OCL Contracts and an Example	165
7.1.	Standard State Infrastructure	165
7.2.	Invariant	165
7.3.	The contract of a recursive query :	166
7.4.	The contract of a method.	166
8.	Part V: OCL Contracts and an Example	169
8.0.1.	Introduction	169
8.0.2.	Outlining the Example	169
8.1.	Example Data-Universe and its Infrastructure	169
8.2.	Instantiation of the generic strict equality. We instantiate the referential equality on <i>Person</i> and <i>OclAny</i>	171
8.3.	OclAsType	171
8.3.1.	Definition	171
8.3.2.	Context Passing	172
8.3.3.	Execution with invalid or null as argument	173
8.4.	OclIsTypeOf	173
8.4.1.	Definition	173
8.4.2.	Context Passing	174
8.4.3.	Execution with invalid or null as argument	175
8.4.4.	up down cast	176
8.5.	OclIsKindOf	177
8.5.1.	Definition	177
8.5.2.	Context Passing	178
8.5.3.	Execution with invalid or null as argument	178
8.5.4.	up down cast	179
8.6.	OclAllInstances	180
8.6.1.	IsTypeOf	180
8.6.2.	IsKindOf	181
8.7.	dot boss, dot salary	182
8.7.1.	Definition	182
8.7.2.	Context Passing	183
8.7.3.	Execution with invalid or null as argument	184
8.8.	A little infra-structure on example states.	185

9. Part V (bis): OCL Contracts and an Example	193
9.1. Standard State Infrastructure	193
9.2. Invariant	193
9.3. The contract of a recursive query :	194
9.4. The contract of a method.	194
 III. Conclusion	 197
10. Conclusion	199
10.1. Lessons Learned	199
10.2. Conclusion and Future Work	199

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 $A_1 \Longrightarrow \dots \Longrightarrow A_n \Longrightarrow A_{n+1}$, which are viewed as a *rule* of the form “from assumptions A_1 to A_n , infer conclusion A_{n+1} ” and which is written in Isabelle as

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

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

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

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

```

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

(2.3)

This proof script instructs Isabelle to prove ϕ by case distinction followed by a simplification of the resulting proof state. Such a proof state is an implicitly conjoint sequence of generalized Horn-clauses (called *subgoals*) ϕ_1, \dots, ϕ_n and a *goal* ϕ . Proof states were

usually denoted by:

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

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

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

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

As another example of a conservative extension, typed sets were built in the Isabelle libraries conservatively on top of the kernel of HOL as functions to `bool`; consequently,

the constant definitions for membership is as follows:¹

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

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

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

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

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

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

respectively

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

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

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

¹To 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:

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

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

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 logical 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 $(\alpha_{\perp})_{\perp}$ is an instance of this type-class with $\text{bot} \equiv \perp$ and $\text{null} \equiv \perp_{\perp}$. Now, any OCL type can be represented by an HOL type of the form:

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

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

$$\begin{aligned} I[\![\text{invalid} :: V(\alpha)]\!] \tau &\equiv \text{bot} & I[\![\text{null} :: V(\alpha)]\!] \tau &\equiv \text{null} \\ I[\![\text{true} :: \text{Boolean}]\!] \tau &= \lfloor \text{true} \rfloor & I[\![\text{false}]\!] \tau &= \lfloor \text{false} \rfloor \end{aligned}$$

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

$$\begin{aligned} I[\![X.\text{oclIsValid}()]\!] \tau &= \\ &(\text{if } I[\![X]\!] \tau = \text{bot} \text{ then } I[\![\text{true}]\!] \tau \text{ else } I[\![\text{false}]\!] \tau) \end{aligned}$$

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 δX for `not X.oclIsUndefined()` and $v X$ for `not X.oclIsValid()` throughout this paper.

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

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

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

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

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

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:

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

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

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

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:

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

It enjoys nearly the laws of a congruence:

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

where the predicate **cp** stands for *context-passing*, a property that is 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.:

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

Together with the general case-distinction

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

which is possible for any OCL type, a case distinction on the variables in a formula can be performed; due to strictness rules, formulae containing somewhere a variable x that is known to be **invalid** or **null** reduce usually quickly to contradictions. For example, we can infer from an invariant $\tau \models x \doteq y-3$ that we have actually $\tau \models x \doteq y-3 \wedge \tau \models \delta x \wedge \tau \models \delta y$. We call the latter formula the δ -closure of the former. Now, we can convert a formula like $\tau \models x > 0 \ \mathbf{or} \ 3*y > x*x$ 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:

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

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

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

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 `_ + _`):

$$\begin{aligned}
\text{invalid} + x &= \text{invalid} & x + \text{invalid} &= \text{invalid} \\
x + \text{null} &= \text{invalid} & \text{null} + x &= \text{invalid} \\
\text{null.asType}(X) &= \text{invalid}
\end{aligned}$$

besides “classical” exceptional behavior:

$$\begin{aligned} 1 / 0 &= \text{invalid} & 1 / \text{null} &= \text{invalid} \\ \text{null} \rightarrow \text{isEmpty}() &= \text{true} \end{aligned}$$

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:

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

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:

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

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

²In 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.:

```

δ Set{} = true
δ (X->including(x)) = δX and δx
Set{}->includes(x) = (if v x then false
                      else invalid endif)
(X->including(x)->includes(y)) =
  (if δ X
   then if x ≐ y
        then true
        else X->includes(y)
        endif
   else invalid
   endif)

```

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

<code>Set{}->including(1)->including(2)</code>
--

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:

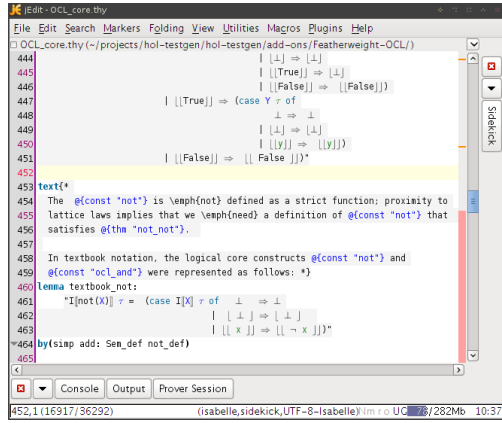
value $\tau \models (\text{Set}\{\text{Set}\{2, \text{null}\}\} \doteq \text{Set}\{\text{Set}\{\text{null}, 2\}\})$

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

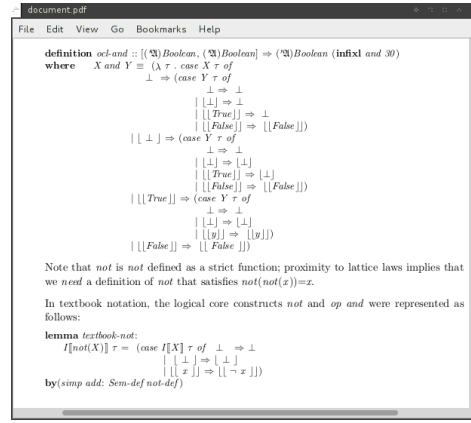
2.4. A Machine-checked Annex A

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 $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ -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*:

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

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:

$$(\sigma, \sigma') \models P \equiv (P(\sigma, \sigma') = \text{true}).$$

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 τ_\perp . All types of expressions are of a form captured by

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

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

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

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:

$$I[\text{not } X] \tau \equiv \begin{cases} \neg I[X] \tau & \text{if } I[X] \tau \neq \perp, \\ \perp & \text{otherwise.} \end{cases}$$

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., $\text{not } _$ are HOL functions (in this case of HOL type $V(\text{bool}) \Rightarrow V(\text{bool})$) that can be viewed as *transformers on evaluations*.

The binary case of the integer addition is analogous:

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

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

$$I\llbracket X \text{+union}(Y) \rrbracket \tau \equiv \begin{cases} S\llbracket I\llbracket X \rrbracket \tau \sqcup I\llbracket Y \rrbracket \tau \rrbracket & \text{if } I\llbracket X \rrbracket \tau \neq \perp \text{ and } I\llbracket Y \rrbracket \tau \neq \perp, \\ \perp & \text{otherwise.} \end{cases}$$

Here, S (“smash”) is a function that maps a lifted set $\llbracket X \rrbracket$ to \perp if and only if $\perp \in X$ 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:

$$\begin{aligned} \text{Boolean} &:= V(\text{bool}), & \alpha \text{ Set} &:= V(\alpha \text{ set}), \\ \text{Integer} &:= V(\text{int}), \text{ and} & \alpha \text{ Sequence} &:= V(\alpha \text{ list}). \end{aligned}$$

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:

$$I\llbracket \text{true} \rrbracket \tau \equiv \llbracket \text{true} \rrbracket, \quad I\llbracket \text{false} \rrbracket \tau \equiv \llbracket \text{false} \rrbracket, \text{ and} \quad I\llbracket \text{invalid} \rrbracket \tau \equiv \perp.$$

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:

$$I\llbracket X \triangleq Y \rrbracket \tau \equiv (I\llbracket X \rrbracket \tau = I\llbracket Y \rrbracket \tau),$$

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

$$I\llbracket X \text{ and } Y \rrbracket \tau \equiv \begin{cases} \llbracket x \rrbracket \wedge \llbracket y \rrbracket & \text{if } x \neq \perp \text{ and } y \neq \perp, \\ \llbracket \text{false} \rrbracket & \text{if } x = \llbracket \text{false} \rrbracket \text{ or } y = \llbracket \text{false} \rrbracket, \\ \perp & \text{otherwise.} \end{cases}$$

where $x = I\llbracket X \rrbracket \tau$ and $y = I\llbracket Y \rrbracket \tau$. The other Boolean connectives were just shortcuts: $X \text{ or } Y \equiv \text{not}(\text{not } X \text{ and not } Y)$ and $X \text{ implies } Y \equiv \text{not } X \text{ or } Y$.

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, \dots, 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 $_a :: C_i \rightarrow B$ and $_a@pre :: C_i \rightarrow B$ for $a \in A_{C_i}$ and $B \in \{V(\dots), C_1, \dots, C_n\}$), type casts that can change the static type of an object of a class (denoted $_[C_i]$ of type $C_j \rightarrow C_i$) and two dynamic type tests (denoted $isType_{C_i} -$ and $isKind_{C_i} -$). 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: $C := (\text{oid} \times C_t \times A_1 \times \dots \times A_k)$ where a unique tag-type C_t (ensuring type-safety) is created for each class type, where the types A_1, \dots, 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, \dots, C_n , we define states as partial functions from *oid*’s to object representations satisfying a *state invariant* inv_σ :

$$\text{state} := \{f :: \text{oid} \rightarrow (C_1 + \dots + C_n) \mid inv_\sigma(f)\}$$

where $inv_\sigma(f)$ states two conditions: 1) there is no object representation for `nullid`: `nullid` $\notin (\text{dom } f)$. 2) there is a “one-to-one” correspondence between object representations and *oid*’s: $\forall oid \in \text{dom } f. oid = \text{OidOf } \lceil f(oid) \rceil$. 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):

$$I[\![self.a]\!](\sigma, \sigma') \equiv \begin{cases} \perp & \text{if } O = \perp \vee \text{OidOf } \ulcorner O \urcorner \notin \text{dom } \sigma' \\ \text{get}_D u & \text{if } \sigma'(\text{get}_C \ulcorner \sigma'(\text{OidOf } \ulcorner O \urcorner) \urcorner . a^{(0)}) = \ulcorner u \urcorner, \\ \perp & \text{otherwise.} \end{cases}$$

$$I[\![self.a@pre]\!](\sigma, \sigma') \equiv \begin{cases} \perp & \text{if } O = \perp \vee \text{OidOf } \ulcorner O \urcorner \notin \text{dom } \sigma \\ \text{get}_D u & \text{if } \sigma(\text{get}_C \ulcorner \sigma(\text{OidOf } \ulcorner O \urcorner) \urcorner . a) = \ulcorner u \urcorner, \\ \perp & \text{otherwise.} \end{cases}$$

where $O = I[\![self]\!](\sigma, \sigma')$. Here, get_D is the projection function from the object universe to D_\perp , and $x.a$ 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 (get_D may yield \perp 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 $\text{dom } \sigma$, 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 $(\text{invalid} \doteq X) \triangleq \text{invalid}$.

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:

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

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

$$I[\![\text{invalid}]\!] \tau \equiv \perp \text{ and } I[\![\text{null}_{\text{Integer}}]\!] \tau \equiv \perp_{\perp}.$$

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

$$I[\![X + Y]\!] \tau \equiv \begin{cases} \perp_{\perp} \text{ if } x \neq \perp, y \neq \perp, \lceil x \rceil \neq \perp \text{ and } \lceil y \rceil \neq \perp \\ \perp \text{ otherwise.} \end{cases}$$

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 $_ \doteq _$, $_.\text{oclIsInvalid}()$, $_.\text{oclIsUndefined}()$, 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 $\text{null}_{\text{Boolean}} \text{ and false} \triangleq \text{false}$ and $\text{null}_{\text{Boolean}} \text{ and true} \triangleq \perp$.

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

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

where the test $x .\text{oclIsUndefined}()$ covers two cases: $x .\text{oclIsInvalid}()$ and $x \doteq \text{null}$ (i.e., x is invalid or null). As a consequence, for the inverse case $(X + Y) .\text{oclIsInvalid}()$ 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 $\text{not } (X + Y \doteq \text{null}_{\text{Integer}})$), 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 $\text{null}_{\text{Set}(T)}$ collection as well as collections that contain null values (such as $\text{Set}\{\text{null}_T\}$) but never invalid.

³The same holds for $(X + Y) .\text{oclIsUndefined}()$.

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, \dots, A_n . Then we define its null object representation by:*

$$I[\llbracket \text{null}_{C_i} \rrbracket \tau] \equiv \llbracket (\text{nullid}, \text{arb}_t, a_1, \dots, a_n) \rrbracket$$

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 `nullT`-elements are not equal, but isomorphic: Each type, has its own unique `nullT`-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.

⁴In 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 $C_1 + \dots + C_n + \alpha$. 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:

$$I\llbracket obj.a \rrbracket(\sigma, \sigma') \equiv \begin{cases} \perp & \text{if } I\llbracket obj \rrbracket(\sigma, \sigma') = \perp \vee \text{OidOf}\ulcorner I\llbracket obj \rrbracket(\sigma, \sigma') \urcorner \notin \text{dom } \sigma' \\ \text{null}_D & \text{if } \text{get}_C\ulcorner \sigma'(\text{OidOf}\ulcorner I\llbracket obj \rrbracket(\sigma, \sigma') \urcorner) \urcorner.a^{(0)} = \text{nullid} \\ \text{get}_D u & \text{if } \sigma'(\text{get}_C\ulcorner \sigma'(\text{OidOf}\ulcorner I\llbracket obj \rrbracket(\sigma, \sigma') \urcorner) \urcorner.a^{(0)}) = \ulcorner u \urcorner, \\ \perp & \text{otherwise.} \end{cases}$$

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:

$$\begin{aligned} \text{invalid}.a &\triangleq \text{invalid} & \text{invalid}_{[C]} &\triangleq \text{invalid} \\ & & \text{isType}_C \text{ invalid} &\triangleq \text{invalid} \end{aligned}$$

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.

$$\begin{aligned} \text{null}_A.a &\triangleq \text{invalid} & \text{null}_{A[B]} &\triangleq \text{null}_B \\ & & \text{isType}_A \text{ null}_A &\triangleq \text{true} \end{aligned}$$

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 $T.\text{allInstances}() \rightarrow \text{excludes}(\text{null})$ 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

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

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

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

where $X' = I[X](\sigma, \sigma')$ and $M = (\text{dom } \sigma \cap \text{dom } \sigma') - \{\text{OidOf } x \mid x \in \lceil X \rceil\}$. 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 $\tau \models X \rightarrow \text{modifiedOnly}()$ and $\tau \models X \rightarrow \text{excludes}(s.a)$, we can infer that $\tau \models s.a = s.a \text{ @pre}$ (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

⁵We 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 ( $\lfloor(-)\rfloor$ )
notation None ( $\perp$ )
```

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

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

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 ('A)state =
  heap   :: oid → 'A
  assocs2 :: oid → (oid × oid) list
```

$$assoc_3 :: oid \rightarrow (oid \times oid \times oid) \text{ list}$$

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

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

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

instance *option* $:: (plus) \text{ plus by intro-classes}$
instance *fun* $:: (type, plus) \text{ plus by intro-classes}$

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

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

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)  $\equiv$  (None::'a option)
instance proof show  $\exists x::'a \text{ option}. x \neq \text{bot}$ 
  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)  $\equiv$  [ bot ]
  instance proof show (null::'a::bot option)  $\neq$  bot
    by( simp add:null-option-def bot-option-def)
    qed
end

```

```

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

  instance proof show  $\exists (x::'a \Rightarrow 'b). x \neq \text{bot}$ 
    apply(rule-tac x= $\lambda -. (\text{SOME } y. y \neq \text{bot})$  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  $\Rightarrow$  'b::null)  $\equiv$  ( $\lambda x. \text{null}$ )

  instance proof
    show (null::'a  $\Rightarrow$  'b::null)  $\neq$  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 \mathcal{A}) to an arbitrary null-type (i.e. containing at least a distinguished *null* and *invalid* element).

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

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

definition $\text{Sem} :: 'a \Rightarrow 'a \ (I[-])$
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 $\text{invalid} :: (\mathcal{A}, \alpha::\text{bot}) \text{ val}$
where $\text{invalid} \equiv \lambda \tau. \text{bot}$

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

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

Note that the definition :

definition $\text{null} :: ('\<AA>, '\<alpha>::\text{null}) \text{ val}$
where $\text{"null"} \ \<equiv> \ \<lambda> \ \<tau>. \text{null}"$

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

lemma *textbook-null-fun*: $I[\text{null}::(\mathcal{A}, \alpha::\text{null}) \text{ val}] \tau = (\text{null}::\alpha::\text{null})$
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 $(\mathcal{A}) \text{ Boolean} = (\mathcal{A}, \text{bool option option}) \text{ val}$

3.2.1. Basic Constants

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

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

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

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

lemma *bool-split*: $X \tau = \text{invalid } \tau \vee X \tau = \text{null } \tau \vee$
 $X \tau = \text{true } \tau \vee X \tau = \text{false } \tau$
apply(*simp add: invalid-def null-def true-def false-def*)
apply(*case-tac X \tau, 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*) = $\lfloor \text{False} \rfloor$
by(*simp add: false-def*)

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

lemma *textbook-true*: $I \llbracket \text{true} \rrbracket \tau = \lfloor \text{True} \rfloor$
by(*simp add: Sem-def true-def*)

lemma *textbook-false*: $I \llbracket \text{false} \rrbracket \tau = \lfloor \text{False} \rfloor$
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* :: ('A, 'a::null)val \Rightarrow ('A)Boolean (*v* - [100]100)
where *v X* $\equiv \lambda \tau. \text{if } X \tau = \text{bot } \tau \text{ then false } \tau \text{ else true } \tau$

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

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

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

invalid-def true-def false-def)

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

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]*: $v \text{ true} = true$

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]*: $v \text{ false} = true$

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*: $(v \ X) \ \tau = (v \ (\lambda \ -. \ X \ \tau)) \ \tau$

by(*simp add: valid-def*)

definition *defined* :: $(\mathfrak{A}, 'a::null)val \Rightarrow (\mathfrak{A})Boolean \ (\delta - [100]100)$

where $\delta \ X \equiv \lambda \ \tau . \text{ if } X \ \tau = bot \ \tau \ \vee \ X \ \tau = null \ \tau \text{ then false } \tau \text{ else true } \tau$

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

lemma *defined1[simp]*: $\delta \ invalid = false$

by(*rule ext,simp add: defined-def bot-fun-def bot-option-def
null-def invalid-def true-def false-def*)

lemma *defined2[simp]*: $\delta \ null = false$

by(*rule ext,simp add: defined-def bot-fun-def bot-option-def
null-def null-option-def null-fun-def invalid-def true-def false-def*)

lemma *defined3[simp]*: $\delta \ true = true$

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

lemma *defined6[simp]: $\delta v X = \text{true}$*
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]: $v v X = \text{true}$*
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]: $v \delta X = \text{true}$*
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: $(\delta X)\tau = (\delta (\lambda -. X \tau)) \tau$*
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: $I[\delta(X)] \tau = (\text{if } I[X] \tau = I[\text{bot}] \tau \vee I[X] \tau = I[\text{null}] \tau$*
then $I[\text{false}] \tau$
else $I[\text{true}] \tau$)
by(*simp add: Sem-def defined-def*)

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

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

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

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

Name	Theorem
<i>defined1</i>	$\delta\ invalid = false$
<i>defined2</i>	$\delta\ null = false$
<i>defined3</i>	$\delta\ true = true$
<i>defined4</i>	$\delta\ false = true$
<i>defined5</i>	$\delta\ \delta\ ?X = true$
<i>defined6</i>	$\delta\ v\ ?X = true$

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

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

lemma *StrongEq-subst* :
assumes *cp*: $\bigwedge X. P(X)\tau = P(\lambda \cdot. X \tau)\tau$
and *eq*: $(X \triangleq Y)\tau = \text{true } \tau$
shows $(P X \triangleq P Y)\tau = \text{true } \tau$
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]*: $\delta (X \triangleq Y) = \text{true}$
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]*: $v (X \triangleq Y) = \text{true}$
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*: $(X \triangleq Y) \tau = ((\lambda \cdot. X \tau) \triangleq (\lambda \cdot. Y \tau)) \tau$
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 than having the truth-table aside when reading fails its purpose in our view.

Practically, this means that we want to give a definition to the core operations to be as close as possible to the lattice laws; this makes also powerful symbolic 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 *not* $X \equiv \lambda \tau . \text{case } X \text{ } \tau \text{ of}$

$$\begin{array}{lcl} \perp & \Rightarrow & \perp \\ | \lfloor \perp \rfloor & \Rightarrow & \lfloor \perp \rfloor \\ | \lfloor \lfloor x \rfloor \rfloor & \Rightarrow & \lfloor \lfloor \neg x \rfloor \rfloor \end{array}$$

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

definition *OclAnd* :: [(\mathfrak{A})Boolean, (\mathfrak{A})Boolean] \Rightarrow (\mathfrak{A})Boolean (**infixl** and 30)

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

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

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

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

lemma *textbook-OclNot*:

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

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

lemma *textbook-OclAnd*:

$$I[X \text{ and } Y] \tau = (\text{case } I[X] \tau \text{ of } \begin{array}{l} \perp \Rightarrow (\text{case } I[Y] \tau \text{ of } \\ \quad \perp \Rightarrow \perp \\ \quad | \lfloor \perp \rfloor \Rightarrow \perp \\ \quad | \lfloor \lfloor \text{True} \rfloor \rfloor \Rightarrow \perp \\ \quad | \lfloor \lfloor \text{False} \rfloor \rfloor \Rightarrow \lfloor \lfloor \text{False} \rfloor \rfloor) \\ | \lfloor \perp \rfloor \Rightarrow (\text{case } I[Y] \tau \text{ of } \\ \quad \perp \Rightarrow \perp \\ \quad | \lfloor \perp \rfloor \Rightarrow \lfloor \perp \rfloor \\ \quad | \lfloor \lfloor \text{True} \rfloor \rfloor \Rightarrow \lfloor \perp \rfloor \\ \quad | \lfloor \lfloor \text{False} \rfloor \rfloor \Rightarrow \lfloor \lfloor \text{False} \rfloor \rfloor) \\ | \lfloor \lfloor \text{True} \rfloor \rfloor \Rightarrow (\text{case } I[Y] \tau \text{ of } \\ \quad \perp \Rightarrow \perp \\ \quad | \lfloor \perp \rfloor \Rightarrow \lfloor \perp \rfloor \\ \quad | \lfloor \lfloor y \rfloor \rfloor \Rightarrow \lfloor \lfloor y \rfloor \rfloor) \\ | \lfloor \lfloor \text{False} \rfloor \rfloor \Rightarrow \lfloor \lfloor \text{False} \rfloor \rfloor) \end{array})$$

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

definition *OclOr* :: $((\mathfrak{A})\text{Boolean}, (\mathfrak{A})\text{Boolean}) \Rightarrow (\mathfrak{A})\text{Boolean}$ (**infixl** or 25)

where $X \text{ or } Y \equiv \text{not}(\text{not } X \text{ and } \text{not } Y)$

definition *OclImplies* :: $((\mathfrak{A})\text{Boolean}, (\mathfrak{A})\text{Boolean}) \Rightarrow (\mathfrak{A})\text{Boolean}$ (**infixl** implies 25)

where $X \text{ implies } Y \equiv \text{not } X \text{ or } Y$

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

by(*simp add: OclAnd-def*)

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

apply(*simp add: OclOr-def*)

apply(*subst cp-OclNot[of not $(\lambda \neg. X \tau)$ and not $(\lambda \neg. Y \tau)$]*)

apply(*subst cp-OclAnd[of not $(\lambda \neg. X \tau)$ not $(\lambda \neg. Y \tau)$]*)

by(*simp add: cp-OclNot[symmetric] cp-OclAnd[symmetric]*)

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

apply(*simp add: OclImplies-def*)

apply(*subst cp-OclOr[of not $(\lambda \neg. X \tau)$ $(\lambda \neg. Y \tau)$]*)

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

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

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

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

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

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

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

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

lemma OclAnd-assoc: ( $X$  and ( $Y$  and  $Z$ )) = ( $X$  and  $Y$  and  $Z$ )
  apply(rule ext, simp add: OclAnd-def)
  apply(auto simp:true-def false-def null-def invalid-def)

```

split: option.split option.split-asm
bool.split bool.split-asm)

done

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

by(rule *ext,simp add: OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def bot-option-def*)

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

by(rule *ext,simp add: OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def bot-option-def*)

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

by(rule *ext,simp add: OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def bot-option-def*

null-fun-def null-option-def)

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

by(rule *ext,simp add: OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def bot-option-def*)

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

by(rule *ext,simp add: OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def bot-option-def*

null-fun-def null-option-def)

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

by(rule *ext,simp add: OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def bot-option-def*

null-fun-def null-option-def)

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

by(rule *ext,simp add: OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def bot-option-def*

null-fun-def null-option-def)

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

by(rule *ext,simp add: OclOr-def OclNot-def OclAnd-def null-def invalid-def true-def false-def bot-option-def*

null-fun-def null-option-def)

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

by(*simp add: OclOr-def*)

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

by(*simp add: OclOr-def OclAnd-commute*)

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

by(*simp add: OclOr-def*)

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

by(*simp add: OclOr-def*)

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

```

by(simp add: OclOr-def)

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

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

lemma OclOr-null1[simp]:  $\bigwedge \tau. X \ \tau \neq \text{true} \ \tau \implies X \ \tau \neq \text{bot } \tau \implies (\text{null or } X) \ \tau = \text{null } \tau$ 
  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]:  $\bigwedge \tau. X \ \tau \neq \text{true} \ \tau \implies X \ \tau \neq \text{bot } \tau \implies (X \text{ or null}) \ \tau = \text{null } \tau$ 
  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* :: $[(\mathfrak{A})st, (\mathfrak{A})Boolean] \Rightarrow \text{bool} \ ((I(-)/ \models (-)) \ 50)$
where $\tau \models P \equiv ((P \ \tau) = \text{true } \tau)$

3.3.1. Global vs. Local Judgements

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

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

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

lemma *transform2-rev*: $\forall \tau. (\tau \models \delta P) \wedge (\tau \models \delta Q) \wedge (\tau \models P) = (\tau \models Q) \implies P = Q$
apply(*rule ext, auto simp: OclValid-def true-def defined-def*)
apply(*erule-tac x=a in allE*)
apply(*erule-tac x=b in allE*)
apply(*auto simp: false-def true-def defined-def bot-Boolean-def null-Boolean-def*
split: option.split-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 $H : P = \text{true} \implies Q = \text{true}$
shows $\tau \models P \implies \tau \models Q$
apply(*simp add: OclValid-def*)
apply(*rule H[THEN fun-cong]*)
apply(*rule ext*)
oops

3.3.2. Local Validity and Meta-logic

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

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

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

lemma *foundation4*[*simp*]: $\neg(\tau \models \text{null})$
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*]:
 $(\tau \models (x \triangleq \text{invalid})) \vee (\tau \models (x \triangleq \text{null})) \vee (\tau \models (x \triangleq \text{true})) \vee (\tau \models (x \triangleq \text{false}))$
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*:
 $(\tau \models \delta x) = ((\neg(\tau \models (x \triangleq \text{invalid}))) \wedge (\neg(\tau \models (x \triangleq \text{null}))))$
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*:

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

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*:

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

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*]:

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

by(*simp add: OclNot-def OclValid-def true-def false-def defined-def*
split: option.split option.split-asm)

lemma *foundation7'*[*simp*]:

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

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*:

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

proof –

have 1 : $(\tau \models \delta x) \vee (\neg (\tau \models \delta x))$ **by** *auto*

have 2 : $(\neg (\tau \models \delta x)) = ((\tau \models (x \triangleq \text{invalid})) \vee (\tau \models (x \triangleq \text{null})))$

by(*simp only: def-split-local, simp*)

show ?thesis **by**(*insert 1, simp add:2*)

qed

lemma *foundation9*:

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

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*:

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

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*:

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

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*:

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

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

by(*auto simp: OclNot-def OclValid-def invalid-def true-def null-def StrongEq-def
split:bool.split-asm bool.split*)

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

by(*auto simp: OclNot-def OclValid-def invalid-def false-def true-def null-def StrongEq-def
split:bool.split-asm bool.split option.split*)

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

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

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

by(*auto simp: OclValid-def valid-def false-def true-def bot-fun-def invalid-def
split:split-if-asm*)

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

by(*auto simp: OclValid-def valid-def false-def true-def bot-fun-def
split:split-if-asm*)

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

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

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

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

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

lemmas *cp-validity=foundation23*

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

lemma *valid-not-I* : $\tau \models v (x) \implies \tau \models v (\text{not } x)$
by(*auto simp: 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* : $\tau \models \delta (x) \implies \tau \models \delta (y) \implies \tau \models \delta (x \text{ and } y)$
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* : $\tau \models v (x) \implies \tau \models v (y) \implies \tau \models v (x \text{ and } y)$
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-reft*: $\tau \models (x \triangleq x)$
by(*simp add: OclValid-def StrongEq-def*)

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

lemma *StrongEq-L-trans*: $\tau \models (x \triangleq y) \implies \tau \models (y \triangleq z) \implies \tau \models (x \triangleq z)$
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:
 $(\forall X \tau. f X \tau = f(\lambda-. X \tau) \tau) \Longrightarrow cp P \Longrightarrow cp(\lambda X. f (P X))$
apply(auto simp: true-def cp-def)
apply(rule exI, (rule allI)+)
by(erule-tac x=P X in allE, auto)

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

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

3.3.4. Laws to Establish Definedness (δ -closure)

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

```

lemma OclNot-defargs:
   $\tau \models (not P) \implies \tau \models \delta P$ 
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 $OclIf :: [(A)Boolean, (A, \alpha :: null) val, (A, \alpha) val] \Rightarrow (A, \alpha) val$
 $(if (-) then (-) else (-) endif [10,10,10] 50)$

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

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

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 $OclIf-true' [simp]: \tau \models P \implies (if P then B_1 else B_2 endif) \tau = B_1 \tau$
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 $OclIf-false' [simp]: \tau \models not P \implies (if P then B_1 else B_2 endif) \tau = B_2 \tau$
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 $OclIf-idem2[simp]: (if v X then A else A endif) = A$
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

4.1.1. The construction of the Void Type

```
type-synonym ('A) Void = ('A, unit option) 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 ('A) Integer = ('A, int option option) val
```

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

```
definition OclInt0 :: ('A) Integer (0)
where      0 = (λ - . [| 0 :: int |])
```

```
definition OclInt1 :: ('A) Integer (1)
where      1 = (λ - . [| 1 :: int |])
```

```
definition OclInt2 :: ('A) Integer (2)
where      2 = (λ - . [| 2 :: int |])
```

```
definition OclInt3 :: ('A) Integer (3)
where      3 = (λ - . [| 3 :: int |])
```

```
definition OclInt4 :: ('A) Integer (4)
where      4 = (λ - . [| 4 :: int |])
```

```
definition OclInt5 :: ('A) Integer (5)
where      5 = (λ - . [| 5 :: int |])
```

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

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

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

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

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

4.1.3. Validity and Definedness Properties

lemma $\delta(\text{null}::(\mathcal{A})\text{Integer}) = \text{false}$ **by** *simp*

lemma $v(\text{null}::(\mathcal{A})\text{Integer}) = \text{true}$ **by** *simp*

lemma [*simp,code-unfold*]: $\delta(\lambda -. \llbracket n \rrbracket) = \text{true}$
by(*simp add:defined-def true-def*
bot-fun-def bot-option-def null-fun-def null-option-def)

lemma [*simp,code-unfold*]: $v(\lambda -. \llbracket n \rrbracket) = \text{true}$
by(*simp add:valid-def true-def*
bot-fun-def bot-option-def)

lemma [*simp,code-unfold*]: $\delta \mathbf{0} = \text{true}$ **by**(*simp add:OclInt0-def*)

lemma [*simp,code-unfold*]: $v \mathbf{0} = \text{true}$ **by**(*simp add:OclInt0-def*)

lemma [*simp,code-unfold*]: $\delta \mathbf{1} = \text{true}$ **by**(*simp add:OclInt1-def*)

lemma [*simp,code-unfold*]: $v \mathbf{1} = \text{true}$ **by**(*simp add:OclInt1-def*)

lemma [*simp,code-unfold*]: $\delta \mathbf{2} = \text{true}$ **by**(*simp add:OclInt2-def*)

lemma [*simp,code-unfold*]: $v \mathbf{2} = \text{true}$ **by**(*simp add:OclInt2-def*)

lemma [*simp,code-unfold*]: $\delta \mathbf{6} = \text{true}$ **by**(*simp add:OclInt6-def*)

lemma [*simp,code-unfold*]: $v \mathbf{6} = \text{true}$ **by**(*simp add:OclInt6-def*)

lemma [*simp,code-unfold*]: $\delta \mathbf{8} = \text{true}$ **by**(*simp add:OclInt8-def*)

lemma [*simp,code-unfold*]: $v \mathbf{8} = \text{true}$ **by**(*simp add:OclInt8-def*)

lemma [*simp,code-unfold*]: $\delta \mathbf{9} = \text{true}$ **by**(*simp add:OclInt9-def*)

lemma [*simp,code-unfold*]: $v \mathbf{9} = \text{true}$ **by**(*simp add:OclInt9-def*)

4.1.4. Arithmetical Operations on Integer

Definition

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

Note that we can not follow the lexis of 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 $OclAdd_{Integer} :: ('A)Integer \Rightarrow ('A)Integer \Rightarrow ('A)Integer$ (**infix** $+_{ocl}$ 40)
where $x +_{ocl} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \wedge (\delta y) \tau = \text{true} \wedge$
 $\text{then } [[\llbracket x \tau \rrbracket] + \llbracket y \tau \rrbracket]]$
 $\text{else } \text{invalid } \tau$

definition $OclLess_{Integer} :: ('A)Integer \Rightarrow ('A)Integer \Rightarrow ('A)Boolean$ (**infix** $<_{ocl}$ 40)
where $x <_{ocl} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \wedge (\delta y) \tau = \text{true} \wedge$
 $\text{then } [[\llbracket x \tau \rrbracket < \llbracket y \tau \rrbracket]]$
 $\text{else } \text{invalid } \tau$

definition $OclLe_{Integer} :: ('A)Integer \Rightarrow ('A)Integer \Rightarrow ('A)Boolean$ (**infix** \leq_{ocl} 40)
where $x \leq_{ocl} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \wedge (\delta y) \tau = \text{true} \wedge$
 $\text{then } [[\llbracket x \tau \rrbracket \leq \llbracket y \tau \rrbracket]]$
 $\text{else } \text{invalid } \tau$

abbreviation $OclAdd_{Integer}$ (**infix** $+_I$ 40) **where** $x +_I y \equiv x +_{ocl} y$
abbreviation $OclLess_{Integer}$ (**infix** $<_I$ 40) **where** $x <_I y \equiv x <_{ocl} y$
abbreviation $OclLe_{Integer}$ (**infix** \leq_I 40) **where** $x \leq_I y \equiv x \leq_{ocl} y$

Basic properties

lemma $OclAdd_{Integer}\text{-commute}$: $(X +_{ocl} Y) = (Y +_{ocl} X)$
by(*rule ext, auto simp: true-def false-def OclAdd_{Integer}-def invalid-def*
 $\text{split: option.split option.split-asm}$
 $\text{bool.split bool.split-asm}$)

Execution with invalid or null or zero as argument

lemma $OclAdd_{Integer}\text{-strict1}$ [*simp, code-unfold*]: $(x +_{ocl} \text{invalid}) = \text{invalid}$
by(*rule ext, simp add: OclAdd_{Integer}-def true-def false-def*)

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

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

apply(*rule ext, rename-tac τ*)
proof – **fix** τ **show** $(x +_I \mathbf{0}) \tau = (\text{if } v \ x \text{ and not } (\delta x) \text{ then invalid else } x \text{ endif}) \tau$
apply(*case-tac $(v \ x \text{ and not } (\delta x)) \tau = \text{true } \tau$*)
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  $\tau$ )
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 OclAddInteger-zero2[simp,code-unfold] : (0 +ocl x) = (if v x and not ( $\delta$  x) then invalid
else x endif)
by(subst OclAddInteger-commute, simp)

```

Context Passing

```

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

```

```

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

```

```

lemma cp-OclLeInteger:(X  $\leq_{ocl}$  Y)  $\tau$  = (( $\lambda$  -. X  $\tau$ )  $\leq_{ocl}$  ( $\lambda$  -. Y  $\tau$ ))  $\tau$ 
by(simp add: OclLeInteger-def cp-defined[symmetric])

```

Test Statements

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

```

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

```

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 :: (('A,'a)val,('A,'a)val)  $\Rightarrow$  ('A)Boolean (infixl  $\doteq$  30)

```

syntax

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

translations

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

defs $\text{StrictRefEq}_{\text{Boolean}}[\text{code-unfold}] :$

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

defs $\text{StrictRefEq}_{\text{Integer}}[\text{code-unfold}] :$

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

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

by $(\text{simp add: StrictRefEq}_{\text{Boolean}})$

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

by $(\text{simp add: StrictRefEq}_{\text{Boolean}})$

value $\tau \models 1 <> 2$

value $\tau \models 2 <> 1$

value $\tau \models 2 \doteq 2$

value $\tau \models \text{true} <> \text{false}$

value $\tau \models \text{false} <> \text{true}$

4.2.2. Logic and Algebraic Layer on Basic Types

Validity and Definedness Properties (I)

lemma $\text{StrictRefEq}_{\text{Boolean}}\text{-defined-args-valid}:$

$(\tau \models \delta((x :: ('A) \text{Boolean}) \doteq y)) = ((\tau \models (v\ x)) \wedge (\tau \models (v\ y)))$

by $(\text{auto simp: StrictRefEq}_{\text{Boolean}} \text{OclValid-def true-def valid-def false-def StrongEq-def}$
 $\quad \text{defined-def invalid-def null-fun-def bot-fun-def null-option-def bot-option-def}$
 $\quad \text{split: bool.split-asm HOL.split-if-asm option.split})$

lemma $\text{StrictRefEq}_{\text{Integer}}\text{-defined-args-valid}:$

$(\tau \models \delta((x :: ('A) \text{Integer}) \doteq y)) = ((\tau \models (v\ x)) \wedge (\tau \models (v\ y)))$

by $(\text{auto simp: StrictRefEq}_{\text{Integer}} \text{OclValid-def true-def valid-def false-def StrongEq-def}$
 $\quad \text{defined-def invalid-def null-fun-def bot-fun-def null-option-def bot-option-def}$
 $\quad \text{split: bool.split-asm HOL.split-if-asm option.split})$

Validity and Definedness Properties (II)

lemma $\text{StrictRefEq}_{\text{Boolean}}\text{-defargs}:$

$\tau \models ((x :: ('A) \text{Boolean}) \doteq y) \implies (\tau \models (v\ x)) \wedge (\tau \models (v\ y))$

by $(\text{simp add: StrictRefEq}_{\text{Boolean}} \text{OclValid-def true-def invalid-def}$
 $\quad \text{bot-option-def}$
 $\quad \text{split: bool.split-asm HOL.split-if-asm})$

lemma *StrictRefEqInteger-defargs:*
 $\tau \models ((x::(\mathfrak{A})Integer) \doteq y) \implies (\tau \models (v\ x)) \wedge (\tau \models (v\ y))$
by(simp add: StrictRefEqInteger OclValid-def true-def invalid-def valid-def bot-option-def
split: bool.split-asm HOL.split-if-asm)

