

Featherweight OCL

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

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

February 6, 2026

*SAP SE

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

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

[‡]IRT SystemX

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

Abstract

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

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

In this paper, we provide a formalization of the core of OCL in HOL. It provides denotational definitions, a logical calculus and operational rules that allow for the execution of OCL expressions by a mixture of term rewriting and code compilation. Moreover, we describe a coding-scheme for UML class models that were annotated by code-invariants and code contracts. An implementation of this coding-scheme has been undertaken: it consists of a kind of compiler that takes a UML class model and translates it into a family of definitions and derived theorems over them capturing the properties of constructors and selectors, tests and casts resulting from the class model. However, this compiler is *not* included in this document.

Our formalization reveals several inconsistencies and contradictions in the current version of the OCL standard. They reflect a challenge to define and implement OCL tools in a uniform manner. Overall, this document is intended to provide the basis for a machine-checked text “Annex A” of the OCL standard targeting at tool implementors.

Contents

I. Formal Semantics of OCL	13
0.1. Introduction	15
0.2. Background	18
0.2.1. A Running Example for UML/OCL	18
0.2.2. Formal Foundation	20
A Gentle Introduction to Isabelle	20
Higher-order Logic (HOL)	22
0.2.3. How this Annex A was Generated from Isabelle/HOL Theories	24
0.3. The Essence of UML-OCL Semantics	25
0.3.1. The Theory Organization	25
0.3.2. Denotational Semantics of Types	25
0.3.3. Denotational Semantics of Constants and Operations	26
0.3.4. Logical Layer	28
0.3.5. Algebraic Layer	29
0.3.6. Object-oriented Datatype Theories	31
A Denotational Space for Class-Models: Object Universes	32
Denotational Semantics of Accessors on Objects and Associations	33
Logic Properties of Class-Models	35
Algebraic Properties of the Class-Models	36
Other Operations on States	36
0.3.7. Data Invariants	37
0.3.8. Operation Contracts	37
1. Formalization I: OCL Types and Core Definitions	39
1.1. Preliminaries	39
1.1.1. Notations for the Option Type	39
1.1.2. Common Infrastructure for all OCL Types	39
1.1.3. Accommodation of Basic Types to the Abstract Interface	40
1.1.4. The Common Infrastructure of Object Types (Class Types) and States.	41
1.1.5. Common Infrastructure for all OCL Types (II): Valuations as OCL Types	42
1.1.6. The fundamental constants 'invalid' and 'null' in all OCL Types	42
1.2. Basic OCL Value Types	42
1.3. Some OCL Collection Types	43
1.3.1. The Construction of the Pair Type (Tuples)	44
1.3.2. The Construction of the Set Type	45
1.3.3. The Construction of the Bag Type	45
1.3.4. The Construction of the Sequence Type	46
1.3.5. Discussion: The Representation of UML/OCL Types in Featherweight OCL	47
2. Formalization II: OCL Terms and Library Operations	49
2.1. The Operations of the Boolean Type and the OCL Logic	49
2.1.1. Basic Constants	49
2.1.2. Validity and Definedness	49
2.1.3. The Equalities of OCL	51
Definition	52

	Fundamental Predicates on Strong Equality	52
2.1.4.	Logical Connectives and their Universal Properties	53
2.1.5.	A Standard Logical Calculus for OCL	58
	Global vs. Local Judgements	59
	Local Validity and Meta-logic	59
	Local Judgements and Strong Equality	63
2.1.6.	OCL's if then else endif	65
2.1.7.	Fundamental Predicates on Basic Types: Strict (Referential) Equality	66
2.1.8.	Laws to Establish Definedness (δ -closure)	66
2.1.9.	A Side-calculus for Constant Terms	67
2.2.	Property Profiles for OCL Operators via Isabelle Locales	70
2.2.1.	Property Profiles for Monadic Operators	70
2.2.2.	Property Profiles for Single	72
2.2.3.	Property Profiles for Binary Operators	72
2.2.4.	Fundamental Predicates on Basic Types: Strict (Referential) Equality	76
2.2.5.	Test Statements on Boolean Operations.	77
2.3.	Basic Type Void: Operations	77
2.3.1.	Fundamental Properties on Voids: Strict Equality	77
	Definition	77
2.3.2.	Basic Void Constants	78
2.3.3.	Validity and Definedness Properties	78
2.3.4.	Test Statements	79
2.4.	Basic Type Integer: Operations	79
2.4.1.	Fundamental Predicates on Integers: Strict Equality	79
2.4.2.	Basic Integer Constants	79
2.4.3.	Validity and Definedness Properties	79
2.4.4.	Arithmetical Operations	80
	Definition	80
	Basic Properties	81
	Execution with Invalid or Null or Zero as Argument	81
2.4.5.	Test Statements	82
2.5.	Basic Type Real: Operations	83
2.5.1.	Fundamental Predicates on Reals: Strict Equality	83
2.5.2.	Basic Real Constants	83
2.5.3.	Validity and Definedness Properties	84
2.5.4.	Arithmetical Operations	84
	Definition	84
	Basic Properties	85
	Execution with Invalid or Null or Zero as Argument	85
2.5.5.	Test Statements	86
2.6.	Basic Type String: Operations	87
2.6.1.	Fundamental Properties on Strings: Strict Equality	87
2.6.2.	Basic String Constants	87
2.6.3.	Validity and Definedness Properties	88
2.6.4.	String Operations	88
	Definition	88
	Basic Properties	88
2.6.5.	Test Statements	88
2.7.	Collection Type Pairs: Operations	89
2.7.1.	Semantic Properties of the Type Constructor	89
2.7.2.	Fundamental Properties of Strict Equality	90
2.7.3.	Standard Operations Definitions	90
	Definition: Pair Constructor	90

	Definition: First	90
	Definition: Second	90
2.7.4.	Logical Properties	91
2.7.5.	Algebraic Execution Properties	91
2.7.6.	Test Statements	91
2.8.	Collection Type Bag: Operations	92
2.8.1.	As a Motivation for the (infinite) Type Construction: Type-Extensions as Bags	92
2.8.2.	Basic Properties of the Bag Type	95
2.8.3.	Definition: Strict Equality	96
2.8.4.	Constants: mtBag	96
2.8.5.	Definition: Including	97
2.8.6.	Definition: Excluding	97
2.8.7.	Definition: Includes	97
2.8.8.	Definition: Excludes	98
2.8.9.	Definition: Size	98
2.8.10.	Definition: IsEmpty	98
2.8.11.	Definition: NotEmpty	98
2.8.12.	Definition: Any	98
2.8.13.	Definition: Forall	98
2.8.14.	Definition: Exists	99
2.8.15.	Definition: Iterate	99
2.8.16.	Definition: Select	99
2.8.17.	Definition: Reject	100
2.8.18.	Definition: IncludesAll	100
2.8.19.	Definition: ExcludesAll	100
2.8.20.	Definition: Union	100
2.8.21.	Definition: Intersection	101
2.8.22.	Definition: Count	101
2.8.23.	Definition (future operators)	101
2.8.24.	Logical Properties	101
2.8.25.	Execution Laws with Invalid or Null or Infinite Set as Argument	105
	Context Passing	106
	Const	108
2.8.26.	Test Statements	108
2.9.	Collection Type Set: Operations	109
2.9.1.	As a Motivation for the (infinite) Type Construction: Type-Extensions as Sets	109
2.9.2.	Basic Properties of the Set Type	111
2.9.3.	Definition: Strict Equality	113
2.9.4.	Constants: mtSet	113
2.9.5.	Definition: Including	113
2.9.6.	Definition: Excluding	114
2.9.7.	Definition: Includes	115
2.9.8.	Definition: Excludes	115
2.9.9.	Definition: Size	115
2.9.10.	Definition: IsEmpty	115
2.9.11.	Definition: NotEmpty	115
2.9.12.	Definition: Any	116
2.9.13.	Definition: Forall	116
2.9.14.	Definition: Exists	116
2.9.15.	Definition: Iterate	116
2.9.16.	Definition: Select	117
2.9.17.	Definition: Reject	117
2.9.18.	Definition: IncludesAll	117

2.9.19. Definition: ExcludesAll	117
2.9.20. Definition: Union	117
2.9.21. Definition: Intersection	118
2.9.22. Definition (future operators)	119
2.9.23. Logical Properties	119
2.9.24. Execution Laws with Invalid or Null or Infinite Set as Argument	122
Context Passing	124
Const	125
2.9.25. General Algebraic Execution Rules	126
Execution Rules on Including	126
Execution Rules on Excluding	128
Execution Rules on Includes	135
Execution Rules on Excludes	138
Execution Rules on Size	139
Execution Rules on IsEmpty	140
Execution Rules on NotEmpty	141
Execution Rules on Any	141
Execution Rules on Forall	142
Execution Rules on Exists	145
Execution Rules on Iterate	145
Execution Rules on Select	147
Execution Rules on Reject	151
Execution Rules Combining Previous Operators	151
2.9.26. Test Statements	161
2.10. Collection Type Sequence: Operations	162
2.10.1. Basic Properties of the Sequence Type	162
2.10.2. Definition: Strict Equality	162
2.10.3. Constants: mtSequence	163
2.10.4. Definition: Prepend	163
2.10.5. Definition: Including	164
2.10.6. Definition: Excluding	165
2.10.7. Definition: Append	165
2.10.8. Definition: Union	165
2.10.9. Definition: At	166
2.10.10. Definition: First	166
2.10.11. Definition: Last	166
2.10.12. Definition: Iterate	166
2.10.13. Definition: Forall	167
2.10.14. Definition: Exists	167
2.10.15. Definition: Collect	167
2.10.16. Definition: Select	167
2.10.17. Definition: Size	167
2.10.18. Definition: IsEmpty	167
2.10.19. Definition: NotEmpty	168
2.10.20. Definition: Any	168
2.10.21. Definition (future operators)	168
2.10.22. Logical Properties	168
2.10.23. Execution Laws with Invalid or Null as Argument	168
Context Passing	168
Const	169
2.10.24. General Algebraic Execution Rules	169
Execution Rules on Iterate	169
2.10.25. Test Statements	170

2.11. Miscellaneous Stuff	171
2.11.1. Definition: asBoolean	171
2.11.2. Definition: asInteger	171
2.11.3. Definition: asReal	171
2.11.4. Definition: asPair	171
2.11.5. Definition: asSet	172
2.11.6. Definition: asSequence	172
2.11.7. Definition: asBag	172
2.11.8. Collection Types	173
2.11.9. Test Statements	173
3. Formalization III: UML/OCL constructs: State Operations and Objects	175
3.1. Introduction: States over Typed Object Universes	175
3.1.1. Fundamental Properties on Objects: Core Referential Equality	175
Definition	175
Strictness and context passing	175
3.1.2. Logic and Algebraic Layer on Object	176
Validity and Definedness Properties	176
Symmetry	176
Behavior vs StrongEq	176
3.2. Operations on Object	177
3.2.1. Initial States (for testing and code generation)	177
3.2.2. OclAllInstances	177
OclAllInstances (@post)	182
OclAllInstances (@pre)	184
@post or @pre	185
3.2.3. OclIsNew, OclIsDeleted, OclIsMaintained, OclIsAbsent	186
3.2.4. OclIsModifiedOnly	187
Definition	187
Execution with Invalid or Null or Null Element as Argument	187
Context Passing	187
3.2.5. OclSelf	187
3.2.6. Framing Theorem	188
3.2.7. Miscellaneous	190
3.3. Accessors on Object	191
3.3.1. Definition	191
3.3.2. Validity and Definedness Properties	191
4. Example: The Employee Analysis Model	203
4.1. Introduction	203
4.1.1. Outlining the Example	203
4.2. Example Data-Universe and its Infrastructure	204
4.3. Instantiation of the Generic Strict Equality	205
4.4. OclAsType	205
4.4.1. Definition	205
4.4.2. Context Passing	206
4.4.3. Execution with Invalid or Null as Argument	207
4.5. OclIsTypeOf	207
4.5.1. Definition	207
4.5.2. Context Passing	208
4.5.3. Execution with Invalid or Null as Argument	209
4.5.4. Up Down Casting	209

4.6.	OclIsKindOf	210
4.6.1.	Definition	210
4.6.2.	Context Passing	211
4.6.3.	Execution with Invalid or Null as Argument	212
4.6.4.	Up Down Casting	212
4.7.	OclAllInstances	213
4.7.1.	OclIsTypeOf	213
4.7.2.	OclIsKindOf	214
4.8.	The Accessors (any, boss, salary)	215
4.8.1.	Definition (of the association Employee-Boss)	215
4.8.2.	Context Passing	218
4.8.3.	Execution with Invalid or Null as Argument	218
4.8.4.	Representation in States	219
4.9.	A Little Infra-structure on Example States	219
4.10.	OCL Part: Invariant	225
4.11.	OCL Part: The Contract of a Recursive Query	227
4.12.	OCL Part: The Contract of a User-defined Method	230
5.	Example: The Employee Design Model	233
5.1.	Introduction	233
5.1.1.	Outlining the Example	233
5.2.	Example Data-Universe and its Infrastructure	233
5.3.	Instantiation of the Generic Strict Equality	234
5.4.	OclAsType	235
5.4.1.	Definition	235
5.4.2.	Context Passing	236
5.4.3.	Execution with Invalid or Null as Argument	237
5.5.	OclIsTypeOf	237
5.5.1.	Definition	237
5.5.2.	Context Passing	238
5.5.3.	Execution with Invalid or Null as Argument	239
5.5.4.	Up Down Casting	239
5.6.	OclIsKindOf	240
5.6.1.	Definition	240
5.6.2.	Context Passing	241
5.6.3.	Execution with Invalid or Null as Argument	242
5.6.4.	Up Down Casting	242
5.7.	OclAllInstances	242
5.7.1.	OclIsTypeOf	243
5.7.2.	OclIsKindOf	244
5.8.	The Accessors (any, boss, salary)	245
5.8.1.	Definition	245
5.8.2.	Context Passing	247
5.8.3.	Execution with Invalid or Null as Argument	247
5.8.4.	Representation in States	248
5.9.	A Little Infra-structure on Example States	249
5.10.	OCL Part: Invariant	255
5.11.	OCL Part: The Contract of a Recursive Query	257

II. Conclusion	259
6. Conclusion	261
6.1. Lessons Learned and Contributions	261
6.2. Lessons Learned	262
6.3. Conclusion and Future Work	262
III. Appendix	269
A. The OCL And Featherweight OCL Syntax	271

Part I.

Formal Semantics of OCL

0.1. Introduction

The Unified Modeling Language (UML) [30, 31] is one of the few modeling languages that is widely used in industry. UML is defined in an open process by the Object Management Group (OMG), i. e., an industry consortium. While UML is mostly known as diagrammatic modeling language (e. g., visualizing class models), it also comprises a textual language, called Object Constraint Language (OCL) [32]. OCL is a textual annotation language, originally conceived as a three-valued logic, that turns substantial parts of UML into a formal language. Unfortunately the semantics of this specification language, captured in the “Annex A” (originally, based on the work of Richters [33]) of the OCL standard leads to different interpretations of corner cases. Many of these corner cases had been subject to formal analysis since more than nearly fifteen years (see, e. g., [5, 10, 18, 22, 26]).

At its origins [28, 33], OCL was conceived as a strict semantics for undefinedness (e. g., denoted by the element `invalid`¹), with the exception of the logical connectives of type `Boolean` that constitute a three-valued propositional logic. At its core, OCL comprises four layers:

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

Motivated by the need for aligning OCL closer with UML, recent versions of the OCL standard [29, 32] added a second exception element. While the first exception element `invalid` has a strict semantics, `null` has a non strict semantic interpretation. Unfortunately, this extension results in several inconsistencies and contradictions. These problems are reflected in difficulties to define interpreters, code-generators, specification animators or theorem provers for OCL in a uniform manner and resulting incompatibilities of various tools.

For the OCL community, the semantics of `invalid` and `null` as well as many related issues resulted in the challenge to define a consistent version of the OCL standard that is well aligned with the recent developments of the UML. A syntactical and semantical consistent standard requires a major revision of both the informal and formal parts of the standard. To discuss the future directions of the standard, several OCL experts met in November 2013 in Aachen to discuss possible mid-term improvements of OCL, strategies of standardization of OCL within the OMG, and a vision for possible long-term developments of the language [14]. During this meeting, a Request for Proposals (RFP) for OCL 2.5 was finalized and meanwhile proposed. In particular, this RFP requires that the future OCL 2.5 standard document shall be generated from a machine-checked source. This will ensure

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

In this document, we present a formalization using Isabelle/HOL [27] of a core language of OCL. The semantic theory, based on a “shallow embedding”, is called *Featherweight OCL*, since it focuses on a formal treatment of the key-elements of the language (rather than a full treatment of all operators and thus, a “complete” implementation). In contrast to full OCL, it comprises just the logic captured in `Boolean`, the basic data types `Integer` `Real` and `String`, the collection types `Set`, `Sequence` and `Bag`, as well as the generic construction principle of class models, which is instantiated and demonstrated for two examples (an automated support for this type-safe construction is out of the scope of Featherweight

¹In earlier versions of the OCL standard, this element was called `oclUndefined`.

OCL). This formal semantics definition is intended to be a proposal for the standardization process of OCL 2.5, which should ultimately replace parts of the mandatory part of the standard document [32] as well as replace completely its informative “Annex A.”

The semantic definitions are in large parts executable, in some parts only provable, namely the essence of Set-and Bag-constructions. The first goal of its construction is *consistency*, i. e., it should be possible to apply logical rules and/or evaluation rules for OCL in an arbitrary manner always yielding the same result. Moreover, except in pathological cases, this result should be unambiguously defined, i. e., represent a value.

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

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

and:

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

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

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

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

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

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

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

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

Here is a feature-list of Featherweight OCL:

- it specifies key built-in types such as `Boolean`, `Void`, `Integer`, `Real` and `String` as well as generic types such as `Pair(T,T')`, `Sequence(T)` and `Set(T)`.
- it defines the semantics of the operations of these types in *denotational form*—see explanation below—, and thus in an unambiguous (and in Isabelle/HOL executable or animatable) way.
- it develops the *theory* of these definitions, i. e., the collection of lemmas and theorems that can be proven from these definitions.
- all types in Featherweight OCL contain the elements `null` and `invalid`; since this extends to `Boolean` type, this results in a four-valued logic. Consequently, Featherweight OCL contains the derivation of the *logic* of OCL.
- collection types may contain `null` (so `Set{null}` is a defined set) but not `invalid` (`Set{invalid}` is just `invalid`).
- Wrt. to the static types, Featherweight OCL is a strongly typed language in the Hindley-Milner tradition. We assume that a pre-process for full OCL eliminates all implicit conversions due to subtyping by introducing explicit casts (e. g., `oclAsType(Class)`).⁴
- Featherweight OCL types may be arbitrarily nested. For example, the expression `Set{Set{1,2}} = Set{Set{2,1}}` is legal and true.
- Featherweight OCL types may be higher-order nested. For example, the expression `\<lambda> X. Set{X} = Set{Set{2,1}}` is legal. Higher-order pattern-matching can be easily extended following the principles in the HOL library, which can be applied also to Featherweight OCL types.
- All objects types are represented in an object universe⁵. The universe construction also gives semantics to type casts, dynamic type tests, as well as functions such as `allInstances()`, or `oclIsNew()`. The object universe construction is conceptually described and demonstrated at an example.
- As part of the OCL logic, Featherweight OCL develops the theory of equality in UML/OCL. This includes the standard equality, which is a computable strict equality using the object references for comparison, and the not necessarily computable logical equality, which expresses the Leibniz principle that ‘equals may be replaced by equals’ in OCL terms.
- Technically, Featherweight OCL is a *semantic embedding* into a powerful semantic meta-language and environment, namely Isabelle/HOL [27]. It is a so-called *shallow embedding* in HOL; this means that types in OCL were mapped one-to-one to types in Isabelle/HOL. Ill-typed OCL specifications can therefore not be represented in Featherweight OCL and a type in Featherweight OCL contains exactly the values that are possible in OCL .

Context. This document stands in a more than fifteen years tradition of giving a formal semantics to the core of UML and its annotation language OCL, starting from Richters [33] and [18, 22, 26], leading to a number of formal, machine-checked versions, most notably HOL-OCL [4, 6, 7, 10] and more recent approaches [15]. All of them have in common the attempt to reconcile the conflicting demands of an industrially used specification language and its various stakeholders, the needs of OMG standardization process and the desire for sufficient logical precision for tool-implementors, in particular from the Formal Methods research community. To discuss the future directions of the standard, several OCL experts met in November 2013 in Aachen to discuss possible mid-term improvements of OCL, strategies of

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

⁵following the tradition of HOL-OCL [7]

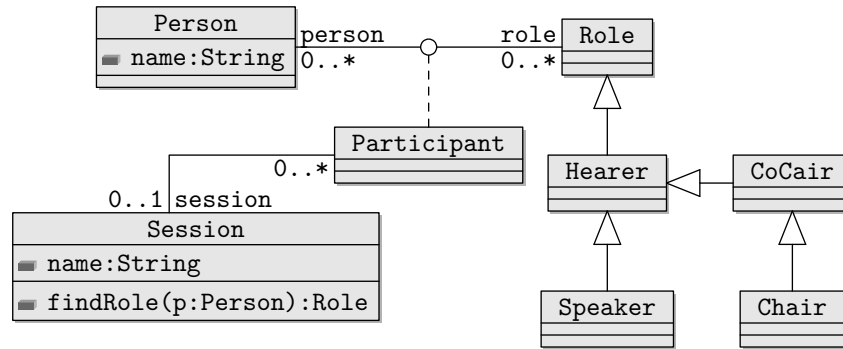


Figure 1.: A simple UML class model representing a conference system for organizing conference sessions: persons can participate, in different roles, in a session.

standardization of OCL within the OMG, and a vision for possible long-term developments of the language [14]. The participants agreed that future proposals for a formal semantics should be machine-check, to ensure the absence of syntax errors, the consistency of the formal semantics, as well as provide a suite of corner-cases relevant for OCL tool implementors.

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

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

0.2. Background

0.2.1. A Running Example for UML/OCL

The Unified Modeling Language (UML) [30, 31] comprises a variety of model types for describing static (e. g., class models, object models) and dynamic (e. g., state-machines, activity graphs) system properties. One of the more prominent model types of the UML is the *class model* (visualized as *class diagram*) for modeling the underlying data model of a system in an object-oriented manner. As a running example, we model a part of a conference management system. Such a system usually supports the conference organizing process, e. g., creating a conference Website, reviewing submissions, registering attendees, organizing the different sessions and tracks, and indexing and producing the resulting proceedings. In this example, we constrain ourselves to the process of organizing conference sessions; Figure 1 shows the class model. We model the hierarchy of roles of our system as a hierarchy of classes (e. g., *Hearer*, *Speaker*, or *Chair*) using an *inheritance* relation (also called *generalization*).

In particular, *inheritance* establishes a *subtyping* relationship, i.e., every **Speaker** (*subclass*) is also a **Hearer** (*superclass*).

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

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

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

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

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

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

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

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

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

```
pre: self.participates.person->includes(person)
```

is actually a syntactic abbreviation for a constraint referring to the previous state:

```
self.participates@pre.person@pre->includes(person).
```

Note, further, that conventions for full-OCCL permit the suppression of the `self`-parameter, following similar syntactic conventions in other object-oriented languages such as Java:

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

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

A key idea of defining the semantics of UML and extensions like SecureUML [11] is to translate the diagrammatic UML features into a combination of more elementary features of UML and OCL expressions [20]. For example, associations (i. e., relations on objects) can be implemented in specifications at the design level by aggregations, i. e., collection-valued class attributes together with OCL constraints expressing the multiplicity. Thus, having a semantics for a subset of UML and OCL is tantamount for the foundation of the entire method.

0.2.2. Formal Foundation

A Gentle Introduction to Isabelle

Isabelle [27] is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and natural deduction inference rules. Among other logics, Isabelle supports first-order logic, Zermelo-Fraenkel set theory and the instance for Church's higher-order logic (HOL).

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

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

where:

- C is the set of *constant symbols* like "fst" or "snd" as operators on pairs. Note that Isabelle's syntax engine supports mixfix-notation for terms: " $(_ \implies _) A B$ " or " $(_ + _) A B$ " can be parsed and printed as " $A \implies B$ " or " $A + B$ ", respectively.
- V is the set of *variable symbols* like " x ", " y ", " z ", ... Variables standing in the scope of a λ -operator were called *bound* variables, all others are *free* variables.
- " $\lambda V. T$ " is called λ -abstraction. For example, consider the identity function $\lambda x.x$. A λ -abstraction forms a scope for the variable V .
- $T T'$ is called an *application*.

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

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

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

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

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

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

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

Isabelle accepts also the notation $t :: \tau$ as type assertion in the term-language; $t :: \tau$ means "t is required to have type τ ". Note that typed terms *can* contain free variables; terms like $x + y = y + x$ reflecting common mathematical notation (and the convention that free variables are implicitly universally quantified) are possible and common in Isabelle theories.¹⁰

An environment providing Ξ, TC as well as a map from constant symbols C to types (built over these Ξ and TC) is called a *global context*; it provides a kind of signature, i. e., a mechanism to construct the syntactic material of a logical theory.

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

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

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

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

Isabelle supports forward- and backward reasoning on rules. For backward-reasoning, a *proof-state* can be initialized in a given global context and further transformed into others. For example, a proof of ϕ , using the Isar [36] language, will look as follows in Isabelle:

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

(0.5)

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

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

(0.6)

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

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

Subgoals and goals may be extracted from the proof state into theorems of the form $\llbracket \phi_1; \dots; \phi_n \rrbracket \Longrightarrow \phi$ at any time;

By extensions of global contexts with axioms and proofs of theorems, *theories* can be constructed step by step. Beyond the basic mechanisms to extend a global context by a type-constructor-, type-class-constant-definition or an axiom, Isabelle offers a number of *commands* that allow for more complex extensions of theories in a logically safe way (avoiding the use of axioms directly).

Higher-order Logic (HOL)

Higher-order logic (HOL) [1, 16] is a classical logic based on a simple type system. Isabelle/HOL is a theory extension of the basic Isabelle core-language with operators and the 7 axioms of HOL; together with large libraries this constitutes an implementation of HOL. Isabelle/HOL provides the usual logical connectives like $_ \wedge _$, $_ \rightarrow _$, $\neg _$ as well as the object-logical quantifiers $\forall x. Px$ and $\exists x. Px$; in contrast to first-order logic, quantifiers may range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centered around extensional equality $_ = _ :: \alpha \Rightarrow \alpha \Rightarrow \text{bool}$. Extensional equality means that two functions f and g are equal if and only if they are point-wise equal; this is captured by the rule: $(\bigwedge x. f\ x = g\ x) \Longrightarrow f = g$. HOL is more expressive than first-order logic, since, among many other things, induction schemes can be expressed inside the logic. For example, the standard induction rule on natural numbers in HOL:

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

is just an ordinary rule in Isabelle which is in fact a proven theorem in the theory of natural numbers. This example exemplifies an important design principle of Isabelle: theorems and rules are technically the same, paving the way to *derived rules* and automated decision procedures based on them. This has the consequence that that these procedures are consequently sound by construction with respect to their logical aspects (they may be incomplete or failing, though).

On the other hand, Isabelle/HOL can also be viewed as a functional programming language like SML or Haskell. Isabelle/HOL definitions can usually be read just as another functional **programming** language; if not interested in proofs and the possibilities of a **specification** language providing powerful logical quantifiers or equivalent free variables, the reader can just ignore these aspects in theories.

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

For instance, the library includes the type constructor $\tau_{\perp} := \perp \mid _ _ : \alpha$ that assigns to each type τ a type τ_{\perp} *disjointly extended* by the exceptional element \perp . The function $\ulcorner _ \urcorner : \alpha_{\perp} \rightarrow \alpha$ is the inverse of $_ _$ (unspecified for \perp). Partial functions $\alpha \rightarrow \beta$ are defined as functions $\alpha \Rightarrow \beta_{\perp}$ supporting the usual concepts of domain ($\text{dom } _$) and range ($\text{ran } _$).

As another example of a conservative extension, typed sets were built in the Isabelle libraries conservatively on top of the kernel of HOL as functions to `bool`; consequently, the constant definitions for membership is as follows:¹¹

types	α set	$= \alpha \Rightarrow \text{bool}$	
definition	Collect	$:: (\alpha \Rightarrow \text{bool}) \Rightarrow \alpha$ set	— set comprehension
where	Collect S	$\equiv S$	(0.7)
definition	member	$:: \alpha \Rightarrow \alpha \Rightarrow \text{bool}$	— membership test
where	member $s\ S$	$\equiv S\ s$	

Isabelle's syntax engine is instructed to accept the notation $\{x \mid P\}$ for `Collect $\lambda x. P$` and the notation $s \in S$ for `member $s\ S$` . As can be inferred from the example, constant definitions are axioms that introduce a fresh constant symbol by some non-recursive expressions not containing free variables; this

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

type of axiom is logically safe since it works like an abbreviation. The syntactic side conditions of this axiom are mechanically checked. It is straightforward to express the usual operations on sets like $_ \cup _ , _ \cap _ :: \alpha \text{ set} \Rightarrow \alpha \text{ set} \Rightarrow \alpha \text{ set}$ as conservative extensions, too, while the rules of typed set theory were derived by proofs from these definitions.

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

$$\begin{aligned} \text{datatype } \text{option} &= \text{None} \mid \text{Some } \alpha \\ \text{datatype } \alpha \text{ list} &= \text{Nil} \mid \text{Cons } a \ l \end{aligned} \quad (0.8)$$

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

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

respectively

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

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

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

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

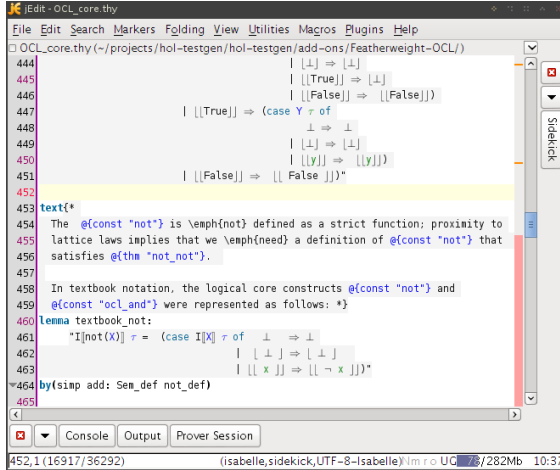
$$\begin{aligned} \text{fun } \text{ins} &:: [\alpha :: \text{linorder}, \alpha \text{ list}] \Rightarrow \alpha \text{ list} \\ \text{where } \text{ins } x \ [] &= [x] \\ \text{ins } x \ (y\#ys) &= \text{if } x < y \text{ then } x\#y\#ys \text{ else } y\#(\text{ins } x \ ys) \end{aligned} \quad (0.12)$$

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

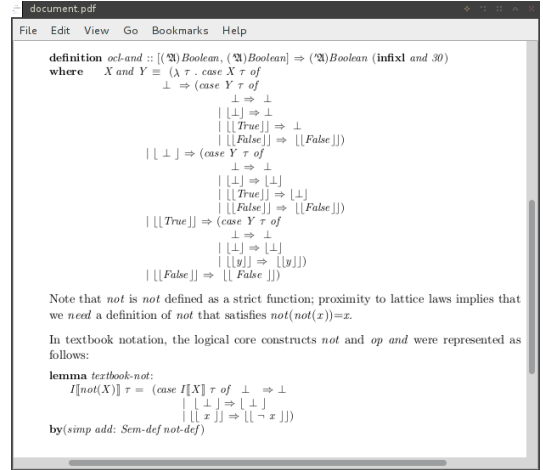
The internal (non-recursive) constant definition for these operations is quite involved; however, the logical compiler will finally derive all the equations in the statements above from this definition and make them available for automated simplification.

Thus, Isabelle/HOL also provides a large collection of theories like sets, lists, orderings, and various arithmetic theories which only contain rules derived from conservative definitions. This library constitutes a comfortable basis for defining the OCL library and language constructs.

In particular, Isabelle manages a set of *executable types and operators*, i. e., types and operators for which a compilation to SML, OCaml or Haskell is possible. Setups for arithmetic types such as `int` have been done; moreover any datatype and any recursive function were included in this executable set (providing that they only consist of executable operators). This forms the basis that many OCL terms can be executed directly. Using the value command, it is possible to compile many OCL ground expressions (no free variables) to code and to execute them; for example value "3 + 7" just answers with 10 in Isabelle's output window. This is even true for many expressions containing types which in themselves are not executable. For example, the `Set` type, which is defined in Featherweight OCL as the type of potentially infinite sets, is consequently not in itself executable; however, due to special setups of the code-generator, expressions like value "Set{1,2}" are, because the underlying constructors in this expression allow for automatically establishing that this set is finite and reducible to constructs that *are* in this special case executable.



(a) The Isabelle jEdit environment.



(b) The generated formal document.

Figure 2.: Generating documents with guaranteed syntactical and semantical consistency.

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

Isabelle, as a framework for building formal tools [35], provides the means for generating *formal documents*. With formal documents (such as the one you are currently reading) we refer to documents that are machine-generated and ensure certain formal guarantees. In particular, all formal content (e.g., definitions, formulae, types) are checked for consistency during the document generation.

For writing documents, Isabelle supports the embedding of informal texts using a $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ -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 $\text{@}\{\text{thm "not_not"}\}$ will instruct Isabelle to lock-up the (formally proven) theorem of name `ocL_not_not` and to replace the antiquotation with the actual theorem, i.e., $\text{not}(\text{not } x) = x$.

Figure 2 illustrates this approach: Figure 2a shows the jEdit-based development environment of Isabelle with an excerpt of one of the core theories of Featherweight OCL. Figure 2b shows the generated PDF document where all antiquotations are replaced. Moreover, the document generation tools allows for defining syntactic sugar as well as skipping technical details of the formalization.

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

1. It is an embedding into a powerful semantic meta-language and environment, namely Isabelle/HOL [27].
2. It is a *shallow embedding* in HOL; types in OCL were injectively mapped to types in Featherweight OCL. Ill-typed OCL specifications cannot be represented in Featherweight OCL and a type in Featherweight OCL contains exactly the values that are possible in OCL. Thus, sets may contain null (`Set{null}` is a defined set) but not invalid (`Set{invalid}` is just *invalid*).
3. Any Featherweight OCL type contains at least `invalid` and `null` (the type `Void` contains only these instances). The logic is consequently four-valued, and there is a null-element in the type `Set(A)`.

4. It is a strongly typed language in the Hindley-Milner tradition. We assume that a pre-process eliminates all implicit conversions due to sub-typing 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 [7]. 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. For example, the expression `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 sub-calculus, “cp” (a detailed discussion of the different equalities as well as the sub-calculus “cp”—for three-valued OCL 2.0—is given in [9]), which is nasty but can be hidden from the user inside tools.

Overall, this would contribute to one of the main goals of the OCL 2.5 RFP, as discussed at the OCL meeting in Aachen [14].

0.3. The Essence of UML-OCL Semantics

0.3.1. The Theory Organization

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

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

0.3.2. Denotational Semantics of Types

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

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

Types were directly represented in Featherweight OCL by types in HOL; consequently, any Featherweight OCL type must provide elements for a bottom element (also denoted \perp) and a null element; this is enforced in Isabelle by a type-class `null` that contains two distinguishable elements `bot` and `null` (see Chapter 1 for the details of the construction).

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

For most base types like `Booleanbase` or `Integerbase`, it suffices to double-lift a HOL library type:

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

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

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

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

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

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

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

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

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

0.3.3. Denotational Semantics of Constants and Operations

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

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

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

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

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

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

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

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

we can therefore write:

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

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

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

$$I[\text{not } X] \tau = (\text{case } I[X] \tau \text{ of} \\ \perp \Rightarrow \perp \\ |\perp \perp \Rightarrow \perp\!\!\!\perp \\ |\perp x \perp \Rightarrow \perp\!\!\!\perp x \perp)$$

$$I[X \text{ and } Y] \tau = (\text{case } I[X] \tau \text{ of} \\ \perp \Rightarrow (\text{case } I[Y] \tau \text{ of} \\ \perp \Rightarrow \perp \\ |\perp \perp \Rightarrow \perp \\ |\perp \text{true} \perp \Rightarrow \perp \\ |\perp \text{false} \perp \Rightarrow \perp\!\!\!\perp \text{false} \perp) \\ |\perp \perp \Rightarrow (\text{case } I[Y] \tau \text{ of} \\ \perp \Rightarrow \perp \\ |\perp \perp \Rightarrow \perp\!\!\!\perp \\ |\perp \text{true} \perp \Rightarrow \perp\!\!\!\perp \\ |\perp \text{false} \perp \Rightarrow \perp\!\!\!\perp \text{false} \perp) \\ |\perp \text{true} \perp \Rightarrow (\text{case } I[Y] \tau \text{ of} \\ \perp \Rightarrow \perp \\ |\perp \perp \Rightarrow \perp\!\!\!\perp \\ |\perp y \perp \Rightarrow \perp\!\!\!\perp y \perp) \\ |\perp \text{false} \perp \Rightarrow \perp\!\!\!\perp \text{false} \perp)$$

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+** or **+null+**. The definition of the addition for integers as default variant reads as follows:

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

where the operator “+” on the left-hand side of the equation denotes the OCL addition of type $\text{Integer} \Rightarrow \text{Integer} \Rightarrow \text{Integer}$ 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.

0.3.4. Logical Layer

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

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

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

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

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

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

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

The mandatory part of the OCL standard refers to an equality (written $x = y$ or $x \llcorner y$ for its negation), which is intended to be a strict operation (thus: `invalid = y` evaluates to `invalid`) and which uses the references of objects in a state when comparing objects, similarly to C++ or Java. In order to avoid confusions, we will use the following notations for equality:

1. The symbol $_ = _$ remains to be reserved to the HOL equality, i. e., the equality of our semantic meta-language,
2. The symbol $_ \triangleq _$ will be used for the *strong logical equality*, which follows the general logical principle that “equals can be replaced by equals,”¹² and is at the heart of the OCL logic,
3. The symbol $_ \doteq _$ is used for the strict referential equality, i. e., the equality the mandatory part of the OCL standard refers to by the $_ = _$ symbol.

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

$$I\llbracket X \triangleq Y \rrbracket\tau \equiv \perp\llbracket X \rrbracket\tau = I\llbracket Y \rrbracket\tau\perp$$

It enjoys nearly the laws of a congruence:

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

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

where the predicate cp stands for *context-passing*, a property that is true for all pure OCL expressions (but not arbitrary mixtures of OCL and HOL) in Featherweight OCL. The necessary side-calculus for establishing cp can be fully automated; the reader interested in the details is referred to Section 2.1.3.

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

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

The logical layer of the Featherweight OCL rules gives also a means to convert an OCL formula living in its four-valued world into a representation that is classically two-valued and can be processed by standard SMT solvers such as CVC3 [2] or Z3 [19]. δ -closure rules for all logical connectives have the following format, e. g.:

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

Together with the already mentioned general case-distinction

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

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

0.3.5. Algebraic Layer

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

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

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

```

(true and true) = true      (true and false) = false
(true and null) = null     (true and invalid) = invalid
(invalid and true) = invalid (invalid and false) = false
(invalid and null) = invalid (invalid and invalid) = invalid

```

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

```

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

```

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

The above equations explain the behavior for the most-important non-strict operations. The clarification of the exceptional behaviors is of key-importance for a semantic definition of the standard and the major deviation point from HOL-OCL [6, 8] to Featherweight OCL as presented here. Expressed in algebraic equations, “strictness-principles” boil down to:

```

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

```

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

```

 $\delta$  Set{} = true
 $\delta$  (X->including(x)) =  $\delta$  X and v x
Set{}->includes(x) = (if v x then false
                    else invalid endif)
(X->including(x)->includes(y)) =
  (if  $\delta$  X
   then if x  $\doteq$  y
        then true
        else X->includes(y)
        endif
   else invalid
   endif)

```

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

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

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

```
value "τ |= (Set{Set{2,null}} ≐ Set{Set{null,2}})"
```

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

0.3.6. Object-oriented Datatype Theories

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

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

1. C is a set of class names (written as $\{C_1, \dots, C_n\}$). To each class name a type of data in OCL is associated. Moreover, class names declare two projector functions to the set of all objects in a state: $C_i.\text{allInstances}()$ and $C_i.\text{allInstances@pre}()$,
2. $_ < _$ is an inheritance relation on classes,
3. $\text{Attrib}(C_i)$ is a collection of attributes associated to classes C_i . It declares two families of accessors; for each attribute $a \in \text{Attrib}(C_i)$ in a class definition C_i (denoted $X.a :: C_i \rightarrow A$ and $X.a@pre :: C_i \rightarrow A$ for $A \in \text{TYPES}(C)$),
4. $\text{Assoc}(C_i, C_j)$ is a collection of associations¹³. An association $(n, rn_{from}, rn_{to}) \in \text{Assoc}(C_i, C_j)$ between to classes C_i and C_j is a triple consisting of a (unique) association name n , and the role-names rn_{to} and rn_{from} . To each role-name belong two families of accessors denoted $X.a :: C_i \rightarrow A$ and $X.a@pre :: C_i \rightarrow A$ for $A \in \text{TYPES}(C)$,
5. for each pair $C_i < C_j$ ($C_i, C_j < C$), there is a cast operation of type $C_j \rightarrow C_i$ that can change the static type of an object of type C_i : $obj :: C_i.\text{oclAsType}(C_j)$,
6. for each class $C_i \in C$, there are two dynamic type tests ($X.\text{oclIsTypeOf}(C_i)$ and $X.\text{oclIsKindOf}(C_i)$),
7. and last but not least, for each class name $C_i \in C$ there is an instance of the overloaded referential equality (written $_ \doteq _$).

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

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

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

A Denotational Space for Class-Models: Object Universes

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

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

where inv_σ is a to be discussed invariant on states.

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

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

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

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

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

$$\mathfrak{A} := C_1 + \cdots + C_n. \quad (0.17)$$

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

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

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

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

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

Example instances of this scheme—outlining a compiler—can be found in Chapter 4 and Chapter 5.

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

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

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

We will work out concrete examples for the construction of the object-universes in Chapter 4 and Chapter 5 and the derivation of the respective datatype theories. While an automatization is clearly possible and desirable for concrete applications of Featherweight OCL, we consider this out of the scope of this document which has a focus on the semantic construction and its presentation.

Denotational Semantics of Accessors on Objects and Associations

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

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

The first phase directly translates into the following formalization:

definition

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

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

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

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

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

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

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

```

definition    in_pre_state = fst      first component
definition    in_post_state = snd     second component
definition    reconst_basetype = id   identity function

```

(0.22)

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

```

definition    __.getBase  :: C => A_base
where        X.getBase   = eval_extract X (deref_oid_C in_post_state
                                           (select_getBase reconst_basetype))

```

(0.23)

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

```

definition    __.getObject :: C => A_object
where        X.getObject   = eval_extract X (deref_oid_C in_post_state
                                           (select_getObject (deref_oid_C in_post_state)))

```

(0.24)

The variant for an accessor yielding a collection is omitted here; its construction follows by the application of the principles of the former two. The respective variants `__.a@pre` were produced when `in_post_state` is replaced by `in_pre_state`.

Examples for the construction of accessors via associations can be found in Section 4.8, the construction of accessors via attributes in Section 5.8. The construction of casts and type tests `->oclIsTypeOf()` and `->oclIsKindOf()` is similarly.

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

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

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

```

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

```

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

`null` is not counted. On the other hand, the operation `size` defined for collections in OCL does count `null`.

We propose to resolve this dilemma by regarding multiplicities as optional. This point of view complies with the UML standard, that does not require lower and upper bounds to be defined for multiplicities.¹⁴ In case a multiplicity is specified for an attribute, i. e., a lower and an upper bound are provided, we require for any collection the attribute evaluates to a collection not containing `null`. This allows for a straightforward interpretation of the multiplicity constraint. If bounds are not provided for an attribute, we consider the attribute values to not be restricted in any way. Because in particular the cardinality of the attribute's values is not bounded, the result of an evaluation of the attribute is of collection type. As the range of values that the attribute can assume is not restricted, the attribute can evaluate to a collection containing `null`. The attribute can also evaluate to `invalid`. Allowing multiplicities to be optional in this way gives the modeler the freedom to define attributes that can assume the full ranges of values provided by their types. However, we do not permit the omission of multiplicities for association ends, since the values of association ends are not only restricted by multiplicities, but also by other constraints enforcing the semantics of associations. Hence, the values of association ends cannot be completely unrestricted.

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

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

If the upper bound `n` is infinite, the second invariant is omitted. For the definition of these invariants we are making use of the conversion of single values to sets described in paragraph 0.3.6. 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.

Logic Properties of Class-Models

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

$$\begin{aligned} \tau \models X.\text{oclAsType}(C_i) &\triangleq X \\ \tau \models \text{invalid}.\text{oclAsType}(C_i) &\triangleq \text{invalid} \\ \tau \models \text{null}.\text{oclAsType}(C_i) &\triangleq \text{null} \\ \tau \models ((X :: C_i).\text{oclAsType}(C_j) \text{ .oclAsType}(C_i)) &\triangleq X \\ \tau \models X.\text{oclAsType}(C_j) \text{ .oclAsType}(C_i) &\triangleq X \\ \tau \models (X :: \text{OclAny}) \text{ .oclAsType}(\text{OclAny}) &\triangleq X \\ \tau \models v(X :: C_i) \implies \tau \models (X.\text{oclIsTypeOf}(C_i) \text{ implies } & (X.\text{oclAsType}(C_j).\text{oclAsType}(C_i)) \triangleq X) \end{aligned}$$

¹⁴We are however aware that a well-formedness rule of the UML standard does define a default bound of one in case a lower or upper bound is not specified.

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

Algebraic Properties of the Class-Models

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

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

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

$$\begin{aligned}
\text{invalid} .a &= \text{invalid} & \text{null} .a &= \text{invalid} \\
\text{invalid} .a@pre &= \text{invalid} & \text{null} .a@pre &= \text{invalid} \\
\text{invalid} .r &= \text{invalid} & \text{null} .r &= \text{invalid} \\
\text{invalid} .r@pre &= \text{invalid} & \text{null} .r@pre &= \text{invalid}
\end{aligned}$$

Other Operations on States

Defining $_ .\text{allInstances}()$ 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. OCL semantics admits states with “dangling references,”; it is the semantics of accessors or roles which maps these references to `invalid`, which makes it possible to rule out these situations in invariants.

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

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

where S is a set of object representations, encoding a set of oid's. The semantics of this operator is defined such that for any object whose oid is *not* represented in S and that is defined in pre and post

state, the corresponding object representation will not change in the state transition. A simplified presentation is as follows:

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

where $X' = I[\llbracket X \rrbracket](\sigma, \sigma')$ 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 $\text{Set}\{\} \rightarrow \text{oclIsModifiedOnly}()$ and exclude via $_.\text{oclIsNew}()$ and $_.\text{oclIsDeleted}()$ the existence of new or deleted objects, the operation is a query in the sense of the OCL standard, i. e., the `isQuery` property is true. So, whenever we have $\tau \models X \rightarrow \text{excluding}(s.a) \rightarrow \text{oclIsModifiedOnly}()$ and $\tau \models X \rightarrow \text{forAll}(x \text{ not } (x \doteq s.a))$, we can infer that $\tau \models s.a \triangleq s.a \text{ @pre}$.

0.3.7. Data Invariants

Since the present OCL semantics uses one interpretation function¹⁵, we express the effect of OCL terms occurring in preconditions and invariants by a syntactic transformation $_ \text{pre}$ which replaces:

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

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

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

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

0.3.8. Operation Contracts

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

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

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

where SOME x . $P(x)$ is the Hilbert-Choice Operator that chooses an arbitrary element satisfying P ; if such an element does not exist, it chooses an arbitrary one¹⁶. Thus, using the Hilbert-Choice Operator, a contract can be associated to a function definition:

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

provided that neither ϕ nor ψ contain recursive method calls of op . In the case of a query operation (i. e., τ must have the form: (σ, σ) , which means that query operations do not change the state; c.f. `oclIsModifiedOnly()` in Section 0.3.6), this constraint can be relaxed: the above equation is then stated as *axiom*. Note however, that the consistency of the overall theory is for recursive query contracts left to the user (it can be shown, for example, by a proof of termination, i. e., by showing that all recursive calls were applied to argument vectors that are smaller wrt. a well-founded ordering). Under this condition, an f_{op} resulting from recursive query operations can be used safely inside pre- and post-conditions of other contracts.

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

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

under the conditions:

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

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

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

where

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

Currently, Featherweight OCL neither supports overloading nor overriding for user-defined operations: the Featherweight OCL compiler needs to be extended to generate pre-conditions that constrain the classes on which an overridden function can be called as well as the dispatch order. This construction, overall, is similar to the virtual function table that, e.g., is generated by C++ compilers. Moreover, to avoid logical contradictions (inconsistencies) between different instances of an overridden operation, the user has to prove Liskov’s principle for these situations: pre-conditions of the superclass must imply pre-conditions of the subclass, and post-conditions of a subclass must imply post-conditions of the superclass.

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

1. Formalization I: OCL Types and Core Definitions

```
theory UML-Types
imports Complex-Main
begin
```

1.1. Preliminaries

1.1.1. Notations for the Option Type

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

```
unbundle no floor-ceiling-syntax
```

```
type-notation option ( $\langle\langle-\rangle_{\perp}\rangle$ )
notation Some ( $\langle\langle-\rangle_{\perp}\rangle$ )
notation None ( $\langle\langle-\rangle_{\perp}\rangle$ )
```

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

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

```
fun drop :: ' $\alpha$  option  $\Rightarrow$  ' $\alpha$  ( $\langle\langle-\rangle_{\perp}\rangle$ )
where drop-lift[simp]:  $\langle\langle v \rangle_{\perp}\rangle = v$ 
```

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

```
definition Sem :: ' $a \Rightarrow$  ' $a$  ( $\langle\langle I[-]\rangle$ )
where  $I[x] \equiv x$ 
```

1.1.2. Common Infrastructure for all OCL Types

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

data-invariant must be expressed in terms of the interface. In a second step, our base-types will be shown to be instances of this interface.

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

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

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

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

1.1.3. Accommodation of Basic Types to the Abstract Interface

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

```
instantiation option :: (type)bot
begin
  definition bot-option-def:  $(\text{bot}::'a \text{ option}) \equiv (\text{None}::'a \text{ option})$ 
  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:  $(\text{null}::'a::\text{bot} \text{ option}) \equiv \perp \text{ bot} \perp$ 
  instance proof show  $(\text{null}::'a::\text{bot} \text{ option}) \neq \text{bot}$ 
    by( simp add : null-option-def bot-option-def)
  qed
end
```

```
instantiation fun :: (type,bot) bot
begin
  definition bot-fun-def:  $\text{bot} \equiv (\lambda x. \text{bot})$ 
  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 ⇒ 'b::null) ≡ (λ x. null)
  instance proof
    show (null::'a ⇒ 'b::null) ≠ bot
    apply(auto simp: null-fun-def bot-fun-def)
    apply(drule-tac x=x in fun-cong)
    apply(erule contrapos-pp, simp add: null-is-valid)
  done
qed
end

```

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

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

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

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

```
type-synonym oid = nat
```

We refrained from the alternative:

```
type-synonym oid = ind
```

which is slightly more abstract but non-executable.

States in UML/OCL are a pair of

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

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

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

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

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

We will require for all objects that there is a function that projects the oid of an object in the state (we will settle the question how to define this function later). We will use the Isabelle type class mechanism [21] to capture this:

```
class object = fixes oid-of :: 'a ⇒ oid
```

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

typ 'A :: object

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

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

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

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

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

```
type-synonym ('A,'α) val = 'A st ⇒ 'α::null
```

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

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

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

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

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

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

Note that the definition :

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

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

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

1.2. Basic OCL Value Types

The structure of this section roughly follows the structure of Chapter 11 of the OCL standard [32], which introduces the OCL Library.

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

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

lemma $Boolean_{base}$ -cases:

fixes $b :: \langle Boolean_{base} \rangle$
obtains $\langle b = \perp \rangle \mid \langle b = \perp_{\perp} \rangle \mid \langle b = \perp_{\perp}False_{\perp} \rangle \mid \langle b = \perp_{\perp}True_{\perp} \rangle$
using *that by (cases b) auto*

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

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

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

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

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

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

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

typedef $Void_{base} = \{X::unit\ option\ option. X = bot \vee X = null\}$ **by**(*rule-tac x=bot in exI, simp*)

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

1.3. Some OCL Collection Types

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

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

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

1.3.1. The Construction of the Pair Type (Tuples)

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

```
typedef (overloaded) ( $\alpha, \beta$ )  $Pair_{base} = \{X :: (\alpha :: null \times \beta :: null) \text{ option option}.$ 
 $X = bot \vee X = null \vee (fst^{\ulcorner X \urcorner} \neq bot \wedge snd^{\ulcorner X \urcorner} \neq bot)\}$ 
by (rule-tac  $x=bot$  in  $exI, simp$ )
```

```
setup-lifting type-definition-Pair_base
```

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

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

```
lift-definition  $bot\text{-}Pair_{base} :: (\alpha, \beta) Pair_{base} \text{ is } None$ 
by (simp add: bot-option-def)
```

```
instance by (standard; transfer)
(auto simp add: null-option-def bot-option-def)
```

```
end
```

```
lemma  $Abs\text{-}Pair_{base}\text{-}invalid\text{-}eq$  [simp]:
 $\langle Abs\text{-}Pair_{base} (invalid \tau) = invalid \tau \rangle$ 
by (simp add: invalid-def bot-Pair_base-def bot-option-def)
```

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

```
lift-definition  $null\text{-}Pair_{base} :: (\alpha, \beta) Pair_{base} \text{ is } \ulcorner None \urcorner$ 
by (simp add: null-option-def bot-option-def)
```

```
instance by (standard; transfer)
(auto simp add: null-option-def bot-option-def)
```

```
end
```

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

```
lift-definition  $equal\text{-}Pair_{base} :: (\alpha, \beta) Pair_{base} \Rightarrow (\alpha, \beta) Pair_{base} \Rightarrow bool$ 
is  $HOL.equal :: (\alpha \times \beta) \text{ option option} \Rightarrow -$ 
.
```

```
instance by (standard; transfer)
(simp add: equal)
```

```
end
```

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

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

1.3.2. The Construction of the Set Type

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

```
typedef (overloaded) ' $\alpha$   $Set_{base} = \{X :: ('a :: null) \text{ set option option. } X = bot \vee X = null \vee (\forall x \in {}^\top X^\top. x \neq bot)\}$ 
  by (rule-tac  $x=bot$  in  $exI$ ,  $simp$ )
```

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

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

```
  instance proof show  $\exists x :: 'a Set_{base}. x \neq bot$ 
    apply(rule-tac  $x=Abs-Set_{base} \_None\_$  in  $exI$ )
    by( $simp$  add:  $bot-Set_{base-def}$   $Abs-Set_{base}-inject$   $null-option-def$   $bot-option-def$ )
```

```
    qed
```

```
end
```

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

```
  definition  $null-Set_{base-def} : (null :: ('a :: null) Set_{base}) \equiv Abs-Set_{base} \_None\_$ 
```

```
  instance proof show  $(null :: ('a :: null) Set_{base}) \neq bot$ 
    by( $simp$  add:  $null-Set_{base-def}$   $bot-Set_{base-def}$   $Abs-Set_{base}-inject$ 
       $null-option-def$   $bot-option-def$ )
```

```
    qed
```

```
end
```

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

```
type-synonym ( $'\mathfrak{A}, 'a$ )  $Set = ('a, 'a Set_{base}) \text{ val}$ 
type-notation  $Set_{base} (\langle Set'(-) \rangle)$ 
```

1.3.3. The Construction of the Bag Type

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

```
typedef (overloaded) ' $\alpha$   $Bag_{base} = \{X :: ('a :: null \Rightarrow nat) \text{ option option. } X = bot \vee X = null \vee {}^\top X^\top bot = 0\}$ 
  by (rule-tac  $x=bot$  in  $exI$ ,  $simp$ )
```

```
instantiation  $Bag_{base} :: (null)bot$ 
begin
```

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

```
  instance proof show  $\exists x :: 'a Bag_{base}. x \neq bot$ 
```

```

apply(rule-tac x=Abs-Bagbase ⊥ None in exI)
by(simp add: bot-Bagbase-def Abs-Bagbase-inject
    null-option-def bot-option-def)
  qed
end

instantiation Bagbase :: (null)null
begin

  definition null-Bagbase-def: (null::('a::null) Bagbase) ≡ Abs-Bagbase ⊥ None

  instance proof show (null::('a::null) Bagbase) ≠ bot
    by(simp add:null-Bagbase-def bot-Bagbase-def Abs-Bagbase-inject
      null-option-def bot-option-def)
  qed
end

```

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

```

type-synonym (' $\mathfrak{A}$ , ' $\alpha$ ) Bag = (' $\mathfrak{A}$ , ' $\alpha$  Bagbase) val
type-notation Bagbase (⟨Bag'(-)⟩)

```

1.3.4. The Construction of the Sequence Type

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

```

typedef (overloaded) ' $\alpha$  Sequencebase = {X::(' $\alpha$ ::null) list option option.
    X = bot ∨ X = null ∨ (∀ x ∈ set ⌈X⌈. x ≠ bot)}
  by (rule-tac x=bot in exI, simp)

```

```

setup-lifting type-definition-Sequencebase

```

```

instantiation Sequencebase :: (null) bot
begin

```

```

lift-definition bot-Sequencebase :: ' $a$  Sequencebase is None
  by (simp add: null-option-def bot-option-def)

```

```

instance by (standard; transfer)
  (auto simp add: null-option-def bot-option-def)

```

```

end

```

```

lemma Sequencebase-invalid-eq [simp]:
  ⟨Abs-Sequencebase (invalid  $\tau$ ) = invalid  $\tau$ ⟩
  by (simp add: invalid-def bot-Sequencebase-def bot-option-def)

```

```

instantiation Sequencebase :: (null) null
begin

```

```

lift-definition null-Sequencebase :: ' $a$  Sequencebase is ⊥ None
  by (simp add: null-option-def bot-option-def)

```

```

instance by (standard; transfer)
  (auto simp add: null-option-def bot-option-def)

```

```

end

```

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

lift-definition $equal-Sequence_{base} :: 'a Sequence_{base} \Rightarrow 'a Sequence_{base} \Rightarrow bool$
is $HOL.equal :: 'a list option option \Rightarrow -$

.

instance by (*standard; transfer*)
(*simp add: equal*)

end

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

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

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

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

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

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

OCL Type	HOL Type
Boolean	$'A Boolean$
Boolean -> Boolean	$'A Boolean \Rightarrow 'A Boolean$
(Integer, Integer) -> Boolean	$'A Integer \Rightarrow 'A Integer \Rightarrow 'A Boolean$
Set(Integer)	$('A, Integer_{base}) Set$
Set(Integer)-> Real	$('A, Integer_{base}) Set \Rightarrow 'A Real$
Set(Pair(Integer, Boolean))	$('A, Pair(Integer_{base}, Boolean_{base})) Set$
Set(<T>)	$('A, 'a) Set$

Table 1.1.: Correspondance between OCL types and HOL types

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

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

Note, furthermore, that our construction of “fully abstract types” (no junk, no confusion) assures that the logical equality to be defined in the next section works correctly and comes as element of the “lingua franca”, i. e. HOL.

end

2. Formalization II: OCL Terms and Library Operations

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

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

2.1.1. 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) = ( $\lambda \tau. \perp\!\!\!\perp$ )
by(simp add: null-fun-def null-option-def bot-option-def)
```

```
definition true :: ('A)Boolean
where true  $\equiv \lambda \tau. \perp\!True_{\perp}$ 
```

```
definition false :: ('A)Boolean
where false  $\equiv \lambda \tau. \perp\!False_{\perp}$ 
```

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

 $X \tau = true \tau \vee X \tau = false \tau$ 
apply(simp add: invalid-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) =  $\perp\!False_{\perp}$ 
by(simp add: false-def)
```

```
lemma [simp]: true (a, b) =  $\perp\!True_{\perp}$ 
by(simp add: true-def)
```

```
lemma textbook-true:  $I\llbracket true \rrbracket \tau = \perp\!True_{\perp}$ 
by(simp add: Sem-def true-def)
```

```
lemma textbook-false:  $I\llbracket false \rrbracket \tau = \perp\!False_{\perp}$ 
by(simp add: Sem-def false-def)
```

2.1.2. Validity and Definedness

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

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

Table 2.1.: Basic semantic constant definitions of the logic

definition *valid* :: ($\mathfrak{A}, 'a::\text{null}$)*val* \Rightarrow (\mathfrak{A})*Boolean* ($\langle v \rightarrow [100]100$)

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

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

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

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

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

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

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

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

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

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

by(*simp add: valid-def*) **definition** *defined* :: ($\mathfrak{A}, 'a::\text{null}$)*val* \Rightarrow (\mathfrak{A})*Boolean* ($\langle \delta \rightarrow [100]100$)

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

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

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

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

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

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

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

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 *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*: $I[\delta(X)] \tau = (if\ I[X] \tau = I[bot] \tau \vee I[X] \tau = I>null] \tau$
 then $I[false] \tau$
 else $I>true] \tau$)

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

lemma *textbook-valid*: $I[v(X)] \tau = (if\ I[X] \tau = I[bot] \tau$
 then $I[false] \tau$
 else $I>true] \tau$)

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

Table 2.2 and Table 2.3 summarize the results of this section.

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

Table 2.2.: 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 2.3.: Laws of the basic predicates of the logic.

2.1.3. The Equalities of OCL

The OCL contains a particular version of equality, written in Standard documents $_ = _$ and $_ <> _$ for its negation, which is referred as *weak referential equality* hereafter and for which we use the symbol $_ \doteq _$ throughout the formal part of this document. Its semantics is motivated by the desire of fast execution, and similarity to languages like Java and C, but does not satisfy the needs of logical reasoning over OCL expressions and specifications. We therefore introduce a second equality, referred as *strong equality* or *logical equality* and written $_ \triangleq _$ which is not present in the current standard but was discussed in prior texts on OCL like the Amsterdam Manifesto [18] and was identified as desirable extension of OCL in the Aachen Meeting [14] in the future 2.5 OCL Standard. The purpose of strong equality is to define and reason over OCL. It is therefore a natural task in Featherweight OCL to formally investigate the somewhat quite complex relationship between these two.

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

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

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

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

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

$$s = t \implies P(s) = P(t) \tag{2.1}$$

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

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

Definition

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

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

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

From this follow already elementary properties like:

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*)

Fundamental Predicates on Strong Equality

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

lemma *StrongEq-refl* [*simp*]: $(X \triangleq X) = true$
by(*rule ext*, *simp add: 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: invalid-def true-def false-def StrongEq-def*)
apply(*drule-tac x=x in fun-cong*)
by *auto*

it is only in a limited sense a congruence, at least from the point of view of this semantic theory. The point is that it is only a congruence on OCL expressions, not arbitrary HOL expressions (with which we can mix Featherweight OCL expressions). A semantic—not syntactic—characterization of OCL expressions is that they are *context-passing* or *context-invariant*, i. e., the context of an entire OCL expression, i. e. the pre and post state it refers to, is passed constantly and unmodified to the sub-expressions, i. e., all sub-expressions inside an OCL expression refer to the same context. Expressed formally, this boils down to:

lemma *StrongEq-subst* :
assumes $cp: \bigwedge X. P(X)\tau = P(\lambda -. X \tau)\tau$
and $eq: (X \triangleq Y)\tau = true \tau$
shows $(P X \triangleq P Y)\tau = true \tau$
apply(*insert cp eq*)
apply(*simp add: invalid-def true-def false-def StrongEq-def*)
apply(*subst cp[of X]*)
apply(*subst cp[of Y]*)
by *simp*

lemma *defined7*[*simp*]: $\delta (X \triangleq Y) = true$
by(*rule ext*,
auto simp: defined-def true-def false-def StrongEq-def
bot-fun-def bot-option-def null-option-def null-fun-def)

lemma *valid7*[*simp*]: $v (X \triangleq Y) = true$
by(*rule ext*,
auto simp: valid-def true-def false-def StrongEq-def
bot-fun-def bot-option-def null-option-def null-fun-def)

lemma *cp-StrongEq*: $(X \triangleq Y) \tau = ((\lambda -. X \tau) \triangleq (\lambda -. Y \tau)) \tau$
by(*simp add: StrongEq-def*)

2.1.4. Logical Connectives and their Universal Properties

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

Practically, this means that we want to give a definition to the core operations to be as close as possible to the lattice laws; this makes also powerful symbolic normalization of OCL specifications possible as a pre-requisite for automated theorem provers. For example, it is still possible to compute without any definedness and validity reasoning the DNF of an OCL specification; be it for test-case generations or for a smooth transition to a two-valued representation of the specification amenable to fast standard SMT-solvers, for example.