Validity and Definedness Properties (III) Miscellaneous

lemma *StrictRefEqBoolean-strict''* : $\delta\ ((x::(\mathfrak{A})Boolean) \doteq y) = (v(x) \text{ and } v(y))$
by(auto intro!: transform2-rev defined-and-I simp: foundation10 StrictRefEqBoolean-defined-args-valid)

lemma *StrictRefEqInteger-strict''* : $\delta\ ((x::(\mathfrak{A})Integer) \doteq y) = (v(x) \text{ and } v(y))$
by(auto intro!: transform2-rev defined-and-I simp: foundation10 StrictRefEqInteger-defined-args-valid)

lemma *StrictRefEqInteger-strict* :
assumes *A*: $v\ (x::(\mathfrak{A})Integer) = \text{true}$
and *B*: $v\ y = \text{true}$
shows $v\ (x \doteq y) = \text{true}$
apply(insert *A B*)
apply(rule ext, simp add: StrongEq-def StrictRefEqInteger true-def valid-def defined-def
bot-fun-def bot-option-def)
done

lemma *StrictRefEqInteger-strict'* :
assumes *A*: $v\ (((x::(\mathfrak{A})Integer)) \doteq y) = \text{true}$
shows $v\ x = \text{true} \wedge v\ y = \text{true}$
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 StrictRefEqInteger
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 *StrictRefEqBoolean-refl[simp,code-unfold]* :
 $((x::(\mathfrak{A})Boolean) \doteq x) = (\text{if } (v\ x) \text{ then true else invalid endif})$
by(rule ext, simp add: StrictRefEqBoolean OclIf-def)

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

Execution with invalid or null as argument

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

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

lemma *true-non-null* [simp,code-unfold]:(true \doteq null) = false
apply(rule ext, simp add: StrictRefEq_{Boolean} StrongEq-def false-def)
by (metis defined3 foundation1 foundation16 null-fun-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 *false-non-null* [simp,code-unfold]:(false \doteq null) = false
apply(rule ext, simp add: StrictRefEq_{Boolean} StrongEq-def false-def)
by (metis OCL-core.drop.simps null-Boolean-def option.distinct(1))

lemma *null-non-false* [simp,code-unfold]:(null \doteq false) = false
apply(rule ext, simp add: StrictRefEq_{Boolean} StrongEq-def false-def)
by (metis OCL-core.drop.simps null-Boolean-def option.distinct(1))

lemma *integer-non-null* [simp]: ((λ -. [|n|]) \doteq (null::('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::('A)Integer) \doteq (λ -. [|n|])) = 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]: (0 \doteq null) = false **by**(simp add: OclInt0-def)
lemma *null-non-OclInt0* [simp,code-unfold]: (null \doteq 0) = false **by**(simp add: OclInt0-def)
lemma *OclInt1-non-null* [simp,code-unfold]: (1 \doteq null) = false **by**(simp add: OclInt1-def)
lemma *null-non-OclInt1* [simp,code-unfold]: (null \doteq 1) = false **by**(simp add: OclInt1-def)
lemma *OclInt2-non-null* [simp,code-unfold]: (2 \doteq null) = false **by**(simp add: OclInt2-def)
lemma *null-non-OclInt2* [simp,code-unfold]: (null \doteq 2) = false **by**(simp add: OclInt2-def)
lemma *OclInt6-non-null* [simp,code-unfold]: (6 \doteq null) = false **by**(simp add: OclInt6-def)
lemma *null-non-OclInt6* [simp,code-unfold]: (null \doteq 6) = false **by**(simp add: OclInt6-def)
lemma *OclInt8-non-null* [simp,code-unfold]: (8 \doteq null) = false **by**(simp add: OclInt8-def)
lemma *null-non-OclInt8* [simp,code-unfold]: (null \doteq 8) = false **by**(simp add: OclInt8-def)
lemma *OclInt9-non-null* [simp,code-unfold]: (9 \doteq null) = false **by**(simp add: OclInt9-def)
lemma *null-non-OclInt9* [simp,code-unfold]: (null \doteq 9) = false **by**(simp add: OclInt9-def)

Behavior vs StrongEq

lemma *StrictRefEq_{Boolean}-vs-StrongEq*:

$\tau \models (v\ x) \implies \tau \models (v\ y) \implies (\tau \models (((x::(\mathfrak{A})\ Boolean) \doteq y) \triangleq (x \triangleq y)))$

apply(simp add: *StrictRefEq_{Boolean} OclValid-def*)

apply(subst *cp-StrongEq*)**back**

by simp

lemma *StrictRefEq_{Integer}-vs-StrongEq*:

$\tau \models (v\ x) \implies \tau \models (v\ y) \implies (\tau \models (((x::(\mathfrak{A})\ Integer) \doteq y) \triangleq (x \triangleq y)))$

apply(simp add: *StrictRefEq_{Integer} OclValid-def*)

apply(subst *cp-StrongEq*)**back**

by simp

Context Passing

lemma *cp-StrictRefEq_{Boolean}*:

$((X::(\mathfrak{A})\ Boolean) \doteq Y) \tau = ((\lambda \cdot. X\ \tau) \doteq (\lambda \cdot. Y\ \tau))\ \tau$

by(auto simp: *StrictRefEq_{Boolean} StrongEq-def defined-def valid-def cp-defined[symmetric]*)

lemma *cp-StrictRefEq_{Integer}*:

$((X::(\mathfrak{A})\ Integer) \doteq Y) \tau = ((\lambda \cdot. X\ \tau) \doteq (\lambda \cdot. Y\ \tau))\ \tau$

by(auto simp: *StrictRefEq_{Integer} StrongEq-def valid-def cp-defined[symmetric]*)

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

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 $\tau \models v(true)$

value $\tau \models \delta(false)$

value $\neg(\tau \models \delta(null))$

value $\neg(\tau \models \delta(invalid))$

value $\tau \models v((null::(\mathfrak{A})\ Boolean))$

value $\neg(\tau \models v(invalid))$

value $\tau \models (true\ and\ true)$

value $\tau \models (true\ and\ true \triangleq true)$

value $\tau \models ((null\ or\ null) \triangleq null)$

value $\tau \models ((\text{null or null}) \doteq \text{null})$
value $\tau \models ((\text{true} \triangleq \text{false}) \triangleq \text{false})$
value $\tau \models ((\text{invalid} \triangleq \text{false}) \triangleq \text{false})$
value $\tau \models ((\text{invalid} \doteq \text{false}) \triangleq \text{invalid})$

Elementary computations on Integer

value $\tau \models v \ 4$
value $\tau \models \delta \ 4$
value $\tau \models v \ (\text{null}::('A)\text{Integer})$
value $\tau \models (\text{invalid} \triangleq \text{invalid})$
value $\tau \models (\text{null} \triangleq \text{null})$
value $\tau \models (4 \triangleq 4)$
value $\neg(\tau \models (9 \triangleq 10))$
value $\neg(\tau \models (\text{invalid} \triangleq 10))$
value $\neg(\tau \models (\text{null} \triangleq 10))$
value $\neg(\tau \models (\text{invalid} \doteq (\text{invalid}::('A)\text{Integer})))$
value $\neg(\tau \models v \ (\text{invalid} \doteq (\text{invalid}::('A)\text{Integer})))$
value $\neg(\tau \models (\text{invalid} <> (\text{invalid}::('A)\text{Integer})))$
value $\neg(\tau \models v \ (\text{invalid} <> (\text{invalid}::('A)\text{Integer})))$
value $\tau \models (\text{null} \doteq (\text{null}::('A)\text{Integer}))$
value $\tau \models (\text{null} \doteq (\text{null}::('A)\text{Integer}))$
value $\tau \models (4 \doteq 4)$
value $\neg(\tau \models (4 <> 4))$
value $\neg(\tau \models (4 \doteq 10))$
value $\tau \models (4 <> 10)$
value $\neg(\tau \models (0 <_{ocl} \text{null}))$
value $\neg(\tau \models (\delta \ (0 <_{ocl} \text{null})))$

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

4.3.1. The construction of the Set Type

no-notation *None* (\perp)

notation *bot* (\perp)

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 $\text{Set}(\text{Set}(\text{Sequences}(\text{Pairs}(X, Y))))$).

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

The core of an own type construction is done via a type definition which provides the raw-type $'\alpha \text{ Set-0}$. It is shown that this type "fits" indeed into the abstract type interface discussed in the previous section.

```

typedef  $'\alpha \text{ Set-0} = \{X :: ('a :: \text{null}) \text{ set option option}.$ 
     $X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$ 
    by (rule-tac  $x = \text{bot}$  in  $exI$ , simp)

instantiation  $\text{Set-0} :: (\text{null})\text{bot}$ 
begin

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

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

```

```

instantiation  $\text{Set-0} :: (\text{null})\text{null}$ 
begin

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

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

```

4.3.2. Validity and Definedness Properties

Every element in a defined set is valid.

```

lemma Set-inv-lemma:  $\tau \models (\delta X) \implies \forall x \in \llbracket \text{Rep-Set-0 } (X \ \tau) \rrbracket. x \neq \text{bot}$ 
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 :  $e \in \llbracket \text{Rep-Set-0 } (X \tau) \rrbracket$ 
    shows  $\tau \models v (\lambda \cdot e)$ 
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-Set-0  $\llbracket \llbracket \text{Rep-Set-0 } (S \tau) \rrbracket \rrbracket = S \tau$ 
proof –
  have discr-eq-false-true :  $\bigwedge \tau. (\text{false } \tau = \text{true } \tau) = \text{False}$  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 =  $\lambda S. (\text{if } S = \perp \tau \vee S = \text{null } \tau \text{ then false } \tau$ 
  else true  $\tau) = \text{true } \tau \longrightarrow \text{Abs-Set-0 } \llbracket \llbracket \text{Rep-Set-0 } S \rrbracket \rrbracket = S$ ]])
    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 :  $(\tau :: \mathfrak{A} \text{ st}) \models \delta S$ 
    shows  $\exists S'. (\lambda a. (-::\mathfrak{A} \text{ st}). a) \llbracket \llbracket \text{Rep-Set-0 } (S \tau) \rrbracket \rrbracket = (\lambda a. (-::\mathfrak{A} \text{ st}). \llbracket a \rrbracket) \llbracket S' \rrbracket$ 
apply(rule-tac x =  $(\lambda a. \llbracket a \rrbracket) \llbracket \llbracket \text{Rep-Set-0 } (S \tau) \rrbracket \rrbracket$  in exI)
apply(simp only: image-comp[symmetric])
apply(simp add: comp-def)
apply(subgoal-tac  $\forall x \in \llbracket \llbracket \text{Rep-Set-0 } (S \tau) \rrbracket \rrbracket. \llbracket x \rrbracket = x$ )
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 =  $\lambda x \tau. \llbracket x \rrbracket$  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]: $\delta(\text{invalid}::(\mathfrak{A},'\alpha::\text{null}) \text{ Set}) = \text{false}$  by simp
lemma null-set-OclNot-defined [simp,code-unfold]: $\delta(\text{null}::(\mathfrak{A},'\alpha::\text{null}) \text{ Set}) = \text{false}$ 
by(simp add: defined-def null-fun-def)
lemma invalid-set-valid [simp,code-unfold]: $v(\text{invalid}::(\mathfrak{A},'\alpha::\text{null}) \text{ Set}) = \text{false}$ 
by simp
lemma null-set-valid [simp,code-unfold]: $v(\text{null}::(\mathfrak{A},'\alpha::\text{null}) \text{ Set}) = \text{true}$ 
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 $(\mathfrak{A},(\mathfrak{A},(\mathfrak{A}) \text{ Integer}) \text{ Set}) \text{ Set}$ corresponding exactly to $\text{Set}(\text{Set}(\text{Integer}))$ in OCL notation. Note that the parameter \mathfrak{A} still refers to the object universe; making the OCL semantics entirely parametric in the object universe makes it possible to study (and prove) its properties independently from a concrete class diagram.

4.3.3. Constants on Sets

```

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

```

```

lemma mtSet-defined[simp,code-unfold]: $\delta(\text{Set}\{\}) = \text{true}$ 
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-valid[simp,code-unfold]: $v(\text{Set}\{\}) = \text{true}$ 
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-rep-set:  $[[\text{Rep-Set-0 } (\text{Set}\{\}) \tau]] = \{\}$ 
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* :: $[(\mathcal{A}, \alpha :: \text{null}) \text{ Set}, (\mathcal{A}, \alpha) \text{ val}] \Rightarrow (\mathcal{A}, \alpha) \text{ Set}$
where $\text{OclIncluding } x \ y = (\lambda \tau. \text{ if } (\delta \ x) \ \tau = \text{true} \ \tau \wedge (v \ y) \ \tau = \text{true} \ \tau$
 $\quad \text{then Abs-Set-0 } [\llbracket \text{Rep-Set-0 } (x \ \tau) \rrbracket \cup \{y \ \tau\} \rrbracket$
 $\quad \text{else } \perp)$

notation *OclIncluding* $(\rightarrow \text{including}'(-))$

syntax

-OclFinset :: $\text{args} \Rightarrow (\mathcal{A}, \alpha :: \text{null}) \text{ Set} \quad (\text{Set}\{-\})$

translations

$\text{Set}\{x, xs\} == \text{CONST } \text{OclIncluding } (\text{Set}\{xs\}) \ x$
 $\text{Set}\{x\} == \text{CONST } \text{OclIncluding } (\text{Set}\{\}) \ x$

definition *OclExcluding* :: $[(\mathcal{A}, \alpha :: \text{null}) \text{ Set}, (\mathcal{A}, \alpha) \text{ val}] \Rightarrow (\mathcal{A}, \alpha) \text{ Set}$
where $\text{OclExcluding } x \ y = (\lambda \tau. \text{ if } (\delta \ x) \ \tau = \text{true} \ \tau \wedge (v \ y) \ \tau = \text{true} \ \tau$
 $\quad \text{then Abs-Set-0 } [\llbracket \text{Rep-Set-0 } (x \ \tau) \rrbracket - \{y \ \tau\} \rrbracket$
 $\quad \text{else } \perp)$

notation *OclExcluding* $(\rightarrow \text{excluding}'(-))$

definition *OclIncludes* :: $[(\mathcal{A}, \alpha :: \text{null}) \text{ Set}, (\mathcal{A}, \alpha) \text{ val}] \Rightarrow \mathcal{A} \text{ Boolean}$
where $\text{OclIncludes } x \ y = (\lambda \tau. \text{ if } (\delta \ x) \ \tau = \text{true} \ \tau \wedge (v \ y) \ \tau = \text{true} \ \tau$
 $\quad \text{then } \llbracket (y \ \tau) \in \llbracket \text{Rep-Set-0 } (x \ \tau) \rrbracket \rrbracket$
 $\quad \text{else } \perp)$

notation *OclIncludes* $(\rightarrow \text{includes}'(-) \ [66,65]65)$

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

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* :: $(\mathcal{A}, \alpha :: \text{null}) \text{ Set} \Rightarrow \mathcal{A} \text{ Integer}$
where $\text{OclSize } x = (\lambda \tau. \text{ if } (\delta \ x) \ \tau = \text{true} \ \tau \wedge \text{finite}(\llbracket \text{Rep-Set-0 } (x \ \tau) \rrbracket)$
 $\quad \text{then } \llbracket \text{int}(\text{card } \llbracket \text{Rep-Set-0 } (x \ \tau) \rrbracket) \rrbracket$
 $\quad \text{else } \perp)$

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* $:: ('A, 'α::null) Set \Rightarrow 'A Boolean$
where *OclIsEmpty* $x = ((v\ x\ and\ not\ (\delta\ x))\ or\ ((OclSize\ x) \doteq 0))$
notation *OclIsEmpty* $(-->isEmpty'()$ [66])

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

definition *OclAny* $:: [('A, 'α::null) Set] \Rightarrow ('A, 'α) val$
where *OclAny* $x = (\lambda\ \tau.\ if\ (v\ x)\ \tau = true\ \tau$
then if $(\delta\ x\ and\ OclNotEmpty\ x)\ \tau = true\ \tau\ then\ SOME\ y.\ y \in [[Rep-Set-0$
 $(x\ \tau)]]$
else null τ
else \perp)
notation *OclAny* $(-->any'()$)

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

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

syntax
 $-OclForall :: [('A, 'α::null) Set, id, ('A) Boolean] \Rightarrow 'A Boolean \quad ((-)->forall'(-|-'))$
translations
 $X ->forall(x\ |\ P) == CONST\ OclForall\ X\ (\%x.\ P)$

Like *OclForall*, *OclExists* is also not strict.

definition *OclExists* $:: ('A, 'α::null) Set, ('A, 'α) val \Rightarrow ('A) Boolean$
where *OclExists* $S\ P = not(OclForall\ S\ (\lambda\ X.\ not\ (P\ X)))$

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

definition *OclIterate_{Set}* $:: ('A, 'α::null) Set, ('A, 'β::null) val,$

$(\mathfrak{A}, \alpha) \text{val} \Rightarrow (\mathfrak{A}, \beta) \text{val} \Rightarrow (\mathfrak{A}, \beta) \text{val} \Rightarrow (\mathfrak{A}, \beta) \text{val}$
where $\text{OclIterate}_{\text{set}} S A F = (\lambda \tau. \text{if } (\delta S) \tau = \text{true } \tau \wedge (v A) \tau = \text{true } \tau \wedge \text{finite} \llbracket \text{Rep-Set-0 } (S \tau) \rrbracket$

$\text{then } (\text{Finite-Set.fold } (F) (A) ((\lambda a \tau. a) ' \llbracket \text{Rep-Set-0 } (S \tau) \rrbracket)) \tau$
 $\text{else } \perp$

syntax

$\text{-OclIterate} :: [(\mathfrak{A}, \alpha :: \text{null}) \text{Set}, \text{idt}, \text{idt}, \alpha, \beta] \Rightarrow (\mathfrak{A}, \gamma) \text{val}$
 $(- \rightarrow \text{iterate}'(-; \text{--} \mid -) \llbracket 71, 100, 70 \rrbracket 50)$

translations

$X \rightarrow \text{iterate}(a; x = A \mid P) == \text{CONST OclIterate}_{\text{set}} X A (\% a. (\% x. P))$

definition $\text{OclSelect}_{\text{set}} :: [(\mathfrak{A}, \alpha :: \text{null}) \text{Set}, (\mathfrak{A}, \alpha) \text{val} \Rightarrow (\mathfrak{A}) \text{Boolean}] \Rightarrow (\mathfrak{A}, \alpha) \text{Set}$

where $\text{OclSelect}_{\text{set}} S P = (\lambda \tau. \text{if } (\delta S) \tau = \text{true } \tau$
 $\text{then if } (\exists x \in \llbracket \text{Rep-Set-0 } (S \tau) \rrbracket. P(\lambda -. x) \tau = \perp \tau)$
 $\text{then } \perp$
 $\text{else Abs-Set-0 } \llbracket \{ x \in \llbracket \text{Rep-Set-0 } (S \tau) \rrbracket. P(\lambda -. x) \tau \neq \text{false } \tau \}$
 \rrbracket
 $\text{else } \perp$

syntax

$\text{-OclSelect} :: [(\mathfrak{A}, \alpha :: \text{null}) \text{Set}, \text{id}, (\mathfrak{A}) \text{Boolean}] \Rightarrow \mathfrak{A} \text{ Boolean} \quad ((-) \rightarrow \text{select}'(- \mid -))$

translations

$X \rightarrow \text{select}(x \mid P) == \text{CONST OclSelect}_{\text{set}} X (\% x. P)$

definition $\text{OclReject}_{\text{set}} :: [(\mathfrak{A}, \alpha :: \text{null}) \text{Set}, (\mathfrak{A}, \alpha) \text{val} \Rightarrow (\mathfrak{A}) \text{Boolean}] \Rightarrow (\mathfrak{A}, \alpha :: \text{null}) \text{Set}$

where $\text{OclReject}_{\text{set}} S P = \text{OclSelect}_{\text{set}} S (\text{not } o P)$

syntax

$\text{-OclReject} :: [(\mathfrak{A}, \alpha :: \text{null}) \text{Set}, \text{id}, (\mathfrak{A}) \text{Boolean}] \Rightarrow \mathfrak{A} \text{ Boolean} \quad ((-) \rightarrow \text{reject}'(- \mid -))$

translations

$X \rightarrow \text{reject}(x \mid P) == \text{CONST OclReject}_{\text{set}} X (\% x. P)$

Definition (futur operators)

consts

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

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:*

$(\tau \models \delta(X \rightarrow \text{including}(x))) = ((\tau \models (\delta \ X)) \wedge (\tau \models (v \ x)))$

proof –

```

have A :  $\perp \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$  by (simp add: bot-option-def)
have B :  $\llbracket \perp \rrbracket \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$  by (simp add: null-option-def bot-option-def)
have C :  $(\tau \models (\delta \ X)) \implies (\tau \models (v \ x)) \implies \llbracket \text{insert } (x \ \tau) \llbracket \text{Rep-Set-0 } (X \ \tau) \rrbracket \rrbracket \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$ 
  apply (frule Set-inv-lemma)
  apply (simp add: foundation18 invalid-def)
  done
have D :  $(\tau \models \delta(X \rightarrow \text{including}(x))) \implies ((\tau \models (\delta \ X)) \wedge (\tau \models (v \ x)))$ 
  by (auto simp: OclIncluding-def OclValid-def true-def valid-def false-def StrongEq-def
    defined-def invalid-def bot-fun-def null-fun-def
    split: bool.split-asm HOL.split-if-asm option.split)
have E :  $(\tau \models (\delta \ X)) \implies (\tau \models (v \ x)) \implies (\tau \models \delta(X \rightarrow \text{including}(x)))$ 
  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:*

$(\tau \models v(X \rightarrow \text{including}(x))) = ((\tau \models (\delta \ X)) \wedge (\tau \models (v \ x)))$

proof –

have D : $(\tau \models v(X \rightarrow \text{including}(x))) \implies ((\tau \models (\delta \ X)) \wedge (\tau \models (v \ x)))$


```

    by(auto simp: OclIncluding-def OclValid-def true-def valid-def false-def StrongEq-def
        defined-def invalid-def bot-fun-def null-fun-def
        split: bool.split-asm HOL.split-if-asm option.split)
  have E: ( $\tau \models (\delta X)$ )  $\implies$  ( $\tau \models (v x)$ )  $\implies$  ( $\tau \models v(X \rightarrow \text{including}(x))$ )
    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]:
 $\delta(X \rightarrow \text{including}(x)) = ((\delta X) \text{ and } (v x))$ 
by(auto intro!: transform2-rev simp:including-defined-args-valid foundation10 defined-and-I)

```

```

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

```

OclExcluding

```

lemma excluding-defined-args-valid:
( $\tau \models \delta(X \rightarrow \text{excluding}(x))$ ) = (( $\tau \models (\delta X)$ )  $\wedge$  ( $\tau \models (v x)$ ))
proof -
  have A :  $\perp \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in [[X]]. x \neq \text{bot})\}$  by(simp add: bot-option-def)
  have B :  $\lfloor \perp \rfloor \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in [[X]]. x \neq \text{bot})\}$  by(simp add: null-option-def bot-option-def)
  have C : ( $\tau \models (\delta X)$ )  $\implies$  ( $\tau \models (v x)$ )  $\implies$   $\lfloor \lfloor [\text{Rep-Set-0 } (X \ \tau)] \rfloor - \{x \ \tau\} \rfloor \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in [[X]]. x \neq \text{bot})\}$ 
    apply(frule Set-inv-lemma)
    apply(simp add: foundation18 invalid-def)
    done
  have D: ( $\tau \models \delta(X \rightarrow \text{excluding}(x))$ )  $\implies$  (( $\tau \models (\delta X)$ )  $\wedge$  ( $\tau \models (v x)$ ))
    by(auto simp: OclExcluding-def OclValid-def true-def valid-def false-def StrongEq-def
        defined-def invalid-def bot-fun-def null-fun-def
        split: bool.split-asm HOL.split-if-asm option.split)
  have E: ( $\tau \models (\delta X)$ )  $\implies$  ( $\tau \models (v x)$ )  $\implies$  ( $\tau \models \delta(X \rightarrow \text{excluding}(x))$ )
    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:
( $\tau \models v(X \rightarrow \text{excluding}(x))$ ) = (( $\tau \models (\delta X)$ )  $\wedge$  ( $\tau \models (v x)$ ))
proof -
  have D: ( $\tau \models v(X \rightarrow \text{excluding}(x))$ )  $\implies$  (( $\tau \models (\delta X)$ )  $\wedge$  ( $\tau \models (v x)$ ))

```

```

    by(auto simp: OclExcluding-def OclValid-def true-def valid-def false-def StrongEq-def
        defined-def invalid-def bot-fun-def null-fun-def
        split: bool.split-asm HOL.split-if-asm option.split)
  have E: ( $\tau \models (\delta X)$ )  $\implies$  ( $\tau \models (v x)$ )  $\implies$  ( $\tau \models v(X \rightarrow \text{excluding}(x))$ )
    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]:
 $\delta(X \rightarrow \text{excluding}(x)) = ((\delta X) \text{ and } (v x))$ 
by(auto intro!: transform2-rev simp:excluding-defined-args-valid foundation10 defined-and-I)

```

```

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

```

OclIncludes

```

lemma includes-defined-args-valid:
( $\tau \models \delta(X \rightarrow \text{includes}(x))$ ) = (( $\tau \models (\delta X)$ )  $\wedge$  ( $\tau \models (v x)$ ))
proof -
  have A: ( $\tau \models \delta(X \rightarrow \text{includes}(x))$ )  $\implies$  (( $\tau \models (\delta X)$ )  $\wedge$  ( $\tau \models (v x)$ ))
    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: ( $\tau \models (\delta X)$ )  $\implies$  ( $\tau \models (v x)$ )  $\implies$  ( $\tau \models \delta(X \rightarrow \text{includes}(x))$ )
    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:
( $\tau \models v(X \rightarrow \text{includes}(x))$ ) = (( $\tau \models (\delta X)$ )  $\wedge$  ( $\tau \models (v x)$ ))
proof -
  have A: ( $\tau \models v(X \rightarrow \text{includes}(x))$ )  $\implies$  (( $\tau \models (\delta X)$ )  $\wedge$  ( $\tau \models (v x)$ ))
    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: ( $\tau \models (\delta X)$ )  $\implies$  ( $\tau \models (v x)$ )  $\implies$  ( $\tau \models v(X \rightarrow \text{includes}(x))$ )
    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*]:
 $\delta(X \rightarrow \text{includes}(x)) = ((\delta X) \text{ and } (v x))$
by (*auto intro!*: *transform2-rev simp:includes-defined-args-valid foundation10 defined-and-I*)

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

OclNotEmpty

lemma *notempty-has-elt* : $\tau \models \delta X \implies$
 $\tau \models X \rightarrow \text{notEmpty}() \implies$
 $\exists e. e \in [[\text{Rep-Set-0 } (X \ \tau)]]$
apply (*simp add: OclNotEmpty-def OclIsEmpty-def deMorgan1 deMorgan2, drule foundation5*)
apply (*subst (asm) (2) OclNot-def,*
simp add: OclValid-def StrictRefEqInteger StrongEq-def
split: split-if-asm)
prefer 2
apply (*simp add: invalid-def bot-option-def true-def*)
apply (*simp add: OclSize-def valid-def split: split-if-asm, simp-all add: false-def true-def bot-option-def*
bot-fun-def OclInt0-def)
by (*metis equalsOI*)

Ocl Any

lemma *any-valid-args-valid* [*simp,code-unfold*]:
 $(\tau \models v(X \rightarrow \text{any}())) = (\tau \models v X)$
proof –
have *A*: $(\tau \models v(X \rightarrow \text{any}())) \implies ((\tau \models (v X)))$
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*: $(\tau \models (v X)) \implies (\tau \models v(X \rightarrow \text{any}()))$
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* (δX) $\tau = \text{true } \tau$)
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** $X \ \tau \neq \text{null} \implies$
 $(\text{SOME } y. y \in [[\text{Rep-Set-0 } (X \ \tau)]] = \perp \implies$
 $\forall x \in [[\text{Rep-Set-0 } (X \ \tau)]] . x \neq \perp \implies e \in [[\text{Rep-Set-0 } (X \ \tau)]] \implies$

False

```

    apply(subgoal-tac (SOME x. x ∈ [[Rep-Set-0 (X τ)]])) ≠ ⊥)
  prefer 2
    apply(rule someI2[where Q = λx. x ≠ ⊥ and P = λy. y ∈ [[Rep-Set-0 (X τ)]]
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]:
v(X->any()) = (v 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*:

$(X \rightarrow \text{including}(x)) \tau = ((\lambda -. X \tau) \rightarrow \text{including}(\lambda -. x \tau)) \tau$
by(*auto simp: OclIncluding-def StrongEq-def invalid-def*
cp-defined[symmetric] cp-valid[symmetric])

lemma *cp-OclExcluding*:

$(X \rightarrow \text{excluding}(x)) \tau = ((\lambda -. X \tau) \rightarrow \text{excluding}(\lambda -. x \tau)) \tau$
by(*auto simp: OclExcluding-def StrongEq-def invalid-def*
cp-defined[symmetric] cp-valid[symmetric])

lemma *cp-OclIncludes*:

$(X \rightarrow \text{includes}(x)) \tau = (\text{OclIncludes } (\lambda -. X \tau) (\lambda -. x \tau) \tau)$
by(*auto simp: OclIncludes-def StrongEq-def invalid-def*
cp-defined[symmetric] cp-valid[symmetric])

lemma *cp-OclIncludes1*:

$(X \rightarrow \text{includes}(x)) \tau = (\text{OclIncludes } X (\lambda -. x \tau) \tau)$
by(*auto simp: OclIncludes-def StrongEq-def invalid-def*
cp-defined[symmetric] cp-valid[symmetric])

lemma *cp-OclSize*: $X \rightarrow \text{size}() \tau = (\lambda -. X \tau) \rightarrow \text{size}() \tau$

by(*simp add: OclSize-def cp-defined[symmetric]*)

lemma *cp-OclIsEmpty*: $X \rightarrow \text{isEmpty}() \tau = (\lambda -. X \tau) \rightarrow \text{isEmpty}() \tau$

apply(*simp add: OclIsEmpty-def*)
apply(*subst (2) cp-OclOr*)
apply(*subst cp-OclAnd*)
apply(*subst cp-OclNot*)
apply(*subst cp-StrictRefEqInteger*)
apply(*simp add: cp-defined[symmetric] cp-valid[symmetric]*
cp-StrictRefEqInteger[symmetric] cp-OclSize[symmetric] cp-OclNot[symmetric] cp-OclAnd[symmetric]
cp-OclOr[symmetric])
done

lemma *cp-OclNotEmpty*: $X \rightarrow \text{notEmpty}() \tau = (\lambda -. X \tau) \rightarrow \text{notEmpty}() \tau$

apply(*simp add: OclNotEmpty-def*)
apply(*subst (2) cp-OclNot*)
apply(*simp add: cp-OclNot[symmetric] cp-OclIsEmpty[symmetric]*)
done

lemma *cp-OclAny*: $X \rightarrow \text{any}() \tau = (\lambda -. X \tau) \rightarrow \text{any}() \tau$

apply(*simp only: OclAny-def*)
apply(*subst (2) cp-OclAnd*)
apply(*simp only: cp-OclAnd[symmetric] cp-defined[symmetric] cp-valid[symmetric] cp-OclNotEmpty[symmetric]*)
done

lemma *cp-OclForall*:

$(S \rightarrow \text{forAll}(x \mid P x)) \tau = ((\lambda -. S \tau) \rightarrow \text{forAll}(x \mid P (\lambda -. x \tau))) \tau$

by(simp add: OclForall-def cp-defined[symmetric])

lemma cp-OclForall1 [simp,intro!]:
 cp $S \implies$ cp $(\lambda X. ((S X) \rightarrow \text{forAll}(x \mid P x)))$
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 $(\lambda X St x. P (\lambda \tau. x) X St) \implies$ cp $S \implies$ cp $(\lambda X. (S X) \rightarrow \text{forAll}(x \mid P x X))$
apply(simp only: cp-def)
oops

lemma cp-OclForall:
 cp $S \implies$
 $(\bigwedge x. cp(P x)) \implies$
 cp $(\lambda X. ((S X) \rightarrow \text{forAll}(x \mid P x X)))$
oops

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

lemmas cp-intro''[simp,intro!,code-unfold] =
 cp-intro'
 cp-OclIncluding [THEN allI[THEN allI[THEN allI[THEN cpI2]], of OclIncluding]]
 cp-OclExcluding [THEN allI[THEN allI[THEN allI[THEN cpI2]], of OclExcluding]]
 cp-OclIncludes [THEN allI[THEN allI[THEN allI[THEN cpI2]], of OclIncludes]]
 cp-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  StrictRefEqSet :
  (x::('A,'α::null)Set) ≐ y ≡ λ τ. if (v x) τ = true τ ∧ (v y) τ = true τ
                                then (x ≐ y)τ
                                else invalid τ

```

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

4.5.2. Logic and Algebraic Layer on Set

Reflexivity

To become operational, we derive:

```

lemma StrictRefEqSet-refl[simp,code-unfold]:
  ((x::('A,'α::null)Set) ≐ x) = (if (v x) then true else invalid endif)
by(rule ext, simp add: StrictRefEqSet OclIf-def)

```

Symmetry

```

lemma StrictRefEqSet-sym:
  ((x::('A,'α::null)Set) ≐ y) = (y ≐ x)
by(simp add: StrictRefEqSet, subst StrongEq-sym, rule ext, simp)

```

Execution with invalid or null as argument

```

lemma StrictRefEqSet-strict1[simp,code-unfold]: ((x::('A,'α::null)Set) ≐ invalid) = invalid
by(simp add: StrictRefEqSet false-def true-def)

```

```

lemma StrictRefEqSet-strict2[simp,code-unfold]: (invalid ≐ (y::('A,'α::null)Set)) = invalid
by(simp add: StrictRefEqSet false-def true-def)

```

```

lemma StrictRefEqSet-strictEq-valid-args-valid:
  (τ ⊨ δ ((x::('A,'α::null)Set) ≐ y)) = ((τ ⊨ (v x)) ∧ (τ ⊨ v y))

```

proof –

```

  have A: τ ⊨ δ (x ≐ y) ⇒ τ ⊨ v x ∧ τ ⊨ v y
    apply(simp add: StrictRefEqSet valid-def OclValid-def defined-def)
    apply(simp add: invalid-def bot-fun-def split: split-if-asm)
    done
  have B: (τ ⊨ v x) ∧ (τ ⊨ v y) ⇒ τ ⊨ δ (x ≐ y)
    apply(simp add: StrictRefEqSet, elim conjE)
    apply(drule foundation13[THEN iffD2],drule foundation13[THEN iffD2])
    apply(rule cp-validity[THEN iffD2])
    apply(subst cp-defined, simp add: foundation22)
    apply(simp add: cp-defined[symmetric] cp-validity[symmetric])

```



```

    done
  show ?thesis by(auto intro!: A B)
qed

```

Behavior vs StrongEq

```

lemma StrictRefEqSet-vs-StrongEq:
 $\tau \models v \ x \implies \tau \models v \ y \implies (\tau \models ((x::('A, 'a::null) Set) \doteq y) \triangleq (x \triangleq y)))$ 
apply(drule foundation13[THEN iffD2], drule foundation13[THEN iffD2])
by(simp add:StrictRefEqSet foundation22)

```

Context Passing

```

lemma cp-StrictRefEqSet:((X::('A, 'a::null) Set) \doteq Y)  $\tau = ((\lambda-. X \ \tau) \doteq (\lambda-. Y \ \tau)) \ \tau$ 
by(simp add:StrictRefEqSet cp-StrongEq[symmetric] cp-valid[symmetric])

```

4.6. Execution on Set's Operators

4.6.1. OclIncluding

```

lemma including-cha0[simp]:
assumes val-x: $\tau \models (v \ x)$ 
shows  $\tau \models \text{not}(\text{Set}\{\}->\text{includes}(x))$ 
using val-x
apply(auto simp: OclValid-def OclIncludes-def OclNot-def false-def true-def)
apply(auto simp: mtSet-def OCL-lib.Set-0.Abs-Set-0-inverse)
done

```

```

lemma including-cha0'[simp, code-unfold]:
 $\text{Set}\{\}->\text{includes}(x) = (\text{if } v \ x \text{ then false else invalid endif})$ 
proof -
  have A:  $\bigwedge \tau. (\text{Set}\{\}->\text{includes}(\text{invalid})) \ \tau = (\text{if } (v \ \text{invalid}) \text{ then false else invalid endif}) \ \tau$ 
  by simp
  have B:  $\bigwedge \tau \ x. \tau \models (v \ x) \implies (\text{Set}\{\}->\text{includes}(x)) \ \tau = (\text{if } v \ x \text{ then false else invalid endif}) \ \tau$ 
  by (auto)
  apply(frule including-cha0, simp add: OclValid-def)
  apply(rule foundation21[THEN fun-cong, simplified StrongEq-def, simplified, THEN iffD1, of - - false])
  by simp
show ?thesis
  apply(rule ext, rename-tac  $\tau$ )
  apply(case-tac  $\tau \models (v \ x)$ )
  apply(simp-all add: B foundation18)
  apply(subst cp-OclIncludes, simp add: cp-OclIncludes[symmetric] A)
done
qed

```

```

lemma including-cha1:
assumes def-X: $\tau \models (\delta \ X)$ 
assumes val-x: $\tau \models (v \ x)$ 
shows  $\tau \models (X \rightarrow \text{including}(x) \rightarrow \text{includes}(x))$ 
proof -
  have C :  $[[\text{insert } (x \ \tau) \ [[\text{Rep-Set-0 } (X \ \tau)]]]] \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in [[X]]. x \neq \text{bot})\}$ 
    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-cha2:
assumes def-X: $\tau \models (\delta \ X)$ 
and val-x: $\tau \models (v \ x)$ 
and val-y: $\tau \models (v \ y)$ 
and neq : $\tau \models \text{not}(x \triangleq y)$ 
shows  $\tau \models (X \rightarrow \text{including}(x) \rightarrow \text{includes}(y)) \triangleq (X \rightarrow \text{includes}(y))$ 
proof -
  have C :  $[[\text{insert } (x \ \tau) \ [[\text{Rep-Set-0 } (X \ \tau)]]]] \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in [[X]]. x \neq \text{bot})\}$ 
    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*: $(x \doteq \text{invalid}) = \text{invalid}$

and *strict2*: $(\text{invalid} \doteq y) = \text{invalid}$

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

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

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

shows

$$(X \rightarrow \text{including}(x::('A, 'a::\text{null})\text{val}) \rightarrow \text{includes}(y)) = \\ (\text{if } \delta X \text{ then if } x \doteq y \text{ then true else } X \rightarrow \text{includes}(y) \text{ endif else invalid endif})$$

proof –

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

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

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

have *C*: $\bigwedge \tau. \tau \models (x \triangleq \text{invalid}) \implies$
 $(X \rightarrow \text{including}(x) \rightarrow \text{includes}(y)) \tau =$
 $(\text{if } x \doteq y \text{ then true else } X \rightarrow \text{includes}(y) \text{ endif}) \tau$
apply(*rule foundation22[THEN iffD1]*)
apply(*erule StrongEq-L-subst2-rev, simp, simp*)
by (*simp add: strict2*)

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

have *E*: $\bigwedge \tau. \tau \models v x \implies \tau \models v y \implies$
 $(\text{if } x \doteq y \text{ then true else } X \rightarrow \text{includes}(y) \text{ endif}) \tau =$
 $(\text{if } x \triangleq y \text{ then true else } X \rightarrow \text{includes}(y) \text{ endif}) \tau$
apply(*subst cp-OclIf*)
apply(*subst StrictRefEq-vs-StrongEq[THEN foundation22[THEN iffD1]]*)
by(*simp-all add: cp-OclIf[symmetric]*)

have *F*: $\bigwedge \tau. \tau \models (x \triangleq y) \implies$

```

      (X->including(x)->includes(y))  $\tau$  = (X->including(x)->includes(x))  $\tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev,simp, simp)
  show ?thesis
    apply(rule ext, rename-tac  $\tau$ )
    apply(case-tac  $\neg (\tau \models (\delta \ X))$ , simp add: def-split-local, elim disjE A B)
    apply(case-tac  $\neg (\tau \models (v \ x))$ ,
      simp add: foundation18 foundation22[symmetric],
      drule StrongEq-L-sym)
    apply(simp add: foundation22 C)
    apply(case-tac  $\neg (\tau \models (v \ y))$ ,
      simp add: foundation18 foundation22[symmetric],
      drule StrongEq-L-sym, simp add: foundation22 D, simp)
    apply(subst E,simp-all)
    apply(case-tac  $\tau \models \text{not}(x \triangle y)$ )
    apply(simp add: including-chn2[simplified foundation22])
    apply(simp add: foundation9 F
      including-chn1[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 StrictRefEqInteger-strict1 StrictRefEqInteger-strict2
  cp-StrictRefEqInteger
  StrictRefEqInteger-vs-StrongEq], simp-all)

```

```

schematic-lemma includes-execute-bool[simp,code-unfold]: ?X
by(rule includes-execute-generic[OF StrictRefEqBoolean-strict1 StrictRefEqBoolean-strict2
  cp-StrictRefEqBoolean
  StrictRefEqBoolean-vs-StrongEq], simp-all)

```

```

schematic-lemma includes-execute-set[simp,code-unfold]: ?X
by(rule includes-execute-generic[OF StrictRefEqSet-strict1 StrictRefEqSet-strict2
  cp-StrictRefEqSet
  StrictRefEqSet-vs-StrongEq], simp-all)

```

```

lemma finite-including-rep-set :
  assumes X-def :  $\tau \models \delta \ X$ 
  and x-val :  $\tau \models v \ x$ 
  shows finite  $\llbracket \text{Rep-Set-0 } (X \rightarrow \text{including}(x) \ \tau) \rrbracket = \text{finite } \llbracket \text{Rep-Set-0 } (X \ \tau) \rrbracket$ 
proof -
  have C :  $\llbracket \text{insert } (x \ \tau) \llbracket \text{Rep-Set-0 } (X \ \tau) \rrbracket \rrbracket \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$ 
  apply(insert X-def x-val, frule Set-inv-lemma)

```

```

    apply(simp add: foundation18 invalid-def)
  done
show ?thesis
by(insert X-def x-val,
    auto simp: OclIncluding-def Abs-Set-0-inverse[OF C]
    dest: foundation13[THEN iffD2, THEN foundation22[THEN iffD1]])
qed

```

lemma *including-includes-generic* :

assumes *includes-execute-generic* [simp] : $\bigwedge X x y. (X \rightarrow \text{including}(x::(\mathfrak{A}, 'a::\text{null})\text{val}) \rightarrow \text{includes}(y))$

=

```

  (if  $\delta X$  then if  $x \doteq y$  then true else  $X \rightarrow \text{includes}(y)$  endif else invalid endif)
  and StrictRefEq-strict'' :  $\bigwedge x y. \delta ((x::(\mathfrak{A}, 'a::\text{null})\text{val}) \doteq y) = (v(x) \text{ and } v(y))$ 
  and a-val :  $\tau \models v a$ 
  and x-val :  $\tau \models v x$ 
  and S-incl :  $\tau \models (S) \rightarrow \text{includes}((x::(\mathfrak{A}, 'a::\text{null})\text{val}))$ 
  shows  $\tau \models S \rightarrow \text{including}((a::(\mathfrak{A}, 'a::\text{null})\text{val})) \rightarrow \text{includes}(x)$ 

```

proof –

have *discr-eq-bot1-true* : $\bigwedge \tau. (\perp = \text{true } \tau) = \text{False}$ **by** (metis OCL-core.bot-fun-def foundation1 foundation18' valid3)

have *discr-eq-bot2-true* : $\bigwedge \tau. (\perp = \text{true } \tau) = \text{False}$ **by** (metis bot-fun-def discr-eq-bot1-true)

have *discr-neq-invalid-true* : $\bigwedge \tau. (\text{invalid } \tau \neq \text{true } \tau) = \text{True}$ **by** (metis discr-eq-bot2-true invalid-def)

have *discr-eq-invalid-true* : $\bigwedge \tau. (\text{invalid } \tau = \text{true } \tau) = \text{False}$ **by** (metis bot-option-def invalid-def option.simps(2) true-def)

show ?thesis

```

  apply(simp)
  apply(subgoal-tac  $\tau \models \delta S$ )
  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  $\delta S$ ] OclValid-def OclIf-def discr-neq-invalid-true discr-eq-invalid-true
    x-val[simplified OclValid-def])
  by (metis OclValid-def S-incl StrictRefEq-strict'' a-val foundation10 foundation6 x-val)
qed

```

lemmas *including-includes* = *including-includes-generic*[OF *includes-execute-int StrictRefEqInteger-strict''*]

lemma *including-rep-set*:

assumes *S-def* : $\tau \models \delta S$

shows $\llbracket \text{Rep-Set-0 } (S \rightarrow \text{including}(\lambda x. \llbracket x \rrbracket) \tau) \rrbracket = \text{insert } \llbracket x \rrbracket \llbracket \text{Rep-Set-0 } (S \tau) \rrbracket$

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* : $\tau \models v a$

```

shows  $\llbracket \text{Rep-Set-0 } (X \rightarrow \text{including}(a) \tau) \rrbracket \neq \{\}$ 
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  $\tau \models X \rightarrow \text{includes}(x)$ 
shows  $X \rightarrow \text{including}(x) \tau = X \tau$ 
proof -
have includes-def:  $\tau \models X \rightarrow \text{includes}(x) \implies \tau \models \delta X$ 
by (metis OCL-core.bot-fun-def OclIncludes-def OclValid-def defined3 foundation16)

have includes-val:  $\tau \models X \rightarrow \text{includes}(x) \implies \tau \models v x$ 
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-cha0[simp]:
assumes  $\text{val-}x:\tau \models (v x)$ 
shows  $\tau \models ((\text{Set}\{\}-\rightarrow \text{excluding}(x)) \triangleq \text{Set}\{\})$ 
proof -
have  $A : [\text{None}] \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$  by(simp add:
null-option-def bot-option-def)
have  $B : [\{\}] \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$  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-cha0-exec[code-unfold]:
 $(\text{Set}\{\}-\rightarrow \text{excluding}(x)) = (\text{if } (v x) \text{ then } \text{Set}\{\} \text{ else } \text{invalid endif})$ 
proof -
have  $A : \bigwedge \tau. (\text{Set}\{\}-\rightarrow \text{excluding}(\text{invalid})) \tau = (\text{if } (v \text{ invalid}) \text{ then } \text{Set}\{\} \text{ else } \text{invalid endif})$ 
 $\tau$ 
by simp

```

```

have B:  $\bigwedge \tau x. \tau \models (v\ x) \implies (Set\{\} \multimap excluding(x))\ \tau = (if\ (v\ x)\ then\ Set\{\}\ else\ invalid\ endif)\ \tau$ 
by(simp add: excluding-cha0[THEN foundation22[THEN iffD1]])
show ?thesis
apply(rule ext, rename-tac  $\tau$ )
apply(case-tac  $\tau \models (v\ x)$ )
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-cha1:
assumes def-X: $\tau \models (\delta\ X)$ 
and val-x: $\tau \models (v\ x)$ 
and val-y: $\tau \models (v\ y)$ 
and neq : $\tau \models not(x \triangleq y)$ 
shows  $\tau \models ((X \multimap including(x)) \multimap excluding(y)) \triangleq ((X \multimap excluding(y)) \multimap including(x))$ 
proof -
have A :  $\perp \in \{X. X = bot \vee X = null \vee (\forall x \in \llbracket X \rrbracket. x \neq bot)\}$  by(simp add: bot-option-def)
have B :  $\llbracket \perp \rrbracket \in \{X. X = bot \vee X = null \vee (\forall x \in \llbracket X \rrbracket. x \neq bot)\}$  by(simp add: null-option-def bot-option-def)
have C :  $\llbracket insert\ (x\ \tau)\ \llbracket Rep\ Set\ 0\ (X\ \tau) \rrbracket \rrbracket \in \{X. X = bot \vee X = null \vee (\forall x \in \llbracket X \rrbracket. x \neq bot)\}$ 
apply(insert def-X val-x, frule Set-inv-lemma)
apply(simp add: foundation18 invalid-def)
done
have D :  $\llbracket \llbracket Rep\ Set\ 0\ (X\ \tau) \rrbracket - \{y\ \tau\} \rrbracket \in \{X. X = bot \vee X = null \vee (\forall x \in \llbracket X \rrbracket. x \neq bot)\}$ 
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 :  $Abs\ Set\ 0\ \llbracket insert\ (x\ \tau)\ \llbracket Rep\ Set\ 0\ (X\ \tau) \rrbracket \rrbracket \neq Abs\ Set\ 0\ None$ 
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 :  $Abs\ Set\ 0\ \llbracket insert\ (x\ \tau)\ \llbracket Rep\ Set\ 0\ (X\ \tau) \rrbracket \rrbracket \neq Abs\ Set\ 0\ [None]$ 
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 G : (δ (λ-. Abs-Set-0 [[insert (x τ) [[Rep-Set-0 (X τ)]]]]) τ = true τ
  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-Set-0 [[[[Rep-Set-0 (X τ)] - {y τ}]]] ≠ Abs-Set-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-Set-0 [[[[Rep-Set-0 (X τ)] - {y τ}]]] ≠ Abs-Set-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 H : (δ (λ-. Abs-Set-0 [[[[Rep-Set-0 (X τ)] - {y τ}]]]) τ = true τ
  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 τ) [[Rep-Set-0 (X τ)] - {y τ} = insert (x τ) ([[Rep-Set-0 (X τ)] - {y
τ})
  by(auto simp: E)
show ?thesis
  apply(insert def-X[THEN foundation13[THEN iffD2]] val-x[THEN foundation13[THEN
iffD2]]
    val-y[THEN foundation13[THEN iffD2]])
  apply(simp add: foundation22 OclIncluding-def OclExcluding-def 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:τ ⊨ (δ X)
and val-x:τ ⊨ (v x)
shows τ ⊨ (((X->including(x))->excluding(x)) ≜ (X->excluding(x)))
proof -
have A : ⊥ ∈ {X. X = bot ∨ X = null ∨ (∀ x∈[[X]]. x ≠ bot)} by(simp add: bot-option-def)
have B : ⊥ ∈ {X. X = bot ∨ X = null ∨ (∀ x∈[[X]]. x ≠ bot)} by(simp add: null-option-def
bot-option-def)
have C : [[insert (x τ) [[Rep-Set-0 (X τ)]]]] ∈ {X. X = bot ∨ X = null ∨ (∀ x∈[[X]]. x
≠ bot)}
  apply(insert def-X val-x, frule Set-inv-lemma)

```



```

    apply(simp add: foundation18 invalid-def)
  done
have G1 : Abs-Set-0 [[insert (x τ) [[Rep-Set-0 (X τ)]]]] ≠ Abs-Set-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-Set-0 [[insert (x τ) [[Rep-Set-0 (X τ)]]]] ≠ Abs-Set-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
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 ≐ invalid) = invalid
  and strict2: (invalid ≐ y) = invalid
  and StrictRefEq-valid-args-valid:  $\bigwedge (x::('A, 'a::null)val) y \tau. (\tau \models \delta (x \doteq y)) = ((\tau \models (v x)) \wedge (\tau \models v y))$ 
  and cp-StrictRefEq:  $\bigwedge (X::('A, 'a::null)val) Y \tau. (X \doteq Y) \tau = ((\lambda-. X \tau) \doteq (\lambda-. Y \tau)) \tau$ 
  and StrictRefEq-vs-StrongEq:  $\bigwedge (x::('A, 'a::null)val) y \tau. \tau \models v x \implies \tau \models v y \implies (\tau \models ((x \doteq y) \triangleq (x \triangleq y)))$ 
  shows (X -> including(x::('A, 'a::null)val) -> excluding(y)) =
    (if δ X then if x ≐ y
      then X -> excluding(y)
      else X -> excluding(y) -> including(x)
    endif
    else invalid endif)

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

```

```

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

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

have B2:  $\bigwedge \tau. \tau \models (X \triangleq \text{null}) \implies X \rightarrow \text{including}(x) \rightarrow \text{excluding}(y) \tau = \text{invalid } \tau$ 
  apply(rule foundation22[THEN iffD1])
  by(erule StrongEq-L-subst2-rev, simp,simp)

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

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

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

have E:  $\bigwedge \tau. \tau \models v \ x \implies \tau \models v \ y \implies$ 
   $(\text{if } x \doteq y \text{ then } X \rightarrow \text{excluding}(y) \text{ else } X \rightarrow \text{excluding}(y) \rightarrow \text{including}(x) \text{ endif}) \tau =$ 
   $(\text{if } x \triangleq y \text{ then } X \rightarrow \text{excluding}(y) \text{ else } X \rightarrow \text{excluding}(y) \rightarrow \text{including}(x) \text{ endif}) \tau$ 
  apply(subst cp-OclIf)
  apply(subst StrictRefEq-vs-StrongEq[THEN foundation22[THEN iffD1]])
  by(simp-all add: cp-OclIf[symmetric])

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

show ?thesis
  apply(rule ext, rename-tac  $\tau$ )
  apply(case-tac  $\neg (\tau \models (\delta \ X))$ , simp add: def-split-local, elim disjE A1 B1 A2 B2)
  apply(case-tac  $\neg (\tau \models (v \ x))$ ,
    simp add: foundation18 foundation22[symmetric],

```

```

    drule StrongEq-L-sym)
  apply(simp add: foundation22 C)
  apply(case-tac  $\neg (\tau \models (v \ y))$ ),
    simp add: foundation18 foundation22[symmetric],
    drule StrongEq-L-sym, simp add: foundation22 D, simp)
  apply(subst E, simp-all)
  apply(case-tac  $\tau \models \text{not } (x \triangleq y)$ )
  apply(simp add: 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 StrictRefEqInteger-strict1 StrictRefEqInteger-strict2
  StrictRefEqInteger-defined-args-valid
  cp-StrictRefEqInteger StrictRefEqInteger-vs-StrongEq], simp-all)

```

```

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

```

```

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

```

```

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

```

```

lemma excluding-rep-set:
  assumes S-def:  $\tau \models \delta \ S$ 
  shows  $\llbracket \text{Rep-Set-0 } (S \rightarrow \text{excluding}(\lambda x. \llbracket x \rrbracket) \ \tau) \rrbracket = \llbracket \text{Rep-Set-0 } (S \ \tau) \rrbracket - \{\llbracket x \rrbracket\}$ 

```

```

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: $\tau \models \text{not}(\delta(S \rightarrow \text{size}()))$ 
shows  $(\tau \models \text{not}(\delta(S))) \vee \neg \text{finite } \llbracket \text{Rep-Set-0 } (S \ \tau) \rrbracket$ 
apply(insert non-finite, simp)
apply(rule impI)
apply(simp add: OclSize-def OclValid-def defined-def)
apply(case-tac finite  $\llbracket \text{Rep-Set-0 } (S \ \tau) \rrbracket$ ,
      simp-all add:null-fun-def null-option-def bot-fun-def bot-option-def)
done

lemma [simp,code-unfold]:  $\text{Set}\{\} \rightarrow \text{size}() = 0$ 
proof -
  have A1 :  $\llbracket \{\} \rrbracket \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$  by(simp add: mtSet-def)
  have A2 :  $\text{None} \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$  by(simp add:
    bot-option-def)
  have A3 :  $\llbracket \text{None} \rrbracket \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$  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]:  $\delta(\text{Set}\{\} \rightarrow \text{size}()) = \text{true}$ 
by simp

```

```

lemma including-size-defined[simp,code-unfold]:  $\delta((X \rightarrow \text{including}(x)) \rightarrow \text{size}()) = (\delta(X \rightarrow \text{size}())$ 
  and  $v(x))$ 
proof -
  have defined-inject-true :  $\bigwedge \tau P. (\delta P) \ \tau \neq \text{true} \ \tau \implies (\delta P) \ \tau = \text{false} \ \tau$ 
    apply(simp add: defined-def true-def false-def bot-fun-def bot-option-def
      null-fun-def null-option-def)
    by (case-tac  $P \ \tau = \perp \vee P \ \tau = \text{null}$ , simp-all add: true-def)
  have valid-inject-true :  $\bigwedge \tau P. (v P) \ \tau \neq \text{true} \ \tau \implies (v P) \ \tau = \text{false} \ \tau$ 
    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  $\tau = \perp$ , simp-all add: true-def)

have finite-including-rep-set :  $\bigwedge \tau. (\delta X \text{ and } v x) \tau = \text{true } \tau \implies$ 
    finite  $\llbracket \llbracket \text{Rep-Set-0 } (X \rightarrow \text{including}(x) \tau) \rrbracket \rrbracket = \text{finite } \llbracket \llbracket \text{Rep-Set-0 } (X \tau) \rrbracket \rrbracket$ 
  apply(rule finite-including-rep-set)
  apply(metis OclValid-def foundation5)+
done

have card-including-exec :  $\bigwedge \tau. (\delta (\lambda-. \llbracket \llbracket \text{int } (\text{card } \llbracket \llbracket \text{Rep-Set-0 } (X \rightarrow \text{including}(x) \tau) \rrbracket \rrbracket) \rrbracket \rrbracket)) \tau$ 
  =  $(\delta (\lambda-. \llbracket \llbracket \text{int } (\text{card } \llbracket \llbracket \text{Rep-Set-0 } (X \tau) \rrbracket \rrbracket) \rrbracket \rrbracket)) \tau$ 
  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  $\tau$ )
  apply(case-tac  $(\delta (X \rightarrow \text{including}(x) \rightarrow \text{size}())) \tau = \text{true } \tau$ , simp)
  apply(subst cp-OclAnd)
  apply(subst cp-defined)
  apply(simp only: cp-defined[of  $X \rightarrow \text{including}(x) \rightarrow \text{size}()$ ])
  apply(simp add: OclSize-def)
  apply(case-tac  $((\delta X \text{ and } v x) \tau = \text{true } \tau \wedge \text{finite } \llbracket \llbracket \text{Rep-Set-0 } (X \rightarrow \text{including}(x) \tau) \rrbracket \rrbracket)$ ,
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  $\delta X v x$ ]
    cp-OclAnd[of true, THEN sym])
  apply(subgoal-tac  $(\delta X) \tau = \text{true } \tau \wedge (v x) \tau = \text{true } \tau$ , simp)
  apply(rule foundation5[of -  $\delta X v x$ , simplified OclValid-def], simp only: cp-OclAnd[THEN
sym])

  apply(drule defined-inject-true[of  $X \rightarrow \text{including}(x) \rightarrow \text{size}()$ ], simp)
  apply(simp only: cp-OclAnd[of  $\delta (X \rightarrow \text{size}()) v x$ ])
  apply(simp add: cp-defined[of  $X \rightarrow \text{including}(x) \rightarrow \text{size}()$ ] cp-defined[of  $X \rightarrow \text{size}()$ ])
  apply(simp add: OclSize-def card-including-exec)
  apply(case-tac  $(\delta X \text{ and } v x) \tau = \text{true } \tau \wedge \text{finite } \llbracket \llbracket \text{Rep-Set-0 } (X \tau) \rrbracket \rrbracket$ ,
    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 (v x)  $\tau = \text{true}$   $\tau$ , simp add: cp-OclAnd[of  $\delta$  X v x])
apply(drule valid-inject-true[of x], simp add: cp-OclAnd[of - v x])
done
qed

lemma including-size-exec[code-unfold]:
  ((X ->including(x)) ->size()) = (if  $\delta$  X and v x then
    X ->size() +ocl if X ->includes(x) then 0 else 1 endif
    else
    invalid
    endif)

proof -

have valid-inject-true :  $\bigwedge \tau P. (v P) \tau \neq \text{true} \tau \implies (v P) \tau = \text{false} \tau$ 
  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  $\tau = \perp$ , simp-all add: true-def)
have defined-inject-true :  $\bigwedge \tau P. (\delta P) \tau \neq \text{true} \tau \implies (\delta P) \tau = \text{false} \tau$ 
  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  $\tau = \perp \vee P \tau = \text{null}$ , simp-all add: true-def)

show ?thesis
apply(rule ext, rename-tac  $\tau$ )
proof -
fix  $\tau$ 
have includes-notin:  $\neg \tau \models X \text{->includes}(x) \implies (\delta X) \tau = \text{true} \tau \wedge (v x) \tau = \text{true} \tau \implies$ 
 $x \tau \notin \llbracket \text{Rep-Set-0 } (X \tau) \rrbracket$ 
by(simp add: OclIncludes-def OclValid-def true-def)

have includes-def:  $\tau \models X \text{->includes}(x) \implies \tau \models \delta X$ 
by (metis OCL-core.bot-fun-def OclIncludes-def OclValid-def defined3 foundation16)

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

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

  apply(simp add: cp-OclAddInteger[symmetric])
  apply(case-tac  $\tau \models ((v (X \text{->size}())) \text{ and not } (\delta (X \text{->size}())))$ , simp)
  apply(drule foundation5[where P = v 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-OclAddInteger)
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: OclAddInteger-def OclInt1-def)
apply(rule impI)
apply(drule Finite-Set.card.insert[where x = x τ])
apply(rule includes-notin, simp)
apply(simp)
apply (metis Suc-eq-plus1 int-1 of-nat-add)

apply(subst (1 2) OclAddInteger-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 founda-
tion9)
by (metis cp-OclSize foundation18' including-valid-args-valid'' invalid-def size-strict1)
qed
qed

lemma excluding-size-defined[simp,code-unfold]:  $\delta((X \rightarrow \text{excluding}(x)) \rightarrow \text{size}()) = (\delta(X \rightarrow \text{size}()))$ 
and  $v(x)$ )
proof —

```

```

have defined-inject-true :  $\bigwedge \tau P. (\delta P) \tau \neq \text{true } \tau \implies (\delta P) \tau = \text{false } \tau$ 
  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  $\tau = \perp \vee P \tau = \text{null}$ , simp-all add: true-def)

have valid-inject-true :  $\bigwedge \tau P. (\vee P) \tau \neq \text{true } \tau \implies (\vee P) \tau = \text{false } \tau$ 
  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  $\tau = \perp$ , simp-all add: true-def)

have finite-excluding-rep-set :  $\bigwedge \tau. (\delta X \text{ and } \vee x) \tau = \text{true } \tau \implies$ 
   $\text{finite } \llbracket \llbracket \text{Rep-Set-0 } (X \rightarrow \text{excluding}(x) \tau) \rrbracket \rrbracket =$ 
   $\text{finite } \llbracket \llbracket \text{Rep-Set-0 } (X \tau) \rrbracket \rrbracket$ 
  apply(rule finite-excluding-rep-set)
  apply(metis OclValid-def foundation5)+
done

have card-excluding-exec :  $\bigwedge \tau. (\delta (\lambda-. \llbracket \llbracket \text{int } (\text{card } \llbracket \llbracket \text{Rep-Set-0 } (X \rightarrow \text{excluding}(x) \tau) \rrbracket \rrbracket) \rrbracket) \rrbracket) \tau$ 
   $=$ 
   $(\delta (\lambda-. \llbracket \llbracket \text{int } (\text{card } \llbracket \llbracket \text{Rep-Set-0 } (X \tau) \rrbracket \rrbracket) \rrbracket) \rrbracket) \tau$ 
  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  $\tau$ )
apply(case-tac  $(\delta (X \rightarrow \text{excluding}(x) \rightarrow \text{size}())) \tau = \text{true } \tau$ , simp)
apply(subst cp-OclAnd)
apply(subst cp-defined)
apply(simp only: cp-defined[of X  $\rightarrow$  excluding(x)  $\rightarrow$  size()])
apply(simp add: OclSize-def)
apply(case-tac  $((\delta X \text{ and } \vee x) \tau = \text{true } \tau \wedge \text{finite } \llbracket \llbracket \text{Rep-Set-0 } (X \rightarrow \text{excluding}(x) \tau) \rrbracket \rrbracket)$ ,
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  $\delta X \vee x$ ]
  cp-OclAnd[of true, THEN sym])
apply(subgoal-tac  $(\delta X) \tau = \text{true } \tau \wedge (\vee x) \tau = \text{true } \tau$ , simp)
apply(rule foundation5[of -  $\delta X \vee x$ , simplified OclValid-def], simp only: cp-OclAnd[THEN sym])

apply(drule defined-inject-true[of X  $\rightarrow$  excluding(x)  $\rightarrow$  size()], simp)
apply(simp only: cp-OclAnd[of  $\delta (X \rightarrow \text{size}()) \vee x$ ])
apply(simp add: cp-defined[of X  $\rightarrow$  excluding(x)  $\rightarrow$  size()] cp-defined[of X  $\rightarrow$  size()])

```



```

apply(simp add: OclSize-def finite-excluding-rep-set card-excluding-exec)
apply(case-tac ( $\delta X$  and  $v x$ )  $\tau = \text{true } \tau \wedge \text{finite } [[\text{Rep-Set-0 } (X \ \tau)]]$ ,
  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 ( $v x$ )  $\tau = \text{true } \tau$ , simp add: cp-OclAnd[of  $\delta X \ v x$ ])
apply(drule valid-inject-true[of  $x$ ], simp add: cp-OclAnd[of -  $v x$ ])
done
qed

lemma size-defined:
assumes  $X\text{-finite}$ :  $\bigwedge \tau. \text{finite } [[\text{Rep-Set-0 } (X \ \tau)]]$ 
shows  $\delta (X \rightarrow \text{size}()) = \delta X$ 
apply(rule ext, simp add: cp-defined[of  $X \rightarrow \text{size}()$ ] OclSize-def)
apply(simp add: defined-def bot-option-def bot-fun-def null-option-def null-fun-def  $X\text{-finite}$ )
done

lemma size-defined':
assumes  $X\text{-finite}$ :  $\text{finite } [[\text{Rep-Set-0 } (X \ \tau)]]$ 
shows  $(\tau \models \delta (X \rightarrow \text{size}())) = (\tau \models \delta X)$ 
apply(simp add: cp-defined[of  $X \rightarrow \text{size}()$ ] OclSize-def OclValid-def)
apply(simp add: defined-def bot-option-def bot-fun-def null-option-def null-fun-def  $X\text{-finite}$ )
done

lemma [simp]:
assumes  $X\text{-finite}$ :  $\bigwedge \tau. \text{finite } [[\text{Rep-Set-0 } (X \ \tau)]]$ 
shows  $\delta ((X \rightarrow \text{including}(x)) \rightarrow \text{size}()) = (\delta(X) \text{ and } v(x))$ 
by(simp add: size-defined[OF  $X\text{-finite}$ ])

```

4.6.4. OclIsEmpty

```

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

lemma including-not-isempty [simp]:
assumes  $X\text{-def}$ :  $\tau \models \delta X$ 
  and  $X\text{-finite}$ :  $\text{finite } [[\text{Rep-Set-0 } (X \ \tau)]]$ 
  and  $a\text{-val}$ :  $\tau \models v a$ 
shows  $X \rightarrow \text{including}(a) \rightarrow \text{isEmpty}() \ \tau = \text{false } \tau$ 
proof -
  have  $A1 : \bigwedge \tau. X \ \tau = \text{true } \tau \vee X \ \tau = \text{false } \tau \implies (X \text{ and not } X) \ \tau = \text{false } \tau$ 

```

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 : $\bigwedge \tau P. (\delta P) \tau \neq \text{true} \tau \implies (\delta P) \tau = \text{false} \tau$
 apply (simp add: defined-def true-def false-def bot-fun-def bot-option-def
 null-fun-def null-option-def)
 by (case-tac $P \tau = \perp \vee P \tau = \text{null}$, simp-all add: true-def)

have B : $\bigwedge X \tau. \tau \models v X \implies X \tau \neq \mathbf{0} \tau \implies (X \doteq \mathbf{0}) \tau = \text{false} \tau$
 by (metis OclAnd-true2 OclValid-def Sem-def StrictRefEqInteger StrictRefEqInteger-strict'
 StrictRefEqInteger-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]: $\text{Set}\{\}->\text{notEmpty}() = \text{false}$
 by (simp add: OclNotEmpty-def)

lemma including-notempty-true [simp,code-unfold]:
 assumes X-def: $\tau \models \delta X$
 and X-finite: $\text{finite } [[\text{Rep-Set-0 } (X \tau)]]$
 and a-val: $\tau \models v a$
 shows $X->\text{including}(a)->\text{notEmpty}() \tau = \text{true} \tau$
 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]: $\text{Set}\{\}->\text{any}() = \text{null}$
 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  $\tau$ , simp add: mtSet-def Ocl-Any-def)
  apply(case-tac  $\tau \models v$  a)
  apply(simp add: OclValid-def mtSet-defined[simplified mtSet-def] mtSet-valid[simplified mtSet-def]
    mtSet-rep-set[simplified mtSet-def])
  apply(subst (1 2) cp-OclAnd,
    subst (1 2) 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  $\tau = \perp$ )
  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]:
   $X -> includes(X -> any()) = (if \delta X \text{ then } if \delta (X -> size()) \text{ then } not(X -> isEmpty()) \text{ else } X -> includes(null) \text{ endif} \text{ else } invalid \text{ endif})$ 

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

  have valid-inject-true :  $\bigwedge \tau P. (v P) \tau \neq true \tau \implies (v P) \tau = false \tau$ 
  apply(simp add: valid-def true-def false-def bot-fun-def bot-option-def
    null-fun-def null-option-def)
  by (case-tac  $P \tau = \perp$ , simp-all add: true-def)

  have notempty':  $\bigwedge \tau X. \tau \models \delta X \implies finite \llbracket Rep-Set-0 (X \tau) \rrbracket \implies not (X -> isEmpty()) \tau \neq true \tau \implies X \tau = Set\{\}$ 
  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-StrictRefEqInteger,
subst (asm) cp-OclAnd, subst (asm) cp-OclNot)
  apply(simp only: OclValid-def foundation20[simplified OclValid-def]
    cp-OclNot[symmetric] cp-OclAnd[symmetric] cp-OclOr[symmetric])
apply(simp add: Abs-Set-0-inverse split: split-if-asm)
by(simp add: true-def OclInt0-def OclNot-def StrictRefEqInteger StrongEq-def)

have B:  $\bigwedge X \tau. \neg \text{finite } [[\text{Rep-Set-0 } (X \ \tau)]] \implies (\delta (X \rightarrow \text{size}())) \tau = \text{false } \tau$ 
apply(subst cp-defined)
apply(simp add: OclSize-def)
by (metis OCL-core.bot-fun-def defined-def)

show ?thesis
apply(rule ext, rename-tac  $\tau$ , simp only: OclIncludes-def Ocl-Any-def)
apply(subst cp-OclIf, subst (2) cp-valid)
apply(case-tac ( $\delta X$ )  $\tau = \text{true } \tau$ , simp only: foundation20[simplified OclValid-def] cp-OclIf[symmetric],
simp,
  subst (1 2) cp-OclAnd, simp add: cp-OclAnd[symmetric])
apply(case-tac finite  $[[\text{Rep-Set-0 } (X \ \tau)]]$ )
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 ( $X \rightarrow \text{notEmpty}()$ )  $\tau = \text{true } \tau$ , simp)
apply(frule notempty-has-elt[simplified OclValid-def], simp)
apply(simp add: OclNotEmpty-def cp-OclIf[symmetric])
apply(subgoal-tac (SOME y.  $y \in [[\text{Rep-Set-0 } (X \ \tau)]] \in [[\text{Rep-Set-0 } (X \ \tau)]]$ , simp add:
true-def)
apply(metis OclValid-def Set-inv-lemma foundation18' null-option-def true-def)
apply(rule someI-ex, simp)
apply(simp add: OclNotEmpty-def cp-valid[symmetric])
apply(subgoal-tac  $\neg (\text{null } \tau \in [[\text{Rep-Set-0 } (X \ \tau)]]$ ), simp)
apply(subst OclIsEmpty-def, simp add: OclSize-def)
apply(subst cp-OclNot, subst cp-OclOr, subst cp-StrictRefEqInteger, subst cp-OclAnd, subst
cp-OclNot,
  simp add: OclValid-def foundation20[simplified OclValid-def]
    cp-OclNot[symmetric] cp-OclAnd[symmetric] cp-OclOr[symmetric])
apply(frule notempty'[simplified OclValid-def], (simp add: mtSet-def Abs-Set-0-inverse OclInt0-def
false-def)+)
apply(drule notempty'[simplified OclValid-def], simp, simp)
apply (metis (hide-lams, no-types) empty-iff mtSet-rep-set)

apply(frule B)
apply(subst (1 2 3 4) cp-OclIf)
apply(simp)
apply(subst (1 2 3 4) cp-OclIf[symmetric], simp)
apply(case-tac ( $X \rightarrow \text{notEmpty}()$ )  $\tau = \text{true } \tau$ , simp)

```

```

apply(frule notempty-has-elt[simplified OclValid-def], simp)
apply(simp add: OclNotEmpty-def OclIsEmpty-def)
apply(subgoal-tac  $X \rightarrow \text{size}() \tau = \perp$ )
prefer 2
apply (metis (hide-lams, no-types) OclSize-def)
  apply(subst (asm) cp-OclNot, subst (asm) cp-OclOr, subst (asm) cp-StrictRefEqInteger,
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
  StrictRefEqInteger 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] :
  ( $\text{null} \rightarrow \text{forall}(z \mid P(z))$ ) = invalid
by(simp add: OclForall-def invalid-def false-def true-def)

```

```

lemma forall-set-mt-exec[simp,code-unfold] :
  ( $(\text{Set}\{\}) \rightarrow \text{forall}(z \mid P(z))$ ) = 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 ( $(S \rightarrow \text{including}(x)) \rightarrow \text{forall}(z \mid P(z))$ ) = (if  $\delta S$  and  $v x$ 
    then  $P x$  and  $(S \rightarrow \text{forall}(z \mid P(z)))$ 
    else invalid
    endif)

```

proof –

```

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

  have insert-in-Set-0 :  $\bigwedge \tau. (\tau \models (\delta S)) \implies (\tau \models (v x)) \implies \llbracket \text{insert } (x \tau) \llbracket \text{Rep-Set-0 } (S \tau) \rrbracket \rrbracket$ 
     $\in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$ 
    apply(frule Set-inv-lemma)
    apply(simp add: foundation18 invalid-def)
  done

```

```

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

```

```

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

have forall-including-invert :  $\bigwedge \tau f. (f x \tau = f (\lambda \_. x \tau) \tau) \implies$ 
 $\tau \models (\delta S \text{ and } v x) \implies$ 
 $(\forall x \in [[\text{Rep-Set-0 } (S \rightarrow \text{including}(x) \tau)]]]. f (\lambda \_. x) \tau =$ 
 $(f x \tau \wedge (\forall x \in [[\text{Rep-Set-0 } (S \tau)]]]. f (\lambda \_. x) \tau))$ 

apply(simp add: OclIncluding-def)
apply(subst Abs-Set-0-inverse)
apply(rule insert-in-Set-0)
apply(rule d-and-v-destruct-defined, assumption)
apply(rule d-and-v-destruct-valid, assumption)
apply(simp add: d-and-v-destruct-defined d-and-v-destruct-valid)
apply(frule d-and-v-destruct-defined, drule d-and-v-destruct-valid)
apply(simp add: OclValid-def)
done

have exists-including-invert :  $\bigwedge \tau f. (f x \tau = f (\lambda \_. x \tau) \tau) \implies$ 
 $\tau \models (\delta S \text{ and } v x) \implies$ 
 $(\exists x \in [[\text{Rep-Set-0 } (S \rightarrow \text{including}(x) \tau)]]]. f (\lambda \_. x) \tau =$ 
 $(f x \tau \vee (\exists x \in [[\text{Rep-Set-0 } (S \tau)]]]. f (\lambda \_. x) \tau))$ 

apply(subst arg-cong[where f =  $\lambda x. \neg x$ ,
OF forall-including-invert[where f =  $\lambda x \tau. \neg (f x \tau)$ ,
simplified])

by simp-all

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

have foundation10':  $\bigwedge \tau x y. (\tau \models x) \wedge (\tau \models y) \implies \tau \models (x \text{ and } y)$ 
apply(erule conjE)
apply(subst foundation10)
apply(rule foundation6, simp)
apply(rule foundation6, simp)
by simp

have contradict-Rep-Set-0:  $\bigwedge \tau S f.$ 
 $\exists x \in [[\text{Rep-Set-0 } S]]. f (\lambda \_. x) \tau \implies$ 
 $(\forall x \in [[\text{Rep-Set-0 } S]]. \neg (f (\lambda \_. x) \tau)) = \text{False}$ 
by(case-tac  $(\forall x \in [[\text{Rep-Set-0 } S]]. \neg (f (\lambda \_. x) \tau)) = \text{True}, \text{simp-all}$ )

show ?thesis

apply(rule ext, rename-tac  $\tau$ )
apply(simp add: OclIf-def)
apply(simp add: cp-defined[of  $\delta S \text{ and } v x$ ])
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** $y = v x$, *simplified OclValid-def*])

apply(*subgoal-tac* $\tau \models (\delta S \text{ and } v x)$)
prefer 2
apply(*simp* *add*: *OclValid-def*)

apply(*case-tac* $\exists x \in [[\text{Rep-Set-0 } (S \rightarrow \text{including}(x) \tau)]] . P (\lambda -. x) \tau = \text{false } \tau$, *simp-all*)
apply(*subst contradict-Rep-Set-0*[**where** $f = \lambda x \tau . P x \tau = \text{false } \tau$], *simp*) +
apply(*simp* *add*: *exists-including-invert*[**where** $f = \lambda x \tau . P x \tau = \text{false } \tau$, *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* $\tau = \text{false } \tau$)
apply(*simp* *only*: *cp-OclAnd*[*symmetric*], *simp*)
apply(*simp* *add*: *OclForall-def*)

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

apply(*case-tac* $\exists x \in [[\text{Rep-Set-0 } (S \rightarrow \text{including}(x) \tau)]] . P (\lambda -. x) \tau = \text{bot } \tau$, *simp-all*)
apply(*subst contradict-Rep-Set-0*[**where** $f = \lambda x \tau . P x \tau = \text{bot } \tau$], *simp*) +
apply(*simp* *add*: *exists-including-invert*[**where** $f = \lambda x \tau . P x \tau = \text{bot } \tau$, *OF cp-eq*])

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

apply(*subgoal-tac OclForall S P* $\tau \neq \text{false } \tau$)
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* $\tau = \text{bot } \tau$)
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  $f = \lambda x \tau. P x \tau \neq \text{bot } \tau$ , OF cp-OclNot-eq],
      erule conjE)

apply(case-tac  $\exists x \in [\text{Rep-Set-0 } (S \rightarrow \text{including}(x) \tau)]$ .  $P (\lambda \cdot x) \tau = \text{null } \tau$ , simp-all)
apply(subst contradict-Rep-Set-0[where  $f = \lambda x \tau. P x \tau = \text{null } \tau$ ], simp)+
apply(simp add: exists-including-invert[where  $f = \lambda x \tau. P x \tau = \text{null } \tau$ , OF cp-eq])

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

apply(subgoal-tac OclForall  $S P \tau \neq \text{false } \tau \wedge \text{OclForall } S P \tau \neq \text{bot } \tau$ )
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  $S P \tau = \text{null } \tau$ )
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  $f = \lambda x \tau. P x \tau \neq \text{null } \tau$ , OF cp-OclNot-eq],
      erule conjE)

apply(simp add: cp-OclAnd[of  $P x$ ] OclForall-def)
apply(subgoal-tac  $P x \tau = \text{true } \tau$ , 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  $x\text{-def} : \tau \models \delta x$ 
    and  $y\text{-def} : \tau \models \delta y$ 
  shows  $(\tau \models \text{OclForall } x (\text{OclIncludes } y)) = ([\text{Rep-Set-0 } (x \tau)] \subseteq [\text{Rep-Set-0 } (y \tau)])$ 
proof -
  have  $\text{discr-eq-false-true} : \bigwedge \tau. (\text{false } \tau = \text{true } \tau) = \text{False}$  by (metis OclValid-def foundation2)
  have  $\text{discr-eq-bot1-true} : \bigwedge \tau. (\perp \tau = \text{true } \tau) = \text{False}$  by (metis defined3 defined-def discr-eq-false-true)
  have  $\text{discr-eq-bot2-true} : \bigwedge \tau. (\perp = \text{true } \tau) = \text{False}$  by (metis bot-fun-def discr-eq-bot1-true)
  have  $\text{discr-eq-null-true} : \bigwedge \tau. (\text{null } \tau = \text{true } \tau) = \text{False}$  by (metis OclValid-def foundation4)
  show ?thesis