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

definition *OclNot* :: (\mathfrak{A})Boolean \Rightarrow (\mathfrak{A})Boolean ($\langle not \rangle$)

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

$$\begin{array}{l} \perp \Rightarrow \perp \\ | \perp \perp \Rightarrow \perp \perp \\ | \perp x \Rightarrow \perp \neg x \end{array}$$

lemma *cp-OclNot*: $(not\ X)\tau = (not\ (\lambda\ \cdot X\ \tau))\ \tau$
by(*simp add: OclNot-def*)

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

lemma *OclNot2[simp]*: $not\ null = null$
by(*rule ext, simp add: OclNot-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 invalid-def true-def false-def*)

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

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

lemma *OclNot-inject*: $\bigwedge\ x\ y. not\ x = not\ y \implies x = y$
by(*subst OclNot-not[THEN sym], simp*)

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

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

$$\begin{array}{l} \perp \Rightarrow \perp \\ | \perp \Rightarrow (case\ Y\ \tau\ of \\ \quad \perp \Rightarrow \perp \\ \quad \perp \perp \Rightarrow \perp \perp \\ \quad \perp x \Rightarrow \perp \neg x) \\ | \perp \perp \Rightarrow (case\ Y\ \tau\ of \\ \quad \perp \Rightarrow \perp \\ \quad \perp \perp \Rightarrow \perp \perp \\ \quad \perp x \Rightarrow \perp \neg x) \\ | \perp True \Rightarrow Y\ \tau \end{array}$$

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

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

lemma *textbook-OclNot*:
 $I[not(X)]\ \tau = (case\ I[X]\ \tau\ of\ \perp \Rightarrow \perp$

$$\begin{array}{l} | \perp \perp \Rightarrow \perp \perp \\ | \perp x \Rightarrow \perp \neg x \end{array}$$
by(*simp add: Sem-def OclNot-def*)

lemma *textbook-OclAnd*:

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

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

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

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

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

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

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

lemma *OclAnd1*[*simp*]: (*invalid and true*) = *invalid*
by(*rule ext, simp add: OclAnd-def invalid-def true-def false-def bot-option-def*)

lemma *OclAnd2*[*simp*]: (*invalid and false*) = *false*
by(*rule ext, simp add: OclAnd-def invalid-def true-def false-def bot-option-def*)

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

lemma *OclAnd5*[*simp*]: (*null and true*) = *null*
by(*rule ext, simp add: OclAnd-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 invalid-def true-def false-def bot-option-def null-fun-def null-option-def*)

lemma *OclAnd7[simp]*: $(\text{null and null}) = \text{null}$
by(*rule ext,simp add: OclAnd-def invalid-def true-def false-def bot-option-def null-fun-def null-option-def*)

lemma *OclAnd8[simp]*: $(\text{null and invalid}) = \text{invalid}$
by(*rule ext,simp add: OclAnd-def invalid-def true-def false-def bot-option-def null-fun-def null-option-def*)

lemma *OclAnd9[simp]*: $(\text{false and true}) = \text{false}$
by(*rule ext,simp add: OclAnd-def invalid-def true-def false-def*)

lemma *OclAnd10[simp]*: $(\text{false and false}) = \text{false}$
by(*rule ext,simp add: OclAnd-def invalid-def true-def false-def*)

lemma *OclAnd11[simp]*: $(\text{false and null}) = \text{false}$
by(*rule ext,simp add: OclAnd-def invalid-def true-def false-def*)

lemma *OclAnd12[simp]*: $(\text{false and invalid}) = \text{false}$
by(*rule ext,simp add: OclAnd-def invalid-def true-def false-def*)

lemma *OclAnd13[simp]*: $(\text{true and true}) = \text{true}$
by(*rule ext,simp add: OclAnd-def invalid-def true-def false-def*)

lemma *OclAnd14[simp]*: $(\text{true and false}) = \text{false}$
by(*rule ext,simp add: OclAnd-def invalid-def true-def false-def*)

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

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

lemma *OclAnd-idem[simp]*: $(X \text{ and } X) = X$
apply(*rule ext,simp add: OclAnd-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*: $(X \text{ and } Y) = (Y \text{ and } X)$
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]*: $(\text{false and } X) = \text{false}$
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]*: $(X \text{ and false}) = \text{false}$
by(*simp add: OclAnd-commute*)

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

by(*simp add: OclAnd-commute*)

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

apply(*simp add: OclAnd-def*)

apply(*auto simp:true-def false-def bot-fun-def bot-option-def*
split: option.split option.split-asm)

done

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

by(*simp add: OclAnd-commute*)

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

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]*: $\bigwedge \tau. X \tau \neq \text{false} \tau \implies X \tau \neq \text{bot } \tau \implies (X \text{ and null}) \tau = \text{null } \tau$

by(*simp add: OclAnd-commute*)

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

proof (*rule ext*)

fix *p*

show $\langle ?lhs \ p = ?rhs \ p \rangle$

by (*cases* $\langle X \ p \rangle$ *rule: Boolean_base-cases*; *cases* $\langle Y \ p \rangle$ *rule: Boolean_base-cases*; *cases* $\langle Z \ p \rangle$ *rule:*

Boolean_base-cases)

(*simp-all add: OclAnd-def*)

qed

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

by(*simp add: OclOr-def*)

lemma *OclOr-commute*: $(X \text{ or } Y) = (Y \text{ or } X)$
by(*simp add: OclOr-def OclAnd-commute*)

lemma *OclOr-false1*[*simp*]: $(\text{false or } Y) = Y$
by(*simp add: OclOr-def*)

lemma *OclOr-false2*[*simp*]: $(Y \text{ or false}) = Y$
by(*simp add: OclOr-def*)

lemma *OclOr-true1*[*simp*]: $(\text{true or } Y) = \text{true}$
by(*simp add: OclOr-def*)

lemma *OclOr-true2*: $(Y \text{ or true}) = \text{true}$
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) option.sel*)

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*: $(X \text{ or } (Y \text{ or } Z)) = (X \text{ or } Y \text{ or } Z)$
by(*simp add: OclOr-def OclAnd-assoc*)

lemma *deMorgan1*: $\text{not}(X \text{ and } Y) = ((\text{not } X) \text{ or } (\text{not } Y))$
by(*simp add: OclOr-def*)

lemma *deMorgan2*: $\text{not}(X \text{ or } Y) = ((\text{not } X) \text{ and } (\text{not } Y))$
by(*simp add: OclOr-def*)

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

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

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

2.1.5. A Standard Logical Calculus for OCL

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

```

syntax OclNonValid :: [( $\mathcal{A}$ )st, ( $\mathcal{A}$ )Boolean]  $\Rightarrow$  bool ( $\langle(1(-)/ \neq (-))\rangle$  50)
syntax-consts OclNonValid == Not
translations  $\tau \neq P == \neg(\tau \models P)$ 

```

Global vs. Local Judgements

```

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

```

```

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

```

```

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

```

```

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

```

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

```

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

```

Local Validity and Meta-logic

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma bool-split[simp]:
 $(\tau \models (x \triangleq invalid)) \vee (\tau \models (x \triangleq null)) \vee (\tau \models (x \triangleq true)) \vee (\tau \models (x \triangleq false))$ 
apply(insert bool-split-0[of x  $\tau$ ], auto)
apply(simp-all add: OclValid-def StrongEq-def true-def invalid-def)
done

```

```

lemma defined-split:
 $(\tau \models \delta x) = ((\neg(\tau \models (x \triangleq invalid))) \wedge (\neg(\tau \models (x \triangleq null))))$ 

```

by(*simp add: defined-def true-def false-def invalid-def
StrongEq-def OclValid-def bot-fun-def null-fun-def*)

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

lemma *defined-bool-split*: $(\tau \models \delta A) = ((\tau \models A) \vee (\tau \models not A))$
by(*auto simp: defined-def true-def false-def invalid-def OclNot-def
StrongEq-def OclValid-def bot-fun-def bot-option-def null-option-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*]:
 $(\tau \models not (\delta x)) = (\neg (\tau \models \delta x))$
by(*simp add: OclNot-def OclValid-def true-def false-def defined-def
split: option.split option.split-asm*)

lemma *foundation7'*[*simp*]:
 $(\tau \models not (v x)) = (\neg (\tau \models v x))$
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*:
 $(\tau \models \delta x) \vee (\tau \models (x \triangleq invalid)) \vee (\tau \models (x \triangleq null))$
proof –
 have 1 : $(\tau \models \delta x) \vee (\neg(\tau \models \delta x))$ **by** *auto*
 have 2 : $(\neg(\tau \models \delta x)) = ((\tau \models (x \triangleq invalid)) \vee (\tau \models (x \triangleq null)))$
 by(*simp only: defined-split, simp*)
 show ?thesis **by**(*insert 1, simp add:2*)
qed

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

lemma *foundation9'*:
 $\tau \models not x \implies \neg (\tau \models x)$
by(*auto simp: foundation6 foundation9*)

lemma foundation9'':

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

by(metis OclNot3 OclNot-not OclValid-def cp-OclNot cp-defined defined4)

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: defined-split)

by(auto simp: OclAnd-def OclValid-def invalid-def
true-def StrongEq-def null-fun-def null-option-def bot-option-def
split:bool.split-asm)

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

by(auto dest:foundation5 simp:foundation6 foundation10)

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: defined-split)

by(auto simp: OclNot-def OclOr-def OclAnd-def OclValid-def invalid-def
true-def StrongEq-def null-fun-def null-option-def bot-option-def
split:bool.split-asm bool.split)

lemma foundation12:

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

apply(simp add: defined-split)

by(auto simp: OclNot-def OclOr-def OclAnd-def OclImplies-def bot-option-def
OclValid-def invalid-def true-def StrongEq-def null-fun-def null-option-def
split:bool.split-asm bool.split option.split-asm)

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

by(auto simp: OclNot-def OclValid-def invalid-def true-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 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
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:if-split-asm)

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

apply(simp add: foundation16)

by(auto simp: defined-def false-def true-def bot-fun-def null-fun-def OclValid-def StrongEq-def invalid-def)

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

apply(simp add: invalid-def null-fun-def)

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

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:if-split-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:if-split-asm*)

lemma foundation18'': $(\tau \models (v X)) = (\neg(\tau \models (X \triangleq \text{invalid})))$
by(*auto simp:foundation15*)

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.if-split-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: $(\tau \models P) = (\tau \models (\lambda . . P \tau))$
by(*auto simp: OclValid-def true-def*)

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

lemma foundation25: $\tau \models P \implies \tau \models (P \text{ or } Q)$
by(*simp add: OclOr-def OclNot-def OclAnd-def OclValid-def true-def*)

lemma foundation25': $\tau \models Q \implies \tau \models (P \text{ or } Q)$
by(*subst OclOr-commute, simp add: foundation25*)

lemma foundation26:
assumes *defP:* $\tau \models \delta P$
assumes *defQ:* $\tau \models \delta Q$
assumes *H:* $\tau \models (P \text{ or } Q)$
assumes *P:* $\tau \models P \implies R$
assumes *Q:* $\tau \models Q \implies R$
shows *R*
by(*insert H, subst (asm) foundation11[OF defP defQ], erule disjE, simp-all add: P Q*)

lemma foundation27: $\tau \models A \implies (\tau \models A \text{ implies } B) = (\tau \models B)$
by (*simp add: foundation12 foundation6*)

lemma defined-not-I : $\tau \models \delta (x) \implies \tau \models \delta (\text{not } x)$
by(*auto simp: OclNot-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.if-split-asm)*

lemma *valid-not-I* : $\tau \models v(x) \implies \tau \models v(\text{not } x)$
by(*auto simp: OclNot-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.if-split-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 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.if-split-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 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.if-split-asm*)
by(*auto simp: null-option-def split: option.split bool.split*)

lemma *defined-or-I* : $\tau \models \delta(x) \implies \tau \models \delta(y) \implies \tau \models \delta(x \text{ or } y)$
by(*simp add: OclOr-def defined-and-I defined-not-I*)

lemma *valid-or-I* : $\tau \models v(x) \implies \tau \models v(y) \implies \tau \models v(x \text{ or } y)$
by(*simp add: OclOr-def valid-and-I valid-not-I*)

Local Judgements and Strong Equality

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

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

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

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

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

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

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

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

by assumption

lemma *StrongEq-L-subst3*:
assumes *cp*: $cp\ P$
and *eq*: $\tau \models (x \triangleq y)$
shows $(\tau \models P\ x) = (\tau \models P\ y)$
apply(rule *iffI*)
apply(rule *StrongEq-L-subst2*[*OF cp, OF eq*],*simp*)
apply(rule *StrongEq-L-subst2*[*OF cp, OF eq*][*THEN StrongEq-L-sym*],*simp*)
done

lemma *StrongEq-L-subst3-rev*:
assumes *eq*: $\tau \models (x \triangleq y)$
and *cp*: $cp\ P$
shows $(\tau \models P\ x) = (\tau \models P\ y)$
by(insert *cp*, erule *StrongEq-L-subst3*, rule *eq*)

lemma *StrongEq-L-subst4-rev*:
assumes *eq*: $\tau \models (x \triangleq y)$
and *cp*: $cp\ P$
shows $(\neg(\tau \models P\ x)) = (\neg(\tau \models P\ y))$
thm *arg-cong*[*of - - Not*]
apply(rule *arg-cong*[*of - - Not*])
by(insert *cp*, erule *StrongEq-L-subst3*, rule *eq*)

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

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

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

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

lemma *cpI5*:
 $(\forall V\ W\ X\ Y\ Z\ \tau. f\ V\ W\ X\ Y\ Z\ \tau = f(\lambda\cdot. V\ \tau)\ (\lambda\cdot. W\ \tau)(\lambda\cdot. X\ \tau)(\lambda\cdot. Y\ \tau)(\lambda\cdot. Z\ \tau)\ \tau) \implies$
 $cp\ N \implies cp\ P \implies cp\ Q \implies cp\ R \implies cp\ S \implies cp(\lambda X. f\ (N\ X)\ (P\ X)\ (Q\ X)\ (R\ X)\ (S\ X))$
apply(auto *simp*: *cp-def*)
apply(rule *exI*, (rule *allI*)+)

by(erule-tac x=N X in allE, auto)

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

lemma cp-id : cp(λ X. X)
by (simp add: cp-def, fast)**lemmas** cp-intro[*intro!,simp,code-unfold*] =
cp-const
cp-id
cp-defined[*THEN allI[THEN allI[THEN cpI1], of defined*]]
cp-valid[*THEN allI[THEN allI[THEN cpI1], of valid*]]
cp-OclNot[*THEN allI[THEN allI[THEN cpI1], of not*]]
cp-OclAnd[*THEN allI[THEN allI[THEN allI[THEN cpI2]], of (and)*]]
cp-OclOr[*THEN allI[THEN allI[THEN allI[THEN cpI2]], of (or)*]]
cp-OclImplies[*THEN allI[THEN allI[THEN allI[THEN cpI2]], of (implies)*]]
cp-StrongEq[*THEN allI[THEN allI[THEN allI[THEN cpI2]],*
of StrongEq]]

2.1.6. OCL's if then else endif

definition OclIf :: [$(\lambda$ Boolean, (λ ' α ::null val, (λ ' α) val) \Rightarrow (λ ' α) val
(*if* (-) then (-) else (-) endif) [10,10,10]50)

where (*if* C then B₁ else B₂ endif) = (λ τ . *if* (δ C) τ = true τ
then (*if* (C τ) = true τ
then B₁ τ
else B₂ τ)
else *invalid* τ)

lemma cp-OclIf:((*if* C then B₁ else B₂ endif) τ =
(*if* (λ -. C τ) then (λ -. B₁ τ) else (λ -. B₂ τ) endif) τ)
by(simp only: OclIf-def, subst cp-defined, rule refl)**lemmas** cp-intro'[*intro!,simp,code-unfold*] =
cp-intro
cp-OclIf[*THEN allI[THEN allI[THEN allI[THEN allI[THEN cpI3]]], of OclIf*]]**lemma** OclIf-invalid
[*simp*]: (*if* *invalid* then B₁ else B₂ endif) = *invalid*
by(rule ext, auto simp: OclIf-def)

lemma OclIf-null [*simp*]: (*if* null then B₁ else B₂ endif) = *invalid*
by(rule ext, auto simp: OclIf-def)

lemma OclIf-true [*simp*]: (*if* true then B₁ else B₂ endif) = B₁
by(rule ext, auto simp: OclIf-def)

lemma OclIf-true' [*simp*]: $\tau \models P \Longrightarrow$ (*if* P then B₁ else B₂ endif) τ = B₁ τ
apply(subst cp-OclIf, auto simp: OclValid-def)
by(simp add: cp-OclIf[symmetric])

lemma OclIf-true'' [*simp*]: $\tau \models P \Longrightarrow \tau \models$ (*if* P then B₁ else B₂ endif) \triangleq B₁
by(subst OclValid-def, simp add: StrongEq-def true-def)

lemma OclIf-false [*simp*]: (*if* false then B₁ else B₂ endif) = B₂
by(rule ext, auto simp: OclIf-def)

lemma OclIf-false' [*simp*]: $\tau \models$ not P \Longrightarrow (*if* P then B₁ else B₂ endif) τ = B₂ τ
apply(subst cp-OclIf)
apply(auto simp: foundation14[symmetric] foundation22)

by(*auto simp: cp-OclIf[symmetric]*)

lemma *OclIf-idem1*[*simp*]:(*if* δX *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*

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

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

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

with term "not" we can express the notation:

syntax
notequal :: $(\mathfrak{A})Boolean \Rightarrow (\mathfrak{A})Boolean \Rightarrow (\mathfrak{A})Boolean$ (**infix** $\langle \langle \rangle \rangle$ 40)
syntax-consts
notequal == *OclNot*
translations
 $a \langle \rangle b$ == *CONST OclNot*($a \doteq b$)

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

2.1.8. Laws to Establish Definedness (δ -closure)

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

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

lemma *OclNot-contrapos-nn*:
assumes $A: \tau \models \delta A$
assumes $B: \tau \models \text{not } B$
assumes $C: \tau \models A \Longrightarrow \tau \models B$
shows $\tau \models \text{not } A$
proof –
have $D: \tau \models \delta B$ **by**(*rule B[THEN OclNot-defargs]*)
show *?thesis*
apply(*insert B, simp add: A D foundation9*)

by(*erule contrapos-nn*, *auto intro: C*)
qed

2.1.9. A Side-calculus for Constant Terms

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

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

lemma *const-subst*:

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

proof –

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

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

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

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

qed

lemma *const-imp2* :
assumes $\bigwedge \tau \tau'. P \ \tau = P \ \tau' \implies Q \ \tau = Q \ \tau'$
shows $\text{const } P \implies \text{const } Q$
by(*simp add: const-def*, *insert assms*, *blast*)

lemma *const-imp3* :
assumes $\bigwedge \tau \tau'. P \ \tau = P \ \tau' \implies Q \ \tau = Q \ \tau' \implies R \ \tau = R \ \tau'$
shows $\text{const } P \implies \text{const } Q \implies \text{const } R$
by(*simp add: const-def*, *insert assms*, *blast*)

lemma *const-imp4* :
assumes $\bigwedge \tau \tau'. P \ \tau = P \ \tau' \implies Q \ \tau = Q \ \tau' \implies R \ \tau = R \ \tau' \implies S \ \tau = S \ \tau'$
shows $\text{const } P \implies \text{const } Q \implies \text{const } R \implies \text{const } S$
by(*simp add: const-def*, *insert assms*, *blast*)

lemma *const-lam* : $\text{const } (\lambda-. e)$
by(*simp add: const-def*)

lemma *const-true*[simp] : const true
by(simp add: const-def true-def)

lemma *const-false*[simp] : const false
by(simp add: const-def false-def)

lemma *const-null*[simp] : const null
by(simp add: const-def null-fun-def)

lemma *const-invalid* [simp]: const invalid
by(simp add: const-def invalid-def)

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

lemma *const-defined* :
assumes const X
shows const (δ X)
by(rule const-imp2[OF - assms],
simp add: defined-def false-def true-def bot-fun-def bot-option-def null-fun-def null-option-def)

lemma *const-valid* :
assumes const X
shows const (v X)
by(rule const-imp2[OF - assms],
simp add: valid-def false-def true-def bot-fun-def null-fun-def assms)

lemma *const-OclAnd* :
assumes const X
assumes const X'
shows const (X and X')
by(rule const-imp3[OF - assms], subst (1 2) cp-OclAnd, simp add: assms OclAnd-def)

lemma *const-OclNot* :
assumes const X
shows const (not X)
by(rule const-imp2[OF - assms],subst cp-OclNot,simp add: assms OclNot-def)

lemma *const-OclOr* :
assumes const X
assumes const X'
shows const (X or X')
by(simp add: assms OclOr-def const-OclNot const-OclAnd)

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

lemma *const-StrongEq*:
assumes const X
assumes const X'

```

shows const( $X \triangleq X'$ )
apply(simp only: StrongEq-def const-def, intro all)
apply(subst assms(1)[THEN const-charn])
apply(subst assms(2)[THEN const-charn])
by simp

```

```

lemma const-OclIf :
  assumes const B
    and const C1
    and const C2
  shows const (if B then C1 else C2 endif)
apply(rule const-implly4[OF - assms],
  subst (1 2) cp-OclIf, simp only: OclIf-def cp-defined[symmetric])
apply(simp add: const-defined[OF assms(1), simplified const-def, THEN spec, THEN spec]
  const-true[simplified const-def, THEN spec, THEN spec]
  assms[simplified const-def, THEN spec, THEN spec]
  const-invalid[simplified const-def, THEN spec, THEN spec])
by (metis (no-types) bot-fun-def OclValid-def const-def const-true defined-def
  foundation16[THEN iffD1] null-fun-def)

```

```

lemma const-OclValid1:
  assumes const x
  shows ( $\tau \models \delta x = \tau' \models \delta x$ )
apply(simp add: OclValid-def)
apply(subst const-defined[OF assms, THEN const-charn])
by(simp add: true-def)

```

```

lemma const-OclValid2:
  assumes const x
  shows ( $\tau \models v x = \tau' \models v x$ )
apply(simp add: OclValid-def)
apply(subst const-valid[OF assms, THEN const-charn])
by(simp add: true-def)

```

```

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

```

```

lemma const-HOL-and: const C  $\implies$  const D  $\implies$  const ( $\lambda\tau. C \ \tau \wedge D \ \tau$ )
by(auto simp: const-def)

```

```

lemma const-HOL-eq : const C  $\implies$  const D  $\implies$  const ( $\lambda\tau. C \ \tau = D \ \tau$ )
apply(auto simp: const-def)
apply(erule-tac x= $\tau$  in allE)
apply(erule-tac x= $\tau$  in allE)
apply(erule-tac x= $\tau'$  in allE)
apply(erule-tac x= $\tau'$  in allE)
apply simp
apply(erule-tac x= $\tau$  in allE)
apply(erule-tac x= $\tau$  in allE)
apply(erule-tac x= $\tau'$  in allE)
apply(erule-tac x= $\tau'$  in allE)
by simp

```

```

lemmas const-ss = const-bot const-null const-invalid const-false const-true const-lam

```

*const-defined const-valid const-StrongEq const-OclNot const-OclAnd
const-OclOr const-OclImplies const-OclIf
const-HOL-if const-HOL-and const-HOL-eq*

Miscellaneous: Overloading the syntax of “bottom”

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

end

theory *Featherweight-OCL-Assert*

imports *Main*

keywords *Assert* :: *thy-decl*

and *Assert-local* :: *thy-decl*

begin

end

theory *UML-PropertyProfiles*

imports *UML-Logic Featherweight-OCL-Assert*

begin

2.2. Property Profiles for OCL Operators via Isabelle Locales

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

2.2.1. Property Profiles for Monadic Operators

locale *profile-mono-scheme-defined* =

fixes $f :: ('A, 'α :: null) val \Rightarrow ('A, 'β :: null) val$

fixes g

assumes *def-scheme*: $(f\ x) \equiv \lambda \tau. \text{if } (\delta\ x)\ \tau = \text{true } \tau \text{ then } g\ (x\ \tau) \text{ else } \text{invalid } \tau$

begin

lemma *strict[simp,code-unfold]*: $f\ \text{invalid} = \text{invalid}$

by(*rule ext, simp add: def-scheme true-def false-def*)

lemma *null-strict[simp,code-unfold]*: $f\ \text{null} = \text{invalid}$

by(*rule ext, simp add: def-scheme true-def false-def*)

lemma *cp0* : $f\ X\ \tau = f\ (\lambda \cdot. X\ \tau)\ \tau$

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

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

by(*rule cpI1[of f], intro allI, rule cp0, simp-all*)

end

locale *profile-mono-schemeV* =

```

fixes  $f :: ('\mathfrak{A}, '\alpha :: \text{null}) \text{val} \Rightarrow ('\mathfrak{A}, '\beta :: \text{null}) \text{val}$ 
fixes  $g$ 
assumes  $\text{def-scheme}: (f\ x) \equiv \lambda \tau. \text{if } (v\ x)\ \tau = \text{true}\ \tau \text{ then } g\ (x\ \tau) \text{ else invalid } \tau$ 
begin
  lemma  $\text{strict}[simp, \text{code-unfold}]: f\ \text{invalid} = \text{invalid}$ 
  by( $\text{rule ext, simp add: def-scheme true-def false-def}$ )

  lemma  $\text{cp0} : f\ X\ \tau = f\ (\lambda \cdot. X\ \tau)\ \tau$ 
  by( $\text{simp add: def-scheme cp-valid[symmetric]}$ )

  lemma  $\text{cp}[simp, \text{code-unfold}] : \text{cp}\ P \Longrightarrow \text{cp}\ (\lambda X. f\ (P\ X))$ 
  by( $\text{rule cpI1[of f], intro allI, rule cp0, simp-all}$ )

end

locale  $\text{profile-mono}_a = \text{profile-mono-scheme-defined} +$ 
  assumes  $\bigwedge x. x \neq \text{bot} \Longrightarrow x \neq \text{null} \Longrightarrow g\ x \neq \text{bot}$ 
begin

  lemma  $\text{const}[simp, \text{code-unfold}] :$ 
    assumes  $C1 : \text{const}\ X$ 
    shows  $\text{const}(f\ X)$ 
  proof –
    have  $\text{const-g} : \text{const}\ (\lambda \tau. g\ (X\ \tau))$  by( $\text{insert } C1, \text{auto simp: const-def, metis}$ )
    show  $?thesis$  by( $\text{simp-all add : def-scheme const-ss } C1\ \text{const-g}$ )
  qed

end

locale  $\text{profile-mono0} = \text{profile-mono-scheme-defined} +$ 
  assumes  $\text{def-body}: \bigwedge x. x \neq \text{bot} \Longrightarrow x \neq \text{null} \Longrightarrow g\ x \neq \text{bot} \wedge g\ x \neq \text{null}$ 

sublocale  $\text{profile-mono0} < \text{profile-mono}_a$ 
by( $\text{unfold-locales, simp add: def-scheme, simp add: def-body}$ )

context  $\text{profile-mono0}$ 
begin
  lemma  $\text{def-homo}[simp, \text{code-unfold}]: \delta(f\ x) = (\delta\ x)$ 
  apply( $\text{rule ext, rename-tac } \tau, \text{subst foundation22[symmetric]}$ )
  apply( $\text{case-tac } \neg(\tau \models \delta\ x), \text{simp add: defined-split, elim disjE}$ )
  apply( $\text{erule StrongEq-L-subst2-rev, simp, simp}$ )
  apply( $\text{erule StrongEq-L-subst2-rev, simp, simp}$ )
  apply( $\text{simp}$ )
  apply( $\text{rule foundation13[THEN iffD2, THEN StrongEq-L-subst2-rev, where } y = \delta\ x]$ )
  apply( $\text{simp-all add: def-scheme}$ )
  apply( $\text{simp add: OclValid-def}$ )
  by( $\text{auto simp: foundation13 StrongEq-def false-def true-def defined-def bot-fun-def null-fun-def def-body}$ 
     $\text{split: if-split-asm}$ )

  lemma  $\text{def-valid-then-def}: v(f\ x) = (\delta(f\ x))$ 
  apply( $\text{rule ext, rename-tac } \tau, \text{subst foundation22[symmetric]}$ )
  apply( $\text{case-tac } \neg(\tau \models \delta\ x), \text{simp add: defined-split, elim disjE}$ )
  apply( $\text{erule StrongEq-L-subst2-rev, simp, simp}$ )
  apply( $\text{erule StrongEq-L-subst2-rev, simp, simp}$ )
  apply  $\text{simp}$ 
  apply( $\text{simp-all add: def-scheme}$ )
  apply( $\text{simp add: OclValid-def valid-def, subst cp-StrongEq}$ )
  apply( $\text{subst } (2)\ \text{cp-defined, simp, simp add: cp-defined[symmetric]}$ )

```

```

by(auto simp: foundation13 StrongEq-def false-def true-def defined-def bot-fun-def null-fun-def def-body
split: if-split-asm)
end

```

2.2.2. Property Profiles for Single

```

locale profile-single =
  fixes d:: ('A, 'a::null)val => 'A Boolean
  assumes d-strict[simp, code-unfold]: d invalid = false
  assumes d-cp0: d X τ = d (λ -. X τ) τ
  assumes d-const[simp, code-unfold]: const X ==> const (d X)

```

2.2.3. Property Profiles for Binary Operators

```

definition bin' f g d_x d_y X Y =
  (f X Y = (λ τ. if (d_x X) τ = true τ ∧ (d_y Y) τ = true τ
    then g X Y τ
    else invalid τ ))

```

```

definition bin f g = bin' f (λ X Y τ. g (X τ) (Y τ))

```

```

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

```

```

locale profile-bin-scheme =
  fixes d_x:: ('A, 'a::null)val => 'A Boolean
  fixes d_y:: ('A, 'b::null)val => 'A Boolean
  fixes f:: ('A, 'a::null)val => ('A, 'b::null)val => ('A, 'c::null)val
  fixes g
  assumes d_x' : profile-single d_x
  assumes d_y' : profile-single d_y
  assumes d_x-d_y-homo[simp, code-unfold]: cp (f X) ==>
    cp (λ x. f x Y) ==>
    f X invalid = invalid ==>
    f invalid Y = invalid ==>
    (¬ (τ ⊨ d_x X) ∨ ¬ (τ ⊨ d_y Y)) ==>
    τ ⊨ (δ f X Y ≜ (d_x X and d_y Y))
  assumes def-scheme''[simplified]: bin f g d_x d_y X Y
  assumes 1: τ ⊨ d_x X ==> τ ⊨ d_y Y ==> τ ⊨ δ f X Y

```

```

begin

```

```

  interpretation d_x : profile-single d_x by (rule d_x')
  interpretation d_y : profile-single d_y by (rule d_y')

```

```

  lemma strict1[simp, code-unfold]: f invalid y = invalid
  by(rule ext, simp add: def-scheme'' true-def false-def)

```

```

  lemma strict2[simp, code-unfold]: f x invalid = invalid
  by(rule ext, simp add: def-scheme'' true-def false-def)

```

```

  lemma cp0 : f X Y τ = f (λ -. X τ) (λ -. Y τ) τ
  by(simp add: def-scheme'' d_x.d-cp0[symmetric] d_y.d-cp0[symmetric] cp-defined[symmetric])

```

```

  lemma cp[simp, code-unfold] : cp P ==> cp Q ==> cp (λ X. f (P X) (Q X))
  by(rule cpI2[of f], intro allI, rule cp0, simp-all)

```

```

  lemma def-homo[simp, code-unfold]: δ(f x y) = (d_x x and d_y y)
  apply(rule ext, rename-tac τ, subst foundation22[symmetric])
  apply(case-tac ¬(τ ⊨ d_x x), simp)

```

```

apply(case-tac  $\neg(\tau \models d_y y)$ , simp)
apply(simp)
apply(rule foundation13[THEN iffD2,THEN StrongEq-L-subst2-rev, where  $y = d_x x$ ])
  apply(simp-all)
apply(rule foundation13[THEN iffD2,THEN StrongEq-L-subst2-rev, where  $y = d_y y$ ])
  apply(simp-all add: 1 foundation13)
done

```

```

lemma def-valid-then-def:  $v(f x y) = (\delta(f x y))$ 
  apply(rule ext, rename-tac  $\tau$ )
  apply(simp-all add: valid-def defined-def def-scheme''
    true-def false-def invalid-def
    null-fun-def null-option-def bot-fun-def)
  by (metis 1 OclValid-def def-scheme'' foundation16 true-def)

```

```

lemma defined-args-valid:  $(\tau \models \delta(f x y)) = ((\tau \models d_x x) \wedge (\tau \models d_y y))$ 
  by(simp add: foundation10')

```

```

lemma const[simp,code-unfold] :
  assumes  $C1 : \text{const } X$  and  $C2 : \text{const } Y$ 
  shows  $\text{const}(f X Y)$ 
proof –
  have const-g :  $\text{const}(\lambda\tau. g(X \tau)(Y \tau))$ 
    by(insert C1 C2, auto simp:const-def, metis)
  show ?thesis
  by(simp-all add : def-scheme'' const-ss C1 C2 const-g)
qed

```

end

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

```

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

```

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

```

proof –
  interpret  $d_y : \text{profile-single } d_y$  by (rule d_y)
  show profile-bin-scheme defined d_y f g
  apply(unfold-locales)
  apply(simp)+
  apply(subst cp-defined, simp)

```

```

    apply(rule const-defined, simp)
    apply(simp add:defined-split, elim disjE)
    apply(erule StrongEq-L-subst2-rev, simp, simp)+
    apply(simp)
    apply(simp add: def-scheme')
    apply(simp add: defined-def OclValid-def false-def true-def
      bot-fun-def null-fun-def def-scheme' split: if-split-asm, rule def-body')
  by(simp add: true-def)+
qed

locale profile-bina-d =
  fixes f::('A,'a::null)val ⇒ ('A,'b::null)val ⇒ ('A,'c::null)val
  fixes g
  assumes def-scheme[simplified]: bin f g defined defined X Y
  assumes def-body: ∧ x y. x≠bot ⇒ x≠null ⇒ y≠bot ⇒ y≠null ⇒
    g x y ≠ bot ∧ g x y ≠ null
begin
  lemma strict4[simp,code-unfold]: f x null = invalid
  by(rule ext, simp add: def-scheme true-def false-def)
end

sublocale profile-bina-d < profile-bin-scheme-defined defined
apply(unfold-locales)
  apply(simp)+
  apply(subst cp-defined, simp)+
  apply(rule const-defined, simp)+
  apply(simp add:defined-split, elim disjE)
  apply(erule StrongEq-L-subst2-rev, simp, simp)+
  apply(simp add: def-scheme)
apply(simp add: defined-def OclValid-def false-def true-def bot-fun-def null-fun-def def-scheme)
apply(rule def-body, simp-all add: true-def false-def split:if-split-asm)
done

locale profile-bina-v =
  fixes f::('A,'a::null)val ⇒ ('A,'b::null)val ⇒ ('A,'c::null)val
  fixes g
  assumes def-scheme[simplified]: bin f g defined valid X Y
  assumes def-body: ∧ x y. x≠bot ⇒ x≠null ⇒ y≠bot ⇒ g x y ≠ bot ∧ g x y ≠ null

sublocale profile-bina-v < profile-bin-scheme-defined valid
apply(unfold-locales)
  apply(simp)
  apply(subst cp-valid, simp)
  apply(rule const-valid, simp)
  apply(simp add:foundation18'')
  apply(erule StrongEq-L-subst2-rev, simp, simp)
  apply(simp add: def-scheme)
  by (metis OclValid-def def-body foundation18')

locale profile-binStrongEq-v-v =
  fixes f :: ('A,'α::null)val ⇒ ('A,'α::null)val ⇒ ('A) Boolean
  assumes def-scheme[simplified]: bin' f StrongEq valid valid X Y

sublocale profile-binStrongEq-v-v < profile-bin-scheme valid valid f λx y. ⊥x = y⊥
  apply(unfold-locales)
  apply(simp)
  apply(subst cp-valid, simp)

```

```

apply (simp add: const-valid)
apply (metis (opaque-lifting, mono-tags) OclValid-def def-scheme defined5 defined6 defined-and-I foundation1
foundation10' foundation16' foundation18 foundation21 foundation22 foundation9)
apply(simp add: def-scheme, subst StrongEq-def, simp)
by (metis OclValid-def def-scheme defined7 foundation16)

```

context *profile-bin_{StrongEq^{-v}-v}*

begin

```

lemma idem[simp,code-unfold]: f null null = true
by(rule ext, simp add: def-scheme true-def false-def)

```

```

lemma defargs:  $\tau \models f x y \implies (\tau \models v x) \wedge (\tau \models v y)$ 
by(simp add: def-scheme OclValid-def true-def invalid-def valid-def bot-option-def
split: bool.split-asm HOL.if-split-asm)

```

```

lemma defined-args-valid' :  $\delta (f x y) = (v x \text{ and } v y)$ 
by(auto intro!: transform2-rev defined-and-I simp:foundation10 defined-args-valid)

```

```

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

```

```

lemma sym :  $\tau \models (f x y) \implies \tau \models (f y x)$ 
apply(case-tac  $\tau \models v x$ )
apply(auto simp: def-scheme OclValid-def)
by(fold OclValid-def, erule StrongEq-L-sym)

```

```

lemma symmetric :  $(f x y) = (f y x)$ 
by(rule ext, rename-tac  $\tau$ , auto simp: def-scheme StrongEq-sym)

```

```

lemma trans :  $\tau \models (f x y) \implies \tau \models (f y z) \implies \tau \models (f x z)$ 
apply(case-tac  $\tau \models v x$ )
apply(case-tac  $\tau \models v y$ )
apply(auto simp: def-scheme OclValid-def)
by(fold OclValid-def, auto elim: StrongEq-L-trans)

```

```

lemma StrictRefEq-vs-StrongEq:  $\tau \models (v x) \implies \tau \models (v y) \implies (\tau \models ((f x y) \hat{=} (x \hat{=} y)))$ 
apply(simp add: def-scheme OclValid-def)
apply(subst cp-StrongEq[of - (x  $\hat{=}$  y)])
by simp

```

end

locale *profile-bin_v-v* =

```

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

```

sublocale *profile-bin_v-v* < *profile-bin-scheme valid valid*

```

apply(unfold-locales)
apply(simp, subst cp-valid, simp, rule const-valid, simp)+
apply (metis (opaque-lifting, mono-tags) OclValid-def def-scheme defined5 defined6 defined-and-I
foundation1 foundation10' foundation16' foundation18 foundation21 foundation22 foundation9)
apply(simp add: def-scheme)

```

```

apply(simp add: defined-def OclValid-def false-def true-def
        bot-fun-def null-fun-def def-scheme split: if-split-asm, rule def-body)
by (metis OclValid-def foundation18' true-def)+

end

```

```

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

```

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

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

```

overloading StrictRefEq  $\equiv$  StrictRefEq :: [ $('\mathfrak{A})$ Boolean,  $('\mathfrak{A})$ Boolean]  $\Rightarrow$   $('\mathfrak{A})$ Boolean
begin

```

```

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

```

```

end

```

which implies elementary properties like:

```

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

```

```

by(simp add:StrictRefEqBoolean)

```

```

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

```

```

by(simp add:StrictRefEqBoolean)

```

```

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

```

```

apply(rule ext, simp add: StrictRefEqBoolean StrongEq-def false-def)

```

```

by (metis drop.simps cp-valid false-def is-none-code(2) Option.is-none-def valid4
      bot-option-def null-fun-def null-option-def)

```

```

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

```

```

apply(rule ext, simp add: StrictRefEqBoolean StrongEq-def false-def)

```

```

by(simp add: true-def bot-option-def null-fun-def null-option-def)

```

```

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

```

```

apply(rule ext, simp add: StrictRefEqBoolean StrongEq-def false-def)

```

```

by(metis drop.simps cp-valid false-def is-none-code(2) Option.is-none-def valid4
      bot-option-def null-fun-def null-option-def )

```

```

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

```

```

apply(rule ext, simp add: StrictRefEqBoolean StrongEq-def false-def)

```

```

by(simp add: true-def bot-option-def null-fun-def null-option-def)

```

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

```

interpretation StrictRefEqBoolean : profile-binStrongEqv-v  $\lambda x y$ . ( $x::(')\mathfrak{A})$ Boolean  $\doteq$   $y$ 
  by unfold-locales (auto simp:StrictRefEqBoolean)

```

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

```

lemma (invalid  $\doteq$  false) = invalid by(simp)

```

```

lemma (invalid  $\doteq$  true) = invalid by(simp)

```

```

lemma (false ≐ invalid) = invalid by(simp)
lemma (true ≐ invalid) = invalid by(simp)
lemma ((invalid::('A)Boolean) ≐ invalid) = invalid by(simp)

```

Thus, the weak equality is *not* reflexive.

2.2.5. Test Statements on Boolean Operations.

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

Elementary computations on Boolean

```

Assert τ ⊨ v(true)
Assert τ ⊨ δ(false)
Assert τ ⊭ δ(null)
Assert τ ⊭ δ(invalid)
Assert τ ⊨ v((null::('A)Boolean))
Assert τ ⊭ v(invalid)
Assert τ ⊨ (true and true)
Assert τ ⊨ (true and true ≐ true)
Assert τ ⊨ ((null or null) ≐ null)
Assert τ ⊨ ((null or null) ≐ null)
Assert τ ⊨ ((true ≐ false) ≐ false)
Assert τ ⊨ ((invalid ≐ false) ≐ false)
Assert τ ⊨ ((invalid ≐ false) ≐ invalid)
Assert τ ⊨ (true <> false)
Assert τ ⊨ (false <> true)

```

end

```

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

```

2.3. Basic Type Void: Operations

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

2.3.1. Fundamental Properties on Voids: Strict Equality

Definition

```

instantiation Voidbase :: bot

```

```

begin

```

```

  definition bot-Void-def: (bot-class.bot :: Voidbase) ≐ Abs-Voidbase None

```

```

  instance proof show ∃ x:: Voidbase. x ≠ bot
    apply(rule-tac x=Abs-Voidbase _None_ in exI)
    apply(simp add:bot-Void-def, subst Abs-Voidbase-inject)
    apply(simp-all add: null-option-def bot-option-def)
    done

```

```

  qed

```

end

instantiation $\text{Void}_{base} :: \text{null}$

begin

definition $\text{null-void-def} : (\text{null} :: \text{Void}_{base}) \equiv \text{Abs-void}_{base} _ \text{None} _$

instance proof show $(\text{null} :: \text{Void}_{base}) \neq \text{bot}$

apply $(\text{simp add: null-void-def bot-void-def, subst Abs-void}_{base}\text{-inject})$

apply $(\text{simp-all add: null-option-def bot-option-def})$

done

qed

end

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

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

begin

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

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

end

Property proof in terms of $\text{profile-bin}_{\text{StrongEq-v-v}}$

interpretation $\text{StrictRefEq}_{\text{void}} : \text{profile-bin}_{\text{StrongEq-v-v}} \lambda x y. (x :: (\mathcal{A}) \text{Void}) \doteq y$
by $\text{unfold-locales (auto simp: StrictRefEq}_{\text{void}})$

2.3.2. Basic Void Constants

2.3.3. Validity and Definedness Properties

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

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

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

apply $(\text{simp add: defined-def true-def}$
 $\text{bot-fun-def bot-option-def})$

apply $(\text{rule ext, simp split, intro conjI impI})$

by $(\text{simp add: bot-void-def})$

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

apply $(\text{simp add: valid-def true-def}$
 $\text{bot-fun-def bot-option-def})$

apply $(\text{rule ext, simp split, intro conjI impI})$

by $(\text{simp add: bot-void-def})$

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

apply $(\text{simp add: defined-def true-def}$
 $\text{bot-fun-def bot-option-def null-fun-def null-option-def})$

apply $(\text{rule ext, simp split, intro conjI impI})$

by $(\text{simp add: null-void-def})$

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

apply $(\text{simp add: valid-def true-def}$
 $\text{bot-fun-def bot-option-def})$

apply $(\text{rule ext, simp split, intro conjI impI})$

by $(\text{metis null-void-def null-is-valid, simp add: true-def})$

2.3.4. Test Statements

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

end

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

2.4. Basic Type Integer: Operations

2.4.1. Fundamental Predicates on Integers: Strict Equality

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

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

begin

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

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

end

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

interpretation $\text{StrictRefEq}_{\text{Integer}} : \text{profile-bin}_{\text{StrongEq}^{-v^{-v}}} \lambda x\ y. (x::(\mathcal{A})\text{Integer}) \doteq y$
by unfold-locales ($\text{auto simp: StrictRefEq}_{\text{Integer}}$)

2.4.2. Basic Integer Constants

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

definition $\text{OclInt0} :: (\mathcal{A})\text{Integer} \langle \mathbf{0} \rangle$ where $\mathbf{0} = (\lambda - . \underline{_0}::\text{int}_{\perp})$

definition $\text{OclInt1} :: (\mathcal{A})\text{Integer} \langle \mathbf{1} \rangle$ where $\mathbf{1} = (\lambda - . \underline{_1}::\text{int}_{\perp})$

definition $\text{OclInt2} :: (\mathcal{A})\text{Integer} \langle \mathbf{2} \rangle$ where $\mathbf{2} = (\lambda - . \underline{_2}::\text{int}_{\perp})$

Etc.

definition $\text{OclInt3} :: (\mathcal{A})\text{Integer} \langle \mathbf{3} \rangle$ where $\mathbf{3} = (\lambda - . \underline{_3}::\text{int}_{\perp})$

definition $\text{OclInt4} :: (\mathcal{A})\text{Integer} \langle \mathbf{4} \rangle$ where $\mathbf{4} = (\lambda - . \underline{_4}::\text{int}_{\perp})$

definition $\text{OclInt5} :: (\mathcal{A})\text{Integer} \langle \mathbf{5} \rangle$ where $\mathbf{5} = (\lambda - . \underline{_5}::\text{int}_{\perp})$

definition $\text{OclInt6} :: (\mathcal{A})\text{Integer} \langle \mathbf{6} \rangle$ where $\mathbf{6} = (\lambda - . \underline{_6}::\text{int}_{\perp})$

definition $\text{OclInt7} :: (\mathcal{A})\text{Integer} \langle \mathbf{7} \rangle$ where $\mathbf{7} = (\lambda - . \underline{_7}::\text{int}_{\perp})$

definition $\text{OclInt8} :: (\mathcal{A})\text{Integer} \langle \mathbf{8} \rangle$ where $\mathbf{8} = (\lambda - . \underline{_8}::\text{int}_{\perp})$

definition $\text{OclInt9} :: (\mathcal{A})\text{Integer} \langle \mathbf{9} \rangle$ where $\mathbf{9} = (\lambda - . \underline{_9}::\text{int}_{\perp})$

definition $\text{OclInt10} :: (\mathcal{A})\text{Integer} \langle \mathbf{10} \rangle$ where $\mathbf{10} = (\lambda - . \underline{_10}::\text{int}_{\perp})$

2.4.3. 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 $[\text{simp}, \text{code-unfold}] : \delta(\lambda - . \underline{_n}_{\perp}) = \text{true}$

by ($\text{simp add: defined-def true-def}$
 $\text{bot-fun-def bot-option-def null-fun-def null-option-def}$)

lemma $[simp,code-unfold]: v (\lambda-. \llcorner n \llcorner) = true$
by $(simp\ add:valid-def\ true-def\ bot-fun-def\ bot-option-def)$

lemma $[simp,code-unfold]: \delta\ 0 = true$ **by** $(simp\ add:OclInt0-def)$
lemma $[simp,code-unfold]: v\ 0 = true$ **by** $(simp\ add:OclInt0-def)$
lemma $[simp,code-unfold]: \delta\ 1 = true$ **by** $(simp\ add:OclInt1-def)$
lemma $[simp,code-unfold]: v\ 1 = true$ **by** $(simp\ add:OclInt1-def)$
lemma $[simp,code-unfold]: \delta\ 2 = true$ **by** $(simp\ add:OclInt2-def)$
lemma $[simp,code-unfold]: v\ 2 = true$ **by** $(simp\ add:OclInt2-def)$
lemma $[simp,code-unfold]: \delta\ 6 = true$ **by** $(simp\ add:OclInt6-def)$
lemma $[simp,code-unfold]: v\ 6 = true$ **by** $(simp\ add:OclInt6-def)$
lemma $[simp,code-unfold]: \delta\ 8 = true$ **by** $(simp\ add:OclInt8-def)$
lemma $[simp,code-unfold]: v\ 8 = true$ **by** $(simp\ add:OclInt8-def)$
lemma $[simp,code-unfold]: \delta\ 9 = true$ **by** $(simp\ add:OclInt9-def)$
lemma $[simp,code-unfold]: v\ 9 = true$ **by** $(simp\ add:OclInt9-def)$

2.4.4. Arithmetical Operations

Definition

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

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

definition $OclAdd_{Integer} :: ('A)Integer \Rightarrow ('A)Integer \Rightarrow ('A)Integer$ (**infix** $\langle +_{int} \rangle$ 40)
where $x +_{int} y \equiv \lambda \tau. \text{if } (\delta x) \tau = true \ \tau \wedge (\delta y) \tau = true \ \tau$
 $\text{then } \llcorner^{\tau} x \llcorner^{\tau} + \llcorner^{\tau} y \llcorner^{\tau} \llcorner$
 $\text{else } invalid \ \tau$

interpretation $OclAdd_{Integer} : profile-bin_{d-d} (+_{int}) \lambda x y. \llcorner^{\tau} x \llcorner^{\tau} + \llcorner^{\tau} y \llcorner^{\tau} \llcorner$
by $unfold-locales (auto\ simp:OclAdd_{Integer}-def\ bot-option-def\ null-option-def)$

definition $OclMinus_{Integer} :: ('A)Integer \Rightarrow ('A)Integer \Rightarrow ('A)Integer$ (**infix** $\langle -_{int} \rangle$ 41)
where $x -_{int} y \equiv \lambda \tau. \text{if } (\delta x) \tau = true \ \tau \wedge (\delta y) \tau = true \ \tau$
 $\text{then } \llcorner^{\tau} x \llcorner^{\tau} - \llcorner^{\tau} y \llcorner^{\tau} \llcorner$
 $\text{else } invalid \ \tau$

interpretation $OclMinus_{Integer} : profile-bin_{d-d} (-_{int}) \lambda x y. \llcorner^{\tau} x \llcorner^{\tau} - \llcorner^{\tau} y \llcorner^{\tau} \llcorner$
by $unfold-locales (auto\ simp:OclMinus_{Integer}-def\ bot-option-def\ null-option-def)$

definition $OclMult_{Integer} :: ('A)Integer \Rightarrow ('A)Integer \Rightarrow ('A)Integer$ (**infix** $\langle *_{int} \rangle$ 45)
where $x *_{int} y \equiv \lambda \tau. \text{if } (\delta x) \tau = true \ \tau \wedge (\delta y) \tau = true \ \tau$
 $\text{then } \llcorner^{\tau} x \llcorner^{\tau} * \llcorner^{\tau} y \llcorner^{\tau} \llcorner$
 $\text{else } invalid \ \tau$

interpretation $OclMult_{Integer} : profile-bin_{d-d} OclMult_{Integer} \lambda x y. \llcorner^{\tau} x \llcorner^{\tau} * \llcorner^{\tau} y \llcorner^{\tau} \llcorner$
by $unfold-locales (auto\ simp:OclMult_{Integer}-def\ bot-option-def\ null-option-def)$

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

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

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

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

interpretation $OclLess_{Integer} : \text{profile-bin}_{d-d} (<_{int}) \lambda x y. \llcorner^{\lceil x \rceil} \tau^{\lceil} < \lceil^{\lceil y \rceil} \tau^{\lceil} \llcorner$
by *unfold-locales (auto simp: OclLess_{Integer}-def bot-option-def null-option-def)*

definition $OclLe_{Integer} :: ('A)Integer \Rightarrow ('A)Integer \Rightarrow ('A)Boolean$ (**infix** $\langle \leq_{int} \rangle$ 35)
where $x \leq_{int} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
 then $\llcorner^{\lceil x \rceil} \tau^{\lceil} \leq \lceil^{\lceil y \rceil} \tau^{\lceil} \llcorner$
 else invalid τ

interpretation $OclLe_{Integer} : \text{profile-bin}_{d-d} (\leq_{int}) \lambda x y. \llcorner^{\lceil x \rceil} \tau^{\lceil} \leq \lceil^{\lceil y \rceil} \tau^{\lceil} \llcorner$
by *unfold-locales (auto simp: OclLe_{Integer}-def bot-option-def null-option-def)*

Basic Properties

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

Execution with Invalid or Null or Zero as Argument

lemma $OclAdd_{Integer}\text{-zero1}$ [*simp, code-unfold*] :
 $(x +_{int} \mathbf{0}) = (\text{if } v \ x \ \text{and not } (\delta x) \ \text{then invalid else } x \ \text{endif})$
proof (*rule ext, rename-tac τ , case-tac (v x and not (δx)) $\tau = \text{true } \tau$*)
fix τ **show** $(v \ x \ \text{and not } (\delta x)) \tau = \text{true } \tau \implies$
 $(x +_{int} \mathbf{0}) \tau = (\text{if } v \ x \ \text{and not } (\delta x) \ \text{then invalid else } x \ \text{endif}) \tau$
apply (*subst OclIf-true', simp add: OclValid-def*)
by (*metis OclAdd_{Integer}-def OclNot-defargs OclValid-def foundation5 foundation9*)
next fix τ
have $A: \bigwedge \tau. (\tau \models \text{not } (v \ x \ \text{and not } (\delta x))) = (x \tau = \text{invalid } \tau \vee \tau \models \delta x)$
by (*metis OclNot-not OclOr-def defined5 defined6 defined-not-I foundation11 foundation18'*
 foundation6 foundation7 foundation9 invalid-def)
have $B: \tau \models \delta x \implies \llcorner^{\lceil x \rceil} \tau^{\lceil} \llcorner = x \ \tau$
apply (*cases x τ , metis bot-option-def foundation16*)
apply (*rename-tac x' , case-tac x' , metis bot-option-def foundation16 null-option-def*)
by (*simp*)
show $(x +_{int} \mathbf{0}) \tau = (\text{if } v \ x \ \text{and not } (\delta x) \ \text{then invalid else } x \ \text{endif}) \tau$
when $\tau \models \text{not } (v \ x \ \text{and not } (\delta x))$
apply (*insert that, subst OclIf-false', simp, simp add: A, auto simp: OclAdd_{Integer}-def OclInt0-def*)

apply (*simp add: foundation16 '[simplified OclValid-def]*)
apply (*simp add: B*)
by (*simp add: OclValid-def*)
qed (*metis OclValid-def defined5 defined6 defined-and-I defined-not-I foundation9*)

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

by(subst OclAddInteger-commute, simp)

2.4.5. Test Statements

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

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

lemma integer-non-null [simp]: $((\lambda-. \sqcup n_{\sqcup}) \doteq (\text{null}::('A)Integer)) = \text{false}$
by(rule ext,auto simp: StrictRefEqInteger valid-def
bot-fun-def bot-option-def null-fun-def null-option-def StrongEq-def)

lemma null-non-integer [simp]: $((\text{null}::('A)Integer) \doteq (\lambda-. \sqcup n_{\sqcup})) = \text{false}$
by(rule ext,auto simp: StrictRefEqInteger valid-def
bot-fun-def bot-option-def null-fun-def null-option-def StrongEq-def)

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

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

Elementary computations on Integer

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

Assert $\tau \models \mathbf{1} <> \mathbf{2}$

Assert $\tau \models \mathbf{2} <> \mathbf{1}$

Assert $\tau \models \mathbf{2} \doteq \mathbf{2}$

Assert $\tau \models v \ \mathbf{4}$

Assert $\tau \models \delta \ \mathbf{4}$

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

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

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

Assert $\tau \models (\mathbf{4} \triangleq \mathbf{4})$

Assert $\tau \not\models (\mathbf{9} \triangleq \mathbf{10})$

Assert $\tau \not\models (\text{invalid} \triangleq \mathbf{10})$

Assert $\tau \not\models (\text{null} \triangleq \mathbf{10})$

Assert $\tau \not\models (\text{invalid} \doteq (\text{invalid}::('A)Integer))$

```

Assert  $\tau \mid \neq v$  (invalid  $\doteq$  (invalid::('A)Integer))
Assert  $\tau \mid \neq$  (invalid  $\langle \rangle$  (invalid::('A)Integer))
Assert  $\tau \mid \neq v$  (invalid  $\langle \rangle$  (invalid::('A)Integer))
Assert  $\tau \models$  (null  $\doteq$  (null::('A)Integer))
Assert  $\tau \models$  (null  $\doteq$  (null::('A)Integer))
Assert  $\tau \models$  (4  $\doteq$  4)
Assert  $\tau \mid \neq$  (4  $\langle \rangle$  4)
Assert  $\tau \mid \neq$  (4  $\doteq$  10)
Assert  $\tau \models$  (4  $\langle \rangle$  10)
Assert  $\tau \mid \neq$  (0  $\langle_{int}$  null)
Assert  $\tau \mid \neq$  ( $\delta$  (0  $\langle_{int}$  null))

```

end

```

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

```

2.5. Basic Type Real: Operations

2.5.1. Fundamental Predicates on Reals: Strict Equality

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

```

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

```

begin

```

definition StrictRefEqReal [code-unfold] :

```

```

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

```

end

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

```

interpretation StrictRefEqReal : profile-binStrongEq-v-v  $\lambda x y$ . ( $x::(\mathcal{A})Real$ )  $\doteq$   $y$ 
    by unfold-locales (auto simp: StrictRefEqReal)

```

2.5.2. Basic Real Constants

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

```

definition OclReal0 :: ( $\mathcal{A}$ )Real  $\langle$ 0.0 $\rangle$  where 0.0 = ( $\lambda$  . .  $\underline{\underline{0}}::real_{\perp}$ )

```

```

definition OclReal1 :: ( $\mathcal{A}$ )Real  $\langle$ 1.0 $\rangle$  where 1.0 = ( $\lambda$  . .  $\underline{\underline{1}}::real_{\perp}$ )

```

```

definition OclReal2 :: ( $\mathcal{A}$ )Real  $\langle$ 2.0 $\rangle$  where 2.0 = ( $\lambda$  . .  $\underline{\underline{2}}::real_{\perp}$ )

```

Etc.

```

definition OclReal3 :: ( $\mathcal{A}$ )Real  $\langle$ 3.0 $\rangle$  where 3.0 = ( $\lambda$  . .  $\underline{\underline{3}}::real_{\perp}$ )

```

```

definition OclReal4 :: ( $\mathcal{A}$ )Real  $\langle$ 4.0 $\rangle$  where 4.0 = ( $\lambda$  . .  $\underline{\underline{4}}::real_{\perp}$ )

```

```

definition OclReal5 :: ( $\mathcal{A}$ )Real  $\langle$ 5.0 $\rangle$  where 5.0 = ( $\lambda$  . .  $\underline{\underline{5}}::real_{\perp}$ )

```

```

definition OclReal6 :: ( $\mathcal{A}$ )Real  $\langle$ 6.0 $\rangle$  where 6.0 = ( $\lambda$  . .  $\underline{\underline{6}}::real_{\perp}$ )

```

```

definition OclReal7 :: ( $\mathcal{A}$ )Real  $\langle$ 7.0 $\rangle$  where 7.0 = ( $\lambda$  . .  $\underline{\underline{7}}::real_{\perp}$ )

```

```

definition OclReal8 :: ( $\mathcal{A}$ )Real  $\langle$ 8.0 $\rangle$  where 8.0 = ( $\lambda$  . .  $\underline{\underline{8}}::real_{\perp}$ )

```

```

definition OclReal9 :: ( $\mathcal{A}$ )Real  $\langle$ 9.0 $\rangle$  where 9.0 = ( $\lambda$  . .  $\underline{\underline{9}}::real_{\perp}$ )

```

```

definition OclReal10 :: ( $\mathcal{A}$ )Real  $\langle$ 10.0 $\rangle$  where 10.0 = ( $\lambda$  . .  $\underline{\underline{10}}::real_{\perp}$ )

```

```

definition OclRealpi :: ( $\mathcal{A}$ )Real  $\langle$  $\pi$  $\rangle$  where  $\pi$  = ( $\lambda$  . .  $\underline{\underline{\pi}}_{\perp}$ )

```

2.5.3. Validity and Definedness Properties

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

lemma $v(\text{null}::('A)\text{Real}) = \text{true}$ **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*]: $\delta \mathbf{0.0} = \text{true}$ **by**(*simp add:OclReal0-def*)
lemma [*simp,code-unfold*]: $v \mathbf{0.0} = \text{true}$ **by**(*simp add:OclReal0-def*)
lemma [*simp,code-unfold*]: $\delta \mathbf{1.0} = \text{true}$ **by**(*simp add:OclReal1-def*)
lemma [*simp,code-unfold*]: $v \mathbf{1.0} = \text{true}$ **by**(*simp add:OclReal1-def*)
lemma [*simp,code-unfold*]: $\delta \mathbf{2.0} = \text{true}$ **by**(*simp add:OclReal2-def*)
lemma [*simp,code-unfold*]: $v \mathbf{2.0} = \text{true}$ **by**(*simp add:OclReal2-def*)
lemma [*simp,code-unfold*]: $\delta \mathbf{6.0} = \text{true}$ **by**(*simp add:OclReal6-def*)
lemma [*simp,code-unfold*]: $v \mathbf{6.0} = \text{true}$ **by**(*simp add:OclReal6-def*)
lemma [*simp,code-unfold*]: $\delta \mathbf{8.0} = \text{true}$ **by**(*simp add:OclReal8-def*)
lemma [*simp,code-unfold*]: $v \mathbf{8.0} = \text{true}$ **by**(*simp add:OclReal8-def*)
lemma [*simp,code-unfold*]: $\delta \mathbf{9.0} = \text{true}$ **by**(*simp add:OclReal9-def*)
lemma [*simp,code-unfold*]: $v \mathbf{9.0} = \text{true}$ **by**(*simp add:OclReal9-def*)

2.5.4. Arithmetical Operations

Definition

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

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

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

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

interpretation *OclAddReal* : *profile-bin_{d-d}* ($+_{\text{real}}$) $\lambda x y. \llbracket x \tau \rrbracket + \llbracket y \tau \rrbracket$
by *unfold-locales (auto simp:OclAddReal-def bot-option-def null-option-def)*

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

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

interpretation *OclMinusReal* : *profile-bin_{d-d}* ($-_{\text{real}}$) $\lambda x y. \llbracket x \tau \rrbracket - \llbracket y \tau \rrbracket$
by *unfold-locales (auto simp:OclMinusReal-def bot-option-def null-option-def)*

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

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

interpretation *OclMultReal* : *profile-bin_{d-d}* *OclMultReal* $\lambda x y. \llbracket x \tau \rrbracket * \llbracket y \tau \rrbracket$

by *unfold-locals (auto simp:OclMult_{Real}-def bot-option-def null-option-def)*

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

definition *OclDivision_{Real}* :: (' \mathfrak{A})*Real* \Rightarrow (' \mathfrak{A})*Real* \Rightarrow (' \mathfrak{A})*Real* (**infix** <*div_{real}*> 45)

where $x \text{ div}_{real} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
 then $\text{if } y \tau \neq \text{OclReal0 } \tau \text{ then } \lfloor \lceil x \rceil \tau^\top / \lceil y \rceil \tau^\top \rfloor$ else *invalid* τ
 else *invalid* τ

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

definition *OclModulus_{Real}* :: (' \mathfrak{A})*Real* \Rightarrow (' \mathfrak{A})*Real* \Rightarrow (' \mathfrak{A})*Real* (**infix** <*mod_{real}*> 45)

where $x \text{ mod}_{real} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
 then $\text{if } y \tau \neq \text{OclReal0 } \tau \text{ then } \lfloor \text{mod-float } \lceil x \rceil \tau^\top \lceil y \rceil \tau^\top \rfloor$ else *invalid* τ
 else *invalid* τ

definition *OclLess_{Real}* :: (' \mathfrak{A})*Real* \Rightarrow (' \mathfrak{A})*Real* \Rightarrow (' \mathfrak{A})*Boolean* (**infix** <*<_{real}*> 35)

where $x <_{real} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
 then $\lfloor \lceil x \rceil \tau^\top < \lceil y \rceil \tau^\top \rfloor$
 else *invalid* τ

interpretation *OclLess_{Real}* : *profile-bin_{d-d}* (<*real*) $\lambda x y. \lfloor \lceil x \rceil \tau^\top < \lceil y \rceil \tau^\top \rfloor$
 by *unfold-locals (auto simp:OclLess_{Real}-def bot-option-def null-option-def)*

definition *OclLe_{Real}* :: (' \mathfrak{A})*Real* \Rightarrow (' \mathfrak{A})*Real* \Rightarrow (' \mathfrak{A})*Boolean* (**infix** <*<=_{real}*> 35)

where $x \leq_{real} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
 then $\lfloor \lceil x \rceil \tau^\top \leq \lceil y \rceil \tau^\top \rfloor$
 else *invalid* τ

interpretation *OclLe_{Real}* : *profile-bin_{d-d}* (<=*real*) $\lambda x y. \lfloor \lceil x \rceil \tau^\top \leq \lceil y \rceil \tau^\top \rfloor$
 by *unfold-locals (auto simp:OclLe_{Real}-def bot-option-def null-option-def)*

Basic Properties

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

by (*rule ext, auto simp:true-def false-def OclAdd_{Real}-def invalid-def*
split: option.split option.split-asm
bool.split bool.split-asm)

Execution with Invalid or Null or Zero as Argument

lemma *OclAdd_{Real}-zero1* [*simp, code-unfold*] :

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

proof (*rule ext, rename-tac* τ , *case-tac* $(v \ x \ \text{and not } (\delta x)) \ \tau = \text{true } \tau$)

fix τ **show** $(v \ x \ \text{and not } (\delta x)) \ \tau = \text{true } \tau \Longrightarrow$

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

apply (*subst OclIf-true', simp add: OclValid-def*)

by (*metis OclAdd_{Real}-def OclNot-defargs OclValid-def foundation5 foundation9*)

next fix τ

have $A: \bigwedge \tau. (\tau \models \text{not } (v \ x \ \text{and not } (\delta x))) = (x \ \tau = \text{invalid } \tau \vee \tau \models \delta x)$

by (*metis OclNot-not OclOr-def defined5 defined6 defined-not-I foundation11 foundation18'*
foundation6 foundation7 foundation9 invalid-def)

have $B: \tau \models \delta x \Longrightarrow \lfloor \lceil x \rceil \tau^\top \rfloor = x \ \tau$

apply (*cases x* τ , *metis bot-option-def foundation16*)

apply (*rename-tac x', case-tac x', metis bot-option-def foundation16 null-option-def*)

by (*simp*)

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

when $\tau \models \text{not } (v \ x \ \text{and not } (\delta x))$

apply (*insert that, subst OclIf-false', simp, simp add: A, auto simp: OclAdd_{Real}-def OclReal0-def*)

```

  apply(simp add: foundation16'[simplified OclValid-def])
  apply(simp add: B)
  by(simp add: OclValid-def)
qed(metis OclValid-def defined5 defined6 defined-and-I defined-not-I foundation9)

```

```

lemma OclAddReal-zero2[simp,code-unfold] :
(0.0 +real x) = (if v x and not (δ x) then invalid else x endif)
by(subst OclAddReal-commute, simp)

```

2.5.5. Test Statements

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

```

Assert τ ⊨ ( 9.0 ≤real 10.0 )
Assert τ ⊨ (( 4.0 +real 4.0 ) ≤real 10.0 )
Assert τ ⊨ (( 4.0 +real ( 4.0 +real 4.0 )) <real 10.0 )
Assert τ ⊨ not (v (null +real 1.0))
Assert τ ⊨ (((9.0 *real 4.0) divreal 10.0) ≤real 4.0)
Assert τ ⊨ not (δ (1.0 divreal 0.0))
Assert τ ⊨ not (v (1.0 divreal 0.0))

```

```

lemma real-non-null [simp]: ((λ-. ⊥n_⊥) ≐ (null::('A)Real)) = false
by(rule ext,auto simp: StrictRefEqReal valid-def
  bot-fun-def bot-option-def null-fun-def null-option-def StrongEq-def)

```

```

lemma null-non-real [simp]: ((null::('A)Real) ≐ (λ-. ⊥n_⊥)) = false
by(rule ext,auto simp: StrictRefEqReal valid-def
  bot-fun-def bot-option-def null-fun-def null-option-def StrongEq-def)

```

```

lemma OclReal0-non-null [simp,code-unfold]: (0.0 ≐ null) = false by(simp add: OclReal0-def)
lemma null-non-OclReal0 [simp,code-unfold]: (null ≐ 0.0) = false by(simp add: OclReal0-def)
lemma OclReal1-non-null [simp,code-unfold]: (1.0 ≐ null) = false by(simp add: OclReal1-def)
lemma null-non-OclReal1 [simp,code-unfold]: (null ≐ 1.0) = false by(simp add: OclReal1-def)
lemma OclReal2-non-null [simp,code-unfold]: (2.0 ≐ null) = false by(simp add: OclReal2-def)
lemma null-non-OclReal2 [simp,code-unfold]: (null ≐ 2.0) = false by(simp add: OclReal2-def)
lemma OclReal6-non-null [simp,code-unfold]: (6.0 ≐ null) = false by(simp add: OclReal6-def)
lemma null-non-OclReal6 [simp,code-unfold]: (null ≐ 6.0) = false by(simp add: OclReal6-def)
lemma OclReal8-non-null [simp,code-unfold]: (8.0 ≐ null) = false by(simp add: OclReal8-def)
lemma null-non-OclReal8 [simp,code-unfold]: (null ≐ 8.0) = false by(simp add: OclReal8-def)
lemma OclReal9-non-null [simp,code-unfold]: (9.0 ≐ null) = false by(simp add: OclReal9-def)
lemma null-non-OclReal9 [simp,code-unfold]: (null ≐ 9.0) = false by(simp add: OclReal9-def)

```

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

Elementary computations on Real

```

Assert τ ⊨ 1.0 <> 2.0
Assert τ ⊨ 2.0 <> 1.0
Assert τ ⊨ 2.0 ≐ 2.0

Assert τ ⊨ v 4.0
Assert τ ⊨ δ 4.0
Assert τ ⊨ v (null::('A)Real)
Assert τ ⊨ (invalid ≐ invalid)

```

```

Assert  $\tau \models (null \triangleq null)$ 
Assert  $\tau \models (4.0 \triangleq 4.0)$ 
Assert  $\tau \not\models (9.0 \triangleq 10.0)$ 
Assert  $\tau \not\models (invalid \triangleq 10.0)$ 
Assert  $\tau \not\models (null \triangleq 10.0)$ 
Assert  $\tau \not\models (invalid \dot{=} (invalid::('A)Real))$ 
Assert  $\tau \not\models v (invalid \dot{=} (invalid::('A)Real))$ 
Assert  $\tau \not\models (invalid \langle \rangle (invalid::('A)Real))$ 
Assert  $\tau \not\models v (invalid \langle \rangle (invalid::('A)Real))$ 
Assert  $\tau \models (null \dot{=} (null::('A)Real) )$ 
Assert  $\tau \models (null \dot{=} (null::('A)Real) )$ 
Assert  $\tau \models (4.0 \dot{=} 4.0)$ 
Assert  $\tau \not\models (4.0 \langle \rangle 4.0)$ 
Assert  $\tau \not\models (4.0 \dot{=} 10.0)$ 
Assert  $\tau \models (4.0 \langle \rangle 10.0)$ 
Assert  $\tau \not\models (0.0 <_{real} null)$ 
Assert  $\tau \not\models (\delta (0.0 <_{real} null))$ 

```

end

```

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

```

2.6. Basic Type String: Operations

2.6.1. Fundamental Properties on Strings: Strict Equality

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

```

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

```

end

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

```

interpretation StrictRefEqString : profile-binStrongEq-v-v  $\lambda$   $x\ y. (x::('A)String) \dot{=} y$ 
  by unfold-locales (auto simp: StrictRefEqString)

```

2.6.2. Basic String Constants

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

```

definition OclStringa :: ('A)String  $\langle$ a $\rangle$    where a = ( $\lambda . \underline{\underline{''a''}}$ )
definition OclStringb :: ('A)String  $\langle$ b $\rangle$    where b = ( $\lambda . \underline{\underline{''b''}}$ )
definition OclStringc :: ('A)String  $\langle$ c $\rangle$    where c = ( $\lambda . \underline{\underline{''c''}}$ )

```

Etc.

2.6.3. Validity and Definedness Properties

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

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

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

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

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

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

2.6.4. String Operations

Definition

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

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

definition *OclAddString* :: ('A)\String \Rightarrow ('A)\String \Rightarrow ('A)\String (**infix** <+string> 40)
where $x +_{\text{string}} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true} \tau \wedge (\delta y) \tau = \text{true} \tau$
then $\llcorner \text{concat } [\llcorner x \llcorner, \llcorner y \llcorner] \llcorner$
else *invalid* τ

interpretation *OclAddString* : *profile-bind-d* (+string) $\lambda x y. \llcorner \text{concat } [\llcorner x \llcorner, \llcorner y \llcorner] \llcorner$
by *unfold-locales (auto simp:OclAddString-def bot-option-def null-option-def)*

Basic Properties

lemma *OclAddString-not-commute*: $\exists X Y. (X +_{\text{string}} Y) \neq (Y +_{\text{string}} X)$
apply(*rule-tac x = \lambda-. \llcorner 'b' \llcorner* **in** *exI*)
apply(*rule-tac x = \lambda-. \llcorner 'a' \llcorner* **in** *exI*)
apply(*simp-all add:OclAddString-def*)
by(*auto, drule fun-cong, auto*)

2.6.5. Test Statements

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

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

Elementary computations on String

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

Assert $\tau \models v a$
Assert $\tau \models \delta a$
Assert $\tau \models v (\text{null}::('A)\text{String})$

```

Assert  $\tau \models (\text{invalid} \triangleq \text{invalid})$ 
Assert  $\tau \models (\text{null} \triangleq \text{null})$ 
Assert  $\tau \models (\text{a} \triangleq \text{a})$ 
Assert  $\tau \not\models (\text{a} \triangleq \text{b})$ 
Assert  $\tau \not\models (\text{invalid} \triangleq \text{b})$ 
Assert  $\tau \not\models (\text{null} \triangleq \text{b})$ 
Assert  $\tau \not\models (\text{invalid} \doteq (\text{invalid}::(\mathfrak{A})String))$ 
Assert  $\tau \not\models v (\text{invalid} \doteq (\text{invalid}::(\mathfrak{A})String))$ 
Assert  $\tau \not\models (\text{invalid} \langle \rangle (\text{invalid}::(\mathfrak{A})String))$ 
Assert  $\tau \not\models v (\text{invalid} \langle \rangle (\text{invalid}::(\mathfrak{A})String))$ 
Assert  $\tau \models (\text{null} \doteq (\text{null}::(\mathfrak{A})String))$ 
Assert  $\tau \models (\text{null} \doteq (\text{null}::(\mathfrak{A})String))$ 
Assert  $\tau \models (\text{b} \doteq \text{b})$ 
Assert  $\tau \not\models (\text{b} \langle \rangle \text{b})$ 
Assert  $\tau \not\models (\text{b} \doteq \text{c})$ 
Assert  $\tau \models (\text{b} \langle \rangle \text{c})$ 

```

end

```

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

```

2.7. Collection Type Pairs: Operations

The OCL standard provides the concept of *Tuples*, i. e. a family of record-types with projection functions. In FeatherWeight OCL, only the theory of a special case is developed, namely the type of Pairs, which is, however, sufficient for all applications since it can be used to mimick all tuples. In particular, it can be used to express operations with multiple arguments, roles of n-ary associations, ...

2.7.1. Semantic Properties of the Type Constructor

```

lemma A[simp]:Rep-Pairbase x ≠ None ⇒ Rep-Pairbase x ≠ null ⇒ (fst ⊢Rep-Pairbase x⊢) ≠ bot
by(insert Rep-Pairbase[of x],auto simp:null-option-def bot-option-def)

```

```

lemma A'[simp]: x ≠ bot ⇒ x ≠ null ⇒ (fst ⊢Rep-Pairbase x⊢) ≠ bot
apply(insert Rep-Pairbase[of x], simp add: bot-Pairbase-def null-Pairbase-def)
apply(auto simp:null-option-def bot-option-def)
apply(erule contrapos-np[of x = Abs-Pairbase None])
apply(subst Rep-Pairbase-inject[symmetric], simp)
apply(subst Pairbase.Abs-Pairbase-inverse, simp-all,simp add: bot-option-def)
apply(erule contrapos-np[of x = Abs-Pairbase ⊥None])
apply(subst Rep-Pairbase-inject[symmetric], simp)
apply(subst Pairbase.Abs-Pairbase-inverse, simp-all,simp add: null-option-def bot-option-def)
done

```

```

lemma B[simp]:Rep-Pairbase x ≠ None ⇒ Rep-Pairbase x ≠ null ⇒ (snd ⊢Rep-Pairbase x⊢) ≠ bot
by(insert Rep-Pairbase[of x],auto simp:null-option-def bot-option-def)

```

```

lemma B'[simp]:x ≠ bot ⇒ x ≠ null ⇒ (snd ⊢Rep-Pairbase x⊢) ≠ bot
apply(insert Rep-Pairbase[of x], simp add: bot-Pairbase-def null-Pairbase-def)
apply(auto simp:null-option-def bot-option-def)
apply(erule contrapos-np[of x = Abs-Pairbase None])
apply(subst Rep-Pairbase-inject[symmetric], simp)

```

```

apply(subst Pairbase.Abs-Pairbase-inverse, simp-all, simp add: bot-option-def)
apply(erule contrapos-np[of x = Abs-Pairbase _None_])
apply(subst Rep-Pairbase-inject[symmetric], simp)
apply(subst Pairbase.Abs-Pairbase-inverse, simp-all, simp add: null-option-def bot-option-def)
done

```

2.7.2. Fundamental Properties of Strict Equality

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

overloading

```

StrictRefEq ≡ StrictRefEq :: (('A, 'α::null, 'β::null)Pair, ('A, 'α::null, 'β::null)Pair) ⇒ ('A)Boolean

```

begin

definition StrictRefEq_{Pair} :

```

((x::('A, 'α::null, 'β::null)Pair) ≐ y) ≡ (λ τ. if (v x) τ = true τ ∧ (v y) τ = true τ
                                     then (x ≐ y)τ
                                     else invalid τ)

```

end

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

```

interpretation StrictRefEqPair : profile-binStrongEq-v-v λ x y. (x::('A, 'α::null, 'β::null)Pair) ≐ y
by unfold-locales (auto simp: StrictRefEqPair)

```

2.7.3. Standard Operations Definitions

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

Definition: Pair Constructor

definition OclPair::('A, 'α) val ⇒

```

('A, 'β) val ⇒
('A, 'α::null, 'β::null) Pair (⟨Pair{(-),(-)}⟩)

```

where Pair{X, Y} ≡ (λ τ. if (v X) τ = true τ ∧ (v Y) τ = true τ
then Abs-Pair_{base} ⊔ (X τ, Y τ) ⊔
else invalid τ)

interpretation OclPair : profile-bin_{v-v}

```

OclPair λ x y. Abs-Pairbase ⊔ (x, y) ⊔

```

```

apply(unfold-locales, auto simp: OclPair-def bot-Pairbase-def null-Pairbase-def)

```

```

by(auto simp: Abs-Pairbase-inject null-option-def bot-option-def)

```

Definition: First

definition OclFirst:: ('A, 'α::null, 'β::null) Pair ⇒ ('A, 'α) val (⟨ - .First'(')⟩)

where X .First() ≡ (λ τ. if (δ X) τ = true τ
then fst [⊔]Rep-Pair_{base} (X τ) [⊔]
else invalid τ)

interpretation OclFirst : profile-mono_d OclFirst λx. fst [⊔]Rep-Pair_{base} (x) [⊔]