```



```

apply(case-tac  $\tau \models \text{OclForall } x \ (\text{OclIncludes } y)$ )

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  $\forall x \in [\text{Rep-Set-0 } (x \ \tau)] . (\tau \models y \text{--> includes}((\lambda \cdot. x)))$ )
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  $P = \lambda x. \tau \models y \text{--> includes}((\lambda \cdot. x))$  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  $(\exists x \in [\text{Rep-Set-0 } (x \ \tau)] . (y \text{--> includes}((\lambda \cdot. x))) \ \tau = \text{false } \tau$ 
 $\vee (y \text{--> includes}((\lambda \cdot. x))) \ \tau = \perp \ \tau$ 
 $\vee (y \text{--> includes}((\lambda \cdot. x))) \ \tau = \text{null } \tau)$ )

prefer 2
apply metis
apply(erule bexE, rename-tac e)
apply(simp add: OclIncludes-def y-def[simplified OclValid-def])

apply(case-tac  $\tau \models v \ (\lambda \cdot. e)$ , 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  $x\text{-def} : \tau \models \delta \ x$ 
and  $y\text{-def} : \tau \models \delta \ y$ 
shows  $(\text{OclForall } x \ (\text{OclIncludes } y) \ \tau = \text{false } \tau) = (\neg [\text{Rep-Set-0 } (x \ \tau)]) \subseteq [\text{Rep-Set-0 } (y \ \tau)]$ )
proof –
have discr-eq-false-true :  $\bigwedge \tau. (\text{false } \tau = \text{true } \tau) = \text{False}$  by (metis OclValid-def foundation2)
have discr-eq-null-true :  $\bigwedge \tau. (\text{null } \tau = \text{true } \tau) = \text{False}$  by (metis OclValid-def foundation4)
have discr-eq-null-false :  $\bigwedge \tau. (\text{null } \tau = \text{false } \tau) = \text{False}$  by (metis defined4 foundation1 foundation16 null-fun-def)
have discr-neq-false-true :  $\bigwedge \tau. (\text{false } \tau \neq \text{true } \tau) = \text{True}$  by (metis discr-eq-false-true)
have discr-neq-true-false :  $\bigwedge \tau. (\text{true } \tau \neq \text{false } \tau) = \text{True}$  by (metis discr-eq-false-true)
have discr-eq-bot1-true :  $\bigwedge \tau. (\perp \ \tau = \text{true } \tau) = \text{False}$  by (metis defined3 defined-def discr-eq-false-true)
have discr-eq-bot2-true :  $\bigwedge \tau. (\perp = \text{true } \tau) = \text{False}$  by (metis bot-fun-def discr-eq-bot1-true)
have discr-eq-bot1-false :  $\bigwedge \tau. (\perp \ \tau = \text{false } \tau) = \text{False}$  by (metis OCL-core.bot-fun-def defined4)

```

```

foundation1 foundation16)
  have discr-eq-bot2-false :  $\bigwedge \tau. (\perp = \text{false } \tau) = \text{False}$  by (metis foundation1 foundation18'
valid4)
  show ?thesis
    apply(subgoal-tac  $\neg (\text{OclForall } x (\text{OclIncludes } y) \tau = \text{false } \tau) = (\neg (\neg \llbracket \text{Rep-Set-0 } (x \ \tau) \rrbracket$ 
 $\subseteq \llbracket \text{Rep-Set-0 } (y \ \tau) \rrbracket))$ , 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  $(\forall x \in \llbracket \text{Rep-Set-0 } (x \ \tau) \rrbracket. ((y -> \text{includes}((\lambda -. x))) \tau = \text{true } \tau \vee (y -> \text{includes}((\lambda -.$ 
 $x))) \tau = \text{false } \tau))$ )
    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  $\llbracket \text{Rep-Set-0 } (S \ \tau) \rrbracket$ 
  shows  $S -> \text{forAll}(x \mid P \ x) \ \tau = (S -> \text{iterate}(x; \text{acc} = \text{true} \mid \text{acc and } P \ x)) \ \tau$ 
proof -
  have and-comm : comp-fun-commute  $(\lambda x \text{ acc. acc and } P \ x)$ 
  apply(simp add: comp-fun-commute-def comp-def)
  by (metis OclAnd-assoc OclAnd-commute)

  have ex-insert :  $\bigwedge x \ F \ P. (\exists x \in \text{insert } x \ F. P \ x) = (P \ x \vee (\exists x \in F. P \ x))$ 
  by (metis insert-iff)

  have destruct-ocl :  $\bigwedge x \ \tau. x = \text{true } \tau \vee x = \text{false } \tau \vee x = \text{null } \tau \vee x = \perp \ \tau$ 
  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 :  $\bigwedge P1 \ P2 \ P3 \ P4 \ R.$ 
 $(P1 \vee P2 \vee P3 \vee P4) \implies (P1 \implies R) \implies (P2 \implies R) \implies (P3 \implies R) \implies (P4 \implies R)$ 

```

$\implies R$

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 $\exists x \in [[Rep-Set-0 (S \tau)]]$. $P (\lambda-. x) \tau = false \tau$ then false τ*

else if $\exists x \in [[Rep-Set-0 (S \tau)]]$. $P (\lambda-. x) \tau = \perp \tau$ then $\perp \tau$

else if $\exists x \in [[Rep-Set-0 (S \tau)]]$. $P (\lambda-. x) \tau = null \tau$ then null τ

else true τ) = Finite-Set.fold ($\lambda x acc. acc$ and $P x$) true (($\lambda a \tau. a$) ‘

[[Rep-Set-0 (S τ)]]) τ ,

simp add: S-finite)

apply(*case-tac $[[Rep-Set-0 (S \tau)]] = \{\}$, simp*)

apply(*rule finite-ne-induct[where $P = \lambda set. (if \exists x \in set. P (\lambda-. x) \tau = false \tau$ then false τ*

else if $\exists x \in set. P (\lambda-. x) \tau = \perp \tau$ then $\perp \tau$

else if $\exists x \in set. P (\lambda-. x) \tau = null \tau$ then null τ else true τ) =

Finite-Set.fold ($\lambda x acc. acc$ 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 $(\delta S) \tau = true \tau \implies \exists x. x \in F \implies$

(if $P (\lambda-. x) \tau = false \tau \vee (\exists x \in F. P (\lambda-. x) \tau = false \tau)$ then false τ

else if $P (\lambda-. x) \tau = \perp \tau \vee (\exists x \in F. P (\lambda-. x) \tau = \perp \tau)$ then $\perp \tau$

else if $P (\lambda-. x) \tau = null \tau \vee (\exists x \in F. P (\lambda-. x) \tau = null \tau)$ then null τ else $(\delta$

S) τ) =

(($\lambda-. if \exists x \in F. P (\lambda-. x) \tau = false \tau$ then false τ

else if $\exists x \in F. P (\lambda-. x) \tau = \perp \tau$ then $\perp \tau$

else if $\exists x \in F. P (\lambda-. x) \tau = null \tau$ then null τ else $(\delta S) \tau$) and

($\lambda-. P (\lambda \tau. x) \tau$))

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 | P(z))) = invalid
by(simp add: OclExists-def)

```

```

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

```

4.6.9. OclIterate

```

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

```

```

lemma OclIterateSet-empty[simp,code-unfold]: ((Set{}) -> iterate(a; x = A | P a x)) = A
proof -
  have A1 :  $\llbracket \{\} \rrbracket \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$  by(simp add: mtSet-def)
  have C :  $\bigwedge \tau. (\delta (\lambda \tau. \text{Abs-Set-0 } \llbracket \{\} \rrbracket)) \tau = \text{true } \tau$ 
  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: OclIterateSet-def mtSet-def Abs-Set-0-inverse valid-def C)
    apply(rule ext)
    apply(case-tac A  $\tau = \perp$   $\tau$ , 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 $A = \text{null}$.

```

lemma OclIterateSet-including:

```

```

assumes S-finite:  $\tau \models \delta(S \rightarrow \text{size}())$ 
and F-valid-arg:  $(v\ A)\ \tau = (v\ (F\ a\ A))\ \tau$ 
and F-commute: comp-fun-commute F
and F-cp:  $\bigwedge x\ y\ \tau. F\ x\ y\ \tau = F\ (\lambda\ -. x\ \tau)\ y\ \tau$ 
shows  $((S \rightarrow \text{including}(a)) \rightarrow \text{iterate}(a; x = A \mid F\ a\ x))\ \tau =$ 
 $((S \rightarrow \text{excluding}(a)) \rightarrow \text{iterate}(a; x = F\ a\ A \mid F\ a\ x))\ \tau$ 
proof –

have valid-inject-true :  $\bigwedge \tau\ P. (v\ P)\ \tau \neq \text{true}\ \tau \implies (v\ P)\ \tau = \text{false}\ \tau$ 
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  $\tau = \perp$ , simp-all add: true-def)

have insert-in-Set-0 :  $\bigwedge \tau. (\tau \models (\delta\ S)) \implies (\tau \models (v\ a)) \implies \llbracket \text{insert}\ (a\ \tau)\ [\llbracket \text{Rep-Set-0}\ (S\ \tau) \rrbracket] \rrbracket$ 
 $\in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$ 
apply(frule Set-inv-lemma)
apply(simp add: foundation18 invalid-def)
done

have insert-defined :  $\bigwedge \tau. (\tau \models (\delta\ S)) \implies (\tau \models (v\ a)) \implies$ 
 $(\delta\ (\lambda\ -. \text{Abs-Set-0}\ \llbracket \llbracket \text{insert}\ (a\ \tau)\ [\llbracket \text{Rep-Set-0}\ (S\ \tau) \rrbracket] \rrbracket \rrbracket))\ \tau = \text{true}\ \tau$ 
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  $\llbracket \llbracket \text{Rep-Set-0}\ (S\ \tau) \rrbracket \rrbracket \implies \text{finite}\ ((\lambda a\ \tau. a)\ '(\llbracket \llbracket \text{Rep-Set-0}\ (S\ \tau) \rrbracket \rrbracket - \{a\ \tau\}))$ 
by(simp)

have remove-in-Set-0 :  $\bigwedge \tau. (\tau \models (\delta\ S)) \implies (\tau \models (v\ a)) \implies \llbracket \llbracket \llbracket \text{Rep-Set-0}\ (S\ \tau) \rrbracket \rrbracket - \{a\ \tau\} \rrbracket$ 
 $\in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\}$ 
apply(frule Set-inv-lemma)
apply(simp add: foundation18 invalid-def)
done

have remove-defined :  $\bigwedge \tau. (\tau \models (\delta\ S)) \implies (\tau \models (v\ a)) \implies$ 
 $(\delta\ (\lambda\ -. \text{Abs-Set-0}\ \llbracket \llbracket \llbracket \text{Rep-Set-0}\ (S\ \tau) \rrbracket \rrbracket - \{a\ \tau\} \rrbracket \rrbracket))\ \tau = \text{true}\ \tau$ 
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:  $\bigwedge x. \llbracket x \rrbracket \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \llbracket X \rrbracket. x \neq \text{bot})\} \implies \llbracket \llbracket \text{Rep-Set-0} \text{ (Abs-Set-0 } \llbracket x \rrbracket \rrbracket) \rrbracket = x$ 
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-OclIterateSet[of  $S \rightarrow \text{including}(a)$ ] cp-OclIterateSet[of  $S \rightarrow \text{excluding}(a)$ ])
apply(subst OclIncluding-def, subst OclExcluding-def)
apply(case-tac  $\neg ((\delta S) \tau = \text{true } \tau \wedge (\nu a) \tau = \text{true } \tau)$ , simp)

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

apply(simp add: OclIterateSet-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: OclIterateSet-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: OclIterateSet-def)

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

apply(case-tac  $\neg ((\nu A) \tau = \text{true } \tau)$ , 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  $A = ((\lambda a \tau. a) \text{ ' } (\llbracket \text{Rep-Set-0 } (S \ \tau) \rrbracket) - \{a \ \tau\}))$ , symmetric, OF F-commute])
apply(rule remove-finite, simp)

apply(subst image-set-diff[OF inject], simp)
apply(subgoal-tac Finite-Set.fold  $F \ A \ (\text{insert } (\lambda \tau'. a \ \tau) ((\lambda a \tau. a) \text{ ' } \llbracket \text{Rep-Set-0 } (S \ \tau) \rrbracket))) \ \tau$ 

```

```

=
  F (λτ'. a τ) (Finite-Set.fold F A ((λa τ. a) ' [[Rep-Set-0 (S τ)]] - {λτ'. a τ})) τ)
  apply(subst F-cp)
  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]: OclSelectset mtSet P = mtSet
  apply(rule ext, rename-tac τ)
  apply(simp add: OclSelectset-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]:
  OclSelectset (X->including(y)) P =
  (if δ X then
    if v y then
      if δ(X->size()) then
        if P y then (OclSelectset X P)->including(y)
        else (OclSelectset X P)
      endif
    else invalid
    endif
  else invalid
  endif)
sorry

```

4.6.11. Strict Equality

```

lemma StrictRefEqset-exec[simp,code-unfold] :
  ((x::('A,'α::null)Set) ≐ y) =
  (if δ x then (if δ y
    then ((x->forAll(z| y->includes(z)) and (y->forAll(z| x->includes(z))))
    else if v y
      then false (* x'->includes = null *)
      else invalid
    endif)
  else if v x (* null = ??? *)

```

```

      then if v y then not( $\delta$  y) else invalid endif
    else invalid
    endif
  endif)
proof –

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

  have valid-inject-true :  $\bigwedge \tau P. \neg (\tau \models v P) \implies (v P) \tau = \text{false } \tau$ 
  by (metis bot-fun-def foundation18' valid-def)

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

  have null-simp :  $\bigwedge \tau y. \tau \models v y \implies \neg (\tau \models \delta y) \implies y \tau = \text{null } \tau$ 
  by (simp add: foundation16 foundation18' null-fun-def)

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

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

  have strongeq-false :  $\bigwedge \tau x y. ([x \tau = y \tau]) = \text{false } \tau \implies (x \tau \neq y \tau)$ 
  apply (case-tac x  $\tau \neq y \tau$ , simp add: false-def)
  apply (simp add: false-def true-def)
done

  have rep-set-inj :  $\bigwedge \tau. (\delta x) \tau = \text{true } \tau \implies$ 
     $(\delta y) \tau = \text{true } \tau \implies$ 
     $x \tau \neq y \tau \implies$ 
     $[[\text{Rep-Set-0 } (y \tau)]] \neq [[\text{Rep-Set-0 } (x \tau)]]$ 
  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  $\tau$ )

apply(simp add: cp-OclIf[of  $\delta$   $x$ ])
apply(case-tac  $\neg (\tau \models v \ x)$ )
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 StrictRefEqSet)

apply(simp)

apply(case-tac  $\neg (\tau \models v \ y)$ )
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 StrictRefEqSet)

apply(simp)

apply(simp add: cp-OclIf[of  $\delta$   $y$ ])
apply(simp add: cp-OclIf[symmetric])

apply(simp add: cp-OclIf[of  $\delta$   $x$ ])
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  $\delta$   $y$ ])
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
StrictRefEqSet-vs-StrongEq true-def)
apply(simp add: OclValid-def cp-OclNot[symmetric])

apply(simp add: null-simp[simplified OclValid-def, of  $x$ ] StrictRefEqSet 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  $\delta$   $y$ ])
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-StrictRefEqSet[of x])
apply(simp add: cp-StrictRefEqSet[symmetric])
apply(simp add: null-simp[simplified OclValid-def, of y] StrictRefEqSet 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: StrictRefEqSet StrongEq-def)

apply(subgoal-tac [x  $\tau = y \tau$ ] = true  $\tau \vee$  [x  $\tau = y \tau$ ] = 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 ( $\tau \models \text{OclForall } x (\text{OclIncludes } y) \wedge (\tau \models \text{OclForall } y (\text{OclIncludes } x))$ ))
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)  $\tau = \text{false}$   $\tau \vee \text{OclForall } y (\text{OclIncludes } x) \tau =$ 
false  $\tau$ )
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:  $\text{Set}\{\mathbf{2}, \mathbf{1}\} = (\text{Set}\{\} \rightarrow \text{including}(\mathbf{1}) \rightarrow \text{including}(\mathbf{2}))$   

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  $H: (\text{Set}\{\mathbf{2}\} \doteq \text{null}) = (\text{false}::('A)\text{Boolean})$   

shows  $(\tau::('A)\text{st}) \models (\text{Set}\{\text{Set}\{\mathbf{2}\}, \text{null}\} \rightarrow \text{includes}(\text{null}))$   

by (simp add: includes-execute-set H)
```

```
lemma short-cut'[simp, code-unfold]:  $(\mathbf{8} \doteq \mathbf{6}) = \text{false}$   

apply (rule ext)  

apply (simp add: StrictRefEqInteger StrongEq-def OclInt8-def OclInt6-def  

            true-def false-def invalid-def bot-option-def)  

done
```

```
lemma short-cut''[simp, code-unfold]:  $(\mathbf{2} \doteq \mathbf{1}) = \text{false}$   

apply (rule ext)  

apply (simp add: StrictRefEqInteger StrongEq-def OclInt2-def OclInt1-def  

            true-def false-def invalid-def bot-option-def)  

done
```

```
lemma short-cut'''[simp, code-unfold]:  $(\mathbf{1} \doteq \mathbf{2}) = \text{false}$   

apply (rule ext)  

apply (simp add: StrictRefEqInteger StrongEq-def OclInt2-def OclInt1-def  

            true-def false-def invalid-def bot-option-def)  

done
```

Elementary computations on Sets.

```
value  $\neg (\tau \models v(\text{invalid}::('A, 'a::\text{null}) \text{Set}))$   

value  $\tau \models v(\text{null}::('A, 'a::\text{null}) \text{Set})$   

value  $\neg (\tau \models \delta(\text{null}::('A, 'a::\text{null}) \text{Set}))$   

value  $\tau \models v(\text{Set}\{\})$   

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

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

value  $\tau \models (\text{Set}\{\mathbf{2}, \mathbf{1}\} \rightarrow \text{includes}(\mathbf{1}))$   

value  $\neg (\tau \models (\text{Set}\{\mathbf{2}\} \rightarrow \text{includes}(\mathbf{1})))$   

value  $\neg (\tau \models (\text{Set}\{\mathbf{2}, \mathbf{1}\} \rightarrow \text{includes}(\text{null})))$   

value  $\tau \models (\text{Set}\{\mathbf{2}, \text{null}\} \rightarrow \text{includes}(\text{null}))$   

value  $\tau \models (\text{Set}\{\text{null}, \mathbf{2}\} \rightarrow \text{includes}(\text{null}))$   

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

value  $\tau \models ((\text{Set}\{\mathbf{2}, \mathbf{1}\}) \rightarrow \text{forAll}(z \mid \mathbf{0} <_{\text{ocl}} z))$   

value  $\neg (\tau \models ((\text{Set}\{\mathbf{2}, \mathbf{1}\}) \rightarrow \text{exists}(z \mid z <_{\text{ocl}} \mathbf{0})))$   

value  $\neg (\tau \models \delta(\text{Set}\{\mathbf{2}, \text{null}\}) \rightarrow \text{forAll}(z \mid \mathbf{0} <_{\text{ocl}} z))$ 
```

```

value  $\neg (\tau \models ((Set\{2, null\}) \rightarrow forAll(z \mid 0 <_{ocl} z)))$ 
value  $\tau \models ((Set\{2, null\}) \rightarrow exists(z \mid 0 <_{ocl} z))$ 

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

value  $(\tau \models (Set\{\lambda-. [|x|]\} \doteq Set\{\lambda-. [|x|]\}))$ 
value  $(\tau \models (Set\{\lambda-. [x]\} \doteq Set\{\lambda-. [x]\}))$ 

lemma  $\neg (\tau \models (Set\{true\} \doteq Set\{false\}))$  by simp
lemma  $\neg (\tau \models (Set\{true, true\} \doteq Set\{false\}))$  by simp
lemma  $\neg (\tau \models (Set\{2\} \doteq Set\{1\}))$  by simp
lemma  $\tau \models (Set\{2, null, 2\} \doteq Set\{null, 2\})$  by simp
lemma  $\tau \models (Set\{1, null, 2\} <> Set\{null, 2\})$  by simp
lemma  $\tau \models (Set\{Set\{2, null\}\} \doteq Set\{Set\{null, 2\}\})$  by simp
lemma  $\tau \models (Set\{Set\{2, null\}\} <> Set\{Set\{null, 2\}, null\})$  by simp
lemma  $\neg (\tau \models (Set\{null\} \rightarrow select(x \mid not\ x) \doteq Set\{null\}))$  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 \mathcal{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  $\rightarrow$  '\<AA> "
  assocs  :: "oid  $\rightarrow$  (oid  $\times$  oid) list "
```

```
type_synonym ('\<AA>)st = "'\<AA> state  $\times$  '\<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 '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 $StrictRefEq_{Object} :: ('A, 'a :: \{object, null\})val \Rightarrow ('A, 'a)val \Rightarrow ('A)Boolean$
where $StrictRefEq_{Object} \ x \ y$
 $\equiv \lambda \tau. \text{ if } (v \ x) \ \tau = true \ \tau \wedge (v \ y) \ \tau = true \ \tau$
 $\quad \text{then if } x \ \tau = null \vee y \ \tau = null$
 $\quad \quad \text{then } \llbracket x \ \tau = null \wedge y \ \tau = null \rrbracket$
 $\quad \quad \text{else } \llbracket (oid-of \ (x \ \tau)) = (oid-of \ (y \ \tau)) \rrbracket$
 $\quad \text{else invalid } \tau$

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 $StrictRefEq_{Object}\text{-defargs}$:
 $\tau \models (StrictRefEq_{Object} \ x \ (y :: ('A, 'a :: \{null, object\})val)) \implies (\tau \models (v \ x)) \wedge (\tau \models (v \ y))$
by(*simp add: StrictRefEq_{Object}-def OclValid-def true-def invalid-def bot-option-def*
split: bool.split-asm HOL.split-if-asm)

Symmetry

lemma $StrictRefEq_{Object}\text{-sym}$: **assumes** $x\text{-val} : \tau \models v \ x$ **shows** $\tau \models StrictRefEq_{Object} \ x \ x$
by(*simp add: StrictRefEq_{Object}-def true-def OclValid-def x-val[simplified OclValid-def]*)

Execution with invalid or null as argument

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

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

Context Passing

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

lemmas $cp\text{-}intro''$ [*simp, intro!, code-unfold*] =
 $cp\text{-}intro''$
 $cp\text{-}StrictRefEq_{Object}[THEN \ allI[THEN \ allI[THEN \ allI[THEN \ cpI2]],$
 $\text{ of } StrictRefEq_{Object}]$

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

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 $\text{StrictRefEq}_{Object} \text{-vs-StrongEq}$:
 $WFF \tau \Longrightarrow \tau \models (v \ x) \Longrightarrow \tau \models (v \ y) \Longrightarrow$
 $(x \ \tau \in \text{ran}(\text{heap}(\text{fst } \tau)) \wedge y \ \tau \in \text{ran}(\text{heap}(\text{fst } \tau))) \wedge$
 $(x \ \tau \in \text{ran}(\text{heap}(\text{snd } \tau)) \wedge y \ \tau \in \text{ran}(\text{heap}(\text{snd } \tau))) \Longrightarrow (* \text{ } x \text{ and } y \text{ must be object representations}$
 $\text{that exist in either the pre or post state } *)$
 $(\tau \models (\text{StrictRefEq}_{Object} \ x \ y)) = (\tau \models (x \triangleq y))$
apply(*auto simp: StrictRefEq_{Object}-def OclValid-def WFF-def StrongEq-def true-def Ball-def*)
apply(*erule-tac x=x \tau 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 $\tau_0 :: ('A)st$
where $\tau_0 \equiv ((\text{heap} = \text{Map.empty}, \text{assocs}_2 = \text{Map.empty}, \text{assocs}_3 = \text{Map.empty}),$
 $(\text{heap} = \text{Map.empty}, \text{assocs}_2 = \text{Map.empty}, \text{assocs}_3 = \text{Map.empty}))$

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

definition *OclAllInstances-at-post* :: (' \mathfrak{A} \Rightarrow ' α option) \Rightarrow (' \mathfrak{A} :: object, ' α option option) Set
 (- .allInstances'())

where *OclAllInstances-at-post* $H \tau = \text{OclAllInstances snd } H \tau$

definition *OclAllInstances-at-pre* :: (' \mathfrak{A} \Rightarrow ' α option) \Rightarrow (' \mathfrak{A} :: object, ' α option option) Set
 (- .allInstances@pre'())

where *OclAllInstances-at-pre* $H \tau = \text{OclAllInstances fst } H \tau$

lemma *OclAllInstances-defined*: $\tau \models \delta (X .\text{allInstances}())$

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 $\tau_0 \models H .\text{allInstances}() \triangleq \text{Set}\{\}$

by(simp add: StrongEq-def OclAllInstances-at-post-def OclValid-def τ_0 -def mtSet-def)

lemma $\tau_0 \models H .\text{allInstances}@pre() \triangleq \text{Set}\{\}$

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())

(σ , ($\text{heap} = \text{empty}$, $\text{assocs}_2 = A$, $\text{assocs}_3 = B$))

=

Set{}

(σ , ($\text{heap} = \text{empty}$, $\text{assocs}_2 = A$, $\text{assocs}_3 = B$))

by(simp add: OclAllInstances-at-post-def mtSet-def)

lemma *state-update-vs-allInstances-including'*:

assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$

and Type Object \neq None

shows (Type .allInstances())

(σ , ($\text{heap} = \sigma'(\text{oid} \mapsto \text{Object})$, $\text{assocs}_2 = A$, $\text{assocs}_3 = B$))

=

((Type .allInstances()) \rightarrow including($\lambda -. \llbracket \text{drop (Type Object) } \rrbracket$))

(σ , ($\text{heap} = \sigma'$, $\text{assocs}_2 = A$, $\text{assocs}_3 = B$))

proof –

have *allinst-def* : (σ , ($\text{heap} = \sigma'$, $\text{assocs}_2 = A$, $\text{assocs}_3 = B$)) $\models (\delta (\text{Type .allInstances}()))$


```

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 :  $\bigwedge x. x \neq \text{None} \implies \lfloor x \rfloor = x$ 
by(case-tac x, simp+)

have insert-diff :  $\bigwedge x S. \text{insert } \lfloor x \rfloor (S - \{\text{None}\}) = (\text{insert } \lfloor x \rfloor S) - \{\text{None}\}$ 
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 ( $\sigma'(oid \mapsto \text{Object})$ ) = insert Object (ran  $\sigma'$ ), simp)
apply(case-tac  $\neg (\exists x. \sigma' oid = \text{Some } x)$ )
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  $\bigwedge x. \sigma' oid = \text{Some } x \implies x = \text{Object}$ 
and Type Object  $\neq \text{None}$ 
shows (Type .allInstances())
  ( $\sigma, (\lfloor \text{heap} = \sigma'(oid \mapsto \text{Object}) \rfloor, \text{assocs}_2 = A, \text{assocs}_3 = B \rfloor)$ )
  =
  (( $\lambda -. (Type .allInstances()) (\sigma, (\lfloor \text{heap} = \sigma', \text{assocs}_2 = A, \text{assocs}_3 = B \rfloor))$ )  $\rightarrow$  including( $\lambda -. \lfloor \lfloor$ 
drop (Type Object)  $\rfloor \rfloor$ ))
  ( $\sigma, (\lfloor \text{heap} = \sigma'(oid \mapsto \text{Object}) \rfloor, \text{assocs}_2 = A, \text{assocs}_3 = B \rfloor)$ )
proof -
have allinst-def :  $(\sigma, (\lfloor \text{heap} = \sigma', \text{assocs}_2 = A, \text{assocs}_3 = B \rfloor)) \models (\delta (Type .allInstances()))$ 
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  $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$ 
and Type Object = None
shows (Type .allInstances())
  ( $\sigma, (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}_2=A, \text{assocs}_3=B)$ )
  =
  (Type .allInstances())
  ( $\sigma, (\text{heap}=\sigma', \text{assocs}_2=A, \text{assocs}_3=B)$ )
proof -
have allinst-def : ( $\sigma, (\text{heap} = \sigma', \text{assocs}_2=A, \text{assocs}_3=B)$ )  $\models (\delta \text{ (Type .allInstances())})$ 
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 :  $\bigwedge x. x \neq \text{None} \implies \lfloor x \rfloor = x$ 
by(case-tac x, simp+)

have insert-diff :  $\bigwedge x S. \text{insert } \lfloor x \rfloor (S - \{\text{None}\}) = (\text{insert } \lfloor x \rfloor S) - \{\text{None}\}$ 
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 ( $\sigma'(\text{oid} \mapsto \text{Object})$ ) = insert Object (ran  $\sigma'$ ), simp add: assms)
apply(case-tac  $\neg (\exists x. \sigma' \text{ oid} = \text{Some } x)$ )
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  $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$ 
and Type Object = None
shows (Type .allInstances())

```

```

    (σ, (heap=σ'(oid↦Object), assocs2=A, assocs3=B))
  =
    (λ-. (Type .allInstances()) (σ, (heap=σ', assocs2=A, assocs3=B)))
    (σ, (heap=σ'(oid↦Object), assocs2=A, assocs3=B))
  by (subst state-update-vs-allInstances-noincluding', (simp add: assms)+)

theorem state-update-vs-allInstances:
assumes oid ∉ dom σ'
and cp P
shows ((σ, (heap=σ'(oid↦Object), assocs2=A, assocs3=B)) ⊨ (P (Type .allInstances()))) =
    ((σ, (heap=σ', assocs2=A, assocs3=B)) ⊨ (P ((Type .allInstances()) -> including (λ -.
    [[ drop (Type Object) ] ]))))
proof -
  have P-cp : ∧ x τ. P x τ = P (λ-. x τ) τ
  by (metis (full-types) assms(2) cp-def)
oops

```

```

theorem state-update-vs-allInstances-at-pre:
assumes oid ∉ dom σ
and cp P
shows ((heap=σ(oid↦Object), assocs2=A, assocs3=B), σ') ⊨ (P (Type .allInstances@pre()))
  =
    ((heap=σ, assocs2=A, assocs3=B), σ') ⊨ (P ((Type .allInstances@pre()) -> including (λ
    -. [[ drop (Type Object) ] ]))))
oops

```

5.3.3. OclIsNew

```

definition OclIsNew:: ('A, 'α::{null,object}) val ⇒ ('A) Boolean ((-).oclIsNew'())
where X .oclIsNew() ≡ (λτ . if (δ X) τ = true τ
    then [[oid-of (X τ) ∉ dom(heap(fst τ)) ∧
    oid-of (X τ) ∈ dom(heap(snd τ))]]
    else invalid τ)

```

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

```

definition OclIsOld:: ('A, 'α::{null,object}) val ⇒ ('A) Boolean ((-).oclIsOld'())
where X .oclIsOld() ≡ (λτ . if (δ X) τ = true τ
    then [[oid-of (X τ) ∈ dom(heap(fst τ)) ∧
    oid-of (X τ) ∉ dom(heap(snd τ))]]
    else invalid τ)

```

```

definition OclIsMaintained:: ('A, 'α::{null,object}) val ⇒ ('A) Boolean ((-).oclIsMaintained'())
where X .oclIsMaintained() ≡ (λτ . if (δ X) τ = true τ
    then [[oid-of (X τ) ∈ dom(heap(fst τ)) ∧
    oid-of (X τ) ∈ dom(heap(snd τ))]]
    else invalid τ)

```

```

definition OclIsAbsent:: ('A, 'α::{null,object}) val ⇒ ('A) Boolean ((-).oclIsAbsent'())

```

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

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

lemma *notNew-vs-others* : $\tau \models \delta X \implies (\neg \tau \models (X.\text{oclIsNew}())) = (\tau \models (X.\text{oclIsOld}()) \vee \tau \models (X.\text{oclIsMaintained}()) \vee \tau \models (X.\text{oclIsAbsent}()))$
by(*simp add: OclIsOld-def OclIsNew-def OclIsMaintained-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* :: ($\mathcal{A}::\text{object}, 'a::\{\text{null}, \text{object}\}\rangle \text{Set} \Rightarrow \mathcal{A} \text{ Boolean}$
 $(-\>\text{oclIsModifiedOnly}')()$)
where $X-\>\text{oclIsModifiedOnly}() \equiv (\lambda(\sigma, \sigma'). \text{let } X' = (\text{oid-of } ' \llbracket \text{Rep-Set-0}(X(\sigma, \sigma')) \rrbracket \rrbracket);$
 $S = ((\text{dom}(\text{heap } \sigma) \cap \text{dom}(\text{heap } \sigma')) - X')$
 $\text{in if } (\delta X) (\sigma, \sigma') = \text{true } (\sigma, \sigma') \wedge (\forall x \in \llbracket \text{Rep-Set-0}(X(\sigma, \sigma')) \rrbracket .$
 $x \neq \text{null})$
 $\text{then } \llbracket \forall x \in S. (\text{heap } \sigma) x = (\text{heap } \sigma') x \rrbracket$
 $\text{else invalid } (\sigma, \sigma')$)

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

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

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

definition *OclSelf-at-post* :: ($\mathcal{A}::\text{object}, 'a::\{\text{null}, \text{object}\}\text{val} \Rightarrow$
 $(\mathcal{A} \Rightarrow 'a) \Rightarrow$
 $(\mathcal{A}::\text{object}, 'a::\{\text{null}, \text{object}\}\text{val} ((-)\text{@post}(-))$
where $x \text{@post } H = \text{OclSelf } x \text{ H snd}$

lemma *all-oid-diff*:

assumes $\text{def-}x : \tau \models \delta \ x$
assumes $\text{def-}X : \tau \models \delta \ X$
assumes $\text{def-}X' : \bigwedge x. x \in [[\text{Rep-Set-0 } (X \ \tau)]] \Rightarrow x \neq \text{null}$

defines $P \equiv (\lambda a. \text{not } (\text{StrictRefEq}_{\text{Object}} \ x \ a))$

shows $(\tau \models X \rightarrow \text{forAll}(a \mid P \ a)) = (\text{oid-of } (x \ \tau) \notin \text{oid-of } ' [[\text{Rep-Set-0 } (X \ \tau)]])$

proof –

have $\text{discr-eq-false-true} : \bigwedge \tau. (\text{false } \tau = \text{true } \tau) = \text{False}$ **by** (*metis OclValid-def foundation2*)
have $\text{discr-eq-bot1-true} : \bigwedge \tau. (\perp \ \tau = \text{true } \tau) = \text{False}$ **by** (*metis defined3 defined-def discr-eq-false-true*)
have $\text{discr-eq-bot2-true} : \bigwedge \tau. (\perp = \text{true } \tau) = \text{False}$ **by** (*metis bot-fun-def discr-eq-bot1-true*)
have $\text{discr-eq-null-true} : \bigwedge \tau. (\text{null } \tau = \text{true } \tau) = \text{False}$ **by** (*metis OclValid-def foundation4*)

have $P\text{-null} : \neg (\exists x \in [[\text{Rep-Set-0 } (X \ \tau)]] . P \ (\lambda(-:: 'a \ \text{state} \times 'a \ \text{state}). x) \ \tau = \text{null } \tau)$

apply(*simp, rule ballI*)
apply(*simp add: P-def StrictRefEq_{Object}-def*)
apply(*subst cp-OclNot, simp*)
apply(*subgoal-tac x \ \tau \neq \text{null} \wedge xa \neq \text{null}, simp*)
apply(*simp add: OclNot-def null-fun-def null-option-def bot-option-def invalid-def*)
by (*metis def-X' def-x foundation17*)

have $P\text{-bot} : \neg (\exists x \in [[\text{Rep-Set-0 } (X \ \tau)]] . P \ (\lambda(-:: 'a \ \text{state} \times 'a \ \text{state}). x) \ \tau = \perp \ \tau)$

apply(*simp, rule ballI*)
apply(*simp add: P-def StrictRefEq_{Object}-def*)
apply(*subst cp-OclNot, simp*)
apply(*subgoal-tac x \ \tau \neq \text{null} \wedge xa \neq \text{null}, simp*)
apply(*simp add: OclNot-def null-fun-def null-option-def bot-option-def bot-fun-def invalid-def*)
apply (*metis OCL-core.bot-fun-def OclValid-def Set-inv-lemma def-X def-x defined-def valid-def*)
by (*metis def-X' def-x foundation17*)

have $\text{not-inj} : \bigwedge x \ y. ((\text{not } x) \ \tau = (\text{not } y) \ \tau) = (x \ \tau = y \ \tau)$

by (*metis foundation21 foundation22*)

have $P\text{-false} : \exists x \in [[\text{Rep-Set-0 } (X \ \tau)]] . P \ (\lambda-. x) \ \tau = \text{false } \tau \Rightarrow \text{oid-of } (x \ \tau) \in \text{oid-of } ' [[\text{Rep-Set-0 } (X \ \tau)]]$

apply(*erule bexE, rename-tac x'*)
apply(*simp add: P-def*)
apply(*simp only: OclNot3[symmetric], simp only: not-inj*)
apply(*simp add: StrictRefEq_{Object}-def split: split-if-asm*)
apply(*subgoal-tac x \ \tau \neq \text{null} \wedge x' \neq \text{null}, simp*)
apply (*metis (mono-tags) OCL-core.drop.simps def-x foundation17 true-def*)
by(*simp add: invalid-def bot-option-def true-def*)+

have $P\text{-true} : \forall x \in [[\text{Rep-Set-0 } (X \ \tau)]] . P \ (\lambda-. x) \ \tau = \text{true } \tau \Rightarrow \text{oid-of } (x \ \tau) \notin \text{oid-of } ' [[\text{Rep-Set-0 } (X \ \tau)]]$

```

[[Rep-Set-0 (X τ)]]
  apply(subgoal-tac ∀ x' ∈ [[Rep-Set-0 (X τ)]] . oid-of x' ≠ oid-of (x τ))
  apply (metis imageE)
  apply(rule ballI, drule-tac x = x' in ballE) prefer 3 apply assumption
  apply(simp add: P-def)
  apply(simp only: OclNot4[symmetric], simp only: not-inj)
  apply(simp add: StrictRefEqObject-def false-def split: split-if-asm)
  apply(subgoal-tac x τ ≠ null ∧ x' ≠ null, simp)
  apply (metis def-X' def-x foundation17)
by(simp add: invalid-def bot-option-def false-def)+

have bool-split : ∀ x ∈ [[Rep-Set-0 (X τ)]] . P (λ-. x) τ ≠ null τ ⇒
  ∀ x ∈ [[Rep-Set-0 (X τ)]] . P (λ-. x) τ ≠ ⊥ τ ⇒
  ∀ x ∈ [[Rep-Set-0 (X τ)]] . P (λ-. x) τ ≠ false τ ⇒
  ∀ x ∈ [[Rep-Set-0 (X τ)]] . P (λ-. x) τ = true τ
  apply(rule ballI)
  apply(drule-tac x = x in ballE) prefer 3 apply assumption
  apply(drule-tac x = x in ballE) prefer 3 apply assumption
  apply(drule-tac x = x in ballE) prefer 3 apply assumption
  apply (metis (full-types) OCL-core.bot-fun-def OclNot4 OclValid-def foundation16 founda-
tion18' foundation9 not-inj null-fun-def)
by(simp-all)

show ?thesis
  apply(simp add: OclForall-def OclValid-def)
  apply(simp-all add: discr-eq-false-true discr-eq-bot1-true discr-eq-null-true discr-eq-bot2-true)+
  apply(simp add: P-null P-bot)
  apply((rule impI)+ | (rule conjI)+)+
  apply(rule P-false, simp)

  apply(rule impI, simp add: def-X[simplified OclValid-def])
  apply(rule P-true)
  apply(drule bool-split)
by simp
qed

theorem framing:
  assumes modifiesclause: τ ⊨ (X -> excluding(x :: ('A::object, 'α::{'null, object'}) val)) -> oclIsModifiedOnly()
  and   represented-x: τ ⊨ δ(x @pre (H::('A ⇒ 'α)))
  and   oid-is-typerepr : τ ⊨ X -> forAll(a | not (StrictRefEqObject x a))
  shows τ ⊨ (x @pre H ≜ (x @post H))
proof -
  have def-x : τ ⊨ δ 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)
  have def-X : τ ⊨ δ X
  apply(insert oid-is-typerepr, simp add: OclForall-def OclValid-def split: split-if-asm)
  by(simp add: bot-option-def true-def)
  have def-X' : ∧x. x ∈ [[Rep-Set-0 (X τ)]] ⇒ x ≠ null

```

```

apply(insert modifiesclause, simp add: OclIsModifiedOnly-def OclValid-def split: split-if-asm)
apply(case-tac  $\tau$ , simp split: split-if-asm)
apply(simp add: OclExcluding-def split: split-if-asm)
apply(subst (asm) (2) Abs-Set-0-inverse)
apply(simp, (rule disjI2)+)
apply (metis (hide-lams, mono-tags) Diff-iff Set-inv-lemma def-X)
apply(simp)
apply(erule ballE[where  $P = \lambda x. x \neq \text{null}$ ]) apply(assumption)
apply(simp)
apply (metis (hide-lams, no-types) def-x foundation17)
apply (metis (hide-lams, no-types) OclValid-def def-X def-x excluding-valid-args-valid excluding-valid-args-valid''
foundation20)
by(simp add: invalid-def bot-option-def)
have oid-is-typerepr : oid-of ( $x \ \tau$ )  $\notin$  oid-of '  $\llbracket \text{Rep-Set-0 } (X \ \tau) \rrbracket$ 
by(rule all-oid-diff[THEN iffD1, OF def-x def-X def-X' oid-is-typerepr])

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  $f = \lambda x. H \llbracket x \rrbracket$  in arg-cong)
apply(insert modifiesclause[simplified OclIsModifiedOnly-def OclValid-def])
apply(case-tac  $\tau$ , rename-tac  $\sigma \ \sigma'$ , simp split: split-if-asm)
apply(subst (asm) (2) OclExcluding-def)
apply(erule 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  $x = \text{oid-of } (x \ (\sigma, \sigma'))$  in ballE) apply simp+
apply (metis (hide-lams, no-types) DiffD1 image-iff image-insert insert-Diff-single insert-absorb
oid-is-typerepr)
apply(simp add: invalid-def bot-option-def)+
by blast
qed

lemma pre-post-new:  $\tau \models (x \ .\text{oclIsNew}()) \implies \neg (\tau \models v(x \ @\text{pre } H1)) \wedge \neg (\tau \models v(x \ @\text{post } H2))$ 
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:  $\tau \models (x \ .\text{oclIsOld}()) \implies \neg (\tau \models v(x \ @\text{pre } H1)) \wedge \neg (\tau \models v(x \ @\text{post } H2))$ 
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*: $\tau \models (x .oclIsAbsent()) \implies \neg (\tau \models v(x @pre H1)) \wedge \neg (\tau \models v(x @post H2))$

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-maintained*: $(\tau \models v(x @pre H1) \vee \tau \models v(x @post H2)) \implies \tau \models (x .oclIsMaintained())$

by(*simp add: OclIsMaintained-def OclSelf-at-pre-def OclSelf-at-post-def*
OclValid-def StrongEq-def true-def false-def
bot-option-def invalid-def bot-fun-def valid-def
split: split-if-asm)

lemma *pre-post-maintained'*: $\tau \models (x .oclIsMaintained()) \implies (\tau \models v(x @pre (Some o H1)) \wedge \tau \models v(x @post (Some o H2)))$

by(*simp add: OclIsMaintained-def OclSelf-at-pre-def OclSelf-at-post-def*
OclValid-def StrongEq-def true-def false-def
bot-option-def invalid-def bot-fun-def valid-def
split: split-if-asm)

lemma *framing-same-state*: $(\sigma, \sigma) \models (x @pre H \triangleq (x @post H))$

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

6. Part IV: OCL Contracts and an Example

```
theory
  Employee-AnalysisModel-UMLPart
imports
  ../src/OCL-main
begin
```

6.0.5. Introduction

For certain concepts like Classes and Class-types, only a generic definition for its resulting semantics can be given. Generic means, there is a function outside HOL that "compiles" a concrete, closed-world class diagram into a "theory" of this data model, consisting of a bunch of definitions for classes, accessors, method, casts, and tests for actual types, as well as proofs for the fundamental properties of these operations in this concrete data model.