```

by unfold-locales (auto simp: OclFirst-def)

```

Definition: Second

definition OclSecond:: ('A, 'α::null, 'β::null) Pair ⇒ ('A, 'β) val (⟨ - .Second'(')⟩)

where X .Second() ≡ (λ τ. if (δ X) τ = true τ

then snd $\ulcorner \text{Rep-Pair}_{base} (X \tau) \urcorner$
 else invalid τ)

interpretation $\text{OclSecond} : \text{profile-mono}_d \text{OclSecond} \lambda x. \text{snd } \ulcorner \text{Rep-Pair}_{base} (x) \urcorner$
 by *unfold-locales* (*auto simp: OclSecond-def*)

2.7.4. Logical Properties

lemma 1 : $\tau \models v Y \implies \tau \models \text{Pair}\{X, Y\} .\text{First}() \triangleq X$
apply(*case-tac* $\neg(\tau \models v X)$)
apply(*erule foundation7* [*THEN iffD2*, *THEN foundation15* [*THEN iffD2*,
THEN StrongEq-L-subst2-rev]], *simp-all add: foundation18'*)
apply(*auto simp: OclValid-def valid-def defined-def StrongEq-def OclFirst-def OclPair-def*
true-def false-def invalid-def bot-fun-def null-fun-def)
apply(*auto simp: Abs-Pair_{base}-inject null-option-def bot-option-def bot-Pair_{base}-def null-Pair_{base}-def*)
by(*simp add: Abs-Pair_{base}-inverse*)

lemma 2 : $\tau \models v X \implies \tau \models \text{Pair}\{X, Y\} .\text{Second}() \triangleq Y$
apply(*case-tac* $\neg(\tau \models v Y)$)
apply(*erule foundation7* [*THEN iffD2*, *THEN foundation15* [*THEN iffD2*,
THEN StrongEq-L-subst2-rev]], *simp-all add: foundation18'*)
apply(*auto simp: OclValid-def valid-def defined-def StrongEq-def OclSecond-def OclPair-def*
true-def false-def invalid-def bot-fun-def null-fun-def)
apply(*auto simp: Abs-Pair_{base}-inject null-option-def bot-option-def bot-Pair_{base}-def null-Pair_{base}-def*)
by(*simp add: Abs-Pair_{base}-inverse*)

2.7.5. Algebraic Execution Properties

lemma proj1-exec [*simp, code-unfold*] : $\text{Pair}\{X, Y\} .\text{First}() = (\text{if } (v Y) \text{ then } X \text{ else invalid endif})$
apply(*rule ext, rename-tac* τ , *simp add: foundation22[symmetric]*)
apply(*case-tac* $\neg(\tau \models v Y)$)
apply(*erule foundation7* [*THEN iffD2*,
THEN foundation15 [*THEN iffD2*,
THEN StrongEq-L-subst2-rev]], *simp-all*)
apply(*subgoal-tac* $\tau \models v Y$)
apply(*erule foundation13* [*THEN iffD2*, *THEN StrongEq-L-subst2-rev*], *simp-all*)
by(*erule 1*)

lemma proj2-exec [*simp, code-unfold*] : $\text{Pair}\{X, Y\} .\text{Second}() = (\text{if } (v X) \text{ then } Y \text{ else invalid endif})$
apply(*rule ext, rename-tac* τ , *simp add: foundation22[symmetric]*)
apply(*case-tac* $\neg(\tau \models v X)$)
apply(*erule foundation7* [*THEN iffD2*, *THEN foundation15* [*THEN iffD2*,
THEN StrongEq-L-subst2-rev]], *simp-all*)
apply(*subgoal-tac* $\tau \models v X$)
apply(*erule foundation13* [*THEN iffD2*, *THEN StrongEq-L-subst2-rev*], *simp-all*)
by(*erule 2*)

2.7.6. Test Statements

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

end

```
theory UML-Bag
imports ../basic-types/UML-Void
        ../basic-types/UML-Boolean
        ../basic-types/UML-Integer
        ../basic-types/UML-String
        ../basic-types/UML-Real
begin
```

```
no-notation None (⟦⊥⟧)
```

2.8. Collection Type Bag: Operations

```
definition Rep-Bag-base' x = {(x0, y). y < ⊢Rep-Bagbase x⊢ x0 }
```

```
definition Rep-Bag-base x τ = {(x0, y). y < ⊢Rep-Bagbase (x τ)⊢ x0 }
```

```
definition Rep-Set-base x τ = fst ' {(x0, y). y < ⊢Rep-Bagbase (x τ)⊢ x0 }
```

```
definition ApproxEq (infixl ≅) 30)
```

```
where X ≅ Y ≡ λ τ. ⊢Rep-Set-base X τ = Rep-Set-base Y τ⊢
```

2.8.1. As a Motivation for the (infinite) Type Construction: Type-Extensions as Bags

Our notion of typed bag goes beyond the usual notion of a finite executable bag and is powerful enough to capture *the extension of a type* in UML and OCL. This means we can have in Featherweight OCL Bags containing all possible elements of a type, not only those (finite) ones representable in a state. This holds for base types as well as class types, although the notion for class-types — involving object id's not occurring in a state — requires some care.

In a world with *invalid* and *null*, there are two notions extensions possible:

1. the bag of all *defined* values of a type T (for which we will introduce the constant T)
2. the bag of all *valid* values of a type T , so including *null* (for which we will introduce the constant T_{null}).

We define the bag extensions for the base type *Integer* as follows:

```
definition Integer :: ('A, Integerbase) Bag
```

```
where Integer ≡ (λ τ. (Abs-Bagbase o Some o Some) (λ None ⇒ 0 | Some None ⇒ 0 | - ⇒ 1))
```

```
definition Integernull :: ('A, Integerbase) Bag
```

```
where Integernull ≡ (λ τ. (Abs-Bagbase o Some o Some) (λ None ⇒ 0 | - ⇒ 1))
```

```
lemma Integer-defined : δ Integer = true
```

```
apply(rule ext, auto simp: Integer-def defined-def false-def true-def
      bot-fun-def null-fun-def null-option-def)
```

```
by(simp-all add: Abs-Bagbase-inject bot-option-def bot-Bagbase-def null-Bagbase-def null-option-def)
```

```
lemma Integernull-defined : δ Integernull = true
```

```
apply(rule ext, auto simp: Integernull-def defined-def false-def true-def)
```

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

by(*simp-all add: Abs-Bag_{base}-inject bot-option-def bot-Bag_{base}-def null-Bag_{base}-def null-option-def*)

This allows the theorems:

$\tau \models \delta x \implies \tau \models (\text{Integer} \rightarrow \text{includes}_{\text{Bag}}(x)) \quad \tau \models \delta x \implies \tau \models \text{Integer} \triangleq$
 $(\text{Integer} \rightarrow \text{including}_{\text{Bag}}(x))$

and

$\tau \models v x \implies \tau \models (\text{Integer}_{\text{null}} \rightarrow \text{includes}_{\text{Bag}}(x)) \quad \tau \models v x \implies \tau \models \text{Integer}_{\text{null}} \triangleq$
 $(\text{Integer}_{\text{null}} \rightarrow \text{including}_{\text{Bag}}(x))$

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

In the same spirit, we proceed similarly for the remaining base types:

definition $\text{Void}_{\text{null}} :: ('A, \text{Void}_{\text{base}}) \text{Bag}$

where $\text{Void}_{\text{null}} \equiv (\lambda \tau. (\text{Abs-Bag}_{\text{base}} \circ \text{Some} \circ \text{Some}) (\lambda x. \text{if } x = \text{Abs-Void}_{\text{base}} (\text{Some } \text{None}) \text{ then } 1 \text{ else } 0))$

definition $\text{Void}_{\text{empty}} :: ('A, \text{Void}_{\text{base}}) \text{Bag}$

where $\text{Void}_{\text{empty}} \equiv (\lambda \tau. (\text{Abs-Bag}_{\text{base}} \circ \text{Some} \circ \text{Some}) (\lambda \cdot. 0))$

lemma $\text{Void}_{\text{null}}\text{-defined} : \delta \text{Void}_{\text{null}} = \text{true}$

apply(*rule ext, auto simp: Void_{null}-def defined-def false-def true-def*

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

bot-Bag_{base}-def null-Bag_{base}-def)

by((*subst (asm) Abs-Bag_{base}-inject, auto simp add: bot-option-def null-option-def bot-Void-def*),
(subst (asm) Abs-Void_{base}-inject, auto simp add: bot-option-def null-option-def))+

lemma $\text{Void}_{\text{empty}}\text{-defined} : \delta \text{Void}_{\text{empty}} = \text{true}$

apply(*rule ext, auto simp: Void_{empty}-def defined-def false-def true-def*

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

bot-Bag_{base}-def null-Bag_{base}-def)

by((*subst (asm) Abs-Bag_{base}-inject, auto simp add: bot-option-def null-option-def bot-Void-def*))+

lemma assumes $\tau \models \delta (V :: ('A, \text{Void}_{\text{base}}) \text{Bag})$

shows $\tau \models V \cong \text{Void}_{\text{null}} \vee \tau \models V \cong \text{Void}_{\text{empty}}$

proof –

have $A: \bigwedge x y. x \neq \{\} \implies \exists y. y \in x$

by (*metis all-not-in-conv*)

show *?thesis*

apply(*case-tac V* τ)

proof – **fix** y **show** $V \tau = \text{Abs-Bag}_{\text{base}} y \implies$

$y \in \{X. X = \perp \vee X = \text{null} \vee \ulcorner X \urcorner \perp = 0\} \implies$

$\tau \models V \cong \text{Void}_{\text{null}} \vee \tau \models V \cong \text{Void}_{\text{empty}}$

apply(*insert assms, case-tac y, simp add: bot-option-def, simp add: bot-Bag_{base}-def foundation16*)

apply(*simp add: bot-option-def null-option-def*)

apply(*erule disjE, metis OclValid-def defined-def foundation2 null-Bag_{base}-def null-fun-def true-def*)

proof – **fix** a **show** $V \tau = \text{Abs-Bag}_{\text{base}} \ulcorner a \urcorner \implies \ulcorner a \urcorner \perp = 0 \implies \tau \models V \cong \text{Void}_{\text{null}} \vee \tau \models V \cong \text{Void}_{\text{empty}}$

apply(*case-tac a, simp, insert assms, metis OclValid-def foundation16 null-Bag_{base}-def true-def*)

apply(*simp*)

proof – **fix** aa **show** $V \tau = \text{Abs-Bag}_{\text{base}} \ulcorner aa \urcorner \implies aa \perp = 0 \implies \tau \models V \cong \text{Void}_{\text{null}} \vee \tau \models V \cong \text{Void}_{\text{empty}}$

apply(*case-tac aa (Abs-Void_{base} $\ulcorner \text{None} \urcorner) = 0$,*

rule disjI2,

insert assms,

simp add: Void_{empty}-def OclValid-def ApproxEq-def Rep-Set-base-def true-def Abs-Bag_{base}-inverse im-

age-def)

apply(*intro allI*)

proof – **fix** x **fix** b **show** $V \tau = \text{Abs-Bag}_{\text{base}} \ulcorner aa \urcorner \implies aa \perp = 0 \implies aa (\text{Abs-Void}_{\text{base}} \ulcorner \text{None} \urcorner) = 0 \implies$
 $(\delta V) \tau = \ulcorner \text{True} \urcorner \implies \neg b < aa x$

```

apply (case-tac x, auto)
apply (simp add: bot-Void-def bot-option-def)
apply (simp add: bot-option-def null-option-def)
done
apply-end(simp+, rule disjI1)
show  $V \tau = \text{Abs-Bag}_{\text{base}} \llcorner aa \llcorner \implies aa \perp = 0 \implies 0 < aa \text{ (Abs-Void}_{\text{base}} \llcorner \text{None}\llcorner) \implies \tau \models \delta V \implies \tau \models V$ 
 $\cong \text{Void}_{\text{null}}$ 
apply(simp add: Voidnull-def OclValid-def ApproxEq-def Rep-Set-base-def true-def Abs-Bagbase-inverse im-
age-def,
subst Abs-Bagbase-inverse, simp)
using bot-Void-def apply auto[1]
apply(simp)
apply(rule equalityI, rule subsetI, simp)
proof - fix x show  $V \tau = \text{Abs-Bag}_{\text{base}} \llcorner aa \llcorner \implies$ 
 $aa \perp = 0 \implies 0 < aa \text{ (Abs-Void}_{\text{base}} \llcorner \text{None}\llcorner) \implies (\delta V) \tau = \llcorner \text{True}\llcorner \implies \exists b. b < aa x \implies x =$ 
Abs-Voidbase  $\llcorner \text{None}\llcorner$ 
apply( case-tac x, auto)
apply (simp add: bot-Void-def bot-option-def)
by (simp add: bot-option-def null-option-def)
qed ((simp add: bot-Void-def bot-option-def)+, blast)
qed qed qed qed qed

```

```

definition Boolean :: (' $\mathcal{A}$ , Booleanbase) Bag
where Boolean  $\equiv (\lambda \tau. (\text{Abs-Bag}_{\text{base}} \circ \text{Some} \circ \text{Some}) (\lambda \text{None} \Rightarrow 0 \mid \text{Some None} \Rightarrow 0 \mid - \Rightarrow 1))$ 

```

```

definition Booleannull :: (' $\mathcal{A}$ , Booleanbase) Bag
where Booleannull  $\equiv (\lambda \tau. (\text{Abs-Bag}_{\text{base}} \circ \text{Some} \circ \text{Some}) (\lambda \text{None} \Rightarrow 0 \mid - \Rightarrow 1))$ 

```

```

lemma Boolean-defined :  $\delta \text{ Boolean} = \text{true}$ 
apply(rule ext, auto simp: Boolean-def defined-def false-def true-def
bot-fun-def null-fun-def null-option-def)
by(simp-all add: Abs-Bagbase-inject bot-option-def bot-Bagbase-def null-Bagbase-def null-option-def)

```

```

lemma Booleannull-defined :  $\delta \text{ Boolean}_{\text{null}} = \text{true}$ 
apply(rule ext, auto simp: Booleannull-def defined-def false-def true-def
bot-fun-def null-fun-def null-option-def)
by(simp-all add: Abs-Bagbase-inject bot-option-def bot-Bagbase-def null-Bagbase-def null-option-def)

```

```

definition String :: (' $\mathcal{A}$ , Stringbase) Bag
where String  $\equiv (\lambda \tau. (\text{Abs-Bag}_{\text{base}} \circ \text{Some} \circ \text{Some}) (\lambda \text{None} \Rightarrow 0 \mid \text{Some None} \Rightarrow 0 \mid - \Rightarrow 1))$ 

```

```

definition Stringnull :: (' $\mathcal{A}$ , Stringbase) Bag
where Stringnull  $\equiv (\lambda \tau. (\text{Abs-Bag}_{\text{base}} \circ \text{Some} \circ \text{Some}) (\lambda \text{None} \Rightarrow 0 \mid - \Rightarrow 1))$ 

```

```

lemma String-defined :  $\delta \text{ String} = \text{true}$ 
apply(rule ext, auto simp: String-def defined-def false-def true-def
bot-fun-def null-fun-def null-option-def)
by(simp-all add: Abs-Bagbase-inject bot-option-def bot-Bagbase-def null-Bagbase-def null-option-def)

```

```

lemma Stringnull-defined :  $\delta \text{ String}_{\text{null}} = \text{true}$ 
apply(rule ext, auto simp: Stringnull-def defined-def false-def true-def
bot-fun-def null-fun-def null-option-def)
by(simp-all add: Abs-Bagbase-inject bot-option-def bot-Bagbase-def null-Bagbase-def null-option-def)

```

```

definition Real :: (' $\mathcal{A}$ , Realbase) Bag
where Real  $\equiv (\lambda \tau. (\text{Abs-Bag}_{\text{base}} \circ \text{Some} \circ \text{Some}) (\lambda \text{None} \Rightarrow 0 \mid \text{Some None} \Rightarrow 0 \mid - \Rightarrow 1))$ 

```

definition $Real_{null} :: (\mathfrak{A}, Real_{base}) Bag$
where $Real_{null} \equiv (\lambda \tau. (Abs-Bag_{base} \circ Some \circ Some) (\lambda None \Rightarrow 0 \mid - \Rightarrow 1))$

lemma $Real_defined : \delta Real = true$
apply(rule ext, auto simp: Real-def defined-def false-def true-def
 bot-fun-def null-fun-def null-option-def)
by(simp-all add: Abs-Bag_{base}-inject bot-option-def bot-Bag_{base}-def null-Bag_{base}-def null-option-def)

lemma $Real_{null}_defined : \delta Real_{null} = true$
apply(rule ext, auto simp: Real_{null}-def defined-def false-def true-def
 bot-fun-def null-fun-def null-option-def)
by(simp-all add: Abs-Bag_{base}-inject bot-option-def bot-Bag_{base}-def null-Bag_{base}-def null-option-def)

2.8.2. Basic Properties of the Bag Type

Every element in a defined bag is valid.

lemma $Bag_inv_lemma: \tau \models (\delta X) \implies \ulcorner Rep-Bag_{base} (X \tau) \urcorner bot = 0$
apply(insert Rep-Bag_{base} [of X τ], simp)
apply(auto simp: OclValid-def defined-def false-def true-def cp-def
 bot-fun-def bot-Bag_{base}-def null-Bag_{base}-def null-fun-def
 split:if-split-asm)
apply(erule contrapos-pp [of Rep-Bag_{base} (X τ) = bot])
apply(subst Abs-Bag_{base}-inject[symmetric], rule Rep-Bag_{base}, simp)
apply(simp add: Rep-Bag_{base}-inverse bot-Bag_{base}-def bot-option-def)
apply(erule contrapos-pp [of Rep-Bag_{base} (X τ) = null])
apply(subst Abs-Bag_{base}-inject[symmetric], rule Rep-Bag_{base}, simp)
apply(simp add: Rep-Bag_{base}-inverse null-option-def)
by (simp add: bot-option-def)

lemma Bag_inv_lemma' :
assumes $x_def : \tau \models \delta X$
and $e_mem : \ulcorner Rep-Bag_{base} (X \tau) \urcorner e \geq 1$
shows $\tau \models v (\lambda \cdot e)$
apply(case-tac $e = bot$, insert assms, drule Bag-inv-lemma, simp)
by (simp add: foundation18')

lemma abs_rep_simp' :
assumes $S_all_def : \tau \models \delta S$
shows $Abs-Bag_{base} \llcorner \ulcorner Rep-Bag_{base} (S \tau) \urcorner \llcorner = S \tau$

proof –
have $discr_eq_false_true : \bigwedge \tau. (false \tau = true \tau) = False$ **by**(simp add: false-def true-def)
show ?thesis
apply(insert S-all-def, simp add: OclValid-def defined-def)
apply(rule mp[OF Abs-Bag_{base}-induct[**where** $P = \lambda S. (if S = \perp \tau \vee S = null \tau$
 then false τ else true $\tau) = true \tau \longrightarrow$
 $Abs-Bag_{base} \llcorner \ulcorner Rep-Bag_{base} S \urcorner \llcorner = S$]],
 rename-tac S')
apply(simp add: Abs-Bag_{base}-inverse discr-eq-false-true)
apply(case-tac S') **apply**(simp add: bot-fun-def bot-Bag_{base}-def)+
apply(rename-tac S'' , case-tac S'') **apply**(simp add: null-fun-def null-Bag_{base}-def)+
done
qed

lemma $invalid_bag_OclNot_defined$ [simp,code-unfold]: $\delta(invalid::(\mathfrak{A}, \alpha::null) Bag) = false$ **by** simp

lemma $null_bag_OclNot_defined$ [simp,code-unfold]: $\delta(null::(\mathfrak{A}, \alpha::null) Bag) = false$

by(simp add: defined-def null-fun-def)

lemma $invalid_bag_valid$ [simp,code-unfold]: $v(invalid::(\mathfrak{A}, \alpha::null) Bag) = false$

```

by simp
lemma null-bag-valid [simp,code-unfold]:v(null::('A,'α::null) Bag) = true
apply(simp add: valid-def null-fun-def bot-fun-def bot-Bagbase-def null-Bagbase-def)
apply(subst Abs-Bagbase-inject,simp-all add: null-option-def bot-option-def)
done

```

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

2.8.3. Definition: Strict Equality

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

```

overloading StrictRefEq ≡ StrictRefEq :: [(('A,'α::null) Bag),('A,'α::null) Bag] ⇒ ('A) Boolean
begin
  definition StrictRefEqBag :
    (x::('A,'α::null) Bag) ≐ y ≡ λ τ. if (v x) τ = true τ ∧ (v y) τ = true τ
      then (x ≐ y) τ
      else invalid τ
end

```

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

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

```

interpretation StrictRefEqBag : profile-binStrongEqv-v λ x y. (x::('A,'α::null) Bag) ≐ y
  by unfold-locales (auto simp: StrictRefEqBag)

```

2.8.4. Constants: mtBag

```

definition mtBag::('A,'α::null) Bag (⋅Bag{⋅})
where   Bag{⋅} ≡ (λ τ. Abs-Bagbase ⊥ λ . 0::nat ⊥)

```

```

lemma mtBag-defined[simp,code-unfold]:δ(Bag{⋅}) = true
apply(rule ext, auto simp: mtBag-def defined-def null-Bagbase-def
  bot-Bagbase-def bot-fun-def null-fun-def)
by(simp-all add: Abs-Bagbase-inject bot-option-def null-option-def)

```

```

lemma mtBag-valid[simp,code-unfold]:v(Bag{⋅}) = true
apply(rule ext,auto simp: mtBag-def valid-def
  bot-Bagbase-def bot-fun-def null-fun-def)
by(simp-all add: Abs-Bagbase-inject bot-option-def null-option-def)

```

```

lemma mtBag-rep-bag:  $\ulcorner \text{Rep-Bag}_{base} (\text{Bag}\{\cdot\} \tau) \urcorner = (\lambda \cdot . 0)$ 
  apply(simp add: mtBag-def, subst Abs-Bagbase-inverse)
by(simp add: bot-option-def)+lemma [simp,code-unfold]: const Bag{⋅}
by(simp add: const-def mtBag-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.

2.8.5. Definition: Including

definition $OclIncluding :: [(\mathfrak{A}, \alpha :: null) Bag, (\mathfrak{A}, \alpha) val] \Rightarrow (\mathfrak{A}, \alpha) Bag$
where $OclIncluding x y = (\lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (v y) \tau = true \tau$
 $\text{then } Abs-Bag_{base} \perp \ulcorner Rep-Bag_{base}(x \tau) \urcorner$
 $\quad \quad \quad ((y \tau) := \ulcorner Rep-Bag_{base}(x \tau) \urcorner (y \tau) + 1)$
 $\quad \quad \quad \text{else } invalid \tau \urcorner)$

notation $OclIncluding \langle \langle - \rangle \rightarrow including_{Bag} '(-)' \rangle$

interpretation $OclIncluding : profile-bin_{d-v} OclIncluding \lambda x y. Abs-Bag_{base} \perp \ulcorner Rep-Bag_{base} x \urcorner$
 $(y := \ulcorner Rep-Bag_{base} x \urcorner y + 1) \perp$

proof –

let $?X = \lambda x y. \ulcorner Rep-Bag_{base}(x) \urcorner ((y) := \ulcorner Rep-Bag_{base}(x) \urcorner (y) + 1)$
show $profile-bin_{d-v} OclIncluding (\lambda x y. Abs-Bag_{base} \perp ?X x y \perp)$
apply $unfold-locales$
apply $(auto simp: OclIncluding-def bot-option-def null-option-def$
 $\quad \quad \quad bot-Bag_{base}-def null-Bag_{base}-def)$
by $(subst (asm) Abs-Bag_{base}-inject, simp-all,$
 $\quad \quad \quad metis (mono-tags, lifting) Rep-Bag_{base} Rep-Bag_{base}-inverse bot-option-def mem-Collect-eq$
 $\quad \quad \quad null-option-def,$
 $\quad \quad \quad simp add: bot-option-def null-option-def) +$
qed

syntax

$-OclFinbag :: args \Rightarrow (\mathfrak{A}, \alpha :: null) Bag \quad \langle Bag\{-\} \rangle$

syntax-consts

$-OclFinbag == OclIncluding$

translations

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

$Bag\{x\} == CONST OclIncluding (Bag\{\}) x$

2.8.6. Definition: Excluding

definition $OclExcluding :: [(\mathfrak{A}, \alpha :: null) Bag, (\mathfrak{A}, \alpha) val] \Rightarrow (\mathfrak{A}, \alpha) Bag$
where $OclExcluding x y = (\lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (v y) \tau = true \tau$
 $\text{then } Abs-Bag_{base} \perp \ulcorner Rep-Bag_{base}(x \tau) \urcorner ((y \tau) := 0 :: nat) \perp$
 $\quad \quad \quad \text{else } invalid \tau \urcorner)$

notation $OclExcluding \langle \langle - \rangle \rightarrow excluding_{Bag} '(-)' \rangle$

interpretation $OclExcluding : profile-bin_{d-v} OclExcluding$
 $\lambda x y. Abs-Bag_{base} \perp \ulcorner Rep-Bag_{base}(x) \urcorner (y := 0 :: nat) \perp$

proof –

show $profile-bin_{d-v} OclExcluding (\lambda x y. Abs-Bag_{base} \perp \ulcorner Rep-Bag_{base} x \urcorner (y := 0) \perp)$
apply $unfold-locales$
apply $(auto simp: OclExcluding-def bot-option-def null-option-def$
 $\quad \quad \quad null-Bag_{base}-def bot-Bag_{base}-def)$
by $(subst (asm) Abs-Bag_{base}-inject,$
 $\quad \quad \quad simp-all add: bot-option-def null-option-def,$
 $\quad \quad \quad metis (mono-tags, lifting) Rep-Bag_{base} Rep-Bag_{base}-inverse bot-option-def$
 $\quad \quad \quad mem-Collect-eq null-option-def) +$
qed

2.8.7. Definition: Includes

definition $OclIncludes :: [(\mathfrak{A}, \alpha :: null) Bag, (\mathfrak{A}, \alpha) val] \Rightarrow \mathfrak{A} Boolean$
where $OclIncludes x y = (\lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (v y) \tau = true \tau$
 $\text{then } \perp \ulcorner Rep-Bag_{base}(x \tau) \urcorner (y \tau) > 0 \perp$

notation $OclIncludes$ $(\langle \dashrightarrow includes_{Bag} '(-)' \rangle)$

interpretation $OclIncludes : profile-bin_{d-v} OclIncludes \lambda x y. \perp \ulcorner Rep-Bag_{base} x \urcorner y > 0 \perp$
by(*unfold-locales, auto simp:OclIncludes-def bot-option-def null-option-def invalid-def*)

2.8.8. Definition: Excludes

definition $OclExcludes :: ('\mathfrak{A}, '\alpha :: null) Bag, ('\mathfrak{A}, '\alpha) val] \Rightarrow '\mathfrak{A} Boolean$
where $OclExcludes x y = (not(OclIncludes x y))$
notation $OclExcludes (\langle \dashrightarrow excludes_{Bag} '(-)' \rangle)$

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

interpretation $OclExcludes : profile-bin_{d-v} OclExcludes \lambda x y. \perp \ulcorner Rep-Bag_{base} x \urcorner y \leq 0 \perp$
by(*unfold-locales, auto simp:OclExcludes-def OclIncludes-def OclNot-def bot-option-def null-option-def invalid-def*)

2.8.9. Definition: Size

definition $OclSize :: ('\mathfrak{A}, '\alpha :: null) Bag \Rightarrow '\mathfrak{A} Integer$
where $OclSize x = (\lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge finite (Rep-Bag-base x \tau)$
 $\text{then } \perp int (card (Rep-Bag-base x \tau)) \perp$
 $\text{else } \perp)$
notation $OclSize (\langle \dashrightarrow size_{Bag} '(-)' \rangle)$

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

2.8.10. Definition: IsEmpty

definition $OclIsEmpty :: ('\mathfrak{A}, '\alpha :: null) Bag \Rightarrow '\mathfrak{A} Boolean$
where $OclIsEmpty x = ((v x \text{ and not } (\delta x)) \text{ or } ((OclSize x) \doteq 0))$
notation $OclIsEmpty (\langle \dashrightarrow isEmpty_{Bag} '(-)' \rangle)$

2.8.11. Definition: NotEmpty

definition $OclNotEmpty :: ('\mathfrak{A}, '\alpha :: null) Bag \Rightarrow '\mathfrak{A} Boolean$
where $OclNotEmpty x = not(OclIsEmpty x)$
notation $OclNotEmpty (\langle \dashrightarrow notEmpty_{Bag} '(-)' \rangle)$

2.8.12. Definition: Any

definition $OclANY :: [(' \mathfrak{A}, '\alpha :: null) Bag] \Rightarrow (' \mathfrak{A}, '\alpha) val$
where $OclANY x = (\lambda \tau. \text{if } (v x) \tau = true \tau$
 $\text{then if } (\delta x \text{ and } OclNotEmpty x) \tau = true \tau$
 $\text{then SOME } y. y \in (Rep-Set-base x \tau)$
 $\text{else null } \tau$
 $\text{else } \perp)$
notation $OclANY (\langle \dashrightarrow any_{Bag} '(-)' \rangle)$

2.8.13. Definition: Forall

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

definition *OclForall* :: [($\mathfrak{A}, \alpha :: \text{null}$) Bag, (\mathfrak{A}, α) val \Rightarrow (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean
where *OclForall* S P = ($\lambda \tau$. if (δ S) $\tau = \text{true}$ τ
then if ($\exists x \in \text{Rep-Set-base } S \tau$. P ($\lambda \cdot$. x) $\tau = \text{false}$ τ)
then false τ
else if ($\exists x \in \text{Rep-Set-base } S \tau$. P ($\lambda \cdot$. x) $\tau = \text{invalid}$ τ)
then invalid τ
else if ($\exists x \in \text{Rep-Set-base } S \tau$. P ($\lambda \cdot$. x) $\tau = \text{null}$ τ)
then null τ
else true τ
else \perp)

syntax

-*OclForall* Bag :: [($\mathfrak{A}, \alpha :: \text{null}$) Bag, id, (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean ($\langle (-) \rightarrow \text{forall}_{\text{Bag}} '(-)' \rangle$)

syntax-consts

-*OclForall* Bag == UML-Bag.OclForall

translations

X \rightarrow forall_{Bag}(x | P) == CONST UML-Bag.OclForall X (%x. P)

2.8.14. Definition: Exists

Like OclForall, OclExists is also not strict.

definition *OclExists* :: [($\mathfrak{A}, \alpha :: \text{null}$) Bag, (\mathfrak{A}, α) val \Rightarrow (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean
where *OclExists* S P = not(UML-Bag.OclForall S (λX . not (P X)))

syntax

-*OclExist* Bag :: [($\mathfrak{A}, \alpha :: \text{null}$) Bag, id, (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean ($\langle (-) \rightarrow \text{exists}_{\text{Bag}} '(-)' \rangle$)

syntax-consts

-*OclExist* Bag == UML-Bag.OclExists

translations

X \rightarrow exists_{Bag}(x | P) == CONST UML-Bag.OclExists X (%x. P)

2.8.15. Definition: Iterate

definition *OclIterate* :: [($\mathfrak{A}, \alpha :: \text{null}$) Bag, ($\mathfrak{A}, \beta :: \text{null}$) val,
(\mathfrak{A}, α) val \Rightarrow (\mathfrak{A}, β) val \Rightarrow (\mathfrak{A}, β) val]
where *OclIterate* S A F = ($\lambda \tau$. if (δ S) $\tau = \text{true}$ $\tau \wedge \text{finite}(\text{Rep-Bag-base } S \tau)$
then Finite-Set.fold (F o ($\lambda a \tau$. a) o fst) A (Rep-Bag-base S τ)
else \perp)

syntax

-*OclIterate* Bag :: [($\mathfrak{A}, \alpha :: \text{null}$) Bag, idt, idt, α , β] \Rightarrow (\mathfrak{A}, γ) val
($\langle - \rightarrow \text{iterate}_{\text{Bag}} '(-; == | -)' \rangle$)

syntax-consts

-*OclIterate* Bag == OclIterate

translations

X \rightarrow iterate_{Bag}(a; x = A | P) == CONST OclIterate X A (%a. (% x. P))

2.8.16. Definition: Select

definition *OclSelect* :: [($\mathfrak{A}, \alpha :: \text{null}$) Bag, (\mathfrak{A}, α) val \Rightarrow (\mathfrak{A}) Boolean] \Rightarrow (\mathfrak{A}, α) Bag
where *OclSelect* S P = ($\lambda \tau$. if (δ S) $\tau = \text{true}$ τ
then if ($\exists x \in \text{Rep-Set-base } S \tau$. P ($\lambda \cdot$. x) $\tau = \text{invalid}$ τ)
then invalid τ
else Abs-Bag_{base} \perp λx .
let n = $\prod^{\top} \text{Rep-Bag}_{\text{base}}(S \tau) \top x$ in
if n = 0 | P ($\lambda \cdot$. x) $\tau = \text{false}$ τ then
0
else
n \perp)

else invalid τ)

syntax

-*OclSelectBag* :: [($\mathfrak{A}, \alpha :: \text{null}$) Bag, id, (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean ($\langle (-) \rightarrow \text{select}_{Bag} '(-)' \rangle$)

syntax-consts

-*OclSelectBag* == *OclSelect*

translations

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

2.8.17. Definition: Reject

definition *OclReject* :: [($\mathfrak{A}, \alpha :: \text{null}$) Bag, (\mathfrak{A}, α) val \Rightarrow (\mathfrak{A}) Boolean] \Rightarrow ($\mathfrak{A}, \alpha :: \text{null}$) Bag

where *OclReject* *S P* = *OclSelect S (not o P)*

syntax

-*OclRejectBag* :: [($\mathfrak{A}, \alpha :: \text{null}$) Bag, id, (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean ($\langle (-) \rightarrow \text{reject}_{Bag} '(-)' \rangle$)

syntax-consts

-*OclRejectBag* == *OclReject*

translations

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

2.8.18. Definition: IncludesAll

definition *OclIncludesAll* :: [($\mathfrak{A}, \alpha :: \text{null}$) Bag, (\mathfrak{A}, α) Bag] \Rightarrow \mathfrak{A} Boolean

where *OclIncludesAll* $x y = (\lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
then $\perp\!\!\!\perp \text{Rep-Bag-base } y \tau \subseteq \text{Rep-Bag-base } x \tau \perp\!\!\!\perp$
else \perp)

notation *OclIncludesAll* ($\langle \rightarrow \text{includesAll}_{Bag} '(-)' \rangle$)

interpretation *OclIncludesAll* : *profile-bin_{d-d}* *OclIncludesAll* $\lambda x y. \perp\!\!\!\perp \text{Rep-Bag-base}' y \subseteq \text{Rep-Bag-base}' x \perp\!\!\!\perp$

by(*unfold-locales*, *auto simp: OclIncludesAll-def bot-option-def null-option-def invalid-def*
Rep-Bag-base-def Rep-Bag-base'-def)

2.8.19. Definition: ExcludesAll

definition *OclExcludesAll* :: [($\mathfrak{A}, \alpha :: \text{null}$) Bag, (\mathfrak{A}, α) Bag] \Rightarrow \mathfrak{A} Boolean

where *OclExcludesAll* $x y = (\lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
then $\perp\!\!\!\perp \text{Rep-Bag-base } y \tau \cap \text{Rep-Bag-base } x \tau = \{\} \perp\!\!\!\perp$
else \perp)

notation *OclExcludesAll* ($\langle \rightarrow \text{excludesAll}_{Bag} '(-)' \rangle$)

interpretation *OclExcludesAll* : *profile-bin_{d-d}* *OclExcludesAll* $\lambda x y. \perp\!\!\!\perp \text{Rep-Bag-base}' y \cap \text{Rep-Bag-base}' x = \{\} \perp\!\!\!\perp$

by(*unfold-locales*, *auto simp: OclExcludesAll-def bot-option-def null-option-def invalid-def*
Rep-Bag-base-def Rep-Bag-base'-def)

2.8.20. Definition: Union

definition *OclUnion* :: [($\mathfrak{A}, \alpha :: \text{null}$) Bag, (\mathfrak{A}, α) Bag] \Rightarrow (\mathfrak{A}, α) Bag

where *OclUnion* $x y = (\lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
then $\text{Abs-Bag}_{base} \perp\!\!\!\perp \lambda X. \ulcorner \text{Rep-Bag}_{base} (x \tau) \urcorner X +$
 $\ulcorner \text{Rep-Bag}_{base} (y \tau) \urcorner X \perp\!\!\!\perp$
else invalid τ)

notation *OclUnion* ($\langle \rightarrow \text{union}_{Bag} '(-)' \rangle$)

interpretation *OclUnion* :

profile-bin_{d-d} *OclUnion* $\lambda x y. \text{Abs-Bag}_{base} \perp\!\!\!\perp \lambda X. \ulcorner \text{Rep-Bag}_{base} x \urcorner X +$
 $\ulcorner \text{Rep-Bag}_{base} y \urcorner X \perp\!\!\!\perp$

proof –

show *profile-bin_{d-d}* *OclUnion* ($\lambda x y. \text{Abs-Bag}_{base} \perp\!\!\!\perp \lambda X. \ulcorner \text{Rep-Bag}_{base} x \urcorner X + \ulcorner \text{Rep-Bag}_{base} y \urcorner X \perp\!\!\!\perp$)

apply *unfold-locales*
apply(*auto simp:OclUnion-def bot-option-def null-option-def*
null-Bag_{base}-def bot-Bag_{base}-def)
by(*subst (asm) Abs-Bag_{base}-inject,*
simp-all add: bot-option-def null-option-def,
metis (mono-tags, lifting) Rep-Bag_{base} Rep-Bag_{base}-inverse bot-option-def mem-Collect-eq
null-option-def)⁺

qed

2.8.21. Definition: Intersection

definition *OclIntersection* :: [$(\mathfrak{A}, \alpha :: \text{null}) \text{ Bag}, (\mathfrak{A}, \alpha) \text{ Bag}$] $\Rightarrow (\mathfrak{A}, \alpha) \text{ Bag}$
where *OclIntersection* $x y = (\lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
then Abs-Bag_{base} $\perp\!\!\!\perp$ $\lambda X. \min ((\ulcorner \text{Rep-Bag}_{base} (x \tau) \urcorner) X)$
 $(\ulcorner \text{Rep-Bag}_{base} (y \tau) \urcorner) X \perp\!\!\!\perp$
else \perp)

notation *OclIntersection*($\langle \dashv \rangle \text{intersection}_{\text{Bag}} '(-)'$)

interpretation *OclIntersection* :

profile-bin_{d-d} OclIntersection $\lambda x y. \text{Abs-Bag}_{base} \perp\!\!\!\perp \lambda X. \min ((\ulcorner \text{Rep-Bag}_{base} x \urcorner) X)$
 $(\ulcorner \text{Rep-Bag}_{base} y \urcorner) X \perp\!\!\!\perp$

proof –

show *profile-bin_{d-d} OclIntersection* $(\lambda x y. \text{Abs-Bag}_{base} \perp\!\!\!\perp \lambda X. \min ((\ulcorner \text{Rep-Bag}_{base} x \urcorner) X)$
 $(\ulcorner \text{Rep-Bag}_{base} y \urcorner) X \perp\!\!\!\perp)$

apply *unfold-locales*

apply(*auto simp:OclIntersection-def bot-option-def null-option-def*
null-Bag_{base}-def bot-Bag_{base}-def invalid-def)

by(*subst (asm) Abs-Bag_{base}-inject,*
simp-all add: bot-option-def null-option-def,
metis (mono-tags, lifting) Rep-Bag_{base} Rep-Bag_{base}-inverse bot-option-def mem-Collect-eq min-OR
null-option-def)⁺

qed

2.8.22. Definition: Count

definition *OclCount* :: [$(\mathfrak{A}, \alpha :: \text{null}) \text{ Bag}, (\mathfrak{A}, \alpha) \text{ val}$] $\Rightarrow (\mathfrak{A}) \text{ Integer}$
where *OclCount* $x y = (\lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
then $\perp\!\!\!\perp \text{int}((\ulcorner \text{Rep-Bag}_{base} (x \tau) \urcorner) (y \tau)) \perp\!\!\!\perp$
else invalid τ)

notation *OclCount* ($\langle \dashv \rangle \text{count}_{\text{Bag}} '(-)'$)

interpretation *OclCount* : *profile-bin_{d-d} OclCount* $\lambda x y. \perp\!\!\!\perp \text{int}((\ulcorner \text{Rep-Bag}_{base} x \urcorner) y) \perp\!\!\!\perp$

by(*unfold-locales, auto simp:OclCount-def bot-option-def null-option-def*)

2.8.23. Definition (future operators)

consts

OclSum :: $(\mathfrak{A}, \alpha :: \text{null}) \text{ Bag} \Rightarrow \mathfrak{A} \text{ Integer}$

notation *OclSum* ($\langle \dashv \rangle \text{sum}_{\text{Bag}} '(-)'$)

2.8.24. Logical Properties

OclIncluding

lemma *OclIncluding-valid-args-valid*:

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

by (*metis (opaque-lifting, no-types) OclIncluding.def-valid-then-def OclIncluding.defined-args-valid*)

lemma *OclIncluding-valid-args-valid''[simp,code-unfold]*:

$v(X \rightarrow \text{including}_{Bag}(x)) = ((\delta X) \text{ and } (v x))$

by (*simp add: OclIncluding.def-valid-then-def*)

etc. etc.

OclExcluding

lemma *OclExcluding-valid-args-valid*:

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

by (*metis OclExcluding.def-valid-then-def OclExcluding.defined-args-valid*)

lemma *OclExcluding-valid-args-valid''[simp,code-unfold]*:

$v(X \rightarrow \text{excluding}_{Bag}(x)) = ((\delta X) \text{ and } (v x))$

by (*simp add: OclExcluding.def-valid-then-def*)

OclIncludes

lemma *OclIncludes-valid-args-valid*:

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

by (*simp add: OclIncludes.def-valid-then-def foundation10'*)

lemma *OclIncludes-valid-args-valid''[simp,code-unfold]*:

$v(X \rightarrow \text{includes}_{Bag}(x)) = ((\delta X) \text{ and } (v x))$

by (*simp add: OclIncludes.def-valid-then-def*)

OclExcludes

lemma *OclExcludes-valid-args-valid*:

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

by (*simp add: OclExcludes.def-valid-then-def foundation10'*)

lemma *OclExcludes-valid-args-valid''[simp,code-unfold]*:

$v(X \rightarrow \text{excludes}_{Bag}(x)) = ((\delta X) \text{ and } (v x))$

by (*simp add: OclExcludes.def-valid-then-def*)

OclSize

lemma *OclSize-defined-args-valid*: $\tau \models \delta (X \rightarrow \text{size}_{Bag}()) \implies \tau \models \delta X$

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

lemma *OclSize-infinite*:

assumes *non-finite*: $\tau \models \text{not}(\delta(S \rightarrow \text{size}_{Bag}()))$

shows $(\tau \models \text{not}(\delta(S))) \vee \neg \text{finite} (\text{Rep-Bag-base } S \tau)$

apply(*insert non-finite, simp*)

apply(*rule impI*)

apply(*simp add: OclSize-def OclValid-def defined-def bot-fun-def null-fun-def bot-option-def null-option-def
split: if-split-asm*)

done

lemma $\tau \models \delta X \implies \neg \text{finite} (\text{Rep-Bag-base } X \tau) \implies \neg \tau \models \delta (X \rightarrow \text{size}_{Bag}())$

by(*simp add: OclSize-def OclValid-def defined-def bot-fun-def false-def true-def*)

lemma *size-defined*:

assumes *X-finite*: $\bigwedge \tau. \text{finite} (\text{Rep-Bag-base } X \tau)$

shows $\delta (X \rightarrow \text{size}_{Bag}()) = \delta X$

apply(*rule ext, simp add: cp-defined[of X \rightarrow size_{Bag}()] OclSize-def*)

apply(*simp add: defined-def bot-option-def bot-fun-def null-option-def null-fun-def X-finite*)

done

lemma *size-defined'*:

assumes *X-finite*: *finite (Rep-Bag-base X τ)*
shows $(\tau \models \delta (X \rightarrow \text{size}_{\text{Bag}}())) = (\tau \models \delta X)$
apply(*simp add: cp-defined[of X \rightarrow size_{Bag}()]* *OclSize-def OclValid-def*)
apply(*simp add: defined-def bot-option-def bot-fun-def null-option-def null-fun-def X-finite*)
done

OclIsEmpty

lemma *OclIsEmpty-defined-args-valid*: $\tau \models \delta (X \rightarrow \text{isEmpty}_{\text{Bag}}()) \implies \tau \models v X$
apply(*auto simp: OclIsEmpty-def OclValid-def defined-def valid-def false-def true-def*
bot-fun-def null-fun-def OclAnd-def OclOr-def OclNot-def
split: if-split-asm)
apply(*case-tac (X \rightarrow size_{Bag}() \doteq 0) τ , simp add: bot-option-def, simp, rename-tac x*)
apply(*case-tac x, simp add: null-option-def bot-option-def, simp*)
apply(*simp add: OclSize-def StrictRefEqInteger valid-def*)
by (*metis (opaque-lifting, no-types)*
bot-fun-def OclValid-def defined-def foundation2 invalid-def)

lemma $\tau \models \delta (\text{null} \rightarrow \text{isEmpty}_{\text{Bag}}())$

by(*auto simp: OclIsEmpty-def OclValid-def defined-def valid-def false-def true-def*
bot-fun-def null-fun-def OclAnd-def OclOr-def OclNot-def null-is-valid
split: if-split-asm)

lemma *OclIsEmpty-infinite*: $\tau \models \delta X \implies \neg \text{finite} (\text{Rep-Bag-base } X \tau) \implies \neg \tau \models \delta (X \rightarrow \text{isEmpty}_{\text{Bag}}())$

apply(*auto simp: OclIsEmpty-def OclValid-def defined-def valid-def false-def true-def*
bot-fun-def null-fun-def OclAnd-def OclOr-def OclNot-def
split: if-split-asm)
apply(*case-tac (X \rightarrow size_{Bag}() \doteq 0) τ , simp add: bot-option-def, simp, rename-tac x*)
apply(*case-tac x, simp add: null-option-def bot-option-def, simp*)
by(*simp add: OclSize-def StrictRefEqInteger valid-def bot-fun-def false-def true-def invalid-def*)

OclNotEmpty

lemma *OclNotEmpty-defined-args-valid*: $\tau \models \delta (X \rightarrow \text{notEmpty}_{\text{Bag}}()) \implies \tau \models v X$

by (*metis (opaque-lifting, no-types) OclNotEmpty-def OclNot-defargs OclNot-not foundation6 foundation9*
OclIsEmpty-defined-args-valid)

lemma $\tau \models \delta (\text{null} \rightarrow \text{notEmpty}_{\text{Bag}}())$

by (*metis (opaque-lifting, no-types) OclNotEmpty-def OclAnd-false1 OclAnd-idem OclIsEmpty-def*
OclNot3 OclNot4 OclOr-def defined2 defined4 transform1 valid2)

lemma *OclNotEmpty-infinite*: $\tau \models \delta X \implies \neg \text{finite} (\text{Rep-Bag-base } X \tau) \implies \neg \tau \models \delta (X \rightarrow \text{notEmpty}_{\text{Bag}}())$

apply(*simp add: OclNotEmpty-def*)
apply(*drule OclIsEmpty-infinite, simp*)
by (*metis OclNot-defargs OclNot-not foundation6 foundation9*)

lemma *OclNotEmpty-has-elt* : $\tau \models \delta X \implies$
 $\tau \models X \rightarrow \text{notEmpty}_{\text{Bag}}() \implies$
 $\exists e. e \in (\text{Rep-Bag-base } X \tau)$

proof –

have *s-non-empty*: $\bigwedge S. S \neq \{\} \implies \exists x. x \in S$

by *blast*

show $\tau \models \delta X \implies$

$\tau \models X \rightarrow \text{notEmpty}_{\text{Bag}}() \implies$

?thesis

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: if-split-asm)
prefer 2
apply(*simp add: invalid-def bot-option-def true-def*)
apply(*simp add: OclSize-def valid-def split: if-split-asm,*
simp-all add: false-def true-def bot-option-def bot-fun-def OclInt0-def)
apply(*drule s-non-empty[of Rep-Bag-base X τ], erule exE, case-tac x*)
by *blast*
qed

lemma *OclNotEmpty-has-elt' : $\tau \models \delta X \implies$*

$$\tau \models X \rightarrow \text{notEmpty}_{Bag}() \implies$$

$$\exists e. e \in (\text{Rep-Set-base } X \ \tau)$$
apply(*drule OclNotEmpty-has-elt, simp*)
by(*simp add: Rep-Bag-base-def Rep-Set-base-def image-def*)

OclANY

lemma *OclANY-defined-args-valid: $\tau \models \delta (X \rightarrow \text{any}_{Bag}()) \implies \tau \models \delta X$*
by(*auto simp: OclANY-def OclValid-def true-def valid-def false-def StrongEq-def*
defined-def invalid-def bot-fun-def null-fun-def OclAnd-def
split: bool.split-asm HOL.if-split-asm option.split)

lemma $\tau \models \delta X \implies \tau \models X \rightarrow \text{isEmpty}_{Bag}() \implies \neg \tau \models \delta (X \rightarrow \text{any}_{Bag}())$
apply(*simp add: OclANY-def OclValid-def*)
apply(*subst cp-defined, subst cp-OclAnd, simp add: OclNotEmpty-def, subst (1 2) cp-OclNot,*
simp add: cp-OclNot[symmetric] cp-OclAnd[symmetric] cp-defined[symmetric],
simp add: false-def true-def)
by(*drule foundation20[simplified OclValid-def true-def], simp*)

lemma *OclANY-valid-args-valid:*
 $(\tau \models v(X \rightarrow \text{any}_{Bag}())) = (\tau \models v X)$

proof –

have *A: $(\tau \models v(X \rightarrow \text{any}_{Bag}())) \implies ((\tau \models (v X)))$*
by(*auto simp: OclANY-def OclValid-def true-def valid-def false-def StrongEq-def*
defined-def invalid-def bot-fun-def null-fun-def
split: bool.split-asm HOL.if-split-asm option.split)

have *B: $(\tau \models (v X)) \implies (\tau \models v(X \rightarrow \text{any}_{Bag}()))$*
apply(*auto simp: OclANY-def OclValid-def true-def false-def StrongEq-def*
defined-def invalid-def valid-def bot-fun-def null-fun-def
bot-option-def null-option-def null-is-valid
OclAnd-def
split: bool.split-asm HOL.if-split-asm option.split)

apply(*frule Bag-inv-lemma[OF foundation16[THEN iffD2], OF conjI], simp*)

apply(*subgoal-tac (δX) $\tau = \text{true } \tau$)*

prefer 2

apply(*metis (opaque-lifting, no-types) OclValid-def foundation16*)

apply(*simp add: true-def,*

drule OclNotEmpty-has-elt'[simplified OclValid-def true-def], simp)

apply(*erule exE,*

rule someI2[where $Q = \lambda x. x \neq \perp$ and $P = \lambda y. y \in (\text{Rep-Set-base } X \ \tau),$
simplified not-def, THEN mp], simp, auto)

by(*simp add: Rep-Set-base-def image-def*)

show *?thesis by(auto dest:A intro:B)*

qed

lemma *OclANY-valid-args-valid''[simp,code-unfold]:*

$v(X \rightarrow \text{any}_{Bag}()) = (v X)$
by(*auto intro!*: *OclANY-valid-args-valid transform2-rev*)

2.8.25. Execution Laws with Invalid or Null or Infinite Set as Argument

OclIncluding

OclExcluding

OclIncludes

OclExcludes

OclSize

lemma *OclSize-invalid*[*simp,code-unfold*]:(*invalid* \rightarrow *size*_{Bag}()) = *invalid*
by(*simp add: bot-fun-def OclSize-def invalid-def defined-def valid-def false-def true-def*)

lemma *OclSize-null*[*simp,code-unfold*]:(*null* \rightarrow *size*_{Bag}()) = *invalid*

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

OclIsEmpty

lemma *OclIsEmpty-invalid*[*simp,code-unfold*]:(*invalid* \rightarrow *isEmpty*_{Bag}()) = *invalid*
by(*simp add: OclIsEmpty-def*)

lemma *OclIsEmpty-null*[*simp,code-unfold*]:(*null* \rightarrow *isEmpty*_{Bag}()) = *true*
by(*simp add: OclIsEmpty-def*)

OclNotEmpty

lemma *OclNotEmpty-invalid*[*simp,code-unfold*]:(*invalid* \rightarrow *notEmpty*_{Bag}()) = *invalid*
by(*simp add: OclNotEmpty-def*)

lemma *OclNotEmpty-null*[*simp,code-unfold*]:(*null* \rightarrow *notEmpty*_{Bag}()) = *false*
by(*simp add: OclNotEmpty-def*)

OclANY

lemma *OclANY-invalid*[*simp,code-unfold*]:(*invalid* \rightarrow *any*_{Bag}()) = *invalid*
by(*simp add: bot-fun-def OclANY-def invalid-def defined-def valid-def false-def true-def*)

lemma *OclANY-null*[*simp,code-unfold*]:(*null* \rightarrow *any*_{Bag}()) = *null*
by(*simp add: OclANY-def false-def true-def*)

OclForall

lemma *OclForall-invalid*[*simp,code-unfold*]:*invalid* \rightarrow *forAll*_{Bag}(*a* | *P a*) = *invalid*
by(*simp add: bot-fun-def invalid-def OclForall-def defined-def valid-def false-def true-def*)

lemma *OclForall-null*[*simp,code-unfold*]:*null* \rightarrow *forAll*_{Bag}(*a* | *P a*) = *invalid*
by(*simp add: bot-fun-def invalid-def OclForall-def defined-def valid-def false-def true-def*)

OclExists

lemma *OclExists-invalid*[*simp,code-unfold*]:*invalid* \rightarrow *exists*_{Bag}(*a* | *P a*) = *invalid*
by(*simp add: OclExists-def*)

lemma *OclExists-null*[*simp,code-unfold*]:*null* \rightarrow *exists*_{Bag}(*a* | *P a*) = *invalid*
by(*simp add: OclExists-def*)

OclIterate

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

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

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

An open question is this ...

lemma *S*→*iterate*_{Bag}(*a*; *x* = *null* | *P a x*) = *invalid*
oops

lemma *OclIterate-infinite*:
assumes *non-finite*: $\tau \models \text{not}(\delta(S \rightarrow \text{size}_{\text{Bag}}()))$
shows (*OclIterate S A F*) $\tau = \text{invalid } \tau$
apply(*insert non-finite [THEN OclSize-infinite]*)
apply(*subst (asm) foundation9, simp*)
by(*metis OclIterate-def OclValid-def invalid-def*)

OclSelect

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

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

OclReject

lemma *OclReject-invalid*[simp,code-unfold]:*invalid*→*reject*_{Bag}(*a* | *P a*) = *invalid*
by(*simp add*: *OclReject-def*)

lemma *OclReject-null*[simp,code-unfold]:*null*→*reject*_{Bag}(*a* | *P a*) = *invalid*
by(*simp add*: *OclReject-def*)

Context Passing

lemma *cp-OclIncludes1*:
(*X*→*includes*_{Bag}(*x*)) $\tau = (X \rightarrow \text{includes}_{\text{Bag}}(\lambda \cdot. x \tau)) \tau$
by(*auto simp: OclIncludes-def StrongEq-def invalid-def*
cp-defined[symmetric] cp-valid[symmetric])

lemma *cp-OclSize*: *X*→*size*_{Bag}() $\tau = ((\lambda \cdot. X \tau) \rightarrow \text{size}_{\text{Bag}}()) \tau$
by(*simp add: OclSize-def cp-defined[symmetric] Rep-Bag-base-def*)

lemma *cp-OclIsEmpty*: *X*→*isEmpty*_{Bag}() $\tau = ((\lambda \cdot. X \tau) \rightarrow \text{isEmpty}_{\text{Bag}}()) \tau$
apply(*simp only: OclIsEmpty-def*)
apply(*subst (2) cp-OclOr,*
subst cp-OclAnd,
subst cp-OclNot,
subst StrictRefEqInteger.cp0)
by(*simp add: cp-defined[symmetric] cp-valid[symmetric] StrictRefEqInteger.cp0[symmetric]*
cp-OclSize[symmetric] cp-OclNot[symmetric] cp-OclAnd[symmetric] cp-OclOr[symmetric])

lemma *cp-OclNotEmpty*: *X*→*notEmpty*_{Bag}() $\tau = ((\lambda \cdot. X \tau) \rightarrow \text{notEmpty}_{\text{Bag}}()) \tau$
apply(*simp only: OclNotEmpty-def*)

apply(subst (2) cp-OclNot)
by(simp add: cp-OclNot[symmetric] cp-OclIsEmpty[symmetric])

lemma cp-OclANY: $X \rightarrow \text{any}_{Bag}() \tau = ((\lambda \cdot. X \tau) \rightarrow \text{any}_{Bag}()) \tau$
apply(simp only: OclANY-def)
apply(subst (2) cp-OclAnd)
by(simp only: cp-OclAnd[symmetric] cp-defined[symmetric] cp-valid[symmetric]
cp-OclNotEmpty[symmetric] Rep-Set-base-def)

lemma cp-OclForall:
 $(S \rightarrow \text{forAll}_{Bag}(x \mid P x)) \tau = ((\lambda \cdot. S \tau) \rightarrow \text{forAll}_{Bag}(x \mid P (\lambda \cdot. x \tau))) \tau$
by(auto simp add: OclForall-def cp-defined[symmetric] Rep-Set-base-def)

lemma cp-OclForall1 [simp,intro]:
 $cp S \implies cp (\lambda X. ((S X) \rightarrow \text{forAll}_{Bag}(x \mid P x)))$
apply(simp add: cp-def)
apply(erule exE, rule exI, intro allI)
apply(erule-tac x=X in allE)
by(subst cp-OclForall, simp)

lemma
 $cp (\lambda X St x. P (\lambda \tau. x) X St) \implies cp S \implies cp (\lambda X. (S X) \rightarrow \text{forAll}_{Bag}(x \mid P x X))$
apply(simp only: cp-def)
oops

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

lemma cp-OclExists:
 $(S \rightarrow \text{exists}_{Bag}(x \mid P x)) \tau = ((\lambda \cdot. S \tau) \rightarrow \text{exists}_{Bag}(x \mid P (\lambda \cdot. x \tau))) \tau$
by(simp add: OclExists-def OclNot-def, subst cp-OclForall, simp)

lemma cp-OclExists1 [simp,intro]:
 $cp S \implies cp (\lambda X. ((S X) \rightarrow \text{exists}_{Bag}(x \mid P x)))$
apply(simp add: cp-def)
apply(erule exE, rule exI, intro allI)
apply(erule-tac x=X in allE)
by(subst cp-OclExists, simp)

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

lemma cp-OclSelect: $(X \rightarrow \text{select}_{Bag}(a \mid P a)) \tau =$
 $((\lambda \cdot. X \tau) \rightarrow \text{select}_{Bag}(a \mid P a)) \tau$
by(simp add: OclSelect-def cp-defined[symmetric] Rep-Set-base-def)

lemma cp-OclReject: $(X \rightarrow \text{reject}_{Bag}(a \mid P a)) \tau = ((\lambda \cdot. X \tau) \rightarrow \text{reject}_{Bag}(a \mid P a)) \tau$
by(simp add: OclReject-def, subst cp-OclSelect, simp)

lemmas cp-intro''_{Bag}[intro!,simp,code-unfold] =

```

cp-OclSize      [THEN allI[THEN allI[THEN cpI1], of OclSize]]
cp-OclIsEmpty   [THEN allI[THEN allI[THEN cpI1], of OclIsEmpty]]
cp-OclNotEmpty  [THEN allI[THEN allI[THEN cpI1], of OclNotEmpty]]
cp-OclANY       [THEN allI[THEN allI[THEN cpI1], of OclANY]]

```

Const

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

assumes *const-x* : *const x*

and *const-S* : *const S*

shows *const (S->including_{Bag}(x))*

proof –

have $A: \bigwedge \tau \tau'. \neg (\tau \models v x) \implies (S \text{->including}_{Bag}(x) \tau) = (S \text{->including}_{Bag}(x) \tau')$

apply(simp add: foundation18)

apply(erule const-subst[OF const-x const-invalid],simp-all)

by(rule const-chnr[OF const-invalid])

have $B: \bigwedge \tau \tau'. \neg (\tau \models \delta S) \implies (S \text{->including}_{Bag}(x) \tau) = (S \text{->including}_{Bag}(x) \tau')$

apply(simp add: foundation16', elim disjE)

apply(erule const-subst[OF const-S const-invalid],simp-all)

apply(rule const-chnr[OF const-invalid])

apply(erule const-subst[OF const-S const-null],simp-all)

by(rule const-chnr[OF const-invalid])

show ?thesis

apply(simp only: const-def,intro allI, rename-tac $\tau \tau'$)

apply(case-tac $\neg (\tau \models v x)$, simp add: A)

apply(case-tac $\neg (\tau \models \delta S)$, simp-all add: B)

apply(frule-tac $\tau'1 = \tau'$ in const-OclValid2[OF const-x, THEN iffD1])

apply(frule-tac $\tau'1 = \tau'$ in const-OclValid1[OF const-S, THEN iffD1])

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

apply(subst (1 2) const-chnr[OF const-x])

apply(subst (1 2) const-chnr[OF const-S])

by simp

qed

2.8.26. Test Statements

instantiation *Bag_{base}* :: (equal)equal

begin

definition *HOL.equal* $k l \longleftrightarrow (k::('a::equal)Bag_{base}) = l$

instance **by** standard (rule equal-Bag_{base}-def)

end

lemma *equal-Bag_{base}-code* [code]:

HOL.equal $k (l::('a::\{equal,null\})Bag_{base}) \longleftrightarrow Rep\text{-}Bag_{base} k = Rep\text{-}Bag_{base} l$

by (auto simp add: equal Bag_{base}.Rep-Bag_{base}-inject)

Assert $\tau \models (Bag\{\} \doteq Bag\{\})$

end

theory *UML-Set*

imports ../basic-types/UML-Void

```

../basic-types/UML-Boolean
../basic-types/UML-Integer
../basic-types/UML-String
../basic-types/UML-Real

```

begin

no-notation *None* ($\langle \perp \rangle$)

2.9. Collection Type Set: Operations

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

Our notion of typed set goes beyond the usual notion of a finite executable set and is powerful enough to capture *the extension of a type* in UML and OCL. This means we can have in Featherweight OCL Sets containing all possible elements of a type, not only those (finite) ones representable in a state. This holds for base types as well as class types, although the notion for class-types — involving object id's not occurring in a state — requires some care.

In a world with *invalid* and *null*, there are two notions extensions possible:

1. the set of all *defined* values of a type T (for which we will introduce the constant T)
2. the set of all *valid* values of a type T , so including *null* (for which we will introduce the constant T_{null}).

We define the set extensions for the base type *Integer* as follows:

definition *Integer* :: ($\mathfrak{A}, \text{Integer}_{base}$) Set

where $\text{Integer} \equiv (\lambda \tau. (\text{Abs-Set}_{base} \circ \text{Some} \circ \text{Some})) ((\text{Some} \circ \text{Some}) '(\text{UNIV}::\text{int set}))$

definition *Integer_{null}* :: ($\mathfrak{A}, \text{Integer}_{base}$) Set

where $\text{Integer}_{null} \equiv (\lambda \tau. (\text{Abs-Set}_{base} \circ \text{Some} \circ \text{Some})) (\text{Some} '(\text{UNIV}::\text{int option set}))$

lemma *Integer-defined* : $\delta \text{Integer} = \text{true}$

apply(rule *ext*, auto *simp*: *Integer-def defined-def false-def true-def*
bot-fun-def null-fun-def null-option-def)

by(*simp-all add: Abs-Set_{base}-inject bot-option-def bot-Set_{base}-def null-Set_{base}-def null-option-def*)

lemma *Integer_{null}-defined* : $\delta \text{Integer}_{null} = \text{true}$

apply(rule *ext*, auto *simp*: *Integer_{null}-def defined-def false-def true-def*
bot-fun-def null-fun-def null-option-def)

by(*simp-all add: Abs-Set_{base}-inject bot-option-def bot-Set_{base}-def null-Set_{base}-def null-option-def*)

This allows the theorems:

$\tau \models \delta x \implies \tau \models (\text{Integer} \rightarrow \text{includes}_{Set}(x))$ $\tau \models \delta x \implies \tau \models \text{Integer} \triangleq (\text{Integer} \rightarrow \text{including}_{Set}(x))$
and
 $\tau \models v x \implies \tau \models (\text{Integer}_{null} \rightarrow \text{includes}_{Set}(x))$ $\tau \models v x \implies \tau \models \text{Integer}_{null} \triangleq$
 $(\text{Integer}_{null} \rightarrow \text{including}_{Set}(x))$

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

In the same spirit, we proceed similarly for the remaining base types:

definition *Void_{null}* :: ($\mathfrak{A}, \text{Void}_{base}$) Set

where $\text{Void}_{null} \equiv (\lambda \tau. (\text{Abs-Set}_{base} \circ \text{Some} \circ \text{Some})) \{\text{Abs-Void}_{base} (\text{Some None})\}$

definition *Void_{empty}* :: ($\mathfrak{A}, \text{Void}_{base}$) Set

where $\text{Void}_{empty} \equiv (\lambda \tau. (\text{Abs-Set}_{base} \circ \text{Some} \circ \text{Some})) \{\}$

```

lemma Voidnull-defined :  $\delta$  Voidnull = true
apply(rule ext, auto simp: Voidnull-def defined-def false-def true-def
      bot-fun-def null-fun-def null-option-def
      bot-Setbase-def null-Setbase-def)
by((subst (asm) Abs-Setbase-inject, auto simp add: bot-option-def null-option-def bot-Void-def),
     (subst (asm) Abs-Voidbase-inject, auto simp add: bot-option-def null-option-def))+

lemma Voidempty-defined :  $\delta$  Voidempty = true
apply(rule ext, auto simp: Voidempty-def defined-def false-def true-def
      bot-fun-def null-fun-def null-option-def
      bot-Setbase-def null-Setbase-def)
by((subst (asm) Abs-Setbase-inject, auto simp add: bot-option-def null-option-def bot-Void-def))+

lemma assumes  $\tau \models \delta$  (V :: ( $\mathfrak{A}$ , Voidbase) Set)
  shows  $\tau \models V \triangleq \text{Void}_{null} \vee \tau \models V \triangleq \text{Void}_{empty}$ 
proof -
  have  $A: \bigwedge x y. x \neq \{\} \implies \exists y. y \in x$ 
  by (metis all-not-in-conv)
show ?thesis
  apply(case-tac V  $\tau$ )
  proof - fix y show  $V \tau = \text{Abs-Set}_{base} y \implies$ 
     $y \in \{X. X = \perp \vee X = \text{null} \vee (\forall x \in \ulcorner X \urcorner. x \neq \perp)\} \implies$ 
     $\tau \models V \triangleq \text{Void}_{null} \vee \tau \models V \triangleq \text{Void}_{empty}$ 
  apply(insert assms, case-tac y, simp add: bot-option-def, simp add: bot-Setbase-def foundation16)
  apply(simp add: bot-option-def null-option-def)
  apply(erule disjE, metis OclValid-def defined-def foundation2 null-Setbase-def null-fun-def true-def)
  proof - fix a show  $V \tau = \text{Abs-Set}_{base} \ulcorner a \urcorner \implies \forall x \in \ulcorner a \urcorner. x \neq \perp \implies \tau \models V \triangleq \text{Void}_{null} \vee \tau \models V \triangleq \text{Void}_{empty}$ 
  apply(case-tac a, simp, insert assms, metis OclValid-def foundation16 null-Setbase-def true-def)
  apply(simp)
  proof - fix aa show  $V \tau = \text{Abs-Set}_{base} \ulcorner aa \urcorner \implies \forall x \in aa. x \neq \perp \implies \tau \models V \triangleq \text{Void}_{null} \vee \tau \models V \triangleq \text{Void}_{empty}$ 
  apply(case-tac aa = \{\},
    rule disjI2,
    insert assms,
    simp add: Voidempty-def OclValid-def StrongEq-def true-def,
    rule disjI1)
  apply(subgoal-tac aa = \{Abs-Voidbase \ulcorner None \urcorner\}, simp add: StrongEq-def OclValid-def true-def Voidnull-def)
  apply(drule A, erule exE)
  proof - fix y show  $V \tau = \text{Abs-Set}_{base} \ulcorner aa \urcorner \implies$ 
     $\forall x \in aa. x \neq \perp \implies$ 
     $\tau \models \delta V \implies$ 
     $y \in aa \implies$ 
     $aa = \{Abs-Void_{base} \ulcorner None \urcorner\}$ 
  apply(rule equalityI, rule subsetI, simp)
  proof - fix x show  $V \tau = \text{Abs-Set}_{base} \ulcorner aa \urcorner \implies$ 
     $\forall x \in aa. x \neq \perp \implies \tau \models \delta V \implies y \in aa \implies x \in aa \implies x = \text{Abs-Void}_{base} \ulcorner None \urcorner$ 
  apply(case-tac x, simp)
  by (metis bot-Void-def bot-option-def null-option-def)
apply-end(simp-all)

apply-end(erule ballE[where x = y], simp-all)
apply-end(case-tac y,
  simp add: bot-option-def null-option-def OclValid-def defined-def split: if-split-asm,
  simp add: false-def true-def)
  qed (erule disjE, simp add: bot-Void-def, simp)
qed qed qed qed qed

```

definition *Boolean* :: (' \mathcal{A} , Boolean_{base}) Set
where *Boolean* $\equiv (\lambda \tau. (\text{Abs-Set}_{\text{base}} \circ \text{Some} \circ \text{Some}) ((\text{Some} \circ \text{Some}) '(\text{UNIV}::\text{bool set})))$

definition *Boolean_{null}* :: (' \mathcal{A} , Boolean_{base}) Set
where *Boolean_{null}* $\equiv (\lambda \tau. (\text{Abs-Set}_{\text{base}} \circ \text{Some} \circ \text{Some}) (\text{Some} '(\text{UNIV}::\text{bool option set})))$

lemma *Boolean-defined* : $\delta \text{ Boolean} = \text{true}$
apply(rule ext, auto simp: Boolean-def defined-def false-def true-def
bot-fun-def null-fun-def null-option-def)
by(simp-all add: Abs-Set_{base}-inject bot-option-def bot-Set_{base}-def null-Set_{base}-def null-option-def)

lemma *Boolean_{null}-defined* : $\delta \text{ Boolean}_{\text{null}} = \text{true}$
apply(rule ext, auto simp: Boolean_{null}-def defined-def false-def true-def
bot-fun-def null-fun-def null-option-def)
by(simp-all add: Abs-Set_{base}-inject bot-option-def bot-Set_{base}-def null-Set_{base}-def null-option-def)

definition *String* :: (' \mathcal{A} , String_{base}) Set
where *String* $\equiv (\lambda \tau. (\text{Abs-Set}_{\text{base}} \circ \text{Some} \circ \text{Some}) ((\text{Some} \circ \text{Some}) '(\text{UNIV}::\text{string set})))$

definition *String_{null}* :: (' \mathcal{A} , String_{base}) Set
where *String_{null}* $\equiv (\lambda \tau. (\text{Abs-Set}_{\text{base}} \circ \text{Some} \circ \text{Some}) (\text{Some} '(\text{UNIV}::\text{string option set})))$

lemma *String-defined* : $\delta \text{ String} = \text{true}$
apply(rule ext, auto simp: String-def defined-def false-def true-def
bot-fun-def null-fun-def null-option-def)
by(simp-all add: Abs-Set_{base}-inject bot-option-def bot-Set_{base}-def null-Set_{base}-def null-option-def)

lemma *String_{null}-defined* : $\delta \text{ String}_{\text{null}} = \text{true}$
apply(rule ext, auto simp: String_{null}-def defined-def false-def true-def
bot-fun-def null-fun-def null-option-def)
by(simp-all add: Abs-Set_{base}-inject bot-option-def bot-Set_{base}-def null-Set_{base}-def null-option-def)

definition *Real* :: (' \mathcal{A} , Real_{base}) Set
where *Real* $\equiv (\lambda \tau. (\text{Abs-Set}_{\text{base}} \circ \text{Some} \circ \text{Some}) ((\text{Some} \circ \text{Some}) '(\text{UNIV}::\text{real set})))$

definition *Real_{null}* :: (' \mathcal{A} , Real_{base}) Set
where *Real_{null}* $\equiv (\lambda \tau. (\text{Abs-Set}_{\text{base}} \circ \text{Some} \circ \text{Some}) (\text{Some} '(\text{UNIV}::\text{real option set})))$

lemma *Real-defined* : $\delta \text{ Real} = \text{true}$
apply(rule ext, auto simp: Real-def defined-def false-def true-def
bot-fun-def null-fun-def null-option-def)
by(simp-all add: Abs-Set_{base}-inject bot-option-def bot-Set_{base}-def null-Set_{base}-def null-option-def)

lemma *Real_{null}-defined* : $\delta \text{ Real}_{\text{null}} = \text{true}$
apply(rule ext, auto simp: Real_{null}-def defined-def false-def true-def
bot-fun-def null-fun-def null-option-def)
by(simp-all add: Abs-Set_{base}-inject bot-option-def bot-Set_{base}-def null-Set_{base}-def null-option-def)

2.9.2. Basic Properties of the Set Type

Every element in a defined set is valid.

lemma *Set-inv-lemma*: $\tau \models (\delta X) \implies \forall x \in {}^{\tau}\text{Rep-Set}_{\text{base}}(X \tau)^{\tau}. x \neq \text{bot}$
apply(insert Rep-Set_{base} [of X τ], simp)
apply(auto simp: OclValid-def defined-def false-def true-def cp-def
bot-fun-def bot-Set_{base}-def null-Set_{base}-def null-fun-def
split:if-split-asm)

```

apply(erule contrapos-pp [of Rep-Setbase (X τ) = bot])
apply(subst Abs-Setbase-inject[symmetric], rule Rep-Setbase, simp)
apply(simp add: Rep-Setbase-inverse bot-Setbase-def bot-option-def)
apply(erule contrapos-pp [of Rep-Setbase (X τ) = null])
apply(subst Abs-Setbase-inject[symmetric], rule Rep-Setbase, simp)
apply(simp add: Rep-Setbase-inverse null-option-def)
by (simp add: bot-option-def)

```

```

lemma Set-inv-lemma' :
assumes x-def : τ ⊨ δ X
  and e-mem : e ∈ ⊢Rep-Setbase (X τ)⊢
  shows τ ⊨ v (λ-. e)
apply(rule Set-inv-lemma[OF x-def, THEN ballE[where x = e]])
apply(simp add: foundation18')
by(simp add: e-mem)

```

```

lemma abs-rep-simp' :
assumes S-all-def : τ ⊨ δ S
  shows Abs-Setbase ⊥ ⊢Rep-Setbase (S τ)⊢ = S τ
proof -
have discr-eq-false-true : ∧τ. (false τ = true τ) = False by(simp add: false-def true-def)
show ?thesis
  apply(insert S-all-def, simp add: OclValid-def defined-def)
  apply(rule mp[OF Abs-Setbase-induct[where P = λS. (if S = ⊥ τ ∨ S = null τ
    then false τ else true τ) = true τ →
    Abs-Setbase ⊥ ⊢Rep-Setbase S⊢ = S]],
    rename-tac S')
  apply(simp add: Abs-Setbase-inverse discr-eq-false-true)
  apply(case-tac S') apply(simp add: bot-fun-def bot-Setbase-def)+
  apply(rename-tac S'', case-tac S'') apply(simp add: null-fun-def null-Setbase-def)+
done
qed

```

```

lemma S-lift' :
assumes S-all-def : (τ :: 'A st) ⊨ δ S
  shows ∃ S'. (λa (:-: 'A st). a) ⊢Rep-Setbase (S τ)⊢ = (λa (:-: 'A st). ⊥a) ' S'
  apply(rule-tac x = (λa. ⊢ a) ⊢Rep-Setbase (S τ)⊢ in exI)
  apply(simp only: image-comp)
  apply(simp add: comp-def)
  apply(rule image-cong, fast)

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

```

```

lemma invalid-set-OclNot-defined [simp,code-unfold]:δ(invalid::('A,'α::null) Set) = false by simp
lemma null-set-OclNot-defined [simp,code-unfold]:δ(null::('A,'α::null) Set) = false
by(simp add: defined-def null-fun-def)
lemma invalid-set-valid [simp,code-unfold]:v(invalid::('A,'α::null) Set) = false
by simp
lemma null-set-valid [simp,code-unfold]:v(null::('A,'α::null) Set) = true
apply(simp add: valid-def null-fun-def bot-fun-def bot-Setbase-def null-Setbase-def)
apply(subst Abs-Setbase-inject,simp-all add: null-option-def bot-option-def)
done

```

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

2.9.3. Definition: Strict Equality

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

overloading

$StrictRefEq \equiv StrictRefEq :: [(\mathfrak{A}, ' \alpha :: null) Set, (\mathfrak{A}, ' \alpha :: null) Set] \Rightarrow (\mathfrak{A}) Boolean$

begin

definition $StrictRefEq_{Set}$:

$(x :: (\mathfrak{A}, ' \alpha :: null) Set) \doteq y \equiv \lambda \tau. \text{if } (v \ x) \ \tau = true \ \tau \wedge (v \ y) \ \tau = true \ \tau$
 $\text{then } (x \ \underline{\doteq} \ y) \ \tau$
 $\text{else } invalid \ \tau$

end

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

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

interpretation $StrictRefEq_{Set}$: $profile-bin_{StrongEq^{-v}v} \ \lambda \ x \ y. (x :: (\mathfrak{A}, ' \alpha :: null) Set) \doteq y$
 by $unfold-locales \ (auto \ simp: \ StrictRefEq_{Set})$

2.9.4. Constants: mtSet

definition $mtSet :: (\mathfrak{A}, ' \alpha :: null) Set \ (\langle Set \{ \} \rangle)$

where $Set \{ \} \equiv (\lambda \ \tau. \ Abs-Set_{base} \ \perp \{ \} :: ' \alpha \ set_{\perp})$

lemma $mtSet-defined[simp, code-unfold]: \delta(Set \{ \}) = true$

apply($rule \ ext, \ auto \ simp: \ mtSet-def \ defined-def \ null-Set_{base}-def$
 $bot-Set_{base}-def \ bot-fun-def \ null-fun-def$)

by($simp-all \ add: \ Abs-Set_{base}-inject \ bot-option-def \ null-Set_{base}-def \ null-option-def$)

lemma $mtSet-valid[simp, code-unfold]: \ v(Set \{ \}) = true$

apply($rule \ ext, \ auto \ simp: \ mtSet-def \ valid-def \ null-Set_{base}-def$
 $bot-Set_{base}-def \ bot-fun-def \ null-fun-def$)

by($simp-all \ add: \ Abs-Set_{base}-inject \ bot-option-def \ null-Set_{base}-def \ null-option-def$)

lemma $mtSet-rep-set: \ \top Rep-Set_{base} (Set \{ \} \ \tau) \top = \{ \}$

apply($simp \ add: \ mtSet-def, \ subst \ Abs-Set_{base}-inverse$)

by($simp \ add: \ bot-option-def$) $+$

lemma $[simp, code-unfold]: \ const \ Set \{ \}$

by($simp \ add: \ const-def \ mtSet-def$)

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

2.9.5. Definition: Including

definition $OclIncluding :: [(\mathfrak{A}, ' \alpha :: null) Set, (\mathfrak{A}, ' \alpha) val] \Rightarrow (\mathfrak{A}, ' \alpha) Set$

where $OclIncluding \ x \ y = (\lambda \ \tau. \ \text{if } (\delta \ x) \ \tau = true \ \tau \wedge (v \ y) \ \tau = true \ \tau$
 $\text{then } Abs-Set_{base} \ \perp \ \top Rep-Set_{base} (x \ \tau) \top \cup \{y \ \tau\} \ \perp$
 $\text{else } invalid \ \tau)$

notation $OclIncluding \ (\langle \dashrightarrow including_{Set} '(-) \rangle)$

interpretation $OclIncluding : profile-bin_{d-v} OclIncluding \lambda x y. Abs-Set_{base_{\perp}} \ulcorner Rep-Set_{base} x^{\top} \cup \{y\} \urcorner$
proof –
have $A : None \in \{X. X = bot \vee X = null \vee (\forall x \in \ulcorner X^{\top}. x \neq bot)\}$ **by** (*simp add: bot-option-def*)
have $B : \lrcorner None_{\perp} \in \{X. X = bot \vee X = null \vee (\forall x \in \ulcorner X^{\top}. x \neq bot)\}$
by (*simp add: null-option-def bot-option-def*)
have $C : \bigwedge x y. x \neq \perp \implies x \neq null \implies y \neq \perp \implies$
 $\ulcorner insert y \ulcorner Rep-Set_{base} x^{\top} \urcorner \in \{X. X = bot \vee X = null \vee (\forall x \in \ulcorner X^{\top}. x \neq bot)\}$
by (*auto intro!: Set-inv-lemma[simplified OclValid-def*
defined-def false-def true-def null-fun-def bot-fun-def])
show $profile-bin_{d-v} OclIncluding (\lambda x y. Abs-Set_{base_{\perp}} \ulcorner Rep-Set_{base} x^{\top} \cup \{y\} \urcorner)$
apply *unfold-locales*
apply (*auto simp: OclIncluding-def bot-option-def null-option-def null-Set_{base}-def bot-Set_{base}-def*)
apply (*erule-tac Q=Abs-Set_{base_{\perp}} insert y \ulcorner Rep-Set_{base} x^{\top} \urcorner = Abs-Set_{base} None in contrapos-pp*)
apply (*subst Abs-Set_{base}-inject[OF C A]*)
apply (*simp-all add: null-Set_{base}-def bot-Set_{base}-def bot-option-def*)
apply (*erule-tac Q=Abs-Set_{base_{\perp}} insert y \ulcorner Rep-Set_{base} x^{\top} \urcorner = Abs-Set_{base} \lrcorner None_{\perp} in contrapos-pp*)
apply (*subst Abs-Set_{base}-inject[OF C B]*)
apply (*simp-all add: null-Set_{base}-def bot-Set_{base}-def bot-option-def*)
done

qed

syntax

-OclFinset :: args => ('a, 'a::null) Set (Set{(-)})

syntax-consts

-OclFinset == OclIncluding

translations

Set{x, xs} == CONST OclIncluding (Set{xs}) x

Set{x} == CONST OclIncluding (Set{ }) x

2.9.6. Definition: Excluding

definition $OclExcluding :: [(\alpha, 'a::null) Set, (\alpha, 'a) val] \Rightarrow (\alpha, 'a) Set$

where $OclExcluding x y = (\lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (v y) \tau = true \tau$
then $Abs-Set_{base_{\perp}} \ulcorner Rep-Set_{base} (x \tau)^{\top} - \{y \tau\} \urcorner$
else \perp)

notation $OclExcluding (\ulcorner \dashv \dashv \rangle excluding_{Set} '(-) \urcorner)$

lemma $OclExcluding-inv: (x :: Set('b::\{null\})) \neq \perp \implies x \neq null \implies y \neq \perp \implies$

$\ulcorner Rep-Set_{base} x^{\top} - \{y\} \urcorner \in \{X. X = bot \vee X = null \vee (\forall x \in \ulcorner X^{\top}. x \neq bot)\}$

proof – **fix** $X :: 'a state \times 'a state \Rightarrow Set('b)$ **fix** τ

show $x \neq \perp \implies x \neq null \implies y \neq \perp \implies ?thesis$

when $x = X \tau$

by (*simp add: that Set-inv-lemma[simplified OclValid-def*
defined-def null-fun-def bot-fun-def, of X \tau])

qed *simp-all*

interpretation $OclExcluding : profile-bin_{d-v} OclExcluding \lambda x y. Abs-Set_{base_{\perp}} \ulcorner Rep-Set_{base} x^{\top} - \{y\} \urcorner$

proof –

have $A : None \in \{X. X = bot \vee X = null \vee (\forall x \in \ulcorner X^{\top}. x \neq bot)\}$ **by** (*simp add: bot-option-def*)

have $B : \lrcorner None_{\perp} \in \{X. X = bot \vee X = null \vee (\forall x \in \ulcorner X^{\top}. x \neq bot)\}$

by (*simp add: null-option-def bot-option-def*)

show $profile-bin_{d-v} OclExcluding (\lambda x y. Abs-Set_{base_{\perp}} \ulcorner Rep-Set_{base} (x :: Set('b))^{\top} - \{y\} \urcorner)$

apply *unfold-locales*

apply (*auto simp: OclExcluding-def bot-option-def null-option-def null-Set_{base}-def bot-Set_{base}-def invalid-def*)

```

apply(erule-tac Q=Abs-Setbase⊥⊔Rep-Setbase x⊔ - {y}⊥ = Abs-Setbase None in contrapos-pp)
apply(subst Abs-Setbase-inject[OF OclExcluding-inv A])
apply(simp-all add: null-Setbase-def bot-Setbase-def bot-option-def)
apply(erule-tac Q=Abs-Setbase⊥⊔Rep-Setbase x⊔ - {y}⊥ = Abs-Setbase ⊥None in contrapos-pp)
apply(subst Abs-Setbase-inject[OF OclExcluding-inv B])
apply(simp-all add: null-Setbase-def bot-Setbase-def bot-option-def)
done

```

qed

2.9.7. Definition: Includes

definition *OclIncludes* :: $(\mathfrak{A}, \alpha :: \text{null}) \text{ Set}, (\mathfrak{A}, \alpha) \text{ val}] \Rightarrow \mathfrak{A} \text{ Boolean}$
where $\text{OclIncludes } x \ y = (\lambda \tau. \text{ if } (\delta \ x) \ \tau = \text{true } \tau \wedge (v \ y) \ \tau = \text{true } \tau$
 $\text{ then } \perp(y \ \tau) \in {}^{\top}\text{Rep-Set}_{\text{base}}(x \ \tau)^{\top} \perp$
 $\text{ else } \perp)$

notation *OclIncludes* $(\langle \dashrightarrow \text{includes}_{\text{Set}} \rangle)$

interpretation *OclIncludes* : *profile-bin_{d-v}* *OclIncludes* $\lambda x \ y. \perp y \in {}^{\top}\text{Rep-Set}_{\text{base}} x^{\top} \perp$
by(*unfold-locales*, *auto simp:OclIncludes-def bot-option-def null-option-def invalid-def*)

2.9.8. Definition: Excludes

definition *OclExcludes* :: $(\mathfrak{A}, \alpha :: \text{null}) \text{ Set}, (\mathfrak{A}, \alpha) \text{ val}] \Rightarrow \mathfrak{A} \text{ Boolean}$
where $\text{OclExcludes } x \ y = (\text{not}(\text{OclIncludes } x \ y))$
notation *OclExcludes* $(\langle \dashrightarrow \text{excludes}_{\text{Set}} \rangle)$

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

interpretation *OclExcludes* : *profile-bin_{d-v}* *OclExcludes* $\lambda x \ y. \perp y \notin {}^{\top}\text{Rep-Set}_{\text{base}} x^{\top} \perp$
by(*unfold-locales*, *auto simp:OclExcludes-def OclIncludes-def OclNot-def bot-option-def null-option-def invalid-def*)

2.9.9. Definition: Size

definition *OclSize* :: $(\mathfrak{A}, \alpha :: \text{null}) \text{ Set} \Rightarrow \mathfrak{A} \text{ Integer}$
where $\text{OclSize } x = (\lambda \tau. \text{ if } (\delta \ x) \ \tau = \text{true } \tau \wedge \text{finite}({}^{\top}\text{Rep-Set}_{\text{base}}(x \ \tau)^{\top})$
 $\text{ then } \perp \text{int}(\text{card } {}^{\top}\text{Rep-Set}_{\text{base}}(x \ \tau)^{\top}) \perp$
 $\text{ else } \perp)$

notation *OclSize* $(\langle \dashrightarrow \text{size}_{\text{Set}} \rangle)$

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

2.9.10. Definition: IsEmpty

definition *OclIsEmpty* :: $(\mathfrak{A}, \alpha :: \text{null}) \text{ Set} \Rightarrow \mathfrak{A} \text{ Boolean}$
where $\text{OclIsEmpty } x = ((v \ x \ \text{and not } (\delta \ x)) \ \text{or } ((\text{OclSize } x) \doteq \mathbf{0}))$
notation *OclIsEmpty* $(\langle \dashrightarrow \text{isEmpty}_{\text{Set}} \rangle)$

2.9.11. Definition: NotEmpty

definition *OclNotEmpty* :: $(\mathfrak{A}, \alpha :: \text{null}) \text{ Set} \Rightarrow \mathfrak{A} \text{ Boolean}$
where $\text{OclNotEmpty } x = \text{not}(\text{OclIsEmpty } x)$
notation *OclNotEmpty* $(\langle \dashrightarrow \text{notEmpty}_{\text{Set}} \rangle)$

2.9.12. Definition: Any

definition $OclANY$:: $[(\mathfrak{A}, \alpha :: null) Set] \Rightarrow (\mathfrak{A}, \alpha) val$
where $OclANY$ $x = (\lambda \tau. \text{if } (v \ x) \ \tau = true \ \tau$
 $\text{then if } (\delta \ x \ \text{and } OclNotEmpty \ x) \ \tau = true \ \tau$
 $\text{then SOME } y. y \in {}^{\top}Rep-Set_{base} \ (x \ \tau)^{\top}$
 $\text{else null } \tau$
 $\text{else } \perp)$
notation $OclANY$ $(\langle - \rangle \rightarrow any_{Set} '(\cdot)')$

2.9.13. Definition: Forall

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

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

syntax

- $OclForallSet$:: $[(\mathfrak{A}, \alpha :: null) Set, id, (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean$ $(\langle (-) \rangle \rightarrow forAll_{Set} '(-)')$

syntax-consts

- $OclForallSet == UML-Set.OclForall$

translations

$X \rightarrow forAll_{Set}(x \mid P) == CONST \ UML-Set.OclForall \ X \ (\%x. P)$

2.9.14. Definition: Exists

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

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

syntax

- $OclExistSet$:: $[(\mathfrak{A}, \alpha :: null) Set, id, (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean$ $(\langle (-) \rangle \rightarrow exists_{Set} '(-)')$

syntax-consts

- $OclExistSet == UML-Set.OclExists$

translations

$X \rightarrow exists_{Set}(x \mid P) == CONST \ UML-Set.OclExists \ X \ (\%x. P)$

2.9.15. Definition: Iterate

definition $OclIterate$:: $[(\mathfrak{A}, \alpha :: null) Set, (\mathfrak{A}, \beta :: null) val,$
 $(\mathfrak{A}, \alpha) val \Rightarrow (\mathfrak{A}, \beta) val \Rightarrow (\mathfrak{A}, \beta) val] \Rightarrow (\mathfrak{A}, \beta) val$
where $OclIterate$ $S \ A \ F = (\lambda \tau. \text{if } (\delta \ S) \ \tau = true \ \tau \wedge (v \ A) \ \tau = true \ \tau \wedge finite^{\top}Rep-Set_{base} \ (S \ \tau)^{\top}$
 $\text{then } (Finite-Set.fold \ (F) \ (A) \ ((\lambda a \ \tau. a) \ '({}^{\top}Rep-Set_{base} \ (S \ \tau)^{\top})) \ \tau$
 $\text{else } \perp)$

syntax

- $OclIterateSet$:: $[(\mathfrak{A}, \alpha :: null) Set, idt, idt, \alpha, \beta] \Rightarrow (\mathfrak{A}, \gamma) val$
 $(\langle - \rangle \rightarrow iterates_{Set} '(-; == | -)')$

syntax-consts

- $OclIterateSet == OclIterate$

translations

$X \rightarrow iterates_{Set}(a; x = A \mid P) == CONST \ OclIterate \ X \ A \ (\%a. (\%x. P))$

2.9.16. Definition: Select

definition $OclSelect :: [(\mathfrak{A}, \alpha :: null) Set, (\mathfrak{A}, \alpha) val \Rightarrow (\mathfrak{A}) Boolean] \Rightarrow (\mathfrak{A}, \alpha) Set$
where $OclSelect S P = (\lambda \tau. \text{if } (\delta S) \tau = true \ \tau$
 then if $(\exists x \in {}^\top Rep-Set_{base} (S \ \tau)^\top. P(\lambda \cdot. x) \ \tau = invalid \ \tau$
 then $invalid \ \tau$
 else $Abs-Set_{base} \perp \{x \in {}^\top Rep-Set_{base} (S \ \tau)^\top. P(\lambda \cdot. x) \ \tau \neq false \ \tau\} \perp$
 else $invalid \ \tau)$

syntax

$-OclSelectSet :: [(\mathfrak{A}, \alpha :: null) Set, id, (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean \quad (\langle (-) \rightarrow select_{Set} '(-)' \rangle)$

syntax-consts

$-OclSelectSet == OclSelect$

translations

$X \rightarrow select_{Set}(x \mid P) == CONST \ OclSelect \ X \ (\% \ x. \ P)$

2.9.17. Definition: Reject

definition $OclReject :: [(\mathfrak{A}, \alpha :: null) Set, (\mathfrak{A}, \alpha) val \Rightarrow (\mathfrak{A}) Boolean] \Rightarrow (\mathfrak{A}, \alpha :: null) Set$
where $OclReject S P = OclSelect S (not \ o \ P)$

syntax

$-OclRejectSet :: [(\mathfrak{A}, \alpha :: null) Set, id, (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean \quad (\langle (-) \rightarrow reject_{Set} '(-)' \rangle)$

syntax-consts

$-OclRejectSet == OclReject$

translations

$X \rightarrow reject_{Set}(x \mid P) == CONST \ OclReject \ X \ (\% \ x. \ P)$

2.9.18. Definition: IncludesAll

definition $OclIncludesAll :: [(\mathfrak{A}, \alpha :: null) Set, (\mathfrak{A}, \alpha) Set] \Rightarrow \mathfrak{A} Boolean$
where $OclIncludesAll \ x \ y = (\lambda \tau. \text{if } (\delta x) \tau = true \ \tau \wedge (\delta y) \tau = true \ \tau$
 then $\perp \perp Rep-Set_{base} (y \ \tau)^\top \subseteq {}^\top Rep-Set_{base} (x \ \tau)^\top \perp$
 else \perp)

notation $OclIncludesAll \ (\langle -- \rangle includesAll_{Set} '(-)' \rangle)$

interpretation $OclIncludesAll : profile-bin_{d-d} \ OclIncludesAll \ \lambda x \ y. \perp \perp Rep-Set_{base} \ y^\top \subseteq {}^\top Rep-Set_{base} \ x^\top \perp$
by $(unfold-locales, \ auto \ simp: OclIncludesAll-def \ bot-option-def \ null-option-def \ invalid-def)$

2.9.19. Definition: ExcludesAll

definition $OclExcludesAll :: [(\mathfrak{A}, \alpha :: null) Set, (\mathfrak{A}, \alpha) Set] \Rightarrow \mathfrak{A} Boolean$
where $OclExcludesAll \ x \ y = (\lambda \tau. \text{if } (\delta x) \tau = true \ \tau \wedge (\delta y) \tau = true \ \tau$
 then $\perp \perp Rep-Set_{base} (y \ \tau)^\top \cap {}^\top Rep-Set_{base} (x \ \tau)^\top = \{\} \perp$
 else \perp)

notation $OclExcludesAll \ (\langle -- \rangle excludesAll_{Set} '(-)' \rangle)$

interpretation $OclExcludesAll : profile-bin_{d-d} \ OclExcludesAll \ \lambda x \ y. \perp \perp Rep-Set_{base} \ y^\top \cap {}^\top Rep-Set_{base} \ x^\top = \{\} \perp$
by $(unfold-locales, \ auto \ simp: OclExcludesAll-def \ bot-option-def \ null-option-def \ invalid-def)$

2.9.20. Definition: Union

definition $OclUnion :: [(\mathfrak{A}, \alpha :: null) Set, (\mathfrak{A}, \alpha) Set] \Rightarrow (\mathfrak{A}, \alpha) Set$
where $OclUnion \ x \ y = (\lambda \tau. \text{if } (\delta x) \tau = true \ \tau \wedge (\delta y) \tau = true \ \tau$
 then $Abs-Set_{base} \perp \perp \perp Rep-Set_{base} (y \ \tau)^\top \cup {}^\top Rep-Set_{base} (x \ \tau)^\top \perp$
 else \perp)

notation $OclUnion \ (\langle -- \rangle union_{Set} '(-)' \rangle)$

lemma $OclUnion-inv: (x :: Set('b :: \{null\})) \neq \perp \implies x \neq null \implies y \neq \perp \implies y \neq null \implies$

$\perp \sqcup \ulcorner \text{Rep-Set}_{base} y \urcorner \cup \ulcorner \text{Rep-Set}_{base} x \urcorner \sqcup \in \{X. X = bot \vee X = null \vee (\forall x \in \ulcorner X \urcorner. x \neq bot)\}$
proof – **fix** $X Y :: 'a \text{ state} \times 'a \text{ state} \Rightarrow \text{Set}(b)$ **fix** τ
show $x \neq \perp \Longrightarrow x \neq null \Longrightarrow y \neq \perp \Longrightarrow y \neq null \Longrightarrow ?thesis$
when $x = X \tau \ y = Y \tau$
by(*auto simp: that,*
insert
Set-inv-lemma[simplified OclValid-def
defined-def null-fun-def bot-fun-def, of Y \tau]
Set-inv-lemma[simplified OclValid-def
defined-def null-fun-def bot-fun-def, of X \tau],
auto)
qed simp-all

interpretation $OclUnion : \text{profile-bin}_{d-d} OclUnion \lambda x y. Abs\text{-Set}_{base} \ulcorner \text{Rep-Set}_{base} y \urcorner \cup \ulcorner \text{Rep-Set}_{base} x \urcorner \sqcup$
proof –
have $A : None \in \{X. X = bot \vee X = null \vee (\forall x \in \ulcorner X \urcorner. x \neq bot)\}$ **by**(*simp add: bot-option-def*)
have $B : \ulcorner None \urcorner \in \{X. X = bot \vee X = null \vee (\forall x \in \ulcorner X \urcorner. x \neq bot)\}$
by(*simp add: null-option-def bot-option-def*)
show $\text{profile-bin}_{d-d} OclUnion (\lambda x y. Abs\text{-Set}_{base} \ulcorner \text{Rep-Set}_{base} y \urcorner \cup \ulcorner \text{Rep-Set}_{base} x \urcorner \sqcup)$
apply *unfold-locales*
apply(*auto simp: OclUnion-def bot-option-def null-option-def null-Set_{base}-def bot-Set_{base}-def invalid-def*)
apply(*erule-tac Q=Abs-Set_{base} \ulcorner \text{Rep-Set}_{base} y \urcorner \cup \ulcorner \text{Rep-Set}_{base} x \urcorner \sqcup = Abs-Set_{base} None* **in** *contra-pos-pp*)
apply(*subst Abs-Set_{base}-inject[OF OclUnion-inv A]*)
apply(*simp-all add: null-Set_{base}-def bot-Set_{base}-def bot-option-def*)
apply(*erule-tac Q=Abs-Set_{base} \ulcorner \text{Rep-Set}_{base} y \urcorner \cup \ulcorner \text{Rep-Set}_{base} x \urcorner \sqcup = Abs-Set_{base} \ulcorner None \urcorner* **in** *contra-pos-pp*)
apply(*subst Abs-Set_{base}-inject[OF OclUnion-inv B]*)
apply(*simp-all add: null-Set_{base}-def bot-Set_{base}-def bot-option-def*)
done
qed

2.9.21. Definition: Intersection

definition $OclIntersection :: [(\mathfrak{A}, \alpha :: null) \text{ Set}, (\mathfrak{A}, \alpha) \text{ Set}] \Rightarrow (\mathfrak{A}, \alpha) \text{ Set}$
where $OclIntersection \ x \ y = (\lambda \tau. \text{if } (\delta \ x) \ \tau = \text{true} \ \tau \wedge (\delta \ y) \ \tau = \text{true} \ \tau$
 $\text{then } Abs\text{-Set}_{base} \ulcorner \text{Rep-Set}_{base} (y \ \tau) \urcorner$
 $\cap \ulcorner \text{Rep-Set}_{base} (x \ \tau) \urcorner \sqcup$
 $\text{else } \perp \)$

notation $OclIntersection(\langle - \rangle \text{intersection}_{Set} \langle - \rangle)$

lemma $OclIntersection\text{-inv}: (x :: \text{Set}(b :: \{null\})) \neq \perp \Longrightarrow x \neq null \Longrightarrow y \neq \perp \Longrightarrow y \neq null \Longrightarrow$
 $\ulcorner \text{Rep-Set}_{base} y \urcorner \cap \ulcorner \text{Rep-Set}_{base} x \urcorner \sqcup \in \{X. X = bot \vee X = null \vee (\forall x \in \ulcorner X \urcorner. x \neq bot)\}$
proof – **fix** $X Y :: 'a \text{ state} \times 'a \text{ state} \Rightarrow \text{Set}(b)$ **fix** τ
show $x \neq \perp \Longrightarrow x \neq null \Longrightarrow y \neq \perp \Longrightarrow y \neq null \Longrightarrow ?thesis$
when $x = X \tau \ y = Y \tau$
by(*auto simp: that,*
insert
Set-inv-lemma[simplified OclValid-def
defined-def null-fun-def bot-fun-def, of Y \tau]
Set-inv-lemma[simplified OclValid-def
defined-def null-fun-def bot-fun-def, of X \tau],
auto)
qed simp-all

interpretation $OclIntersection : \text{profile-bin}_{d-d} OclIntersection \lambda x y. Abs\text{-Set}_{base} \ulcorner \text{Rep-Set}_{base} y \urcorner \cap \ulcorner \text{Rep-Set}_{base} x \urcorner \sqcup$

proof –
have $A : \text{None} \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in {}^{\top}X^{\top}. x \neq \text{bot})\}$ **by** (*simp add: bot-option-def*)
have $B : \lfloor \text{None} \rfloor \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in {}^{\top}X^{\top}. x \neq \text{bot})\}$
by (*simp add: null-option-def bot-option-def*)
show *profile-bin_{d-d} OclIntersection* $(\lambda x y. \text{Abs-Set}_{\text{base}} \lfloor \text{Rep-Set}_{\text{base}} y^{\top} \cap {}^{\top} \text{Rep-Set}_{\text{base}} x^{\top} \rfloor)$
apply *unfold-locales*
apply (*auto simp: OclIntersection-def bot-option-def null-option-def null-Set_{base}-def bot-Set_{base}-def invalid-def*)
apply (*erule-tac Q=Abs-Set_{base} \lfloor \text{Rep-Set}_{\text{base}} y^{\top} \cap {}^{\top} \text{Rep-Set}_{\text{base}} x^{\top} \rfloor = Abs-Set_{\text{base}} \text{None}* **in** *contra-pos-pp*)
apply (*subst Abs-Set_{base}-inject[OF OclIntersection-inv A]*)
apply (*simp-all add: null-Set_{base}-def bot-Set_{base}-def bot-option-def*)
apply (*erule-tac Q=Abs-Set_{base} \lfloor \text{Rep-Set}_{\text{base}} y^{\top} \cap {}^{\top} \text{Rep-Set}_{\text{base}} x^{\top} \rfloor = Abs-Set_{\text{base}} \lfloor \text{None} \rfloor* **in** *contra-pos-pp*)
apply (*subst Abs-Set_{base}-inject[OF OclIntersection-inv B]*)
apply (*simp-all add: null-Set_{base}-def bot-Set_{base}-def bot-option-def*)
done
qed

2.9.22. Definition (future operators)

consts
OclCount $:: [(\mathfrak{A}, \alpha :: \text{null}) \text{Set}, (\mathfrak{A}, \alpha) \text{Set}] \Rightarrow \mathfrak{A} \text{ Integer}$
OclSum $:: (\mathfrak{A}, \alpha :: \text{null}) \text{Set} \Rightarrow \mathfrak{A} \text{ Integer}$

notation *OclCount* $(\langle - \rangle \text{count}_{\text{Set}} \langle - \rangle)$
notation *OclSum* $(\langle - \rangle \text{sum}_{\text{Set}} \langle - \rangle)$

2.9.23. Logical Properties

OclIncluding

lemma *OclIncluding-valid-args-valid*:
 $(\tau \models v(X \rightarrow \text{including}_{\text{Set}}(x))) = ((\tau \models (\delta X)) \wedge (\tau \models (v x)))$
by (*metis (opaque-lifting, no-types) OclIncluding.def-valid-then-def OclIncluding.defined-args-valid*)

lemma *OclIncluding-valid-args-valid''[simp,code-unfold]*:
 $v(X \rightarrow \text{including}_{\text{Set}}(x)) = ((\delta X) \text{ and } (v x))$
by (*simp add: OclIncluding.def-valid-then-def*)

etc. etc.

OclExcluding

lemma *OclExcluding-valid-args-valid*:
 $(\tau \models v(X \rightarrow \text{excluding}_{\text{Set}}(x))) = ((\tau \models (\delta X)) \wedge (\tau \models (v x)))$
by (*metis OclExcluding.def-valid-then-def OclExcluding.defined-args-valid*)

lemma *OclExcluding-valid-args-valid''[simp,code-unfold]*:
 $v(X \rightarrow \text{excluding}_{\text{Set}}(x)) = ((\delta X) \text{ and } (v x))$
by (*simp add: OclExcluding.def-valid-then-def*)

OclIncludes

lemma *OclIncludes-valid-args-valid*:
 $(\tau \models v(X \rightarrow \text{includes}_{\text{Set}}(x))) = ((\tau \models (\delta X)) \wedge (\tau \models (v x)))$
by (*simp add: OclIncludes.def-valid-then-def foundation10'*)

lemma *OclIncludes-valid-args-valid''[simp,code-unfold]*:
 $v(X \rightarrow \text{includes}_{\text{Set}}(x)) = ((\delta X) \text{ and } (v x))$

by (simp add: OclIncludes.def-valid-then-def)

OclExcludes

lemma OclExcludes-valid-args-valid:

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

by (simp add: OclExcludes.def-valid-then-def foundation10')

lemma OclExcludes-valid-args-valid''[simp,code-unfold]:

$v(X \rightarrow \text{excludes}_{\text{Set}}(x)) = ((\delta X) \text{ and } (v x))$

by (simp add: OclExcludes.def-valid-then-def)

OclSize

lemma OclSize-defined-args-valid: $\tau \models \delta (X \rightarrow \text{size}_{\text{Set}}()) \implies \tau \models \delta X$

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

lemma OclSize-infinite:

assumes non-finite: $\tau \models \text{not}(\delta(S \rightarrow \text{size}_{\text{Set}}()))$

shows $(\tau \models \text{not}(\delta(S))) \vee \neg \text{finite } \ulcorner \text{Rep-Set}_{\text{base}}(S \ \tau) \urcorner$

apply(insert non-finite, simp)

apply(rule impI)

apply(simp add: OclSize-def OclValid-def defined-def)

apply(case-tac finite $\ulcorner \text{Rep-Set}_{\text{base}}(S \ \tau) \urcorner$,

simp-all add:null-fun-def null-option-def bot-fun-def bot-option-def)

done

lemma $\tau \models \delta X \implies \neg \text{finite } \ulcorner \text{Rep-Set}_{\text{base}}(X \ \tau) \urcorner \implies \neg \tau \models \delta (X \rightarrow \text{size}_{\text{Set}}())$

by(simp add: OclSize-def OclValid-def defined-def bot-fun-def false-def true-def)

lemma size-defined:

assumes X-finite: $\bigwedge \tau. \text{finite } \ulcorner \text{Rep-Set}_{\text{base}}(X \ \tau) \urcorner$

shows $\delta (X \rightarrow \text{size}_{\text{Set}}()) = \delta X$

apply(rule ext, simp add: cp-defined[of $X \rightarrow \text{size}_{\text{Set}}()$] OclSize-def)

apply(simp add: defined-def bot-option-def bot-fun-def null-option-def null-fun-def X-finite)

done

lemma size-defined':

assumes X-finite: $\text{finite } \ulcorner \text{Rep-Set}_{\text{base}}(X \ \tau) \urcorner$

shows $(\tau \models \delta (X \rightarrow \text{size}_{\text{Set}}())) = (\tau \models \delta X)$

apply(simp add: cp-defined[of $X \rightarrow \text{size}_{\text{Set}}()$] OclSize-def OclValid-def)

apply(simp add: defined-def bot-option-def bot-fun-def null-option-def null-fun-def X-finite)

done

OclIsEmpty

lemma OclIsEmpty-defined-args-valid: $\tau \models \delta (X \rightarrow \text{isEmpty}_{\text{Set}}()) \implies \tau \models v X$

apply(auto simp: OclIsEmpty-def OclValid-def defined-def valid-def false-def true-def
bot-fun-def null-fun-def OclAnd-def OclOr-def OclNot-def
split: if-split-asm)

apply(case-tac $(X \rightarrow \text{size}_{\text{Set}}()) \doteq \mathbf{0}$ τ , simp add: bot-option-def, simp, rename-tac x)

apply(case-tac x, simp add: null-option-def bot-option-def, simp)

apply(simp add: OclSize-def StrictRefEqInteger valid-def)

by (metis (opaque-lifting, no-types)

bot-fun-def OclValid-def defined-def foundation2 invalid-def)

lemma $\tau \models \delta (\text{null} \rightarrow \text{isEmpty}_{\text{Set}}())$

by(auto simp: OclIsEmpty-def OclValid-def defined-def valid-def false-def true-def)

*bot-fun-def null-fun-def OclAnd-def OclOr-def OclNot-def null-is-valid
split: if-split-asm)*

lemma *OclIsEmptyInfinite*: $\tau \models \delta X \implies \neg \text{finite } \ulcorner \text{Rep-Set}_{\text{base}}(X \ \tau) \urcorner \implies \neg \tau \models \delta (X \rightarrow \text{isEmpty}_{\text{Set}}())$
apply(*auto simp: OclIsEmpty-def OclValid-def defined-def valid-def false-def true-def
bot-fun-def null-fun-def OclAnd-def OclOr-def OclNot-def
split: if-split-asm*)
apply(*case-tac (X → size_{Set}() ≐ 0) τ, simp add: bot-option-def, simp, rename-tac x*)
apply(*case-tac x, simp add: null-option-def bot-option-def, simp*)
by(*simp add: OclSize-def StrictRefEqInteger valid-def bot-fun-def false-def true-def invalid-def*)

OclNotEmpty

lemma *OclNotEmpty-defined-args-valid*: $\tau \models \delta (X \rightarrow \text{notEmpty}_{\text{Set}}()) \implies \tau \models v X$
by (*metis (opaque-lifting, no-types) OclNotEmpty-def OclNot-defargs OclNot-not foundation6 foundation9
OclIsEmpty-defined-args-valid*)

lemma $\tau \models \delta (\text{null} \rightarrow \text{notEmpty}_{\text{Set}}())$
by (*metis (opaque-lifting, no-types) OclNotEmpty-def OclAnd-false1 OclAnd-idem OclIsEmpty-def
OclNot3 OclNot4 OclOr-def defined2 defined4 transform1 valid2*)

lemma *OclNotEmpty-infinite*: $\tau \models \delta X \implies \neg \text{finite } \ulcorner \text{Rep-Set}_{\text{base}}(X \ \tau) \urcorner \implies \neg \tau \models \delta (X \rightarrow \text{notEmpty}_{\text{Set}}())$
apply(*simp add: OclNotEmpty-def*)
apply(*drule OclIsEmpty-infinite, simp*)
by (*metis OclNot-defargs OclNot-not foundation6 foundation9*)

lemma *OclNotEmpty-has-elt* : $\tau \models \delta X \implies$
 $\tau \models X \rightarrow \text{notEmpty}_{\text{Set}}() \implies$
 $\exists e. e \in \ulcorner \text{Rep-Set}_{\text{base}}(X \ \tau) \urcorner$
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: if-split-asm*)
prefer 2
apply(*simp add: invalid-def bot-option-def true-def*)
apply(*simp add: OclSize-def valid-def split: if-split-asm,*
simp-all add: false-def true-def bot-option-def bot-fun-def OclInt0-def)
by (*metis equalsOI*)

OclANY

lemma *OclANY-defined-args-valid*: $\tau \models \delta (X \rightarrow \text{anySet}()) \implies \tau \models \delta X$
by(*auto simp: OclANY-def OclValid-def true-def valid-def false-def StrongEq-def
defined-def invalid-def bot-fun-def null-fun-def OclAnd-def
split: bool.split-asm HOL.if-split-asm option.split*)

lemma $\tau \models \delta X \implies \tau \models X \rightarrow \text{isEmpty}_{\text{Set}}() \implies \neg \tau \models \delta (X \rightarrow \text{anySet}())$
apply(*simp add: OclANY-def OclValid-def*)
apply(*subst cp-defined, subst cp-OclAnd, simp add: OclNotEmpty-def, subst (1 2) cp-OclNot,*
*simp add: cp-OclNot[symmetric] cp-OclAnd[symmetric] cp-defined[symmetric],
simp add: false-def true-def*)
by(*drule foundation20[simplified OclValid-def true-def], simp*)

lemma *OclANY-valid-args-valid*:

$(\tau \models v(X \rightarrow \text{anySet}())) = (\tau \models v X)$

proof –

have A: $(\tau \models v(X \rightarrow \text{anySet}())) \implies ((\tau \models (v X)))$

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

```

      split: bool.split-asm HOL.if-split-asm option.split)
have B: ( $\tau \models (v X)$ )  $\implies$  ( $\tau \models v(X \rightarrow any_{Set}())$ )
  apply(auto simp: OclANY-def OclValid-def true-def false-def StrongEq-def
    defined-def invalid-def valid-def bot-fun-def null-fun-def
    bot-option-def null-option-def null-is-valid
    OclAnd-def
    split: bool.split-asm HOL.if-split-asm option.split)
  apply(frule Set-inv-lemma[OF foundation16[THEN iffD2], OF conjI], simp)
  apply(subgoal-tac ( $\delta X$ )  $\tau = true \tau$ )
  prefer 2
  apply(metis (opaque-lifting, no-types) OclValid-def foundation16)
  apply(simp add: true-def,
    drule OclNotEmpty-has-elt[simplified OclValid-def true-def], simp)
by(erule exE,
  insert someI2[where  $Q = \lambda x. x \neq \perp$  and  $P = \lambda y. y \in {}^\top Rep-Set_{base} (X \tau) {}^\top$ ],
  simp)
show ?thesis by(auto dest:A intro:B)
qed

```

lemma *OclANY-valid-args-valid''*[simp,code-unfold]:
 $v(X \rightarrow any_{Set}()) = (v X)$
by(auto intro!: OclANY-valid-args-valid transform2-rev)

2.9.24. Execution Laws with Invalid or Null or Infinite Set as Argument

OclIncluding

OclExcluding

OclIncludes

OclExcludes

OclSize

lemma *OclSize-invalid*[simp,code-unfold]:($invalid \rightarrow size_{Set}()$) = *invalid*
by(simp add: bot-fun-def OclSize-def invalid-def defined-def valid-def false-def true-def)

lemma *OclSize-null*[simp,code-unfold]:($null \rightarrow size_{Set}()$) = *invalid*
by(rule ext,
 simp add: bot-fun-def null-fun-def null-is-valid OclSize-def
 invalid-def defined-def valid-def false-def true-def)

OclIsEmpty

lemma *OclIsEmpty-invalid*[simp,code-unfold]:($invalid \rightarrow isEmpty_{Set}()$) = *invalid*
by(simp add: OclIsEmpty-def)

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

OclNotEmpty

lemma *OclNotEmpty-invalid*[simp,code-unfold]:($invalid \rightarrow notEmpty_{Set}()$) = *invalid*
by(simp add: OclNotEmpty-def)

lemma *OclNotEmpty-null*[simp,code-unfold]:($null \rightarrow notEmpty_{Set}()$) = *false*
by(simp add: OclNotEmpty-def)

OclANY

lemma *OclANY-invalid*[simp,code-unfold]:($invalid \rightarrow any_{Set}()$) = *invalid*

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

lemma *OclANY-null[*simp,code-unfold*]:null \rightarrow any_{Set}() = null*

by(*simp add: OclANY-def false-def true-def*)

OclForall

lemma *OclForall-invalid[*simp,code-unfold*]:invalid \rightarrow forall_{Set}(a | P a) = invalid*

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

lemma *OclForall-null[*simp,code-unfold*]:null \rightarrow forall_{Set}(a | P a) = invalid*

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

OclExists

lemma *OclExists-invalid[*simp,code-unfold*]:invalid \rightarrow exists_{Set}(a | P a) = invalid*

by(*simp add: OclExists-def*)

lemma *OclExists-null[*simp,code-unfold*]:null \rightarrow exists_{Set}(a | P a) = invalid*

by(*simp add: OclExists-def*)

OclIterate

lemma *OclIterate-invalid[*simp,code-unfold*]:invalid \rightarrow iterate_{Set}(a; x = A | P a x) = invalid*

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

lemma *OclIterate-null[*simp,code-unfold*]:null \rightarrow iterate_{Set}(a; x = A | P a x) = invalid*

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

lemma *OclIterate-invalid-args[*simp,code-unfold*]:S \rightarrow iterate_{Set}(a; x = invalid | P a x) = invalid*

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

An open question is this ...

lemma *S \rightarrow iterate_{Set}(a; x = null | P a x) = invalid*

oops

lemma *OclIterate-infinite:*

assumes *non-finite: $\tau \models \text{not}(\delta(S \rightarrow \text{size}_{\text{Set}}()))$*

shows *(OclIterate S A F) $\tau = \text{invalid } \tau$*

apply *(insert non-finite [THEN OclSize-infinite])*

apply *(subst (asm) foundation9, simp)*

by(*metis OclIterate-def OclValid-def invalid-def*)

OclSelect

lemma *OclSelect-invalid[*simp,code-unfold*]:invalid \rightarrow select_{Set}(a | P a) = invalid*

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

lemma *OclSelect-null[*simp,code-unfold*]:null \rightarrow select_{Set}(a | P a) = invalid*

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

OclReject

lemma *OclReject-invalid[*simp,code-unfold*]:invalid \rightarrow reject_{Set}(a | P a) = invalid*

by(*simp add: OclReject-def*)

lemma *OclReject-null[*simp,code-unfold*]:null \rightarrow reject_{Set}(a | P a) = invalid*

by(*simp add: OclReject-def*)

Context Passing

lemma *cp-OclIncludes1*:

$(X \rightarrow \text{includes}_{Set}(x)) \tau = (X \rightarrow \text{includes}_{Set}(\lambda \cdot x \tau)) \tau$

by(*auto simp*: *OclIncludes-def StrongEq-def invalid-def*
cp-defined[symmetric] cp-valid[symmetric])

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

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

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

apply(*simp only*: *OclIsEmpty-def*)

apply(*subst* (2) *cp-OclOr*,

subst cp-OclAnd,

subst cp-OclNot,

subst StrictRefEqInteger.cp0)

by(*simp add*: *cp-defined[symmetric] cp-valid[symmetric] StrictRefEqInteger.cp0[symmetric]*
cp-OclSize[symmetric] cp-OclNot[symmetric] cp-OclAnd[symmetric] cp-OclOr[symmetric])

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

apply(*simp only*: *OclNotEmpty-def*)

apply(*subst* (2) *cp-OclNot*)

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

lemma *cp-OclANY*: $X \rightarrow \text{any}_{Set}() \tau = ((\lambda \cdot X \tau) \rightarrow \text{any}_{Set}()) \tau$

apply(*simp only*: *OclANY-def*)

apply(*subst* (2) *cp-OclAnd*)

by(*simp only*: *cp-OclAnd[symmetric] cp-defined[symmetric] cp-valid[symmetric]*
cp-OclNotEmpty[symmetric])

lemma *cp-OclForall*:

$(S \rightarrow \text{forAll}_{Set}(x \mid P x)) \tau = ((\lambda \cdot S \tau) \rightarrow \text{forAll}_{Set}(x \mid P (\lambda \cdot x \tau))) \tau$

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

lemma *cp-OclForall1* [*simp,intro!*]:

$cp S \implies cp (\lambda X. ((S X) \rightarrow \text{forAll}_{Set}(x \mid P x)))$

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

apply(*erule exE, rule exI, intro allI*)

apply(*erule-tac x=X in allE*)

by(*subst cp-OclForall, simp*)

lemma

$cp (\lambda X St x. P (\lambda \tau. x) X St) \implies cp S \implies cp (\lambda X. (S X) \rightarrow \text{forAll}_{Set}(x \mid P x X))$

apply(*simp only*: *cp-def*)

oops

lemma

$cp S \implies$

$(\bigwedge x. cp(P x)) \implies$

$cp(\lambda X. ((S X) \rightarrow \text{forAll}_{Set}(x \mid P x X)))$

oops

lemma *cp-OclExists*:

$(S \rightarrow \text{exists}_{Set}(x \mid P x)) \tau = ((\lambda \cdot S \tau) \rightarrow \text{exists}_{Set}(x \mid P (\lambda \cdot x \tau))) \tau$

by(*simp add*: *OclExists-def OclNot-def, subst cp-OclForall, simp*)

lemma *cp-OclExists1* [*simp,intro!*]:
cp S \implies *cp* ($\lambda X. ((S X) \rightarrow \text{exists}_{Set}(x \mid P x))$)
apply (*simp add: cp-def*)
apply (*erule exE, rule exI, intro allI*)
apply (*erule-tac x=X in allE*)
by (*subst cp-OclExists, simp*)

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

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

lemma *cp-OclReject*: $(X \rightarrow \text{reject}_{Set}(a \mid P a)) \tau = ((\lambda -. X \tau) \rightarrow \text{reject}_{Set}(a \mid P a)) \tau$
by (*simp add: OclReject-def, subst cp-OclSelect, simp*)

lemmas *cp-intro''_{Set}* [*intro!,simp,code-unfold*] =
cp-OclSize [*THEN allI[THEN allI[THEN cpI1], of OclSize]*]
cp-OclIsEmpty [*THEN allI[THEN allI[THEN cpI1], of OclIsEmpty]*]
cp-OclNotEmpty [*THEN allI[THEN allI[THEN cpI1], of OclNotEmpty]*]
cp-OclANY [*THEN allI[THEN allI[THEN cpI1], of OclANY]*]

Const

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

qed

2.9.25. General Algebraic Execution Rules

Execution Rules on Including

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

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

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

lemma *OclIncluding-includes0*:
assumes $\tau \models X \rightarrow \text{includes}_{\text{Set}}(x)$
shows $X \rightarrow \text{including}_{\text{Set}}(x) \tau = X \tau$
proof –
have $\text{includes-def} : \tau \models X \rightarrow \text{includes}_{\text{Set}}(x) \implies \tau \models \delta X$
by ($\text{metis } \text{bot-fun-def } \text{OclIncludes-def } \text{OclValid-def } \text{defined3 } \text{foundation16}$)

have $\text{includes-val} : \tau \models X \rightarrow \text{includes}_{\text{Set}}(x) \implies \tau \models v x$
using $\text{foundation5 } \text{foundation6}$ **by** fastforce

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

lemma *OclIncluding-includes*:
assumes $\tau \models X \rightarrow \text{includes}_{\text{Set}}(x)$
shows $\tau \models X \rightarrow \text{including}_{\text{Set}}(x) \triangleq X$
by($\text{simp add: } \text{StrongEq-def } \text{OclValid-def } \text{true-def } \text{OclIncluding-includes0}[OF \text{assms}]$)

lemma *OclIncluding-commute0* :
assumes $S\text{-def} : \tau \models \delta S$

and $i\text{-val} : \tau \models v \ i$
and $j\text{-val} : \tau \models v \ j$
shows $\tau \models ((S \ :: \ (\mathcal{A}, \ 'a::\text{null}) \ \text{Set}) \rightarrow \text{including}_{\text{Set}}(i) \rightarrow \text{including}_{\text{Set}}(j)) \ \triangleq$
 $(S \rightarrow \text{including}_{\text{Set}}(j) \rightarrow \text{including}_{\text{Set}}(i))$
proof –
have $A : \llbracket \text{insert} \ (i \ \tau) \ \top \text{Rep-Set}_{\text{base}} \ (S \ \tau) \rrbracket_{\perp} \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \top X^{\top}. x \neq \text{bot})\}$
by(*insert S-def i-val, frule Set-inv-lemma, simp add: foundation18 invalid-def*)
have $B : \llbracket \text{insert} \ (j \ \tau) \ \top \text{Rep-Set}_{\text{base}} \ (S \ \tau) \rrbracket_{\perp} \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \top X^{\top}. x \neq \text{bot})\}$
by(*insert S-def j-val, frule Set-inv-lemma, simp add: foundation18 invalid-def*)

have $G1 : \text{Abs-Set}_{\text{base}} \ \llbracket \text{insert} \ (i \ \tau) \ \top \text{Rep-Set}_{\text{base}} \ (S \ \tau) \rrbracket_{\perp} \neq \text{Abs-Set}_{\text{base}} \ \text{None}$
by(*insert A, simp add: Abs-Set_{base}-inject bot-option-def null-option-def*)
have $G2 : \text{Abs-Set}_{\text{base}} \ \llbracket \text{insert} \ (i \ \tau) \ \top \text{Rep-Set}_{\text{base}} \ (S \ \tau) \rrbracket_{\perp} \neq \text{Abs-Set}_{\text{base}} \ \llbracket \text{None} \rrbracket_{\perp}$
by(*insert A, simp add: Abs-Set_{base}-inject bot-option-def null-option-def*)
have $G3 : \text{Abs-Set}_{\text{base}} \ \llbracket \text{insert} \ (j \ \tau) \ \top \text{Rep-Set}_{\text{base}} \ (S \ \tau) \rrbracket_{\perp} \neq \text{Abs-Set}_{\text{base}} \ \text{None}$
by(*insert B, simp add: Abs-Set_{base}-inject bot-option-def null-option-def*)
have $G4 : \text{Abs-Set}_{\text{base}} \ \llbracket \text{insert} \ (j \ \tau) \ \top \text{Rep-Set}_{\text{base}} \ (S \ \tau) \rrbracket_{\perp} \neq \text{Abs-Set}_{\text{base}} \ \llbracket \text{None} \rrbracket_{\perp}$
by(*insert B, simp add: Abs-Set_{base}-inject bot-option-def null-option-def*)

have $*$: $(\delta \ (\lambda \cdot. \text{Abs-Set}_{\text{base}} \ \llbracket \text{insert} \ (i \ \tau) \ \top \text{Rep-Set}_{\text{base}} \ (S \ \tau) \rrbracket_{\perp})) \ \tau = \llbracket \text{True} \rrbracket_{\perp}$
by(*auto simp: OclValid-def false-def defined-def null-fun-def true-def*
bot-fun-def bot-Set_{base}-def null-Set_{base}-def S-def i-val G1 G2)

have $**$: $(\delta \ (\lambda \cdot. \text{Abs-Set}_{\text{base}} \ \llbracket \text{insert} \ (j \ \tau) \ \top \text{Rep-Set}_{\text{base}} \ (S \ \tau) \rrbracket_{\perp})) \ \tau = \llbracket \text{True} \rrbracket_{\perp}$
by(*auto simp: OclValid-def false-def defined-def null-fun-def true-def*
bot-fun-def bot-Set_{base}-def null-Set_{base}-def S-def i-val G3 G4)

have $***$: $\text{Abs-Set}_{\text{base}} \ \llbracket \text{insert} \ (j \ \tau) \ \top \text{Rep-Set}_{\text{base}} \ (\text{Abs-Set}_{\text{base}} \ \llbracket \text{insert} \ (i \ \tau) \ \top \text{Rep-Set}_{\text{base}} \ (S \ \tau) \rrbracket_{\perp}) \rrbracket_{\perp} =$
 $\text{Abs-Set}_{\text{base}} \ \llbracket \text{insert} \ (i \ \tau) \ \top \text{Rep-Set}_{\text{base}} \ (\text{Abs-Set}_{\text{base}} \ \llbracket \text{insert} \ (j \ \tau) \ \top \text{Rep-Set}_{\text{base}} \ (S \ \tau) \rrbracket_{\perp}) \rrbracket_{\perp}$
by(*simp add: Abs-Set_{base}-inverse[OF A] Abs-Set_{base}-inverse[OF B] Set.insert-commute*)
show *?thesis*
apply(*simp add: OclIncluding-def S-def[simplified OclValid-def]*
i-val[simplified OclValid-def] j-val[simplified OclValid-def]
true-def OclValid-def StrongEq-def)
apply(*subst cp-defined,*
simp add: S-def[simplified OclValid-def]
*i-val[simplified OclValid-def] j-val[simplified OclValid-def] true-def **)
apply(*subst cp-defined,*
simp add: S-def[simplified OclValid-def]
*i-val[simplified OclValid-def] j-val[simplified OclValid-def] true-def ** ****)
apply(*subst cp-defined,*
simp add: S-def[simplified OclValid-def]
*i-val[simplified OclValid-def] j-val[simplified OclValid-def] true-def **)
apply(*subst cp-defined,*
simp add: S-def[simplified OclValid-def]
*i-val[simplified OclValid-def] j-val[simplified OclValid-def] true-def **)
apply(*subst cp-defined,*
simp add: S-def[simplified OclValid-def]
*i-val[simplified OclValid-def] j-val[simplified OclValid-def] true-def ** *)*
done
qed

lemma *OclIncluding-commute[simp,code-unfold]*:

$((S \ :: \ (\mathcal{A}, \ 'a::\text{null}) \ \text{Set}) \rightarrow \text{including}_{\text{Set}}(i) \rightarrow \text{including}_{\text{Set}}(j)) = (S \rightarrow \text{including}_{\text{Set}}(j) \rightarrow \text{including}_{\text{Set}}(i))$

proof –

have $A : \bigwedge \tau. \ \tau \models (i \ \triangleq \ \text{invalid}) \implies (S \rightarrow \text{including}_{\text{Set}}(i) \rightarrow \text{including}_{\text{Set}}(j)) \ \tau = \text{invalid} \ \tau$

```

    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
  have A':  $\bigwedge \tau. \tau \models (i \triangleq \text{invalid}) \implies (S \rightarrow \text{including}_{\text{Set}}(j) \rightarrow \text{including}_{\text{Set}}(i)) \tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
  have B':  $\bigwedge \tau. \tau \models (j \triangleq \text{invalid}) \implies (S \rightarrow \text{including}_{\text{Set}}(i) \rightarrow \text{including}_{\text{Set}}(j)) \tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
  have B'':  $\bigwedge \tau. \tau \models (j \triangleq \text{invalid}) \implies (S \rightarrow \text{including}_{\text{Set}}(j) \rightarrow \text{including}_{\text{Set}}(i)) \tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
  have C':  $\bigwedge \tau. \tau \models (S \triangleq \text{invalid}) \implies (S \rightarrow \text{including}_{\text{Set}}(i) \rightarrow \text{including}_{\text{Set}}(j)) \tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
  have C'':  $\bigwedge \tau. \tau \models (S \triangleq \text{invalid}) \implies (S \rightarrow \text{including}_{\text{Set}}(j) \rightarrow \text{including}_{\text{Set}}(i)) \tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
  have D':  $\bigwedge \tau. \tau \models (S \triangleq \text{null}) \implies (S \rightarrow \text{including}_{\text{Set}}(i) \rightarrow \text{including}_{\text{Set}}(j)) \tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
  have D'':  $\bigwedge \tau. \tau \models (S \triangleq \text{null}) \implies (S \rightarrow \text{including}_{\text{Set}}(j) \rightarrow \text{including}_{\text{Set}}(i)) \tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
  show ?thesis
    apply(rule ext, rename-tac  $\tau$ )
    apply(case-tac  $\tau \models (v \ i)$ )
    apply(case-tac  $\tau \models (v \ j)$ )
    apply(case-tac  $\tau \models (\delta \ S)$ )
    apply(simp only: OclIncluding-commute0[THEN foundation22[THEN iffD1]])
    apply(simp add: foundation16', elim disjE)
    apply(simp add: C'[OF foundation22[THEN iffD2]] C''[OF foundation22[THEN iffD2]])
    apply(simp add: D'[OF foundation22[THEN iffD2]] D''[OF foundation22[THEN iffD2]])
    apply(simp add: foundation18 B'[OF foundation22[THEN iffD2]] B''[OF foundation22[THEN iffD2]])
    apply(simp add: foundation18 A'[OF foundation22[THEN iffD2]] A''[OF foundation22[THEN iffD2]])
  done
qed

```

Execution Rules on Excluding

```

lemma OclExcluding-finite-rep-set :
  assumes X-def :  $\tau \models \delta \ X$ 
    and x-val :  $\tau \models v \ x$ 
    shows finite  $\ulcorner \text{Rep-Set}_{\text{base}} (X \rightarrow \text{excluding}_{\text{Set}}(x) \tau) \urcorner = \text{finite } \ulcorner \text{Rep-Set}_{\text{base}} (X \ \tau) \urcorner$ 
  proof -
    have C :  $\ulcorner \text{Rep-Set}_{\text{base}} (X \ \tau) \urcorner - \{x \ \tau\}_{\perp} \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \ulcorner X \urcorner. x \neq \text{bot})\}$ 
      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-Setbase-inverse[OF C]
        dest: foundation13[THEN iffD2, THEN foundation22[THEN iffD1]])
  qed

```

```

lemma OclExcluding-rep-set:
  assumes S-def:  $\tau \models \delta \ S$ 
    shows  $\ulcorner \text{Rep-Set}_{\text{base}} (S \rightarrow \text{excluding}_{\text{Set}}(\lambda \cdot. \ulcorner x \urcorner) \tau) \urcorner = \ulcorner \text{Rep-Set}_{\text{base}} (S \ \tau) \urcorner - \{\ulcorner x \urcorner\}$ 
  apply(simp add: OclExcluding-def S-def[simplified OclValid-def])

```

apply(*subst Abs-Set_{base}-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*)

lemma *OclExcluding-excludes0*:

assumes $\tau \models X \rightarrow \text{excludes}_{\text{Set}}(x)$

shows $X \rightarrow \text{excluding}_{\text{Set}}(x) \tau = X \tau$

proof –

have *excludes-def*: $\tau \models X \rightarrow \text{excludes}_{\text{Set}}(x) \implies \tau \models \delta X$

by (*metis OclExcludes.def-valid-then-def OclExcludes-valid-args-valid'' foundation10' foundation6*)

have *excludes-val*: $\tau \models X \rightarrow \text{excludes}_{\text{Set}}(x) \implies \tau \models v x$

by (*metis OclExcludes.def-valid-then-def OclExcludes-valid-args-valid'' foundation10' foundation6*)

show *?thesis*

apply(*insert excludes-def[OF assms] excludes-val[OF assms] assms,*

simp add: OclExcluding-def OclExcludes-def OclIncludes-def OclNot-def OclValid-def true-def)

by (*metis (opaque-lifting, no-types) abs-rep-simp' assms excludes-def*)

qed

lemma *OclExcluding-excludes*:

assumes $\tau \models X \rightarrow \text{excludes}_{\text{Set}}(x)$

shows $\tau \models X \rightarrow \text{excluding}_{\text{Set}}(x) \triangleq X$

by(*simp add: StrongEq-def OclValid-def true-def OclExcluding-excludes0[OF assms]*)

lemma *OclExcluding-chn0[*simp*]*:

assumes $\text{val-}x:\tau \models (v x)$

shows $\tau \models ((\text{Set}\{\}\rightarrow \text{excluding}_{\text{Set}}(x)) \triangleq \text{Set}\{\})$

proof –

have $A : \perp_{\text{None}} \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in {}^\top X^\top. x \neq \text{bot})\}$

by(*simp add: null-option-def bot-option-def*)

have $B : \perp_{\{\}} \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in {}^\top X^\top. x \neq \text{bot})\}$ **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_{base}-def)

apply(*auto simp: mtSet-def Set_{base}.Abs-Set_{base}-inverse*

Set_{base}.Abs-Set_{base}-inject[OF B A])

done

qed

lemma *OclExcluding-commute0* :

assumes $S\text{-def} : \tau \models \delta S$

and $i\text{-val} : \tau \models v i$

and $j\text{-val} : \tau \models v j$

shows $\tau \models ((S \text{ :: } ('A, 'a::\text{null}) \text{ Set}) \rightarrow \text{excluding}_{\text{Set}}(i) \rightarrow \text{excluding}_{\text{Set}}(j)) \triangleq$

$(S \rightarrow \text{excluding}_{\text{Set}}(j) \rightarrow \text{excluding}_{\text{Set}}(i))$

proof –

have $A : \perp_{\text{Rep-Set}_{\text{base}}} (S \tau)^\top - \{i \tau\}_{\perp} \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in {}^\top X^\top. x \neq \text{bot})\}$

by(*insert S-def i-val, frule Set-inv-lemma, simp add: foundation18 invalid-def*)

have $B : \perp_{\text{Rep-Set}_{\text{base}}} (S \tau)^\top - \{j \tau\}_{\perp} \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in {}^\top X^\top. x \neq \text{bot})\}$

by(*insert S-def j-val, frule Set-inv-lemma, simp add: foundation18 invalid-def*)

have $G1 : \text{Abs-Set}_{\text{base}} \perp_{\text{Rep-Set}_{\text{base}}} (S \tau)^\top - \{i \tau\}_{\perp} \neq \text{Abs-Set}_{\text{base}} \text{None}$

by(*insert A, simp add: Abs-Set_{base}-inject bot-option-def null-option-def*)

have $G2 : \text{Abs-Set}_{\text{base}} \perp_{\text{Rep-Set}_{\text{base}}} (S \tau)^\top - \{i \tau\}_{\perp} \neq \text{Abs-Set}_{\text{base}} \perp_{\text{None}}$

by(*insert A, simp add: Abs-Set_{base}-inject bot-option-def null-option-def*)

have $G3 : Abs\text{-}Set_{base} \sqcup^{\top} Rep\text{-}Set_{base} (S \tau)^{\top} - \{j \tau\}_{\sqcup} \neq Abs\text{-}Set_{base} None$
by(*insert B, simp add: Abs-Set_{base}-inject bot-option-def null-option-def*)
have $G4 : Abs\text{-}Set_{base} \sqcup^{\top} Rep\text{-}Set_{base} (S \tau)^{\top} - \{j \tau\}_{\sqcup} \neq Abs\text{-}Set_{base} \perp None_{\perp}$
by(*insert B, simp add: Abs-Set_{base}-inject bot-option-def null-option-def*)

have * : $(\delta (\lambda \cdot Abs\text{-}Set_{base} \sqcup^{\top} Rep\text{-}Set_{base} (S \tau)^{\top} - \{i \tau\}_{\sqcup})) \tau = \perp True_{\perp}$
by(*auto simp: OclValid-def false-def defined-def null-fun-def true-def bot-fun-def bot-Set_{base}-def null-Set_{base}-def S-def i-val G1 G2*)

have ** : $(\delta (\lambda \cdot Abs\text{-}Set_{base} \sqcup^{\top} Rep\text{-}Set_{base} (S \tau)^{\top} - \{j \tau\}_{\sqcup})) \tau = \perp True_{\perp}$
by(*auto simp: OclValid-def false-def defined-def null-fun-def true-def bot-fun-def bot-Set_{base}-def null-Set_{base}-def S-def i-val G3 G4*)

have *** : $Abs\text{-}Set_{base} \sqcup^{\top} Rep\text{-}Set_{base} (Abs\text{-}Set_{base} \sqcup^{\top} Rep\text{-}Set_{base} (S \tau)^{\top} - \{i \tau\}_{\sqcup})^{\top} - \{j \tau\}_{\sqcup} =$
 $Abs\text{-}Set_{base} \sqcup^{\top} Rep\text{-}Set_{base} (Abs\text{-}Set_{base} \sqcup^{\top} Rep\text{-}Set_{base} (S \tau)^{\top} - \{j \tau\}_{\sqcup})^{\top} - \{i \tau\}_{\sqcup}$
apply(*simp add: Abs-Set_{base}-inverse[OF A] Abs-Set_{base}-inverse[OF B]*)
by (*metis Diff-insert2 insert-commute*)
show ?thesis
apply(*simp add: OclExcluding-def S-def[simplified OclValid-def]*
i-val[simplified OclValid-def] j-val[simplified OclValid-def]
true-def OclValid-def StrongEq-def)
apply(*subst cp-defined,*
simp add: S-def[simplified OclValid-def]
*i-val[simplified OclValid-def] j-val[simplified OclValid-def] true-def **)
apply(*subst cp-defined,*
simp add: S-def[simplified OclValid-def]
*i-val[simplified OclValid-def] j-val[simplified OclValid-def] true-def ** ****)
apply(*subst cp-defined,*
simp add: S-def[simplified OclValid-def]
*i-val[simplified OclValid-def] j-val[simplified OclValid-def] true-def **)
apply(*subst cp-defined,*
simp add: S-def[simplified OclValid-def]
*i-val[simplified OclValid-def] j-val[simplified OclValid-def] true-def ***)

done
qed

lemma *OclExcluding-commute[simp,code-unfold]:*

$((S :: ('A, 'a::null) Set) \rightarrow \text{excluding}_{Set}(i) \rightarrow \text{excluding}_{Set}(j) = (S \rightarrow \text{excluding}_{Set}(j) \rightarrow \text{excluding}_{Set}(i)))$

proof –

have $A : \bigwedge \tau. \tau \models i \triangleq \text{invalid} \implies (S \rightarrow \text{excluding}_{Set}(i) \rightarrow \text{excluding}_{Set}(j)) \tau = \text{invalid } \tau$
apply(*rule foundation22[THEN iffD1]*)
by(*erule StrongEq-L-subst2-rev, simp,simp*)
have $A' : \bigwedge \tau. \tau \models i \triangleq \text{invalid} \implies (S \rightarrow \text{excluding}_{Set}(j) \rightarrow \text{excluding}_{Set}(i)) \tau = \text{invalid } \tau$
apply(*rule foundation22[THEN iffD1]*)
by(*erule StrongEq-L-subst2-rev, simp,simp*)
have $B : \bigwedge \tau. \tau \models j \triangleq \text{invalid} \implies (S \rightarrow \text{excluding}_{Set}(i) \rightarrow \text{excluding}_{Set}(j)) \tau = \text{invalid } \tau$
apply(*rule foundation22[THEN iffD1]*)
by(*erule StrongEq-L-subst2-rev, simp,simp*)
have $B' : \bigwedge \tau. \tau \models j \triangleq \text{invalid} \implies (S \rightarrow \text{excluding}_{Set}(j) \rightarrow \text{excluding}_{Set}(i)) \tau = \text{invalid } \tau$
apply(*rule foundation22[THEN iffD1]*)
by(*erule StrongEq-L-subst2-rev, simp,simp*)
have $C : \bigwedge \tau. \tau \models S \triangleq \text{invalid} \implies (S \rightarrow \text{excluding}_{Set}(i) \rightarrow \text{excluding}_{Set}(j)) \tau = \text{invalid } \tau$
apply(*rule foundation22[THEN iffD1]*)

```

    by(erule StrongEq-L-subst2-rev, simp,simp)
  have C':  $\bigwedge \tau. \tau \models S \triangleq \text{invalid} \implies (S \rightarrow \text{excluding}_{\text{Set}}(j) \rightarrow \text{excluding}_{\text{Set}}(i)) \tau = \text{invalid} \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
  have D:  $\bigwedge \tau. \tau \models S \triangleq \text{null} \implies (S \rightarrow \text{excluding}_{\text{Set}}(i) \rightarrow \text{excluding}_{\text{Set}}(j)) \tau = \text{invalid} \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
  have D':  $\bigwedge \tau. \tau \models S \triangleq \text{null} \implies (S \rightarrow \text{excluding}_{\text{Set}}(j) \rightarrow \text{excluding}_{\text{Set}}(i)) \tau = \text{invalid} \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
show ?thesis
  apply(rule ext, rename-tac  $\tau$ )
  apply(case-tac  $\tau \models (v \ i)$ )
  apply(case-tac  $\tau \models (v \ j)$ )
  apply(case-tac  $\tau \models (\delta \ S)$ )
  apply(simp only: OclExcluding-commute0[THEN foundation22[THEN iffD1]])
  apply(simp add: foundation16', elim disjE)
  apply(simp add: C[OF foundation22[THEN iffD2]] C'[OF foundation22[THEN iffD2]])
  apply(simp add: D[OF foundation22[THEN iffD2]] D'[OF foundation22[THEN iffD2]])
  apply(simp add: foundation18 B[OF foundation22[THEN iffD2]] B'[OF foundation22[THEN iffD2]])
  apply(simp add: foundation18 A[OF foundation22[THEN iffD2]] A'[OF foundation22[THEN iffD2]])
done
qed

```

lemma *OclExcluding-charn0-exec*[simp,code-unfold]:

$(\text{Set}\{\} \rightarrow \text{excluding}_{\text{Set}}(x)) = (\text{if } (v \ x) \text{ then } \text{Set}\{\} \text{ else } \text{invalid} \text{ endif})$

proof –

have A: $\bigwedge \tau. (\text{Set}\{\} \rightarrow \text{excluding}_{\text{Set}}(\text{invalid})) \tau = (\text{if } (v \ \text{invalid}) \text{ then } \text{Set}\{\} \text{ else } \text{invalid} \text{ endif}) \tau$

by simp

have B: $\bigwedge \tau \ x. \tau \models (v \ x) \implies$

$(\text{Set}\{\} \rightarrow \text{excluding}_{\text{Set}}(x)) \tau = (\text{if } (v \ x) \text{ then } \text{Set}\{\} \text{ else } \text{invalid} \text{ endif}) \tau$

by(simp add: OclExcluding-charn0[THEN foundation22[THEN iffD1]])

show ?thesis

apply(rule ext, rename-tac τ)

apply(case-tac $\tau \models (v \ x)$)

apply(simp add: B)

apply(simp add: foundation18)

apply(subst OclExcluding.cp0, simp)

apply(simp add: cp-OclIf[symmetric] OclExcluding.cp0[symmetric] cp-valid[symmetric] A)

done

qed

lemma *OclExcluding-charn1*:

assumes $\text{def-X}:\tau \models (\delta \ X)$

and $\text{val-x}:\tau \models (v \ x)$

and $\text{val-y}:\tau \models (v \ y)$

and $\text{neq}:\tau \models \text{not}(x \triangleq y)$

shows $\tau \models ((X \rightarrow \text{including}_{\text{Set}}(x)) \rightarrow \text{excluding}_{\text{Set}}(y)) \triangleq ((X \rightarrow \text{excluding}_{\text{Set}}(y)) \rightarrow \text{including}_{\text{Set}}(x))$

proof –

have C: $\perp \text{insert } (x \ \tau) \text{ } \ulcorner \text{Rep-Set}_{\text{base}}(X \ \tau) \urcorner \perp \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \ulcorner X \urcorner. x \neq \text{bot})\}$