Such generic function or "compiler" can be implemented in Isabelle on the ML level. This has been done, for a semantics following the open-world assumption, for UML 2.0 in [9]. In this paper, we follow another approach for UML 2.4: we define the concepts of the compilation informally, and present a concrete example which is verified in Isabelle/HOL.

6.0.6. Outlining the Example

We are presenting here a design-model of the (slightly modified) example Figure 7.3, page 20 of the [?]. To be precise, this theory contains the formalization of the DATA-part covered by the UML diagram: ... This means that the association (attached to the association class *EmployeeRanking*) with the association ends **boss** and **employees** is implemented by the attribute **boss** and the operation **employees** (to be discussed in the OCL part captured by the subsequent theory).

6.1. Example Data-Universe and its Infrastructure

Should be generated entirely from a class-diagram.

Our data universe consists in the concrete class diagram just of node's, and implicitly of the class object. Each class implies the existence of a class type defined for the corresponding object representations as follows:

```
datatype typePerson = mkPerson oid
                      int option
```

```
datatype typeOclAny = mkOclAny oid
                     (int option) option
```

Now, we construct a concrete "universe of oclany types" by injection into a sum type containing the class types. This type of oclanys will be used as instance for all resp. type-variables ...

```
datatype  $\mathfrak{A}$  = inPerson typePerson | inOclAny typeOclAny
```

Having fixed the object universe, we can introduce type synonyms that exactly correspond to OCL types. Again, we exploit that our representation of OCL is a "shallow embedding" with a one-to-one correspondance of OCL-types to types of the meta-language HOL.

```
type-synonym Boolean    =  $\mathfrak{A}$  Boolean
type-synonym Integer   =  $\mathfrak{A}$  Integer
type-synonym Void     =  $\mathfrak{A}$  Void
type-synonym OclAny    = ( $\mathfrak{A}$ , typeOclAny option option) val
type-synonym Person    = ( $\mathfrak{A}$ , typePerson option option) val
type-synonym Set-Integer = ( $\mathfrak{A}$ , int option option) Set
type-synonym Set-Person = ( $\mathfrak{A}$ , typePerson option option) Set
```

Just a little check:

```
typ Boolean
```

In order to reuse key-elements of the library like referential equality, we have to show that the object universe belongs to the type class "oclany", i.e. each class type has to provide a function *oid-of* yielding the object id (oid) of the object.

```
instantiation typePerson :: object
begin
  definition oid-of-typePerson-def: oid-of x = (case x of mkPerson oid -  $\Rightarrow$  oid)
  instance ..
end
```

```
instantiation typeOclAny :: object
begin
  definition oid-of-typeOclAny-def: oid-of x = (case x of mkOclAny oid -  $\Rightarrow$  oid)
  instance ..
end
```

```
instantiation  $\mathfrak{A}$  :: object
begin
  definition oid-of- $\mathfrak{A}$ -def: oid-of x = (case x of
                                     inPerson person  $\Rightarrow$  oid-of person
                                     | inOclAny oclany  $\Rightarrow$  oid-of oclany)
  instance ..
end
```

6.2. Instantiation of the generic strict equality. We instantiate the referential equality on *Person* and *OclAny*

```

defs(overloaded) StrictRefEqObject-Person : (x::Person)  $\doteq$  y  $\equiv$  StrictRefEqObject x y
defs(overloaded) StrictRefEqObject-OclAny : (x::OclAny)  $\doteq$  y  $\equiv$  StrictRefEqObject x y

```

lemmas

```

cp-StrictRefEqObject [of x::Person y::Person  $\tau$ ,
                        simplified StrictRefEqObject-Person [symmetric]]
cp-intro(9)          [of P::Person  $\Rightarrow$  PersonQ::Person  $\Rightarrow$  Person,
                        simplified StrictRefEqObject-Person [symmetric]]
StrictRefEqObject-def [of x::Person y::Person,
                        simplified StrictRefEqObject-Person [symmetric]]
StrictRefEqObject-defargs [of - x::Person y::Person,
                             simplified StrictRefEqObject-Person [symmetric]]
StrictRefEqObject-strict1
  [of x::Person,
   simplified StrictRefEqObject-Person [symmetric]]
StrictRefEqObject-strict2
  [of x::Person,
   simplified StrictRefEqObject-Person [symmetric]]

```

For each Class *C*, we will have a casting operation *.oclAsType*(*C*), a test on the actual type *.oclIsTypeOf*(*C*) as well as its relaxed form *.oclIsKindOf*(*C*) (corresponding exactly to Java's *instanceof*-operator).

Thus, since we have two class-types in our concrete class hierarchy, we have two operations to declare and to provide two overloading definitions for the two static types.

6.3. OclAsType

6.3.1. Definition

```

consts OclAsTypeOclAny :: ' $\alpha \Rightarrow$  OclAny ((-) .oclAsType'(OclAny'))
consts OclAsTypePerson :: ' $\alpha \Rightarrow$  Person ((-) .oclAsType'(Person'))

```

definition *OclAsTypeOclAny- \mathfrak{A}* = ($\lambda u.$ [*case u of in*_{*OclAny*} *a* \Rightarrow *a*
| in_{*Person*} (*mkPerson* *oid a*) \Rightarrow *mkOclAny* *oid* [*a*]])

lemma *OclAsTypeOclAny- \mathfrak{A} -some*: *OclAsTypeOclAny- \mathfrak{A}* *x* \neq *None*
by(*simp add: OclAsTypeOclAny- \mathfrak{A} -def*)

```

defs (overloaded) OclAsTypeOclAny-OclAny:
  (X::OclAny) .oclAsType(OclAny)  $\equiv$  X

```

```

defs (overloaded) OclAsTypeOclAny-Person:
  (X::Person) .oclAsType(OclAny)  $\equiv$ 
    ( $\lambda \tau.$  case X  $\tau$  of

```

$$\begin{aligned}
& \perp \Rightarrow \text{invalid } \tau \\
& | \lfloor \perp \rfloor \Rightarrow \text{null } \tau \\
& | \lfloor \lfloor \text{mk}_{\text{Person}} \text{ oid } a \rfloor \rfloor \Rightarrow \lfloor \lfloor \text{mk}_{\text{OclAny}} \text{ oid } \lfloor a \rfloor \rfloor \rfloor
\end{aligned}$$

definition $\text{OclAsType}_{\text{Person}}\text{-}\mathfrak{A} = (\lambda u. \text{case } u \text{ of } \text{in}_{\text{Person}} p \Rightarrow \lfloor p \rfloor$
 $| \text{in}_{\text{OclAny}} (\text{mk}_{\text{OclAny}} \text{ oid } \lfloor a \rfloor) \Rightarrow \lfloor \text{mk}_{\text{Person}} \text{ oid } a \rfloor$
 $| - \Rightarrow \text{None})$

defs (overloaded) $\text{OclAsType}_{\text{Person}}\text{-OclAny}$:
 $(X::\text{OclAny}) .\text{oclAsType}(\text{Person}) \equiv$
 $(\lambda \tau. \text{case } X \text{ } \tau \text{ of}$
 $\perp \Rightarrow \text{invalid } \tau$
 $| \lfloor \perp \rfloor \Rightarrow \text{null } \tau$
 $| \lfloor \lfloor \text{mk}_{\text{OclAny}} \text{ oid } \perp \rfloor \rfloor \Rightarrow \text{invalid } \tau \quad (* \text{down-cast exception} *)$
 $| \lfloor \lfloor \text{mk}_{\text{OclAny}} \text{ oid } \lfloor a \rfloor \rfloor \rfloor \Rightarrow \lfloor \lfloor \text{mk}_{\text{Person}} \text{ oid } a \rfloor \rfloor)$

defs (overloaded) $\text{OclAsType}_{\text{Person}}\text{-Person}$:
 $(X::\text{Person}) .\text{oclAsType}(\text{Person}) \equiv X$

lemmas $[\text{simp}] =$
 $\text{OclAsType}_{\text{OclAny}}\text{-OclAny}$
 $\text{OclAsType}_{\text{Person}}\text{-Person}$

6.3.2. Context Passing

lemma $\text{cp-OclAsType}_{\text{OclAny}}\text{-Person-Person}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P (X::\text{Person}) :: \text{Person}) .\text{oclAsType}(\text{OclAny}))$
by(rule cpI1 , simp-all add: $\text{OclAsType}_{\text{OclAny}}\text{-Person}$)
lemma $\text{cp-OclAsType}_{\text{OclAny}}\text{-OclAny-OclAny}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P (X::\text{OclAny}) :: \text{OclAny}) .\text{oclAsType}(\text{OclAny}))$
by(rule cpI1 , simp-all add: $\text{OclAsType}_{\text{OclAny}}\text{-OclAny}$)
lemma $\text{cp-OclAsType}_{\text{Person}}\text{-Person-Person}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P (X::\text{Person}) :: \text{Person}) .\text{oclAsType}(\text{Person}))$
by(rule cpI1 , simp-all add: $\text{OclAsType}_{\text{Person}}\text{-Person}$)
lemma $\text{cp-OclAsType}_{\text{Person}}\text{-OclAny-OclAny}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P (X::\text{OclAny}) :: \text{OclAny}) .\text{oclAsType}(\text{Person}))$
by(rule cpI1 , simp-all add: $\text{OclAsType}_{\text{Person}}\text{-OclAny}$)
lemma $\text{cp-OclAsType}_{\text{OclAny}}\text{-Person-OclAny}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P (X::\text{Person}) :: \text{OclAny}) .\text{oclAsType}(\text{OclAny}))$
by(rule cpI1 , simp-all add: $\text{OclAsType}_{\text{OclAny}}\text{-OclAny}$)
lemma $\text{cp-OclAsType}_{\text{OclAny}}\text{-OclAny-Person}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P (X::\text{OclAny}) :: \text{Person}) .\text{oclAsType}(\text{OclAny}))$
by(rule cpI1 , simp-all add: $\text{OclAsType}_{\text{OclAny}}\text{-Person}$)
lemma $\text{cp-OclAsType}_{\text{Person}}\text{-Person-OclAny}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P (X::\text{Person}) :: \text{OclAny}) .\text{oclAsType}(\text{Person}))$
by(rule cpI1 , simp-all add: $\text{OclAsType}_{\text{Person}}\text{-OclAny}$)
lemma $\text{cp-OclAsType}_{\text{Person}}\text{-OclAny-Person}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P (X::\text{OclAny}) :: \text{Person}) .\text{oclAsType}(\text{Person}))$
by(rule cpI1 , simp-all add: $\text{OclAsType}_{\text{Person}}\text{-Person}$)

```

lemmas [simp] =
  cp-OclAsTypeOclAny-Person-Person
  cp-OclAsTypeOclAny-OclAny-OclAny
  cp-OclAsTypePerson-Person-Person
  cp-OclAsTypePerson-OclAny-OclAny

  cp-OclAsTypeOclAny-Person-OclAny
  cp-OclAsTypeOclAny-OclAny-Person
  cp-OclAsTypePerson-Person-OclAny
  cp-OclAsTypePerson-OclAny-Person

```

6.3.3. Execution with invalid or null as argument

```

lemma OclAsTypeOclAny-OclAny-strict : (invalid::OclAny) .oclAsType(OclAny) = invalid
by(simp)

```

```

lemma OclAsTypeOclAny-OclAny-nullstrict : (null::OclAny) .oclAsType(OclAny) = null
by(simp)

```

```

lemma OclAsTypeOclAny-Person-strict[simp] : (invalid::Person) .oclAsType(OclAny) = invalid
by(rule ext, simp add: bot-option-def invalid-def
    OclAsTypeOclAny-Person)

```

```

lemma OclAsTypeOclAny-Person-nullstrict[simp] : (null::Person) .oclAsType(OclAny) = null
by(rule ext, simp add: null-fun-def null-option-def bot-option-def
    OclAsTypeOclAny-Person)

```

```

lemma OclAsTypePerson-OclAny-strict[simp] : (invalid::OclAny) .oclAsType(Person) = invalid
by(rule ext, simp add: bot-option-def invalid-def
    OclAsTypePerson-OclAny)

```

```

lemma OclAsTypePerson-OclAny-nullstrict[simp] : (null::OclAny) .oclAsType(Person) = null
by(rule ext, simp add: null-fun-def null-option-def bot-option-def
    OclAsTypePerson-OclAny)

```

```

lemma OclAsTypePerson-Person-strict : (invalid::Person) .oclAsType(Person) = invalid
by(simp)

```

```

lemma OclAsTypePerson-Person-nullstrict : (null::Person) .oclAsType(Person) = null
by(simp)

```

6.4. OclIsTypeOf

6.4.1. Definition

```

consts OclIsTypeOfOclAny :: 'α ⇒ Boolean ((-) .oclIsTypeOf '(OclAny'))
consts OclIsTypeOfPerson :: 'α ⇒ Boolean ((-) .oclIsTypeOf '(Person'))

```

```

defs (overloaded) OclIsTypeOfOclAny-OclAny:

```

$$\begin{aligned}
& (X::OclAny) .oclIsTypeOf(OclAny) \equiv \\
& \quad (\lambda\tau. \text{ case } X \ \tau \text{ of} \\
& \quad \quad \perp \Rightarrow \text{invalid } \tau \\
& \quad \quad | \lfloor \perp \rfloor \Rightarrow \text{true } \tau \quad (* \text{ invalid } ?? *) \\
& \quad \quad | \lfloor \lfloor mk_{OclAny} \text{ oid } \perp \rfloor \rfloor \Rightarrow \text{true } \tau \\
& \quad \quad | \lfloor \lfloor mk_{OclAny} \text{ oid } \lfloor - \rfloor \rfloor \rfloor \Rightarrow \text{false } \tau)
\end{aligned}$$

defs (overloaded) $OclIsTypeOf_{OclAny-Person}$:

$$\begin{aligned}
& (X::Person) .oclIsTypeOf(OclAny) \equiv \\
& \quad (\lambda\tau. \text{ case } X \ \tau \text{ of} \\
& \quad \quad \perp \Rightarrow \text{invalid } \tau \\
& \quad \quad | \lfloor \perp \rfloor \Rightarrow \text{true } \tau \quad (* \text{ invalid } ?? *) \\
& \quad \quad | \lfloor \lfloor - \rfloor \rfloor \Rightarrow \text{false } \tau)
\end{aligned}$$

defs (overloaded) $OclIsTypeOf_{Person-OclAny}$:

$$\begin{aligned}
& (X::OclAny) .oclIsTypeOf(Person) \equiv \\
& \quad (\lambda\tau. \text{ case } X \ \tau \text{ of} \\
& \quad \quad \perp \Rightarrow \text{invalid } \tau \\
& \quad \quad | \lfloor \perp \rfloor \Rightarrow \text{true } \tau \\
& \quad \quad | \lfloor \lfloor mk_{OclAny} \text{ oid } \perp \rfloor \rfloor \Rightarrow \text{false } \tau \\
& \quad \quad | \lfloor \lfloor mk_{OclAny} \text{ oid } \lfloor - \rfloor \rfloor \rfloor \Rightarrow \text{true } \tau)
\end{aligned}$$

defs (overloaded) $OclIsTypeOf_{Person-Person}$:

$$\begin{aligned}
& (X::Person) .oclIsTypeOf(Person) \equiv \\
& \quad (\lambda\tau. \text{ case } X \ \tau \text{ of} \\
& \quad \quad \perp \Rightarrow \text{invalid } \tau \\
& \quad \quad | - \Rightarrow \text{true } \tau)
\end{aligned}$$

6.4.2. Context Passing

lemma $cp\text{-}OclIsTypeOf_{OclAny-Person-Person}$: $cp \ P \implies cp(\lambda X. (P \ (X::Person) :: Person) .oclIsTypeOf(OclAny))$

by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny-Person}$)

lemma $cp\text{-}OclIsTypeOf_{OclAny-OclAny-OclAny}$: $cp \ P \implies cp(\lambda X. (P \ (X::OclAny) :: OclAny) .oclIsTypeOf(OclAny))$

by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny-OclAny}$)

lemma $cp\text{-}OclIsTypeOf_{Person-Person-Person}$: $cp \ P \implies cp(\lambda X. (P \ (X::Person) :: Person) .oclIsTypeOf(Person))$

by(rule $cpI1$, simp-all add: $OclIsTypeOf_{Person-Person}$)

lemma $cp\text{-}OclIsTypeOf_{Person-OclAny-OclAny}$: $cp \ P \implies cp(\lambda X. (P \ (X::OclAny) :: OclAny) .oclIsTypeOf(Person))$

by(rule $cpI1$, simp-all add: $OclIsTypeOf_{Person-OclAny}$)

lemma $cp\text{-}OclIsTypeOf_{OclAny-Person-OclAny}$: $cp \ P \implies cp(\lambda X. (P \ (X::Person) :: OclAny) .oclIsTypeOf(OclAny))$

by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny-OclAny}$)

lemma $cp\text{-}OclIsTypeOf_{OclAny-OclAny-Person}$: $cp \ P \implies cp(\lambda X. (P \ (X::OclAny) :: Person) .oclIsTypeOf(Person))$


```

.ocIsTypeOf (OclAny))
by(rule cpI1, simp-all add: OcIsTypeOfOclAny-Person)
lemma cp-OcIsTypeOfPerson-Person-OclAny: cp P  $\implies$  cp( $\lambda X. (P (X::Person) :: OclAny)$ 
.ocIsTypeOf (Person))
by(rule cpI1, simp-all add: OcIsTypeOfPerson-OclAny)
lemma cp-OcIsTypeOfPerson-OclAny-Person: cp P  $\implies$  cp( $\lambda X. (P (X::OclAny) :: Person)$ 
.ocIsTypeOf (Person))
by(rule cpI1, simp-all add: OcIsTypeOfPerson-Person)

lemmas [simp] =
  cp-OcIsTypeOfOclAny-Person-Person
  cp-OcIsTypeOfOclAny-OclAny-OclAny
  cp-OcIsTypeOfPerson-Person-Person
  cp-OcIsTypeOfPerson-OclAny-OclAny

  cp-OcIsTypeOfOclAny-Person-OclAny
  cp-OcIsTypeOfOclAny-OclAny-Person
  cp-OcIsTypeOfPerson-Person-OclAny
  cp-OcIsTypeOfPerson-OclAny-Person

```

6.4.3. Execution with invalid or null as argument

```

lemma OcIsTypeOfOclAny-OclAny-strict1[simp]:
  (invalid::OclAny) .ocIsTypeOf (OclAny) = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OcIsTypeOfOclAny-OclAny)
lemma OcIsTypeOfOclAny-OclAny-strict2[simp]:
  (null::OclAny) .ocIsTypeOf (OclAny) = true
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OcIsTypeOfOclAny-OclAny)
lemma OcIsTypeOfOclAny-Person-strict1[simp]:
  (invalid::Person) .ocIsTypeOf (OclAny) = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OcIsTypeOfOclAny-Person)
lemma OcIsTypeOfOclAny-Person-strict2[simp]:
  (null::Person) .ocIsTypeOf (OclAny) = true
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OcIsTypeOfOclAny-Person)
lemma OcIsTypeOfPerson-OclAny-strict1[simp]:
  (invalid::OclAny) .ocIsTypeOf (Person) = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OcIsTypeOfPerson-OclAny)
lemma OcIsTypeOfPerson-OclAny-strict2[simp]:
  (null::OclAny) .ocIsTypeOf (Person) = true
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OcIsTypeOfPerson-OclAny)
lemma OcIsTypeOfPerson-Person-strict1[simp]:
  (invalid::Person) .ocIsTypeOf (Person) = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def

```

```

      OclIsTypeOfPerson-Person)
lemma OclIsTypeOfPerson-Person-strict2[simp]:
  (null::Person) .oclIsTypeOf(Person) = true
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
      OclIsTypeOfPerson-Person)

```

6.4.4. up down cast

```

lemma actualType-larger-staticType:
assumes isdef:  $\tau \models (\delta X)$ 
shows  $\tau \models (X::Person) .oclIsTypeOf(OclAny) \triangleq false$ 
using isdef
by(auto simp : null-option-def bot-option-def
      OclIsTypeOfOclAny-Person foundation22 foundation16)

lemma down-cast-type:
assumes isOclAny:  $\tau \models (X::OclAny) .oclIsTypeOf(OclAny)$ 
and non-null:  $\tau \models (\delta X)$ 
shows  $\tau \models (X .oclAsType(Person)) \triangleq invalid$ 
using isOclAny non-null
apply(auto simp : bot-fun-def null-fun-def null-option-def bot-option-def null-def invalid-def
      OclAsTypeOclAny-Person OclAsTypePerson-OclAny foundation22 foundation16
      split: option.split typeOclAny.split typePerson.split)
by(simp add: OclIsTypeOfOclAny-OclAny OclValid-def false-def true-def)

```

```

lemma down-cast-type':
assumes isOclAny:  $\tau \models (X::OclAny) .oclIsTypeOf(OclAny)$ 
and non-null:  $\tau \models (\delta X)$ 
shows  $\tau \models not (v (X .oclAsType(Person)))$ 
by(rule foundation15[THEN iffD1], simp add: down-cast-type[OF assms])

```

```

lemma up-down-cast :
assumes isdef:  $\tau \models (\delta X)$ 
shows  $\tau \models ((X::Person) .oclAsType(OclAny) .oclAsType(Person) \triangleq X)$ 
using isdef
by(auto simp : null-fun-def null-option-def bot-option-def null-def invalid-def
      OclAsTypeOclAny-Person OclAsTypePerson-OclAny foundation22 foundation16
      split: option.split typePerson.split)

```

```

lemma up-down-cast-Person-OclAny-Person [simp]:
shows  $((X::Person) .oclAsType(OclAny) .oclAsType(Person) = X)$ 
apply(rule ext, rename-tac  $\tau$ )
apply(rule foundation22[THEN iffD1])
apply(case-tac  $\tau \models (\delta X)$ , simp add: up-down-cast)
apply(simp add: def-split-local, elim disjE)
apply(erule StrongEq-L-subst2-rev, simp, simp)+
done

```

```

lemma up-down-cast-Person-OclAny-Person': assumes  $\tau \models v \ X$ 
shows  $\tau \models (((X :: \text{Person}) .\text{oclAsType}(\text{OclAny}) .\text{oclAsType}(\text{Person})) \doteq X)$ 
  apply(simp only: up-down-cast-Person-OclAny-Person StrictRefEqObject-Person)
by(rule StrictRefEqObject-sym, simp add: assms)

lemma up-down-cast-Person-OclAny-Person'': assumes  $\tau \models v \ (X :: \text{Person})$ 
shows  $\tau \models (X .\text{oclIsTypeOf}(\text{Person}) \text{ implies } (X .\text{oclAsType}(\text{OclAny}) .\text{oclAsType}(\text{Person})) \doteq X)$ 
  apply(simp add: OclValid-def)
  apply(subst cp-OclImplies)
  apply(simp add: StrictRefEqObject-Person StrictRefEqObject-sym[OF assms, simplified OclValid-def])
  apply(subst cp-OclImplies[symmetric])
by (simp add: OclImplies-true)

```

6.5. OclIsKindOf

6.5.1. Definition

```

consts OclIsKindOfOclAny :: ' $\alpha \Rightarrow \text{Boolean}$  ((-).oclIsKindOf'(OclAny'))
consts OclIsKindOfPerson :: ' $\alpha \Rightarrow \text{Boolean}$  ((-).oclIsKindOf'(Person'))

```

```

defs (overloaded) OclIsKindOfOclAny-OclAny:
  (X::OclAny) .oclIsKindOf(OclAny)  $\equiv$ 
    ( $\lambda\tau$ . case X  $\tau$  of
       $\perp \Rightarrow \text{invalid } \tau$ 
      |  $- \Rightarrow \text{true } \tau$ )

```

```

defs (overloaded) OclIsKindOfOclAny-Person:
  (X::Person) .oclIsKindOf(OclAny)  $\equiv$ 
    ( $\lambda\tau$ . case X  $\tau$  of
       $\perp \Rightarrow \text{invalid } \tau$ 
      |  $- \Rightarrow \text{true } \tau$ )

```

```

defs (overloaded) OclIsKindOfPerson-OclAny:
  (X::OclAny) .oclIsKindOf(Person)  $\equiv$ 
    ( $\lambda\tau$ . case X  $\tau$  of
       $\perp \Rightarrow \text{invalid } \tau$ 
      |  $\lfloor \perp \rfloor \Rightarrow \text{true } \tau$ 
      |  $\lfloor \text{mk}_{\text{OclAny}} \text{ oid } \perp \rfloor \rfloor \Rightarrow \text{false } \tau$ 
      |  $\lfloor \text{mk}_{\text{OclAny}} \text{ oid } [-] \rfloor \rfloor \Rightarrow \text{true } \tau$ )

```

```

defs (overloaded) OclIsKindOfPerson-Person:
  (X::Person) .oclIsKindOf(Person)  $\equiv$ 
    ( $\lambda\tau$ . case X  $\tau$  of
       $\perp \Rightarrow \text{invalid } \tau$ 
      |  $- \Rightarrow \text{true } \tau$ )

```

6.5.2. Context Passing

```

lemma cp-OclIsKindOfOclAny-Person-Person: cp P  $\implies$  cp( $\lambda X. (P (X::Person) :: Person)$ 
  .oclIsKindOf(OclAny))
by(rule cpI1, simp-all add: OclIsKindOfOclAny-Person)
lemma cp-OclIsKindOfOclAny-OclAny-OclAny: cp P  $\implies$  cp( $\lambda X. (P (X::OclAny) :: OclAny)$ 
  .oclIsKindOf(OclAny))
by(rule cpI1, simp-all add: OclIsKindOfOclAny-OclAny)
lemma cp-OclIsKindOfPerson-Person-Person: cp P  $\implies$  cp( $\lambda X. (P (X::Person) :: Person)$ 
  .oclIsKindOf(Person))
by(rule cpI1, simp-all add: OclIsKindOfPerson-Person)
lemma cp-OclIsKindOfPerson-OclAny-OclAny: cp P  $\implies$  cp( $\lambda X. (P (X::OclAny) :: OclAny)$ 
  .oclIsKindOf(Person))
by(rule cpI1, simp-all add: OclIsKindOfPerson-OclAny)

lemma cp-OclIsKindOfOclAny-Person-OclAny: cp P  $\implies$  cp( $\lambda X. (P (X::Person) :: OclAny)$ 
  .oclIsKindOf(OclAny))
by(rule cpI1, simp-all add: OclIsKindOfOclAny-OclAny)
lemma cp-OclIsKindOfOclAny-OclAny-Person: cp P  $\implies$  cp( $\lambda X. (P (X::OclAny) :: Person)$ 
  .oclIsKindOf(OclAny))
by(rule cpI1, simp-all add: OclIsKindOfOclAny-Person)
lemma cp-OclIsKindOfPerson-Person-OclAny: cp P  $\implies$  cp( $\lambda X. (P (X::Person) :: OclAny)$ 
  .oclIsKindOf(Person))
by(rule cpI1, simp-all add: OclIsKindOfPerson-OclAny)
lemma cp-OclIsKindOfPerson-OclAny-Person: cp P  $\implies$  cp( $\lambda X. (P (X::OclAny) :: Person)$ 
  .oclIsKindOf(Person))
by(rule cpI1, simp-all add: OclIsKindOfPerson-Person)

lemmas [simp] =
  cp-OclIsKindOfOclAny-Person-Person
  cp-OclIsKindOfOclAny-OclAny-OclAny
  cp-OclIsKindOfPerson-Person-Person
  cp-OclIsKindOfPerson-OclAny-OclAny

  cp-OclIsKindOfOclAny-Person-OclAny
  cp-OclIsKindOfOclAny-OclAny-Person
  cp-OclIsKindOfPerson-Person-OclAny
  cp-OclIsKindOfPerson-OclAny-Person

```

6.5.3. Execution with invalid or null as argument

```

lemma OclIsKindOfOclAny-OclAny-strict1[simp] : (invalid::OclAny) .ocIsKindOf(OclAny) =
  invalid
by(rule ext, simp add: invalid-def bot-option-def
  OclIsKindOfOclAny-OclAny)

lemma OclIsKindOfOclAny-OclAny-strict2[simp] : (null::OclAny) .ocIsKindOf(OclAny) = true
by(rule ext, simp add: null-fun-def null-option-def
  OclIsKindOfOclAny-OclAny)

```

lemma *OclIsKindOf_{OclAny}-Person-strict1*[simp] : (invalid::Person) .oclIsKindOf(OclAny) = invalid

by(rule ext, simp add: bot-option-def invalid-def
OclIsKindOf_{OclAny}-Person)

lemma *OclIsKindOf_{OclAny}-Person-strict2*[simp] : (null::Person) .oclIsKindOf(OclAny) = true

by(rule ext, simp add: null-fun-def null-option-def bot-option-def
OclIsKindOf_{OclAny}-Person)

lemma *OclIsKindOf_{Person}-OclAny-strict1*[simp] : (invalid::OclAny) .oclIsKindOf(Person) = invalid

by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
OclIsKindOf_{Person}-OclAny)

lemma *OclIsKindOf_{Person}-OclAny-strict2*[simp] : (null::OclAny) .oclIsKindOf(Person) = true

by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
OclIsKindOf_{Person}-OclAny)

lemma *OclIsKindOf_{Person}-Person-strict1*[simp] : (invalid::Person) .oclIsKindOf(Person) = invalid

by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
OclIsKindOf_{Person}-Person)

lemma *OclIsKindOf_{Person}-Person-strict2*[simp] : (null::Person) .oclIsKindOf(Person) = true

by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
OclIsKindOf_{Person}-Person)

6.5.4. up down cast

lemma *actualKind-larger-staticKind*:

assumes isdef: $\tau \models (\delta \ X)$

shows $\tau \models (X::Person) .oclIsKindOf(OclAny) \triangleq true$

using isdef

by(auto simp : bot-option-def
OclIsKindOf_{OclAny}-Person foundation22 foundation16)

lemma *down-cast-kind*:

assumes isOclAny: $\neg \tau \models (X::OclAny) .oclIsKindOf(Person)$

and non-null: $\tau \models (\delta \ X)$

shows $\tau \models (X .oclAsType(Person)) \triangleq invalid$

using isOclAny non-null

apply(auto simp : bot-fun-def null-fun-def null-option-def bot-option-def null-def invalid-def
OclAsType_{OclAny}-Person OclAsType_{Person}-OclAny foundation22 foundation16
split: option.split type_{OclAny}.split type_{Person}.split)

by(simp add: OclIsKindOf_{Person}-OclAny OclValid-def false-def true-def)

6.6. OclAllInstances

Recall that in order to denote OCL-types occurring 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 $Person \equiv OclAsType_{Person}\mathcal{A}$

definition $OclAny \equiv OclAsType_{OclAny}\mathcal{A}$

lemmas $[simp] = Person\text{-}def\ OclAny\text{-}def$

lemma $OclAllInstances_{OclAny}\text{-}exec: OclAny.allInstances() =$
 $(\lambda\tau. Abs\text{-}Set\text{-}0 \ll Some\ 'OclAny\ 'ran\ (heap\ (snd\ \tau)) \gg)$
apply(rule ext, simp add: OclAllInstances-at-post-def)
apply(subgoal-tac (OclAny 'ran (heap (snd τ)) - {None}) = (OclAny 'ran (heap (snd τ))),
simp)
apply(simp add: image-def)
apply(rule equalityI)
apply(rule subsetI)
apply (metis (lifting) Diff-iff)
apply(rule subsetI)
apply(simp)
apply(drule bexE) **prefer** 2 **apply** assumption
apply(case-tac x, metis OclAsType_{OclAny}\mathcal{A}\text{-}some)
apply(blast)
done

lemma $OclAny.allInstances@pre() =$
 $(\lambda\tau. Abs\text{-}Set\text{-}0 \ll Some\ 'OclAny\ 'ran\ (heap\ (fst\ \tau)) \gg)$
apply(rule ext, simp add: OclAllInstances-at-pre-def)
apply(subgoal-tac (OclAny 'ran (heap (fst τ)) - {None}) = (OclAny 'ran (heap (fst τ))),
simp)
apply(simp add: image-def)
apply(rule equalityI)
apply(rule subsetI)
apply (metis (lifting) Diff-iff)
apply(rule subsetI)
apply(simp)
apply(drule bexE) **prefer** 2 **apply** assumption
apply(case-tac x, metis OclAsType_{OclAny}\mathcal{A}\text{-}some)
apply(blast)
done

6.6.1. IsTypeOf

lemma $OclAny.allInstances\text{-}IsTypeOf_{OclAny}1:$

$\exists\tau. (\tau \models (OclAny.allInstances()) \text{-}> \text{forAll}(X|X.oclIsTypeOf(OclAny))))$
apply(rule-tac $x = \tau_0$ in exI, simp add: $\tau_0\text{-}def\ OclValid\text{-}def$)
apply(simp only: OclForall-def OclAllInstances-defined[simplified OclValid-def] refl if-True)
apply(simp only: OclAllInstances-at-post-def OclAllInstances-def)
apply(subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)

by(simp add: OclIsTypeOf_{OclAny}-OclAny)

lemma OclAny-allInstances-IsTypeOf_{OclAny}2:

$\exists \tau. (\tau \models \text{not } (\text{OclAny} . \text{allInstances}() \rightarrow \text{forAll}(X | X . \text{oclIsTypeOf}(\text{OclAny}))))$

proof – **fix** oid a **show** ?thesis

apply(rule-tac x = (fst τ_0 , ($\text{heap} = \text{empty}(oid \mapsto \text{in}_{\text{OclAny}}(\text{mk}_{\text{OclAny}} \text{oid } [a])),$
 $\text{assocs}_2 = \text{empty}, \text{assocs}_3 = \text{empty}))$ in exI, simp add: OclValid-def)

apply(simp only: OclForall-def OclAllInstances-defined[simplified OclValid-def] refl if-True)

apply(simp only: OclAllInstances-at-post-def OclAllInstances-def OclAsType_{OclAny}- \mathcal{A} -def)

apply(subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)

by(simp add: OclIsTypeOf_{OclAny}-OclAny OclNot-def OclAny-def)

qed

lemma Person-allInstances-IsTypeOf_{Person}:

$\tau \models (\text{Person} . \text{allInstances}() \rightarrow \text{forAll}(X | X . \text{oclIsTypeOf}(\text{Person})))$

apply(simp add: OclValid-def)

apply(simp only: OclForall-def OclAllInstances-defined[simplified OclValid-def] refl if-True)

apply(simp only: OclAllInstances-at-post-def OclAllInstances-def)

apply(subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)

by(simp add: OclIsTypeOf_{Person}-Person)

6.6.2. IsKindOf

lemma OclAny-allInstances-IsKindOf_{OclAny}:

$\tau \models (\text{OclAny} . \text{allInstances}() \rightarrow \text{forAll}(X | X . \text{oclIsKindOf}(\text{OclAny})))$

apply(simp add: OclValid-def)

apply(simp only: OclForall-def OclAllInstances-defined[simplified OclValid-def] refl if-True)

apply(simp only: OclAllInstances-at-post-def OclAllInstances-def)

apply(subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)

by(simp add: OclIsKindOf_{OclAny}-OclAny)

lemma Person-allInstances-IsKindOf_{OclAny}:

$\tau \models (\text{Person} . \text{allInstances}() \rightarrow \text{forAll}(X | X . \text{oclIsKindOf}(\text{OclAny})))$

apply(simp add: OclValid-def)

apply(simp only: OclForall-def OclAllInstances-defined[simplified OclValid-def] refl if-True)

apply(simp only: OclAllInstances-at-post-def OclAllInstances-def)

apply(subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)

by(simp add: OclIsKindOf_{OclAny}-Person)

lemma Person-allInstances-IsKindOf_{Person}:

$\tau \models (\text{Person} . \text{allInstances}() \rightarrow \text{forAll}(X | X . \text{oclIsKindOf}(\text{Person})))$

apply(simp add: OclValid-def)

apply(simp only: OclForall-def OclAllInstances-defined[simplified OclValid-def] refl if-True)

apply(simp only: OclAllInstances-at-post-def OclAllInstances-def)

apply(subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)

by(simp add: OclIsKindOf_{Person}-Person)

6.7. dot boss, dot salary

Should be generated entirely from a class-diagram.

6.7.1. Definition (of the association Employee-Boss)

We start with a oid for the association; this oid can be used in presence of association classes to represent the association inside an object, pretty much similar to the `Employee_DesignModel_UMLPart`, where we stored an oid inside the class as "pointer".

definition $oid_{PersonBOSS} :: oid$ **where** $oid_{PersonBOSS} = 10$

From there on, we can already define an empty state which must contain for $oid \setminus \langle \text{sub} \rangle P \setminus \langle \text{sub} \rangle e \setminus \langle \text{sub} \rangle \tau$ the empty relation (encoded as association list, since there are associations with a Sequence-like structure).

definition $eval_extract :: ('A, ('a::object) option option) val$
 $\Rightarrow (oid \Rightarrow ('A, 'c::null) val)$
 $\Rightarrow ('A, 'c::null) val$
where $eval_extract\ X\ f = (\lambda\ \tau. \text{case } X\ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau \quad (* \text{ exception propagation } *)$
 $\quad | \lfloor \perp \rfloor \Rightarrow \text{invalid } \tau \quad (* \text{ dereferencing null pointer } *)$
 $\quad | \lfloor \lfloor obj \rfloor \rfloor \Rightarrow f\ (oid\text{-of } obj)\ \tau)$

definition $choose_2-1 = fst$

definition $choose_2-2 = snd$

definition $choose_3-1 = fst$

definition $choose_3-2 = fst \circ snd$

definition $choose_3-3 = snd \circ snd$

definition $deref_assocs_2 :: ('A\ state \times 'A\ state \Rightarrow 'A\ state)$
 $\Rightarrow (oid \times oid \Rightarrow oid \times oid)$
 $\Rightarrow oid$
 $\Rightarrow (oid\ list \Rightarrow oid \Rightarrow ('A, 'f) val)$
 $\Rightarrow oid$
 $\Rightarrow ('A, 'f::null) val$
where $deref_assocs_2\ pre\text{-}post\ to\text{-}from\ assoc\text{-}oid\ f\ oid =$
 $(\lambda\tau. \text{case } (assocs_2\ (pre\text{-}post\ \tau))\ assoc\text{-}oid\ \text{of}$
 $\quad \lfloor S \rfloor \Rightarrow f\ (map\ (choose_2-2 \circ to\text{-}from)$
 $\quad \quad (filter\ (\lambda\ p. choose_2-1\ (to\text{-}from\ p) = oid)\ S))$
 $\quad \quad oid\ \tau$
 $\quad | - \Rightarrow \text{invalid } \tau)$

The *pre-post*-parameter is configured with *fst* or *snd*, the *to-from*-parameter either with the identity *id* or the following combinator *switch*:

definition $switch_2-1 = id$

definition $switch_2-2 = (\lambda(x,y). (y,x))$

definition $switch_3-1 = id$

definition $switch_{3-2} = (\lambda(x,y,z). (x,z,y))$

definition $switch_{3-3} = (\lambda(x,y,z). (y,x,z))$

definition $switch_{3-4} = (\lambda(x,y,z). (y,z,x))$

definition $switch_{3-5} = (\lambda(x,y,z). (z,x,y))$

definition $switch_{3-6} = (\lambda(x,y,z). (z,y,x))$

definition $select-object :: (('A, 'b::null)val)$
 $\Rightarrow (('A, 'b)val \Rightarrow ('A, 'c)val \Rightarrow ('A, 'b)val)$
 $\Rightarrow (('A, 'b)val \Rightarrow ('A, 'd)val)$
 $\Rightarrow (oid \Rightarrow ('A, 'c::null)val)$
 $\Rightarrow oid\ list$
 $\Rightarrow oid$
 $\Rightarrow ('A, 'd)val$

where $select-object\ mt\ incl\ smash\ deref\ l\ oid = smash(foldl\ incl\ mt\ (map\ deref\ l))$
 $(*\ smash\ returns\ null\ with\ mt\ in\ input\ (in\ this\ case,\ object\ contains\ null\ pointer)\ *)$

The continuation f is usually instantiated with a smashing function which is either the identity id or, for 0..1 cardinalities of associations, the $OclAny$ - selector which also handles the $null$ -cases appropriately. A standard use-case for this combinator is for example:

term $(select-object\ mtSet\ OclIncluding\ OclAny\ f\ l\ oid) :: ('A, 'a::null)val$

definition $deref-oid_{Person} :: (A\ state \times A\ state \Rightarrow A\ state)$
 $\Rightarrow (type_{Person} \Rightarrow (A, 'c::null)val)$
 $\Rightarrow oid$
 $\Rightarrow (A, 'c::null)val$

where $deref-oid_{Person}\ fst-snd\ f\ oid = (\lambda\tau. case\ (heap\ (fst-snd\ \tau))\ oid\ of$
 $\mid in_{Person}\ obj \mid \Rightarrow f\ obj\ \tau$
 $\mid - \mid \Rightarrow invalid\ \tau)$

definition $deref-oid_{OclAny}\ fst-snd\ f\ oid = (\lambda\tau. case\ (heap\ (fst-snd\ \tau))\ oid\ of$
 $\mid in_{OclAny}\ obj \mid \Rightarrow f\ obj\ \tau$
 $\mid - \mid \Rightarrow invalid\ \tau)$

pointer undefined in state or not referencing a type conform object representation

definition $select_{OclAny} \mathcal{ANY}\ f = (\lambda\ X. case\ X\ of$
 $(mk_{OclAny} - \perp) \Rightarrow null$
 $\mid (mk_{OclAny} - \lfloor any \rfloor) \Rightarrow f\ (\lambda x -. \lfloor \lfloor x \rfloor \rfloor)\ any)$

definition $select_{Person} \mathcal{BOSS}\ f = select-object\ mtSet\ OclIncluding\ OclAny\ (f\ (\lambda x -. \lfloor \lfloor x \rfloor \rfloor))$

definition $select_{Person} \mathcal{SALARY}\ f = (\lambda\ X. case\ X\ of$
 $(mk_{Person} - \perp) \Rightarrow null$
 $\mid (mk_{Person} - \lfloor salary \rfloor) \Rightarrow f\ (\lambda x -. \lfloor \lfloor x \rfloor \rfloor)\ salary)$

definition $deref-assocs_2\mathcal{BOSS} \text{ fst-snd } f = (\lambda mk_{Person} oid - \Rightarrow deref-assocs_2 \text{ fst-snd } switch_2-1 \text{ oid}_{Person} \mathcal{BOSS} f oid)$

definition $in\text{-}pre\text{-}state = fst$

definition $in\text{-}post\text{-}state = snd$

definition $reconst\text{-}basetype = (\lambda convert x. convert x)$

definition $dot_{OclAny}\mathcal{ANY} :: OclAny \Rightarrow - \ ((1(-).any) \ 50)$

where $(X).any = eval\text{-}extract \ X$
 $(deref\text{-}oid_{OclAny} \ in\text{-}post\text{-}state$
 $(select_{OclAny}\mathcal{ANY}$
 $reconst\text{-}basetype))$

definition $dot_{Person}\mathcal{BOSS} :: Person \Rightarrow Person \ ((1(-).boss) \ 50)$

where $(X).boss = eval\text{-}extract \ X$
 $(deref\text{-}oid_{Person} \ in\text{-}post\text{-}state$
 $(deref\text{-}assocs_2\mathcal{BOSS} \ in\text{-}post\text{-}state$
 $(select_{Person}\mathcal{BOSS}$
 $(deref\text{-}oid_{Person} \ in\text{-}post\text{-}state))))$

definition $dot_{Person}\mathcal{SALARY} :: Person \Rightarrow Integer \ ((1(-).salary) \ 50)$

where $(X).salary = eval\text{-}extract \ X$
 $(deref\text{-}oid_{Person} \ in\text{-}post\text{-}state$
 $(select_{Person}\mathcal{SALARY}$
 $reconst\text{-}basetype))$

definition $dot_{OclAny}\mathcal{ANY}\text{-}at\text{-}pre :: OclAny \Rightarrow - \ ((1(-).any@pre) \ 50)$

where $(X).any@pre = eval\text{-}extract \ X$
 $(deref\text{-}oid_{OclAny} \ in\text{-}pre\text{-}state$
 $(select_{OclAny}\mathcal{ANY}$
 $reconst\text{-}basetype))$

definition $dot_{Person}\mathcal{BOSS}\text{-}at\text{-}pre :: Person \Rightarrow Person \ ((1(-).boss@pre) \ 50)$

where $(X).boss@pre = eval\text{-}extract \ X$
 $(deref\text{-}oid_{Person} \ in\text{-}pre\text{-}state$
 $(deref\text{-}assocs_2\mathcal{BOSS} \ in\text{-}pre\text{-}state$
 $(select_{Person}\mathcal{BOSS}$
 $(deref\text{-}oid_{Person} \ in\text{-}pre\text{-}state))))$

definition $dot_{Person}\mathcal{SALARY}\text{-}at\text{-}pre :: Person \Rightarrow Integer \ ((1(-).salary@pre) \ 50)$

where $(X).salary@pre = eval\text{-}extract \ X$
 $(deref\text{-}oid_{Person} \ in\text{-}pre\text{-}state$
 $(select_{Person}\mathcal{SALARY}$
 $reconst\text{-}basetype))$

lemmas $[simp] =$
 $dot_{OclAny}\mathcal{ANY}\text{-}def$

$\text{dot}_{Person}BOSS\text{-def}$
 $\text{dot}_{Person}SALARY\text{-def}$
 $\text{dot}_{OclAny}ANY\text{-at-pre-def}$
 $\text{dot}_{Person}BOSS\text{-at-pre-def}$
 $\text{dot}_{Person}SALARY\text{-at-pre-def}$

6.7.2. Context Passing

lemma $\text{cp-dot}_{OclAny}ANY$: $((X).any) \tau = ((\lambda-. X \tau).any) \tau$ **by**(simp add: eval-extract-def)
lemma $\text{cp-dot}_{Person}BOSS$: $((X).boss) \tau = ((\lambda-. X \tau).boss) \tau$ **by**(simp add: eval-extract-def)
lemma $\text{cp-dot}_{Person}SALARY$: $((X).salary) \tau = ((\lambda-. X \tau).salary) \tau$ **by**(simp add: eval-extract-def)

lemma $\text{cp-dot}_{OclAny}ANY\text{-at-pre}$: $((X).any@pre) \tau = ((\lambda-. X \tau).any@pre) \tau$ **by**(simp add: eval-extract-def)
lemma $\text{cp-dot}_{Person}BOSS\text{-at-pre}$: $((X).boss@pre) \tau = ((\lambda-. X \tau).boss@pre) \tau$ **by**(simp add: eval-extract-def)
lemma $\text{cp-dot}_{Person}SALARY\text{-at-pre}$: $((X).salary@pre) \tau = ((\lambda-. X \tau).salary@pre) \tau$ **by**(simp add: eval-extract-def)

lemmas $\text{cp-dot}_{OclAny}ANY\text{-I}$ [simp, intro!]=
 $\text{cp-dot}_{OclAny}ANY[THEN\ allI[THEN\ allI],$
 $\text{of } \lambda X -. X \lambda - \tau. \tau, THEN\ cpI1]$
lemmas $\text{cp-dot}_{OclAny}ANY\text{-at-pre-I}$ [simp, intro!]=
 $\text{cp-dot}_{OclAny}ANY\text{-at-pre}[THEN\ allI[THEN\ allI],$
 $\text{of } \lambda X -. X \lambda - \tau. \tau, THEN\ cpI1]$

lemmas $\text{cp-dot}_{Person}BOSS\text{-I}$ [simp, intro!]=
 $\text{cp-dot}_{Person}BOSS[THEN\ allI[THEN\ allI],$
 $\text{of } \lambda X -. X \lambda - \tau. \tau, THEN\ cpI1]$
lemmas $\text{cp-dot}_{Person}BOSS\text{-at-pre-I}$ [simp, intro!]=
 $\text{cp-dot}_{Person}BOSS\text{-at-pre}[THEN\ allI[THEN\ allI],$
 $\text{of } \lambda X -. X \lambda - \tau. \tau, THEN\ cpI1]$

lemmas $\text{cp-dot}_{Person}SALARY\text{-I}$ [simp, intro!]=
 $\text{cp-dot}_{Person}SALARY[THEN\ allI[THEN\ allI],$
 $\text{of } \lambda X -. X \lambda - \tau. \tau, THEN\ cpI1]$
lemmas $\text{cp-dot}_{Person}SALARY\text{-at-pre-I}$ [simp, intro!]=
 $\text{cp-dot}_{Person}SALARY\text{-at-pre}[THEN\ allI[THEN\ allI],$
 $\text{of } \lambda X -. X \lambda - \tau. \tau, THEN\ cpI1]$

6.7.3. Execution with invalid or null as argument

lemma $\text{dot}_{OclAny}ANY\text{-nullstrict}$ [simp]: $(null).any = invalid$
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def eval-extract-def)
lemma $\text{dot}_{OclAny}ANY\text{-at-pre-nullstrict}$ [simp]: $(null).any@pre = invalid$
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def eval-extract-def)
lemma $\text{dot}_{OclAny}ANY\text{-strict}$ [simp]: $(invalid).any = invalid$
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def eval-extract-def)
lemma $\text{dot}_{OclAny}ANY\text{-at-pre-strict}$ [simp]: $(invalid).any@pre = invalid$
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def eval-extract-def)

lemma *dot_{Person}BOSS-nullstrict* [simp]: (null).boss = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def eval-extract-def)
lemma *dot_{Person}BOSS-at-pre-nullstrict* [simp] : (null).boss@pre = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def eval-extract-def)
lemma *dot_{Person}BOSS-strict* [simp] : (invalid).boss = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def eval-extract-def)
lemma *dot_{Person}BOSS-at-pre-strict* [simp] : (invalid).boss@pre = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def eval-extract-def)

lemma *dot_{Person}SALARY-nullstrict* [simp]: (null).salary = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def eval-extract-def)
lemma *dot_{Person}SALARY-at-pre-nullstrict* [simp] : (null).salary@pre = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def eval-extract-def)
lemma *dot_{Person}SALARY-strict* [simp] : (invalid).salary = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def eval-extract-def)
lemma *dot_{Person}SALARY-at-pre-strict* [simp] : (invalid).salary@pre = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def eval-extract-def)

6.8. A little infra-structure on example states.

definition *OclInt1000* (1000) **where** *OclInt1000* = (λ . . [[1000]])
definition *OclInt1200* (1200) **where** *OclInt1200* = (λ . . [[1200]])
definition *OclInt1300* (1300) **where** *OclInt1300* = (λ . . [[1300]])
definition *OclInt1800* (1800) **where** *OclInt1800* = (λ . . [[1800]])
definition *OclInt2600* (2600) **where** *OclInt2600* = (λ . . [[2600]])
definition *OclInt2900* (2900) **where** *OclInt2900* = (λ . . [[2900]])
definition *OclInt3200* (3200) **where** *OclInt3200* = (λ . . [[3200]])
definition *OclInt3500* (3500) **where** *OclInt3500* = (λ . . [[3500]])

definition *oid0* \equiv 0
definition *oid1* \equiv 1
definition *oid2* \equiv 2
definition *oid3* \equiv 3
definition *oid4* \equiv 4
definition *oid5* \equiv 5
definition *oid6* \equiv 6
definition *oid7* \equiv 7
definition *oid8* \equiv 8

definition *person1* \equiv mk_{Person} *oid0* [1300]
definition *person2* \equiv mk_{Person} *oid1* [1800]
definition *person3* \equiv mk_{Person} *oid2* None
definition *person4* \equiv mk_{Person} *oid3* [2900]
definition *person5* \equiv mk_{Person} *oid4* [3500]
definition *person6* \equiv mk_{Person} *oid5* [2500]
definition *person7* \equiv mk_{OclAny} *oid6* [[3200]]

definition $person8 \equiv mk_{OclAny} \text{ oid7 } None$

definition $person9 \equiv mk_{Person} \text{ oid8 } [0]$

definition

$\sigma_1 \equiv () \text{ heap} = \text{empty}(\text{oid0} \mapsto \text{in}_{Person} (mk_{Person} \text{ oid0 } [1000]))$
 $(\text{oid1} \mapsto \text{in}_{Person} (mk_{Person} \text{ oid1 } [1200]))$
 $(*\text{oid2}*)$
 $(\text{oid3} \mapsto \text{in}_{Person} (mk_{Person} \text{ oid3 } [2600]))$
 $(\text{oid4} \mapsto \text{in}_{Person} person5)$
 $(\text{oid5} \mapsto \text{in}_{Person} (mk_{Person} \text{ oid5 } [2300]))$
 $(*\text{oid6}*)$
 $(*\text{oid7}*)$
 $(\text{oid8} \mapsto \text{in}_{Person} person9),$
 $\text{assocs}_2 = \text{empty}(\text{oid}_{Person} BOSS \mapsto [(\text{oid0}, \text{oid1}), (\text{oid3}, \text{oid4}), (\text{oid5}, \text{oid3})]),$
 $\text{assocs}_3 = \text{empty } ()$

definition

$\sigma_1' \equiv () \text{ heap} = \text{empty}(\text{oid0} \mapsto \text{in}_{Person} person1)$
 $(\text{oid1} \mapsto \text{in}_{Person} person2)$
 $(\text{oid2} \mapsto \text{in}_{Person} person3)$
 $(\text{oid3} \mapsto \text{in}_{Person} person4)$
 $(*\text{oid4}*)$
 $(\text{oid5} \mapsto \text{in}_{Person} person6)$
 $(\text{oid6} \mapsto \text{in}_{OclAny} person7)$
 $(\text{oid7} \mapsto \text{in}_{OclAny} person8)$
 $(\text{oid8} \mapsto \text{in}_{Person} person9),$
 $\text{assocs}_2 = \text{empty}(\text{oid}_{Person} BOSS \mapsto [(\text{oid0}, \text{oid1}), (\text{oid1}, \text{oid1}), (\text{oid5}, \text{oid6}), (\text{oid6}, \text{oid6})]),$
 $\text{assocs}_3 = \text{empty } ()$

definition $\sigma_0 \equiv () \text{ heap} = \text{empty}, \text{assocs}_2 = \text{empty}, \text{assocs}_3 = \text{empty } ()$

lemma $\text{basic-}\tau\text{-wff: } WFF(\sigma_1, \sigma_1')$

by($\text{auto simp: } WFF\text{-def } \sigma_1\text{-def } \sigma_1'\text{-def}$

$\text{oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def}$

$\text{oid-of-}\mathcal{A}\text{-def oid-of-type}_{Person}\text{-def oid-of-type}_{OclAny}\text{-def}$

$\text{person1-def person2-def person3-def person4-def person5-def person6-def person7-def}$

$\text{person8-def person9-def})$

lemma $[\text{simp}, \text{code-unfold}]: \text{dom } (\text{heap } \sigma_1) = \{\text{oid0}, \text{oid1}, (*, \text{oid2}*)\text{oid3}, \text{oid4}, \text{oid5}(*, \text{oid6}, \text{oid7}*), \text{oid8}\}$

by($\text{auto simp: } \sigma_1\text{-def}$)

lemma $[\text{simp}, \text{code-unfold}]: \text{dom } (\text{heap } \sigma_1') = \{\text{oid0}, \text{oid1}, \text{oid2}, \text{oid3}, (*, \text{oid4}*)\text{oid5}, \text{oid6}, \text{oid7}, \text{oid8}\}$

by($\text{auto simp: } \sigma_1'\text{-def}$)

definition $X_{Person1} :: Person \equiv \lambda - . [\text{person1}]$

definition $X_{Person2} :: Person \equiv \lambda - . [\text{person2}]$

definition $X_{Person3} :: Person \equiv \lambda - . [\text{person3}]$

definition $X_{Person4} :: Person \equiv \lambda - . [\text{person4}]$

definition $X_{Person5} :: Person \equiv \lambda - . [[\text{person5}]]$
definition $X_{Person6} :: Person \equiv \lambda - . [[\text{person6}]]$
definition $X_{Person7} :: OclAny \equiv \lambda - . [[\text{person7}]]$
definition $X_{Person8} :: OclAny \equiv \lambda - . [[\text{person8}]]$
definition $X_{Person9} :: Person \equiv \lambda - . [[\text{person9}]]$

lemma $[code-unfold]: ((x::Person) \doteq y) = StrictRefEq_{Object} \ x \ y \text{ by } (simp \ only: StrictRefEq_{Object-}Person)$
lemma $[code-unfold]: ((x::OclAny) \doteq y) = StrictRefEq_{Object} \ x \ y \text{ by } (simp \ only: StrictRefEq_{Object-OclAny})$

lemmas $[code-unfold, simp] =$

$OclAsType_{OclAny-OclAny}$
 $OclAsType_{OclAny-Person}$
 $OclAsType_{Person-OclAny}$
 $OclAsType_{Person-Person}$

$OclIsTypeOf_{OclAny-OclAny}$
 $OclIsTypeOf_{OclAny-Person}$
 $OclIsTypeOf_{Person-OclAny}$
 $OclIsTypeOf_{Person-Person}$

$OclIsKindOf_{OclAny-OclAny}$
 $OclIsKindOf_{OclAny-Person}$
 $OclIsKindOf_{Person-OclAny}$
 $OclIsKindOf_{Person-Person}$

value $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models (X_{Person1} . salary \ <> \mathbf{1000})$
value $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models (X_{Person1} . salary \ \doteq \mathbf{1300})$
value $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models (X_{Person1} . salary@pre \ \doteq \mathbf{1000})$
value $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models (X_{Person1} . salary@pre \ <> \mathbf{1300})$

lemma $(\sigma_1, \sigma_1') \models (X_{Person1} . oclIsMaintained())$
by $(simp \ add: OclValid-def \ OclIsMaintained-def$
 $\sigma_1-def \ \sigma_1'-def$
 $X_{Person1}-def \ person1-def$
 $oid0-def \ oid1-def \ oid2-def \ oid3-def \ oid4-def \ oid5-def \ oid6-def$
 $oid-of-option-def \ oid-of-type_{Person}-def)$

lemma $\bigwedge_{s_{pre} \ s_{post}} . (s_{pre}, s_{post}) \models ((X_{Person1} . oclAsType(OclAny) . oclAsType(Person)) \doteq X_{Person1})$
by $(rule \ up-down-cast-Person-OclAny-Person', \ simp \ add: X_{Person1}-def)$
value $\bigwedge_{s_{pre} \ s_{post}} . (s_{pre}, s_{post}) \models (X_{Person1} . oclIsTypeOf(Person))$
value $\bigwedge_{s_{pre} \ s_{post}} . (s_{pre}, s_{post}) \models not(X_{Person1} . oclIsTypeOf(OclAny))$
value $\bigwedge_{s_{pre} \ s_{post}} . (s_{pre}, s_{post}) \models (X_{Person1} . oclIsKindOf(Person))$
value $\bigwedge_{s_{pre} \ s_{post}} . (s_{pre}, s_{post}) \models (X_{Person1} . oclIsKindOf(OclAny))$
value $\bigwedge_{s_{pre} \ s_{post}} . (s_{pre}, s_{post}) \models not(X_{Person1} . oclAsType(OclAny) . oclIsTypeOf(OclAny))$

value $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person2} .salary \doteq 1800)$
value $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person2} .salary@pre \doteq 1200)$
lemma $(\sigma_1, \sigma_1') \models (X_{Person2} .oclIsMaintained())$
by(simp add: OclValid-def OclIsMaintained-def
 σ_1 -def σ_1' -def
 $X_{Person2}$ -def person2-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def
oid-of-option-def oid-of-type_{Person}-def)

value $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person3} .salary \doteq null)$
value $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models not(v(X_{Person3} .salary@pre))$
value $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person3} .salary@pre <> null)$
lemma $(\sigma_1, \sigma_1') \models (X_{Person3} .oclIsNew())$
by(simp add: OclValid-def OclIsNew-def
 σ_1 -def σ_1' -def
 $X_{Person3}$ -def person3-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid8-def
oid-of-option-def oid-of-type_{Person}-def)

lemma $(\sigma_1, \sigma_1') \models (X_{Person4} .oclIsMaintained())$
by(simp add: OclValid-def OclIsMaintained-def
 σ_1 -def σ_1' -def
 $X_{Person4}$ -def person4-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def
oid-of-option-def oid-of-type_{Person}-def)

value $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models not(v(X_{Person5} .salary))$
value $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person5} .salary@pre \doteq 3500)$
lemma $(\sigma_1, \sigma_1') \models (X_{Person5} .oclIsOld())$
by(simp add: OclNot-def OclValid-def OclIsOld-def
 σ_1 -def σ_1' -def
 $X_{Person5}$ -def person5-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def
oid-of-option-def oid-of-type_{Person}-def)

lemma $(\sigma_1, \sigma_1') \models (X_{Person6} .oclIsMaintained())$
by(simp add: OclValid-def OclIsMaintained-def
 σ_1 -def σ_1' -def
 $X_{Person6}$ -def person6-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def
oid-of-option-def oid-of-type_{Person}-def)

value $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models v(X_{Person7} .oclAsType(Person))$
lemma $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models ((X_{Person7} .oclAsType(Person) .oclAsType(OclAny) .oclAsType(Person)) \doteq (X_{Person7} .oclAsType(Person)))$
by(rule up-down-cast-Person-OclAny-Person', simp add: X_{Person7}-def OclValid-def valid-def person7-def)
lemma $(\sigma_1, \sigma_1') \models (X_{Person7} .oclIsNew())$
by(simp add: OclValid-def OclIsNew-def
 σ_1 -def σ_1' -def
X_{Person7}-def person7-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid8-def
oid-of-option-def oid-of-type_{OclAny}-def)

value $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models (X_{Person8} <> X_{Person7})$
value $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models not(v(X_{Person8} .oclAsType(Person)))$
value $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models (X_{Person8} .oclIsTypeOf(OclAny))$
value $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models not(X_{Person8} .oclIsTypeOf(Person))$
value $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models not(X_{Person8} .oclIsKindOf(Person))$
value $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models (X_{Person8} .oclIsKindOf(OclAny))$

lemma σ -modifiedonly: $(\sigma_1, \sigma_1') \models (Set\{ X_{Person1} .oclAsType(OclAny) , X_{Person2} .oclAsType(OclAny) (*, X_{Person3} .oclAsType(OclAny)* , X_{Person4} .oclAsType(OclAny) (*, X_{Person5} .oclAsType(OclAny)* , X_{Person6} .oclAsType(OclAny) (*, X_{Person7} .oclAsType(OclAny)* (*, X_{Person8} .oclAsType(OclAny)* (*, X_{Person9} .oclAsType(OclAny)*)\} \rightarrow oclIsModifiedOnly())$
apply(simp add: OclIsModifiedOnly-def OclValid-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def
X_{Person1}-def X_{Person2}-def X_{Person3}-def X_{Person4}-def X_{Person5}-def X_{Person6}-def
X_{Person7}-def X_{Person8}-def X_{Person9}-def
person1-def person2-def person3-def person4-def person5-def person6-def person7-def
person8-def person9-def
image-def)
apply(simp add: including-rep-set mtSet-rep-set null-option-def bot-option-def)
apply(simp add: oid-of-option-def oid-of-type_{OclAny}-def, clarsimp)
apply(simp add: σ_1 -def σ_1' -def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)
done

lemma $(\sigma_1, \sigma_1') \models ((X_{Person9} @pre (\lambda x. \lfloor OclAsType_{Person} \mathfrak{A} x \rfloor)) \triangleq X_{Person9})$
by(simp add: OclSelf-at-pre-def σ_1 -def oid-of-option-def oid-of-type_{Person}-def
X_{Person9}-def person9-def oid8-def OclValid-def StrongEq-def OclAsType_{Person}- \mathfrak{A} -def)

lemma $(\sigma_1, \sigma_1') \models ((X_{Person9} @post (\lambda x. \lfloor OclAsType_{Person} \mathfrak{A} x \rfloor)) \triangleq X_{Person9})$
by (*simp add: OclSelf-at-post-def* σ_1' -def *oid-of-option-def* *oid-of-type_{Person}-def*
X_{Person9}-def *person9-def* *oid8-def* *OclValid-def* *StrongEq-def* *OclAsType_{Person}- \mathfrak{A} -def*)

lemma $(\sigma_1, \sigma_1') \models (((X_{Person9} .oclAsType(OclAny)) @pre (\lambda x. \lfloor OclAsType_{OclAny} \mathfrak{A} x \rfloor)) \triangleq$
 $((X_{Person9} .oclAsType(OclAny)) @post (\lambda x. \lfloor OclAsType_{OclAny} \mathfrak{A} x \rfloor)))$

proof –

have *including4* : $\bigwedge a b c d \tau. Set\{\lambda\tau. \lfloor \lfloor a \rfloor \rfloor, \lambda\tau. \lfloor \lfloor b \rfloor \rfloor, \lambda\tau. \lfloor \lfloor c \rfloor \rfloor, \lambda\tau. \lfloor \lfloor d \rfloor \rfloor\} \tau = Abs-Set-0 \lfloor \lfloor \{ \lfloor \lfloor a \rfloor \rfloor, \lfloor \lfloor b \rfloor \rfloor, \lfloor \lfloor c \rfloor \rfloor, \lfloor \lfloor d \rfloor \rfloor \} \rfloor \rfloor$
apply (*subst abs-rep-simp'[symmetric]*, *simp*)
by (*simp add: including-rep-set mtSet-rep-set*)

have *excluding1* : $\bigwedge S a b c d e \tau. (\lambda-. Abs-Set-0 \lfloor \lfloor \{ \lfloor \lfloor a \rfloor \rfloor, \lfloor \lfloor b \rfloor \rfloor, \lfloor \lfloor c \rfloor \rfloor, \lfloor \lfloor d \rfloor \rfloor \} \rfloor \rfloor) \rightarrow excluding(\lambda\tau. \lfloor \lfloor e \rfloor \rfloor) \tau = Abs-Set-0 \lfloor \lfloor \{ \lfloor \lfloor a \rfloor \rfloor, \lfloor \lfloor b \rfloor \rfloor, \lfloor \lfloor c \rfloor \rfloor, \lfloor \lfloor d \rfloor \rfloor \} - \{ \lfloor \lfloor e \rfloor \rfloor \} \rfloor \rfloor$
apply (*simp add: OclExcluding-def*)
apply (*simp add: defined-def* *OclValid-def* *bot-fun-def* *bot-Set-0-def* *null-fun-def* *null-Set-0-def* *false-def* *true-def*)
apply (*rule conjI*)
apply (*rule impI*, *subst (asm) Abs-Set-0-inject*) **apply** (*simp add: bot-option-def*) +
apply (*rule conjI*)
apply (*rule impI*, *subst (asm) Abs-Set-0-inject*) **apply** (*simp add: bot-option-def* *null-option-def*) +
apply (*subst Abs-Set-0-inverse*, *simp add: bot-option-def*, *simp*)
done

show ?thesis
apply (*rule framing* [**where** $X = Set\{ X_{Person1} .oclAsType(OclAny)$
 $, X_{Person2} .oclAsType(OclAny)$
 $(*, X_{Person3} .oclAsType(OclAny)*)$
 $, X_{Person4} .oclAsType(OclAny)$
 $(*, X_{Person5} .oclAsType(OclAny)*)$
 $, X_{Person6} .oclAsType(OclAny)$
 $(*, X_{Person7} .oclAsType(OclAny)*)$
 $(*, X_{Person8} .oclAsType(OclAny)*)$
 $(*, X_{Person9} .oclAsType(OclAny)*) \}$])
apply (*cut-tac* σ -modifiedonly)
apply (*simp only: OclValid-def*
 $X_{Person1}$ -def $X_{Person2}$ -def $X_{Person3}$ -def $X_{Person4}$ -def $X_{Person5}$ -def
 $X_{Person6}$ -def $X_{Person7}$ -def $X_{Person8}$ -def $X_{Person9}$ -def
 $person1$ -def $person2$ -def $person3$ -def $person4$ -def $person5$ -def $person6$ -def $person7$ -def
 $person8$ -def $person9$ -def
 $OclAsType_{OclAny}$ -Person)
apply (*subst cp-OclIsModifiedOnly*, *subst cp-OclExcluding*,
subst (asm) cp-OclIsModifiedOnly, *simp add: including4* *excluding1*)
apply (*simp add: OclValid-def* *defined-def*
 $X_{Person9}$ -def $person9$ -def *OclSelf-at-pre-def*)

```

oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def
oid-of-option-def oid-of-typeOclAny-def
σ1-def OclAsTypeOclAny-def bot-fun-def null-fun-def bot-option-def null-option-def
apply(simp only: XPerson1-def XPerson2-def XPerson3-def XPerson4-def XPerson5-def XPerson6-def
XPerson7-def XPerson8-def XPerson9-def
person1-def person2-def person3-def person4-def person5-def person6-def person7-def
person8-def person9-def)
apply(simp add: including-rep-set mtSet-rep-set
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)
apply(simp add: StrictRefEqObject-def null-option-def bot-option-def oid-of-option-def oid-of-typeOclAny-def
OclNot-def OclValid-def)
done
qed

```

lemma perm0 : $\bigwedge S \text{ oid7 oid6 person8 person7. oid7} \neq \text{oid6} \implies$
 $S (\text{oid7} \mapsto \text{person8}) (\text{oid6} \mapsto \text{person7}) = S (\text{oid6} \mapsto \text{person7}) (\text{oid7} \mapsto \text{person8})$
by (metis fun-upd-twist)

lemma perm-σ₁' : σ₁' = $\langle \mid \text{heap} = \text{empty}$
 $(\text{oid8} \mapsto \text{in}_{\text{Person}} \text{person9})$
 $(\text{oid7} \mapsto \text{in}_{\text{OclAny}} \text{person8})$
 $(\text{oid6} \mapsto \text{in}_{\text{OclAny}} \text{person7})$
 $(\text{oid5} \mapsto \text{in}_{\text{Person}} \text{person6})$
 $(*\text{oid4}*)$
 $(\text{oid3} \mapsto \text{in}_{\text{Person}} \text{person4})$
 $(\text{oid2} \mapsto \text{in}_{\text{Person}} \text{person3})$
 $(\text{oid1} \mapsto \text{in}_{\text{Person}} \text{person2})$
 $(\text{oid0} \mapsto \text{in}_{\text{Person}} \text{person1})$
 $, \text{assocs}_2 = \text{assocs}_2 \sigma_1'$
 $, \text{assocs}_3 = \text{assocs}_3 \sigma_1' \mid \rangle$

proof –
note p0 = perm0
show ?thesis
apply(simp add: σ₁'-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)
apply(subst (1) p0, simp)
apply(subst (2) p0, simp) **apply**(subst (1) p0, simp)
apply(subst (3) p0, simp) **apply**(subst (2) p0, simp) **apply**(subst (1) p0, simp)
apply(subst (4) p0, simp) **apply**(subst (3) p0, simp) **apply**(subst (2) p0, simp) **apply**(subst
(1) p0, simp)
apply(subst (5) p0, simp) **apply**(subst (4) p0, simp) **apply**(subst (3) p0, simp) **apply**(subst
(2) p0, simp) **apply**(subst (1) p0, simp)
apply(subst (6) p0, simp) **apply**(subst (5) p0, simp) **apply**(subst (4) p0, simp) **apply**(subst
(3) p0, simp) **apply**(subst (2) p0, simp) **apply**(subst (1) p0, simp)
apply(subst (7) p0, simp) **apply**(subst (6) p0, simp) **apply**(subst (5) p0, simp) **apply**(subst
(4) p0, simp) **apply**(subst (3) p0, simp) **apply**(subst (2) p0, simp) **apply**(subst (1) p0, simp)
by(simp)
qed

end

7. Part IV (bis): OCL Contracts and an Example

```
theory
  Employee-AnalysisModel-OCLPart
imports
  Employee-AnalysisModel-UMLPart
begin
```

7.1. Standard State Infrastructure

These definitions should be generated — again — from the class diagram.

7.2. Invariant

These recursive predicates can be defined conservatively by greatest fix-point constructions - automatically. See HOL-OCL Book for details. For the purpose of this example, we state them as axioms here.

```
axiomatization inv-Person :: Person  $\Rightarrow$  Boolean
where A : ( $\tau \models (\delta \text{ self})$ )  $\longrightarrow$ 
  ( $\tau \models \text{inv-Person}(\text{self})$ ) =
    (( $\tau \models (\text{self}.\text{boss} \doteq \text{null})$ )  $\vee$ 
     ( $\tau \models (\text{self}.\text{boss} <> \text{null}) \wedge (\tau \models ((\text{self}.\text{salary}) \leq_{ocl} (\text{self}.\text{boss}.\text{salary}))) \wedge$ 
     ( $\tau \models (\text{inv-Person}(\text{self}.\text{boss}))$ )))
```

```
axiomatization inv-Person-at-pre :: Person  $\Rightarrow$  Boolean
where B : ( $\tau \models (\delta \text{ self})$ )  $\longrightarrow$ 
  ( $\tau \models \text{inv-Person-at-pre}(\text{self})$ ) =
    (( $\tau \models (\text{self}.\text{boss@pre} \doteq \text{null})$ )  $\vee$ 
     ( $\tau \models (\text{self}.\text{boss@pre} <> \text{null}) \wedge$ 
     ( $\tau \models (\text{self}.\text{boss@pre}.\text{salary@pre} \leq_{ocl} \text{self}.\text{salary@pre})$ )  $\wedge$ 
     ( $\tau \models (\text{inv-Person-at-pre}(\text{self}.\text{boss@pre}))$ )))
```

A very first attempt to characterize the axiomatization by an inductive definition - this can not be the last word since too weak (should be equality!)

```
coinductive inv :: Person  $\Rightarrow$  ( $\mathcal{A}$ )st  $\Rightarrow$  bool where
  ( $\tau \models (\delta \text{ self})$ )  $\implies$  (( $\tau \models (\text{self}.\text{boss} \doteq \text{null})$ )  $\vee$ 
    ( $\tau \models (\text{self}.\text{boss} <> \text{null}) \wedge (\tau \models (\text{self}.\text{boss}.\text{salary} \leq_{ocl} \text{self}.\text{salary})) \wedge$ 
    ( $(\text{inv}(\text{self}.\text{boss}))\tau$ )))
```

$$\implies (\text{inv self } \tau)$$

7.3. The contract of a recursive query :

The original specification of a recursive query :

```
context Person::contents():Set(Integer)
post:  result = if self.boss = null
           then Set{i}
           else self.boss.contents()->including(i)
           endif
```

consts *dot-contents* :: *Person* \Rightarrow *Set-Integer* ((1(-).contents'(')) 50)

axiomatization where *dot-contents-def*:

```
( $\tau \models ((\text{self}).\text{contents}() \triangleq \text{result})$ ) =
  (if ( $\delta \text{ self}$ )  $\tau = \text{true}$   $\tau$ 
    then (( $\tau \models \text{true}$ )  $\wedge$ 
          ( $\tau \models (\text{result} \triangleq \text{if } (\text{self}. \text{boss} \doteq \text{null})$ 
            then (Set{self .salary})
            else (self .boss .contents()->including(self .salary))
            endif))))
    else  $\tau \models \text{result} \triangleq \text{invalid}$ )
```

consts *dot-contents-AT-pre* :: *Person* \Rightarrow *Set-Integer* ((1(-).contents@pre'(')) 50)

axiomatization where *dot-contents-AT-pre-def*:

```
( $\tau \models (\text{self}).\text{contents@pre}() \triangleq \text{result}$ ) =
  (if ( $\delta \text{ self}$ )  $\tau = \text{true}$   $\tau$ 
    then  $\tau \models \text{true} \wedge$ 
          ( $\tau \models (\text{result} \triangleq \text{if } (\text{self}).\text{boss@pre} \doteq \text{null}$ 
            then Set{(self).salary@pre}
            else (self).boss@pre .contents@pre()->including(self .salary@pre)
            endif))
          (* pre *)
          (* post *)
    else  $\tau \models \text{result} \triangleq \text{invalid}$ )
```

Note that these @pre variants on methods are only available on queries, i.e. operations without side-effect.