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

have D: $\perp \text{ } \ulcorner \text{Rep-Set}_{\text{base}}(X \ \tau) \urcorner \perp - \{y \ \tau\} \perp \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \ulcorner X \urcorner. x \neq \text{bot})\}$

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

have E: $x \ \tau \neq y \ \tau$

by(insert neq,

auto simp: OclValid-def bot-fun-def OclIncluding-def OclIncludes-def

*false-def true-def defined-def valid-def bot-Set_{base}-def
null-fun-def null-Set_{base}-def StrongEq-def OclNot-def*

```

have G1 : Abs-Setbase ⊔ insert (x τ) (⊔Rep-Setbase (X τ)⊔) ≠ Abs-Setbase None
  by(insert C, simp add: Abs-Setbase-inject bot-option-def null-option-def)
have G2 : Abs-Setbase ⊔ insert (x τ) (⊔Rep-Setbase (X τ)⊔) ≠ Abs-Setbase ⊔None
  by(insert C, simp add: Abs-Setbase-inject bot-option-def null-option-def)
have G : (δ (λ-. Abs-Setbase ⊔ insert (x τ) (⊔Rep-Setbase (X τ)⊔)) τ = true τ
  by(auto simp: OclValid-def false-def true-def defined-def
    bot-fun-def bot-Setbase-def null-fun-def null-Setbase-def G1 G2)

have H1 : Abs-Setbase ⊔ (⊔Rep-Setbase (X τ)⊔) - {y τ}⊔ ≠ Abs-Setbase None
  by(insert D, simp add: Abs-Setbase-inject bot-option-def null-option-def)
have H2 : Abs-Setbase ⊔ (⊔Rep-Setbase (X τ)⊔) - {y τ}⊔ ≠ Abs-Setbase ⊔None
  by(insert D, simp add: Abs-Setbase-inject bot-option-def null-option-def)
have H : (δ (λ-. Abs-Setbase ⊔ (⊔Rep-Setbase (X τ)⊔) - {y τ}⊔)) τ = true τ
  by(auto simp: OclValid-def false-def true-def defined-def
    bot-fun-def bot-Setbase-def null-fun-def null-Setbase-def H1 H2)

have Z : insert (x τ) (⊔Rep-Setbase (X τ)⊔) - {y τ} = insert (x τ) (⊔Rep-Setbase (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 foundation16[THEN iffD1]])
apply(subst cp-defined, simp)+
apply(simp add: G H Abs-Setbase-inverse[OF C] Abs-Setbase-inverse[OF D] Z)
done
qed

```

lemma *OclExcluding-charn2:*

```

assumes def-X:τ ⊨ (δ X)
and val-x:τ ⊨ (v x)
shows τ ⊨ (((X->includingSet(x))->excludingSet(x)) ≐ (X->excludingSet(x)))
proof -
have C : ⊔ insert (x τ) (⊔Rep-Setbase (X τ)⊔) ∈ {X. X = bot ∨ X = null ∨ (∀ x∈⊔X⊔. x ≠ bot)}
  by(insert def-X val-x, frule Set-inv-lemma, simp add: foundation18 invalid-def)
have G1 : Abs-Setbase ⊔ insert (x τ) (⊔Rep-Setbase (X τ)⊔) ≠ Abs-Setbase None
  by(insert C, simp add: Abs-Setbase-inject bot-option-def null-option-def)
have G2 : Abs-Setbase ⊔ insert (x τ) (⊔Rep-Setbase (X τ)⊔) ≠ Abs-Setbase ⊔None
  by(insert C, simp add: Abs-Setbase-inject bot-option-def null-option-def)
show ?thesis
apply(insert def-X[THEN foundation16[THEN iffD1]]
  val-x[THEN foundation18[THEN iffD1]])
apply(auto simp: OclValid-def bot-fun-def OclIncluding-def OclIncludes-def false-def true-def
  invalid-def defined-def valid-def bot-Setbase-def null-fun-def null-Setbase-def
  StrongEq-def)
apply(subst OclExcluding.cp0)
apply(auto simp: OclExcluding-def)
  apply(simp add: Abs-Setbase-inverse[OF C])
  apply(simp-all add: false-def true-def defined-def valid-def
    null-fun-def bot-fun-def null-Setbase-def bot-Setbase-def
    split: bool.split-asm HOL.if-split-asm option.split)
apply(auto simp: G1 G2)
done

```

qed

theorem *OclExcluding-charn3*: $((X \rightarrow \text{including}_{\text{Set}}(x)) \rightarrow \text{excluding}_{\text{Set}}(x)) = (X \rightarrow \text{excluding}_{\text{Set}}(x))$

proof –

```

have A1 :  $\bigwedge \tau. \tau \models (X \triangleq \text{invalid}) \implies (X \rightarrow \text{including}_{\text{Set}}(x) \rightarrow \text{excluding}_{\text{Set}}(x)) \tau = \text{invalid } \tau$ 
  apply(rule foundation22[THEN iffD1])
  by(erule StrongEq-L-subst2-rev, simp,simp)
have A1':  $\bigwedge \tau. \tau \models (X \triangleq \text{invalid}) \implies (X \rightarrow \text{excluding}_{\text{Set}}(x)) \tau = \text{invalid } \tau$ 
  apply(rule foundation22[THEN iffD1])
  by(erule StrongEq-L-subst2-rev, simp,simp)
have A2 :  $\bigwedge \tau. \tau \models (X \triangleq \text{null}) \implies (X \rightarrow \text{including}_{\text{Set}}(x) \rightarrow \text{excluding}_{\text{Set}}(x)) \tau = \text{invalid } \tau$ 
  apply(rule foundation22[THEN iffD1])
  by(erule StrongEq-L-subst2-rev, simp,simp)
have A2':  $\bigwedge \tau. \tau \models (X \triangleq \text{null}) \implies (X \rightarrow \text{excluding}_{\text{Set}}(x)) \tau = \text{invalid } \tau$ 
  apply(rule foundation22[THEN iffD1])
  by(erule StrongEq-L-subst2-rev, simp,simp)
have A3 :  $\bigwedge \tau. \tau \models (x \triangleq \text{invalid}) \implies (X \rightarrow \text{including}_{\text{Set}}(x) \rightarrow \text{excluding}_{\text{Set}}(x)) \tau = \text{invalid } \tau$ 
  apply(rule foundation22[THEN iffD1])
  by(erule StrongEq-L-subst2-rev, simp,simp)
have A3':  $\bigwedge \tau. \tau \models (x \triangleq \text{invalid}) \implies (X \rightarrow \text{excluding}_{\text{Set}}(x)) \tau = \text{invalid } \tau$ 
  apply(rule foundation22[THEN iffD1])
  by(erule StrongEq-L-subst2-rev, simp,simp)

```

show *?thesis*

```

apply(rule ext, rename-tac  $\tau$ )
apply(case-tac  $\tau \models (v \ x)$ )
apply(case-tac  $\tau \models (\delta \ X)$ )
  apply(simp only: OclExcluding-charn2[THEN foundation22[THEN iffD1]])
  apply(simp add: foundation16', elim disjE)
  apply(simp add: A1[OF foundation22[THEN iffD2]] A1'[OF foundation22[THEN iffD2]])
  apply(simp add: A2[OF foundation22[THEN iffD2]] A2'[OF foundation22[THEN iffD2]])
apply(simp add: foundation18 A3[OF foundation22[THEN iffD2]] A3'[OF foundation22[THEN iffD2]])
done

```

qed

One would like a generic theorem of the form:

lemma *OclExcluding_charn_exec*:

$$\begin{aligned}
 & "(X \rightarrow \text{including}_{\text{Set}}(x::('A, 'a)::\text{null})\text{val}) \rightarrow \text{excluding}_{\text{Set}}(y)) = \\
 & \quad (\text{if } \delta \ X \text{ then if } x \doteq y \\
 & \quad \quad \text{then } X \rightarrow \text{excluding}_{\text{Set}}(y) \\
 & \quad \quad \text{else } X \rightarrow \text{excluding}_{\text{Set}}(y) \rightarrow \text{including}_{\text{Set}}(x) \\
 & \quad \quad \text{endif} \\
 & \quad \text{else invalid endif})"
 \end{aligned}$$

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

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

lemma *OclExcluding-charn-exec*:

```

assumes strict1:  $(\text{invalid} \doteq y) = \text{invalid}$ 
and strict2:  $(x \doteq \text{invalid}) = \text{invalid}$ 

```

and *StrictRefEq-valid-args-valid*: $\bigwedge (x::('A, 'a::\text{null})\text{val}) y \tau.$
 $(\tau \models \delta (x \doteq y)) = ((\tau \models (v x)) \wedge (\tau \models v y))$

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

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

shows $(X \rightarrow \text{including}_{\text{Set}}(x::('A, 'a::\text{null})\text{val}) \rightarrow \text{excluding}_{\text{Set}}(y)) =$
 $(\text{if } \delta X \text{ then if } x \doteq y$
 $\quad \text{then } X \rightarrow \text{excluding}_{\text{Set}}(y)$
 $\quad \text{else } X \rightarrow \text{excluding}_{\text{Set}}(y) \rightarrow \text{including}_{\text{Set}}(x)$
 $\quad \text{endif}$
 $\quad \text{else invalid endif})$

proof –

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

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

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

have *B2*: $\bigwedge \tau. \tau \models (X \triangleq \text{null}) \implies X \rightarrow \text{including}_{\text{Set}}(x) \rightarrow \text{excluding}_{\text{Set}}(y) \tau = \text{invalid } \tau$
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*: $\bigwedge \tau. \tau \models (x \triangleq \text{invalid}) \implies$
 $(X \rightarrow \text{including}_{\text{Set}}(x) \rightarrow \text{excluding}_{\text{Set}}(y)) \tau =$
 $(\text{if } x \doteq y \text{ then } X \rightarrow \text{excluding}_{\text{Set}}(y) \text{ else } X \rightarrow \text{excluding}_{\text{Set}}(y) \rightarrow \text{including}_{\text{Set}}(x) \text{ endif}) \tau$
apply(*rule foundation22[THEN iffD1]*)
apply(*erule StrongEq-L-subst2-rev, simp, simp*)
by(*simp add: strict1*)

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

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

have *F*: $\bigwedge \tau. \tau \models \delta X \implies \tau \models v x \implies \tau \models (x \triangleq y) \implies$
 $(X \rightarrow \text{including}_{\text{Set}}(x) \rightarrow \text{excluding}_{\text{Set}}(y)) \tau = (X \rightarrow \text{excluding}_{\text{Set}}(y)) \tau$
apply(*drule StrongEq-L-sym*)

```

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

```

```

show ?thesis
apply(rule ext, rename-tac  $\tau$ )
apply(case-tac  $\neg (\tau \models (\delta X))$ ), simp add:defined-split,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: OclExcluding-charn1[simplified foundation22]
  OclExcluding-charn2[simplified foundation22])
apply(simp add: foundation9 F)
done
qed

```

```

schematic-goal OclExcluding-charn-execInteger[simp,code-unfold]: ?X
by(rule OclExcluding-charn-exec[OF StrictRefEqInteger.strict1 StrictRefEqInteger.strict2
  StrictRefEqInteger.defined-args-valid
  StrictRefEqInteger.cp0 StrictRefEqInteger.StrictRefEq-vs-StrongEq], simp-all)

```

```

schematic-goal OclExcluding-charn-execBoolean[simp,code-unfold]: ?X
by(rule OclExcluding-charn-exec[OF StrictRefEqBoolean.strict1 StrictRefEqBoolean.strict2
  StrictRefEqBoolean.defined-args-valid
  StrictRefEqBoolean.cp0 StrictRefEqBoolean.StrictRefEq-vs-StrongEq], simp-all)

```

```

schematic-goal OclExcluding-charn-execSet[simp,code-unfold]: ?X
by(rule OclExcluding-charn-exec[OF StrictRefEqSet.strict1 StrictRefEqSet.strict2
  StrictRefEqSet.defined-args-valid
  StrictRefEqSet.cp0 StrictRefEqSet.StrictRefEq-vs-StrongEq], simp-all)

```

Execution Rules on Includes

```

lemma OclIncludes-charn0[simp]:
assumes val-x: $\tau \models (v x)$ 
shows  $\tau \models \text{not}(\text{Set}\{\}\text{->includes}_{\text{Set}}(x))$ 
using val-x
apply(auto simp: OclValid-def OclIncludes-def OclNot-def false-def true-def)
apply(auto simp: mtSet-def Setbase.Abs-Setbase-inverse)
done

```

```

lemma OclIncludes-charn0'[simp,code-unfold]:
Set $\{\}\text{->includes}_{\text{Set}}(x) = (\text{if } v x \text{ then false else invalid endif})$ 
proof -
  have A:  $\bigwedge \tau. (\text{Set}\{\}\text{->includes}_{\text{Set}}(\text{invalid})) \tau = (\text{if } (v \text{ invalid}) \text{ then false else invalid endif}) \tau$ 
    by simp
  have B:  $\bigwedge \tau x. \tau \models (v x) \implies (\text{Set}\{\}\text{->includes}_{\text{Set}}(x)) \tau = (\text{if } v x \text{ then false else invalid endif}) \tau$ 
    apply(frule OclIncludes-charn0, simp add: OclValid-def)

```

```

apply(rule foundation21[THEN fun-cong, simplified StrongEq-def,simplified,
  THEN iffD1, of - - false])
  by simp
show ?thesis
  apply(rule ext, rename-tac  $\tau$ )
  apply(case-tac  $\tau \models (v x)$ )
  apply(simp-all add: B foundation18)
  apply(subst OclIncludes.cp0, simp add: OclIncludes.cp0[symmetric] A)
done
qed

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

```

```

lemma OclIncludes-charn2:
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  $\tau \models (X \rightarrow \text{including}_{Set}(x) \rightarrow \text{includes}_{Set}(y)) \triangleq (X \rightarrow \text{includes}_{Set}(y))$ 
proof -
have C :  $\perp \text{insert } (x \tau) \text{ } \ulcorner \text{Rep-Set}_{base} (X \tau) \urcorner \perp \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \ulcorner X \urcorner. x \neq \text{bot})\}$ 
  by(insert def-X val-x, frule Set-inv-lemma, simp add: foundation18 invalid-def)
show ?thesis
  apply(subst OclIncludes-def,
    simp add: 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-Setbase-inverse[OF C] true-def)
by(metis foundation22 foundation6 foundation9 neq)
qed

```

Here is again a generic theorem similar as above.

```

lemma OclIncludes-execute-generic:
assumes strict1:  $(\text{invalid} \triangleq y) = \text{invalid}$ 
and strict2:  $(x \triangleq \text{invalid}) = \text{invalid}$ 
and cp-StrictRefEq:  $\bigwedge (X::(\mathfrak{A}, 'a::\text{null})\text{val}) Y \tau. (X \triangleq Y) \tau = ((\lambda \cdot. X \tau) \triangleq (\lambda \cdot. Y \tau)) \tau$ 
and StrictRefEq-vs-StrongEq:  $\bigwedge (x::(\mathfrak{A}, 'a::\text{null})\text{val}) y \tau.
  \tau \models v x \implies \tau \models v y \implies (\tau \models ((x \triangleq y) \triangleq (x \triangleq y)))$ 
shows
   $(X \rightarrow \text{including}_{Set}(x::(\mathfrak{A}, 'a::\text{null})\text{val}) \rightarrow \text{includes}_{Set}(y)) =$ 

```

```

    (if  $\delta X$  then if  $x \doteq y$  then true else  $X \rightarrow \text{includes}_{Set}(y)$  endif else invalid endif)
proof -
  have A:  $\bigwedge \tau. \tau \models (X \triangleq \text{invalid}) \implies$ 
    ( $X \rightarrow \text{including}_{Set}(x) \rightarrow \text{includes}_{Set}(y)$ )  $\tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev,simp,simp)
  have B:  $\bigwedge \tau. \tau \models (X \triangleq \text{null}) \implies$ 
    ( $X \rightarrow \text{including}_{Set}(x) \rightarrow \text{includes}_{Set}(y)$ )  $\tau = \text{invalid } \tau$ 
    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:  $\bigwedge \tau. \tau \models (x \triangleq \text{invalid}) \implies$ 
    ( $X \rightarrow \text{including}_{Set}(x) \rightarrow \text{includes}_{Set}(y)$ )  $\tau =$ 
    (if  $x \doteq y$  then true else  $X \rightarrow \text{includes}_{Set}(y)$  endif)  $\tau$ 
    apply(rule foundation22[THEN iffD1])
    apply(erule StrongEq-L-subst2-rev,simp,simp)
    by (simp add: strict1)
  have D:  $\bigwedge \tau. \tau \models (y \triangleq \text{invalid}) \implies$ 
    ( $X \rightarrow \text{including}_{Set}(x) \rightarrow \text{includes}_{Set}(y)$ )  $\tau =$ 
    (if  $x \doteq y$  then true else  $X \rightarrow \text{includes}_{Set}(y)$  endif)  $\tau$ 
    apply(rule foundation22[THEN iffD1])
    apply(erule StrongEq-L-subst2-rev,simp,simp)
    by (simp add: strict2)
  have E:  $\bigwedge \tau. \tau \models v x \implies \tau \models v y \implies$ 
    (if  $x \doteq y$  then true else  $X \rightarrow \text{includes}_{Set}(y)$  endif)  $\tau =$ 
    (if  $x \triangleq y$  then true else  $X \rightarrow \text{includes}_{Set}(y)$  endif)  $\tau$ 
    apply(subst cp-OclIf)
    apply(subst StrictRefEq-vs-StrongEq[THEN foundation22[THEN iffD1]])
    by(simp-all add: cp-OclIf[symmetric])
  have F:  $\bigwedge \tau. \tau \models (x \triangleq y) \implies$ 
    ( $X \rightarrow \text{including}_{Set}(x) \rightarrow \text{includes}_{Set}(y)$ )  $\tau = (X \rightarrow \text{including}_{Set}(x) \rightarrow \text{includes}_{Set}(x)) \tau$ 
    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: defined-split, 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  $\tau \models \text{not}(x \triangleq y)$ )
  apply(simp add: OclIncludes-charn2[simplified foundation22])
  apply(simp add: foundation9 F
    OclIncludes-charn1[THEN foundation13[THEN iffD2],
      THEN foundation22[THEN iffD1]])

done
qed

```

schematic-goal *OclIncludes-execute_{Integer}*[simp,code-unfold]: ?X

by(rule OclIncludes-execute-generic[OF StrictRefEqInteger.strict1 StrictRefEqInteger.strict2
 StrictRefEqInteger.cp0
 StrictRefEqInteger.StrictRefEq-vs-StrongEq], simp-all)

schematic-goal OclIncludes-execute_{Boolean}[simp,code-unfold]: ?X
 by(rule OclIncludes-execute-generic[OF StrictRefEqBoolean.strict1 StrictRefEqBoolean.strict2
 StrictRefEqBoolean.cp0
 StrictRefEqBoolean.StrictRefEq-vs-StrongEq], simp-all)

schematic-goal OclIncludes-execute_{Set}[simp,code-unfold]: ?X
 by(rule OclIncludes-execute-generic[OF StrictRefEqSet.strict1 StrictRefEqSet.strict2
 StrictRefEqSet.cp0
 StrictRefEqSet.StrictRefEq-vs-StrongEq], simp-all)

lemma OclIncludes-including-generic :
assumes OclIncludes-execute-generic [simp] : $\bigwedge X x y.$
 $(X \rightarrow \text{including}_{Set}(x::('A,'a::\text{null})\text{val}) \rightarrow \text{includes}_{Set}(y)) =$
 $(\text{if } \delta X \text{ then if } x \doteq y \text{ then true else } X \rightarrow \text{includes}_{Set}(y) \text{ endif else invalid endif})$
and StrictRefEq-strict'' : $\bigwedge x y. \delta ((x::('A,'a::\text{null})\text{val}) \doteq y) = (v(x) \text{ and } v(y))$
and a-val : $\tau \models v a$
and x-val : $\tau \models v x$
and S-incl : $\tau \models (S) \rightarrow \text{includes}_{Set}((x::('A,'a::\text{null})\text{val}))$
shows $\tau \models S \rightarrow \text{including}_{Set}((a::('A,'a::\text{null})\text{val})) \rightarrow \text{includes}_{Set}(x)$

proof –
have discr-eq-bot1-true : $\bigwedge \tau. (\perp \tau = \text{true } \tau) = \text{False}$
by (metis 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 δS] OclValid-def OclIf-def x-val[simplified OclValid-def]
 discr-neq-invalid-true discr-eq-invalid-true)
by (metis OclValid-def S-incl StrictRefEq-strict'' a-val foundation10 foundation6 x-val)
qed

lemmas OclIncludes-including_{Integer} =
 OclIncludes-including-generic[OF OclIncludes-execute_{Integer} StrictRefEqInteger.def-homo]

Execution Rules on Excludes

lemma OclExcludes-charn1 :
assumes def-X: $\tau \models (\delta X)$
assumes val-x: $\tau \models (v x)$
shows $\tau \models (X \rightarrow \text{excluding}_{Set}(x) \rightarrow \text{excludes}_{Set}(x))$
proof –
let ?OclSet = $\lambda S. \perp S_{\perp} \in \{X. X = \perp \vee X = \text{null} \vee (\forall x \in {}^{\top}X^{\top}. x \neq \perp)\}$
have diff-in-Set_{base} : ?OclSet (${}^{\top}\text{Rep-Set}_{base}(X \tau)^{\top} - \{x \tau\}$)
apply (simp, (rule disjI2)+)

by (metis (opaque-lifting, no-types) Diff-iff Set-inv-lemma def-X)

show ?thesis

apply(subst OclExcludes-def, simp add: foundation10[simplified OclValid-def] OclValid-def
 def-X[simplified OclValid-def] val-x[simplified OclValid-def])
 apply(subst OclIncludes-def, simp add: OclNot-def)
 apply(simp add: OclExcluding-def def-X[simplified OclValid-def] val-x[simplified OclValid-def]
 Abs-Set_{base}-inverse[OF diff-in-Set_{base}] true-def)
 by(simp add: OclAnd-def def-X[simplified OclValid-def] val-x[simplified OclValid-def] true-def)
 qed

Execution Rules on Size

lemma [simp,code-unfold]: Set{} \rightarrow size_{Set}() = 0

apply(rule ext)

apply(simp add: defined-def mtSet-def OclSize-def
 bot-Set_{base}-def bot-fun-def
 null-Set_{base}-def null-fun-def)

apply(subst Abs-Set_{base}-inject, simp-all add: bot-option-def null-option-def) +
 by(simp add: Abs-Set_{base}-inverse bot-option-def null-option-def OclInt0-def)

lemma OclSize-including-exec[simp,code-unfold]:

((X \rightarrow including_{Set}(x)) \rightarrow size_{Set}()) = (if δ X and v x then
 X \rightarrow size_{Set}() +_{int} if X \rightarrow includes_{Set}(x) then 0 else 1 endif
 else
 invalid
 endif)

proof –

have valid-inject-true : $\bigwedge \tau$ P. (v P) $\tau \neq$ true $\tau \implies$ (v P) $\tau =$ false τ

apply(simp add: valid-def true-def false-def bot-fun-def bot-option-def
 null-fun-def null-option-def)

by (case-tac P $\tau = \perp$, simp-all add: true-def)

have defined-inject-true : $\bigwedge \tau$ P. (δ P) $\tau \neq$ true $\tau \implies$ (δ P) $\tau =$ false τ

apply(simp add: defined-def true-def false-def bot-fun-def bot-option-def
 null-fun-def null-option-def)

by (case-tac P $\tau = \perp \vee$ P $\tau =$ null, simp-all add: true-def)

show ?thesis

apply(rule ext, rename-tac τ)

proof –

fix τ

have includes-notin: $\neg \tau \models X \rightarrow$ includes_{Set}(x) \implies (δ X) $\tau =$ true $\tau \wedge$ (v x) $\tau =$ true $\tau \implies$
 $x \tau \notin \ulcorner \text{Rep-Set}_{base} (X \tau) \urcorner$

by(simp add: OclIncludes-def OclValid-def true-def)

have includes-def: $\tau \models X \rightarrow$ includes_{Set}(x) \implies $\tau \models \delta$ X

by (metis bot-fun-def OclIncludes-def OclValid-def defined3 foundation16)

have includes-val: $\tau \models X \rightarrow$ includes_{Set}(x) \implies $\tau \models v$ x

using foundation5 foundation6 by fastforce

have ins-in-Set_{base}: $\tau \models \delta$ X \implies $\tau \models v$ x \implies

$\ulcorner \text{insert} (x \tau) \ulcorner \text{Rep-Set}_{base} (X \tau) \urcorner \urcorner \in \{X. X = \perp \vee X = \text{null} \vee (\forall x \in \ulcorner X \urcorner. x \neq \perp)\}$

apply(simp add: bot-option-def null-option-def)

by (metis (opaque-lifting, no-types) Set-inv-lemma foundation18' foundation5)

have m : $\bigwedge \tau$. ($\lambda \cdot$ \perp) = ($\lambda \cdot$ invalid τ) by (rule ext, simp add: invalid-def)

```

show  $X \rightarrow \text{including}_{\text{Set}}(x) \rightarrow \text{size}_{\text{Set}}()$   $\tau = (\text{if } \delta X \text{ and } v x$ 
       $\text{ then } X \rightarrow \text{size}_{\text{Set}}() +_{\text{int}} \text{ if } X \rightarrow \text{includes}_{\text{Set}}(x) \text{ then } \mathbf{0} \text{ else } \mathbf{1} \text{ endif}$ 
       $\text{ else invalid endif}) \tau$ 
apply(case-tac  $\tau \models \delta X \text{ and } v x$ , simp)
apply(subst OclAddInteger.cp0)
apply(case-tac  $\tau \models X \rightarrow \text{includes}_{\text{Set}}(x)$ , simp add: OclAddInteger.cp0[symmetric])
apply(case-tac  $\tau \models ((v (X \rightarrow \text{size}_{\text{Set}}())) \text{ and not } (\delta (X \rightarrow \text{size}_{\text{Set}}())))$ , simp)
  apply(drule foundation5[where  $P = v X \rightarrow \text{size}_{\text{Set}}()$ ], erule conjE)
  apply(drule OclSize-infinite)
  apply(frule includes-def, drule includes-val, simp)
  apply(subst OclSize-def, subst OclIncluding-finite-rep-set, assumption+)
  apply(metis (opaque-lifting, no-types) invalid-def)

  apply(subst OclIf-false',
    metis (opaque-lifting, no-types) defined5 defined6 defined-and-I defined-not-I
      foundation1 foundation9)
apply(subst cp-OclSize, simp add: OclIncluding-includes0 cp-OclSize[symmetric])

apply(subst OclIf-false', subst foundation9, auto, simp add: OclSize-def)
apply(drule foundation5)
apply(subst (1 2) OclIncluding-finite-rep-set, fast+)
apply(subst (1 2) cp-OclAnd, subst (1 2) OclAddInteger.cp0, simp)
apply(rule conjI)
apply(simp add: OclIncluding-def)
apply(subst Abs-Setbase-inverse[OF ins-in-Setbase], fast+)
apply(subst (asm) (2 3) OclValid-def, simp add: OclAddInteger-def OclInt1-def)
apply(rule impI)
apply(drule Finite-Set.card.insert[where  $x = x \tau$ ])
apply(rule includes-notin, simp, simp)
apply(metis Suc-eq-plus1 of-nat-1 of-nat-add)

apply(subst (1 2) m[of  $\tau$ ], simp only: OclAddInteger.cp0[symmetric], simp, simp add: invalid-def)
apply(subst OclIncluding-finite-rep-set, fast+, simp add: OclValid-def)

apply(subst OclIf-false', metis (opaque-lifting, no-types) defined6 foundation1 foundation9
      OclExcluding-valid-args-valid')
by (metis cp-OclSize foundation18' OclIncluding-valid-args-valid'' invalid-def OclSize-invalid)
qed
qed

```

Execution Rules on IsEmpty

```

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

```

```

lemma OclIsEmpty-including [simp]:

```

```

assumes X-def:  $\tau \models \delta X$ 
  and X-finite:  $\text{finite } \ulcorner \text{Rep-Set}_{\text{base}}(X \tau) \urcorner$ 
  and a-val:  $\tau \models v a$ 

```

```

shows  $X \rightarrow \text{including}_{\text{Set}}(a) \rightarrow \text{isEmpty}_{\text{Set}}() \tau = \text{false } \tau$ 

```

```

proof –

```

```

have A1 :  $\bigwedge \tau X. X \tau = \text{true } \tau \vee X \tau = \text{false } \tau \implies (X \text{ and not } X) \tau = \text{false } \tau$ 

```

```

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 :  $\bigwedge X \tau. \tau \models v X \implies X \tau \neq \mathbf{0} \tau \implies (X \doteq \mathbf{0}) \tau = \text{false } \tau$ 
  apply(simp add: foundation22[symmetric] foundation14 foundation9)
  apply(erule StrongEq-L-subst4-rev[THEN iffD2, OF StrictRefEqInteger.StrictRefEq-vs-StrongEq])
  by(simp-all)

show ?thesis
  apply(simp add: OclIsEmpty-def del: OclSize-including-exec)
  apply(subst cp-OclOr, subst A1)
  apply (metis OclExcludes.def-homo defined-inject-true)
  apply(simp add: cp-OclOr[symmetric] del: OclSize-including-exec)
  apply(rule B,
    rule foundation20,
    metis OclIncluding.def-homo OclIncluding-finite-rep-set X-def X-finite a-val foundation10' size-defined')
  apply(simp add: OclSize-def OclIncluding-finite-rep-set[OF X-def a-val] X-finite OclInt0-def)
  by (metis OclValid-def X-def a-val foundation10 foundation6
    OclIncluding-notempty-rep-set[OF X-def a-val])
qed

```

Execution Rules on NotEmpty

```

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

lemma OclNotEmpty-including [simp,code-unfold]:
  assumes X-def:  $\tau \models \delta X$ 
    and X-finite:  $\text{finite } \ulcorner \text{Rep-Set}_{\text{base}}(X \tau) \urcorner$ 
    and a-val:  $\tau \models v a$ 
  shows  $X\text{->includingSet}(a)\text{->notEmptySet}() \tau = \text{true } \tau$ 
  apply(simp add: OclNotEmpty-def)
  apply(subst cp-OclNot, subst OclIsEmpty-including, simp-all add: assms)
  by (metis OclNot4 cp-OclNot)

```

Execution Rules on Any

```

lemma [simp,code-unfold]:  $\text{Set}\{\}\text{->anySet}() = \text{null}$ 
  by(rule ext, simp add: OclANY-def, simp add: false-def true-def)

lemma OclANY-singleton-exec[simp,code-unfold]:
   $(\text{Set}\{\}\text{->includingSet}(a)\text{->anySet}())\text{->anySet}() = a$ 
  apply(rule ext, rename-tac  $\tau$ , simp add: mtSet-def OclANY-def)
  apply(case-tac  $\tau \models v a$ )
  apply(simp add: OclValid-def mtSet-defined[simplified mtSet-def]
    mtSet-valid[simplified mtSet-def] mtSet-rep-set[simplified mtSet-def])
  apply(subst (1 2) cp-OclAnd,
    subst (1 2) OclNotEmpty-including[where X =  $\text{Set}\{\}$ , simplified mtSet-def])
  apply(simp add: mtSet-defined[simplified mtSet-def])
  apply(metis (opaque-lifting, 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_base-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 τ = ⊥)
prefer 2
apply(simp add: OclValid-def valid-def bot-fun-def split: if-split-asm)
apply(simp)
apply(subst (1 2 3 4) cp-OclAnd,
      simp add: mtSet-defined[simplified mtSet-def] valid-def bot-fun-def)
by(simp add: cp-OclAnd[symmetric], rule impI, simp add: false-def true-def)

```

Execution Rules on Forall

```

lemma OclForall-mtSet-exec[simp,code-unfold] : ((Set{ }) -> forallSet(z | P(z))) = true
apply(simp add: OclForall-def)
apply(subst mtSet-def)+
apply(subst Abs-Setbase-inverse, simp-all add: true-def)+
done

```

The following rule is a main theorem of our approach: From a denotational definition that assures consistency, but may be — as in the case of the *OclForall X P* — dauntingly complex, we derive operational rules that can serve as a gold-standard for operational execution, since they may be evaluated in whatever situation and according to whatever strategy. In the case of *OclForall X P*, the operational rule gives immediately a way to evaluation in any finite (in terms of conventional OCL: denotable) set, although the rule also holds for the infinite case:

```

Integernull -> forallSet(x | Integernull -> forallSet(y | x +int y  $\triangleq$  y +int x))
or even:
Integer -> forallSet(x | Integer -> forallSet(y | x +int y  $\doteq$  y +int x))
are valid OCL statements in any context  $\tau$ .

```

theorem *OclForall-including-exec[*simp,code-unfold*] :*

```

assumes cp0 : cp P
shows      ((S -> includingSet(x) -> forallSet(z | P(z))) = (if δ S and v x
                                                    then P x and (S -> forallSet(z | P(z)))
                                                    else invalid
                                                    endif)

```

proof –

```

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

```

```

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

```

```

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

```

```

have insert-in-Setbase:  $\bigwedge \tau. (\tau \models (\delta\ S)) \implies (\tau \models (v\ x)) \implies$ 
       $\perp \text{insert}(x\ \tau) \text{Rep-Set}_{base}(S\ \tau) \perp \in$ 
       $\{X. X = bot \vee X = null \vee (\forall x \in \text{Rep-Set}_{base}(S\ \tau). x \neq bot)\}$ 
by(frule Set-inv-lemma, simp add: foundation18 invalid-def)

```

```

have forall-including-invert:  $\bigwedge \tau\ f. (f\ x\ \tau = f\ (\lambda \cdot. x\ \tau)\ \tau) \implies$ 
       $\tau \models (\delta\ S\ \text{and}\ v\ x) \implies$ 
       $(\forall x \in \text{Rep-Set}_{base}(S\ \tau). (S -> including_{Set}(x)\ \tau)^\top. f\ (\lambda \cdot. x)\ \tau) =$ 
       $(f\ x\ \tau \wedge (\forall x \in \text{Rep-Set}_{base}(S\ \tau). f\ (\lambda \cdot. x)\ \tau))$ 

```

```

apply(drule foundation5, simp add: OclIncluding-def)

```

```

apply(subst Abs-Setbase-inverse)

```

```

apply(rule insert-in-Setbase, fast+)

```

```

by(simp add: OclValid-def)

```

```

have exists-including-invert:  $\bigwedge \tau\ f. (f\ x\ \tau = f\ (\lambda \cdot. x\ \tau)\ \tau) \implies$ 
       $\tau \models (\delta\ S\ \text{and}\ v\ x) \implies$ 
       $(\exists x \in \text{Rep-Set}_{base}(S\ \tau). (S -> including_{Set}(x)\ \tau)^\top. f\ (\lambda \cdot. x)\ \tau) =$ 
       $(f\ x\ \tau \vee (\exists x \in \text{Rep-Set}_{base}(S\ \tau). f\ (\lambda \cdot. x)\ \tau))$ 

```

```

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 contradict-Rep-Setbase:  $\bigwedge \tau S f. \exists x \in {}^\top \text{Rep-Set}_{base} S {}^\top. f (\lambda \cdot. x) \tau \implies$ 
   $(\forall x \in {}^\top \text{Rep-Set}_{base} S {}^\top. \neg (f (\lambda \cdot. x) \tau)) = \text{False}$ 
by(case-tac  $(\forall x \in {}^\top \text{Rep-Set}_{base} S {}^\top. \neg (f (\lambda \cdot. x) \tau)) = \text{True}$ , simp-all)

have bot-invalid :  $\perp = \text{invalid}$  by(rule ext, simp add: invalid-def bot-fun-def)

have bot-invalid2 :  $\bigwedge \tau. \perp = \text{invalid } \tau$  by(simp add: invalid-def)

have C1 :  $\bigwedge \tau. P x \tau = \text{false } \tau \vee (\exists x \in {}^\top \text{Rep-Set}_{base} (S \tau) {}^\top. P (\lambda \cdot. x) \tau = \text{false } \tau) \implies$ 
   $\tau \models (\delta S \text{ and } v x) \implies$ 
   $\text{false } \tau = (P x \text{ and } \text{OclForall } S P) \tau$ 
apply(simp add: cp-OclAnd[of P x])
apply(elim disjE, simp)
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(fold OclValid-def, simp add: foundation10')
done

have C2 :  $\bigwedge \tau. \tau \models (\delta S \text{ and } v x) \implies$ 
   $P x \tau = \text{null } \tau \vee (\exists x \in {}^\top \text{Rep-Set}_{base} (S \tau) {}^\top. P (\lambda \cdot. x) \tau = \text{null } \tau) \implies$ 
   $P x \tau = \text{invalid } \tau \vee (\exists x \in {}^\top \text{Rep-Set}_{base} (S \tau) {}^\top. P (\lambda \cdot. x) \tau = \text{invalid } \tau) \implies$ 
   $\forall x \in {}^\top \text{Rep-Set}_{base} (S \rightarrow \text{including}_{Set}(x) \tau) {}^\top. P (\lambda \cdot. x) \tau \neq \text{false } \tau \implies$ 
   $\text{invalid } \tau = (P x \text{ and } \text{OclForall } S P) \tau$ 
apply(subgoal-tac  $(\delta S) \tau = \text{true } \tau$ )
prefer 2 apply(simp add: foundation10', simp add: OclValid-def)
apply(drule forall-including-invert[of  $\lambda x \tau. P x \tau \neq \text{false } \tau$ , OF cp-OclNot-eq, THEN iffD1])
apply(assumption)
apply(simp add: cp-OclAnd[of P x], elim disjE, simp-all)
apply(simp add: invalid-def null-fun-def null-option-def bot-fun-def bot-option-def)
apply(subgoal-tac OclForall S P  $\tau = \text{invalid } \tau$ )
apply(simp only: cp-OclAnd[symmetric], simp, simp add: invalid-def bot-fun-def)
apply(unfold OclForall-def, simp add: invalid-def false-def bot-fun-def, simp)
apply(simp add: cp-OclAnd[symmetric], simp)
apply(erule conjE)
apply(subgoal-tac  $(P x \tau = \text{invalid } \tau) \vee (P x \tau = \text{null } \tau) \vee (P x \tau = \text{true } \tau) \vee (P x \tau = \text{false } \tau)$ )
prefer 2 apply(rule bool-split-0)
apply(elim disjE, simp-all)
apply(simp only: cp-OclAnd[symmetric], simp)+
done

have A :  $\bigwedge \tau. \tau \models (\delta S \text{ and } v x) \implies$ 
   $\text{OclForall } (S \rightarrow \text{including}_{Set}(x)) P \tau = (P x \text{ and } \text{OclForall } S P) \tau$ 
proof – fix  $\tau$ 
assume 0 :  $\tau \models (\delta S \text{ and } v x)$ 
let ?S =  $\lambda \text{ocl}. P x \tau \neq \text{ocl } \tau \wedge (\forall x \in {}^\top \text{Rep-Set}_{base} (S \tau) {}^\top. P (\lambda \cdot. x) \tau \neq \text{ocl } \tau)$ 
let ?S' =  $\lambda \text{ocl}. \forall x \in {}^\top \text{Rep-Set}_{base} (S \rightarrow \text{including}_{Set}(x) \tau) {}^\top. P (\lambda \cdot. x) \tau \neq \text{ocl } \tau$ 
let ?assms-1 = ?S' null
let ?assms-2 = ?S' invalid
let ?assms-3 = ?S' false
have 4 : ?assms-3  $\implies$  ?S false

```

```

    apply(subst forall-including-invert[of  $\lambda x \tau. P x \tau \neq \text{false } \tau, \text{symmetric}$ ])
    by(simp-all add: cp-OclNot-eq 0)
  have 5 : ?assms-2  $\implies$  ?S invalid
    apply(subst forall-including-invert[of  $\lambda x \tau. P x \tau \neq \text{invalid } \tau, \text{symmetric}$ ])
    by(simp-all add: cp-OclNot-eq 0)
  have 6 : ?assms-1  $\implies$  ?S null
    apply(subst forall-including-invert[of  $\lambda x \tau. P x \tau \neq \text{null } \tau, \text{symmetric}$ ])
    by(simp-all add: cp-OclNot-eq 0)
  have 7 : ( $\delta S$ )  $\tau = \text{true } \tau$ 
    by(insert 0, simp add: foundation10', simp add: OclValid-def)
show ?thesis  $\tau$ 
  apply(subst OclForall-def)
  apply(simp add: cp-OclAnd[THEN sym] OclValid-def contradict-Rep-Setbase)
  apply(intro conjI impI, fold OclValid-def)
  apply(simp-all add: exists-including-invert[where  $f = \lambda x \tau. P x \tau = \text{null } \tau, OF cp\text{-eq}$ ])
  apply(simp-all add: exists-including-invert[where  $f = \lambda x \tau. P x \tau = \text{invalid } \tau, OF cp\text{-eq}$ ])
  apply(simp-all add: exists-including-invert[where  $f = \lambda x \tau. P x \tau = \text{false } \tau, OF cp\text{-eq}$ ])
  proof -
    assume 1 :  $P x \tau = \text{null } \tau \vee (\exists x \in {}^\Gamma \text{Rep-Set}_{base} (S \tau)^\Gamma. P (\lambda \cdot. x) \tau = \text{null } \tau)$ 
    and 2 : ?assms-2
    and 3 : ?assms-3
    show  $\text{null } \tau = (P x \text{ and } OclForall S P) \tau$ 
    proof -
      note 4 = 4[OF 3]
      note 5 = 5[OF 2]
      have 6 :  $P x \tau = \text{null } \tau \vee P x \tau = \text{true } \tau$ 
        by(metis 4 5 bool-split-0)
      show ?thesis
      apply(insert 6, elim disjE)
      apply(subst cp-OclAnd)
      apply(simp add: OclForall-def 7 4[THEN conjunct2] 5[THEN conjunct2])
      apply(simp-all add: cp-OclAnd[symmetric])
      apply(subst cp-OclAnd, simp-all add: cp-OclAnd[symmetric] OclForall-def)
      apply(simp add: 4[THEN conjunct2] 5[THEN conjunct2] 0[simplified OclValid-def] 7)
      apply(insert 1, elim disjE, auto)
      done
    qed
  next
    assume 1 : ?assms-1
    and 2 :  $P x \tau = \text{invalid } \tau \vee (\exists x \in {}^\Gamma \text{Rep-Set}_{base} (S \tau)^\Gamma. P (\lambda \cdot. x) \tau = \text{invalid } \tau)$ 
    and 3 : ?assms-3
    show  $\text{invalid } \tau = (P x \text{ and } OclForall S P) \tau$ 
    proof -
      note 4 = 4[OF 3]
      note 6 = 6[OF 1]
      have 5 :  $P x \tau = \text{invalid } \tau \vee P x \tau = \text{true } \tau$ 
        by(metis 4 6 bool-split-0)
      show ?thesis
      apply(insert 5, elim disjE)
      apply(subst cp-OclAnd)
      apply(simp add: OclForall-def 4[THEN conjunct2] 6[THEN conjunct2] 7)
      apply(simp-all add: cp-OclAnd[symmetric])
      apply(subst cp-OclAnd, simp-all add: cp-OclAnd[symmetric] OclForall-def)
      apply(insert 2, elim disjE, simp add: invalid-def true-def bot-option-def)
      apply(simp add: 0[simplified OclValid-def] 4[THEN conjunct2] 6[THEN conjunct2] 7)
      by(auto)
    qed
  qed

```

```

next
  assume 1 : ?assms-1
  and 2 : ?assms-2
  and 3 : ?assms-3
  show true  $\tau = (P\ x\ \text{and}\ \text{OclForall}\ S\ P)\ \tau$ 
  proof -
    note 4 = 4[OF 3]
    note 5 = 5[OF 2]
    note 6 = 6[OF 1]
    have 8 :  $P\ x\ \tau = \text{true}\ \tau$ 
      by(metis 4 5 6 bool-split-0)
    show ?thesis
    apply(simp cp-OclAnd, simp add: 8 cp-OclAnd[symmetric])
    by(simp add: OclForall-def 4 5 6 7)
  qed
qed ( simp add: 0
  | rule C1, simp+
  | rule C2, simp add: 0 )+
qed

```

```

have B :  $\bigwedge \tau. \neg (\tau \models (\delta\ S\ \text{and}\ v\ x)) \implies$ 
  OclForall ( $S \rightarrow \text{including}_{Set}(x)$ )  $P\ \tau = \text{invalid}\ \tau$ 
  apply(rule foundation22[THEN iffD1])
  apply(simp only: foundation10' de-Morgan-conj foundation18'', elim disjE)
  apply(simp add: defined-split, elim disjE)
  apply(erule StrongEq-L-subst2-rev, simp+)+
done

```

```

show ?thesis
  apply(rule ext, rename-tac  $\tau$ )
  apply(simp add: OclIf-def)
  apply(simp add: cp-defined[of  $\delta\ S\ \text{and}\ v\ x$ ] cp-defined[THEN sym])
  apply(intro conjI impI)
  by(auto intro!: A B simp: OclValid-def)
qed

```

Execution Rules on Exists

```

lemma OclExists-mtSet-exec[simp,code-unfold] :
  ((Set{ })  $\rightarrow$  exists $_{Set}(z \mid P(z))$ ) = false
  by(simp add: OclExists-def)

```

```

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

```

Execution Rules on Iterate

```

lemma OclIterate-empty[simp,code-unfold]: ((Set{ })  $\rightarrow$  iterate $_{Set}(a; x = A \mid P\ a\ x)$ ) = A
proof -
  have C :  $\bigwedge \tau. (\delta (\lambda \tau. \text{Abs-Set}_{base\ \perp\ \perp}\{\}_{\perp}))\ \tau = \text{true}\ \tau$ 
  by (metis (no-types) defined-def mtSet-def mtSet-defined null-fun-def)
  show ?thesis
    apply(simp add: OclIterate-def mtSet-def Abs-Set $_{base}$ -inverse valid-def C)

```

```

apply(rule ext, rename-tac  $\tau$ )
apply(case-tac  $A \tau = \perp \tau$ , simp-all, simp add:true-def false-def bot-fun-def)
apply(simp add: Abs-Setbase-inverse)
done
qed

```

In particular, this does hold for $A = \text{null}$.

lemma *OclIterate-including*:

```

assumes S-finite:  $\tau \models \delta(S \rightarrow \text{size}_{\text{Set}}())$ 
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  $\rightarrow$  includingSet(a))  $\rightarrow$  iterateSet(a; x = A | F a x))  $\tau =$ 
  ((S  $\rightarrow$  excludingSet(a))  $\rightarrow$  iterateSet(a; x = F a A | F a x))  $\tau$ 
proof -
have insert-in-Setbase :  $\bigwedge \tau. (\tau \models (\delta S)) \implies (\tau \models (v a)) \implies$ 
   $\llcorner \text{insert} (a \tau) \ulcorner \text{Rep-Set}_{\text{base}} (S \tau) \ulcorner \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \ulcorner X \urcorner. x \neq \text{bot})\}$ 
by(frule Set-inv-lemma, simp add: foundation18 invalid-def)

```

```

have insert-defined :  $\bigwedge \tau. (\tau \models (\delta S)) \implies (\tau \models (v a)) \implies$ 
   $(\delta (\lambda -. \text{Abs-Set}_{\text{base}} \llcorner \text{insert} (a \tau) \ulcorner \text{Rep-Set}_{\text{base}} (S \tau) \ulcorner)) \tau = \text{true } \tau$ 
apply(subst defined-def)
apply(simp add: bot-Setbase-def bot-fun-def null-Setbase-def null-fun-def)
by(subst Abs-Setbase-inject,
  rule insert-in-Setbase, simp-all add: null-option-def bot-option-def)+

```

```

have remove-finite : finite  $\ulcorner \text{Rep-Set}_{\text{base}} (S \tau) \urcorner \implies$ 
  finite  $((\lambda a \tau. a) \ulcorner \ulcorner \text{Rep-Set}_{\text{base}} (S \tau) \urcorner \urcorner - \{a \tau\})$ 
by(simp)

```

```

have remove-in-Setbase :  $\bigwedge \tau. (\tau \models (\delta S)) \implies (\tau \models (v a)) \implies$ 
   $\llcorner \ulcorner \text{Rep-Set}_{\text{base}} (S \tau) \urcorner - \{a \tau\} \ulcorner \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \ulcorner X \urcorner. x \neq \text{bot})\}$ 
by(frule Set-inv-lemma, simp add: foundation18 invalid-def)

```

```

have remove-defined :  $\bigwedge \tau. (\tau \models (\delta S)) \implies (\tau \models (v a)) \implies$ 
   $(\delta (\lambda -. \text{Abs-Set}_{\text{base}} \llcorner \ulcorner \text{Rep-Set}_{\text{base}} (S \tau) \urcorner - \{a \tau\} \ulcorner)) \tau = \text{true } \tau$ 
apply(subst defined-def)
apply(simp add: bot-Setbase-def bot-fun-def null-Setbase-def null-fun-def)
by(subst Abs-Setbase-inject,
  rule remove-in-Setbase, simp-all add: null-option-def bot-option-def)+

```

```

have abs-rep:  $\bigwedge x. \llcorner x \ulcorner \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \ulcorner X \urcorner. x \neq \text{bot})\} \implies$ 
   $\ulcorner \text{Rep-Set}_{\text{base}} (\text{Abs-Set}_{\text{base}} \llcorner x \ulcorner) \urcorner = x$ 
by(subst Abs-Setbase-inverse, simp-all)

```

```

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

```

interpret *F-commute*: comp-fun-commute *F*

```

by (fact F-commute)
show ?thesis
apply(subst (1 2) cp-OclIterate, subst OclIncluding-def, subst OclExcluding-def)
apply(case-tac  $\neg ((\delta S) \tau = \text{true } \tau \wedge (v a) \tau = \text{true } \tau)$ , simp add: invalid-def)

```

```

apply(subgoal-tac OclIterate  $(\lambda -. \perp) A F \tau = \text{OclIterate } (\lambda -. \perp) (F a A) F \tau$ , simp)
apply(rule conjI, blast+)
apply(simp add: OclIterate-def defined-def bot-option-def bot-fun-def false-def true-def)

```

apply(*simp add: OclIterate-def*)
apply((*subst abs-rep[OF insert-in-Set_{base}[simplified OclValid-def], of τ], simp-all*)+,
(*subst abs-rep[OF remove-in-Set_{base}[simplified OclValid-def], of τ], simp-all*)+,
(*subst insert-defined, simp-all add: OclValid-def*)+,
(*subst remove-defined, simp-all add: OclValid-def*)+)

apply(*case-tac $\neg ((v A) \tau = true \tau)$, (simp add: F-valid-arg)*)+
apply(*rule impI,*
subst F-commute.fold-fun-left-comm[symmetric],
rule remove-finite, simp)

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

by(*subst F-commute.fold-insert-remove, simp*+)

qed

Execution Rules on Select

lemma *OclSelect-mtSet-exec[simp,code-unfold]: OclSelect mtSet P = mtSet*
apply(*rule ext, rename-tac τ*)
apply(*simp add: OclSelect-def mtSet-def defined-def false-def true-def*
bot-Set_{base}-def bot-fun-def null-Set_{base}-def null-fun-def)
by((*subst (1 2 3 4 5) Abs-Set_{base}-inverse*
| *subst Abs-Set_{base}-inject*), (*simp add: null-option-def bot-option-def*)+)+

definition *OclSelect-body :: $- \Rightarrow - \Rightarrow - \Rightarrow ('a, 'a \text{ option option}) \text{ Set}$*
 $\equiv (\lambda P x \text{ acc. if } P x \doteq \text{false then acc else acc} \rightarrow \text{including}_{\text{Set}}(x) \text{ endif})$

theorem *OclSelect-including-exec[simp,code-unfold]:*

assumes *P-cp : cp P*
shows *OclSelect (X \rightarrow including_{Set}(y)) P = OclSelect-body P y (OclSelect (X \rightarrow excluding_{Set}(y)) P)*
(is - = ?select)

proof –

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

have *ex-including : $\bigwedge f X y \tau. \tau \models \delta X \implies \tau \models v y \implies$*
 $(\exists x \in \ulcorner \text{Rep-Set}_{\text{base}} (X \rightarrow \text{including}_{\text{Set}}(y) \tau) \urcorner. f (P (\lambda-. x) \tau) =$
 $(f (P (\lambda-. y \tau)) \tau \vee (\exists x \in \ulcorner \text{Rep-Set}_{\text{base}} (X \tau) \urcorner. f (P (\lambda-. x) \tau)))$
apply(*simp add: OclIncluding-def OclValid-def*)
apply(*subst Abs-Set_{base}-inverse, simp, (rule disjI2)*)+
by (*metis (opaque-lifting, no-types) OclValid-def Set-inv-lemma foundation18', simp*)

have *al-including : $\bigwedge f X y \tau. \tau \models \delta X \implies \tau \models v y \implies$*
 $(\forall x \in \ulcorner \text{Rep-Set}_{\text{base}} (X \rightarrow \text{including}_{\text{Set}}(y) \tau) \urcorner. f (P (\lambda-. x) \tau) =$
 $(f (P (\lambda-. y \tau)) \tau \wedge (\forall x \in \ulcorner \text{Rep-Set}_{\text{base}} (X \tau) \urcorner. f (P (\lambda-. x) \tau)))$
apply(*simp add: OclIncluding-def OclValid-def*)
apply(*subst Abs-Set_{base}-inverse, simp, (rule disjI2)*)+
by (*metis (opaque-lifting, no-types) OclValid-def Set-inv-lemma foundation18', simp*)

have *ex-excluding1 : $\bigwedge f X y \tau. \tau \models \delta X \implies \tau \models v y \implies \neg (f (P (\lambda-. y \tau)) \tau) \implies$*
 $(\exists x \in \ulcorner \text{Rep-Set}_{\text{base}} (X \tau) \urcorner. f (P (\lambda-. x) \tau) =$
 $(\exists x \in \ulcorner \text{Rep-Set}_{\text{base}} (X \rightarrow \text{excluding}_{\text{Set}}(y) \tau) \urcorner. f (P (\lambda-. x) \tau))$
apply(*simp add: OclExcluding-def OclValid-def*)
apply(*subst Abs-Set_{base}-inverse, simp, (rule disjI2)*)+

by (*metis* (*no-types*) *Diff-iff OclValid-def Set-inv-lemma*) *auto*

have *al-excluding1* : $\bigwedge f X y \tau. \tau \models \delta X \implies \tau \models v y \implies f (P (\lambda \cdot. y \tau)) \tau \implies$
 $(\forall x \in {}^\top \text{Rep-Set}_{\text{base}} (X \tau)^\top. f (P (\lambda \cdot. x)) \tau) =$
 $(\forall x \in {}^\top \text{Rep-Set}_{\text{base}} (X \rightarrow \text{excluding}_{\text{Set}}(y) \tau)^\top. f (P (\lambda \cdot. x)) \tau)$
apply(*simp add: OclExcluding-def OclValid-def*)
apply(*subst Abs-Set_{base}-inverse, simp, (rule disjI2)+*)
by (*metis* (*no-types*) *Diff-iff OclValid-def Set-inv-lemma*) *auto*

have *in-including* : $\bigwedge f X y \tau. \tau \models \delta X \implies \tau \models v y \implies$
 $\{x \in {}^\top \text{Rep-Set}_{\text{base}} (X \rightarrow \text{including}_{\text{Set}}(y) \tau)^\top. f (P (\lambda \cdot. x) \tau)\} =$
 $(\text{let } s = \{x \in {}^\top \text{Rep-Set}_{\text{base}} (X \tau)^\top. f (P (\lambda \cdot. x) \tau)\} \text{ in}$
 $\text{if } f (P (\lambda \cdot. y \tau) \tau) \text{ then insert } (y \tau) s \text{ else } s)$
apply(*simp add: OclIncluding-def OclValid-def*)
apply(*subst Abs-Set_{base}-inverse, simp, (rule disjI2)+*)
apply (*metis* (*opaque-lifting, no-types*) *OclValid-def Set-inv-lemma foundation18'*)
by(*simp add: Let-def, auto*)

let *?OclSet* = $\lambda S. \sqcup S_{\perp} \in \{X. X = \perp \vee X = \text{null} \vee (\forall x \in {}^\top X^\top. x \neq \perp)\}$

have *diff-in-Set_{base}* : $\bigwedge \tau. (\delta X) \tau = \text{true} \tau \implies ?\text{OclSet } ({}^\top \text{Rep-Set}_{\text{base}} (X \tau)^\top - \{y \tau\})$
apply(*simp, (rule disjI2)+*)
by (*metis* (*mono-tags*) *Diff-iff OclValid-def Set-inv-lemma*)

have *ins-in-Set_{base}* : $\bigwedge \tau. (\delta X) \tau = \text{true} \tau \implies (v y) \tau = \text{true} \tau \implies$
 $?\text{OclSet } (\text{insert } (y \tau) \{x \in {}^\top \text{Rep-Set}_{\text{base}} (X \tau)^\top. P (\lambda \cdot. x) \tau \neq \text{false } \tau\})$
apply(*simp, (rule disjI2)+*)
by (*metis* (*opaque-lifting, no-types*) *OclValid-def Set-inv-lemma foundation18'*)

have *ins-in-Set_{base}'* : $\bigwedge \tau. (\delta X) \tau = \text{true} \tau \implies (v y) \tau = \text{true} \tau \implies$
 $?\text{OclSet } (\text{insert } (y \tau) \{x \in {}^\top \text{Rep-Set}_{\text{base}} (X \tau)^\top. x \neq y \tau \wedge P (\lambda \cdot. x) \tau \neq \text{false } \tau\})$
apply(*simp, (rule disjI2)+*)
by (*metis* (*opaque-lifting, no-types*) *OclValid-def Set-inv-lemma foundation18'*)

have *ins-in-Set_{base}''* : $\bigwedge \tau. (\delta X) \tau = \text{true} \tau \implies$
 $?\text{OclSet } \{x \in {}^\top \text{Rep-Set}_{\text{base}} (X \tau)^\top. P (\lambda \cdot. x) \tau \neq \text{false } \tau\}$
apply(*simp, (rule disjI2)+*)
by (*metis* (*opaque-lifting, no-types*) *OclValid-def Set-inv-lemma*)

have *ins-in-Set_{base}'''* : $\bigwedge \tau. (\delta X) \tau = \text{true} \tau \implies$
 $?\text{OclSet } \{x \in {}^\top \text{Rep-Set}_{\text{base}} (X \tau)^\top. x \neq y \tau \wedge P (\lambda \cdot. x) \tau \neq \text{false } \tau\}$
apply(*simp, (rule disjI2)+*)
by(*metis* (*opaque-lifting, no-types*) *OclValid-def Set-inv-lemma*)

have *if-same* : $\bigwedge a b c d \tau. \tau \models \delta a \implies b \tau = d \tau \implies c \tau = d \tau \implies$
 $(\text{if } a \text{ then } b \text{ else } c \text{ endif}) \tau = d \tau$
by(*simp add: OclIf-def OclValid-def*)

have *invert-including* : $\bigwedge P y \tau. P \tau = \perp \implies P \rightarrow \text{including}_{\text{Set}}(y) \tau = \perp$
by (*metis* (*opaque-lifting, no-types*) *foundation16[THEN iffD1]*
foundation18' OclIncluding-valid-args-valid)

have *exclude-defined* : $\bigwedge \tau. \tau \models \delta X \implies$
 $(\delta(\lambda \cdot. \text{Abs-Set}_{\text{base}} \sqcup \{x \in {}^\top \text{Rep-Set}_{\text{base}} (X \tau)^\top. x \neq y \tau \wedge P (\lambda \cdot. x) \tau \neq \text{false } \tau\}_{\perp})) \tau = \text{true } \tau$
apply(*subst defined-def,*
simp add: false-def true-def bot-Set_{base}-def bot-fun-def null-Set_{base}-def null-fun-def)
by(*subst Abs-Set_{base}-inject[OF ins-in-Set_{base}''''[simplified false-def]],*

(simp add: OclValid-def bot-option-def null-option-def)+

have *if-eq* : $\bigwedge x A B \tau. \tau \models v x \implies \tau \models ((\text{if } x \doteq \text{false then } A \text{ else } B \text{ endif}) \triangleq (\text{if } x \triangleq \text{false then } A \text{ else } B \text{ endif}))$

apply(simp add: StrictRefEq_{Boolean} OclValid-def)
apply(subst (2) StrongEq-def)
by(subst cp-OclIf, simp add: cp-OclIf[symmetric] true-def)

have *OclSelect-body-bot*: $\bigwedge \tau. \tau \models \delta X \implies \tau \models v y \implies P y \tau \neq \perp \implies (\exists x \in {}^\Gamma \text{Rep-Set}_{\text{base}} (X \tau)^\Gamma. P (\lambda \cdot x) \tau = \perp) \implies \perp = ?\text{select } \tau$

apply(drule ex-excluding1[**where** $X2 = X$ **and** $y2 = y$ **and** $f2 = \lambda x \tau. x \tau = \perp$],
(simp add: P-cp[symmetric])+)
apply(subgoal-tac $\tau \models (\perp \triangleq ?\text{select})$, simp add: OclValid-def StrongEq-def true-def bot-fun-def)
apply(simp add: OclSelect-body-def)
apply(subst StrongEq-L-subst3[OF - if-eq], simp, metis foundation18')
apply(simp add: OclValid-def, subst StrongEq-def, subst true-def, simp)
apply(subgoal-tac $\exists x \in {}^\Gamma \text{Rep-Set}_{\text{base}} (X \rightarrow \text{excluding}_{\text{Set}}(y) \tau)^\Gamma. P (\lambda \cdot x) \tau = \perp \tau$)
prefer 2 **apply** (metis bot-fun-def)
apply(subst if-same[**where** $d5 = \perp$])
apply (metis defined7 transform1)
apply(simp add: OclSelect-def bot-option-def bot-fun-def invalid-def)
apply(subst invert-including)
by(simp add: OclSelect-def bot-option-def bot-fun-def invalid-def)+

have *d-and-v-inject* : $\bigwedge \tau X y. (\delta X \text{ and } v y) \tau \neq \text{true } \tau \implies (\delta X \text{ and } v y) \tau = \text{false } \tau$

apply(fold OclValid-def, subst foundation22[symmetric])
apply(auto simp: foundation10' defined-split)
apply(erule StrongEq-L-subst2-rev, simp, simp)
apply(erule StrongEq-L-subst2-rev, simp, simp)
by(erule foundation7'[THEN iffD2, THEN foundation15[THEN iffD2, THEN StrongEq-L-subst2-rev]], simp, simp)

have *OclSelect-body-bot'*: $\bigwedge \tau. (\delta X \text{ and } v y) \tau \neq \text{true } \tau \implies \perp = ?\text{select } \tau$

apply(drule d-and-v-inject)
apply(simp add: OclSelect-def OclSelect-body-def)
apply(subst cp-OclIf, subst OclIncluding.cp0, simp add: false-def true-def)
apply(subst cp-OclIf[symmetric], subst OclIncluding.cp0[symmetric])
by (metis (lifting, no-types) OclIf-def foundation18 foundation18' invert-including)

have *conj-split2* : $\bigwedge a b c \tau. ((a \triangleq \text{false}) \tau = \text{false } \tau \longrightarrow b) \wedge ((a \triangleq \text{false}) \tau = \text{true } \tau \longrightarrow c) \implies (a \tau \neq \text{false } \tau \longrightarrow b) \wedge (a \tau = \text{false } \tau \longrightarrow c)$

by (metis OclValid-def defined7 foundation14 foundation22 foundation9)

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 *cp-OclSelect-body* : $\bigwedge \tau. ?\text{select } \tau = \text{OclSelect-body } P y (\lambda \cdot (\text{OclSelect } (X \rightarrow \text{excluding}_{\text{Set}}(y)) P) \tau) \tau$

apply(simp add: OclSelect-body-def)
by(subst (1 2) cp-OclIf, subst (1 2) OclIncluding.cp0, blast)

have *OclSelect-body-strict1* : *OclSelect-body* $P y \text{ invalid} = \text{invalid}$

```

by(rule ext, simp add: OclSelect-body-def OclIf-def)

have bool-invalid:  $\bigwedge(x:(\mathbb{A})\text{Boolean}) y \tau. \neg (\tau \models v x) \implies \tau \models ((x \doteq y) \triangleq \text{invalid})$ 
  by(simp add: StrictRefEqBoolean OclValid-def StrongEq-def true-def)

have conj-comm :  $\bigwedge p q r. (p \wedge q \wedge r) = ((p \wedge q) \wedge r)$  by blast

have inv-bot :  $\bigwedge \tau. \text{invalid } \tau = \perp \tau$  by (metis bot-fun-def invalid-def)
have inv-bot' :  $\bigwedge \tau. \text{invalid } \tau = \perp$  by (simp add: invalid-def)

show ?thesis
  apply(rule ext, rename-tac  $\tau$ )
  apply(subst OclSelect-def)
  apply(case-tac ( $\delta (X \rightarrow \text{includingSet}(y))$ )  $\tau = \text{true } \tau$ , simp)
  apply((subst ex-including | subst in-including),
    metis OclValid-def foundation5,
    metis OclValid-def foundation5)+
  apply(simp add: Let-def inv-bot)
  apply(subst (2 4 7 9) bot-fun-def)

  apply(subst (4) false-def, subst (4) bot-fun-def, simp add: bot-option-def P-cp[symmetric])

  apply(case-tac  $\neg (\tau \models (v P y))$ )
  apply(subgoal-tac  $P y \tau \neq \text{false } \tau$ )
  prefer 2
  apply (metis (opaque-lifting, no-types) foundation1 foundation18' valid4)
  apply(simp)

  apply(subst conj-comm, rule conjI)
  apply(drule-tac  $y1 = \text{false}$  in bool-invalid)
  apply(simp only: OclSelect-body-def,
    metis OclIf-def OclValid-def defined-def foundation2 foundation22
    bot-fun-def invalid-def)

  apply(drule foundation5[simplified OclValid-def],
    subst al-including[simplified OclValid-def],
    simp,
    simp)
  apply(simp add: P-cp[symmetric])
  apply (metis bot-fun-def foundation18')

  apply(simp add: foundation18' bot-fun-def OclSelect-body-bot OclSelect-body-bot')

  apply(subst (1 2) al-including, metis OclValid-def foundation5, metis OclValid-def foundation5)
  apply(simp add: P-cp[symmetric], subst (4) false-def, subst (4) bot-option-def, simp)

  apply(simp add: OclSelect-def[simplified inv-bot] OclSelect-body-def StrictRefEqBoolean)
  apply(subst (1 2 3 4) cp-OclIf,
    subst (1 2 3 4) foundation18'[THEN iffD2, simplified OclValid-def],
    simp,
    simp only: cp-OclIf[symmetric] refl if-True)
  apply(subst (1 2) OclIncluding.cp0, rule conj-split2, simp add: cp-OclIf[symmetric])
  apply(subst (1 2 3 4 5 6 7 8) cp-OclIf[symmetric], simp)
  apply((subst ex-excluding1[symmetric]
    | subst al-excluding1[symmetric] ),
    metis OclValid-def foundation5,
    metis OclValid-def foundation5,

```

```

    simp add: P-cp[symmetric] bot-fun-def)+
  apply(simp add: bot-fun-def)
  apply(subst (1 2) invert-including, simp+)

  apply(rule conjI, blast)
  apply(intro impI conjI)
  apply(subst OclExcluding-def)
  apply(drule foundation5[simplified OclValid-def], simp)
  apply(subst Abs-Setbase-inverse[OF diff-in-Setbase], fast)
  apply(simp add: OclIncluding-def cp-valid[symmetric])
  apply((erule conjE)+, frule exclude-defined[simplified OclValid-def], simp)
  apply(subst Abs-Setbase-inverse[OF ins-in-Setbase''], simp+)
  apply(subst Abs-Setbase-inject[OF ins-in-Setbase ins-in-Setbase'], fast+)

  apply(simp add: OclExcluding-def)
  apply(simp add: foundation10[simplified OclValid-def])
  apply(subst Abs-Setbase-inverse[OF diff-in-Setbase], simp+)
  apply(subst Abs-Setbase-inject[OF ins-in-Setbase'' ins-in-Setbase''], simp+)
  apply(subgoal-tac P (λ-. y τ) τ = false τ)
  prefer 2
  apply(subst P-cp[symmetric], metis OclValid-def foundation22)
  apply(rule equalityI)
  apply(rule subsetI, simp, metis)
  apply(rule subsetI, simp)

  apply(drule defined-inject-true)
  apply(subgoal-tac ¬ (τ ≡ δ X) ∨ ¬ (τ ≡ v y))
  prefer 2
  apply (metis OclIncluding.def-homo OclIncluding-valid-args-valid OclIncluding-valid-args-valid'' OclValid-def
  foundation18 valid1)
  apply(subst cp-OclSelect-body, subst cp-OclSelect, subst OclExcluding-def)
  apply(simp add: OclValid-def false-def true-def, rule conjI, blast)
  apply(simp add: OclSelect-invalid[simplified invalid-def]
    OclSelect-body-strict1[simplified invalid-def]
    inv-bot')

done
qed

```

Execution Rules on Reject

lemma *OclReject-mtSet-exec*[simp,code-unfold]: *OclReject* mtSet $P = \text{mtSet}$
 by(simp add: *OclReject-def*)

lemma *OclReject-including-exec*[simp,code-unfold]:
 assumes $P\text{-cp} : \text{cp } P$
 shows *OclReject* ($X \rightarrow \text{including}_{\text{Set}}(y)$) $P = \text{OclSelect-body}$ ($\text{not } o P$) y (*OclReject* ($X \rightarrow \text{excluding}_{\text{Set}}(y)$) P)
 apply(simp add: *OclReject-def* comp-def, rule *OclSelect-including-exec*)
 by (metis *assms cp-intro*'(5))

Execution Rules Combining Previous Operators

OclIncluding

lemma *OclIncluding-idem0* :
 assumes $\tau \models \delta S$
 and $\tau \models v i$
 shows $\tau \models (S \rightarrow \text{including}_{\text{Set}}(i) \rightarrow \text{including}_{\text{Set}}(i) \triangleq (S \rightarrow \text{including}_{\text{Set}}(i)))$
 by(simp add: *OclIncluding-includes* *OclIncludes-charn1* *assms*)

theorem *OclIncluding-idem*[simp,code-unfold]: $((S :: (\mathbb{A}, 'a::\text{null})\text{Set}) \rightarrow \text{including}_{\text{Set}}(i) \rightarrow \text{including}_{\text{Set}}(i)) = (S \rightarrow \text{including}_{\text{Set}}(i))$

proof –

have *A*: $\bigwedge \tau. \tau \models (i \triangleq \text{invalid}) \implies (S \rightarrow \text{including}_{\text{Set}}(i) \rightarrow \text{including}_{\text{Set}}(i)) \tau = \text{invalid } \tau$
apply(rule foundation22[THEN iffD1])

by(erule StrongEq-L-subst2-rev, simp,simp)

have *A'*: $\bigwedge \tau. \tau \models (i \triangleq \text{invalid}) \implies (S \rightarrow \text{including}_{\text{Set}}(i)) \tau = \text{invalid } \tau$

apply(rule foundation22[THEN iffD1])

by(erule StrongEq-L-subst2-rev, simp,simp)

have *C*: $\bigwedge \tau. \tau \models (S \triangleq \text{invalid}) \implies (S \rightarrow \text{including}_{\text{Set}}(i) \rightarrow \text{including}_{\text{Set}}(i)) \tau = \text{invalid } \tau$

apply(rule foundation22[THEN iffD1])

by(erule StrongEq-L-subst2-rev, simp,simp)

have *C'*: $\bigwedge \tau. \tau \models (S \triangleq \text{invalid}) \implies (S \rightarrow \text{including}_{\text{Set}}(i)) \tau = \text{invalid } \tau$

apply(rule foundation22[THEN iffD1])

by(erule StrongEq-L-subst2-rev, simp,simp)

have *D*: $\bigwedge \tau. \tau \models (S \triangleq \text{null}) \implies (S \rightarrow \text{including}_{\text{Set}}(i) \rightarrow \text{including}_{\text{Set}}(i)) \tau = \text{invalid } \tau$

apply(rule foundation22[THEN iffD1])

by(erule StrongEq-L-subst2-rev, simp,simp)

have *D'*: $\bigwedge \tau. \tau \models (S \triangleq \text{null}) \implies (S \rightarrow \text{including}_{\text{Set}}(i)) \tau = \text{invalid } \tau$

apply(rule foundation22[THEN iffD1])

by(erule StrongEq-L-subst2-rev, simp,simp)

show ?thesis

apply(rule ext, rename-tac τ)

apply(case-tac $\tau \models (v \ i)$)

apply(case-tac $\tau \models (\delta \ S)$)

apply(simp only: OclIncluding-idem0[THEN foundation22[THEN iffD1]])

apply(simp add: foundation16', elim disjE)

apply(simp add: C[OF foundation22[THEN iffD2]] C'[OF foundation22[THEN iffD2]])

apply(simp add: D[OF foundation22[THEN iffD2]] D'[OF foundation22[THEN iffD2]])

apply(simp add: foundation18 A[OF foundation22[THEN iffD2]] A'[OF foundation22[THEN iffD2]])

done

qed

OclExcluding

lemma *OclExcluding-idem0* :

assumes $\tau \models \delta \ S$

and $\tau \models v \ i$

shows $\tau \models (S \rightarrow \text{excluding}_{\text{Set}}(i) \rightarrow \text{excluding}_{\text{Set}}(i)) \triangleq (S \rightarrow \text{excluding}_{\text{Set}}(i))$

by(simp add: OclExcluding-excludes OclExcludes-charn1 assms)

theorem *OclExcluding-idem*[simp,code-unfold]: $((S \rightarrow \text{excluding}_{\text{Set}}(i) \rightarrow \text{excluding}_{\text{Set}}(i)) = (S \rightarrow \text{excluding}_{\text{Set}}(i)))$

proof –

have *A*: $\bigwedge \tau. \tau \models (i \triangleq \text{invalid}) \implies (S \rightarrow \text{excluding}_{\text{Set}}(i) \rightarrow \text{excluding}_{\text{Set}}(i)) \tau = \text{invalid } \tau$
apply(rule foundation22[THEN iffD1])

by(erule StrongEq-L-subst2-rev, simp,simp)

have *A'*: $\bigwedge \tau. \tau \models (i \triangleq \text{invalid}) \implies (S \rightarrow \text{excluding}_{\text{Set}}(i)) \tau = \text{invalid } \tau$

apply(rule foundation22[THEN iffD1])

by(erule StrongEq-L-subst2-rev, simp,simp)

have *C*: $\bigwedge \tau. \tau \models (S \triangleq \text{invalid}) \implies (S \rightarrow \text{excluding}_{\text{Set}}(i) \rightarrow \text{excluding}_{\text{Set}}(i)) \tau = \text{invalid } \tau$

apply(rule foundation22[THEN iffD1])

by(erule StrongEq-L-subst2-rev, simp,simp)

have *C'*: $\bigwedge \tau. \tau \models (S \triangleq \text{invalid}) \implies (S \rightarrow \text{excluding}_{\text{Set}}(i)) \tau = \text{invalid } \tau$

apply(rule foundation22[THEN iffD1])

```

    by(erule StrongEq-L-subst2-rev, simp,simp)
  have D:  $\bigwedge \tau. \tau \models (S \triangleq \text{null}) \implies (S \rightarrow \text{excluding}_{\text{Set}}(i) \rightarrow \text{excluding}_{\text{Set}}(i)) \tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
  have D':  $\bigwedge \tau. \tau \models (S \triangleq \text{null}) \implies (S \rightarrow \text{excluding}_{\text{Set}}(i)) \tau = \text{invalid } \tau$ 
    apply(rule foundation22[THEN iffD1])
    by(erule StrongEq-L-subst2-rev, simp,simp)
  show ?thesis
    apply(rule ext, rename-tac  $\tau$ )
    apply(case-tac  $\tau \models (v \ i)$ )
    apply(case-tac  $\tau \models (\delta \ S)$ )
    apply(simp only: OclExcluding-idem0[THEN foundation22[THEN iffD1]])
    apply(simp add: foundation16', elim disjE)
    apply(simp add: C[OF foundation22[THEN iffD2]] C'[OF foundation22[THEN iffD2]])
    apply(simp add: D[OF foundation22[THEN iffD2]] D'[OF foundation22[THEN iffD2]])
    apply(simp add: foundation18 A[OF foundation22[THEN iffD2]] A'[OF foundation22[THEN iffD2]])
  done
qed

```

OclIncludes

lemma OclIncludes-any[simp,code-unfold]:

$$\begin{aligned}
 X \rightarrow \text{includes}_{\text{Set}}(X \rightarrow \text{any}_{\text{Set}}()) &= (\text{if } \delta \ X \ \text{then} \\
 &\quad \text{if } \delta \ (X \rightarrow \text{size}_{\text{Set}}()) \ \text{then } \text{not}(X \rightarrow \text{isEmpty}_{\text{Set}}()) \\
 &\quad \text{else } X \rightarrow \text{includes}_{\text{Set}}(\text{null}) \ \text{endif} \\
 &\text{else } \text{invalid} \ \text{endif})
 \end{aligned}$$

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 notempty':  $\bigwedge \tau \ X. \tau \models \delta \ X \implies \text{finite } \ulcorner \text{Rep-Set}_{\text{base}}(X \ \tau) \urcorner \implies \text{not}(X \rightarrow \text{isEmpty}_{\text{Set}}()) \tau \neq \text{true } \tau$ 
   $\implies$ 

```

```

     $X \ \tau = \text{Set}\{\} \ \tau$ 
  apply(case-tac  $X \ \tau$ , rename-tac  $X'$ , simp add: mtSet-def Abs-Setbase-inject)
  apply(erule disjE, metis (opaque-lifting, mono-tags) bot-Setbase-def bot-option-def foundation16)
  apply(erule disjE, metis (opaque-lifting, no-types) bot-option-def
    null-Setbase-def null-option-def foundation16[THEN iffD1])
  apply(case-tac  $X'$ , simp, metis (opaque-lifting, no-types) bot-Setbase-def foundation16[THEN iffD1])
  apply(rename-tac  $X''$ , case-tac  $X''$ , simp)
  apply (metis (opaque-lifting, no-types) foundation16[THEN iffD1] null-Setbase-def)
  apply(simp add: OclIsEmpty-def OclSize-def)
  apply(subst (asm) cp-OclNot, subst (asm) cp-OclOr, subst (asm) StrictRefEqInteger.cp0,
    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-Setbase-inverse split: if-split-asm)
  by(simp add: true-def OclInt0-def OclNot-def StrictRefEqInteger StrongEq-def)

```

```

  have B:  $\bigwedge X \ \tau. \neg \text{finite } \ulcorner \text{Rep-Set}_{\text{base}}(X \ \tau) \urcorner \implies (\delta \ (X \rightarrow \text{size}_{\text{Set}}())) \tau = \text{false } \tau$ 

```

```

apply(subst cp-defined)
apply(simp add: OclSize-def)
by (metis bot-fun-def defined-def)

show ?thesis
apply(rule ext, rename-tac  $\tau$ , simp only: OclIncludes-def OclANY-def)
apply(subst cp-OclIf, subst (2) cp-valid)
apply(case-tac ( $\delta X$ )  $\tau = \text{true } \tau$ ,
  simp only: foundation20[simplified OclValid-def] cp-OclIf[symmetric], simp,
  subst (1 2) cp-OclAnd, simp add: cp-OclAnd[symmetric])
apply(case-tac finite  $\ulcorner \text{Rep-Set}_{\text{base}}(X \tau) \urcorner$ )
apply(frule size-defined'[THEN iffD2, simplified OclValid-def], assumption)
apply(subst (1 2 3 4) cp-OclIf, simp)
apply(subst (1 2 3 4) cp-OclIf[symmetric], simp)
apply(case-tac ( $X \rightarrow \text{notEmpty}_{\text{Set}}()$ )  $\tau = \text{true } \tau$ , simp)
apply(frule OclNotEmpty-has-elt[simplified OclValid-def], simp)
apply(simp add: OclNotEmpty-def cp-OclIf[symmetric])
apply(subgoal-tac (SOME  $y. y \in \ulcorner \text{Rep-Set}_{\text{base}}(X \tau) \urcorner$ )  $\in \ulcorner \text{Rep-Set}_{\text{base}}(X \tau) \urcorner$ , 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  $\neg (\text{null } \tau \in \ulcorner \text{Rep-Set}_{\text{base}}(X \tau) \urcorner)$ , simp)
apply(subst OclIsEmpty-def, simp add: OclSize-def)
apply(subst cp-OclNot, subst cp-OclOr, subst StrictRefEqInteger.cp0, 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-Setbase-inverse OclInt0-def false-def)+)
apply(drule notempty'[simplified OclValid-def], simp, simp)
apply (metis (opaque-lifting, no-types) empty-iff mtSet-rep-set)

apply(frule B)
apply(subst (1 2 3 4) cp-OclIf, simp)
apply(subst (1 2 3 4) cp-OclIf[symmetric], simp)
apply(case-tac ( $X \rightarrow \text{notEmpty}_{\text{Set}}()$ )  $\tau = \text{true } \tau$ , simp)
apply(frule OclNotEmpty-has-elt[simplified OclValid-def], simp)
apply(simp add: OclNotEmpty-def OclIsEmpty-def)
apply(subgoal-tac  $X \rightarrow \text{size}_{\text{Set}}()$   $\tau = \perp$ )
prefer 2
apply (metis (opaque-lifting, no-types) OclSize-def)
apply(subst (asm) cp-OclNot, subst (asm) cp-OclOr, subst (asm) StrictRefEqInteger.cp0,
  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 false-def true-def
  bot-option-def bot-fun-def invalid-def)

apply (metis 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

OclSize

lemma [simp,code-unfold]:  $\delta (\text{Set}\{\} \rightarrow \text{size}_{\text{Set}}()) = \text{true}$ 
by simp

```

lemma [simp,code-unfold]: $\delta ((X \rightarrow \text{including}_{Set}(x)) \rightarrow \text{size}_{Set}()) = (\delta(X \rightarrow \text{size}_{Set}()))$ 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 *OclIncluding-finite-rep-set* : $\bigwedge \tau. (\delta X \text{ and } v x) \tau = \text{true } \tau \implies$
 $\text{finite } \ulcorner \text{Rep-Set}_{base} (X \rightarrow \text{including}_{Set}(x) \tau) \urcorner = \text{finite } \ulcorner \text{Rep-Set}_{base} (X \tau) \urcorner$
apply(rule *OclIncluding-finite-rep-set*)
by(metis *OclValid-def foundation5*)+

have *card-including-exec* : $\bigwedge \tau. (\delta (\lambda \cdot \ulcorner \text{int } (\text{card } \ulcorner \text{Rep-Set}_{base} (X \rightarrow \text{including}_{Set}(x) \tau) \urcorner) \urcorner)) \tau =$
 $(\delta (\lambda \cdot \ulcorner \text{int } (\text{card } \ulcorner \text{Rep-Set}_{base} (X \tau) \urcorner) \urcorner)) \tau$
by(simp add: *defined-def bot-fun-def bot-option-def null-fun-def null-option-def*)

show ?thesis

apply(rule *ext*, *rename-tac* τ)
apply(*case-tac* $(\delta (X \rightarrow \text{including}_{Set}(x) \rightarrow \text{size}_{Set}())) \tau = \text{true } \tau$, *simp del: OclSize-including-exec*)
apply(*subst cp-OclAnd*, *subst cp-defined*, *simp only: cp-defined[of X → including_{Set}(x) → size_{Set}()]*,
simp add: OclSize-def)
apply(*case-tac* $((\delta X \text{ and } v x) \tau = \text{true } \tau \wedge \text{finite } \ulcorner \text{Rep-Set}_{base} (X \rightarrow \text{including}_{Set}(x) \tau) \urcorner)$, *simp*)
apply(*erule conjE*,
simp add: OclIncluding-finite-rep-set[simplified OclValid-def] card-including-exec
cp-OclAnd[of δ X v x]
cp-OclAnd[of true, THEN sym])
apply(*subgoal-tac* $(\delta X) \tau = \text{true } \tau \wedge (v x) \tau = \text{true } \tau$, *simp*)
apply(rule *foundation5*[*of - δ X v x*, *simplified OclValid-def*],
simp only: cp-OclAnd[THEN sym])
apply(*simp*, *simp add: defined-def true-def false-def bot-fun-def bot-option-def*)

apply(*drule defined-inject-true*[*of X → including_{Set}(x) → size_{Set}()*],
simp del: OclSize-including-exec,
simp only: cp-OclAnd[of δ (X → size_{Set}()) v x],
simp add: cp-defined[of X → including_{Set}(x) → size_{Set}()] cp-defined[of X → size_{Set}()]
del: OclSize-including-exec,
simp add: OclSize-def card-including-exec
del: OclSize-including-exec)
apply(*case-tac* $(\delta X \text{ and } v x) \tau = \text{true } \tau \wedge \text{finite } \ulcorner \text{Rep-Set}_{base} (X \tau) \urcorner$,
simp add: OclIncluding-finite-rep-set[simplified OclValid-def] card-including-exec,
simp only: cp-OclAnd[THEN sym],
simp add: defined-def bot-fun-def)

apply(*split if-split-asm*)
apply(*simp add: OclIncluding-finite-rep-set[simplified OclValid-def] card-including-exec*)
apply(*simp only: cp-OclAnd[THEN sym]*, *simp*, *rule impI*, *erule conjE*)
apply(*case-tac* $(v x) \tau = \text{true } \tau$, *simp add: cp-OclAnd[of δ X v x]*)
by(*drule valid-inject-true*[*of x*], *simp add: cp-OclAnd*[*of - v x*])
qed

lemma [simp,code-unfold]: $\delta ((X \rightarrow \text{excluding}_{Set}(x)) \rightarrow \text{size}_{Set}()) = (\delta(X \rightarrow \text{size}_{Set}()))$ 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 *OclExcluding-finite-rep-set* : $\bigwedge \tau. (\delta X \text{ and } v x) \tau = \text{true } \tau \implies$
 $\text{finite } \ulcorner \text{Rep-Set}_{\text{base}} (X \rightarrow \text{excluding}_{\text{Set}}(x) \tau) \urcorner =$
 $\text{finite } \ulcorner \text{Rep-Set}_{\text{base}} (X \tau) \urcorner$
apply(*rule OclExcluding-finite-rep-set*)
by(*metis OclValid-def foundation5*)+

have *card-excluding-exec* : $\bigwedge \tau. (\delta (\lambda \cdot \ulcorner \text{int } (\text{card } \ulcorner \text{Rep-Set}_{\text{base}} (X \rightarrow \text{excluding}_{\text{Set}}(x) \tau) \urcorner) \urcorner)) \tau =$
 $(\delta (\lambda \cdot \ulcorner \text{int } (\text{card } \ulcorner \text{Rep-Set}_{\text{base}} (X \tau) \urcorner) \urcorner)) \tau$
by(*simp add: defined-def bot-fun-def bot-option-def null-fun-def null-option-def*)

show *?thesis*
apply(*rule ext, rename-tac τ*)
apply(*case-tac $(\delta (X \rightarrow \text{excluding}_{\text{Set}}(x) \rightarrow \text{size}_{\text{Set}}())) \tau = \text{true } \tau$, simp*)
apply(*subst cp-OclAnd, subst cp-defined, simp only: cp-defined[of $X \rightarrow \text{excluding}_{\text{Set}}(x) \rightarrow \text{size}_{\text{Set}}()$], simp add: OclSize-def*)
apply(*case-tac $(\delta X \text{ and } v x) \tau = \text{true } \tau \wedge \text{finite } \ulcorner \text{Rep-Set}_{\text{base}} (X \rightarrow \text{excluding}_{\text{Set}}(x) \tau) \urcorner$, simp*)
apply(*erule conjE,*
simp add: OclExcluding-finite-rep-set[simplified OclValid-def] card-excluding-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(*simp, simp add: defined-def true-def false-def bot-fun-def bot-option-def*)

apply(*drule defined-inject-true[of $X \rightarrow \text{excluding}_{\text{Set}}(x) \rightarrow \text{size}_{\text{Set}}()$],*
simp,
simp only: cp-OclAnd[of $\delta (X \rightarrow \text{size}_{\text{Set}}()) v x$,
simp add: cp-defined[of $X \rightarrow \text{excluding}_{\text{Set}}(x) \rightarrow \text{size}_{\text{Set}}()$] cp-defined[of $X \rightarrow \text{size}_{\text{Set}}()$],
simp add: OclSize-def card-excluding-exec)
apply(*case-tac $(\delta X \text{ and } v x) \tau = \text{true } \tau \wedge \text{finite } \ulcorner \text{Rep-Set}_{\text{base}} (X \tau) \urcorner$,*
simp add: OclExcluding-finite-rep-set[simplified OclValid-def] card-excluding-exec,
simp only: cp-OclAnd[THEN sym],
simp add: defined-def bot-fun-def)

apply(*split if-split-asm*)
apply(*simp add: OclExcluding-finite-rep-set[simplified OclValid-def] card-excluding-exec*)+
apply(*simp only: cp-OclAnd[THEN sym], simp, rule impI, erule conjE*)
apply(*case-tac $(v x) \tau = \text{true } \tau$, simp add: cp-OclAnd[of $\delta X v x$])*)
by(*drule valid-inject-true[of x], simp add: cp-OclAnd[of - $v x$])*)

qed

lemma [*simp*]:
assumes *X-finite*: $\bigwedge \tau. \text{finite } \ulcorner \text{Rep-Set}_{\text{base}} (X \tau) \urcorner$
shows $\delta ((X \rightarrow \text{including}_{\text{Set}}(x)) \rightarrow \text{size}_{\text{Set}}()) = (\delta(X) \text{ and } v(x))$
by(*simp add: size-defined[OF X-finite] del: OclSize-including-exec*)

OclForall

lemma *OclForall-rep-set-false*:

assumes $\tau \models \delta X$
shows $(\text{OclForall } X P \tau = \text{false } \tau) = (\exists x \in {}^{\top}\text{Rep-Set}_{\text{base}}(X \tau)^{\top}. P(\lambda\tau. x) \tau = \text{false } \tau)$
by(*insert assms, simp add: OclForall-def OclValid-def false-def true-def invalid-def bot-fun-def bot-option-def null-fun-def null-option-def*)

lemma *OclForall-rep-set-true*:

assumes $\tau \models \delta X$
shows $(\tau \models \text{OclForall } X P) = (\forall x \in {}^{\top}\text{Rep-Set}_{\text{base}}(X \tau)^{\top}. \tau \models P(\lambda\tau. x))$

proof –

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(*rename-tac x', case-tac x'*) **apply** (*metis null-Boolean-def*)
apply(*rename-tac x'', case-tac x''*) **apply** (*metis (full-types) true-def*)
by (*metis (full-types) false-def*)

have *disjE4* : $\bigwedge P1 P2 P3 P4 R.$

$(P1 \vee P2 \vee P3 \vee P4) \implies (P1 \implies R) \implies (P2 \implies R) \implies (P3 \implies R) \implies (P4 \implies R) \implies R$

by *metis*

show *?thesis*

apply(*simp add: OclForall-def OclValid-def true-def false-def invalid-def bot-fun-def bot-option-def null-fun-def null-option-def split: if-split-asm*)
apply(*rule conjI, rule impI*) **apply** (*metis drop.simps option.distinct(1) invalid-def*)
apply(*rule impI, rule conjI, rule impI*) **apply** (*metis option.distinct(1)*)
apply(*rule impI, rule conjI, rule impI*) **apply** (*metis drop.simps*)
apply(*intro conjI impI ballI*)

proof – **fix** x **show** $\forall x \in {}^{\top}\text{Rep-Set}_{\text{base}}(X \tau)^{\top}. P(\lambda\tau. x) \tau \neq \perp \text{None}_{\perp} \implies$
 $\forall x \in {}^{\top}\text{Rep-Set}_{\text{base}}(X \tau)^{\top}. \exists y. P(\lambda\tau. x) \tau = \perp y_{\perp} \implies$
 $\forall x \in {}^{\top}\text{Rep-Set}_{\text{base}}(X \tau)^{\top}. P(\lambda\tau. x) \tau \neq \perp \text{False}_{\perp} \implies$
 $x \in {}^{\top}\text{Rep-Set}_{\text{base}}(X \tau)^{\top} \implies P(\lambda\tau. x) \tau = \perp \text{True}_{\perp}$

apply(*erule-tac x = x in ballE*)+

by(*rule disjE4[OF destruct-ocl[of P(\lambda\tau. x) \tau]]*,

(simp add: true-def false-def null-fun-def null-option-def bot-fun-def bot-option-def)+

qed(*simp add: assms[simplified OclValid-def true-def]*)+

qed

lemma *OclForall-includes* :

assumes $x\text{-def} : \tau \models \delta x$

and $y\text{-def} : \tau \models \delta y$

shows $(\tau \models \text{OclForall } x (\text{OclIncludes } y)) = ({}^{\top}\text{Rep-Set}_{\text{base}}(x \tau)^{\top} \subseteq {}^{\top}\text{Rep-Set}_{\text{base}}(y \tau)^{\top})$

apply(*simp add: OclForall-rep-set-true[OF x-def]*,

simp add: OclIncludes-def OclValid-def y-def[simplified OclValid-def])

apply(*insert Set-inv-lemma[OF x-def], simp add: valid-def false-def true-def bot-fun-def*)

by(*rule iffI, simp add: subsetI, simp add: subsetD*)

lemma *OclForall-not-includes* :

assumes $x\text{-def} : \tau \models \delta x$

and $y\text{-def} : \tau \models \delta y$

shows $(\text{OclForall } x (\text{OclIncludes } y) \tau = \text{false } \tau) = (\neg ({}^{\top}\text{Rep-Set}_{\text{base}}(x \tau)^{\top} \subseteq {}^{\top}\text{Rep-Set}_{\text{base}}(y \tau)^{\top}))$

apply(*simp add: OclForall-rep-set-false[OF x-def]*,

simp add: OclIncludes-def OclValid-def y-def[simplified OclValid-def])

apply(*insert Set-inv-lemma[OF x-def], simp add: valid-def false-def true-def bot-fun-def*)

by(*rule iffI, metis rev-subsetD, metis subsetI*)

lemma *OclForall-iterate*:

assumes $S\text{-finite} : \text{finite } {}^{\top}\text{Rep-Set}_{\text{base}}(S \tau)^{\top}$

shows $S \rightarrow \text{forAll}_{\text{Set}}(x \mid P x) \tau = (S \rightarrow \text{iterate}_{\text{Set}}(x; \text{acc} = \text{true} \mid \text{acc and } P x)) \tau$

```

proof –
interpret 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(rename-tac x', case-tac x') apply (metis null-Boolean-def)
apply(rename-tac x'', case-tac x'') apply (metis (full-types) true-def)
by (metis (full-types) false-def)

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

let ?P-eq =  $\lambda x b \tau. P (\lambda \cdot. x) \tau = b \tau$ 
let ?P =  $\lambda \text{set } b \tau. \exists x \in \text{set}. ?P\text{-eq } x b \tau$ 
let ?if =  $\lambda f b c. \text{if } f b \tau \text{ then } b \tau \text{ else } c$ 
let ?forall =  $\lambda P. ?if P \text{ false } (?if P \text{ invalid } (?if P \text{ null } (\text{true } \tau)))$ 
show ?thesis
apply(simp only: OclForall-def OclIterate-def)
apply(case-tac  $\tau \models \delta S$ , simp only: OclValid-def)
apply(subgoal-tac let set =  $\ulcorner \text{Rep-Set}_{base} (S \tau) \urcorner$  in
  ?forall (?P set) =
    Finite-Set.fold ( $\lambda x \text{ acc. acc and } P x$ ) true (( $\lambda a \tau. a$ ) ‘ set)  $\tau$ ,
  simp only: Let-def, simp add: S-finite, simp only: Let-def)
apply(case-tac  $\ulcorner \text{Rep-Set}_{base} (S \tau) \urcorner = \{\}$ , simp)
apply(rule finite-ne-induct[OF S-finite], simp)

apply(simp only: image-insert)
apply(subst and-comm.fold-insert, simp)
apply (metis empty-iff image-empty)
apply(simp add: invalid-def)
apply (metis bot-fun-def destruct-ocl null-fun-def)

apply(simp only: image-insert)
apply(subst and-comm.fold-insert, simp)
apply (metis (mono-tags) imageE)

apply(subst cp-OclAnd) apply(drule sym, drule sym, simp only:, drule sym, simp only:)
apply(simp only: ex-insert)
apply(subgoal-tac  $\exists x. x \in F$ ) prefer 2
apply(metis all-not-in-conv)
proof – fix  $x F$  show  $(\delta S) \tau = \text{true } \tau \implies \exists x. x \in F \implies$ 
  ?forall ( $\lambda b \tau. ?P\text{-eq } x b \tau \vee ?P F b \tau$ ) =
  ( $\lambda \cdot. ?forall$  (?P F)) and ( $\lambda \cdot. P (\lambda \tau. x) \tau$ )  $\tau$ 
apply(rule disjE4[OF destruct-ocl[where x1 = P ( $\lambda \tau. x) \tau$ ]])
apply(simp-all add: true-def false-def invalid-def OclAnd-def
  null-fun-def null-option-def bot-fun-def bot-option-def)
by (metis (lifting) option.distinct(1)+)
qed(simp add: OclValid-def)+
qed

```

lemma *OclForall-cong*:
assumes $\bigwedge x. x \in {}^\top \text{Rep-Set}_{\text{base}} (X \ \tau)^\top \implies \tau \models P (\lambda \tau. x) \implies \tau \models Q (\lambda \tau. x)$
assumes $P: \tau \models \text{OclForall } X \ P$
shows $\tau \models \text{OclForall } X \ Q$
proof –
have *def-X*: $\tau \models \delta \ X$
by(*insert P, simp add: OclForall-def OclValid-def bot-option-def true-def split: if-split-asm*)
show *?thesis*
apply(*insert P*)
apply(*subst (asm) OclForall-rep-set-true[OF def-X], subst OclForall-rep-set-true[OF def-X]*)
by (*simp add: assms*)
qed

lemma *OclForall-cong'*:
assumes $\bigwedge x. x \in {}^\top \text{Rep-Set}_{\text{base}} (X \ \tau)^\top \implies \tau \models P (\lambda \tau. x) \implies \tau \models Q (\lambda \tau. x) \implies \tau \models R (\lambda \tau. x)$
assumes $P: \tau \models \text{OclForall } X \ P$
assumes $Q: \tau \models \text{OclForall } X \ Q$
shows $\tau \models \text{OclForall } X \ R$
proof –
have *def-X*: $\tau \models \delta \ X$
by(*insert P, simp add: OclForall-def OclValid-def bot-option-def true-def split: if-split-asm*)
show *?thesis*
apply(*insert P Q*)
apply(*subst (asm) (1 2) OclForall-rep-set-true[OF def-X], subst OclForall-rep-set-true[OF def-X]*)
by (*simp add: assms*)
qed

Strict Equality

lemma *StrictRefEqSet-defined* :
assumes *x-def*: $\tau \models \delta \ x$
assumes *y-def*: $\tau \models \delta \ y$
shows $((x::(\mathbb{A}, \alpha::\text{null})\text{Set}) \doteq y) \ \tau =$
 $(x \rightarrow \text{forAll}_{\text{Set}}(z \mid y \rightarrow \text{includes}_{\text{Set}}(z))) \ \text{and} \ (y \rightarrow \text{forAll}_{\text{Set}}(z \mid x \rightarrow \text{includes}_{\text{Set}}(z))) \ \tau$
proof –
have *rep-set-inj* : $\bigwedge \tau. (\delta \ x) \ \tau = \text{true} \ \tau \implies$
 $(\delta \ y) \ \tau = \text{true} \ \tau \implies$
 $x \ \tau \neq y \ \tau \implies$
 ${}^\top \text{Rep-Set}_{\text{base}} (y \ \tau)^\top \neq {}^\top \text{Rep-Set}_{\text{base}} (x \ \tau)^\top$
apply(*simp add: defined-def*)
apply(*split if-split-asm, simp add: false-def true-def*)
apply(*simp add: null-fun-def null-Set_{base}-def bot-fun-def bot-Set_{base}-def*)

apply(*case-tac x \ \tau, rename-tac x'*)
apply(*case-tac x', simp-all, rename-tac x''*)
apply(*case-tac x'', simp-all*)

apply(*case-tac y \ \tau, rename-tac y'*)
apply(*case-tac y', simp-all, rename-tac y''*)
apply(*case-tac y'', simp-all*)

apply(*simp add: Abs-Set_{base}-inverse*)
by(*blast*)

show *?thesis*
apply(*simp add: StrictRefEqSet StrongEq-def*
foundation20[OF x-def, simplified OclValid-def]
foundation20[OF y-def, simplified OclValid-def])

```

apply(subgoal-tac  $\sqcup x \tau = y \tau_{\sqcup} = \text{true} \tau \vee \sqcup x \tau = y \tau_{\sqcup} = \text{false} \tau$ )
prefer 2
apply(simp add: false-def true-def)

apply(erule disjE)
apply(simp add: true-def)

apply(subgoal-tac ( $\tau \models \text{OclForall } x (\text{OclIncludes } y) \wedge (\tau \models \text{OclForall } y (\text{OclIncludes } x))$ ))
apply(subst cp-OclAnd, simp add: true-def OclValid-def)
apply(simp add: OclForall-includes[OF x-def y-def]
      OclForall-includes[OF y-def x-def])

apply(simp)

apply(subgoal-tac  $\text{OclForall } x (\text{OclIncludes } y) \tau = \text{false} \tau \vee$ 
       $\text{OclForall } y (\text{OclIncludes } x) \tau = \text{false} \tau$ )
apply(subst cp-OclAnd, metis OclAnd-false1 OclAnd-false2 cp-OclAnd)
apply(simp only: OclForall-not-includes[OF x-def y-def, simplified OclValid-def]
      OclForall-not-includes[OF y-def x-def, simplified OclValid-def],
      simp add: false-def)
by (metis OclValid-def rep-set-inj subset-antisym x-def y-def)
qed

lemma StrictRefEqSet-exec[simp,code-unfold] :
(( $x::(\mathcal{A}, \alpha::\text{null})\text{Set} \doteq y$ ) =
  (if  $\delta x$  then (if  $\delta y$ 
    then ( $(x \rightarrow \text{forAll}_{\text{Set}}(z) \mid y \rightarrow \text{includes}_{\text{Set}}(z))$  and  $(y \rightarrow \text{forAll}_{\text{Set}}(z) \mid x \rightarrow \text{includes}_{\text{Set}}(z))$ ))
    else if  $v y$ 
      then  $\text{false} \text{--- } x' \rightarrow \text{includes} = \text{null}$ 
      else invalid
      endif
    endif)
  else if  $v x$  ---  $\text{null} = ???$ 
    then if  $v y$  then  $\text{not}(\delta y)$  else invalid endif
    else invalid
    endif
  endif)

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

have valid-inject-true :  $\bigwedge \tau P. (\neg (\tau \models v P)) = ((v P) \tau = \text{false } \tau)$ 
by (metis bot-fun-def OclIf-true' OclIncludes-charn0 OclIncludes-charn0' OclValid-def valid-def
      foundation6 foundation9)
show ?thesis
apply(rule ext, rename-tac  $\tau$ )

apply(simp add: OclIf-def
      defined-inject-true[simplified OclValid-def]
      valid-inject-true[simplified OclValid-def],
      subst false-def, subst true-def, simp)
apply(subst (1 2) cp-OclNot, simp, simp add: cp-OclNot[symmetric])
apply(simp add: StrictRefEqSet-defined[simplified OclValid-def])
by(simp add: StrictRefEqSet StrongEq-def false-def true-def valid-def defined-def)
qed

```

lemma *StrictRefEqSet-L-subst1* : $cp\ P \implies \tau \models v\ x \implies \tau \models v\ y \implies \tau \models v\ P\ x \implies \tau \models v\ P\ y \implies$
 $\tau \models (x :: ('A, 'a :: null) Set) \doteq y \implies \tau \models (P\ x :: ('A, 'a :: null) Set) \doteq P\ y$
apply(*simp only: StrictRefEqSet OclValid-def*)
apply(*split if-split-asm*)
apply(*simp add: StrongEq-L-subst1[simplified OclValid-def]*)
by (*simp add: invalid-def bot-option-def true-def*)

lemma *OclIncluding-cong'* :
shows $\tau \models \delta\ s \implies \tau \models \delta\ t \implies \tau \models v\ x \implies$
 $\tau \models ((s :: ('A, 'a :: null) Set) \doteq t) \implies \tau \models (s \rightarrow including_{Set}(x) \doteq (t \rightarrow including_{Set}(x)))$

proof –
have $cp : cp\ (\lambda s. (s \rightarrow including_{Set}(x)))$
apply(*simp add: cp-def, subst OclIncluding.cp0*)
by (*rule-tac x = (\lambda x ab. ((\lambda-. x ab) \rightarrow including_{Set}(\lambda-. x ab)) ab) in exI, simp*)

show $\tau \models \delta\ s \implies \tau \models \delta\ t \implies \tau \models v\ x \implies \tau \models (s \doteq t) \implies ?thesis$
apply(*rule-tac P = \lambda s. (s \rightarrow including_{Set}(x)) in StrictRefEqSet-L-subst1*)
apply(*rule cp*)
apply(*simp add: foundation20*) **apply**(*simp add: foundation20*)
apply (*simp add: foundation10 foundation6*)+
done
qed

lemma *OclIncluding-cong* : $\bigwedge (s :: ('A, 'a :: null) Set)\ t\ x\ y\ \tau. \tau \models \delta\ t \implies \tau \models v\ y \implies$
 $\tau \models s \doteq t \implies x = y \implies \tau \models s \rightarrow including_{Set}(x) \doteq (t \rightarrow including_{Set}(y))$
apply(*simp only:*)
apply(*rule OclIncluding-cong', simp-all only:*)
by(*auto simp: OclValid-def OclIf-def invalid-def bot-option-def OclNot-def split : if-split-asm*)

lemma *const-StrictRefEqSet-empty* : $const\ X \implies const\ (X \doteq Set\{\})$
apply(*rule StrictRefEqSet.const, assumption*)
by(*simp*)

lemma *const-StrictRefEqSet-including* :
 $const\ a \implies const\ S \implies const\ X \implies const\ (X \doteq S \rightarrow including_{Set}(a))$
apply(*rule StrictRefEqSet.const, assumption*)
by(*rule const-OclIncluding*)

2.9.26. Test Statements

Assert $(\tau \models (Set\{\lambda-. \underline{x}\} \doteq Set\{\lambda-. \underline{x}\}))$
Assert $(\tau \models (Set\{\lambda-. \underline{x}\} \doteq Set\{\lambda-. \underline{x}\}))$

instantiation *Set_base* :: (*equal*)*equal*
begin
definition *HOL.equal* $k\ l \longleftrightarrow (k :: ('a :: equal) Set_{base}) = l$
instance **by** *standard* (*rule equal-Set_base-def*)
end

lemma *equal-Set_base-code* [*code*]:
 $HOL.equal\ k\ (l :: ('a :: \{equal, null\}) Set_{base}) \longleftrightarrow Rep_Set_{base}\ k = Rep_Set_{base}\ l$
by (*auto simp add: equal Set_base.Rep-Set_base-inject*)

Assert $\tau \models (Set\{\} \doteq Set\{\})$
Assert $\tau \models (Set\{\mathbf{1}, \mathbf{2}\} \triangleq Set\{\} \rightarrow including_{Set}(\mathbf{2}) \rightarrow including_{Set}(\mathbf{1}))$
Assert $\tau \models (Set\{\mathbf{1}, invalid, \mathbf{2}\} \triangleq invalid)$

```

Assert  $\tau \models (\text{Set}\{\mathbf{1}, \mathbf{2}\} \rightarrow \text{including}_{\text{Set}}(\text{null}) \triangleq \text{Set}\{\text{null}, \mathbf{1}, \mathbf{2}\})$ 
Assert  $\tau \models (\text{Set}\{\mathbf{1}, \mathbf{2}\} \rightarrow \text{including}_{\text{Set}}(\text{null}) \triangleq \text{Set}\{\mathbf{1}, \mathbf{2}, \text{null}\})$ 

```

end

```

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

no-notation None ( $\langle \perp \rangle$ )

```

2.10. Collection Type Sequence: Operations

2.10.1. Basic Properties of the Sequence Type

Every element in a defined sequence is valid.

```

lemma Sequence-inv-lemma:  $\tau \models (\delta X) \implies \forall x \in \text{set } \ulcorner \text{Rep-Sequence}_{\text{base}}(X \tau) \urcorner. x \neq \text{bot}$ 
apply(insert Rep-Sequencebase [of X  $\tau$ ], simp)
apply(auto simp: OclValid-def defined-def false-def true-def cp-def
          bot-fun-def bot-Sequencebase-def null-Sequencebase-def null-fun-def
          split-if-split-asm)
apply(erule contrapos-pp [of Rep-Sequencebase(X  $\tau$ ) = bot])
apply(subst Abs-Sequencebase-inject[symmetric], rule Rep-Sequencebase, simp)
apply(simp add: Rep-Sequencebase-inverse bot-Sequencebase-def bot-option-def)
apply(erule contrapos-pp [of Rep-Sequencebase(X  $\tau$ ) = null])
apply(subst Abs-Sequencebase-inject[symmetric], rule Rep-Sequencebase, simp)
apply(simp add: Rep-Sequencebase-inverse null-option-def)
by (simp add: bot-option-def)

```

2.10.2. Definition: Strict Equality

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

```

overloading
  StrictRefEq  $\equiv$  StrictRefEq :: [ $(\mathfrak{A}, \alpha::\text{null})\text{Sequence}, (\mathfrak{A}, \alpha::\text{null})\text{Sequence}$ ]  $\Rightarrow$   $(\mathfrak{A})\text{Boolean}$ 
begin
  definition StrictRefEqSeq :
     $((x::(\mathfrak{A}, \alpha::\text{null})\text{Sequence}) \doteq y) \equiv (\lambda \tau. \text{if } (v x) \tau = \text{true } \tau \wedge (v y) \tau = \text{true } \tau$ 
       $\text{then } (x \triangleq y) \tau$ 
       $\text{else invalid } \tau)$ 
end

```

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

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

interpretation $StrictRefEq_{Seq} : profile-bin_{StrongEq^{-v}v} \lambda x y. (x :: ('A, 'a :: null) Sequence) \doteq y$
by $unfold-locales (auto simp: StrictRefEq_{Seq})$

2.10.3. Constants: mtSequence

definition $mtSequence :: ('A, 'a :: null) Sequence \langle \langle Sequence \rangle \rangle$
where $Sequence \{ \} \equiv (\lambda \tau. Abs-Sequence_{base} \sqcup \tau :: 'a list_{\perp})$

lemma $mtSequence-defined[simp, code-unfold]: \delta(Sequence \{ \}) = true$
apply $(rule ext, auto simp: mtSequence-def defined-def null-Sequence_{base}-def$
 $bot-Sequence_{base}-def bot-fun-def null-fun-def)$
by $(simp-all add: Abs-Sequence_{base}-inject bot-option-def null-option-def)$

lemma $mtSequence-valid[simp, code-unfold]: v(Sequence \{ \}) = true$
apply $(rule ext, auto simp: mtSequence-def valid-def null-Sequence_{base}-def$
 $bot-Sequence_{base}-def bot-fun-def null-fun-def)$
by $(simp-all add: Abs-Sequence_{base}-inject bot-option-def null-option-def)$

lemma $mtSequence-rep-set: \ulcorner Rep-Sequence_{base} (Sequence \{ \} \tau) \urcorner = []$
apply $(simp add: mtSequence-def, subst Abs-Sequence_{base}-inverse)$
by $(simp add: bot-option-def) + \text{lemma } [simp, code-unfold]: const Sequence \{ \}$
by $(simp add: const-def mtSequence-def)$

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

2.10.4. Definition: Prepend

definition $OclPrepend :: [('A, 'a :: null) Sequence, ('A, 'a) val] \Rightarrow ('A, 'a) Sequence$
where $OclPrepend x y = (\lambda \tau. \text{if } (\delta x) \tau = true \wedge (v y) \tau = true \wedge$
 $\text{then } Abs-Sequence_{base} \sqcup (y \tau) \# \ulcorner Rep-Sequence_{base} (x \tau) \urcorner$
 $\text{else } invalid \tau)$
notation $OclPrepend \langle \langle -- \rangle \rangle \text{prepend}_{Seq} ('A, 'a)$

interpretation $OclPrepend: profile-bin_{d-v} OclPrepend \lambda x y. Abs-Sequence_{base} \sqcup y \# \ulcorner Rep-Sequence_{base} x \urcorner$
proof –

have $A : \bigwedge x y. x \neq bot \implies x \neq null \implies y \neq bot \implies$
 $\sqcup y \# \ulcorner Rep-Sequence_{base} x \urcorner \in \{ X. X = bot \vee X = null \vee (\forall x \in set \ulcorner X \urcorner. x \neq bot) \}$
by $(auto intro!: Sequence-inv-lemma[simplified OclValid-def$
 $defined-def false-def true-def null-fun-def bot-fun-def])$

show $profile-bin_{d-v} OclPrepend (\lambda x y. Abs-Sequence_{base} \sqcup y \# \ulcorner Rep-Sequence_{base} x \urcorner)$
apply $unfold-locales$
apply $(auto simp: OclPrepend-def bot-option-def null-option-def null-Sequence_{base}-def$
 $bot-Sequence_{base}-def)$
apply $(erule-tac Q = Abs-Sequence_{base} \sqcup y \# \ulcorner Rep-Sequence_{base} x \urcorner = Abs-Sequence_{base} None$
in $contrapos-pp)$
apply $(subst Abs-Sequence_{base}-inject [OF A])$
apply $(simp-all add: null-Sequence_{base}-def bot-Sequence_{base}-def bot-option-def)$
apply $(erule-tac Q = Abs-Sequence_{base} \sqcup y \# \ulcorner Rep-Sequence_{base} x \urcorner = Abs-Sequence_{base} \sqcup None_{\perp}$
in $contrapos-pp)$
apply $(subst Abs-Sequence_{base}-inject [OF A])$
apply $(simp-all add: null-Sequence_{base}-def bot-Sequence_{base}-def$
 $bot-option-def null-option-def)$

done

qed

syntax

-OclFinsequence :: args => (' \mathfrak{A} ', 'a::null) Sequence (\langle Sequence{(-)} \rangle)

syntax-consts

-OclFinsequence == OclPrepend

translations

Sequence{x, xs} == CONST OclPrepend (Sequence{xs}) x

Sequence{x} == CONST OclPrepend (Sequence{ }) x

2.10.5. Definition: Including

definition OclIncluding :: [(' \mathfrak{A} ', 'a::null) Sequence, (' \mathfrak{A} ', 'a) val] => (' \mathfrak{A} ', 'a) Sequence

where OclIncluding x y = ($\lambda \tau$. if (δx) $\tau = true \tau \wedge (v y) \tau = true \tau$
then Abs-Sequence_{base} \perp \ulcorner Rep-Sequence_{base} (x τ) \urcorner @ [y τ] \perp
else invalid τ)

notation OclIncluding (\langle ->including_{Seq}'(-') \rangle)

interpretation OclIncluding :

profile-bin_{d-v} OclIncluding $\lambda x y$. Abs-Sequence_{base} \perp \ulcorner Rep-Sequence_{base} x \urcorner @ [y] \perp

proof -

have A : $\bigwedge x y$. $x \neq bot \implies x \neq null \implies y \neq bot \implies$
 $\perp \ulcorner$ Rep-Sequence_{base} x \urcorner @ [y] $\perp \in \{X. X = bot \vee X = null \vee (\forall x \in set \ulcorner X \urcorner. x \neq bot)\}$
by(auto intro!:Sequence-inv-lemma[simplified OclValid-def
defined-def false-def true-def null-fun-def bot-fun-def])

show profile-bin_{d-v} OclIncluding ($\lambda x y$. Abs-Sequence_{base} \perp \ulcorner Rep-Sequence_{base} x \urcorner @ [y] \perp)

apply unfold-locales

apply(auto simp:OclIncluding-def bot-option-def null-option-def null-Sequence_{base}-def
bot-Sequence_{base}-def)

apply(erule-tac Q=Abs-Sequence_{base} \perp \ulcorner Rep-Sequence_{base} x \urcorner @ [y] $\perp = Abs-Sequence_{base} None$
in contrapos-pp)

apply(subst Abs-Sequence_{base}-inject [OF A])

apply(simp-all add: null-Sequence_{base}-def bot-Sequence_{base}-def bot-option-def)

apply(erule-tac Q=Abs-Sequence_{base} \perp \ulcorner Rep-Sequence_{base} x \urcorner @ [y] $\perp = Abs-Sequence_{base} \perp None \urcorner
in contrapos-pp)$

apply(subst Abs-Sequence_{base}-inject[OF A])

apply(simp-all add: null-Sequence_{base}-def bot-Sequence_{base}-def bot-option-def null-option-def)
done

qed

lemma [simp,code-unfold] : (Sequence{ }->including_{Seq}(a)) = (Sequence{ }->prepend_{Seq}(a))

apply(simp add: OclIncluding-def OclPrepend-def mtSequence-def)

apply(subst (1 2) Abs-Sequence_{base}-inverse, simp)

by(metis drop.simps append-Nil)

lemma [simp,code-unfold] : ((S->prepend_{Seq}(a))->including_{Seq}(b)) =
((S->including_{Seq}(b))->prepend_{Seq}(a))

proof -

have A : $\bigwedge S b \tau$. $S \neq \perp \implies S \neq null \implies b \neq \perp \implies$
 $\perp \ulcorner$ Rep-Sequence_{base} S \urcorner @ [b] $\perp \in \{X. X = bot \vee X = null \vee (\forall x \in set \ulcorner X \urcorner. x \neq \perp)\}$
by(auto intro!:Sequence-inv-lemma[simplified OclValid-def
defined-def false-def true-def null-fun-def bot-fun-def])

have B : $\bigwedge S a \tau$. $S \neq \perp \implies S \neq null \implies a \neq \perp \implies$
 \ulcorner a $\#$ \ulcorner Rep-Sequence_{base} S \urcorner $\perp \in \{X. X = bot \vee X = null \vee (\forall x \in set \ulcorner X \urcorner. x \neq \perp)\}$
by(auto intro!:Sequence-inv-lemma[simplified OclValid-def
defined-def false-def true-def null-fun-def bot-fun-def])

show *?thesis*
apply (*simp add: OclIncluding-def OclPrepend-def mtSequence-def, rule ext*)
apply (*subst (2 5) cp-defined, simp split:*)
apply (*intro conjI impI*)
apply (*subst Abs-Sequence_{base}-inverse[OF B],*
(simp add: foundation16[simplified OclValid-def] foundation18'[simplified OclValid-def])+)
apply (*subst Abs-Sequence_{base}-inverse[OF A],*
(simp add: foundation16[simplified OclValid-def] foundation18'[simplified OclValid-def])+)
apply (*simp add: OclIncluding.def-body*)
apply (*metis OclValid-def foundation16 invalid-def*)
apply (*metis (no-types) OclPrepend.def-body' OclValid-def foundation16*)
by (*metis OclValid-def foundation16 invalid-def*)
qed

2.10.6. Definition: Excluding

definition *OclExcluding* :: [*($\mathfrak{A}, \alpha :: \text{null}$) Sequence, (\mathfrak{A}, α) val*] \Rightarrow (*\mathfrak{A}, α*) *Sequence*
where *OclExcluding* $x\ y = (\lambda\ \tau.$ *if* ($\delta\ x$) $\tau = \text{true}$ $\tau \wedge (v\ y)\ \tau = \text{true}$ τ
then *Abs-Sequence_{base}* \sqcup *filter* ($\lambda x. x = y\ \tau$)
 \sqcup \ulcorner *Rep-Sequence_{base}* ($x\ \tau$) \urcorner
else *invalid* τ)

notation *OclExcluding* ($\langle \dashv \dashrightarrow \text{excluding}_{\text{Seq}}'(-) \rangle$)

interpretation *OclExcluding:profile-bin_{d-v}* *OclExcluding*

$\lambda x\ y. \text{Abs-Sequence}_{\text{base}} \sqcup \text{filter} (\lambda x. x = y) \ulcorner \text{Rep-Sequence}_{\text{base}} (x) \urcorner$

proof –

show *profile-bin_{d-v}* *OclExcluding* ($\lambda x\ y. \text{Abs-Sequence}_{\text{base}} \sqcup [x \leftarrow \ulcorner \text{Rep-Sequence}_{\text{base}} x \urcorner . x = y]_{\sqcup}$)

apply *unfold-locales*

apply (*auto simp: OclExcluding-def bot-option-def null-option-def*

null-Sequence_{base}-def bot-Sequence_{base}-def)

apply (*subst (asm) Abs-Sequence_{base}-inject,*

simp-all add: null-Sequence_{base}-def bot-Sequence_{base}-def bot-option-def null-option-def)
done

qed

2.10.7. Definition: Append

Identical to *OclIncluding*.

definition *OclAppend* :: [*($\mathfrak{A}, \alpha :: \text{null}$) Sequence, (\mathfrak{A}, α) val*] \Rightarrow (*\mathfrak{A}, α*) *Sequence*

where *OclAppend* = *OclIncluding*

notation *OclAppend* ($\langle \dashv \dashrightarrow \text{append}_{\text{Seq}}'(-) \rangle$)

interpretation *OclAppend* :

$\text{profile-bin}_{d-v} \text{OclAppend} \lambda x\ y. \text{Abs-Sequence}_{\text{base}} \sqcup \ulcorner \text{Rep-Sequence}_{\text{base}} x \urcorner @ [y]_{\sqcup}$

apply *unfold-locales*

by (*auto simp: OclAppend-def bin-def bin'-def*

OclIncluding.def-scheme OclIncluding.def-body)

2.10.8. Definition: Union

definition *OclUnion* :: [*($\mathfrak{A}, \alpha :: \text{null}$) Sequence, (\mathfrak{A}, α) Sequence*] \Rightarrow (*\mathfrak{A}, α*) *Sequence*

where *OclUnion* $x\ y = (\lambda\ \tau.$ *if* ($\delta\ x$) $\tau = \text{true}$ $\tau \wedge (\delta\ y)\ \tau = \text{true}$ τ

then *Abs-Sequence_{base}* \sqcup \ulcorner *Rep-Sequence_{base}* ($x\ \tau$) \urcorner @

\ulcorner *Rep-Sequence_{base}* ($y\ \tau$) \urcorner

else *invalid* τ)

notation *OclUnion* ($\langle \dashv \dashrightarrow \text{union}_{\text{Seq}}'(-) \rangle$)

interpretation *OclUnion* :

profile-bin_{d-d} *OclUnion* $\lambda x y. \text{Abs-Sequence}_{\text{base}\perp} \ulcorner \text{Rep-Sequence}_{\text{base}} x \urcorner @ \ulcorner \text{Rep-Sequence}_{\text{base}} y \urcorner \perp$

proof –

have $A : \bigwedge x y. x \neq \perp \implies x \neq \text{null} \implies \forall x \in \text{set } \ulcorner \text{Rep-Sequence}_{\text{base}} x \urcorner. x \neq \perp$

apply(*rule Sequence-inv-lemma*[of τ])

by(*simp add: defined-def OclValid-def bot-fun-def null-fun-def false-def true-def*)

show *profile-bin_{d-d}* *OclUnion* $(\lambda x y. \text{Abs-Sequence}_{\text{base}\perp} \ulcorner \text{Rep-Sequence}_{\text{base}} x \urcorner @ \ulcorner \text{Rep-Sequence}_{\text{base}} y \urcorner \perp)$

apply *unfold-locales*

apply(*auto simp: OclUnion-def bot-option-def null-option-def*

null-Sequence_{base}-def bot-Sequence_{base}-def)

by(*subst (asm) Abs-Sequence_{base}-inject,*

simp-all add: bot-option-def null-option-def Set.ball-Un A null-Sequence_{base}-def bot-Sequence_{base}-def)+

qed

2.10.9. Definition: At

definition *OclAt* :: [$(\mathfrak{A}, \alpha :: \text{null}) \text{Sequence}, (\mathfrak{A}) \text{Integer}$] $\Rightarrow (\mathfrak{A}, \alpha) \text{val}$

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

then if $1 \leq \ulcorner y \tau \urcorner \wedge \ulcorner y \tau \urcorner \leq \text{length } \ulcorner \text{Rep-Sequence}_{\text{base}} (x \tau) \urcorner$

then $\ulcorner \text{Rep-Sequence}_{\text{base}} (x \tau) \urcorner ! (\text{nat } \ulcorner y \tau \urcorner - 1)$

else invalid τ

else invalid τ)

notation *OclAt* $(\langle \dashrightarrow \text{at}_{\text{Seq}} '(-)' \rangle)$

2.10.10. Definition: First

definition *OclFirst* :: [$(\mathfrak{A}, \alpha :: \text{null}) \text{Sequence}$] $\Rightarrow (\mathfrak{A}, \alpha) \text{val}$

where *OclFirst* $x = (\lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \text{ then}$

case $\ulcorner \text{Rep-Sequence}_{\text{base}} (x \tau) \urcorner$ of [] $\Rightarrow \text{invalid } \tau$

| $x \# - \Rightarrow x$

else invalid τ)

notation *OclFirst* $(\langle \dashrightarrow \text{first}_{\text{Seq}} '(-)' \rangle)$

2.10.11. Definition: Last

definition *OclLast* :: [$(\mathfrak{A}, \alpha :: \text{null}) \text{Sequence}$] $\Rightarrow (\mathfrak{A}, \alpha) \text{val}$

where *OclLast* $x = (\lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \text{ then}$

if $\ulcorner \text{Rep-Sequence}_{\text{base}} (x \tau) \urcorner = []$ *then*

invalid τ

else

last $\ulcorner \text{Rep-Sequence}_{\text{base}} (x \tau) \urcorner$

else invalid τ)

notation *OclLast* $(\langle \dashrightarrow \text{last}_{\text{Seq}} '(-)' \rangle)$

2.10.12. Definition: Iterate

definition *OclIterate* :: [$(\mathfrak{A}, \alpha :: \text{null}) \text{Sequence}, (\mathfrak{A}, \beta :: \text{null}) \text{val},$

$(\mathfrak{A}, \alpha) \text{val} \Rightarrow (\mathfrak{A}, \beta) \text{val} \Rightarrow (\mathfrak{A}, \beta) \text{val}$] $\Rightarrow (\mathfrak{A}, \beta) \text{val}$

where *OclIterate* $S A F = (\lambda \tau. \text{if } (\delta S) \tau = \text{true } \tau \wedge (v A) \tau = \text{true } \tau$

then (*foldr* (F) (*map* $(\lambda a \tau. a) \ulcorner \text{Rep-Sequence}_{\text{base}} (S \tau) \urcorner$))(A) τ

else \perp)

syntax

-*OclIterateSeq* :: [$(\mathfrak{A}, \alpha :: \text{null}) \text{Sequence}, \text{idt}, \text{idt}, \alpha, \beta$] $\Rightarrow (\mathfrak{A}, \gamma) \text{val}$

$(\langle \dashrightarrow \text{iterates}_{\text{Seq}} '(-); (-) | -' \rangle)$

syntax-consts

-*OclIterateSeq* == *OclIterate*

translations

$X \dashrightarrow \text{iterates}_{\text{Seq}}(a; x = A | P) == \text{CONST } \text{OclIterate } X A (\%a. (\% x. P))$

2.10.13. Definition: Forall

definition *OclForall* :: [(\mathfrak{A} , $\alpha::\text{null}$) Sequence, (\mathfrak{A} , α) val \Rightarrow (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean
where *OclForall* $S P = (S \rightarrow \text{iterate}_{Seq}(b; x = \text{true} \mid x \text{ and } (P b)))$

syntax

-*OclForallSeq* :: [(\mathfrak{A} , $\alpha::\text{null}$) Sequence, id, (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean ($\langle(-) \rightarrow \text{forall}_{Seq} '(-)'\rangle$)

syntax-consts

-*OclForallSeq* == UML-Sequence.OclForall

translations

$X \rightarrow \text{forall}_{Seq}(x \mid P) == \text{CONST UML-Sequence.OclForall } X (\%x. P)$

2.10.14. Definition: Exists

definition *OclExists* :: [(\mathfrak{A} , $\alpha::\text{null}$) Sequence, (\mathfrak{A} , α) val \Rightarrow (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean
where *OclExists* $S P = (S \rightarrow \text{iterate}_{Seq}(b; x = \text{false} \mid x \text{ or } (P b)))$

syntax

-*OclExistsSeq* :: [(\mathfrak{A} , $\alpha::\text{null}$) Sequence, id, (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean ($\langle(-) \rightarrow \text{exists}_{Seq} '(-)'\rangle$)

syntax-consts

-*OclExistsSeq* == *OclExists*

translations

$X \rightarrow \text{exists}_{Seq}(x \mid P) == \text{CONST OclExists } X (\%x. P)$

2.10.15. Definition: Collect

definition *OclCollect* :: [(\mathfrak{A} , $\alpha::\text{null}$) Sequence, (\mathfrak{A} , α) val \Rightarrow (\mathfrak{A} , β) val \Rightarrow (\mathfrak{A} , $\beta::\text{null}$) Sequence] \Rightarrow (\mathfrak{A} , $\alpha::\text{null}$) Sequence
where *OclCollect* $S P = (S \rightarrow \text{iterate}_{Seq}(b; x = \text{Sequence}\{\} \mid x \rightarrow \text{prepend}_{Seq}(P b)))$

syntax

-*OclCollectSeq* :: [(\mathfrak{A} , $\alpha::\text{null}$) Sequence, id, (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean ($\langle(-) \rightarrow \text{collect}_{Seq} '(-)'\rangle$)

syntax-consts

-*OclCollectSeq* == *OclCollect*

translations

$X \rightarrow \text{collect}_{Seq}(x \mid P) == \text{CONST OclCollect } X (\%x. P)$

2.10.16. Definition: Select

definition *OclSelect* :: [(\mathfrak{A} , $\alpha::\text{null}$) Sequence, (\mathfrak{A} , α) val \Rightarrow (\mathfrak{A}) Boolean] \Rightarrow (\mathfrak{A} , $\alpha::\text{null}$) Sequence
where *OclSelect* $S P = (S \rightarrow \text{iterate}_{Seq}(b; x = \text{Sequence}\{\} \mid \text{if } P \text{ then } x \rightarrow \text{prepend}_{Seq}(b) \text{ else } x \text{ endif}))$

syntax

-*OclSelectSeq* :: [(\mathfrak{A} , $\alpha::\text{null}$) Sequence, id, (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean ($\langle(-) \rightarrow \text{select}_{Seq} '(-)'\rangle$)

syntax-consts

-*OclSelectSeq* == UML-Sequence.OclSelect

translations

$X \rightarrow \text{select}_{Seq}(x \mid P) == \text{CONST UML-Sequence.OclSelect } X (\%x. P)$

2.10.17. Definition: Size

definition *OclSize* :: [(\mathfrak{A} , $\alpha::\text{null}$) Sequence] \Rightarrow (\mathfrak{A}) Integer ($\langle(-) \rightarrow \text{size}_{Seq} '()\rangle$)
where *OclSize* $S = (S \rightarrow \text{iterate}_{Seq}(b; x = \mathbf{0} \mid x +_{int} \mathbf{1}))$

2.10.18. Definition: IsEmpty

definition *OclIsEmpty* :: (\mathfrak{A} , $\alpha::\text{null}$) Sequence \Rightarrow \mathfrak{A} Boolean
where *OclIsEmpty* $x = ((v x \text{ and not } (\delta x)) \text{ or } ((\text{OclSize } x) \doteq \mathbf{0}))$

notation $OclIsEmpty$ $(\langle \dashrightarrow isEmptySeq'(\cdot) \rangle)$

2.10.19. Definition: NotEmpty

definition $OclNotEmpty$ $:: ('\mathfrak{A}, '\alpha :: null) Sequence \Rightarrow '\mathfrak{A} Boolean$

where $OclNotEmpty x = not(OclIsEmpty x)$

notation $OclNotEmpty$ $(\langle \dashrightarrow notEmptySeq'(\cdot) \rangle)$

2.10.20. Definition: Any

definition $OclANY x = (\lambda \tau.$

if $x \tau = invalid \tau$ *then*

\perp

else

case $drop (drop (Rep-Sequence_{base} (x \tau)))$ *of* $[] \Rightarrow \perp$
 $| l \Rightarrow hd l$)

notation $OclANY$ $(\langle \dashrightarrow anySeq'(\cdot) \rangle)$

2.10.21. Definition (future operators)

consts

$OclCount$ $:: [('\mathfrak{A}, '\alpha :: null) Sequence, ('\mathfrak{A}, '\alpha) Sequence] \Rightarrow '\mathfrak{A} Integer$

$OclSum$ $:: ('\mathfrak{A}, '\alpha :: null) Sequence \Rightarrow '\mathfrak{A} Integer$

notation $OclCount$ $(\langle \dashrightarrow countSeq'(\cdot) \rangle)$

notation $OclSum$ $(\langle \dashrightarrow sumSeq'(\cdot) \rangle)$

2.10.22. Logical Properties

2.10.23. Execution Laws with Invalid or Null as Argument

$OclIterate$

lemma $OclIterate-invalid[simp, code-unfold]: invalid \dashrightarrow iterateSeq(a; x = A | P a x) = invalid$
by $(simp add: OclIterate-def false-def true-def, simp add: invalid-def)$

lemma $OclIterate-null[simp, code-unfold]: null \dashrightarrow iterateSeq(a; x = A | P a x) = invalid$
by $(simp add: OclIterate-def false-def true-def, simp add: invalid-def)$

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

Context Passing

lemma $cp-OclIncluding$:

$(X \dashrightarrow includingSeq(x)) \tau = ((\lambda -. X \tau) \dashrightarrow includingSeq(\lambda -. x \tau)) \tau$

by $(auto simp: OclIncluding-def StrongEq-def invalid-def$
 $cp-defined[symmetric] cp-valid[symmetric])$

lemma $cp-OclIterate$:

$(X \dashrightarrow iterateSeq(a; x = A | P a x)) \tau =$

$((\lambda -. X \tau) \dashrightarrow iterateSeq(a; x = A | P a x)) \tau$

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

lemmas *cp-intro''_{Seq}[intro!,simp,code-unfold]* =
cp-OclIncluding [THEN allI[THEN allI[THEN allI[THEN cpI2]], of OclIncluding]

Const

2.10.24. General Algebraic Execution Rules

Execution Rules on Iterate

lemma *OclIterate-empty[simp,code-unfold]:Sequence{ }->iterate_{Seq}(a; x = A | P a x) = A*
 apply(*simp add: OclIterate-def foundation22[symmetric] foundation13,*
rule ext, rename-tac τ)
 apply(*case-tac $\tau \models v A$, simp-all add: foundation18'*)
 apply(*simp add: mtSequence-def*)
 apply(*subst Abs-Sequence_{base}-inverse*) **by auto**

In particular, this does hold for $A = \text{null}$.

lemma *OclIterate-including[simp,code-unfold]:*
 assumes *strict1 : $\bigwedge X. P \text{ invalid } X = \text{invalid}$*
 and *P-valid-arg: $\bigwedge \tau. (v A) \tau = (v (P a A)) \tau$*
 and *P-cp : $\bigwedge x y \tau. P x y \tau = P (\lambda -. x \tau) y \tau$*
 and *P-cp' : $\bigwedge x y \tau. P x y \tau = P x (\lambda -. y \tau) \tau$*
 shows *(S->including_{Seq}(a))->iterate_{Seq}(b; x = A | P b x) = S->iterate_{Seq}(b; x = P a A | P b x)*
 apply(*rule ext*)
 proof -
 have *A: $\bigwedge S b \tau. S \neq \perp \implies S \neq \text{null} \implies b \neq \perp \implies$*

$$\sqsubseteq^{\ulcorner} \text{Rep-Sequence}_{base} S^{\urcorner} @ [b]_{\sqsubseteq} \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \text{set } \ulcorner X^{\urcorner}. x \neq \perp)\}$$

 by(*auto intro!:Sequence-inv-lemma[simplified OclValid-def*
defined-def false-def true-def null-fun-def bot-fun-def])
 have *P: $\bigwedge l A A' \tau. A \tau = A' \tau \implies \text{foldr } P l A \tau = \text{foldr } P l A' \tau$*
 apply(*rule list.induct, simp, simp*)
 by(*subst (1 2) P-cp', simp*)

fix τ

show *OclIterate (S->including_{Seq}(a)) A P $\tau =$ OclIterate S (P a A) P τ*
 apply(*subst cp-OclIterate, subst OclIncluding-def, simp split:*)
 apply(*intro conjI impI*)

apply(*simp add: OclIterate-def*)
 apply(*intro conjI impI*)
 apply(*subst Abs-Sequence_{base}-inverse[OF A],*
(simp add: foundation16[simplified OclValid-def] foundation18'[simplified OclValid-def])+)
 apply(*rule P, metis P-cp*)
 apply(*metis P-valid-arg*)
 apply(*simp add: P-valid-arg[symmetric]*)
 apply(*metis (lifting, no-types) OclIncluding-def-body' OclValid-def foundation16*)
 apply(*simp add: OclIterate-def defined-def invalid-def bot-option-def bot-fun-def false-def true-def*)
 apply(*intro impI, simp add: false-def true-def P-valid-arg*)
 by(*metis P-cp P-valid-arg UML-Types.bot-fun-def cp-valid invalid-def strict1 true-def valid1 valid-def*)
 qed

lemma *OclIterate-prepend[simp,code-unfold]:*
 assumes *strict1 : $\bigwedge X. P \text{ invalid } X = \text{invalid}$*
 and *strict2 : $\bigwedge X. P X \text{ invalid} = \text{invalid}$*
 and *P-cp : $\bigwedge x y \tau. P x y \tau = P (\lambda -. x \tau) y \tau$*
 and *P-cp' : $\bigwedge x y \tau. P x y \tau = P x (\lambda -. y \tau) \tau$*
 shows *(S->prepend_{Seq}(a))->iterate_{Seq}(b; x = A | P b x) = P a (S->iterate_{Seq}(b; x = A | P b x))*

```

apply(rule ext)
proof -
have B:  $\bigwedge S a \tau. S \neq \perp \implies S \neq \text{null} \implies a \neq \perp \implies$ 
 $\perp a \# \ulcorner \text{Rep-Sequence}_{\text{base}} S^\top \perp \in \{X. X = \text{bot} \vee X = \text{null} \vee (\forall x \in \text{set } \ulcorner X^\top. x \neq \perp)\}$ 
by(auto intro!:Sequence-inv-lemma[simplified OclValid-def
defined-def false-def true-def null-fun-def bot-fun-def])

fix  $\tau$ 
show OclIterate (S->prependSeq(a)) A P  $\tau = P a$  (OclIterate S A P)  $\tau$ 
apply(subst cp-OclIterate, subst OclPrepend-def, simp split:)
apply(intro conjI impI)

apply(subst P-cp')
apply(simp add: OclIterate-def)
apply(intro conjI impI)
apply(subst Abs-Sequencebase-inverse[OF B],
(simp add: foundation16[simplified OclValid-def] foundation18'[simplified OclValid-def])+
apply(simp add: P-cp'[symmetric])
apply(subst P-cp, simp add: P-cp[symmetric])
apply (metis (no-types) OclPrepend.def-body' OclValid-def foundation16)
apply (metis P-cp' invalid-def strict2 valid-def)

apply(subst P-cp',
simp add: OclIterate-def defined-def invalid-def bot-option-def bot-fun-def false-def true-def,
intro conjI impI)
apply (metis P-cp' invalid-def strict2 valid-def)
apply (metis P-cp' invalid-def strict2 valid-def)
apply (metis (no-types) P-cp invalid-def strict1 true-def valid1 valid-def)
apply (metis P-cp' invalid-def strict2 valid-def)
done
qed

```

2.10.25. Test Statements

```

Assert  $\tau \models (\text{Sequence}\{\} \doteq \text{Sequence}\{\})$ 
Assert  $\tau \models (\text{Sequence}\{\mathbf{1},\mathbf{2}\} \triangleq \text{Sequence}\{\}->\text{prepend}_{\text{Seq}}(\mathbf{2})->\text{prepend}_{\text{Seq}}(\mathbf{1}))$ 
Assert  $\tau \models (\text{Sequence}\{\mathbf{1},\text{invalid},\mathbf{2}\} \triangleq \text{invalid})$ 
Assert  $\tau \models (\text{Sequence}\{\mathbf{1},\mathbf{2}\}->\text{prepend}_{\text{Seq}}(\text{null}) \triangleq \text{Sequence}\{\text{null},\mathbf{1},\mathbf{2}\})$ 
Assert  $\tau \models (\text{Sequence}\{\mathbf{1},\mathbf{2}\}->\text{including}_{\text{Seq}}(\text{null}) \triangleq \text{Sequence}\{\mathbf{1},\mathbf{2},\text{null}\})$ 

```

end

theory UML-Library

imports

```

basic-types/UML-Boolean
basic-types/UML-Void
basic-types/UML-Integer
basic-types/UML-Real
basic-types/UML-String

```