7.4. The contract of a method.

The specification in high-level OCL input syntax reads as follows:

```
context Person::insert(x:Integer)
post: contents():Set(Integer)
contents() = contents@pre()->including(x)
```

consts *dot-insert* :: *Person* \Rightarrow *Integer* \Rightarrow *Void* ((1(-).insert'(-)) 50)

axiomatization where *dot-insert-def*:

($\tau \models ((self).insert(x) \triangleq result)$) =
(if ($\delta\ self$) $\tau = true$ $\tau \wedge (v\ x)\ \tau = true$ τ
 then $\tau \models true \wedge$
 $\tau \models ((self).contents() \triangleq (self).contents@pre()->including(x))$
 else $\tau \models ((self).insert(x) \triangleq invalid)$)

end

8. Part V: OCL Contracts and an Example

```
theory
  Employee-DesignModel-UMLPart
imports
  ../src/OCL-main
begin
```

8.0.1. Introduction

For certain concepts like Classes and Class-types, only a generic definition for its resulting semantics can be given. Generic means, there is a function outside HOL that "compiles" a concrete, closed-world class diagram into a "theory" of this data model, consisting of a bunch of definitions for classes, accessors, method, casts, and tests for actual types, as well as proofs for the fundamental properties of these operations in this concrete data model.

Such generic function or "compiler" can be implemented in Isabelle on the ML level. This has been done, for a semantics following the open-world assumption, for UML 2.0 in [9]. In this paper, we follow another approach for UML 2.4: we define the concepts of the compilation informally, and present a concrete example which is verified in Isabelle/HOL.

8.0.2. Outlining the Example

We are presenting here a design-model of the (slightly modified) example Figure 7.3, page 20 of the [?]. To be precise, this theory contains the formalization of the DATA-part covered by the UML diagram: ... This means that the association (attached to the association class *EmployeeRanking*) with the association ends **boss** and **employees** is implemented by the attribute **boss** and the operation **employees** (to be discussed in the OCL part captured by the subsequent theory).

8.1. Example Data-Universe and its Infrastructure

Should be generated entirely from a class-diagram.

Our data universe consists in the concrete class diagram just of node's, and implicitly of the class object. Each class implies the existence of a class type defined for the corresponding object representations as follows:

```
datatype typePerson = mkPerson oid
                    int option
```

oid option

```
datatype typeOclAny = mkOclAny oid  
                      (int option × oid option) option
```

Now, we construct a concrete "universe of oclany types" by injection into a sum type containing the class types. This type of oclanys will be used as instance for all resp. type-variables ...

```
datatype  $\mathfrak{A}$  = inPerson typePerson | inOclAny typeOclAny
```

Having fixed the object universe, we can introduce type synonyms that exactly correspond to OCL types. Again, we exploit that our representation of OCL is a "shallow embedding" with a one-to-one correspondance of OCL-types to types of the meta-language HOL.

```
type-synonym Boolean    =  $\mathfrak{A}$  Boolean  
type-synonym Integer   =  $\mathfrak{A}$  Integer  
type-synonym Void      =  $\mathfrak{A}$  Void  
type-synonym OclAny    = ( $\mathfrak{A}$ , typeOclAny option option) val  
type-synonym Person    = ( $\mathfrak{A}$ , typePerson option option) val  
type-synonym Set-Integer = ( $\mathfrak{A}$ , int option option) Set  
type-synonym Set-Person = ( $\mathfrak{A}$ , typePerson option option) Set
```

Just a little check:

```
typ Boolean
```

In order to reuse key-elements of the library like referential equality, we have to show that the object universe belongs to the type class "oclany", i.e. each class type has to provide a function *oid-of* yielding the object id (oid) of the object.

```
instantiation typePerson :: object  
begin  
  definition oid-of-typePerson-def: oid-of x = (case x of mkPerson oid - -  $\Rightarrow$  oid)  
  instance ..  
end
```

```
instantiation typeOclAny :: object  
begin  
  definition oid-of-typeOclAny-def: oid-of x = (case x of mkOclAny oid -  $\Rightarrow$  oid)  
  instance ..  
end
```

```
instantiation  $\mathfrak{A}$  :: object  
begin  
  definition oid-of-A-def: oid-of x = (case x of  
                                         inPerson person  $\Rightarrow$  oid-of person  
                                         | inOclAny oclany  $\Rightarrow$  oid-of oclany)  
  instance ..  
end
```

8.2. Instantiation of the generic strict equality. We instantiate the referential equality on *Person* and *OclAny*

```

defs(overloaded) StrictRefEqObject-Person : (x::Person)  $\doteq$  y  $\equiv$  StrictRefEqObject x y
defs(overloaded) StrictRefEqObject-OclAny : (x::OclAny)  $\doteq$  y  $\equiv$  StrictRefEqObject x y

```

lemmas

```

cp-StrictRefEqObject [of x::Person y::Person  $\tau$ ,
                        simplified StrictRefEqObject-Person [symmetric]]
cp-intro(9)          [of P::Person  $\Rightarrow$  Person Q::Person  $\Rightarrow$  Person,
                        simplified StrictRefEqObject-Person [symmetric]]
StrictRefEqObject-def [of x::Person y::Person,
                        simplified StrictRefEqObject-Person [symmetric]]
StrictRefEqObject-defargs [of - x::Person y::Person,
                             simplified StrictRefEqObject-Person [symmetric]]
StrictRefEqObject-strict1
  [of x::Person,
   simplified StrictRefEqObject-Person [symmetric]]
StrictRefEqObject-strict2
  [of x::Person,
   simplified StrictRefEqObject-Person [symmetric]]

```

For each Class *C*, we will have a casting operation *.oclAsType*(*C*), a test on the actual type *.oclIsTypeOf*(*C*) as well as its relaxed form *.oclIsKindOf*(*C*) (corresponding exactly to Java's *instanceof*-operator).

Thus, since we have two class-types in our concrete class hierarchy, we have two operations to declare and to provide two overloading definitions for the two static types.

8.3. OclAsType

8.3.1. Definition

```

consts OclAsTypeOclAny :: ' $\alpha \Rightarrow$  OclAny ((-) .oclAsType'(OclAny'))
consts OclAsTypePerson :: ' $\alpha \Rightarrow$  Person ((-) .oclAsType'(Person'))

```

definition *OclAsTypeOclAny- \mathfrak{A}* = ($\lambda u.$ [case *u* of *inOclAny* *a* \Rightarrow *a*
| *inPerson* (*mkPerson* *oid a b*) \Rightarrow *mkOclAny* *oid [(a,b)]*])

lemma *OclAsTypeOclAny- \mathfrak{A} -some*: *OclAsTypeOclAny- \mathfrak{A}* *x* \neq *None*
by(*simp add: OclAsTypeOclAny- \mathfrak{A} -def*)

```

defs (overloaded) OclAsTypeOclAny-OclAny:
  (X::OclAny) .oclAsType(OclAny)  $\equiv$  X

```

```

defs (overloaded) OclAsTypeOclAny-Person:
  (X::Person) .oclAsType(OclAny)  $\equiv$ 
    ( $\lambda \tau.$  case X  $\tau$  of

```

$$\begin{aligned}
& \perp \Rightarrow \text{invalid } \tau \\
& | \lfloor \perp \rfloor \Rightarrow \text{null } \tau \\
& | \lfloor \lfloor \text{mk}_{\text{Person}} \text{ oid } a \ b \rfloor \rfloor \Rightarrow \lfloor \lfloor \text{mk}_{\text{OclAny}} \text{ oid } \lfloor (a,b) \rfloor \rfloor \rfloor
\end{aligned}$$

definition $\text{OclAsType}_{\text{Person}}\text{-}\mathfrak{A} = (\lambda u. \text{ case } u \text{ of } \text{in}_{\text{Person}} \ p \Rightarrow \lfloor p \rfloor$
 $| \text{in}_{\text{OclAny}} \ (\text{mk}_{\text{OclAny}} \ \text{oid } \lfloor (a,b) \rfloor) \Rightarrow \lfloor \text{mk}_{\text{Person}} \ \text{oid } a \ b \rfloor$
 $| - \Rightarrow \text{None})$

defs (overloaded) $\text{OclAsType}_{\text{Person}}\text{-OclAny}$:
 $(X :: \text{OclAny}) . \text{oclAsType}(\text{Person}) \equiv$
 $(\lambda \tau. \text{ case } X \ \tau \text{ of}$
 $\perp \Rightarrow \text{invalid } \tau$
 $| \lfloor \perp \rfloor \Rightarrow \text{null } \tau$
 $| \lfloor \lfloor \text{mk}_{\text{OclAny}} \text{ oid } \perp \rfloor \rfloor \Rightarrow \text{invalid } \tau \quad (* \text{ down-cast exception } *)$
 $| \lfloor \lfloor \text{mk}_{\text{OclAny}} \text{ oid } \lfloor (a,b) \rfloor \rfloor \rfloor \Rightarrow \lfloor \lfloor \text{mk}_{\text{Person}} \text{ oid } a \ b \rfloor \rfloor)$

defs (overloaded) $\text{OclAsType}_{\text{Person}}\text{-Person}$:
 $(X :: \text{Person}) . \text{oclAsType}(\text{Person}) \equiv X$

lemmas $[\text{simp}] =$
 $\text{OclAsType}_{\text{OclAny}}\text{-OclAny}$
 $\text{OclAsType}_{\text{Person}}\text{-Person}$

8.3.2. Context Passing

lemma $\text{cp-OclAsType}_{\text{OclAny}}\text{-Person-Person}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P \ (X :: \text{Person}) :: \text{Person}) . \text{oclAsType}(\text{OclAny})$
 $\text{by}(\text{rule } \text{cpI1}, \text{ simp-all add: } \text{OclAsType}_{\text{OclAny}}\text{-Person})$
lemma $\text{cp-OclAsType}_{\text{OclAny}}\text{-OclAny-OclAny}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P \ (X :: \text{OclAny}) :: \text{OclAny})$
 $. \text{oclAsType}(\text{OclAny}))$
 $\text{by}(\text{rule } \text{cpI1}, \text{ simp-all add: } \text{OclAsType}_{\text{OclAny}}\text{-OclAny})$
lemma $\text{cp-OclAsType}_{\text{Person}}\text{-Person-Person}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P \ (X :: \text{Person}) :: \text{Person}) . \text{oclAsType}(\text{Person})$
 $\text{by}(\text{rule } \text{cpI1}, \text{ simp-all add: } \text{OclAsType}_{\text{Person}}\text{-Person})$
lemma $\text{cp-OclAsType}_{\text{Person}}\text{-OclAny-OclAny}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P \ (X :: \text{OclAny}) :: \text{OclAny})$
 $. \text{oclAsType}(\text{Person}))$
 $\text{by}(\text{rule } \text{cpI1}, \text{ simp-all add: } \text{OclAsType}_{\text{Person}}\text{-OclAny})$
lemma $\text{cp-OclAsType}_{\text{OclAny}}\text{-Person-OclAny}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P \ (X :: \text{Person}) :: \text{OclAny})$
 $. \text{oclAsType}(\text{OclAny}))$
 $\text{by}(\text{rule } \text{cpI1}, \text{ simp-all add: } \text{OclAsType}_{\text{OclAny}}\text{-OclAny})$
lemma $\text{cp-OclAsType}_{\text{OclAny}}\text{-OclAny-Person}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P \ (X :: \text{OclAny}) :: \text{Person})$
 $. \text{oclAsType}(\text{OclAny}))$
 $\text{by}(\text{rule } \text{cpI1}, \text{ simp-all add: } \text{OclAsType}_{\text{OclAny}}\text{-Person})$
lemma $\text{cp-OclAsType}_{\text{Person}}\text{-Person-OclAny}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P \ (X :: \text{Person}) :: \text{OclAny})$
 $. \text{oclAsType}(\text{Person}))$
 $\text{by}(\text{rule } \text{cpI1}, \text{ simp-all add: } \text{OclAsType}_{\text{Person}}\text{-OclAny})$
lemma $\text{cp-OclAsType}_{\text{Person}}\text{-OclAny-Person}$: $\text{cp } P \Rightarrow \text{cp}(\lambda X. (P \ (X :: \text{OclAny}) :: \text{Person})$
 $. \text{oclAsType}(\text{Person}))$
 $\text{by}(\text{rule } \text{cpI1}, \text{ simp-all add: } \text{OclAsType}_{\text{Person}}\text{-Person})$

```

lemmas [simp] =
  cp-OclAsTypeOclAny-Person-Person
  cp-OclAsTypeOclAny-OclAny-OclAny
  cp-OclAsTypePerson-Person-Person
  cp-OclAsTypePerson-OclAny-OclAny

  cp-OclAsTypeOclAny-Person-OclAny
  cp-OclAsTypeOclAny-OclAny-Person
  cp-OclAsTypePerson-Person-OclAny
  cp-OclAsTypePerson-OclAny-Person

```

8.3.3. Execution with invalid or null as argument

```

lemma OclAsTypeOclAny-OclAny-strict : (invalid::OclAny) .oclAsType(OclAny) = invalid
by(simp)

```

```

lemma OclAsTypeOclAny-OclAny-nullstrict : (null::OclAny) .oclAsType(OclAny) = null
by(simp)

```

```

lemma OclAsTypeOclAny-Person-strict[simp] : (invalid::Person) .oclAsType(OclAny) = invalid
by(rule ext, simp add: bot-option-def invalid-def
    OclAsTypeOclAny-Person)

```

```

lemma OclAsTypeOclAny-Person-nullstrict[simp] : (null::Person) .oclAsType(OclAny) = null
by(rule ext, simp add: null-fun-def null-option-def bot-option-def
    OclAsTypeOclAny-Person)

```

```

lemma OclAsTypePerson-OclAny-strict[simp] : (invalid::OclAny) .oclAsType(Person) = invalid
by(rule ext, simp add: bot-option-def invalid-def
    OclAsTypePerson-OclAny)

```

```

lemma OclAsTypePerson-OclAny-nullstrict[simp] : (null::OclAny) .oclAsType(Person) = null
by(rule ext, simp add: null-fun-def null-option-def bot-option-def
    OclAsTypePerson-OclAny)

```

```

lemma OclAsTypePerson-Person-strict : (invalid::Person) .oclAsType(Person) = invalid
by(simp)

```

```

lemma OclAsTypePerson-Person-nullstrict : (null::Person) .oclAsType(Person) = null
by(simp)

```

8.4. OclIsTypeOf

8.4.1. Definition

```

consts OclIsTypeOfOclAny :: 'α ⇒ Boolean ((-).oclIsTypeOf'(OclAny'))
consts OclIsTypeOfPerson :: 'α ⇒ Boolean ((-).oclIsTypeOf'(Person'))

```

```

defs (overloaded) OclIsTypeOfOclAny-OclAny:

```

$$\begin{aligned}
& (X::OclAny) .oclIsTypeOf(OclAny) \equiv \\
& \quad (\lambda\tau. \text{ case } X \ \tau \text{ of} \\
& \quad \quad \perp \Rightarrow \text{invalid } \tau \\
& \quad \quad | \lfloor \perp \rfloor \Rightarrow \text{true } \tau \quad (* \text{ invalid } ?? *) \\
& \quad \quad | \lfloor \lfloor mk_{OclAny} \text{ oid } \perp \rfloor \rfloor \Rightarrow \text{true } \tau \\
& \quad \quad | \lfloor \lfloor mk_{OclAny} \text{ oid } \lfloor - \rfloor \rfloor \rfloor \Rightarrow \text{false } \tau)
\end{aligned}$$

defs (overloaded) $OclIsTypeOf_{OclAny-Person}$:

$$\begin{aligned}
& (X::Person) .oclIsTypeOf(OclAny) \equiv \\
& \quad (\lambda\tau. \text{ case } X \ \tau \text{ of} \\
& \quad \quad \perp \Rightarrow \text{invalid } \tau \\
& \quad \quad | \lfloor \perp \rfloor \Rightarrow \text{true } \tau \quad (* \text{ invalid } ?? *) \\
& \quad \quad | \lfloor \lfloor - \rfloor \rfloor \Rightarrow \text{false } \tau)
\end{aligned}$$

defs (overloaded) $OclIsTypeOf_{Person-OclAny}$:

$$\begin{aligned}
& (X::OclAny) .oclIsTypeOf(Person) \equiv \\
& \quad (\lambda\tau. \text{ case } X \ \tau \text{ of} \\
& \quad \quad \perp \Rightarrow \text{invalid } \tau \\
& \quad \quad | \lfloor \perp \rfloor \Rightarrow \text{true } \tau \\
& \quad \quad | \lfloor \lfloor mk_{OclAny} \text{ oid } \perp \rfloor \rfloor \Rightarrow \text{false } \tau \\
& \quad \quad | \lfloor \lfloor mk_{OclAny} \text{ oid } \lfloor - \rfloor \rfloor \rfloor \Rightarrow \text{true } \tau)
\end{aligned}$$

defs (overloaded) $OclIsTypeOf_{Person-Person}$:

$$\begin{aligned}
& (X::Person) .oclIsTypeOf(Person) \equiv \\
& \quad (\lambda\tau. \text{ case } X \ \tau \text{ of} \\
& \quad \quad \perp \Rightarrow \text{invalid } \tau \\
& \quad \quad | - \Rightarrow \text{true } \tau)
\end{aligned}$$

8.4.2. Context Passing

lemma $cp\text{-}OclIsTypeOf_{OclAny-Person-Person}$: $cp \ P \implies cp(\lambda X. (P \ (X::Person) :: Person) .oclIsTypeOf(OclAny))$

by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny-Person}$)

lemma $cp\text{-}OclIsTypeOf_{OclAny-OclAny-OclAny}$: $cp \ P \implies cp(\lambda X. (P \ (X::OclAny) :: OclAny) .oclIsTypeOf(OclAny))$

by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny-OclAny}$)

lemma $cp\text{-}OclIsTypeOf_{Person-Person-Person}$: $cp \ P \implies cp(\lambda X. (P \ (X::Person) :: Person) .oclIsTypeOf(Person))$

by(rule $cpI1$, simp-all add: $OclIsTypeOf_{Person-Person}$)

lemma $cp\text{-}OclIsTypeOf_{Person-OclAny-OclAny}$: $cp \ P \implies cp(\lambda X. (P \ (X::OclAny) :: OclAny) .oclIsTypeOf(Person))$

by(rule $cpI1$, simp-all add: $OclIsTypeOf_{Person-OclAny}$)

lemma $cp\text{-}OclIsTypeOf_{OclAny-Person-OclAny}$: $cp \ P \implies cp(\lambda X. (P \ (X::Person) :: OclAny) .oclIsTypeOf(OclAny))$

by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny-OclAny}$)

lemma $cp\text{-}OclIsTypeOf_{OclAny-OclAny-Person}$: $cp \ P \implies cp(\lambda X. (P \ (X::OclAny) :: Person) .oclIsTypeOf(Person))$

```

.ocIsTypeOf (OclAny))
by(rule cpI1, simp-all add: OclIsTypeOfOclAny-Person)
lemma cp-OclIsTypeOfPerson-Person-OclAny: cp P  $\implies$  cp( $\lambda X. (P (X::Person) :: OclAny)$ 
.ocIsTypeOf (Person))
by(rule cpI1, simp-all add: OclIsTypeOfPerson-OclAny)
lemma cp-OclIsTypeOfPerson-OclAny-Person: cp P  $\implies$  cp( $\lambda X. (P (X::OclAny) :: Person)$ 
.ocIsTypeOf (Person))
by(rule cpI1, simp-all add: OclIsTypeOfPerson-Person)

lemmas [simp] =
  cp-OclIsTypeOfOclAny-Person-Person
  cp-OclIsTypeOfOclAny-OclAny-OclAny
  cp-OclIsTypeOfPerson-Person-Person
  cp-OclIsTypeOfPerson-OclAny-OclAny

  cp-OclIsTypeOfOclAny-Person-OclAny
  cp-OclIsTypeOfOclAny-OclAny-Person
  cp-OclIsTypeOfPerson-Person-OclAny
  cp-OclIsTypeOfPerson-OclAny-Person

```

8.4.3. Execution with invalid or null as argument

```

lemma OclIsTypeOfOclAny-OclAny-strict1[simp]:
  (invalid::OclAny) .ocIsTypeOf (OclAny) = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OclIsTypeOfOclAny-OclAny)
lemma OclIsTypeOfOclAny-OclAny-strict2[simp]:
  (null::OclAny) .ocIsTypeOf (OclAny) = true
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OclIsTypeOfOclAny-OclAny)
lemma OclIsTypeOfOclAny-Person-strict1[simp]:
  (invalid::Person) .ocIsTypeOf (OclAny) = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OclIsTypeOfOclAny-Person)
lemma OclIsTypeOfOclAny-Person-strict2[simp]:
  (null::Person) .ocIsTypeOf (OclAny) = true
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OclIsTypeOfOclAny-Person)
lemma OclIsTypeOfPerson-OclAny-strict1[simp]:
  (invalid::OclAny) .ocIsTypeOf (Person) = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OclIsTypeOfPerson-OclAny)
lemma OclIsTypeOfPerson-OclAny-strict2[simp]:
  (null::OclAny) .ocIsTypeOf (Person) = true
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
  OclIsTypeOfPerson-OclAny)
lemma OclIsTypeOfPerson-Person-strict1[simp]:
  (invalid::Person) .ocIsTypeOf (Person) = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def

```

```

      OclIsTypeOfPerson-Person)
lemma OclIsTypeOfPerson-Person-strict2[simp]:
  (null::Person) .oclIsTypeOf(Person) = true
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
      OclIsTypeOfPerson-Person)

```

8.4.4. up down cast

```

lemma actualType-larger-staticType:
assumes isdef:  $\tau \models (\delta \ X)$ 
shows  $\tau \models (X::Person) .oclIsTypeOf(OclAny) \triangleq false$ 
using isdef
by(auto simp : null-option-def bot-option-def
      OclIsTypeOfOclAny-Person foundation22 foundation16)

lemma down-cast-type:
assumes isOclAny:  $\tau \models (X::OclAny) .oclIsTypeOf(OclAny)$ 
and non-null:  $\tau \models (\delta \ X)$ 
shows  $\tau \models (X .oclAsType(Person)) \triangleq invalid$ 
using isOclAny non-null
apply(auto simp : bot-fun-def null-fun-def null-option-def bot-option-def null-def invalid-def
      OclAsTypeOclAny-Person OclAsTypePerson-OclAny foundation22 foundation16
      split: option.split typeOclAny.split typePerson.split)
by(simp add: OclIsTypeOfOclAny-OclAny OclValid-def false-def true-def)

```

```

lemma down-cast-type':
assumes isOclAny:  $\tau \models (X::OclAny) .oclIsTypeOf(OclAny)$ 
and non-null:  $\tau \models (\delta \ X)$ 
shows  $\tau \models not \ (v \ (X .oclAsType(Person)))$ 
by(rule foundation15[THEN iffD1], simp add: down-cast-type[OF assms])

```

```

lemma up-down-cast :
assumes isdef:  $\tau \models (\delta \ X)$ 
shows  $\tau \models ((X::Person) .oclAsType(OclAny) .oclAsType(Person) \triangleq X)$ 
using isdef
by(auto simp : null-fun-def null-option-def bot-option-def null-def invalid-def
      OclAsTypeOclAny-Person OclAsTypePerson-OclAny foundation22 foundation16
      split: option.split typePerson.split)

```

```

lemma up-down-cast-Person-OclAny-Person [simp]:
shows  $((X::Person) .oclAsType(OclAny) .oclAsType(Person) = X)$ 
apply(rule ext, rename-tac  $\tau$ )
apply(rule foundation22[THEN iffD1])
apply(case-tac  $\tau \models (\delta \ X)$ , simp add: up-down-cast)
apply(simp add: def-split-local, elim disjE)
apply(erule StrongEq-L-subst2-rev, simp, simp)+
done

```



```

lemma up-down-cast-Person-OclAny-Person': assumes  $\tau \models v \ X$ 
shows  $\tau \models (((X :: \text{Person}) .\text{oclAsType}(\text{OclAny}) .\text{oclAsType}(\text{Person})) \doteq X)$ 
  apply(simp only: up-down-cast-Person-OclAny-Person StrictRefEqObject-Person)
by(rule StrictRefEqObject-sym, simp add: assms)

lemma up-down-cast-Person-OclAny-Person'': assumes  $\tau \models v \ (X :: \text{Person})$ 
shows  $\tau \models (X .\text{oclIsTypeOf}(\text{Person}) \text{ implies } (X .\text{oclAsType}(\text{OclAny}) .\text{oclAsType}(\text{Person})) \doteq X)$ 
  apply(simp add: OclValid-def)
  apply(subst cp-OclImplies)
  apply(simp add: StrictRefEqObject-Person StrictRefEqObject-sym[OF assms, simplified OclValid-def])
  apply(subst cp-OclImplies[symmetric])
by (simp add: OclImplies-true)

```

8.5. OclIsKindOf

8.5.1. Definition

```

consts OclIsKindOfOclAny :: ' $\alpha \Rightarrow \text{Boolean}$  ((-).oclIsKindOf'(OclAny'))
consts OclIsKindOfPerson :: ' $\alpha \Rightarrow \text{Boolean}$  ((-).oclIsKindOf'(Person'))

```

```

defs (overloaded) OclIsKindOfOclAny-OclAny:
  (X::OclAny) .oclIsKindOf(OclAny)  $\equiv$ 
    ( $\lambda\tau$ . case X  $\tau$  of
       $\perp \Rightarrow \text{invalid } \tau$ 
      |  $- \Rightarrow \text{true } \tau$ )

```

```

defs (overloaded) OclIsKindOfOclAny-Person:
  (X::Person) .oclIsKindOf(OclAny)  $\equiv$ 
    ( $\lambda\tau$ . case X  $\tau$  of
       $\perp \Rightarrow \text{invalid } \tau$ 
      |  $- \Rightarrow \text{true } \tau$ )

```

```

defs (overloaded) OclIsKindOfPerson-OclAny:
  (X::OclAny) .oclIsKindOf(Person)  $\equiv$ 
    ( $\lambda\tau$ . case X  $\tau$  of
       $\perp \Rightarrow \text{invalid } \tau$ 
      |  $[\perp] \Rightarrow \text{true } \tau$ 
      |  $[[mk_{OclAny} \text{ oid } \perp]] \Rightarrow \text{false } \tau$ 
      |  $[[mk_{OclAny} \text{ oid } [-]]] \Rightarrow \text{true } \tau$ )

```

```

defs (overloaded) OclIsKindOfPerson-Person:
  (X::Person) .oclIsKindOf(Person)  $\equiv$ 
    ( $\lambda\tau$ . case X  $\tau$  of
       $\perp \Rightarrow \text{invalid } \tau$ 
      |  $- \Rightarrow \text{true } \tau$ )

```

8.5.2. Context Passing

```

lemma cp-OclIsKindOfOclAny-Person-Person: cp P  $\implies$  cp( $\lambda X. (P (X::Person) :: Person)$ 
.oclIsKindOf(OclAny))
by(rule cpI1, simp-all add: OclIsKindOfOclAny-Person)
lemma cp-OclIsKindOfOclAny-OclAny-OclAny: cp P  $\implies$  cp( $\lambda X. (P (X::OclAny) :: OclAny)$ 
.oclIsKindOf(OclAny))
by(rule cpI1, simp-all add: OclIsKindOfOclAny-OclAny)
lemma cp-OclIsKindOfPerson-Person-Person: cp P  $\implies$  cp( $\lambda X. (P (X::Person) :: Person)$ 
.oclIsKindOf(Person))
by(rule cpI1, simp-all add: OclIsKindOfPerson-Person)
lemma cp-OclIsKindOfPerson-OclAny-OclAny: cp P  $\implies$  cp( $\lambda X. (P (X::OclAny) :: OclAny)$ 
.oclIsKindOf(Person))
by(rule cpI1, simp-all add: OclIsKindOfPerson-OclAny)

lemma cp-OclIsKindOfOclAny-Person-OclAny: cp P  $\implies$  cp( $\lambda X. (P (X::Person) :: OclAny)$ 
.oclIsKindOf(OclAny))
by(rule cpI1, simp-all add: OclIsKindOfOclAny-OclAny)
lemma cp-OclIsKindOfOclAny-OclAny-Person: cp P  $\implies$  cp( $\lambda X. (P (X::OclAny) :: Person)$ 
.oclIsKindOf(OclAny))
by(rule cpI1, simp-all add: OclIsKindOfOclAny-Person)
lemma cp-OclIsKindOfPerson-Person-OclAny: cp P  $\implies$  cp( $\lambda X. (P (X::Person) :: OclAny)$ 
.oclIsKindOf(Person))
by(rule cpI1, simp-all add: OclIsKindOfPerson-OclAny)
lemma cp-OclIsKindOfPerson-OclAny-Person: cp P  $\implies$  cp( $\lambda X. (P (X::OclAny) :: Person)$ 
.oclIsKindOf(Person))
by(rule cpI1, simp-all add: OclIsKindOfPerson-Person)

lemmas [simp] =
  cp-OclIsKindOfOclAny-Person-Person
  cp-OclIsKindOfOclAny-OclAny-OclAny
  cp-OclIsKindOfPerson-Person-Person
  cp-OclIsKindOfPerson-OclAny-OclAny

  cp-OclIsKindOfOclAny-Person-OclAny
  cp-OclIsKindOfOclAny-OclAny-Person
  cp-OclIsKindOfPerson-Person-OclAny
  cp-OclIsKindOfPerson-OclAny-Person

```

8.5.3. Execution with invalid or null as argument

```

lemma OclIsKindOfOclAny-OclAny-strict1[simp] : (invalid::OclAny) .oclIsKindOf(OclAny) =
invalid
by(rule ext, simp add: invalid-def bot-option-def
OclIsKindOfOclAny-OclAny)

lemma OclIsKindOfOclAny-OclAny-strict2[simp] : (null::OclAny) .oclIsKindOf(OclAny) = true
by(rule ext, simp add: null-fun-def null-option-def
OclIsKindOfOclAny-OclAny)

```

lemma *OclIsKindOf_{OclAny}-Person-strict1*[simp] : (invalid::Person) .oclIsKindOf(OclAny) = invalid

by(rule ext, simp add: bot-option-def invalid-def
OclIsKindOf_{OclAny}-Person)

lemma *OclIsKindOf_{OclAny}-Person-strict2*[simp] : (null::Person) .oclIsKindOf(OclAny) = true

by(rule ext, simp add: null-fun-def null-option-def bot-option-def
OclIsKindOf_{OclAny}-Person)

lemma *OclIsKindOf_{Person}-OclAny-strict1*[simp] : (invalid::OclAny) .oclIsKindOf(Person) = invalid

by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
OclIsKindOf_{Person}-OclAny)

lemma *OclIsKindOf_{Person}-OclAny-strict2*[simp] : (null::OclAny) .oclIsKindOf(Person) = true

by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
OclIsKindOf_{Person}-OclAny)

lemma *OclIsKindOf_{Person}-Person-strict1*[simp] : (invalid::Person) .oclIsKindOf(Person) = invalid

by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
OclIsKindOf_{Person}-Person)

lemma *OclIsKindOf_{Person}-Person-strict2*[simp] : (null::Person) .oclIsKindOf(Person) = true

by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def
OclIsKindOf_{Person}-Person)

8.5.4. up down cast

lemma *actualKind-larger-staticKind*:

assumes isdef: $\tau \models (\delta \ X)$

shows $\tau \models (X::Person) .oclIsKindOf(OclAny) \triangleq true$

using isdef

by(auto simp : bot-option-def
OclIsKindOf_{OclAny}-Person foundation22 foundation16)

lemma *down-cast-kind*:

assumes isOclAny: $\neg \tau \models (X::OclAny) .oclIsKindOf(Person)$

and non-null: $\tau \models (\delta \ X)$

shows $\tau \models (X .oclAsType(Person)) \triangleq invalid$

using isOclAny non-null

apply(auto simp : bot-fun-def null-fun-def null-option-def bot-option-def null-def invalid-def
OclAsType_{OclAny}-Person OclAsType_{Person}-OclAny foundation22 foundation16
split: option.split type_{OclAny}.split type_{Person}.split)

by(simp add: OclIsKindOf_{Person}-OclAny OclValid-def false-def true-def)

8.6. OclAllInstances

Recall that in order to denote OCL-types occurring 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 $Person \equiv OclAsType_{Person}\mathcal{A}$

definition $OclAny \equiv OclAsType_{OclAny}\mathcal{A}$

lemmas $[simp] = Person\text{-}def\ OclAny\text{-}def$

lemma $OclAllInstances_{OclAny}\text{-}exec: OclAny.allInstances() =$
 $(\lambda\tau. Abs\text{-}Set\text{-}0 \ll Some\ 'OclAny\ 'ran\ (heap\ (snd\ \tau)) \gg)$
apply(rule ext, simp add: OclAllInstances-at-post-def)
apply(subgoal-tac (OclAny 'ran (heap (snd τ)) - {None}) = (OclAny 'ran (heap (snd τ))),
simp)
apply(simp add: image-def)
apply(rule equalityI)
apply(rule subsetI)
apply (metis (lifting) Diff-iff)
apply(rule subsetI)
apply(simp)
apply(drule bexE) **prefer** 2 **apply** assumption
apply(case-tac x, metis OclAsType_{OclAny}- \mathcal{A} -some)
apply(blast)
done

lemma $OclAny.allInstances@pre() =$
 $(\lambda\tau. Abs\text{-}Set\text{-}0 \ll Some\ 'OclAny\ 'ran\ (heap\ (fst\ \tau)) \gg)$
apply(rule ext, simp add: OclAllInstances-at-pre-def)
apply(subgoal-tac (OclAny 'ran (heap (fst τ)) - {None}) = (OclAny 'ran (heap (fst τ))),
simp)
apply(simp add: image-def)
apply(rule equalityI)
apply(rule subsetI)
apply (metis (lifting) Diff-iff)
apply(rule subsetI)
apply(simp)
apply(drule bexE) **prefer** 2 **apply** assumption
apply(case-tac x, metis OclAsType_{OclAny}- \mathcal{A} -some)
apply(blast)
done

8.6.1. IsTypeOf

lemma $OclAny\text{-}allInstances\text{-}IsTypeOf_{OclAny}1:$

$\exists\tau. (\tau \models (OclAny.allInstances()) \rightarrow_{forAll(X|X.oclIsTypeOf(OclAny))})$
apply(rule-tac $x = \tau_0$ in exI, simp add: $\tau_0\text{-}def\ OclValid\text{-}def$)
apply(simp only: OclForall-def OclAllInstances-defined[simplified OclValid-def] refl if-True)
apply(simp only: OclAllInstances-at-post-def OclAllInstances-def)
apply(subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)

by(simp add: OclIsTypeOf_{OclAny}-OclAny)

lemma OclAny-allInstances-IsTypeOf_{OclAny}2:

$\exists \tau. (\tau \models \text{not } (\text{OclAny} . \text{allInstances}() \rightarrow \text{forAll}(X | X . \text{oclIsTypeOf}(\text{OclAny}))))$

proof – **fix** oid a **show** ?thesis

apply(rule-tac x = (fst τ_0 , ($\text{heap} = \text{empty}(oid \mapsto \text{in}_{\text{OclAny}}(\text{mk}_{\text{OclAny}} \text{oid } [a])),$
 $\text{assocs}_2 = \text{empty}, \text{assocs}_3 = \text{empty}))$ in exI, simp add: OclValid-def)

apply(simp only: OclForall-def OclAllInstances-defined[simplified OclValid-def] refl if-True)

apply(simp only: OclAllInstances-at-post-def OclAllInstances-def OclAsType_{OclAny}- \mathcal{A} -def)

apply(subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)

by(simp add: OclIsTypeOf_{OclAny}-OclAny OclNot-def OclAny-def)

qed

lemma Person-allInstances-IsTypeOf_{Person}:

$\tau \models (\text{Person} . \text{allInstances}() \rightarrow \text{forAll}(X | X . \text{oclIsTypeOf}(\text{Person})))$

apply(simp add: OclValid-def)

apply(simp only: OclForall-def OclAllInstances-defined[simplified OclValid-def] refl if-True)

apply(simp only: OclAllInstances-at-post-def OclAllInstances-def)

apply(subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)

by(simp add: OclIsTypeOf_{Person}-Person)

8.6.2. IsKindOf

lemma OclAny-allInstances-IsKindOf_{OclAny}:

$\tau \models (\text{OclAny} . \text{allInstances}() \rightarrow \text{forAll}(X | X . \text{oclIsKindOf}(\text{OclAny})))$

apply(simp add: OclValid-def)

apply(simp only: OclForall-def OclAllInstances-defined[simplified OclValid-def] refl if-True)

apply(simp only: OclAllInstances-at-post-def OclAllInstances-def)

apply(subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)

by(simp add: OclIsKindOf_{OclAny}-OclAny)

lemma Person-allInstances-IsKindOf_{OclAny}:

$\tau \models (\text{Person} . \text{allInstances}() \rightarrow \text{forAll}(X | X . \text{oclIsKindOf}(\text{OclAny})))$

apply(simp add: OclValid-def)

apply(simp only: OclForall-def OclAllInstances-defined[simplified OclValid-def] refl if-True)

apply(simp only: OclAllInstances-at-post-def OclAllInstances-def)

apply(subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)

by(simp add: OclIsKindOf_{OclAny}-Person)

lemma Person-allInstances-IsKindOf_{Person}:

$\tau \models (\text{Person} . \text{allInstances}() \rightarrow \text{forAll}(X | X . \text{oclIsKindOf}(\text{Person})))$

apply(simp add: OclValid-def)

apply(simp only: OclForall-def OclAllInstances-defined[simplified OclValid-def] refl if-True)

apply(simp only: OclAllInstances-at-post-def OclAllInstances-def)

apply(subst (1 2 3) Abs-Set-0-inverse, simp add: bot-option-def)

by(simp add: OclIsKindOf_{Person}-Person)

8.7. dot boss, dot salary

Should be generated entirely from a class-diagram.