```

collection-types/UML-Pair

```

collection-types/UML-Bag
collection-types/UML-Set
collection-types/UML-Sequence

begin

2.11. Miscellaneous Stuff

2.11.1. Definition: asBoolean

definition $OclAsBoolean_{Int} :: ('\mathfrak{A}) Integer \Rightarrow ('\mathfrak{A}) Boolean (\langle(-)\rangle \rightarrow oclAsType_{Int}('Boolean'))$
where $OclAsBoolean_{Int} X = (\lambda\tau. \text{if } (\delta X) \tau = \text{true } \tau$
 $\text{then } \perp^{\lceil X \tau \rceil} \neq 0_{\perp}$
 $\text{else } \text{invalid } \tau)$

interpretation $OclAsBoolean_{Int} : \text{profile-mono}_d OclAsBoolean_{Int} \lambda x. \perp^{\lceil x \rceil} \neq 0_{\perp}$
by $\text{unfold-locale} (\text{auto simp: } OclAsBoolean_{Int}\text{-def bot-option-def})$

definition $OclAsBoolean_{Real} :: ('\mathfrak{A}) Real \Rightarrow ('\mathfrak{A}) Boolean (\langle(-)\rangle \rightarrow oclAsType_{Real}('Boolean'))$
where $OclAsBoolean_{Real} X = (\lambda\tau. \text{if } (\delta X) \tau = \text{true } \tau$
 $\text{then } \perp^{\lceil X \tau \rceil} \neq 0_{\perp}$
 $\text{else } \text{invalid } \tau)$

interpretation $OclAsBoolean_{Real} : \text{profile-mono}_d OclAsBoolean_{Real} \lambda x. \perp^{\lceil x \rceil} \neq 0_{\perp}$
by $\text{unfold-locale} (\text{auto simp: } OclAsBoolean_{Real}\text{-def bot-option-def})$

2.11.2. Definition: asInteger

definition $OclAsInteger_{Real} :: ('\mathfrak{A}) Real \Rightarrow ('\mathfrak{A}) Integer (\langle(-)\rangle \rightarrow oclAsType_{Real}('Integer'))$
where $OclAsInteger_{Real} X = (\lambda\tau. \text{if } (\delta X) \tau = \text{true } \tau$
 $\text{then } \perp^{\lfloor X \tau \rfloor}$
 $\text{else } \text{invalid } \tau)$

interpretation $OclAsInteger_{Real} : \text{profile-mono}_d OclAsInteger_{Real} \lambda x. \perp^{\lfloor x \rfloor}$
by $\text{unfold-locale} (\text{auto simp: } OclAsInteger_{Real}\text{-def bot-option-def})$

2.11.3. Definition: asReal

definition $OclAsReal_{Int} :: ('\mathfrak{A}) Integer \Rightarrow ('\mathfrak{A}) Real (\langle(-)\rangle \rightarrow oclAsType_{Int}('Real'))$
where $OclAsReal_{Int} X = (\lambda\tau. \text{if } (\delta X) \tau = \text{true } \tau$
 $\text{then } \perp^{\text{real-of-int } \lceil X \tau \rceil}$
 $\text{else } \text{invalid } \tau)$

interpretation $OclAsReal_{Int} : \text{profile-mono}_d OclAsReal_{Int} \lambda x. \perp^{\text{real-of-int } \lceil x \rceil}$
by $\text{unfold-locale} (\text{auto simp: } OclAsReal_{Int}\text{-def bot-option-def})$

lemma *Integer-subtype-of-Real:*

assumes $\tau \models \delta X$

shows $\tau \models X \rightarrow oclAsType_{Int}(Real) \rightarrow oclAsType_{Real}(Integer) \triangleq X$

apply (*insert assms, simp add: OclAsInteger_{Real}-def OclAsReal_{Int}-def OclValid-def StrongEq-def*)

apply (*subst (2 4) cp-defined, simp add: true-def*)

by (*metis assms bot-option-def drop.elims foundation16 null-option-def*)

2.11.4. Definition: asPair

definition $OclAsPair_{Seq} :: [('\mathfrak{A}, '\alpha::\text{null}) Sequence] \Rightarrow ('\mathfrak{A}, '\alpha::\text{null}, '\alpha::\text{null}) Pair (\langle(-)\rangle \rightarrow \text{asPair}_{Seq}('))$
where $OclAsPair_{Seq} S = (\text{if } S \rightarrow \text{size}_{Seq}() \doteq 2$
 $\text{then } \text{Pair}\{S \rightarrow \text{at}_{Seq}(\mathbf{0}), S \rightarrow \text{at}_{Seq}(\mathbf{1})\}$

else invalid
endif)

definition $OclAsPair_{Set} :: [(\mathfrak{A}, \alpha :: null) Set] \Rightarrow (\mathfrak{A}, \alpha :: null, \alpha :: null) Pair \ (\langle (-) \rightarrow asPair_{Set} \rangle)$
where $OclAsPair_{Set} S = (if\ S \rightarrow size_{Set}() \doteq 2$
 then let $v = S \rightarrow any_{Set}()$ in
 $Pair\{v, S \rightarrow excluding_{Set}(v) \rightarrow any_{Set}()\}$
 else invalid
 endif)

definition $OclAsPair_{Bag} :: [(\mathfrak{A}, \alpha :: null) Bag] \Rightarrow (\mathfrak{A}, \alpha :: null, \alpha :: null) Pair \ (\langle (-) \rightarrow asPair_{Bag} \rangle)$
where $OclAsPair_{Bag} S = (if\ S \rightarrow size_{Bag}() \doteq 2$
 then let $v = S \rightarrow any_{Bag}()$ in
 $Pair\{v, S \rightarrow excluding_{Bag}(v) \rightarrow any_{Bag}()\}$
 else invalid
 endif)

2.11.5. Definition: asSet

definition $OclAsSet_{Seq} :: [(\mathfrak{A}, \alpha :: null) Sequence] \Rightarrow (\mathfrak{A}, \alpha) Set \ (\langle (-) \rightarrow asSet_{Seq} \rangle)$
where $OclAsSet_{Seq} S = (S \rightarrow iterate_{Seq}(b; x = Set\{\} \mid x \rightarrow including_{Set}(b)))$

definition $OclAsSet_{Pair} :: [(\mathfrak{A}, \alpha :: null, \alpha :: null) Pair] \Rightarrow (\mathfrak{A}, \alpha) Set \ (\langle (-) \rightarrow asSet_{Pair} \rangle)$
where $OclAsSet_{Pair} S = Set\{S.First(), S.Second()\}$

definition $OclAsSet_{Bag} :: (\mathfrak{A}, \alpha :: null) Bag \Rightarrow (\mathfrak{A}, \alpha) Set \ (\langle (-) \rightarrow asSet_{Bag} \rangle)$
where $OclAsSet_{Bag} S = (\lambda \tau. \text{if } (\delta S) \tau = true \ \tau$
 then $Abs\text{-}Set_{base} \ \tau$
 else if $(v S) \tau = true \ \tau$ then $null \ \tau$
 else invalid τ)

2.11.6. Definition: asSequence

definition $OclAsSeq_{Set} :: [(\mathfrak{A}, \alpha :: null) Set] \Rightarrow (\mathfrak{A}, \alpha) Sequence \ (\langle (-) \rightarrow asSequence_{Set} \rangle)$
where $OclAsSeq_{Set} S = (S \rightarrow iterate_{Set}(b; x = Sequence\{\} \mid x \rightarrow including_{Seq}(b)))$

definition $OclAsSeq_{Bag} :: [(\mathfrak{A}, \alpha :: null) Bag] \Rightarrow (\mathfrak{A}, \alpha) Sequence \ (\langle (-) \rightarrow asSequence_{Bag} \rangle)$
where $OclAsSeq_{Bag} S = (S \rightarrow iterate_{Bag}(b; x = Sequence\{\} \mid x \rightarrow including_{Seq}(b)))$

definition $OclAsSeq_{Pair} :: [(\mathfrak{A}, \alpha :: null, \alpha :: null) Pair] \Rightarrow (\mathfrak{A}, \alpha) Sequence \ (\langle (-) \rightarrow asSequence_{Pair} \rangle)$
where $OclAsSeq_{Pair} S = Sequence\{S.First(), S.Second()\}$

2.11.7. Definition: asBag

definition $OclAsBag_{Seq} :: [(\mathfrak{A}, \alpha :: null) Sequence] \Rightarrow (\mathfrak{A}, \alpha) Bag \ (\langle (-) \rightarrow asBag_{Seq} \rangle)$
where $OclAsBag_{Seq} S = (\lambda \tau. Abs\text{-}Bag_{base} \ \tau. \text{if } list\text{-}ex \ ((=) s) \ \tau \ \text{Rep}\text{-}Sequence_{base} \ (S \ \tau) \ \text{then } 1 \ \text{else } 0)$

definition $OclAsBag_{Set} :: [(\mathfrak{A}, \alpha :: null) Set] \Rightarrow (\mathfrak{A}, \alpha) Bag \ (\langle (-) \rightarrow asBag_{Set} \rangle)$
where $OclAsBag_{Set} S = (\lambda \tau. Abs\text{-}Bag_{base} \ \tau. \text{if } s \in \ \tau \ \text{Rep}\text{-}Set_{base} \ (S \ \tau) \ \text{then } 1 \ \text{else } 0)$

lemma assumes $\tau \models \delta (S \rightarrow size_{Set}())$
shows $OclAsBag_{Set} S = (S \rightarrow iterate_{Set}(b; x = Bag\{\} \mid x \rightarrow including_{Bag}(b)))$

oops

definition $OclAsBag_{Pair} :: [(\mathfrak{A}, \alpha :: null, \alpha :: null) Pair] \Rightarrow (\mathfrak{A}, \alpha) Bag \ (\langle (-) \rightarrow asBag_{Pair} \rangle)$
where $OclAsBag_{Pair} S = Bag\{S.First(), S.Second()\}$

2.11.8. Collection Types

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

cp-intro''_{Set}
cp-intro''_{Seq}

2.11.9. Test Statements

lemma *syntax-test*: $Set\{2,1\} = (Set\{\}->including_{Set}(1)->including_{Set}(2))$
 by (*rule refl*)

Here is an example of a nested collection.

lemma *semantic-test2*:

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

shows $(\tau::('A)st) \models (Set\{Set\{2\},null\}->includes_{Set}(null))$

by(*simp add: OclIncludes-execute_{Set} H*)

lemma *short-cut'* [*simp,code-unfold*]: $(8 \doteq 6) = false$

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*]: $(2 \doteq 1) = false$

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*]: $(1 \doteq 2) = false$

apply(*rule ext*)

apply(*simp add: StrictRefEqInteger StrongEq-def OclInt2-def OclInt1-def*
true-def false-def invalid-def bot-option-def)

done

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

Elementary computations on Sets.

declare *OclSelect-body-def* [*simp*]

Assert $\neg (\tau \models v(invalid::('A,'\alpha)::null) Set)$

Assert $\tau \models v(null::('A,'\alpha)::null) Set$

Assert $\neg (\tau \models \delta(null::('A,'\alpha)::null) Set)$

Assert $\tau \models v(Set\{\})$

Assert $\tau \models v(Set\{Set\{2\},null\})$

Assert $\tau \models \delta(Set\{Set\{2\},null\})$

Assert $\tau \models (Set\{2,1\}->includes_{Set}(1))$

Assert $\neg (\tau \models (Set\{2\}->includes_{Set}(1)))$

Assert $\neg (\tau \models (Set\{2,1\}->includes_{Set}(null)))$

Assert $\tau \models (Set\{2,null\}->includes_{Set}(null))$

Assert $\tau \models (Set\{null,2\}->includes_{Set}(null))$

Assert $\tau \models ((Set\{\})->forAll_{Set}(z \mid 0 <_{int} z))$

Assert $\tau \models ((Set\{2,1\})->forAll_{Set}(z \mid 0 <_{int} z))$

Assert $\neg (\tau \models ((\text{Set}\{\mathbf{2},\mathbf{1}\}) \rightarrow \text{exists}_{\text{Set}}(z \mid z <_{\text{int}} \mathbf{0})))$
Assert $\neg (\tau \models (\delta(\text{Set}\{\mathbf{2},\text{null}\}) \rightarrow \text{forAll}_{\text{Set}}(z \mid \mathbf{0} <_{\text{int}} z)))$
Assert $\neg (\tau \models ((\text{Set}\{\mathbf{2},\text{null}\}) \rightarrow \text{forAll}_{\text{Set}}(z \mid \mathbf{0} <_{\text{int}} z)))$
Assert $\tau \models ((\text{Set}\{\mathbf{2},\text{null}\}) \rightarrow \text{exists}_{\text{Set}}(z \mid \mathbf{0} <_{\text{int}} z))$

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

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

lemma *const* $(\text{Set}\{\text{Set}\{\mathbf{2},\text{null}\}, \text{invalid}\})$ **by** $(\text{simp add: const-ss})$

Elementary computations on Sequences.

Assert $\neg (\tau \models v(\text{invalid}::('A,'\alpha::\text{null}) \text{ Sequence}))$
Assert $\tau \models v(\text{null}::('A,'\alpha::\text{null}) \text{ Sequence})$
Assert $\neg (\tau \models \delta(\text{null}::('A,'\alpha::\text{null}) \text{ Sequence}))$
Assert $\tau \models v(\text{Sequence}\{\})$

lemma *const* $(\text{Sequence}\{\text{Sequence}\{\mathbf{2},\text{null}\}, \text{invalid}\})$ **by** $(\text{simp add: const-ss})$

end

3. Formalization III: UML/OCL constructs: State Operations and Objects

```
theory UML-State
imports UML-Library
begin
```

```
no-notation None (⟦⊥⟧)
```

3.1. Introduction: States over Typed Object Universes

In the following, we will refine the concepts of a user-defined data-model (implied by a class-diagram) as well as the notion of state used in the previous section to much more detail. Surprisingly, even without a concrete notion of an objects and a universe of object representation, the generic infrastructure of state-related operations is fairly rich.

3.1.1. Fundamental Properties on Objects: Core Referential Equality

Definition

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

```
definition StrictRefEqObject :: ('A,'a::{object,null})val ⇒ ('A,'a)val ⇒ ('A)Boolean
```

```
where   StrictRefEqObject x y
        ≡ λ τ. if (v x) τ = true τ ∧ (v y) τ = true τ
              then if x τ = null ∨ y τ = null
                    then ⊥ x τ = null ∧ y τ = null ⊥
                    else ⊥(oid-of (x τ)) = (oid-of (y τ)) ⊥
              else invalid τ
```

Strictness and context passing

```
lemma StrictRefEqObject-strict1[simp,code-unfold] :
  (StrictRefEqObject x invalid) = invalid
by(rule ext, simp add: StrictRefEqObject-def true-def false-def)
```

```
lemma StrictRefEqObject-strict2[simp,code-unfold] :
  (StrictRefEqObject invalid x) = invalid
by(rule ext, simp add: StrictRefEqObject-def true-def false-def)
```

```
lemma cp-StrictRefEqObject:
  (StrictRefEqObject x y τ) = (StrictRefEqObject (λ-. x τ) (λ-. y τ)) τ
by(auto simp: StrictRefEqObject-def cp-valid[symmetric])lemmas cp0-StrictRefEqObject=
cp-StrictRefEqObject[THEN all[THEN all[THEN all[THEN cpI2]],
of StrictRefEqObject]]
```

```
lemmas cp-intro''[intro!,simp,code-unfold] =
  cp-intro''
  cp-StrictRefEqObject[THEN all[THEN all[THEN all[THEN cpI2]],
of StrictRefEqObject]]
```

3.1.2. Logic and Algebraic Layer on Object

Validity and Definedness Properties

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

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

lemma *defined-StrictRefEqObject-I*:
assumes *val-x* : $\tau \models v\ x$
assumes *val-x* : $\tau \models v\ y$
shows $\tau \models \delta\ (\text{StrictRefEq}_{\text{Object}}\ x\ y)$
apply(*insert assms, simp add: StrictRefEqObject-def OclValid-def*)
by(*subst cp-defined, simp add: true-def*)

lemma *StrictRefEqObject-def-homo* :
 $\delta(\text{StrictRefEq}_{\text{Object}}\ x\ (y::(\mathfrak{A}, 'a::\{\text{null}, \text{object}\})\text{val})) = ((v\ x)\ \text{and}\ (v\ y))$
oops

Symmetry

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

Behavior vs StrongEq

It remains to clarify the role of the state invariant $\text{inv}_\sigma(\sigma)$ mentioned above that states the condition that there is a “one-to-one” correspondence between object representations and oid’s: $\forall \text{oid} \in \text{dom } \sigma. \text{oid} = \text{OidOf } \lceil \sigma(\text{oid}) \rceil$. This condition is also mentioned in [32, Annex A] and goes back to Richters [33]; however, we state this condition as an invariant on states rather than a global axiom. It can, therefore, not be taken for granted that an oid makes sense both in pre- and post-states of OCL expressions.

We capture this invariant in the predicate WFF :

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

It turns out that WFF is a key-concept for linking strict referential equality to logical equality: in well-formed states (i.e. those states where the self (oid-of) field contains the pointer to which the object is associated to in the state), referential equality coincides with logical equality.

We turn now to the generic definition of referential equality on objects: Equality on objects in a state is reduced to equality on the references to these objects. As in HOL-OCL [6, 8], we will store the reference of an object inside the object in a (ghost) field. By establishing certain invariants (“consistent state”), it can be assured that there is a “one-to-one-correspondence” of objects to their references—and therefore the definition below behaves as we expect.

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

theorem *StrictRefEqObject-vs-StrongEq*:
assumes *WFF*: *WFF* τ
and *valid-x*: $\tau \models (v\ x)$

```

and valid-y:  $\tau \models (v\ y)$ 
and x-present-pre:  $x\ \tau \in \text{ran}\ (\text{heap}(\text{fst}\ \tau))$ 
and y-present-pre:  $y\ \tau \in \text{ran}\ (\text{heap}(\text{fst}\ \tau))$ 
and x-present-post:  $x\ \tau \in \text{ran}\ (\text{heap}(\text{snd}\ \tau))$ 
and y-present-post:  $y\ \tau \in \text{ran}\ (\text{heap}(\text{snd}\ \tau))$ 

shows  $(\tau \models (\text{StrictRefEq}_{\text{Object}}\ x\ y)) = (\tau \models (x \triangleq y))$ 
apply(insert WFF valid-x valid-y x-present-pre y-present-pre x-present-post y-present-post)
apply(auto simp: StrictRefEq_{Object}-def OclValid-def WFF-def StrongEq-def true-def Ball-def)
apply(erule-tac x=x \tau in allE', simp-all)
done

```

```

theorem StrictRefEq_{Object}-vs-StrongEq':
assumes WFF: WFF  $\tau$ 
and valid-x:  $\tau \models (v\ (x :: ('A::\text{object}, 'a::\{\text{null}, \text{object}\})\ \text{val})))$ 
and valid-y:  $\tau \models (v\ y)$ 
and oid-preserve:  $\bigwedge x. x \in \text{ran}\ (\text{heap}(\text{fst}\ \tau)) \vee x \in \text{ran}\ (\text{heap}(\text{snd}\ \tau)) \implies$ 
 $H\ x \neq \perp \implies \text{oid-of}\ (H\ x) = \text{oid-of}\ x$ 
and xy-together:  $x\ \tau \in H\ ' \text{ran}\ (\text{heap}(\text{fst}\ \tau)) \wedge y\ \tau \in H\ ' \text{ran}\ (\text{heap}(\text{fst}\ \tau)) \vee$ 
 $x\ \tau \in H\ ' \text{ran}\ (\text{heap}(\text{snd}\ \tau)) \wedge y\ \tau \in H\ ' \text{ran}\ (\text{heap}(\text{snd}\ \tau))$ 

```

```

shows  $(\tau \models (\text{StrictRefEq}_{\text{Object}}\ x\ y)) = (\tau \models (x \triangleq y))$ 
apply(insert WFF valid-x valid-y xy-together)
apply(simp add: WFF-def)
apply(auto simp: StrictRefEq_{Object}-def OclValid-def WFF-def StrongEq-def true-def Ball-def)
by (metis foundation18' oid-preserve valid-x valid-y)+

```

So, if two object descriptions live in the same state (both pre or post), the referential equality on objects implies in a WFF state the logical equality.

3.2. Operations on Object

3.2.1. Initial States (for testing and code generation)

```

definition  $\tau_0 :: ('A)\ \text{st}$ 
where  $\tau_0 \equiv ((\text{heap} = \text{Map.empty}, \text{assocs} = \text{Map.empty}),$ 
 $(\text{heap} = \text{Map.empty}, \text{assocs} = \text{Map.empty}))$ 

```

3.2.2. OclAllInstances

To denote OCL types occurring in OCL expressions syntactically—as, for example, as “argument” of `oclAllInstances()`—we use the inverses of the injection functions into the object universes; we show that this is a sufficient “characterization.”

```

definition OclAllInstances-generic ::  $((('A)::\text{object})\ \text{st} \Rightarrow 'A\ \text{state}) \Rightarrow ('A::\text{object} \rightarrow 'a) \Rightarrow$ 
 $('A,\ 'a\ \text{option}\ \text{option})\ \text{Set}$ 
where OclAllInstances-generic fst-snd  $H =$ 
 $(\lambda\tau. \text{Abs-Set}_{\text{base}}\ \perp\ \text{Some}\ ' ((H\ ' \text{ran}\ (\text{heap}\ (\text{fst-snd}\ \tau))) - \{\text{None}\})\ \perp)$ 

```

```

lemma OclAllInstances-generic-defined:  $\tau \models \delta\ (\text{OclAllInstances-generic}\ \text{pre-post}\ H)$ 
apply(simp add: defined-def OclValid-def OclAllInstances-generic-def false-def true-def
 $\text{bot-fun-def bot-Set}_{\text{base}}\ \text{-def null-fun-def null-Set}_{\text{base}}\ \text{-def}$ )
apply(rule conjI)
apply(rule notI, subst (asm) Abs-Set}_{\text{base}}\ \text{-inject, simp,
 $(\text{rule disjI2})+$ ,
 $\text{metis bot-option-def option.distinct}(1)$ ,
 $(\text{simp add: bot-option-def null-option-def})+$ )

```

done

lemma *OclAllInstances-generic-init-empty:*

assumes [simp]: $\bigwedge x. \text{pre-post } (x, x) = x$
shows $\tau_0 \models \text{OclAllInstances-generic pre-post } H \triangleq \text{Set}\{\}$
by(simp add: StrongEq-def OclAllInstances-generic-def OclValid-def τ_0 -def mtSet-def)

lemma *represented-generic-objects-nonnull:*

assumes $A: \tau \models ((\text{OclAllInstances-generic pre-post } (H::('A::\text{object} \rightarrow 'A))) \rightarrow \text{includes}_{\text{Set}}(x))$
shows $\tau \models \text{not}(x \triangleq \text{null})$

proof –

have $B: \tau \models \delta (\text{OclAllInstances-generic pre-post } H)$
by (simp add: OclAllInstances-generic-defined)
have $C: \tau \models v \ x$
by (metis OclIncludes.def-valid-then-def
OclIncludes-valid-args-valid A foundation6)

show ?thesis

apply(insert A)

apply(simp add: StrongEq-def OclValid-def
OclNot-def true-def OclIncludes-def B[simplified OclValid-def]
C[simplified OclValid-def])

apply(simp add: OclAllInstances-generic-def)

apply(erule contrapos-pn)

apply(subst Set_{base}.Abs-Set_{base}-inverse,
auto simp: null-fun-def null-option-def bot-option-def)

done

qed

lemma *represented-generic-objects-defined:*

assumes $A: \tau \models ((\text{OclAllInstances-generic pre-post } (H::('A::\text{object} \rightarrow 'A))) \rightarrow \text{includes}_{\text{Set}}(x))$
shows $\tau \models \delta (\text{OclAllInstances-generic pre-post } H) \wedge \tau \models \delta \ x$
by (metis OclAllInstances-generic-defined
A[THEN represented-generic-objects-nonnull] OclIncludes.defined-args-valid
A foundation16' foundation18 foundation24 foundation6)

One way to establish the actual presence of an object representation in a state is:

definition *is-represented-in-state* $\text{fst-snd } x \ H \ \tau = (x \ \tau \in (\text{Some } o \ H) \ ' \ \text{ran } (\text{heap } (\text{fst-snd } \tau)))$

lemma *represented-generic-objects-in-state:*

assumes $A: \tau \models (\text{OclAllInstances-generic pre-post } H) \rightarrow \text{includes}_{\text{Set}}(x)$

shows *is-represented-in-state pre-post* $x \ H \ \tau$

proof –

have $B: (\delta (\text{OclAllInstances-generic pre-post } H)) \ \tau = \text{true } \ \tau$
by(simp add: OclValid-def[symmetric] OclAllInstances-generic-defined)

have $C: (v \ x) \ \tau = \text{true } \ \tau$

by (metis OclValid-def UML-Set.OclIncludes-def assms bot-option-def option.distinct(1) true-def)

have $F: \text{Rep-Set}_{\text{base}} (\text{Abs-Set}_{\text{base}} \ \perp\!\!\!\perp \ \text{Some } ' (H \ ' \ \text{ran } (\text{heap } (\text{pre-post } \tau)) - \{\text{None}\})_{\perp\!\!\!\perp}) =$
 $\perp\!\!\!\perp \ \text{Some } ' (H \ ' \ \text{ran } (\text{heap } (\text{pre-post } \tau)) - \{\text{None}\})_{\perp\!\!\!\perp}$

by(subst Set_{base}.Abs-Set_{base}-inverse,simp-all add: bot-option-def)

show ?thesis

apply(insert A)

apply(simp add: is-represented-in-state-def OclIncludes-def OclValid-def ran-def B C image-def true-def)

apply(simp add: OclAllInstances-generic-def)

apply(simp add: F)

apply(simp add: ran-def)

by(fastforce)

qed

lemma *state-update-vs-allInstances-generic-empty*:
assumes [simp]: $\bigwedge a. \text{pre-post } (mk \ a) = a$
shows $(mk \ (\text{heap} = \text{Map.empty}, \text{assocs} = A)) \models \text{OclAllInstances-generic pre-post Type} \doteq \text{Set}\{\}$
proof –
have *state-update-vs-allInstances-empty*:
 $(\text{OclAllInstances-generic pre-post Type}) (mk \ (\text{heap} = \text{Map.empty}, \text{assocs} = A)) =$
 $\text{Set}\{\} (mk \ (\text{heap} = \text{Map.empty}, \text{assocs} = A))$
by (simp add: *OclAllInstances-generic-def mtSet-def*)
show ?thesis
apply (simp only: *OclValid-def*, subst *StrictRefEqSet.cp0*,
 simp only: *state-update-vs-allInstances-empty StrictRefEqSet.refl-ext*)
apply (simp add: *OclIf-def valid-def mtSet-def defined-def*
bot-fun-def null-fun-def null-option-def bot-Set_base-def)
by (subst *Abs-Set_base-inject*, (simp add: *bot-option-def true-def*)+)
 qed

Here comes a couple of operational rules that allow to infer the value of `oclAllInstances` from the context τ . These rules are a special-case in the sense that they are the only rules that relate statements with *different* τ 's. For that reason, new concepts like “constant contexts P” are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

lemma *state-update-vs-allInstances-generic-including'*:
assumes [simp]: $\bigwedge a. \text{pre-post } (mk \ a) = a$
assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and $\text{Type Object} \neq \text{None}$
shows $(\text{OclAllInstances-generic pre-post Type})$
 $(mk \ (\text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A))$
 $=$
 $((\text{OclAllInstances-generic pre-post Type}) \rightarrow \text{including}_{\text{Set}}(\lambda \cdot \perp \text{ drop } (\text{Type Object}) \perp))$
 $(mk \ (\text{heap} = \sigma', \text{assocs} = A))$

proof –
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: *UML-Set.OclIncluding-def OclAllInstances-generic-defined[simplified OclValid-def]*,
 simp add: *OclAllInstances-generic-def*)
apply (subst *Abs-Set_base-inverse*, simp add: *bot-option-def*, simp add: *comp-def*,
 subst *image-insert[symmetric]*,
 subst *drop-none*, simp add: *assms*)
apply (*case-tac Type Object*, simp add: *assms*, simp only:,
 subst *insert-diff*, *drule sym*, simp)
apply (*subgoal-tac ran* ($\sigma'(\text{oid} \mapsto \text{Object}) = \text{insert Object } (\text{ran } \sigma')$), simp)
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* $\in \text{ran } \sigma'$) **prefer** 2
apply (*rule ranI*, simp)
by (subst *insert-absorb*, simp, *metis fun-upd-apply*)

qed

lemma *state-update-vs-allInstances-generic-including*:
assumes [simp]: $\bigwedge a. \text{pre-post } (mk\ a) = a$
assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and *Type Object \neq None*
shows (*OclAllInstances-generic pre-post Type*)
 $(mk\ (\heaps=\sigma'(\text{oid}\mapsto\text{Object}), \text{assocs}=A))$
 $=$
 $((\lambda\cdot. (\text{OclAllInstances-generic pre-post Type})$
 $(mk\ (\heaps=\sigma', \text{assocs}=A))) \rightarrow \text{including}_{Set}(\lambda\cdot. \perp\ \text{drop } (Type\ \text{Object})\ \perp))$
 $(mk\ (\heaps=\sigma'(\text{oid}\mapsto\text{Object}), \text{assocs}=A))$
apply(*subst state-update-vs-allInstances-generic-including'*, (*simp add: assms*)+,
subst UML-Set.OclIncluding.cp0,
simp add: UML-Set.OclIncluding-def)
apply(*subst (1 3) cp-defined[symmetric]*,
simp add: OclAllInstances-generic-defined[simplified OclValid-def])

apply(*simp add: defined-def OclValid-def OclAllInstances-generic-def invalid-def*
bot-fun-def null-fun-def bot-Set_base-def null-Set_base-def)
apply(*subst (1 3) Abs-Set_base-inject*)
by(*simp add: bot-option-def null-option-def*)+

lemma *state-update-vs-allInstances-generic-noincluding'*:
assumes [simp]: $\bigwedge a. \text{pre-post } (mk\ a) = a$
assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and *Type Object = None*
shows (*OclAllInstances-generic pre-post Type*)
 $(mk\ (\heaps=\sigma'(\text{oid}\mapsto\text{Object}), \text{assocs}=A))$
 $=$
 $(\text{OclAllInstances-generic pre-post Type})$
 $(mk\ (\heaps=\sigma', \text{assocs}=A))$
proof –
have *drop-none* : $\bigwedge x. x \neq \text{None} \implies \lceil x \rceil = x$
by(*case-tac x, simp+*)

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

show ?thesis
apply(*simp add: OclIncluding-def OclAllInstances-generic-defined[simplified OclValid-def]*
OclAllInstances-generic-def)
apply(*subgoal-tac ran* ($\sigma'(\text{oid}\mapsto\text{Object}) = \text{insert } \text{Object } (\text{ran } \sigma')$, *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* $\text{Object} \in \text{ran } \sigma'$) **prefer** 2
apply(*rule ranI, simp*)
apply(*subst insert-absorb, simp*)
by (*metis fun-upd-apply*)
qed

theorem *state-update-vs-allInstances-generic-ntc*:
assumes [simp]: $\bigwedge a. \text{pre-post } (mk\ a) = a$
assumes *oid-def*: $\text{oid} \notin \text{dom } \sigma'$

```

and non-type-conform: Type Object = None
and cp-ctxt: cp P
and const-ctxt:  $\bigwedge X. \text{const } X \implies \text{const } (P X)$ 
shows (mk ( $\langle \text{heap}=\sigma'(\text{oid}\mapsto\text{Object}), \text{assocs}=A \rangle$ ))  $\models P$  (OclAllInstances-generic pre-post Type) =
  (mk ( $\langle \text{heap}=\sigma', \text{assocs}=A \rangle$ ))  $\models P$  (OclAllInstances-generic pre-post Type)
  (is ( $? \tau \models P ? \varphi$ ) = ( $? \tau' \models P ? \varphi$ ))
proof -
have P-cp :  $\bigwedge x \tau. P x \tau = P (\lambda \cdot. x \tau) \tau$ 
  by (metis (full-types) cp-ctxt cp-def)
have A : const (P ( $\lambda \cdot. ? \varphi ? \tau$ ))
  by (simp add: const-ctxt const-ss)
have ( $? \tau \models P ? \varphi$ ) = ( $? \tau \models \lambda \cdot. P ? \varphi ? \tau$ )
  by (subst foundation23, rule refl)
also have ... = ( $? \tau \models \lambda \cdot. P (\lambda \cdot. ? \varphi ? \tau) ? \tau$ )
  by (subst P-cp, rule refl)
also have ... = ( $? \tau' \models \lambda \cdot. P (\lambda \cdot. ? \varphi ? \tau) ? \tau'$ )
  apply (simp add: OclValid-def)
  by (subst A[simplified const-def], subst const-true[simplified const-def], simp)
finally have X: ( $? \tau \models P ? \varphi$ ) = ( $? \tau' \models \lambda \cdot. P (\lambda \cdot. ? \varphi ? \tau) ? \tau'$ )
  by simp
show ?thesis
apply (subst X) apply (subst foundation23[symmetric])
apply (rule StrongEq-L-subst3[OF cp-ctxt])
apply (simp add: OclValid-def StrongEq-def true-def)
apply (rule state-update-vs-allInstances-generic-noincluding')
by (insert oid-def, auto simp: non-type-conform)
qed

```

```

theorem state-update-vs-allInstances-generic-tc:
assumes [simp]:  $\bigwedge a. \text{pre-post } (mk a) = a$ 
assumes oid-def:  $\text{oid} \notin \text{dom } \sigma'$ 
and type-conform: Type Object  $\neq$  None
and cp-ctxt: cp P
and const-ctxt:  $\bigwedge X. \text{const } X \implies \text{const } (P X)$ 
shows (mk ( $\langle \text{heap}=\sigma'(\text{oid}\mapsto\text{Object}), \text{assocs}=A \rangle$ ))  $\models P$  (OclAllInstances-generic pre-post Type) =
  (mk ( $\langle \text{heap}=\sigma', \text{assocs}=A \rangle$ ))  $\models P$  ((OclAllInstances-generic pre-post Type)
     $\rightarrow$  includingSet( $\lambda \cdot. \lfloor (\text{Type Object}) \rfloor$ ))
  (is ( $? \tau \models P ? \varphi$ ) = ( $? \tau' \models P ? \varphi'$ ))
proof -
have P-cp :  $\bigwedge x \tau. P x \tau = P (\lambda \cdot. x \tau) \tau$ 
  by (metis (full-types) cp-ctxt cp-def)
have A : const (P ( $\lambda \cdot. ? \varphi ? \tau$ ))
  by (simp add: const-ctxt const-ss)
have ( $? \tau \models P ? \varphi$ ) = ( $? \tau \models \lambda \cdot. P ? \varphi ? \tau$ )
  by (subst foundation23, rule refl)
also have ... = ( $? \tau \models \lambda \cdot. P (\lambda \cdot. ? \varphi ? \tau) ? \tau$ )
  by (subst P-cp, rule refl)
also have ... = ( $? \tau' \models \lambda \cdot. P (\lambda \cdot. ? \varphi ? \tau) ? \tau'$ )
  apply (simp add: OclValid-def)
  by (subst A[simplified const-def], subst const-true[simplified const-def], simp)
finally have X: ( $? \tau \models P ? \varphi$ ) = ( $? \tau' \models \lambda \cdot. P (\lambda \cdot. ? \varphi ? \tau) ? \tau'$ )
  by simp
let ?allInstances = OclAllInstances-generic pre-post Type
have ?allInstances  $? \tau = \lambda \cdot. ? \text{allInstances } ? \tau' \rightarrow$  includingSet( $\lambda \cdot. \lfloor \text{Type Object} \rfloor$ )  $? \tau$ 
  apply (rule state-update-vs-allInstances-generic-including)
  by (insert oid-def, auto simp: type-conform)
also have ... = (( $\lambda \cdot. ? \text{allInstances } ? \tau'$ )  $\rightarrow$  includingSet( $\lambda \cdot. (\lambda \cdot. \lfloor \text{Type Object} \rfloor)$ )  $? \tau'$ )  $? \tau'$ )

```

```

    by(subst const-OclIncluding[simplified const-def], simp+)
  also have ... = (?allInstances->includingSet(λ -. ⊔Type Object_) ?τ')
    apply(subst UML-Set.OclIncluding.cp0[symmetric])
    by(insert type-conform, auto)
  finally have Y : ?allInstances ?τ = (?allInstances->includingSet(λ -. ⊔Type Object_) ?τ')
    by auto
  show ?thesis
    apply(subst X) apply(subst foundation23[symmetric])
    apply(rule StrongEq-L-subst3[OF cp-ctx])
    apply(simp add: OclValid-def StrongEq-def Y true-def)
  done
qed

```

```
declare OclAllInstances-generic-def [simp]
```

OclAllInstances (@post)

```
definition OclAllInstances-at-post :: ('A :: object → 'α) ⇒ ('A, 'α option option) Set
  (λ<- .allInstances'(>))
```

```
where OclAllInstances-at-post = OclAllInstances-generic snd
```

```
lemma OclAllInstances-at-post-defined: τ ⊨ δ (H .allInstances())
```

```
unfolding OclAllInstances-at-post-def
```

```
by(rule OclAllInstances-generic-defined)
```

```
lemma τ0 ⊨ H .allInstances() ≐ Set{}
```

```
unfolding OclAllInstances-at-post-def
```

```
by(rule OclAllInstances-generic-init-empty, simp)
```

```
lemma represented-at-post-objects-nonnull:
```

```
assumes A: τ ⊨ (((H::('A::object → 'α)).allInstances()) -> includesSet(x))
```

```
shows τ ⊨ not(x ≐ null)
```

```
by(rule represented-generic-objects-nonnull[OF A[simplified OclAllInstances-at-post-def]])
```

```
lemma represented-at-post-objects-defined:
```

```
assumes A: τ ⊨ (((H::('A::object → 'α)).allInstances()) -> includesSet(x))
```

```
shows τ ⊨ δ (H .allInstances()) ∧ τ ⊨ δ x
```

```
unfolding OclAllInstances-at-post-def
```

```
by(rule represented-generic-objects-defined[OF A[simplified OclAllInstances-at-post-def]])
```

One way to establish the actual presence of an object representation in a state is:

```
lemma
```

```
assumes A: τ ⊨ H .allInstances()->includesSet(x)
```

```
shows is-represented-in-state snd x H τ
```

```
by(rule represented-generic-objects-in-state[OF A[simplified OclAllInstances-at-post-def]])
```

```
lemma state-update-vs-allInstances-at-post-empty:
```

```
shows (σ, (heap=Map.empty, assocs=A)) ⊨ Type .allInstances() ≐ Set{}
```

```
unfolding OclAllInstances-at-post-def
```

```
by(rule state-update-vs-allInstances-generic-empty[OF snd-conv])
```

Here comes a couple of operational rules that allow to infer the value of `oclAllInstances` from the context τ . These rules are a special-case in the sense that they are the only rules that relate statements with *different* τ 's. For that reason, new concepts like “constant contexts P” are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

lemma *state-update-vs-allInstances-at-post-including'*:
assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and *Type Object* $\neq \text{None}$
shows (*Type .allInstances*())
 $(\sigma, (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$
 $=$
 $((\text{Type .allInstances}()) \rightarrow \text{including}_{\text{Set}}(\lambda \cdot \perp \text{ drop } (\text{Type } \text{Object}) \perp))$
 $(\sigma, (\text{heap}=\sigma', \text{assocs}=A))$
unfolding *OclAllInstances-at-post-def*
by(*rule state-update-vs-allInstances-generic-including'*[*OF snd-conv*], *insert assms*)

lemma *state-update-vs-allInstances-at-post-including*:
assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and *Type Object* $\neq \text{None}$
shows (*Type .allInstances*())
 $(\sigma, (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$
 $=$
 $((\lambda \cdot (\text{Type .allInstances}()))$
 $(\sigma, (\text{heap}=\sigma', \text{assocs}=A))) \rightarrow \text{including}_{\text{Set}}(\lambda \cdot \perp \text{ drop } (\text{Type } \text{Object}) \perp))$
 $(\sigma, (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$
unfolding *OclAllInstances-at-post-def*
by(*rule state-update-vs-allInstances-generic-including*[*OF snd-conv*], *insert assms*)

lemma *state-update-vs-allInstances-at-post-noincluding'*:
assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and *Type Object* $= \text{None}$
shows (*Type .allInstances*())
 $(\sigma, (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A))$
 $=$
 $(\text{Type .allInstances}())$
 $(\sigma, (\text{heap}=\sigma', \text{assocs}=A))$
unfolding *OclAllInstances-at-post-def*
by(*rule state-update-vs-allInstances-generic-noincluding'*[*OF snd-conv*], *insert assms*)

theorem *state-update-vs-allInstances-at-post-ntc*:
assumes *oid-def*: $\text{oid} \notin \text{dom } \sigma'$
and *non-type-conform*: *Type Object* $= \text{None}$
and *cp-ctxt*: $\text{cp } P$
and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P X)$
shows $((\sigma, (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A)) \models (P(\text{Type .allInstances}()))) =$
 $((\sigma, (\text{heap}=\sigma', \text{assocs}=A)) \models (P(\text{Type .allInstances}())))$
unfolding *OclAllInstances-at-post-def*
by(*rule state-update-vs-allInstances-generic-ntc*[*OF snd-conv*], *insert assms*)

theorem *state-update-vs-allInstances-at-post-tc*:
assumes *oid-def*: $\text{oid} \notin \text{dom } \sigma'$
and *type-conform*: *Type Object* $\neq \text{None}$
and *cp-ctxt*: $\text{cp } P$
and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P X)$
shows $((\sigma, (\text{heap}=\sigma'(\text{oid} \mapsto \text{Object}), \text{assocs}=A)) \models (P(\text{Type .allInstances}()))) =$
 $((\sigma, (\text{heap}=\sigma', \text{assocs}=A)) \models (P((\text{Type .allInstances}())$
 $\rightarrow \text{including}_{\text{Set}}(\lambda \cdot \perp (\text{Type } \text{Object}) \perp))))$
unfolding *OclAllInstances-at-post-def*
by(*rule state-update-vs-allInstances-generic-tc*[*OF snd-conv*], *insert assms*)

OclAllInstances (@pre)

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

where *OclAllInstances-at-pre* = *OclAllInstances-generic fst*

lemma *OclAllInstances-at-pre-defined*: $\tau \models \delta$ (*H* .allInstances@pre())

unfolding *OclAllInstances-at-pre-def*

by(rule *OclAllInstances-generic-defined*)

lemma $\tau_0 \models H$.allInstances@pre() $\hat{=}$ Set{}

unfolding *OclAllInstances-at-pre-def*

by(rule *OclAllInstances-generic-init-empty, simp*)

lemma *represented-at-pre-objects-nonnul*:

assumes *A*: $\tau \models (((H::(' \mathfrak{A}::object \rightarrow ' \alpha)).allInstances@pre()) \rightarrow includes_{Set}(x))$

shows $\tau \models not(x \hat{=} null)$

by(rule *represented-generic-objects-nonnul*[OF *A*[*simplified OclAllInstances-at-pre-def*]])

lemma *represented-at-pre-objects-defined*:

assumes *A*: $\tau \models (((H::(' \mathfrak{A}::object \rightarrow ' \alpha)).allInstances@pre()) \rightarrow includes_{Set}(x))$

shows $\tau \models \delta$ (*H* .allInstances@pre()) \wedge $\tau \models \delta$ *x*

unfolding *OclAllInstances-at-pre-def*

by(rule *represented-generic-objects-defined*[OF *A*[*simplified OclAllInstances-at-pre-def*]])

One way to establish the actual presence of an object representation in a state is:

lemma

assumes *A*: $\tau \models H$.allInstances@pre() $\rightarrow includes_{Set}(x)$

shows *is-represented-in-state fst x H* τ

by(rule *represented-generic-objects-in-state*[OF *A*[*simplified OclAllInstances-at-pre-def*]])

lemma *state-update-vs-allInstances-at-pre-empty*:

shows ((*heap*=Map.empty, *assocs*=*A*), σ) \models *Type* .allInstances@pre() $\hat{=} Set\{\}$

unfolding *OclAllInstances-at-pre-def*

by(rule *state-update-vs-allInstances-generic-empty*[OF *fst-conv*])

Here comes a couple of operational rules that allow to infer the value of oclAllInstances@pre from the context τ . These rules are a special-case in the sense that they are the only rules that relate statements with *different* τ 's. For that reason, new concepts like "constant contexts P" are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

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

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

and *Type Object* $\neq None$

shows (*Type* .allInstances@pre())

((*heap*= $\sigma'(oid \mapsto Object)$, *assocs*=*A*), σ)

=

((*Type* .allInstances@pre()) $\rightarrow includes_{Set}(\lambda _ . \perp \perp drop (Type\ Object) \perp \perp)$)

((*heap*= σ' , *assocs*=*A*), σ)

unfolding *OclAllInstances-at-pre-def*

by(rule *state-update-vs-allInstances-generic-including'*[OF *fst-conv*], *insert assms*)

lemma *state-update-vs-allInstances-at-pre-including*:

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

and *Type Object* \neq *None*
shows (*Type* .allInstances@pre())
 ((heap= σ' (oid \mapsto Object), assoc=*A*), σ)
 =
 ((λ -. (*Type* .allInstances@pre())
 ((heap= σ' , assoc=*A*), σ)) \rightarrow including_{Set}(λ -. \perp drop (*Type* *Object*) \perp))
 ((heap= σ' (oid \mapsto Object), assoc=*A*), σ)
unfolding *OclAllInstances-at-pre-def*
by(rule *state-update-vs-allInstances-generic-including*[*OF fst-conv*], insert *assms*)

lemma *state-update-vs-allInstances-at-pre-noincluding'*:
assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and *Type Object* = *None*
shows (*Type* .allInstances@pre())
 ((heap= σ' (oid \mapsto Object), assoc=*A*), σ)
 =
 (*Type* .allInstances@pre())
 ((heap= σ' , assoc=*A*), σ)
unfolding *OclAllInstances-at-pre-def*
by(rule *state-update-vs-allInstances-generic-noincluding'*[*OF fst-conv*], insert *assms*)

theorem *state-update-vs-allInstances-at-pre-ntc*:
assumes *oid-def*: oid \notin dom σ'
and *non-type-conform*: *Type Object* = *None*
and *cp-ctxt*: cp *P*
and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P X)$
shows (((heap= σ' (oid \mapsto Object), assoc=*A*), σ) \models (*P* (*Type* .allInstances@pre()))) =
 ((heap= σ' , assoc=*A*), σ) \models (*P* (*Type* .allInstances@pre()))
unfolding *OclAllInstances-at-pre-def*
by(rule *state-update-vs-allInstances-generic-ntc*[*OF fst-conv*], insert *assms*)

theorem *state-update-vs-allInstances-at-pre-tc*:
assumes *oid-def*: oid \notin dom σ'
and *type-conform*: *Type Object* \neq *None*
and *cp-ctxt*: cp *P*
and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P X)$
shows (((heap= σ' (oid \mapsto Object), assoc=*A*), σ) \models (*P* (*Type* .allInstances@pre()))) =
 ((heap= σ' , assoc=*A*), σ) \models (*P* ((*Type* .allInstances@pre()
 \rightarrow including_{Set}(λ -. \perp (*Type* *Object*) \perp))))
unfolding *OclAllInstances-at-pre-def*
by(rule *state-update-vs-allInstances-generic-tc*[*OF fst-conv*], insert *assms*)

@post or @pre

theorem *StrictRefEqObject-vs-StrongEq''*:
assumes *WFF*: *WFF* τ
and *valid-x*: $\tau \models (v (x :: ('A::\text{object}, 'a::\text{object option option}) \text{val}))$
and *valid-y*: $\tau \models (v y)$
and *oid-preserve*: $\bigwedge x. x \in \text{ran } (\text{heap}(\text{fst } \tau)) \vee x \in \text{ran } (\text{heap}(\text{snd } \tau)) \implies$
 oid-of (*H* *x*) = oid-of *x*
and *xy-together*: $\tau \models ((H \text{ .allInstances}() \rightarrow \text{includes}_{\text{Set}}(x) \text{ and } H \text{ .allInstances}() \rightarrow \text{includes}_{\text{Set}}(y)) \text{ or}$
 (*H* .allInstances@pre() \rightarrow includes_{Set}(*x*) and *H* .allInstances@pre() \rightarrow includes_{Set}(*y*))
shows ($\tau \models (\text{StrictRefEqObject } x y)$) = ($\tau \models (x \triangleq y)$)
proof –
have *at-post-def* : $\bigwedge x. \tau \models v x \implies \tau \models \delta (H \text{ .allInstances}() \rightarrow \text{includes}_{\text{Set}}(x))$

```

apply(simp add: OclIncludes-def OclValid-def
        OclAllInstances-at-post-defined[simplified OclValid-def])
by(subst cp-defined, simp)
have at-pre-def :  $\bigwedge x. \tau \models v \ x \implies \tau \models \delta \ (H \ .allInstances@pre() \rightarrow includes_{Set}(x))$ 
apply(simp add: OclIncludes-def OclValid-def
        OclAllInstances-at-pre-defined[simplified OclValid-def])
by(subst cp-defined, simp)
have F: Rep-Setbase (Abs-Setbase  $\perp$ Some ‘(H ‘ ran (heap (fst  $\tau$ )) - {None}) $\perp$ ) =
         $\perp$ Some ‘(H ‘ ran (heap (fst  $\tau$ )) - {None}) $\perp$ 
        by(subst Setbase.Abs-Setbase-inverse,simp-all add: bot-option-def)
have F': Rep-Setbase (Abs-Setbase  $\perp$ Some ‘(H ‘ ran (heap (snd  $\tau$ )) - {None}) $\perp$ ) =
         $\perp$ Some ‘(H ‘ ran (heap (snd  $\tau$ )) - {None}) $\perp$ 
        by(subst Setbase.Abs-Setbase-inverse,simp-all add: bot-option-def)
show ?thesis
apply(rule StrictRefEqObject-vs-StrongEq[OF WFF valid-x valid-y, where H = Some o H])
apply(subst oid-preserve[symmetric], simp, simp add: oid-of-option-def)
apply(insert xy-together,
        subst (asm) foundation11,
        metis at-post-def defined-and-I valid-x valid-y,
        metis at-pre-def defined-and-I valid-x valid-y)
apply(erule disjE)
by(drule foundation5,
        simp add: OclAllInstances-at-pre-def OclAllInstances-at-post-def
        OclValid-def OclIncludes-def true-def F F'
        valid-x[simplified OclValid-def] valid-y[simplified OclValid-def] bot-option-def
        split: if-split-asm,
        simp add: comp-def image-def, fastforce)+
qed

```

3.2.3. OclIsNew, OclIsDeleted, OclIsMaintained, OclIsAbsent

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

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

definition *OclIsDeleted*:: ($\mathfrak{A}, 'a::\{null,object\}$)val \Rightarrow (\mathfrak{A})Boolean ($\langle(-).oclIsDeleted'(\cdot)\rangle$)
where $X \ .oclIsDeleted() \equiv (\lambda\tau \ . \text{if } (\delta \ X) \ \tau = \text{true } \tau$
 then $\perp oid\text{-of } (X \ \tau) \in \text{dom}(\text{heap}(\text{fst } \tau)) \wedge$
 $oid\text{-of } (X \ \tau) \notin \text{dom}(\text{heap}(\text{snd } \tau))\perp$
 else *invalid* τ)

definition *OclIsMaintained*:: ($\mathfrak{A}, 'a::\{null,object\}$)val \Rightarrow (\mathfrak{A})Boolean($\langle(-).oclIsMaintained'(\cdot)\rangle$)
where $X \ .oclIsMaintained() \equiv (\lambda\tau \ . \text{if } (\delta \ X) \ \tau = \text{true } \tau$
 then $\perp oid\text{-of } (X \ \tau) \in \text{dom}(\text{heap}(\text{fst } \tau)) \wedge$
 $oid\text{-of } (X \ \tau) \in \text{dom}(\text{heap}(\text{snd } \tau))\perp$
 else *invalid* τ)

definition *OclIsAbsent*:: ($\mathfrak{A}, 'a::\{null,object\}$)val \Rightarrow (\mathfrak{A})Boolean ($\langle(-).oclIsAbsent'(\cdot)\rangle$)
where $X \ .oclIsAbsent() \equiv (\lambda\tau \ . \text{if } (\delta \ X) \ \tau = \text{true } \tau$
 then $\perp oid\text{-of } (X \ \tau) \notin \text{dom}(\text{heap}(\text{fst } \tau)) \wedge$
 $oid\text{-of } (X \ \tau) \notin \text{dom}(\text{heap}(\text{snd } \tau))\perp$
 else *invalid* τ)

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

lemma *notNew-vs-others* : $\tau \models \delta X \implies$
 $(\neg \tau \models (X .oclIsNew())) = (\tau \models (X .oclIsDeleted()) \vee$
 $\tau \models (X .oclIsMaintained()) \vee \tau \models (X .oclIsAbsent()))$
by(*simp add: OclIsDeleted-def OclIsNew-def OclIsMaintained-def OclIsAbsent-def*
OclNot-def OclValid-def true-def, blast)

3.2.4. OclIsModifiedOnly

Definition

The following predicate—which is not part of the OCL standard—provides a simple, but powerful means to describe framing conditions. For any formal approach, be it animation of OCL contracts, test-case generation or die-hard theorem proving, the specification of the part of a system transition that *does not change* is of primordial importance. The following operator establishes the equality between old and new objects in the state (provided that they exist in both states), with the exception of those objects.

definition *OclIsModifiedOnly* :: (' \mathcal{A} ::object, ' α ::{null,object})Set \Rightarrow ' \mathcal{A} Boolean
 $(\langle _ \rangle \rightarrow oclIsModifiedOnly')$

where $X \rightarrow oclIsModifiedOnly() \equiv (\lambda(\sigma, \sigma')$
 $let X' = (oid-of \langle _ \rangle^{\top} Rep-Set_{base}(X(\sigma, \sigma')^{\top}))$;
 $S = ((dom(heap \sigma) \cap dom(heap \sigma')) - X')$
 $in if (\delta X) (\sigma, \sigma') = true (\sigma, \sigma') \wedge (\forall x \in \langle _ \rangle^{\top} Rep-Set_{base}(X(\sigma, \sigma')^{\top}). x \neq null)$
 $then \sqcup \forall x \in S. (heap \sigma) x = (heap \sigma') x_{\sqcup}$
 $else invalid (\sigma, \sigma'))$

Execution with Invalid or Null or Null Element as Argument

lemma *invalid* $\rightarrow oclIsModifiedOnly() = invalid$
by(*simp add: OclIsModifiedOnly-def*)

lemma *null* $\rightarrow oclIsModifiedOnly() = invalid$
by(*simp add: OclIsModifiedOnly-def*)

lemma

assumes *X-null* : $\tau \models X \rightarrow includes_{Set}(null)$
shows $\tau \models X \rightarrow oclIsModifiedOnly() \triangleq invalid$
apply(*insert X-null,*
simp add : OclIncludes-def OclIsModifiedOnly-def StrongEq-def OclValid-def true-def)
apply(*case-tac* τ , *simp*)
apply(*simp split: if-split-asm*)
by(*simp add: null-fun-def, blast*)

Context Passing

lemma *cp-OclIsModifiedOnly* : $X \rightarrow oclIsModifiedOnly() \tau = (\lambda \cdot X \tau) \rightarrow oclIsModifiedOnly() \tau$
by(*simp only: OclIsModifiedOnly-def, case-tac* τ , *simp only:, subst cp-defined, simp*)

3.2.5. OclSelf

The following predicate—which is not part of the OCL standard—explicitly retrieves in the pre or post state the original OCL expression given as argument.

definition [*simp*]: *OclSelf* $x H fst-snd = (\lambda \tau . if (\delta x) \tau = true \tau$

then if $oid-of (x \tau) \in dom(heap(fst \tau)) \wedge oid-of (x \tau) \in dom(heap (snd \tau))$
 then $H \uparrow(heap(fst-snd \tau))(oid-of (x \tau))^\uparrow$
 else $invalid \tau$
 else $invalid \tau$)

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

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

3.2.6. Framing Theorem

lemma *all-oid-diff*:

assumes $def-x : \tau \models \delta x$

assumes $def-X : \tau \models \delta X$

assumes $def-X' : \bigwedge x. x \in {}^\uparrow Rep-Set_{base} (X \tau)^\uparrow \implies x \neq null$

defines $P \equiv (\lambda a. not (StrictRefEq_{Object} x a))$

shows $(\tau \models X \rightarrow forAll_{Set}(a) P a) = (oid-of (x \tau) \notin oid-of {}^\uparrow Rep-Set_{base} (X \tau)^\uparrow)$

proof –

have $P-null-bot: \bigwedge b. b = null \vee b = \perp \implies$
 $\neg (\exists x \in {}^\uparrow Rep-Set_{base} (X \tau)^\uparrow. P (\lambda :: 'a state \times 'a state). x) \tau = b \tau)$

apply(*erule disjE*)

apply(*simp, rule ballI*,

simp add: P-def StrictRefEq_{Object}-def, rename-tac x',

subst cp-OclNot, simp,

subgoal-tac x \tau \neq null \wedge x' \neq null, simp,

simp add: OclNot-def null-fun-def null-option-def bot-option-def bot-fun-def invalid-def,

(metis def-X' def-x foundation16[THEN iffD1]

| (metis bot-fun-def OclValid-def Set-inv-lemma def-X def-x defined-def valid-def,

metis def-X' def-x foundation16[THEN iffD1])))+

done

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

by (*metis foundation21 foundation22*)

have $P-false : \exists x \in {}^\uparrow Rep-Set_{base} (X \tau)^\uparrow. P (\lambda \cdot. x) \tau = false \tau \implies$
 $oid-of (x \tau) \in oid-of {}^\uparrow Rep-Set_{base} (X \tau)^\uparrow$

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: if-split-asm*)

apply(*subgoal-tac x \tau \neq null \wedge x' \neq null, simp*)

apply (*metis (mono-tags) drop.simps def-x foundation16[THEN iffD1] true-def*)

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

have $P-true : \forall x \in {}^\uparrow Rep-Set_{base} (X \tau)^\uparrow. P (\lambda \cdot. x) \tau = true \tau \implies$

$oid-of (x \tau) \notin oid-of {}^\uparrow Rep-Set_{base} (X \tau)^\uparrow$

apply(*subgoal-tac \forall x' \in {}^\uparrow Rep-Set_{base} (X \tau)^\uparrow. oid-of x' \neq oid-of (x \tau)*)

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: if-split-asm)
  apply(subgoal-tac  $x \tau \neq \text{null} \wedge x' \neq \text{null}$ , simp)
  apply (metis def-X' def-x foundation16[THEN iffD1])
by(simp add: invalid-def bot-option-def false-def)+

have bool-split :  $\forall x \in {}^\top \text{Rep-Set}_{\text{base}} (X \tau)^\top. P (\lambda \cdot x) \tau \neq \text{null} \tau \implies$ 
   $\forall x \in {}^\top \text{Rep-Set}_{\text{base}} (X \tau)^\top. P (\lambda \cdot x) \tau \neq \perp \tau \implies$ 
   $\forall x \in {}^\top \text{Rep-Set}_{\text{base}} (X \tau)^\top. P (\lambda \cdot x) \tau \neq \text{false} \tau \implies$ 
   $\forall x \in {}^\top \text{Rep-Set}_{\text{base}} (X \tau)^\top. P (\lambda \cdot x) \tau = \text{true} \tau$ 
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) bot-fun-def OclNot4 OclValid-def foundation16
  foundation9 not-inj null-fun-def)
by(fast+)

show ?thesis
apply(subst OclForall-rep-set-true[OF def-X], simp add: OclValid-def)
apply(rule iffI, simp add: P-true)
by (metis P-false P-null-bot bool-split)
qed

theorem framing:
  assumes modifiesclause: $\tau \models (X \rightarrow \text{excluding}_{\text{Set}}(x)) \rightarrow \text{oclIsModifiedOnly}()$ 
  and oid-is-typerepr :  $\tau \models X \rightarrow \text{forAll}_{\text{Set}}(a \mid \text{not } (\text{StrictRefEqObject } x a))$ 
  shows  $\tau \models (x \text{ @pre } P \triangleq (x \text{ @post } P))$ 
apply(case-tac  $\tau \models \delta x$ )
proof - show  $\tau \models \delta x \implies ?thesis$  proof - assume def-x :  $\tau \models \delta x$  show ?thesis proof -

have def-X :  $\tau \models \delta X$ 
apply(insert oid-is-typerepr, simp add: UML-Set.OclForall-def OclValid-def split: if-split-asm)
by(simp add: bot-option-def true-def)

have def-X' :  $\bigwedge x. x \in {}^\top \text{Rep-Set}_{\text{base}} (X \tau)^\top \implies x \neq \text{null}$ 
apply(insert modifiesclause, simp add: OclIsModifiedOnly-def OclValid-def split: if-split-asm)
apply(case-tac  $\tau$ , simp split: if-split-asm)
apply(simp add: UML-Set.OclExcluding-def split: if-split-asm)
apply(subst (asm) (2) Abs-Setbase-inverse)
apply(simp, (rule disjI2)+)
apply (metis (opaque-lifting, 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 (opaque-lifting, no-types) def-x foundation16[THEN iffD1])
apply (metis (opaque-lifting, no-types) OclValid-def def-X def-x foundation20
  OclExcluding-valid-args-valid OclExcluding-valid-args-valid')
by(simp add: invalid-def bot-option-def)

have oid-is-typerepr : oid-of  $(x \tau) \notin \text{oid-of } ({}^\top \text{Rep-Set}_{\text{base}} (X \tau)^\top)$ 
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 =  $\lambda x. P \overline{x}$  in arg-cong)
apply(insert modifiesclause[simplified OclIsModifiedOnly-def OclValid-def])
apply(case-tac  $\tau$ , rename-tac  $\sigma \sigma'$ , simp split: if-split-asm)
apply(subst (asm) (2) UML-Set.OclExcluding-def)
apply(drule foundation5[simplified OclValid-def true-def], simp)
apply(subst (asm) Abs-Setbase-inverse, simp)
apply(rule disjI2)+
apply (metis (opaque-lifting, no-types) DiffD1 OclValid-def Set-inv-lemma def-x
        foundation16 foundation18')
apply(simp)
apply(erule-tac x = oid-of (x ( $\sigma$ ,  $\sigma'$ )) in ballE) apply simp+
apply (metis (opaque-lifting, no-types)
        DiffD1 image-iff image-insert insert-Diff-single insert-absorb oid-is-typerrepr)
apply(simp add: invalid-def bot-option-def)+
by blast
qed qed
qed(simp add: OclSelf-at-post-def OclSelf-at-pre-def OclValid-def StrongEq-def true-def)+

```

As corollary, the framing property can be expressed with only the strong equality as comparison operator.

```

theorem framing':
  assumes wff : WFF  $\tau$ 
  assumes modifiesclause: $\tau \models (X \rightarrow \text{excluding}_{Set}(x)) \rightarrow \text{oclIsModifiedOnly}()$ 
  and oid-is-typerrepr :  $\tau \models X \rightarrow \text{forAll}_{Set}(a \mid \text{not } (x \triangleq a))$ 
  and oid-preserve:  $\bigwedge x. x \in \text{ran } (\text{heap}(\text{fst } \tau)) \vee x \in \text{ran } (\text{heap}(\text{snd } \tau)) \implies$ 
     $\text{oid-of } (H x) = \text{oid-of } x$ 
  and xy-together:
     $\tau \models X \rightarrow \text{forAll}_{Set}(y \mid (H .\text{allInstances}() \rightarrow \text{includes}_{Set}(x) \text{ and } H .\text{allInstances}() \rightarrow \text{includes}_{Set}(y)) \text{ or}$ 
     $(H .\text{allInstances}@pre() \rightarrow \text{includes}_{Set}(x) \text{ and } H .\text{allInstances}@pre() \rightarrow \text{includes}_{Set}(y)))$ 
  shows  $\tau \models (x @pre P \triangleq (x @post P))$ 
proof -
  have def-X :  $\tau \models \delta X$ 
  apply(insert oid-is-typerrepr, simp add: UML-Set.OclForall-def OclValid-def split: if-split-asm)
by(simp add: bot-option-def true-def)
show ?thesis
  apply(case-tac  $\tau \models \delta x$ , drule foundation20)
  apply(rule framing[OF modifiesclause])
  apply(rule OclForall-cong'[OF - oid-is-typerrepr xy-together], rename-tac y)
  apply(cut-tac Set-inv-lemma'[OF def-X]) prefer 2 apply assumption
  apply(rule OclNot-contrapos-nn, simp add: StrictRefEqObject-def)
  apply(simp add: OclValid-def, subst cp-defined, simp,
    assumption)
  apply(rule StrictRefEqObject-vs-StrongEq'[THEN iffD1, OF wff - - oid-preserve], assumption+)
by(simp add: OclSelf-at-post-def OclSelf-at-pre-def OclValid-def StrongEq-def true-def)+
qed

```

3.2.7. Miscellaneous

```

lemma pre-post-new:  $\tau \models (x .\text{oclIsNew}()) \implies \neg (\tau \models v(x @pre H1)) \wedge \neg (\tau \models v(x @post H2))$ 
by(simp add: OclIsNew-def OclSelf-at-pre-def OclSelf-at-post-def
  OclValid-def StrongEq-def true-def false-def
  bot-option-def invalid-def bot-fun-def valid-def
  split: if-split-asm)

```

```

lemma pre-post-old:  $\tau \models (x .\text{oclIsDeleted}()) \implies \neg (\tau \models v(x @pre H1)) \wedge \neg (\tau \models v(x @post H2))$ 
by(simp add: OclIsDeleted-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: if-split-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: if-split-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: if-split-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: if-split-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*)

3.3. Accessors on Object

3.3.1. Definition

definition *select-object mt incl smash deref l = 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 *UML-Sequence.OclANY-selector* which also handles the *null*-cases appropriately. A standard use-case for this combinator is for example:

term (*select-object mtSet UML-Set.OclIncluding UML-Set.OclANY f l oid*)::('A, 'a::null)val

definition *select-object_{Set} = select-object mtSet UML-Set.OclIncluding id*

definition *select-object-any0_{Set} f s-set = UML-Set.OclANY (select-object_{Set} f s-set)*

definition *select-object-any_{Set} f s-set =*

(*let s = select-object_{Set} f s-set in*

if s->size_{Set}() \triangleq 1 then

s->any_{Set}()

else

⊥

endif)

definition *select-object_{Seq} = select-object mtSequence UML-Sequence.OclIncluding id*

definition *select-object-any_{Seq} f s-set = UML-Sequence.OclANY (select-object_{Seq} f s-set)*

definition *select-object_{Pair} f1 f2 = ($\lambda(a,b). OclPair (f1 a) (f2 b)$)*

3.3.2. Validity and Definedness Properties

lemma *select-fold-exec_{Seq}*:

assumes *list-all ($\lambda f. (\tau \models v f)$) l*

shows $\ulcorner \text{Rep-Sequence}_{base} (\text{foldl } UML-Sequence.OclIncluding \text{Sequence}\{ \} l \tau) \urcorner = \text{List.map } (\lambda f. f \tau) l$

proof —

have *def-fold*: $\bigwedge l. \text{list-all } (\lambda f. \tau \models v f) l \implies$
 $\tau \models (\delta \text{ foldl } \text{UML-Sequence.OclIncluding Sequence}\{ \} l)$
apply(*rule rev-induct*[**where** $P = \lambda l. \text{list-all } (\lambda f. (\tau \models v f)) l \longrightarrow \tau \models (\delta \text{ foldl } \text{UML-Sequence.OclIncluding Sequence}\{ \} l)$, *THEN mp*], *simp*)
by(*simp add: foundation10'*)
show *?thesis*
apply(*rule rev-induct*[**where** $P = \lambda l. \text{list-all } (\lambda f. (\tau \models v f)) l \longrightarrow \ulcorner \text{Rep-Sequence}_{base} \text{ (foldl UML-Sequence.OclIncluding Sequence}\{ \} l \tau) \urcorner = \text{List.map } (\lambda f. f \tau) l$, *THEN mp*], *simp*)
apply(*simp add: mtSequence-def*)
apply(*subst Abs-Sequence_{base}-inverse*, (*simp | intro impI*) $+$)
apply(*simp add: UML-Sequence.OclIncluding-def*, *intro conjI impI*)
apply(*subst Abs-Sequence_{base}-inverse*, *simp*, (*rule disjI2*) $+$)
apply(*simp add: list-all-iff foundation18'*, *simp*)
apply(*subst (asm) def-fold[simplified (no-asm) OclValid-def]*, *simp*, *simp add: OclValid-def*)
by (*rule assms*)
qed

lemma *select-fold-exec_{Set}*:

assumes *list-all* $(\lambda f. \tau \models v f) l$

shows $\ulcorner \text{Rep-Set}_{base} \text{ (foldl UML-Set.OclIncluding Set}\{ \} l \tau) \urcorner = \text{set } (\text{List.map } (\lambda f. f \tau) l)$

proof –

have *def-fold*: $\bigwedge l. \text{list-all } (\lambda f. \tau \models v f) l \implies$

$\tau \models (\delta \text{ foldl } \text{UML-Set.OclIncluding Set}\{ \} l)$

apply(*rule rev-induct*[**where** $P = \lambda l. \text{list-all } (\lambda f. (\tau \models v f)) l \longrightarrow \tau \models (\delta \text{ foldl } \text{UML-Set.OclIncluding Set}\{ \} l)$, *THEN mp*], *simp*)

by(*simp add: foundation10'*)

show *?thesis*

apply(*rule rev-induct*[**where** $P = \lambda l. \text{list-all } (\lambda f. (\tau \models v f)) l \longrightarrow \ulcorner \text{Rep-Set}_{base} \text{ (foldl UML-Set.OclIncluding Set}\{ \} l \tau) \urcorner = \text{set } (\text{List.map } (\lambda f. f \tau) l)$, *THEN mp*], *simp*)

apply(*simp add: mtSet-def*)

apply(*subst Abs-Set_{base}-inverse*, (*simp | intro impI*) $+$)

apply(*simp add: UML-Set.OclIncluding-def*, *intro conjI impI*)

apply(*subst Abs-Set_{base}-inverse*, *simp*, (*rule disjI2*) $+$)

apply(*simp add: list-all-iff foundation18'*, *simp*)

apply(*subst (asm) def-fold[simplified (no-asm) OclValid-def]*, *simp*, *simp add: OclValid-def*)

by (*rule assms*)

qed

lemma *fold-val-elim_{Seq}*:

assumes $\tau \models v \text{ (foldl UML-Sequence.OclIncluding Sequence}\{ \} (\text{List.map } (f p) s\text{-set}))$

shows *list-all* $(\lambda x. \tau \models v (f p x)) s\text{-set}$

apply(*rule rev-induct*[**where** $P = \lambda s\text{-set}. \tau \models v \text{ foldl UML-Sequence.OclIncluding Sequence}\{ \} (\text{List.map } (f p) s\text{-set}) \longrightarrow \text{list-all } (\lambda x. \tau \models v f p x) s\text{-set}$, *THEN mp*])

apply(*simp, auto*)

apply (*metis (opaque-lifting, mono-tags) UML-Sequence.OclIncluding.def-valid-then-def UML-Sequence.OclIncluding.defined-args-valid foundation20*) $+$

by(*simp add: assms*)

lemma *fold-val-elim_{Set}*:

assumes $\tau \models v \text{ (foldl UML-Set.OclIncluding Set}\{ \} (\text{List.map } (f p) s\text{-set}))$

shows *list-all* $(\lambda x. \tau \models v (f p x)) s\text{-set}$

apply(*rule rev-induct*[**where** $P = \lambda s\text{-set}. \tau \models v \text{ foldl UML-Set.OclIncluding Set}\{ \} (\text{List.map } (f p) s\text{-set}) \longrightarrow \text{list-all } (\lambda x. \tau \models v f p x) s\text{-set}$, *THEN mp*])

apply(*simp, auto*)

apply (*metis (opaque-lifting, mono-tags) foundation10' foundation20*) $+$
by(*simp add: assms*)

lemma *select-object-any-defined_{Seq}*:
assumes *def-sel*: $\tau \models \delta$ (*select-object-any_{Seq}* *f s-set*)
shows *s-set* $\neq \square$
apply(*insert def-sel, case-tac s-set*)
apply(*simp add: select-object-any_{Seq}-def UML-Sequence.OclANY-def select-object_{Seq}-def select-object-def defined-def OclValid-def false-def true-def bot-fun-def bot-option-def split: if-split-asm*)
apply(*simp add: mtSequence-def, subst (asm) Abs-Sequence_{base}-inverse, simp, simp*)
by(*simp*)

lemma
assumes *def-sel*: $\tau \models \delta$ (*select-object-any0_{Set}* *f s-set*)
shows *s-set* $\neq \square$
apply(*insert def-sel, case-tac s-set*)
apply(*simp add: select-object-any0_{Set}-def UML-Sequence.OclANY-def select-object_{Set}-def select-object-def defined-def OclValid-def false-def true-def bot-fun-def bot-option-def split: if-split-asm*)
by(*simp*)

lemma *select-object-any-defined_{Set}*:
assumes *def-sel*: $\tau \models \delta$ (*select-object-any_{Set}* *f s-set*)
shows *s-set* $\neq \square$
apply(*insert def-sel, case-tac s-set*)
apply(*simp add: select-object-any_{Set}-def UML-Sequence.OclANY-def select-object_{Set}-def select-object-def defined-def OclValid-def false-def true-def bot-fun-def bot-option-def OclInt0-def OclInt1-def StrongEq-def OclIf-def null-fun-def null-option-def split: if-split-asm*)
by(*simp*)

lemma *select-object-any-exec0_{Seq}*:
assumes *def-sel*: $\tau \models \delta$ (*select-object-any_{Seq}* *f s-set*)
shows $\tau \models$ (*select-object-any_{Seq}* *f s-set* \triangleq *f (hd s-set)*)
apply(*insert def-sel[simplified foundation16], simp add: select-object-any_{Seq}-def foundation22 UML-Sequence.OclANY-def split: if-split-asm*)
apply(*case-tac \top Rep-Sequence_{base} (select-object_{Seq} *f s-set* \top) \top , simp add: bot-option-def, simp*)
apply(*simp add: select-object_{Seq}-def select-object-def*)
apply(*subst (asm) select-fold-exec_{Seq}*)
apply(*rule fold-val-elem_{Seq}, simp add: foundation18' invalid-def*)
apply(*simp*)
by(*drule arg-cong[where $f = hd$], subst (asm) hd-map, simp add: select-object-any-defined_{Seq}[OF *def-sel*], simp*)

lemma *select-object-any-exec_{Seq}*:
assumes *def-sel*: $\tau \models \delta$ (*select-object-any_{Seq}* *f s-set*)
shows $\exists e. \text{List.member } s\text{-set } e \wedge (\tau \models (\text{select-object-any_{Seq}} *f s-set* \triangleq *f e*))
apply(*insert select-object-any-exec0_{Seq}[OF *def-sel*]*)
apply(*rule exI[where $x = hd s-set$], simp*)
apply(*case-tac s-set, simp add: select-object-any-defined_{Seq}[OF *def-sel*]*)
apply *simp*
done$

lemma *select-object-any-exec_{Set}*:
assumes *def-sel*: $\tau \models \delta$ (*select-object-any_{Set}* *f s-set*)
shows $\exists e. \text{List.member } s\text{-set } e \wedge (\tau \models (\text{select-object-any_{Set}} *f s-set* \triangleq *f e*))$

```

proof –
have card-singl:  $\bigwedge A a. \text{finite } A \implies \text{card } (\text{insert } a \ A) = 1 \implies A \subseteq \{a\}$ 
by (auto, metis Suc-inject card-Suc-eq card-eq-0-iff insert-absorb insert-not-empty singleton-iff)

have list-same:  $\bigwedge f \ s\text{-set } z' \ x. f \ ' \ \text{set } s\text{-set} = \{z'\} \implies \text{List.member } s\text{-set } x \implies f \ x = z'$ 
by auto

fix z
show ?thesis
when  $\top \text{Rep-Set}_{\text{base}} (\text{select-object}_{\text{Set}} f \ s\text{-set } \tau)^\top = z$ 
apply (insert that def-sel[simplified foundation16],
simp add: select-object-anySet-def foundation22
Let-def null-fun-def bot-fun-def OclIf-def
split: if-split-asm)
apply (simp add: StrongEq-def OclInt1-def true-def UML-Set.OclSize-def
bot-option-def UML-Set.OclANY-def null-fun-def
split: if-split-asm)
apply (subgoal-tac  $\exists z'. z = \{z'\}$ )
prefer 2
apply (rule finite.cases[where a = z], simp, simp, simp)
apply (rule card-singl, simp, simp)
apply (erule exE, clarsimp)

apply (simp add: select-objectSet-def select-object-def)
apply (subst (asm) select-fold-execSet)
apply (rule fold-val-elemSet, simp add: OclValid-def true-def)
apply (simp add: comp-def)

apply (case-tac s-set, simp)
apply auto
done
qed blast+

end

```

```

theory UML-Contracts
imports UML-State
begin

```

Modeling of an operation contract for an operation with 2 arguments, (so depending on three parameters if one takes "self" into account).