8.7.1. Definition

definition $eval_extract\ X\ f = (\lambda\ \tau. \text{case } X\ \tau\ \text{of}$
 $\quad \perp \Rightarrow \text{invalid } \tau\ (*\ \text{exception propagation } *)$
 $\quad | \ \lfloor \perp \rfloor \Rightarrow \text{invalid } \tau\ (*\ \text{dereferencing null pointer } *)$
 $\quad | \ \lfloor \lfloor obj \rfloor \rfloor \Rightarrow f\ (oid\text{-of } obj)\ \tau)$

definition $deref_oid_{Person}\ fst_snd\ f\ oid = (\lambda\tau. \text{case } (heap\ (fst_snd\ \tau))\ oid\ \text{of}$
 $\quad | \ in_{Person}\ obj \rfloor \Rightarrow f\ obj\ \tau$
 $\quad | \ - \Rightarrow \text{invalid } \tau)$

definition $deref_oid_{OclAny}\ fst_snd\ f\ oid = (\lambda\tau. \text{case } (heap\ (fst_snd\ \tau))\ oid\ \text{of}$
 $\quad | \ in_{OclAny}\ obj \rfloor \Rightarrow f\ obj\ \tau$
 $\quad | \ - \Rightarrow \text{invalid } \tau)$

pointer undefined in state or not referencing a type conform object representation

definition $select_{OclAny}\ \mathcal{ANY}\ f = (\lambda\ X. \text{case } X\ \text{of}$
 $\quad (mk_{OclAny}\ -\ \perp) \Rightarrow \text{null}$
 $\quad | \ (mk_{OclAny}\ -\ \lfloor any \rfloor) \Rightarrow f\ (\lambda x\ -. \ \lfloor \lfloor x \rfloor \rfloor)\ any)$

definition $select_{Person}\ \mathcal{BOSS}\ f = (\lambda\ X. \text{case } X\ \text{of}$
 $\quad (mk_{Person}\ -\ \perp) \Rightarrow \text{null}\ (*\ \text{object contains null pointer } *)$
 $\quad | \ (mk_{Person}\ -\ \lfloor boss \rfloor) \Rightarrow f\ (\lambda x\ -. \ \lfloor \lfloor x \rfloor \rfloor)\ boss)$

definition $select_{Person}\ \mathcal{SALARY}\ f = (\lambda\ X. \text{case } X\ \text{of}$
 $\quad (mk_{Person}\ -\ \perp\ -) \Rightarrow \text{null}$
 $\quad | \ (mk_{Person}\ -\ \lfloor salary \rfloor\ -) \Rightarrow f\ (\lambda x\ -. \ \lfloor \lfloor x \rfloor \rfloor)\ salary)$

definition $in_pre_state = fst$

definition $in_post_state = snd$

definition $reconst_basetype = (\lambda\ convert\ x. \text{convert } x)$

definition $dot_{OclAny}\ \mathcal{ANY} :: OclAny \Rightarrow -\ ((1(-).any)\ 50)$
where $(X).any = eval_extract\ X$
 $\quad (deref_oid_{OclAny}\ in_post_state$
 $\quad (select_{OclAny}\ \mathcal{ANY}$
 $\quad reconst_basetype))$

definition $\text{dot}_{\text{Person}}\text{BOSS} :: \text{Person} \Rightarrow \text{Person} \ ((1(-).\text{boss})\ 50)$

where $(X).\text{boss} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{Person}} \text{ in-post-state}$
 $(\text{select}_{\text{Person}}\text{BOSS}$
 $(\text{deref-oid}_{\text{Person}} \text{ in-post-state})))$

definition $\text{dot}_{\text{Person}}\text{SALARY} :: \text{Person} \Rightarrow \text{Integer} \ ((1(-).\text{salary})\ 50)$

where $(X).\text{salary} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{Person}} \text{ in-post-state}$
 $(\text{select}_{\text{Person}}\text{SALARY}$
 $\text{reconst-basetype}))$

definition $\text{dot}_{\text{OclAny}}\text{ANY-at-pre} :: \text{OclAny} \Rightarrow - \ ((1(-).\text{any@pre})\ 50)$

where $(X).\text{any@pre} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{OclAny}} \text{ in-pre-state}$
 $(\text{select}_{\text{OclAny}}\text{ANY}$
 $\text{reconst-basetype}))$

definition $\text{dot}_{\text{Person}}\text{BOSS-at-pre} :: \text{Person} \Rightarrow \text{Person} \ ((1(-).\text{boss@pre})\ 50)$

where $(X).\text{boss@pre} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{Person}} \text{ in-pre-state}$
 $(\text{select}_{\text{Person}}\text{BOSS}$
 $(\text{deref-oid}_{\text{Person}} \text{ in-pre-state})))$

definition $\text{dot}_{\text{Person}}\text{SALARY-at-pre} :: \text{Person} \Rightarrow \text{Integer} \ ((1(-).\text{salary@pre})\ 50)$

where $(X).\text{salary@pre} = \text{eval-extract } X$
 $(\text{deref-oid}_{\text{Person}} \text{ in-pre-state}$
 $(\text{select}_{\text{Person}}\text{SALARY}$
 $\text{reconst-basetype}))$

lemmas $[\text{simp}] =$

$\text{dot}_{\text{OclAny}}\text{ANY-def}$
 $\text{dot}_{\text{Person}}\text{BOSS-def}$
 $\text{dot}_{\text{Person}}\text{SALARY-def}$
 $\text{dot}_{\text{OclAny}}\text{ANY-at-pre-def}$
 $\text{dot}_{\text{Person}}\text{BOSS-at-pre-def}$
 $\text{dot}_{\text{Person}}\text{SALARY-at-pre-def}$

8.7.2. Context Passing

lemma $\text{cp-dot}_{\text{OclAny}}\text{ANY}: ((X).\text{any})\ \tau = ((\lambda-. X\ \tau).\text{any})\ \tau \text{ by } (\text{simp add: eval-extract-def})$

lemma $\text{cp-dot}_{\text{Person}}\text{BOSS}: ((X).\text{boss})\ \tau = ((\lambda-. X\ \tau).\text{boss})\ \tau \text{ by } (\text{simp add: eval-extract-def})$

lemma $\text{cp-dot}_{\text{Person}}\text{SALARY}: ((X).\text{salary})\ \tau = ((\lambda-. X\ \tau).\text{salary})\ \tau \text{ by } (\text{simp add: eval-extract-def})$

lemma $\text{cp-dot}_{\text{OclAny}}\text{ANY-at-pre}: ((X).\text{any@pre})\ \tau = ((\lambda-. X\ \tau).\text{any@pre})\ \tau \text{ by } (\text{simp add: eval-extract-def})$

lemma $\text{cp-dot}_{\text{Person}}\text{BOSS-at-pre}: ((X).\text{boss@pre})\ \tau = ((\lambda-. X\ \tau).\text{boss@pre})\ \tau \text{ by } (\text{simp add: eval-extract-def})$

lemma $cp\text{-}dot_{Person}SALARY\text{-}at\text{-}pre$: $((X).salary@pre) \tau = ((\lambda\text{-}. X \tau).salary@pre) \tau$ **by** ($simp$ add : $eval\text{-}extract\text{-}def$)

lemmas $cp\text{-}dot_{OclAny}ANY\text{-}I$ [$simp$, $intro!$]=
 $cp\text{-}dot_{OclAny}ANY[THEN\ allI[THEN\ allI],$
 $of\ \lambda\ X\ \text{-}. X\ \lambda\ \text{-}\ \tau.\ \tau,\ THEN\ cpI1]$

lemmas $cp\text{-}dot_{OclAny}ANY\text{-}at\text{-}pre\text{-}I$ [$simp$, $intro!$]=
 $cp\text{-}dot_{OclAny}ANY\text{-}at\text{-}pre[THEN\ allI[THEN\ allI],$
 $of\ \lambda\ X\ \text{-}. X\ \lambda\ \text{-}\ \tau.\ \tau,\ THEN\ cpI1]$

lemmas $cp\text{-}dot_{Person}BOSS\text{-}I$ [$simp$, $intro!$]=
 $cp\text{-}dot_{Person}BOSS[THEN\ allI[THEN\ allI],$
 $of\ \lambda\ X\ \text{-}. X\ \lambda\ \text{-}\ \tau.\ \tau,\ THEN\ cpI1]$

lemmas $cp\text{-}dot_{Person}BOSS\text{-}at\text{-}pre\text{-}I$ [$simp$, $intro!$]=
 $cp\text{-}dot_{Person}BOSS\text{-}at\text{-}pre[THEN\ allI[THEN\ allI],$
 $of\ \lambda\ X\ \text{-}. X\ \lambda\ \text{-}\ \tau.\ \tau,\ THEN\ cpI1]$

lemmas $cp\text{-}dot_{Person}SALARY\text{-}I$ [$simp$, $intro!$]=
 $cp\text{-}dot_{Person}SALARY[THEN\ allI[THEN\ allI],$
 $of\ \lambda\ X\ \text{-}. X\ \lambda\ \text{-}\ \tau.\ \tau,\ THEN\ cpI1]$

lemmas $cp\text{-}dot_{Person}SALARY\text{-}at\text{-}pre\text{-}I$ [$simp$, $intro!$]=
 $cp\text{-}dot_{Person}SALARY\text{-}at\text{-}pre[THEN\ allI[THEN\ allI],$
 $of\ \lambda\ X\ \text{-}. X\ \lambda\ \text{-}\ \tau.\ \tau,\ THEN\ cpI1]$

8.7.3. Execution with invalid or null as argument

lemma $dot_{OclAny}ANY\text{-}nullstrict$ [$simp$]: $(null).any = invalid$
by ($rule\ ext$, $simp\ add$: $null\text{-}fun\text{-}def\ null\text{-}option\text{-}def\ bot\text{-}option\text{-}def\ null\text{-}def\ invalid\text{-}def\ eval\text{-}extract\text{-}def$)
lemma $dot_{OclAny}ANY\text{-}at\text{-}pre\text{-}nullstrict$ [$simp$]: $(null).any@pre = invalid$
by ($rule\ ext$, $simp\ add$: $null\text{-}fun\text{-}def\ null\text{-}option\text{-}def\ bot\text{-}option\text{-}def\ null\text{-}def\ invalid\text{-}def\ eval\text{-}extract\text{-}def$)
lemma $dot_{OclAny}ANY\text{-}strict$ [$simp$]: $(invalid).any = invalid$
by ($rule\ ext$, $simp\ add$: $null\text{-}fun\text{-}def\ null\text{-}option\text{-}def\ bot\text{-}option\text{-}def\ null\text{-}def\ invalid\text{-}def\ eval\text{-}extract\text{-}def$)
lemma $dot_{OclAny}ANY\text{-}at\text{-}pre\text{-}strict$ [$simp$]: $(invalid).any@pre = invalid$
by ($rule\ ext$, $simp\ add$: $null\text{-}fun\text{-}def\ null\text{-}option\text{-}def\ bot\text{-}option\text{-}def\ null\text{-}def\ invalid\text{-}def\ eval\text{-}extract\text{-}def$)

lemma $dot_{Person}BOSS\text{-}nullstrict$ [$simp$]: $(null).boss = invalid$
by ($rule\ ext$, $simp\ add$: $null\text{-}fun\text{-}def\ null\text{-}option\text{-}def\ bot\text{-}option\text{-}def\ null\text{-}def\ invalid\text{-}def\ eval\text{-}extract\text{-}def$)
lemma $dot_{Person}BOSS\text{-}at\text{-}pre\text{-}nullstrict$ [$simp$]: $(null).boss@pre = invalid$
by ($rule\ ext$, $simp\ add$: $null\text{-}fun\text{-}def\ null\text{-}option\text{-}def\ bot\text{-}option\text{-}def\ null\text{-}def\ invalid\text{-}def\ eval\text{-}extract\text{-}def$)
lemma $dot_{Person}BOSS\text{-}strict$ [$simp$]: $(invalid).boss = invalid$
by ($rule\ ext$, $simp\ add$: $null\text{-}fun\text{-}def\ null\text{-}option\text{-}def\ bot\text{-}option\text{-}def\ null\text{-}def\ invalid\text{-}def\ eval\text{-}extract\text{-}def$)
lemma $dot_{Person}BOSS\text{-}at\text{-}pre\text{-}strict$ [$simp$]: $(invalid).boss@pre = invalid$
by ($rule\ ext$, $simp\ add$: $null\text{-}fun\text{-}def\ null\text{-}option\text{-}def\ bot\text{-}option\text{-}def\ null\text{-}def\ invalid\text{-}def\ eval\text{-}extract\text{-}def$)

lemma $dot_{Person}SALARY\text{-}nullstrict$ [$simp$]: $(null).salary = invalid$
by ($rule\ ext$, $simp\ add$: $null\text{-}fun\text{-}def\ null\text{-}option\text{-}def\ bot\text{-}option\text{-}def\ null\text{-}def\ invalid\text{-}def\ eval\text{-}extract\text{-}def$)
lemma $dot_{Person}SALARY\text{-}at\text{-}pre\text{-}nullstrict$ [$simp$]: $(null).salary@pre = invalid$


```

by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def eval-extract-def)
lemma dotPersonSALARY-strict [simp] : (invalid).salary = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def eval-extract-def)
lemma dotPersonSALARY-at-pre-strict [simp] : (invalid).salary@pre = invalid
by(rule ext, simp add: null-fun-def null-option-def bot-option-def null-def invalid-def eval-extract-def)

```

8.8. A little infra-structure on example states.

```

definition OclInt1000 (1000) where OclInt1000 = (λ . . [|1000|])
definition OclInt1200 (1200) where OclInt1200 = (λ . . [|1200|])
definition OclInt1300 (1300) where OclInt1300 = (λ . . [|1300|])
definition OclInt1800 (1800) where OclInt1800 = (λ . . [|1800|])
definition OclInt2600 (2600) where OclInt2600 = (λ . . [|2600|])
definition OclInt2900 (2900) where OclInt2900 = (λ . . [|2900|])
definition OclInt3200 (3200) where OclInt3200 = (λ . . [|3200|])
definition OclInt3500 (3500) where OclInt3500 = (λ . . [|3500|])

```

```

definition oid0 ≡ 0
definition oid1 ≡ 1
definition oid2 ≡ 2
definition oid3 ≡ 3
definition oid4 ≡ 4
definition oid5 ≡ 5
definition oid6 ≡ 6
definition oid7 ≡ 7
definition oid8 ≡ 8

```

```

definition person1 ≡ mkPerson oid0 [|1300|] [|oid1|]
definition person2 ≡ mkPerson oid1 [|1800|] [|oid1|]
definition person3 ≡ mkPerson oid2 None None
definition person4 ≡ mkPerson oid3 [|2900|] None
definition person5 ≡ mkPerson oid4 [|3500|] None
definition person6 ≡ mkPerson oid5 [|2500|] [|oid6|]
definition person7 ≡ mkOclAny oid6 [(|3200|], [|oid6|])
definition person8 ≡ mkOclAny oid7 None
definition person9 ≡ mkPerson oid8 [|0|] None

```

definition

```

σ1 ≡ () heap = empty(oid0 ↦ inPerson (mkPerson oid0 [|1000|] [|oid1|]))
                      (oid1 ↦ inPerson (mkPerson oid1 [|1200|] None))
                      (*oid2*)
                      (oid3 ↦ inPerson (mkPerson oid3 [|2600|] [|oid4|]))
                      (oid4 ↦ inPerson person5)
                      (oid5 ↦ inPerson (mkPerson oid5 [|2300|] [|oid3|]))
                      (*oid6*)
                      (*oid7*)
                      (oid8 ↦ inPerson person9),
assocs2 = empty,
assocs3 = empty ()

```

definition

$$\begin{aligned} \sigma_1' \equiv & \langle \text{heap} = \text{empty}(\text{oid0} \mapsto \text{in}_{\text{Person}} \text{person1}) \\ & (\text{oid1} \mapsto \text{in}_{\text{Person}} \text{person2}) \\ & (\text{oid2} \mapsto \text{in}_{\text{Person}} \text{person3}) \\ & (\text{oid3} \mapsto \text{in}_{\text{Person}} \text{person4}) \\ & (*\text{oid4}*) \\ & (\text{oid5} \mapsto \text{in}_{\text{Person}} \text{person6}) \\ & (\text{oid6} \mapsto \text{in}_{\text{OclAny}} \text{person7}) \\ & (\text{oid7} \mapsto \text{in}_{\text{OclAny}} \text{person8}) \\ & (\text{oid8} \mapsto \text{in}_{\text{Person}} \text{person9}), \\ & \text{assocs}_2 = \text{empty}, \\ & \text{assocs}_3 = \text{empty} \rangle \end{aligned}$$

definition $\sigma_0 \equiv \langle \text{heap} = \text{empty}, \text{assocs}_2 = \text{empty}, \text{assocs}_3 = \text{empty} \rangle$

lemma *basic- τ -wff*: $WFF(\sigma_1, \sigma_1')$

by(*auto simp*: $WFF\text{-def}$ $\sigma_1\text{-def}$ $\sigma_1'\text{-def}$
 $\text{oid0}\text{-def}$ $\text{oid1}\text{-def}$ $\text{oid2}\text{-def}$ $\text{oid3}\text{-def}$ $\text{oid4}\text{-def}$ $\text{oid5}\text{-def}$ $\text{oid6}\text{-def}$ $\text{oid7}\text{-def}$ $\text{oid8}\text{-def}$
 $\text{oid-of-}\mathcal{A}\text{-def}$ $\text{oid-of-type}_{\text{Person}}\text{-def}$ $\text{oid-of-type}_{\text{OclAny}}\text{-def}$
 $\text{person1}\text{-def}$ $\text{person2}\text{-def}$ $\text{person3}\text{-def}$ $\text{person4}\text{-def}$ $\text{person5}\text{-def}$ $\text{person6}\text{-def}$ $\text{person7}\text{-def}$
 $\text{person8}\text{-def}$ $\text{person9}\text{-def}$)

lemma [*simp, code-unfold*]: $\text{dom}(\text{heap } \sigma_1) = \{\text{oid0}, \text{oid1}, (*, \text{oid2}*)\text{oid3}, \text{oid4}, \text{oid5}(*, \text{oid6}, \text{oid7}*), \text{oid8}\}$
by(*auto simp*: $\sigma_1\text{-def}$)

lemma [*simp, code-unfold*]: $\text{dom}(\text{heap } \sigma_1') = \{\text{oid0}, \text{oid1}, \text{oid2}, \text{oid3}, (*, \text{oid4}*)\text{oid5}, \text{oid6}, \text{oid7}, \text{oid8}\}$
by(*auto simp*: $\sigma_1'\text{-def}$)

definition $X_{\text{Person}1} :: \text{Person} \equiv \lambda - . \llbracket \text{person1} \rrbracket$

definition $X_{\text{Person}2} :: \text{Person} \equiv \lambda - . \llbracket \text{person2} \rrbracket$

definition $X_{\text{Person}3} :: \text{Person} \equiv \lambda - . \llbracket \text{person3} \rrbracket$

definition $X_{\text{Person}4} :: \text{Person} \equiv \lambda - . \llbracket \text{person4} \rrbracket$

definition $X_{\text{Person}5} :: \text{Person} \equiv \lambda - . \llbracket \text{person5} \rrbracket$

definition $X_{\text{Person}6} :: \text{Person} \equiv \lambda - . \llbracket \text{person6} \rrbracket$

definition $X_{\text{Person}7} :: \text{OclAny} \equiv \lambda - . \llbracket \text{person7} \rrbracket$

definition $X_{\text{Person}8} :: \text{OclAny} \equiv \lambda - . \llbracket \text{person8} \rrbracket$

definition $X_{\text{Person}9} :: \text{Person} \equiv \lambda - . \llbracket \text{person9} \rrbracket$

lemma [*code-unfold*]: $((x :: \text{Person}) \doteq y) = \text{StrictRefEq}_{\text{Object}} x y$ **by**(*simp only*: $\text{StrictRefEq}_{\text{Object-Person}}$)

lemma [*code-unfold*]: $((x :: \text{OclAny}) \doteq y) = \text{StrictRefEq}_{\text{Object}} x y$ **by**(*simp only*: $\text{StrictRefEq}_{\text{Object-OclAny}}$)

lemmas [*code-unfold, simp*] =

$\text{OclAsType}_{\text{OclAny-OclAny}}$

$\text{OclAsType}_{\text{OclAny-Person}}$

$\text{OclAsType}_{\text{Person-OclAny}}$

$\text{OclAsType}_{\text{Person-Person}}$

OclIsTypeOf_{OclAny}-OclAny
OclIsTypeOf_{OclAny}-Person
OclIsTypeOf_{Person}-OclAny
OclIsTypeOf_{Person}-Person

OclIsKindOf_{OclAny}-OclAny
OclIsKindOf_{OclAny}-Person
OclIsKindOf_{Person}-OclAny
OclIsKindOf_{Person}-Person

value $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .salary <> \mathbf{1000})$
value $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .salary \doteq \mathbf{1300})$
value $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .salary@pre \doteq \mathbf{1000})$
value $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .salary@pre <> \mathbf{1300})$
value $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .boss <> X_{Person1})$
value $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .boss .salary \doteq \mathbf{1800})$
value $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .boss .boss <> X_{Person1})$
value $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .boss .boss \doteq X_{Person2})$
value $(\sigma_1, \sigma_1') \models (X_{Person1} .boss@pre .salary \doteq \mathbf{1800})$
value $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .boss@pre .salary@pre \doteq \mathbf{1200})$
value $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .boss@pre .salary@pre <> \mathbf{1800})$
value $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .boss@pre \doteq X_{Person2})$
value $(\sigma_1, \sigma_1') \models (X_{Person1} .boss@pre .boss \doteq X_{Person2})$
value $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .boss@pre .boss@pre \doteq null)$
value $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models not(v(X_{Person1} .boss@pre .boss@pre .boss@pre))$

lemma $(\sigma_1, \sigma_1') \models (X_{Person1} .oclIsMaintained())$
by(*simp add: OclValid-def OclIsMaintained-def*
 σ_1 -def σ_1' -def
 *$X_{Person1}$ -def *person1*-def*
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def
oid-of-option-def oid-of-type_{Person}-def)

lemma $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models ((X_{Person1} .oclAsType(OclAny) .oclAsType(Person)) \doteq X_{Person1})$
by(*rule up-down-cast-Person-OclAny-Person', simp add: $X_{Person1}$ -def*)
value $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person1} .oclIsTypeOf(Person))$
value $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models not(X_{Person1} .oclIsTypeOf(OclAny))$
value $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person1} .oclIsKindOf(Person))$
value $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person1} .oclIsKindOf(OclAny))$
value $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models not(X_{Person1} .oclAsType(OclAny) .oclIsTypeOf(OclAny))$

value $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person2} .salary \doteq \mathbf{1800})$
value $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person2} .salary@pre \doteq \mathbf{1200})$
value $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person2} .boss \doteq X_{Person2})$
value $(\sigma_1, \sigma_1') \models (X_{Person2} .boss .salary@pre \doteq \mathbf{1200})$

value $(\sigma_1, \sigma_1') \models (X_{Person2}.boss.boss@pre \doteq null)$
value $\wedge_{s_{post}.} (\sigma_1, s_{post}) \models (X_{Person2}.boss@pre \doteq null)$
value $\wedge_{s_{post}.} (\sigma_1, s_{post}) \models (X_{Person2}.boss@pre <> X_{Person2})$
value $(\sigma_1, \sigma_1') \models (X_{Person2}.boss@pre <> (X_{Person2}.boss))$
value $\wedge_{s_{post}.} (\sigma_1, s_{post}) \models not(v(X_{Person2}.boss@pre.boss))$
value $\wedge_{s_{post}.} (\sigma_1, s_{post}) \models not(v(X_{Person2}.boss@pre.salary@pre))$
lemma $(\sigma_1, \sigma_1') \models (X_{Person2}.oclIsMaintained())$
by(simp add: OclValid-def OclIsMaintained-def
 σ_1 -def σ_1' -def
 $X_{Person2}$ -def person2-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def
oid-of-option-def oid-of-type_{Person}-def)

value $\wedge_{s_{pre}.} (s_{pre}, \sigma_1') \models (X_{Person3}.salary \doteq null)$
value $\wedge_{s_{post}.} (\sigma_1, s_{post}) \models not(v(X_{Person3}.salary@pre))$
value $\wedge_{s_{post}.} (\sigma_1, s_{post}) \models (X_{Person3}.salary@pre <> null)$
value $\wedge_{s_{pre}.} (s_{pre}, \sigma_1') \models (X_{Person3}.boss \doteq null)$
value $\wedge_{s_{pre}.} (s_{pre}, \sigma_1') \models not(v(X_{Person3}.boss.salary))$
value $\wedge_{s_{post}.} (\sigma_1, s_{post}) \models not(v(X_{Person3}.boss@pre))$
lemma $(\sigma_1, \sigma_1') \models (X_{Person3}.oclIsNew())$
by(simp add: OclValid-def OclIsNew-def
 σ_1 -def σ_1' -def
 $X_{Person3}$ -def person3-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid8-def
oid-of-option-def oid-of-type_{Person}-def)

value $\wedge_{s_{post}.} (\sigma_1, s_{post}) \models (X_{Person4}.boss@pre \doteq X_{Person5})$
value $(\sigma_1, \sigma_1') \models not(v(X_{Person4}.boss@pre.salary))$
value $\wedge_{s_{post}.} (\sigma_1, s_{post}) \models (X_{Person4}.boss@pre.salary@pre \doteq 3500)$
lemma $(\sigma_1, \sigma_1') \models (X_{Person4}.oclIsMaintained())$
by(simp add: OclValid-def OclIsMaintained-def
 σ_1 -def σ_1' -def
 $X_{Person4}$ -def person4-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def
oid-of-option-def oid-of-type_{Person}-def)

value $\wedge_{s_{pre}.} (s_{pre}, \sigma_1') \models not(v(X_{Person5}.salary))$
value $\wedge_{s_{post}.} (\sigma_1, s_{post}) \models (X_{Person5}.salary@pre \doteq 3500)$
value $\wedge_{s_{pre}.} (s_{pre}, \sigma_1') \models not(v(X_{Person5}.boss))$
lemma $(\sigma_1, \sigma_1') \models (X_{Person5}.oclIsOld())$
by(simp add: OclNot-def OclValid-def OclIsOld-def
 σ_1 -def σ_1' -def
 $X_{Person5}$ -def person5-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def
oid-of-option-def oid-of-type_{Person}-def)

value $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models \text{not}(v(X_{Person6} . \text{boss} . \text{salary}@pre))$
value $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person6} . \text{boss}@pre \doteq X_{Person4})$
value $(\sigma_1, \sigma_1') \models (X_{Person6} . \text{boss}@pre . \text{salary} \doteq \mathbf{2900})$
value $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person6} . \text{boss}@pre . \text{salary}@pre \doteq \mathbf{2600})$
value $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person6} . \text{boss}@pre . \text{boss}@pre \doteq X_{Person5})$
lemma $(\sigma_1, \sigma_1') \models (X_{Person6} . \text{oclIsMaintained}())$
by(simp add: OclValid-def OclIsMaintained-def
 σ_1 -def σ_1' -def
 $X_{Person6}$ -def person6-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def
oid-of-option-def oid-of-type_{Person}-def)

value $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models v(X_{Person7} . \text{oclAsType}(Person))$
value $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models \text{not}(v(X_{Person7} . \text{oclAsType}(Person) . \text{boss}@pre))$
lemma $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models ((X_{Person7} . \text{oclAsType}(Person) . \text{oclAsType}(OclAny) . \text{oclAsType}(Person)) \doteq (X_{Person7} . \text{oclAsType}(Person)))$
by(rule up-down-cast-Person-OclAny-Person', simp add: $X_{Person7}$ -def OclValid-def valid-def person7-def)
lemma $(\sigma_1, \sigma_1') \models (X_{Person7} . \text{oclIsNew}())$
by(simp add: OclValid-def OclIsNew-def
 σ_1 -def σ_1' -def
 $X_{Person7}$ -def person7-def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid8-def
oid-of-option-def oid-of-type_{OclAny}-def)

value $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person8} <> X_{Person7})$
value $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models \text{not}(v(X_{Person8} . \text{oclAsType}(Person)))$
value $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person8} . \text{oclIsTypeOf}(OclAny))$
value $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models \text{not}(X_{Person8} . \text{oclIsTypeOf}(Person))$
value $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models \text{not}(X_{Person8} . \text{oclIsKindOf}(Person))$
value $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person8} . \text{oclIsKindOf}(OclAny))$

lemma σ -modifiedonly: $(\sigma_1, \sigma_1') \models (\text{Set}\{ X_{Person1} . \text{oclAsType}(OclAny) , X_{Person2} . \text{oclAsType}(OclAny) , X_{Person3} . \text{oclAsType}(OclAny)* , X_{Person4} . \text{oclAsType}(OclAny) , X_{Person5} . \text{oclAsType}(OclAny)* , X_{Person6} . \text{oclAsType}(OclAny) , X_{Person7} . \text{oclAsType}(OclAny)* , X_{Person8} . \text{oclAsType}(OclAny)* \})$

```

      (*, XPerson9.oclAsType(OclAny*))} -> oclIsModifiedOnly())
apply(simp add: OclIsModifiedOnly-def OclValid-def
      oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def
      XPerson1-def XPerson2-def XPerson3-def XPerson4-def XPerson5-def XPerson6-def
      XPerson7-def XPerson8-def XPerson9-def
      person1-def person2-def person3-def person4-def person5-def person6-def person7-def
      person8-def person9-def
      image-def)
apply(simp add: including-rep-set mtSet-rep-set null-option-def bot-option-def)
apply(simp add: oid-of-option-def oid-of-typeOclAny-def, clarsimp)
apply(simp add:  $\sigma_1$ -def  $\sigma_1'$ -def
      oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)
done

lemma ( $\sigma_1, \sigma_1'$ )  $\models ((X_{Person9} \text{ @pre } (\lambda x. \lfloor OclAsType_{Person} \mathcal{A} \ x \rfloor)) \triangleq X_{Person9})$ 
by(simp add: OclSelf-at-pre-def  $\sigma_1$ -def oid-of-option-def oid-of-typePerson-def
      XPerson9-def person9-def oid8-def OclValid-def StrongEq-def OclAsTypePerson- $\mathcal{A}$ -def)

lemma ( $\sigma_1, \sigma_1'$ )  $\models ((X_{Person9} \text{ @post } (\lambda x. \lfloor OclAsType_{Person} \mathcal{A} \ x \rfloor)) \triangleq X_{Person9})$ 
by(simp add: OclSelf-at-post-def  $\sigma_1'$ -def oid-of-option-def oid-of-typePerson-def
      XPerson9-def person9-def oid8-def OclValid-def StrongEq-def OclAsTypePerson- $\mathcal{A}$ -def)

lemma ( $\sigma_1, \sigma_1'$ )  $\models (((X_{Person9}.oclAsType(OclAny)) \text{ @pre } (\lambda x. \lfloor OclAsType_{OclAny} \mathcal{A} \ x \rfloor)) \triangleq$ 
       $((X_{Person9}.oclAsType(OclAny)) \text{ @post } (\lambda x. \lfloor OclAsType_{OclAny} \mathcal{A} \ x \rfloor)))$ 
proof -

have including4 :  $\bigwedge a \ b \ c \ d \ \tau. \text{Set}\{\lambda \tau. \lfloor \lfloor a \rfloor \rfloor, \lambda \tau. \lfloor \lfloor b \rfloor \rfloor, \lambda \tau. \lfloor \lfloor c \rfloor \rfloor, \lambda \tau. \lfloor \lfloor d \rfloor \rfloor\} \ \tau = \text{Abs-Set-0} \ \lfloor \lfloor \{\lfloor \lfloor a \rfloor \rfloor, \lfloor \lfloor b \rfloor \rfloor, \lfloor \lfloor c \rfloor \rfloor, \lfloor \lfloor d \rfloor \rfloor\} \rfloor \rfloor$ 
apply(subst abs-rep-simp'[symmetric], simp)
by(simp add: including-rep-set mtSet-rep-set)

have excluding1 :  $\bigwedge S \ a \ b \ c \ d \ e \ \tau. (\lambda \tau. \text{Abs-Set-0} \ \lfloor \lfloor \{\lfloor \lfloor a \rfloor \rfloor, \lfloor \lfloor b \rfloor \rfloor, \lfloor \lfloor c \rfloor \rfloor, \lfloor \lfloor d \rfloor \rfloor\} \rfloor \rfloor) -> \text{excluding}(\lambda \tau. \lfloor \lfloor e \rfloor \rfloor) \ \tau = \text{Abs-Set-0} \ \lfloor \lfloor \{\lfloor \lfloor a \rfloor \rfloor, \lfloor \lfloor b \rfloor \rfloor, \lfloor \lfloor c \rfloor \rfloor, \lfloor \lfloor d \rfloor \rfloor\} - \{\lfloor \lfloor e \rfloor \rfloor\} \rfloor \rfloor$ 
apply(simp add: OclExcluding-def)
apply(simp add: defined-def OclValid-def bot-fun-def bot-Set-0-def null-fun-def null-Set-0-def
      false-def true-def)
apply(rule conjI)
apply(rule impI, subst (asm) Abs-Set-0-inject) apply( simp add: bot-option-def)+
apply(rule conjI)
apply(rule impI, subst (asm) Abs-Set-0-inject) apply( simp add: bot-option-def null-option-def)+
apply(subst Abs-Set-0-inverse, simp add: bot-option-def, simp)
done

show ?thesis
apply(rule framing[where  $X = \text{Set}\{ X_{Person1}.oclAsType(OclAny)$ 
      ,  $X_{Person2}.oclAsType(OclAny)$ 
      (*,  $X_{Person3}.oclAsType(OclAny)$ *)
      ,  $X_{Person4}.oclAsType(OclAny)$ 

```

```

      (*, XPerson5 .oclAsType(OclAny)*)
      , XPerson6 .oclAsType(OclAny)
      (*, XPerson7 .oclAsType(OclAny)*)
      (*, XPerson8 .oclAsType(OclAny)*)
      (*, XPerson9 .oclAsType(OclAny)*)}]])
  apply(cut-tac σ-modifiedonly)
  apply(simp only: OclValid-def
    XPerson1-def XPerson2-def XPerson3-def XPerson4-def XPerson5-def
    XPerson6-def XPerson7-def XPerson8-def XPerson9-def
    person1-def person2-def person3-def person4-def person5-def person6-def person7-def
    person8-def person9-def
    OclAsTypeOclAny-Person)
  apply(subst cp-OclIsModifiedOnly, subst cp-OclExcluding,
    subst (asm) cp-OclIsModifiedOnly, simp add: including4 excluding1)

  apply(simp add: OclValid-def defined-def
    XPerson9-def person9-def OclSelf-at-pre-def
    oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def
    oid-of-option-def oid-of-typeOclAny-def
    σ1-def OclAsTypeOclAny-ℳ-def bot-fun-def null-fun-def bot-option-def null-option-def)
  apply(simp only: XPerson1-def XPerson2-def XPerson3-def XPerson4-def XPerson5-def XPerson6-def
    XPerson7-def XPerson8-def XPerson9-def
    person1-def person2-def person3-def person4-def person5-def person6-def person7-def
    person8-def person9-def)
  apply(simp add: including-rep-set mtSet-rep-set
    oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)
  apply(simp add: StrictRefEqObject-def null-option-def bot-option-def oid-of-option-def oid-of-typeOclAny-def
    OclNot-def OclValid-def)
done
qed

```

lemma perm0 : $\bigwedge S \text{ oid7 oid6 person8 person7. oid7} \neq \text{oid6} \implies$
 $S (\text{oid7} \mapsto \text{person8}) (\text{oid6} \mapsto \text{person7}) = S (\text{oid6} \mapsto \text{person7}) (\text{oid7} \mapsto \text{person8})$
by (metis fun-upd-twist)

lemma perm-σ₁' : σ₁' = () heap = empty
 (oid8 \mapsto in_{Person} person9)
 (oid7 \mapsto in_{OclAny} person8)
 (oid6 \mapsto in_{OclAny} person7)
 (oid5 \mapsto in_{Person} person6)
 (*oid4*)
 (oid3 \mapsto in_{Person} person4)
 (oid2 \mapsto in_{Person} person3)
 (oid1 \mapsto in_{Person} person2)
 (oid0 \mapsto in_{Person} person1)
 , assoc_{s2} = assoc_{s2} σ₁'
 , assoc_{s3} = assoc_{s3} σ₁' ()

proof —

```

note  $p0 = perm0$ 
show  $?thesis$ 
  apply(simp add:  $\sigma_1'$ -def
        oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)
  apply(subst (1) p0, simp)
  apply(subst (2) p0, simp) apply(subst (1) p0, simp)
  apply(subst (3) p0, simp) apply(subst (2) p0, simp) apply(subst (1) p0, simp)
  apply(subst (4) p0, simp) apply(subst (3) p0, simp) apply(subst (2) p0, simp) apply(subst
(1) p0, simp)
  apply(subst (5) p0, simp) apply(subst (4) p0, simp) apply(subst (3) p0, simp) apply(subst
(2) p0, simp) apply(subst (1) p0, simp)
  apply(subst (6) p0, simp) apply(subst (5) p0, simp) apply(subst (4) p0, simp) apply(subst
(3) p0, simp) apply(subst (2) p0, simp) apply(subst (1) p0, simp)
  apply(subst (7) p0, simp) apply(subst (6) p0, simp) apply(subst (5) p0, simp) apply(subst
(4) p0, simp) apply(subst (3) p0, simp) apply(subst (2) p0, simp) apply(subst (1) p0, simp)
  by(simp)
qed

end

```


9. Part V (bis): OCL Contracts and an Example

```
theory
  Employee-DesignModel-OCLPart
imports
  Employee-DesignModel-UMLPart
begin
```

9.1. Standard State Infrastructure

These definitions should be generated — again — from the class diagram.

9.2. Invariant

These recursive predicates can be defined conservatively by greatest fix-point constructions - automatically. See HOL-OCL Book for details. For the purpose of this example, we state them as axioms here.

```
axiomatization inv-Person :: Person  $\Rightarrow$  Boolean
where A : ( $\tau \models (\delta \text{ self})$ )  $\longrightarrow$ 
  ( $\tau \models \text{inv-Person}(\text{self})$ ) =
    (( $\tau \models (\text{self}.\text{boss} \doteq \text{null})$ )  $\vee$ 
     ( $\tau \models (\text{self}.\text{boss} <> \text{null}) \wedge (\tau \models ((\text{self}.\text{salary}) \leq_{ocl} (\text{self}.\text{boss}.\text{salary}))) \wedge$ 
     ( $\tau \models (\text{inv-Person}(\text{self}.\text{boss}))$ )))
```

```
axiomatization inv-Person-at-pre :: Person  $\Rightarrow$  Boolean
where B : ( $\tau \models (\delta \text{ self})$ )  $\longrightarrow$ 
  ( $\tau \models \text{inv-Person-at-pre}(\text{self})$ ) =
    (( $\tau \models (\text{self}.\text{boss@pre} \doteq \text{null})$ )  $\vee$ 
     ( $\tau \models (\text{self}.\text{boss@pre} <> \text{null}) \wedge$ 
     ( $\tau \models (\text{self}.\text{boss@pre}.\text{salary@pre} \leq_{ocl} \text{self}.\text{salary@pre})$ )  $\wedge$ 
     ( $\tau \models (\text{inv-Person-at-pre}(\text{self}.\text{boss@pre}))$ )))
```

A very first attempt to characterize the axiomatization by an inductive definition - this can not be the last word since too weak (should be equality!)

```
coinductive inv :: Person  $\Rightarrow$  ( $\mathcal{A}$ )st  $\Rightarrow$  bool where
  ( $\tau \models (\delta \text{ self})$ )  $\implies$  (( $\tau \models (\text{self}.\text{boss} \doteq \text{null})$ )  $\vee$ 
    ( $\tau \models (\text{self}.\text{boss} <> \text{null}) \wedge (\tau \models (\text{self}.\text{boss}.\text{salary} \leq_{ocl} \text{self}.\text{salary})) \wedge$ 
    ( $(\text{inv}(\text{self}.\text{boss}))\tau$ )))
```

$$\implies (inv\ self\ \tau)$$

9.3. The contract of a recursive query :

The original specification of a recursive query :

```
context Person::contents():Set(Integer)
post:  result = if self.boss = null
           then Set{i}
           else self.boss.contents()->including(i)
           endif
```

consts *dot-contents* :: *Person* \Rightarrow *Set-Integer* ((1(-).contents'()) 50)

axiomatization where *dot-contents-def*:

```
( $\tau \models ((self).contents() \triangleq result)$ ) =
  (if ( $\delta\ self$ )  $\tau = true$   $\tau$ 
   then (( $\tau \models true$ )  $\wedge$ 
        ( $\tau \models (result \triangleq if\ (self.boss \doteq null)$ 
                     then (Set{self.salary})
                     else (self.boss.contents()->including(self.salary))
                     endif)))
   else  $\tau \models result \triangleq invalid$ )
```

consts *dot-contents-AT-pre* :: *Person* \Rightarrow *Set-Integer* ((1(-).contents@pre'()) 50)

axiomatization where *dot-contents-AT-pre-def*:

```
( $\tau \models (self).contents@pre() \triangleq result$ ) =
  (if ( $\delta\ self$ )  $\tau = true$   $\tau$ 
   then  $\tau \models true \wedge$ 
        ( $\tau \models (result \triangleq if\ (self).boss@pre \doteq null$ 
                     (* pre *)
                     (* post *)
                     then Set{(self).salary@pre}
                     else (self).boss@pre.contents@pre()->including(self.salary@pre)
                     endif)
   else  $\tau \models result \triangleq invalid$ )
```

Note that these @pre variants on methods are only available on queries, i.e. operations without side-effect.

9.4. The contract of a method.

The specification in high-level OCL input syntax reads as follows:

```
context Person::insert(x:Integer)
post: contents():Set(Integer)
contents() = contents@pre()->including(x)
```

consts *dot-insert* :: *Person* \Rightarrow *Integer* \Rightarrow *Void* ((1(-).insert'(-)) 50)

axiomatization where *dot-insert-def*:

($\tau \models ((self).insert(x) \triangleq result)$) =
(if ($\delta\ self$) $\tau = true$ $\tau \wedge (v\ x)\ \tau = true$ τ
 then $\tau \models true \wedge$
 $\tau \models ((self).contents() \triangleq (self).contents@pre()->including(x))$
 else $\tau \models ((self).insert(x) \triangleq invalid)$)

end

Part III.

Conclusion

10. Conclusion

10.1. 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]), then standard involution does not hold, i.e., `not(not(A)) = A` does not hold universally. Similarly, if `null and null` is `invalid`, then not even idempotence `X and X = X` holds. We strongly argue in favor of a lattice-like organization, where `null` represents “more information” than `invalid` and the logical operators are monotone with respect to this semantical “information ordering.”

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.

10.2. 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.