```

locale contract-scheme =
  fixes f-v
  fixes f-lam
  fixes f ::  $(\mathfrak{A}, 'a0::\text{null})\text{val} \Rightarrow$ 
     $'b \Rightarrow$ 
     $(\mathfrak{A}, 'res::\text{null})\text{val}$ 
  fixes PRE
  fixes POST
  assumes def-scheme':  $f \ \text{self } x \equiv (\lambda \tau. \text{SOME } res. \text{let } res = \lambda \cdot res \ \text{in}$ 
     $\text{if } (\tau \models (\delta \ \text{self})) \wedge f\text{-v } x \ \tau$ 
     $\text{then } (\tau \models \text{PRE } \text{self } x) \wedge$ 
     $(\tau \models \text{POST } \text{self } x \ res)$ 
     $\text{else } \tau \models res \triangleq \text{invalid})$ 
  assumes all-post':  $\forall \sigma \ \sigma' \ \sigma''. ((\sigma, \sigma') \models \text{PRE } \text{self } x) = ((\sigma, \sigma'') \models \text{PRE } \text{self } x)$ 

```

assumes cp_{PRE}' : $PRE (self) x \tau = PRE (\lambda -. self \tau) (f-lam x \tau) \tau$

assumes cp_{POST}' : $POST (self) x (res) \tau = POST (\lambda -. self \tau) (f-lam x \tau) (\lambda -. res \tau) \tau$

assumes $f-v-val$: $\bigwedge a1. f-v (f-lam a1 \tau) \tau = f-v a1 \tau$

begin

lemma $strict0$ [$simp$]: $f invalid X = invalid$

by($rule ext$, $rename-tac \tau$, $simp add: def-scheme' StrongEq-def OclValid-def false-def true-def$)

lemma $nullstrict0$ [$simp$]: $f null X = invalid$

by($rule ext$, $rename-tac \tau$, $simp add: def-scheme' StrongEq-def OclValid-def false-def true-def$)

lemma $cp0$: $f self a1 \tau = f (\lambda -. self \tau) (f-lam a1 \tau) \tau$

proof –

have A : $(\tau \models \delta (\lambda -. self \tau)) = (\tau \models \delta self)$ **by**($simp add: OclValid-def cp-defined[symmetric]$)

have B : $f-v (f-lam a1 \tau) \tau = f-v a1 \tau$ **by** ($rule f-v-val$)

have D : $(\tau \models PRE (\lambda -. self \tau) (f-lam a1 \tau)) = (\tau \models PRE self a1)$

by($simp add: OclValid-def cp_{PRE}'[symmetric]$)

show $?thesis$

apply($auto simp: def-scheme' A B D$)

apply($simp add: OclValid-def$)

by($subst cp_{POST}'$, $simp$)

qed

theorem $unfold'$:

assumes $context-ok$: $cp E$

and $args-def-or-valid$: $(\tau \models \delta self) \wedge f-v a1 \tau$

and $pre-satisfied$: $\tau \models PRE self a1$

and $post-satisfiable$: $\exists res. (\tau \models POST self a1 (\lambda -. res))$

and $sat-for-sols-post$: $(\bigwedge res. \tau \models POST self a1 (\lambda -. res)) \implies \tau \models E (\lambda -. res)$

shows $\tau \models E(f self a1)$

proof –

have $cp0$: $\bigwedge X \tau. E X \tau = E (\lambda -. X \tau) \tau$ **by**($insert context-ok[simplified cp-def]$, $auto$)

show $?thesis$

apply($simp add: OclValid-def, subst cp0, fold OclValid-def$)

apply($simp add: def-scheme' args-def-or-valid pre-satisfied$)

apply($insert post-satisfiable, elim exE$)

apply($rule Hilbert-Choice.someI2, assumption$)

by($rule sat-for-sols-post, simp$)

qed

lemma $unfold2'$:

assumes $context-ok$: $cp E$

and $args-def-or-valid$: $(\tau \models \delta self) \wedge (f-v a1 \tau)$

and $pre-satisfied$: $\tau \models PRE self a1$

and $postsplit-satisfied$: $\tau \models POST' self a1$

and $post-decomposable$: $\bigwedge res. (POST self a1 res) = ((POST' self a1) \text{ and } (res \triangleq (BODY self a1)))$

shows $(\tau \models E(f self a1)) = (\tau \models E(BODY self a1))$

proof –

have $cp0$: $\bigwedge X \tau. E X \tau = E (\lambda -. X \tau) \tau$ **by**($insert context-ok[simplified cp-def]$, $auto$)

show $?thesis$

apply($simp add: OclValid-def, subst cp0, fold OclValid-def$)

apply($simp add: def-scheme' args-def-or-valid pre-satisfied$
 $post-decomposable postsplit-satisfied foundation10'$)

apply($subst some-equality$)

apply($simp add: OclValid-def StrongEq-def true-def$)**+**

by($subst (2) cp0, rule refl$)

```

qed
end

locale contract0 =
  fixes f :: ('A, 'α0::null)val ⇒
    ('A, 'res::null)val
  fixes PRE
  fixes POST
  assumes def-scheme: f self ≡ (λ τ. SOME res. let res = λ -. res in
    if (τ ⊨ (δ self))
    then (τ ⊨ PRE self) ∧
      (τ ⊨ POST self res)
    else τ ⊨ res ≜ invalid)
  assumes all-post: ∀ σ σ' σ''. ((σ,σ') ⊨ PRE self) = ((σ,σ'') ⊨ PRE self)

  assumes cpPRE: PRE (self) τ = PRE (λ -. self τ) τ
  assumes cpPOST: POST (self) (res) τ = POST (λ -. self τ) (λ -. res τ) τ

sublocale contract0 < contract-scheme λ-. True λx -. x λx -. f x λx -. PRE x λx -. POST x
apply(unfold-locales)
  apply(simp add: def-scheme, rule all-post, rule cpPRE, rule cpPOST)
by simp

context contract0
begin
  lemma cp-pre: cp self' ⇒ cp (λX. PRE (self' X) )
  by(rule-tac f=PRE in cpI1, auto intro: cpPRE)

  lemma cp-post: cp self' ⇒ cp res' ⇒ cp (λX. POST (self' X) (res' X))
  by(rule-tac f=POST in cpI2, auto intro: cpPOST)

  lemma cp [simp]: cp self' ⇒ cp res' ⇒ cp (λX. f (self' X) )
  by(rule-tac f=f in cpI1, auto intro:cp0)

  lemmas unfold = unfold'[simplified]

  lemma unfold2 :
    assumes          cp E
    and              (τ ⊨ δ self)
    and              τ ⊨ PRE self
    and              τ ⊨ POST' self
    and              ∧ res. (POST self res) =
                      ((POST' self) and (res ≜ (BODY self)))
    shows (τ ⊨ E(f self)) = (τ ⊨ E(BODY self))
    apply(rule unfold2'[simplified])
    by((rule assms)+)
end

locale contract1 =
  fixes f :: ('A, 'α0::null)val ⇒
    ('A, 'α1::null)val ⇒
    ('A, 'res::null)val
  fixes PRE
  fixes POST

```

```

assumes def-scheme: f self a1 ≡
  (λ τ. SOME res. let res = λ -. res in
    if (τ ⊨ (δ self)) ∧ (τ ⊨ v a1)
    then (τ ⊨ PRE self a1) ∧
      (τ ⊨ POST self a1 res)
    else τ ⊨ res ≜ invalid)
assumes all-post: ∀ σ σ' σ''. ((σ,σ') ⊨ PRE self a1) = ((σ,σ'') ⊨ PRE self a1)

assumes cpPRE: PRE (self) (a1) τ = PRE (λ -. self τ) (λ -. a1 τ) τ

assumes cpPOST: POST (self) (a1) (res) τ = POST (λ -. self τ)(λ -. a1 τ) (λ -. res τ) τ

sublocale contract1 < contract-scheme λa1 τ. (τ ⊨ v a1) λa1 τ. (λ -. a1 τ)
apply(unfold-locales)
apply(rule def-scheme, rule all-post, rule cpPRE, rule cpPOST)
by(simp add: OclValid-def cp-valid[symmetric])

context contract1
begin
lemma strict1[simp]: f self invalid = invalid
by(rule ext, rename-tac τ, simp add: def-scheme StrongEq-def OclValid-def false-def true-def)

lemma defined-mono : τ ⊨ v(f Y Z) ⇒ (τ ⊨ δ Y) ∧ (τ ⊨ v Z)
by(auto simp: valid-def bot-fun-def invalid-def
  def-scheme StrongEq-def OclValid-def false-def true-def
  split: if-split-asm)

lemma cp-pre: cp self' ⇒ cp a1' ⇒ cp (λX. PRE (self' X) (a1' X) )
by(rule-tac f=PRE in cpI2, auto intro: cpPRE)

lemma cp-post: cp self' ⇒ cp a1' ⇒ cp res'
  ⇒ cp (λX. POST (self' X) (a1' X) (res' X))
by(rule-tac f=POST in cpI3, auto intro: cpPOST)

lemma cp [simp]: cp self' ⇒ cp a1' ⇒ cp res' ⇒ cp (λX. f (self' X) (a1' X))
by(rule-tac f=f in cpI2, auto intro:cp0)

lemmas unfold = unfold'
lemmas unfold2 = unfold2'
end

locale contract2 =
fixes f :: ('A,'α0::null)val ⇒
  ('A,'α1::null)val ⇒ ('A,'α2::null)val ⇒
  ('A,'res::null)val
fixes PRE
fixes POST
assumes def-scheme: f self a1 a2 ≡
  (λ τ. SOME res. let res = λ -. res in
    if (τ ⊨ (δ self)) ∧ (τ ⊨ v a1) ∧ (τ ⊨ v a2)
    then (τ ⊨ PRE self a1 a2) ∧
      (τ ⊨ POST self a1 a2 res)
    else τ ⊨ res ≜ invalid)
assumes all-post: ∀ σ σ' σ''. ((σ,σ') ⊨ PRE self a1 a2) = ((σ,σ'') ⊨ PRE self a1 a2)

assumes cpPRE: PRE (self) (a1) (a2) τ = PRE (λ -. self τ) (λ -. a1 τ) (λ -. a2 τ) τ

```

```

assumes  $cp_{POST} : \bigwedge res. POST (self) (a1) (a2) (res) \tau =$ 
            $POST (\lambda -. self \tau) (\lambda -. a1 \tau) (\lambda -. a2 \tau) (\lambda -. res \tau) \tau$ 

sublocale  $contract2 < contract-scheme \lambda(a1,a2) \tau. (\tau \models v a1) \wedge (\tau \models v a2)$ 
            $\lambda(a1,a2) \tau. (\lambda -. a1 \tau, \lambda -. a2 \tau)$ 
            $(\lambda x (a,b). f x a b)$ 
            $(\lambda x (a,b). PRE x a b)$ 
            $(\lambda x (a,b). POST x a b)$ 

apply(unfold-locales)
  apply(auto simp add: def-scheme)
  apply(metis all-post, metis all-post)
  apply(subst cp_{PRE}, simp)
  apply(subst cp_{POST}, simp)
by(simp-all add: OclValid-def cp-valid[symmetric])

context  $contract2$ 
begin
  lemma  $strict0'[simp] : f invalid X Y = invalid$ 
  by(insert strict0[of (X,Y)], simp)

  lemma  $nullstrict0'[simp]: f null X Y = invalid$ 
  by(insert nullstrict0[of (X,Y)], simp)

  lemma  $strict1[simp]: f self invalid Y = invalid$ 
  by(rule ext, rename-tac  $\tau$ , simp add: def-scheme StrongEq-def OclValid-def false-def true-def)

  lemma  $strict2[simp]: f self X invalid = invalid$ 
  by(rule ext, rename-tac  $\tau$ , simp add: def-scheme StrongEq-def OclValid-def false-def true-def)

  lemma  $defined-mono : \tau \models v(f X Y Z) \implies (\tau \models \delta X) \wedge (\tau \models v Y) \wedge (\tau \models v Z)$ 
  by(auto simp: valid-def bot-fun-def invalid-def
      def-scheme StrongEq-def OclValid-def false-def true-def
      split: if-split-asm)

  lemma  $cp-pre: cp self' \implies cp a1' \implies cp a2' \implies cp (\lambda X. PRE (self' X) (a1' X) (a2' X) )$ 
  by(rule-tac f=PRE in cpI3, auto intro: cp_{PRE})

  lemma  $cp-post: cp self' \implies cp a1' \implies cp a2' \implies cp res'$ 
            $\implies cp (\lambda X. POST (self' X) (a1' X) (a2' X) (res' X))$ 
  by(rule-tac f=POST in cpI4, auto intro: cp_{POST})

  lemma  $cp0' : f self a1 a2 \tau = f (\lambda -. self \tau) (\lambda -. a1 \tau) (\lambda -. a2 \tau) \tau$ 
  by(rule cp0[of - (a1,a2), simplified])

  lemma  $cp [simp]: cp self' \implies cp a1' \implies cp a2' \implies cp res'$ 
            $\implies cp (\lambda X. f (self' X) (a1' X) (a2' X))$ 
  by(rule-tac f=f in cpI3, auto intro:cp0')

theorem  $unfold :$ 
  assumes  $cp E$ 
  and  $(\tau \models \delta self) \wedge (\tau \models v a1) \wedge (\tau \models v a2)$ 
  and  $\tau \models PRE self a1 a2$ 
  and  $\exists res. (\tau \models POST self a1 a2 (\lambda -. res))$ 
  and  $(\bigwedge res. \tau \models POST self a1 a2 (\lambda -. res) \implies \tau \models E (\lambda -. res))$ 
  shows  $\tau \models E(f self a1 a2)$ 
  apply(rule unfold'[of - - - (a1, a2), simplified])

```

```

by((rule assms)+)

lemma unfold2 :
  assumes
    cp E
  and
    ( $\tau \models \delta \text{ self}$ )  $\wedge$  ( $\tau \models v \ a1$ )  $\wedge$  ( $\tau \models v \ a2$ )
  and
     $\tau \models \text{PRE self } a1 \ a2$ 
  and
     $\tau \models \text{POST}' \text{ self } a1 \ a2$ 
  and
     $\bigwedge \text{res. } (\text{POST self } a1 \ a2 \ \text{res}) =$ 
      (( $\text{POST}' \text{ self } a1 \ a2$ ) and ( $\text{res} \triangleq (\text{BODY self } a1 \ a2)$ ))
  shows ( $\tau \models E(f \text{ self } a1 \ a2)$ ) = ( $\tau \models E(\text{BODY self } a1 \ a2)$ )
  apply(rule unfold2'[of - - - (a1, a2), simplified])
  by((rule assms)+)
end

locale contract3 =
  fixes f :: (' $\mathfrak{A}$ , ' $\alpha 0$ ::null)val  $\Rightarrow$ 
    (' $\mathfrak{A}$ , ' $\alpha 1$ ::null)val  $\Rightarrow$ 
    (' $\mathfrak{A}$ , ' $\alpha 2$ ::null)val  $\Rightarrow$ 
    (' $\mathfrak{A}$ , ' $\alpha 3$ ::null)val  $\Rightarrow$ 
    (' $\mathfrak{A}$ , ' $\text{res}$ ::null)val

  fixes PRE
  fixes POST
  assumes def-scheme: f self a1 a2 a3  $\equiv$ 
    ( $\lambda \tau. \text{SOME res. let res} = \lambda -. \text{res in}$ 
      if ( $\tau \models (\delta \text{ self})$ )  $\wedge$  ( $\tau \models v \ a1$ )  $\wedge$  ( $\tau \models v \ a2$ )  $\wedge$  ( $\tau \models v \ a3$ )
      then ( $\tau \models \text{PRE self } a1 \ a2 \ a3$ )  $\wedge$ 
        ( $\tau \models \text{POST self } a1 \ a2 \ a3 \ \text{res}$ )
      else  $\tau \models \text{res} \triangleq \text{invalid}$ )
  assumes all-post:  $\forall \sigma \sigma' \sigma''. ((\sigma, \sigma') \models \text{PRE self } a1 \ a2 \ a3) = ((\sigma, \sigma'') \models \text{PRE self } a1 \ a2 \ a3)$ 

  assumes cpPRE: PRE (self) (a1) (a2) (a3)  $\tau = \text{PRE } (\lambda -. \text{self } \tau) (\lambda -. a1 \ \tau) (\lambda -. a2 \ \tau) (\lambda -. a3 \ \tau) \ \tau$ 

  assumes cpPOST:  $\bigwedge \text{res. POST (self) (a1) (a2) (a3) (\text{res}) } \tau =$ 
    POST ( $\lambda -. \text{self } \tau$ ) ( $\lambda -. a1 \ \tau$ ) ( $\lambda -. a2 \ \tau$ ) ( $\lambda -. a3 \ \tau$ ) ( $\lambda -. \text{res } \tau$ )  $\tau$ 

sublocale contract3 < contract-scheme  $\lambda(a1, a2, a3) \tau. (\tau \models v \ a1) \wedge (\tau \models v \ a2) \wedge (\tau \models v \ a3)$ 
   $\lambda(a1, a2, a3) \tau. (\lambda -. a1 \ \tau, \lambda -. a2 \ \tau, \lambda -. a3 \ \tau)$ 
  ( $\lambda x (a, b, c). f \ x \ a \ b \ c$ )
  ( $\lambda x (a, b, c). \text{PRE } x \ a \ b \ c$ )
  ( $\lambda x (a, b, c). \text{POST } x \ a \ b \ c$ )

apply(unfold-locales)
  apply(auto simp add: def-scheme)
    apply (metis all-post, metis all-post)
  apply(subst cpPRE, simp)
  apply(subst cpPOST, simp)
by(simp-all add: OclValid-def cp-valid[symmetric])

context contract3
begin
  lemma strict0'[simp]: f invalid X Y Z = invalid
  by(rule ext, rename-tac  $\tau$ , simp add: def-scheme StrongEq-def OclValid-def false-def true-def)

  lemma nullstrict0'[simp]: f null X Y Z = invalid
  by(rule ext, rename-tac  $\tau$ , simp add: def-scheme StrongEq-def OclValid-def false-def true-def)

  lemma strict1[simp]: f self invalid Y Z = invalid

```

by(rule ext, rename-tac τ , simp add: def-scheme StrongEq-def OclValid-def false-def true-def)

lemma strict2[simp]: $f \text{ self } X \text{ invalid } Z = \text{invalid}$

by(rule ext, rename-tac τ , simp add: def-scheme StrongEq-def OclValid-def false-def true-def)

lemma defined-mono : $\tau \models v(f \ W \ X \ Y \ Z) \implies (\tau \models \delta \ W) \wedge (\tau \models v \ X) \wedge (\tau \models v \ Y) \wedge (\tau \models v \ Z)$

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

lemma cp-pre: $cp \ \text{self}' \implies cp \ a1' \implies cp \ a2' \implies cp \ a3'$
 $\implies cp \ (\lambda X. \ \text{PRE} \ (\text{self}' \ X) \ (a1' \ X) \ (a2' \ X) \ (a3' \ X))$

by(rule-tac f=PRE in cpI4, auto intro: cpPRE)

lemma cp-post: $cp \ \text{self}' \implies cp \ a1' \implies cp \ a2' \implies cp \ a3' \implies cp \ \text{res}'$
 $\implies cp \ (\lambda X. \ \text{POST} \ (\text{self}' \ X) \ (a1' \ X) \ (a2' \ X) \ (a3' \ X) \ (\text{res}' \ X))$

by(rule-tac f=POST in cpI5, auto intro: cpPOST)

lemma cp0' : $f \ \text{self} \ a1 \ a2 \ a3 \ \tau = f \ (\lambda \cdot. \ \text{self} \ \tau) \ (\lambda \cdot. \ a1 \ \tau) \ (\lambda \cdot. \ a2 \ \tau) \ (\lambda \cdot. \ a3 \ \tau) \ \tau$

by (rule cp0[of - (a1,a2,a3), simplified])

lemma cp [simp]: $cp \ \text{self}' \implies cp \ a1' \implies cp \ a2' \implies cp \ a3' \implies cp \ \text{res}'$
 $\implies cp \ (\lambda X. \ f \ (\text{self}' \ X) \ (a1' \ X) \ (a2' \ X) \ (a3' \ X))$

by(rule-tac f=f in cpI4, auto intro:cp0')

theorem unfold :

assumes $cp \ E$
and $(\tau \models \delta \ \text{self}) \wedge (\tau \models v \ a1) \wedge (\tau \models v \ a2) \wedge (\tau \models v \ a3)$
and $\tau \models \text{PRE} \ \text{self} \ a1 \ a2 \ a3$
and $\exists \ \text{res}. \ (\tau \models \text{POST} \ \text{self} \ a1 \ a2 \ a3 \ (\lambda \cdot. \ \text{res}))$
and $(\bigwedge \ \text{res}. \ \tau \models \text{POST} \ \text{self} \ a1 \ a2 \ a3 \ (\lambda \cdot. \ \text{res}) \implies \tau \models E \ (\lambda \cdot. \ \text{res}))$
shows $\tau \models E(f \ \text{self} \ a1 \ a2 \ a3)$
apply(rule unfold'[of - - - (a1, a2, a3), simplified])
by((rule assms)+)

lemma unfold2 :

assumes $cp \ E$
and $(\tau \models \delta \ \text{self}) \wedge (\tau \models v \ a1) \wedge (\tau \models v \ a2) \wedge (\tau \models v \ a3)$
and $\tau \models \text{PRE} \ \text{self} \ a1 \ a2 \ a3$
and $\tau \models \text{POST}' \ \text{self} \ a1 \ a2 \ a3$
and $\bigwedge \ \text{res}. \ (\text{POST} \ \text{self} \ a1 \ a2 \ a3 \ \text{res}) =$
 $((\text{POST}' \ \text{self} \ a1 \ a2 \ a3) \ \text{and} \ (\text{res} \triangleq (\text{BODY} \ \text{self} \ a1 \ a2 \ a3)))$
shows $(\tau \models E(f \ \text{self} \ a1 \ a2 \ a3)) = (\tau \models E(\text{BODY} \ \text{self} \ a1 \ a2 \ a3))$
apply(rule unfold2'[of - - - (a1, a2, a3), simplified])
by((rule assms)+)

end

end

theory UML-Tools
imports UML-Logic
begin

```

lemmas subst1 = StrongEq-L-subst2-rev
  foundation15[THEN iffD2, THEN StrongEq-L-subst2-rev]
  foundation7'[THEN iffD2, THEN foundation15[THEN iffD2,
    THEN StrongEq-L-subst2-rev]]
  foundation14[THEN iffD2, THEN StrongEq-L-subst2-rev]
  foundation13[THEN iffD2, THEN StrongEq-L-subst2-rev]

lemmas subst2 = StrongEq-L-subst3-rev
  foundation15[THEN iffD2, THEN StrongEq-L-subst3-rev]
  foundation7'[THEN iffD2, THEN foundation15[THEN iffD2,
    THEN StrongEq-L-subst3-rev]]
  foundation14[THEN iffD2, THEN StrongEq-L-subst3-rev]
  foundation13[THEN iffD2, THEN StrongEq-L-subst3-rev]

lemmas subst4 = StrongEq-L-subst4-rev
  foundation15[THEN iffD2, THEN StrongEq-L-subst4-rev]
  foundation7'[THEN iffD2, THEN foundation15[THEN iffD2,
    THEN StrongEq-L-subst4-rev]]
  foundation14[THEN iffD2, THEN StrongEq-L-subst4-rev]
  foundation13[THEN iffD2, THEN StrongEq-L-subst4-rev]

lemmas substs = subst1 subst2 subst4 [THEN iffD2] subst4
thm substs
ML⟨
  fun ocl-subst-asm-tac ctxt = FIRST'(map (fn C => (eresolve0-tac [C]) THEN' (simp-tac ctxt))
    @{thms substs})

  val ocl-subst-asm = fn ctxt => SIMPLE-METHOD (ocl-subst-asm-tac ctxt 1);

  val - = Theory.setup
    (Method.setup (Binding.name ocl-subst-asm)
      (Scan.succeed (ocl-subst-asm))
      ocl substitution step)
  ⟩

lemma test1 :  $\tau \models A \implies \tau \models (A \text{ and } B \triangleq B)$ 
apply(tactic ocl-subst-asm-tac @{context} 1)
apply(simp)
done

lemma test2 :  $\tau \models A \implies \tau \models (A \text{ and } B \triangleq B)$ 
by(ocl-subst-asm, simp)

lemma test3 :  $\tau \models A \implies \tau \models (A \text{ and } A)$ 
by(ocl-subst-asm, simp)

lemma test4 :  $\tau \models \text{not } A \implies \tau \models (A \text{ and } B \triangleq \text{false})$ 
by(ocl-subst-asm, simp)

lemma test5 :  $\tau \models (A \triangleq \text{null}) \implies \tau \models (B \triangleq \text{null}) \implies \neg (\tau \models (A \text{ and } B))$ 
by(ocl-subst-asm, ocl-subst-asm, simp)

lemma test6 :  $\tau \models \text{not } A \implies \neg (\tau \models (A \text{ and } B))$ 
by(ocl-subst-asm, simp)

```

```
lemma test7 :  $\neg (\tau \models (v\ A)) \implies \tau \models (\text{not } B) \implies \neg (\tau \models (A\ \text{and } B))$   
by(ocl-subst-asm, ocl-subst-asm, simp)
```

```
lemma X:  $\neg (\tau \models (\text{invalid and } B))$   
apply(insert foundation8[of  $\tau$  B], elim disjE,  
      simp add:defined-bool-split, elim disjE)  
apply(ocl-subst-asm, simp)  
apply(ocl-subst-asm, simp)  
apply(ocl-subst-asm, simp)  
apply(ocl-subst-asm, simp)  
done
```

```
lemma X':  $\neg (\tau \models (\text{invalid and } B))$   
by(simp add:foundation10')  
lemma Y:  $\neg (\tau \models (\text{null and } B))$   
by(simp add: foundation10')  
lemma Z:  $\neg (\tau \models (\text{false and } B))$   
by(simp add: foundation10')  
lemma Z':  $(\tau \models (\text{true and } B)) = (\tau \models B)$   
by(simp)
```

end

```
theory UML-Main  
imports UML-Contracts UML-Tools  
begin  
end
```

4. Example: The Employee Analysis Model

```
theory
  Analysis-UML
imports
  ../../../../UML-Main
begin
```

4.1. Introduction

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

Such generic function or “compiler” can be implemented in Isabelle on the ML level. This has been done, for a semantics following the open-world assumption, for UML 2.0 in [4, 7]. In this paper, we follow another approach for UML 2.4: we define the concepts of the compilation informally, and present a concrete example which is verified in Isabelle/HOL.

4.1.1. Outlining the Example

We are presenting here an “analysis-model” of the (slightly modified) example Figure 7.3, page 20 of the OCL standard [32]. Here, analysis model means that associations were really represented as relation on objects on the state—as is intended by the standard—rather by pointers between objects as is done in our “design model” (see Chapter 5). To be precise, this theory contains the formalization of the data-part covered by the UML class model (see Figure 4.1):

This means that the association (attached to the association class **EmployeeRanking**) with the association ends **boss** and **employees** is implemented by the attribute **boss** and the operation **employees** (to be discussed in the OCL part captured by the subsequent theory).

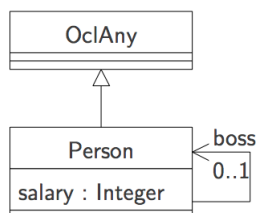


Figure 4.1.: A simple UML class model drawn from Figure 7.3, page 20 of [32].

4.2. Example Data-Universe and its Infrastructure

Ideally, the following is generated automatically from a UML class model.

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

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

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

Now, we construct a concrete “universe of OclAny types” by injection into a sum type containing the class types. This type of OclAny will be used as instance for all respective type-variables.

```
datatype  $\mathfrak{A}$  = 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
```

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
```

4.3. Instantiation of the Generic Strict Equality

We instantiate the referential equality on *Person* and *OclAny*

```
overloading StrictRefEq ≡ StrictRefEq :: [Person, Person] ⇒ Boolean
begin
  definition StrictRefEqObjectPerson : (x::Person) ≐ y ≡ StrictRefEqObject x y
end
```

```
overloading StrictRefEq ≡ StrictRefEq :: [OclAny, OclAny] ⇒ Boolean
begin
  definition StrictRefEqObjectOclAny : (x::OclAny) ≐ y ≡ StrictRefEqObject x y
end
```

```
lemmas cps23 =
  cp-StrictRefEqObject [of x::Person y::Person τ,
    simplified StrictRefEqObjectPerson[symmetric]]
  cp-intro(9) [of P::Person ⇒ Person Q::Person ⇒ Person,
    simplified StrictRefEqObjectPerson[symmetric]]
  StrictRefEqObject-def [of x::Person y::Person,
    simplified StrictRefEqObjectPerson[symmetric]]
  StrictRefEqObject-defargs [of - x::Person y::Person,
    simplified StrictRefEqObjectPerson[symmetric]]
  StrictRefEqObject-strict1
    [of x::Person,
    simplified StrictRefEqObjectPerson[symmetric]]
  StrictRefEqObject-strict2
    [of x::Person,
    simplified StrictRefEqObjectPerson[symmetric]]
for x y τ P Q
```

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

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

4.4. OclAsType

4.4.1. Definition

```
consts OclAsTypeOclAny :: 'α ⇒ OclAny (⟨(-) .oclAsType'(OclAny)⟩)
consts OclAsTypePerson :: 'α ⇒ Person (⟨(-) .oclAsType'(Person)⟩)
```

```
definition OclAsTypeOclAny-ℳ = (λu. ⊔case u of inOclAny a ⇒ a
  | inPerson (mkPerson oid a) ⇒ mkOclAny oid ⊔a)
```

```
lemma OclAsTypeOclAny-ℳ-some: OclAsTypeOclAny-ℳ x ≠ None
by(simp add: OclAsTypeOclAny-ℳ-def)
```

```
overloading OclAsTypeOclAny ≡ OclAsTypeOclAny :: OclAny ⇒ OclAny
begin
  definition OclAsTypeOclAny-OclAny:
    (X::OclAny) .oclAsType(OclAny) ≡ X
end
```

```
overloading OclAsTypeOclAny ≡ OclAsTypeOclAny :: Person ⇒ OclAny
```

```

begin
  definition OclAsTypeOclAny-Person:
    (X::Person) .oclAsType(OclAny) ≡
      (λτ. case X τ of
        | ⊥ ⇒ invalid τ
        | ⊥⊥ ⇒ null τ
        | ⊥⊥ mkPerson oid a⊥ ⇒ ⊥ (mkOclAny oid ⊥⊥)
      )
end

definition OclAsTypePerson-ℳ =
  (λu. case u of inPerson p ⇒ p⊥
    | inOclAny (mkOclAny oid ⊥⊥) ⇒ ⊥ mkPerson oid a⊥
    | - ⇒ None)

overloading OclAsTypePerson ≡ OclAsTypePerson :: OclAny ⇒ Person
begin
  definition OclAsTypePerson-OclAny:
    (X::OclAny) .oclAsType(Person) ≡
      (λτ. case X τ of
        | ⊥ ⇒ invalid τ
        | ⊥⊥ ⇒ null τ
        | ⊥⊥ mkOclAny oid ⊥⊥ ⇒ invalid τ — down-cast exception
        | ⊥⊥ mkOclAny oid ⊥⊥ ⇒ ⊥ mkPerson oid a⊥)
      )
end

overloading OclAsTypePerson ≡ OclAsTypePerson :: Person ⇒ Person
begin
  definition OclAsTypePerson-Person:
    (X::Person) .oclAsType(Person) ≡ X
end
lemmas [simp] =
  OclAsTypeOclAny-OclAny
  OclAsTypePerson-Person

```

4.4.2. Context Passing

```

lemma cp-OclAsTypeOclAny-Person-Person: cp P ⇒ cp(λX. (P (X::Person)::Person) .oclAsType(OclAny))
by(rule cpI1, simp-all add: OclAsTypeOclAny-Person)
lemma cp-OclAsTypeOclAny-OclAny-OclAny: cp P ⇒ cp(λX. (P (X::OclAny)::OclAny) .oclAsType(OclAny))
by(rule cpI1, simp-all add: OclAsTypeOclAny-OclAny)
lemma cp-OclAsTypePerson-Person-Person: cp P ⇒ cp(λX. (P (X::Person)::Person) .oclAsType(Person))
by(rule cpI1, simp-all add: OclAsTypePerson-Person)
lemma cp-OclAsTypePerson-OclAny-OclAny: cp P ⇒ cp(λX. (P (X::OclAny)::OclAny) .oclAsType(Person))
by(rule cpI1, simp-all add: OclAsTypePerson-OclAny)

lemma cp-OclAsTypeOclAny-Person-OclAny: cp P ⇒ cp(λX. (P (X::Person)::OclAny) .oclAsType(OclAny))
by(rule cpI1, simp-all add: OclAsTypeOclAny-OclAny)
lemma cp-OclAsTypeOclAny-OclAny-Person: cp P ⇒ cp(λX. (P (X::OclAny)::Person) .oclAsType(OclAny))
by(rule cpI1, simp-all add: OclAsTypeOclAny-Person)
lemma cp-OclAsTypePerson-Person-OclAny: cp P ⇒ cp(λX. (P (X::Person)::OclAny) .oclAsType(Person))
by(rule cpI1, simp-all add: OclAsTypePerson-OclAny)
lemma cp-OclAsTypePerson-OclAny-Person: cp P ⇒ cp(λX. (P (X::OclAny)::Person) .oclAsType(Person))
by(rule cpI1, simp-all add: OclAsTypePerson-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

4.4.3. 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*)

4.5. OclIsTypeOf

4.5.1. Definition

consts *OclIsTypeOfOclAny* :: ' $\alpha \Rightarrow \text{Boolean} (\langle(-).oclIsTypeOf'(OclAny)'\rangle)$ '

consts *OclIsTypeOfPerson* :: ' $\alpha \Rightarrow \text{Boolean} (\langle(-).oclIsTypeOf'(Person)'\rangle)$ '

overloading *OclIsTypeOfOclAny* \equiv *OclIsTypeOfOclAny* :: *OclAny* \Rightarrow *Boolean*

begin

definition *OclIsTypeOfOclAny-OclAny*:

(*X*::*OclAny*) .*oclIsTypeOf*(*OclAny*) \equiv
 $(\lambda \tau. \text{case } X \ \tau \ \text{of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \perp_{\perp} \Rightarrow \text{true } \tau \text{ — invalid } ??$
 $\quad | \perp_{mkOclAny} \ \text{oid } \perp_{\perp} \Rightarrow \text{true } \tau$
 $\quad | \perp_{mkOclAny} \ \text{oid } \perp_{\perp} \Rightarrow \text{false } \tau)$

end

lemma *OclIsTypeOfOclAny-OclAny'*:

(*X*::*OclAny*) .*oclIsTypeOf*(*OclAny*) =
 $(\lambda \tau. \text{if } \tau \models v \ X \ \text{then } (\text{case } X \ \tau \ \text{of}$
 $\quad \perp_{\perp} \Rightarrow \text{true } \tau \text{ — invalid } ??$
 $\quad | \perp_{mkOclAny} \ \text{oid } \perp_{\perp} \Rightarrow \text{true } \tau$
 $\quad | \perp_{mkOclAny} \ \text{oid } \perp_{\perp} \Rightarrow \text{false } \tau)$
 $\quad \text{else } \text{invalid } \tau)$

apply(*rule ext*, *simp add*: *OclIsTypeOfOclAny-OclAny*)

by(*case-tac* $\tau \models v \ X$, *auto simp*: *foundation18' bot-option-def*)

interpretation *OclIsTypeOfOclAny-OclAny* :

profile-mono-schemeV

OclIsTypeOfOclAny::*OclAny* \Rightarrow *Boolean*

$\lambda X. (\text{case } X \ \text{of}$

$\quad \perp_{None} \Rightarrow \perp_{True} \text{ — invalid } ??$
 $\quad | \perp_{mkOclAny} \ \text{oid } \perp_{None} \Rightarrow \perp_{True}$
 $\quad | \perp_{mkOclAny} \ \text{oid } \perp_{\perp} \Rightarrow \perp_{False})$

apply(*unfold-locals*, *simp add: atomize-eq, rule ext*)
by(*auto simp: OclIsTypeOf_{OclAny}-OclAny' OclValid-def true-def false-def*
split: option.split type_{OclAny}.split)

overloading $OclIsTypeOf_{OclAny} \equiv OclIsTypeOf_{OclAny} :: Person \Rightarrow Boolean$

begin

definition $OclIsTypeOf_{OclAny}\text{-}Person$:

$(X::Person) .oclIsTypeOf(OclAny) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \sqsubset \perp \Rightarrow \text{true } \tau \quad \text{--- invalid ??}$
 $\quad | \sqsubset _ \sqsubset \Rightarrow \text{false } \tau) \quad \text{--- must have actual type } Person \text{ otherwise}$

end

overloading $OclIsTypeOf_{Person} \equiv OclIsTypeOf_{Person} :: OclAny \Rightarrow Boolean$

begin

definition $OclIsTypeOf_{Person}\text{-}OclAny$:

$(X::OclAny) .oclIsTypeOf(Person) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \sqsubset \perp \Rightarrow \text{true } \tau$
 $\quad | \sqsubset mk_{OclAny} \ \text{oid } \perp \sqsubset \Rightarrow \text{false } \tau$
 $\quad | \sqsubset mk_{OclAny} \ \text{oid } \sqsubset \sqsubset \Rightarrow \text{true } \tau)$

end

overloading $OclIsTypeOf_{Person} \equiv OclIsTypeOf_{Person} :: Person \Rightarrow Boolean$

begin

definition $OclIsTypeOf_{Person}\text{-}Person$:

$(X::Person) .oclIsTypeOf(Person) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | _ \Rightarrow \text{true } \tau)$

end

4.5.2. Context Passing

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemmas [*simp*] =

$cp\text{-}OclIsTypeOf_{OclAny}\text{-}Person\text{-}Person$

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

4.5.3. Execution with Invalid or Null as Argument

lemma *OclIsTypeOfOclAny-OclAny-strict1*[simp]:
 (*invalid::OclAny*) .*oclIsTypeOf*(*OclAny*) = *invalid*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def invalid-def*
OclIsTypeOfOclAny-OclAny)
lemma *OclIsTypeOfOclAny-OclAny-strict2*[simp]:
 (*null::OclAny*) .*oclIsTypeOf*(*OclAny*) = *true*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def invalid-def*
OclIsTypeOfOclAny-OclAny)
lemma *OclIsTypeOfOclAny-Person-strict1*[simp]:
 (*invalid::Person*) .*oclIsTypeOf*(*OclAny*) = *invalid*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def invalid-def*
OclIsTypeOfOclAny-Person)
lemma *OclIsTypeOfOclAny-Person-strict2*[simp]:
 (*null::Person*) .*oclIsTypeOf*(*OclAny*) = *true*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def invalid-def*
OclIsTypeOfOclAny-Person)
lemma *OclIsTypeOfPerson-OclAny-strict1*[simp]:
 (*invalid::OclAny*) .*oclIsTypeOf*(*Person*) = *invalid*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def invalid-def*
OclIsTypeOfPerson-OclAny)
lemma *OclIsTypeOfPerson-OclAny-strict2*[simp]:
 (*null::OclAny*) .*oclIsTypeOf*(*Person*) = *true*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-def invalid-def*
OclIsTypeOfPerson-OclAny)
lemma *OclIsTypeOfPerson-Person-strict1*[simp]:
 (*invalid::Person*) .*oclIsTypeOf*(*Person*) = *invalid*
by(*rule ext*, *simp add*: *null-fun-def null-option-def bot-option-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 invalid-def*
OclIsTypeOfPerson-Person)

4.5.4. Up Down Casting

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 invalid-def
      OclAsTypeOclAny-Person OclAsTypePerson-OclAny foundation22 foundation16
      split: option.split typeOclAny.split typePerson.split)
by(simp add: OclIsTypeOfOclAny-OclAny OclValid-def false-def true-def)

```

```

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

```

```

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

```

```

lemma up-down-cast-Person-OclAny-Person [simp]:
shows  $((X :: \text{Person}) . \text{oclAsType}(\text{OclAny}) . \text{oclAsType}(\text{Person}) = X)$ 
apply(rule ext, rename-tac  $\tau$ )
apply(rule foundation22[THEN iffD1])
apply(case-tac  $\tau \models (\delta X)$ , simp add: up-down-cast)
apply(simp add: defined-split, 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

```

4.6. OclIsKindOf

4.6.1. Definition

```

consts OclIsKindOfOclAny :: ' $\alpha \Rightarrow \text{Boolean}$  ( $\langle (-) . \text{oclIsKindOf}'(\text{OclAny}) \rangle$ )
consts OclIsKindOfPerson :: ' $\alpha \Rightarrow \text{Boolean}$  ( $\langle (-) . \text{oclIsKindOf}'(\text{Person}) \rangle$ )

```

overloading $\text{OclIsKindOf}_{\text{OclAny}} \equiv \text{OclIsKindOf}_{\text{OclAny}} :: \text{OclAny} \Rightarrow \text{Boolean}$

begin

```

definition OclIsKindOfOclAny-OclAny:
   $(X :: \text{OclAny}) . \text{oclIsKindOf}(\text{OclAny}) \equiv$ 
   $(\lambda \tau . \text{case } X \ \tau \ \text{of}$ 

```

$$\begin{array}{l} \perp \Rightarrow \text{invalid } \tau \\ | - \Rightarrow \text{true } \tau \end{array}$$

end

overloading $OclIsKindOf_{OclAny} \equiv OclIsKindOf_{OclAny} :: Person \Rightarrow Boolean$

begin

definition $OclIsKindOf_{OclAny}\text{-}Person$:

$$\begin{array}{l} (X::Person) .oclIsKindOf(OclAny) \equiv \\ (\lambda\tau. \text{case } X \ \tau \ \text{of} \\ \quad \perp \Rightarrow \text{invalid } \tau \\ \quad | - \Rightarrow \text{true } \tau) \end{array}$$

end

overloading $OclIsKindOf_{Person} \equiv OclIsKindOf_{Person} :: OclAny \Rightarrow Boolean$

begin

definition $OclIsKindOf_{Person}\text{-}OclAny$:

$$\begin{array}{l} (X::OclAny) .oclIsKindOf(Person) \equiv \\ (\lambda\tau. \text{case } X \ \tau \ \text{of} \\ \quad \perp \Rightarrow \text{invalid } \tau \\ \quad | \perp_{\perp} \Rightarrow \text{true } \tau \\ \quad | \perp_{mkOclAny} \ \text{oid} \ \perp_{\perp} \Rightarrow \text{false } \tau \\ \quad | \perp_{mkOclAny} \ \text{oid} \ \perp_{\perp} \Rightarrow \text{true } \tau) \end{array}$$

end

overloading $OclIsKindOf_{Person} \equiv OclIsKindOf_{Person} :: Person \Rightarrow Boolean$

begin

definition $OclIsKindOf_{Person}\text{-}Person$:

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

end

4.6.2. Context Passing

lemma $cp\text{-}OclIsKindOf_{OclAny}\text{-}Person\text{-}Person$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::Person)::Person).oclIsKindOf(OclAny))$

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

lemma $cp\text{-}OclIsKindOf_{OclAny}\text{-}OclAny\text{-}OclAny$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::OclAny).oclIsKindOf(OclAny))$

by(rule $cpI1$, simp-all add: $OclIsKindOf_{OclAny}\text{-}OclAny$)

lemma $cp\text{-}OclIsKindOf_{Person}\text{-}Person\text{-}Person$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::Person)::Person).oclIsKindOf(Person))$

by(rule $cpI1$, simp-all add: $OclIsKindOf_{Person}\text{-}Person$)

lemma $cp\text{-}OclIsKindOf_{Person}\text{-}OclAny\text{-}OclAny$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::OclAny).oclIsKindOf(Person))$

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

lemma $cp\text{-}OclIsKindOf_{OclAny}\text{-}Person\text{-}OclAny$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(OclAny))$

by(rule $cpI1$, simp-all add: $OclIsKindOf_{OclAny}\text{-}OclAny$)

lemma $cp\text{-}OclIsKindOf_{OclAny}\text{-}OclAny\text{-}Person$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(OclAny))$

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

lemma $cp\text{-}OclIsKindOf_{Person}\text{-}Person\text{-}OclAny$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(Person))$

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

lemma $cp\text{-}OclIsKindOf_{Person}\text{-}OclAny\text{-}Person$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(Person))$

by(rule $cpI1$, simp-all add: $OclIsKindOf_{Person}\text{-}Person$)

lemmas [simp] =

$cp\text{-}OclIsKindOf_{OclAny}\text{-}Person\text{-}Person$

$cp\text{-}OclIsKindOf_{OclAny}\text{-}OclAny\text{-}OclAny$

cp-OclIsKindOf_{Person}-Person-Person
cp-OclIsKindOf_{Person}-OclAny-OclAny

cp-OclIsKindOf_{OclAny}-Person-OclAny
cp-OclIsKindOf_{OclAny}-OclAny-Person
cp-OclIsKindOf_{Person}-Person-OclAny
cp-OclIsKindOf_{Person}-OclAny-Person

4.6.3. Execution with Invalid or Null as Argument

lemma *OclIsKindOf_{OclAny}-OclAny-strict1*[simp] : (*invalid::OclAny*) .*oclIsKindOf*(*OclAny*) = *invalid*
by(*rule ext*, *simp add*: *invalid-def bot-option-def*
OclIsKindOf_{OclAny}-OclAny)

lemma *OclIsKindOf_{OclAny}-OclAny-strict2*[simp] : (*null::OclAny*) .*oclIsKindOf*(*OclAny*) = *true*
by(*rule ext*, *simp add*: *null-fun-def null-option-def*
OclIsKindOf_{OclAny}-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 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 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 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 invalid-def*
OclIsKindOf_{Person}-Person)

4.6.4. Up Down Casting

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 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*)

4.7. OclAllInstances

To denote OCL-types occurring in OCL expressions syntactically—as, for example, as “argument” of `oclAllInstances()`—we use the inverses of the injection functions into the object universes; we show that this is sufficient “characterization.”

definition $Person \equiv OclAsType_{Person}\text{-}\mathfrak{A}$

definition $OclAny \equiv OclAsType_{OclAny}\text{-}\mathfrak{A}$

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

lemma $OclAllInstances\text{-}generic_{OclAny}\text{-}exec: OclAllInstances\text{-}generic\ pre\text{-}post\ OclAny =$
 $(\lambda\tau. Abs\text{-}Set_{base}\ \perp\ Some\ 'OclAny'\ ran\ (heap\ (pre\text{-}post\ \tau))\ \perp)$

proof –

let $?S1 = \lambda\tau. OclAny\ 'ran\ (heap\ (pre\text{-}post\ \tau))$

let $?S2 = \lambda\tau. ?S1\ \tau - \{None\}$

have $B : \bigwedge\tau. ?S2\ \tau \subseteq ?S1\ \tau$ **by** *auto*

have $C : \bigwedge\tau. ?S1\ \tau \subseteq ?S2\ \tau$ **by** (*auto simp: OclAsType_{OclAny}\text{-}\mathfrak{A}\text{-}some*)

show $?thesis$ **by** (*insert equalityI[OF B C], simp*)

qed

lemma $OclAllInstances\text{-}at\text{-}post_{OclAny}\text{-}exec: OclAny.\ allInstances() =$
 $(\lambda\tau. Abs\text{-}Set_{base}\ \perp\ Some\ 'OclAny'\ ran\ (heap\ (snd\ \tau))\ \perp)$

unfolding $OclAllInstances\text{-}at\text{-}post\text{-}def$

by (*rule OclAllInstances\text{-}generic_{OclAny}\text{-}exec*)

lemma $OclAllInstances\text{-}at\text{-}pre_{OclAny}\text{-}exec: OclAny.\ allInstances@pre() =$
 $(\lambda\tau. Abs\text{-}Set_{base}\ \perp\ Some\ 'OclAny'\ ran\ (heap\ (fst\ \tau))\ \perp)$

unfolding $OclAllInstances\text{-}at\text{-}pre\text{-}def$

by (*rule OclAllInstances\text{-}generic_{OclAny}\text{-}exec*)

4.7.1. OclIsTypeOf

lemma $OclAny\text{-}allInstances\text{-}generic\text{-}oclIsTypeOf_{OclAny}1:$

assumes $[simp]: \bigwedge x. pre\text{-}post\ (x, x) = x$

shows $\exists\tau. (\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ OclAny)\text{-}>forAll_{Set}(X|X.\ oclIsTypeOf(OclAny))))$

apply (*rule-tac x = τ_0 in exI, simp add: $\tau_0\text{-}def\ OclValid\text{-}def\ del: OclAllInstances\text{-}generic\text{-}def$*)

apply (*simp only: assms UML-Set.OclForall-def refl if-True*)

$OclAllInstances\text{-}generic\text{-}defined[simplified\ OclValid\text{-}def]$

apply (*simp only: OclAllInstances\text{-}generic\text{-}def*)

apply (*subst (1 2 3) Abs-Set_{base}\text{-}inverse, simp add: bot-option-def*)

by (*simp add: OclIsTypeOf_{OclAny}\text{-}OclAny*)

lemma $OclAny\text{-}allInstances\text{-}at\text{-}post\text{-}oclIsTypeOf_{OclAny}1:$

$\exists\tau. (\tau \models (OclAny.\ allInstances()\text{-}>forAll_{Set}(X|X.\ oclIsTypeOf(OclAny))))$

unfolding $OclAllInstances\text{-}at\text{-}post\text{-}def$

by (*rule OclAny\text{-}allInstances\text{-}generic\text{-}oclIsTypeOf_{OclAny}1, simp*)

lemma $OclAny\text{-}allInstances\text{-}at\text{-}pre\text{-}oclIsTypeOf_{OclAny}1:$

$\exists\tau. (\tau \models (OclAny.\ allInstances@pre()\text{-}>forAll_{Set}(X|X.\ oclIsTypeOf(OclAny))))$

unfolding $OclAllInstances\text{-}at\text{-}pre\text{-}def$

by (*rule OclAny\text{-}allInstances\text{-}generic\text{-}oclIsTypeOf_{OclAny}1, simp*)

lemma $OclAny\text{-}allInstances\text{-}generic\text{-}oclIsTypeOf_{OclAny}2:$

assumes $[simp]: \bigwedge x. pre\text{-}post\ (x, x) = x$

shows $\exists\tau. (\tau \models not\ ((OclAllInstances\text{-}generic\ pre\text{-}post\ OclAny)\text{-}>forAll_{Set}(X|X.\ oclIsTypeOf(OclAny))))$

proof – **fix** oid **let** $?t0 = (heap = Map.empty(oid \mapsto in_{OclAny}\ (mk_{OclAny}\ oid\ \perp_a)))$,

$assoc = Map.empty$) **show** $?thesis$

apply(*rule-tac* $x = (?t0, ?t0)$ **in** exI , *simp add*: *OclValid-def del: OclAllInstances-generic-def*)
apply(*simp only*: *UML-Set.OclForall-def refl if-True*
OclAllInstances-generic-defined[simplified OclValid-def])
apply(*simp only*: *OclAllInstances-generic-def OclAsTypeOclAny- \mathfrak{A} -def*)
apply(*subst* (1 2 3) *Abs-Set_{base}-inverse*, *simp add*: *bot-option-def*)
by(*simp add*: *OclIsTypeOfOclAny-OclAny OclNot-def OclAny-def*)
qed

lemma *OclAny-allInstances-at-post-oclIsTypeOfOclAny2*:
 $\exists \tau. (\tau \models \text{not } (OclAny .allInstances()) \rightarrow \text{forAll}_{Set}(X|X .oclIsTypeOf(OclAny)))$
unfolding *OclAllInstances-at-post-def*
by(*rule OclAny-allInstances-generic-oclIsTypeOfOclAny2*, *simp*)

lemma *OclAny-allInstances-at-pre-oclIsTypeOfOclAny2*:
 $\exists \tau. (\tau \models \text{not } (OclAny .allInstances@pre()) \rightarrow \text{forAll}_{Set}(X|X .oclIsTypeOf(OclAny)))$
unfolding *OclAllInstances-at-pre-def*
by(*rule OclAny-allInstances-generic-oclIsTypeOfOclAny2*, *simp*)

lemma *Person-allInstances-generic-oclIsTypeOfPerson*:
 $\tau \models ((OclAllInstances-generic \text{ pre-post } Person) \rightarrow \text{forAll}_{Set}(X|X .oclIsTypeOf(Person)))$
apply(*simp add*: *OclValid-def del: OclAllInstances-generic-def*)
apply(*simp only*: *UML-Set.OclForall-def refl if-True*
OclAllInstances-generic-defined[simplified OclValid-def])
apply(*simp only*: *OclAllInstances-generic-def*)
apply(*subst* (1 2 3) *Abs-Set_{base}-inverse*, *simp add*: *bot-option-def*)
by(*simp add*: *OclIsTypeOfPerson-Person*)

lemma *Person-allInstances-at-post-oclIsTypeOfPerson*:
 $\tau \models (Person .allInstances() \rightarrow \text{forAll}_{Set}(X|X .oclIsTypeOf(Person)))$
unfolding *OclAllInstances-at-post-def*
by(*rule Person-allInstances-generic-oclIsTypeOfPerson*)

lemma *Person-allInstances-at-pre-oclIsTypeOfPerson*:
 $\tau \models (Person .allInstances@pre() \rightarrow \text{forAll}_{Set}(X|X .oclIsTypeOf(Person)))$
unfolding *OclAllInstances-at-pre-def*
by(*rule Person-allInstances-generic-oclIsTypeOfPerson*)

4.7.2. OclIsKindOf

lemma *OclAny-allInstances-generic-oclIsKindOfOclAny*:
 $\tau \models ((OclAllInstances-generic \text{ pre-post } OclAny) \rightarrow \text{forAll}_{Set}(X|X .oclIsKindOf(OclAny)))$
apply(*simp add*: *OclValid-def del: OclAllInstances-generic-def*)
apply(*simp only*: *UML-Set.OclForall-def refl if-True*
OclAllInstances-generic-defined[simplified OclValid-def])
apply(*simp only*: *OclAllInstances-generic-def*)
apply(*subst* (1 2 3) *Abs-Set_{base}-inverse*, *simp add*: *bot-option-def*)
by(*simp add*: *OclIsKindOfOclAny-OclAny*)

lemma *OclAny-allInstances-at-post-oclIsKindOfOclAny*:
 $\tau \models (OclAny .allInstances() \rightarrow \text{forAll}_{Set}(X|X .oclIsKindOf(OclAny)))$
unfolding *OclAllInstances-at-post-def*
by(*rule OclAny-allInstances-generic-oclIsKindOfOclAny*)

lemma *OclAny-allInstances-at-pre-oclIsKindOfOclAny*:
 $\tau \models (OclAny .allInstances@pre() \rightarrow \text{forAll}_{Set}(X|X .oclIsKindOf(OclAny)))$
unfolding *OclAllInstances-at-pre-def*
by(*rule OclAny-allInstances-generic-oclIsKindOfOclAny*)

lemma *Person-allInstances-generic-oclIsKindOf_{OclAny}*:
 $\tau \models ((\text{OclAllInstances-generic pre-post Person}) \rightarrow \text{forAll}_{\text{Set}}(X|X . \text{oclIsKindOf}(\text{OclAny})))$
apply(simp add: OclValid-def del: OclAllInstances-generic-def)
apply(simp only: UML-Set.OclForall-def refl if-True
OclAllInstances-generic-defined[simplified OclValid-def])
apply(simp only: OclAllInstances-generic-def)
apply(subst (1 2 3) Abs-Set_{base}-inverse, simp add: bot-option-def)
by(simp add: OclIsKindOf_{OclAny}-Person)

lemma *Person-allInstances-at-post-oclIsKindOf_{OclAny}*:
 $\tau \models (\text{Person} . \text{allInstances}() \rightarrow \text{forAll}_{\text{Set}}(X|X . \text{oclIsKindOf}(\text{OclAny})))$
unfolding OclAllInstances-at-post-def
by(rule Person-allInstances-generic-oclIsKindOf_{OclAny})

lemma *Person-allInstances-at-pre-oclIsKindOf_{OclAny}*:
 $\tau \models (\text{Person} . \text{allInstances}@pre() \rightarrow \text{forAll}_{\text{Set}}(X|X . \text{oclIsKindOf}(\text{OclAny})))$
unfolding OclAllInstances-at-pre-def
by(rule Person-allInstances-generic-oclIsKindOf_{OclAny})

lemma *Person-allInstances-generic-oclIsKindOf_{Person}*:
 $\tau \models ((\text{OclAllInstances-generic pre-post Person}) \rightarrow \text{forAll}_{\text{Set}}(X|X . \text{oclIsKindOf}(\text{Person})))$
apply(simp add: OclValid-def del: OclAllInstances-generic-def)
apply(simp only: UML-Set.OclForall-def refl if-True
OclAllInstances-generic-defined[simplified OclValid-def])
apply(simp only: OclAllInstances-generic-def)
apply(subst (1 2 3) Abs-Set_{base}-inverse, simp add: bot-option-def)
by(simp add: OclIsKindOf_{Person}-Person)

lemma *Person-allInstances-at-post-oclIsKindOf_{Person}*:
 $\tau \models (\text{Person} . \text{allInstances}() \rightarrow \text{forAll}_{\text{Set}}(X|X . \text{oclIsKindOf}(\text{Person})))$
unfolding OclAllInstances-at-post-def
by(rule Person-allInstances-generic-oclIsKindOf_{Person})

lemma *Person-allInstances-at-pre-oclIsKindOf_{Person}*:
 $\tau \models (\text{Person} . \text{allInstances}@pre() \rightarrow \text{forAll}_{\text{Set}}(X|X . \text{oclIsKindOf}(\text{Person})))$
unfolding OclAllInstances-at-pre-def
by(rule Person-allInstances-generic-oclIsKindOf_{Person})

4.8. The Accessors (any, boss, salary)

Should be generated entirely from a class-diagram.

4.8.1. Definition (of the association Employee-Boss)

We start with a oid for the association; this oid can be used in presence of association classes to represent the association inside an object, pretty much similar to the Design_UML, where we stored an oid inside the class as “pointer.”

definition *oid_{Person}BOSS* ::oid **where** *oid_{Person}BOSS* = 10

From there on, we can already define an empty state which must contain for *oid_{Person}BOSS* the empty relation (encoded as association list, since there are associations with a Sequence-like structure).

definition *eval-extract* :: (' \mathcal{A} , ('a::object) option option) val
 \Rightarrow (oid \Rightarrow (' \mathcal{A} , 'c::null) val)
 \Rightarrow (' \mathcal{A} , 'c::null) val

where *eval-extract* $X f = (\lambda \tau. \text{case } X \tau \text{ of}$
 $\perp \Rightarrow \text{invalid } \tau$ — exception propagation
 $\lfloor \perp \rfloor \Rightarrow \text{invalid } \tau$ — dereferencing null pointer
 $\lfloor \perp \rfloor \text{obj } \perp \Rightarrow f (\text{oid-of obj}) \tau$

definition *choose₂₋₁* = *fst*

definition *choose₂₋₂* = *snd*

definition *List-flatten* = $(\lambda l. (\text{foldl } ((\lambda \text{acc}. (\lambda l. (\text{foldl } ((\lambda \text{acc}. (\lambda l. (\text{Cons } l) (\text{acc})))) (\text{acc}) ((\text{rev } l)))))) (\text{rev } l)))) (\text{Nil})$

definition *deref-assocs₂* :: $(\mathfrak{A} \text{ state} \times \mathfrak{A} \text{ state} \Rightarrow \mathfrak{A} \text{ state})$
 $\Rightarrow (\text{oid list list} \Rightarrow \text{oid list} \times \text{oid list})$
 $\Rightarrow \text{oid}$
 $\Rightarrow (\text{oid list} \Rightarrow (\mathfrak{A}, 'f) \text{val})$
 $\Rightarrow \text{oid}$
 $\Rightarrow (\mathfrak{A}, 'f::\text{null}) \text{val}$

where *deref-assocs₂* *pre-post to-from assoc-oid f oid* =
 $(\lambda \tau. \text{case } (\text{assocs } (\text{pre-post } \tau)) \text{ assoc-oid of}$
 $\lfloor S \rfloor \Rightarrow f (\text{List-flatten } (\text{map } (\text{choose}_{2-2} \circ \text{to-from})$
 $(\text{filter } (\lambda p. \text{List.member } (\text{choose}_{2-1} (\text{to-from } p)) \text{oid}) S)))$
 τ
 $\lfloor - \rfloor \Rightarrow \text{invalid } \tau$

The *pre-post*-parameter is configured with *fst* or *snd*, the *to-from*-parameter either with the identity *id* or the following combinator *switch*:

definition *switch₂₋₁* = $(\lambda [x,y] \Rightarrow (x,y))$

definition *switch₂₋₂* = $(\lambda [x,y] \Rightarrow (y,x))$

definition *switch₃₋₁* = $(\lambda [x,y,z] \Rightarrow (x,y))$

definition *switch₃₋₂* = $(\lambda [x,y,z] \Rightarrow (x,z))$

definition *switch₃₋₃* = $(\lambda [x,y,z] \Rightarrow (y,x))$

definition *switch₃₋₄* = $(\lambda [x,y,z] \Rightarrow (y,z))$

definition *switch₃₋₅* = $(\lambda [x,y,z] \Rightarrow (z,x))$

definition *switch₃₋₆* = $(\lambda [x,y,z] \Rightarrow (z,y))$

definition *deref-oid_{Person}* :: $(\mathfrak{A} \text{ state} \times \mathfrak{A} \text{ state} \Rightarrow \mathfrak{A} \text{ state})$
 $\Rightarrow (\text{type}_{\text{Person}} \Rightarrow (\mathfrak{A}, 'c::\text{null}) \text{val})$
 $\Rightarrow \text{oid}$
 $\Rightarrow (\mathfrak{A}, 'c::\text{null}) \text{val}$

where *deref-oid_{Person}* *fst-snd f oid* = $(\lambda \tau. \text{case } (\text{heap } (\text{fst-snd } \tau)) \text{ oid of}$
 $\lfloor \text{in}_{\text{Person}} \text{obj } \rfloor \Rightarrow f \text{obj } \tau$
 $\lfloor - \rfloor \Rightarrow \text{invalid } \tau$

definition *deref-oid_{OclAny}* :: $(\mathfrak{A} \text{ state} \times \mathfrak{A} \text{ state} \Rightarrow \mathfrak{A} \text{ state})$
 $\Rightarrow (\text{type}_{\text{OclAny}} \Rightarrow (\mathfrak{A}, 'c::\text{null}) \text{val})$
 $\Rightarrow \text{oid}$
 $\Rightarrow (\mathfrak{A}, 'c::\text{null}) \text{val}$

where *deref-oid_{OclAny}* *fst-snd f oid* = $(\lambda \tau. \text{case } (\text{heap } (\text{fst-snd } \tau)) \text{ oid of}$
 $\lfloor \text{in}_{\text{OclAny}} \text{obj } \rfloor \Rightarrow f \text{obj } \tau$
 $\lfloor - \rfloor \Rightarrow \text{invalid } \tau$

pointer undefined in state or not referencing a type conform object representation

definition *select_{OclAny}ANY* *f* = $(\lambda X. \text{case } X \text{ of}$
 $(\text{mk}_{\text{OclAny}} - \perp) \Rightarrow \text{null}$
 $\lfloor (\text{mk}_{\text{OclAny}} - \perp \text{any}) \rfloor \Rightarrow f (\lambda x - \lfloor x \rfloor) \text{any})$

definition $select_{Person}BOSS f = select-object mtSet UML-Set.OclIncluding UML-Set.OclANY (f (\lambda x -. \perp x))$

definition $select_{Person}SALARY f = (\lambda X. case X of$
 $(mk_{Person} \perp) \Rightarrow null$
 $| (mk_{Person} \perp salary) \Rightarrow f (\lambda x -. \perp x)) salary)$

definition $deref-assocs_2BOSS fst-snd f = (\lambda mk_{Person} oid - \Rightarrow$
 $deref-assocs_2 fst-snd switch_2-1 oid_{Person}BOSS f oid)$

definition $in-pre-state = fst$

definition $in-post-state = snd$

definition $reconst-basetype = (\lambda convert x. convert x)$

definition $dot_{OclAny}ANY :: OclAny \Rightarrow - (\langle(1(-).any)\rangle 50)$
where $(X).any = eval-extract X$
 $(deref-oid_{OclAny} in-post-state$
 $(select_{OclAny}ANY$
 $reconst-basetype))$

definition $dot_{Person}BOSS :: Person \Rightarrow Person (\langle(1(-).boss)\rangle 50)$
where $(X).boss = eval-extract X$
 $(deref-oid_{Person} in-post-state$
 $(deref-assocs_2BOSS in-post-state$
 $(select_{Person}BOSS$
 $(deref-oid_{Person} in-post-state))))$

definition $dot_{Person}SALARY :: Person \Rightarrow Integer (\langle(1(-).salary)\rangle 50)$
where $(X).salary = eval-extract X$
 $(deref-oid_{Person} in-post-state$
 $(select_{Person}SALARY$
 $reconst-basetype))$

definition $dot_{OclAny}ANY-at-pre :: OclAny \Rightarrow - (\langle(1(-).any@pre)\rangle 50)$
where $(X).any@pre = eval-extract X$
 $(deref-oid_{OclAny} in-pre-state$
 $(select_{OclAny}ANY$
 $reconst-basetype))$

definition $dot_{Person}BOSS-at-pre :: Person \Rightarrow Person (\langle(1(-).boss@pre)\rangle 50)$
where $(X).boss@pre = eval-extract X$
 $(deref-oid_{Person} in-pre-state$
 $(deref-assocs_2BOSS in-pre-state$
 $(select_{Person}BOSS$
 $(deref-oid_{Person} in-pre-state))))$

definition $dot_{Person}SALARY-at-pre :: Person \Rightarrow Integer (\langle(1(-).salary@pre)\rangle 50)$
where $(X).salary@pre = eval-extract X$
 $(deref-oid_{Person} in-pre-state$
 $(select_{Person}SALARY$
 $reconst-basetype))$

lemmas $dot-accessor =$
 $dot_{OclAny}ANY-def$

$\text{dot}_{Person}BOSS\text{-def}$
 $\text{dot}_{Person}SALARY\text{-def}$
 $\text{dot}_{OclAny}ANY\text{-at-pre-def}$
 $\text{dot}_{Person}BOSS\text{-at-pre-def}$
 $\text{dot}_{Person}SALARY\text{-at-pre-def}$

4.8.2. Context Passing

lemmas $[\text{simp}] = \text{eval-extract-def}$

lemma $\text{cp-dot}_{OclAny}ANY$: $((X).any) \tau = ((\lambda-. X \tau).any) \tau$ **by** ($\text{simp add: dot-accessor}$)

lemma $\text{cp-dot}_{Person}BOSS$: $((X).boss) \tau = ((\lambda-. X \tau).boss) \tau$ **by** ($\text{simp add: dot-accessor}$)

lemma $\text{cp-dot}_{Person}SALARY$: $((X).salary) \tau = ((\lambda-. X \tau).salary) \tau$ **by** ($\text{simp add: dot-accessor}$)

lemma $\text{cp-dot}_{OclAny}ANY\text{-at-pre}$: $((X).any@pre) \tau = ((\lambda-. X \tau).any@pre) \tau$ **by** ($\text{simp add: dot-accessor}$)

lemma $\text{cp-dot}_{Person}BOSS\text{-at-pre}$: $((X).boss@pre) \tau = ((\lambda-. X \tau).boss@pre) \tau$ **by** ($\text{simp add: dot-accessor}$)

lemma $\text{cp-dot}_{Person}SALARY\text{-at-pre}$: $((X).salary@pre) \tau = ((\lambda-. X \tau).salary@pre) \tau$ **by** ($\text{simp add: dot-accessor}$)

lemmas $\text{cp-dot}_{OclAny}ANY\text{-I}$ $[\text{simp, intro!}] =$
 $\text{cp-dot}_{OclAny}ANY[\text{THEN all}[\text{THEN all}],$
 $\text{of } \lambda X -. X \lambda - \tau. \tau, \text{ THEN cpI1}]$

lemmas $\text{cp-dot}_{OclAny}ANY\text{-at-pre-I}$ $[\text{simp, intro!}] =$
 $\text{cp-dot}_{OclAny}ANY\text{-at-pre}[\text{THEN all}[\text{THEN all}],$
 $\text{of } \lambda X -. X \lambda - \tau. \tau, \text{ THEN cpI1}]$

lemmas $\text{cp-dot}_{Person}BOSS\text{-I}$ $[\text{simp, intro!}] =$
 $\text{cp-dot}_{Person}BOSS[\text{THEN all}[\text{THEN all}],$
 $\text{of } \lambda X -. X \lambda - \tau. \tau, \text{ THEN cpI1}]$

lemmas $\text{cp-dot}_{Person}BOSS\text{-at-pre-I}$ $[\text{simp, intro!}] =$
 $\text{cp-dot}_{Person}BOSS\text{-at-pre}[\text{THEN all}[\text{THEN all}],$
 $\text{of } \lambda X -. X \lambda - \tau. \tau, \text{ THEN cpI1}]$

lemmas $\text{cp-dot}_{Person}SALARY\text{-I}$ $[\text{simp, intro!}] =$
 $\text{cp-dot}_{Person}SALARY[\text{THEN all}[\text{THEN all}],$
 $\text{of } \lambda X -. X \lambda - \tau. \tau, \text{ THEN cpI1}]$

lemmas $\text{cp-dot}_{Person}SALARY\text{-at-pre-I}$ $[\text{simp, intro!}] =$
 $\text{cp-dot}_{Person}SALARY\text{-at-pre}[\text{THEN all}[\text{THEN all}],$
 $\text{of } \lambda X -. X \lambda - \tau. \tau, \text{ THEN cpI1}]$

4.8.3. Execution with Invalid or Null as Argument

lemma $\text{dot}_{OclAny}ANY\text{-nullstrict}$ $[\text{simp}]$: $(null).any = \text{invalid}$
by($\text{rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def}$)

lemma $\text{dot}_{OclAny}ANY\text{-at-pre-nullstrict}$ $[\text{simp}]$: $(null).any@pre = \text{invalid}$

by($\text{rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def}$)

lemma $\text{dot}_{OclAny}ANY\text{-strict}$ $[\text{simp}]$: $(invalid).any = \text{invalid}$

by($\text{rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def}$)

lemma $\text{dot}_{OclAny}ANY\text{-at-pre-strict}$ $[\text{simp}]$: $(invalid).any@pre = \text{invalid}$

by($\text{rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def}$)

lemma $\text{dot}_{Person}BOSS\text{-nullstrict}$ $[\text{simp}]$: $(null).boss = \text{invalid}$

by($\text{rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def}$)

lemma $\text{dot}_{Person}BOSS\text{-at-pre-nullstrict}$ $[\text{simp}]$: $(null).boss@pre = \text{invalid}$

by($\text{rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def}$)

lemma $\text{dot}_{Person}BOSS\text{-strict}$ $[\text{simp}]$: $(invalid).boss = \text{invalid}$

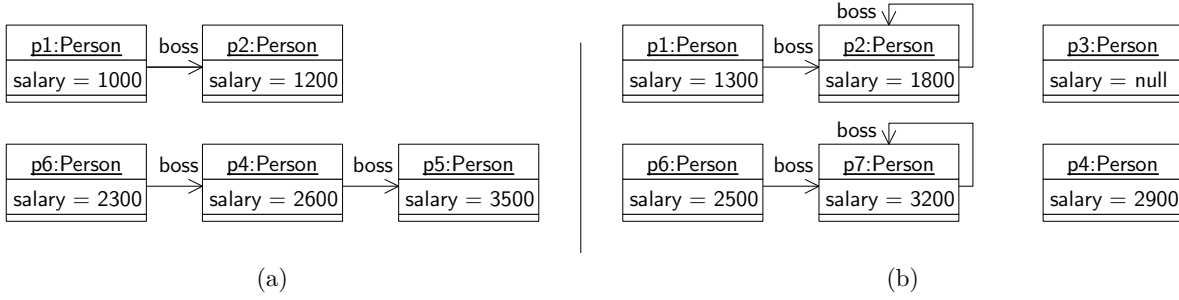


Figure 4.2.: (a) pre-state σ_1 and (b) post-state σ'_1 .

```
by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def)
lemma dot_Person_BOSS-at-pre-strict [simp] : (invalid).boss@pre = invalid
by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def)
```

```
lemma dot_Person_SALARY-nullstrict [simp] : (null).salary = invalid
by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def)
lemma dot_Person_SALARY-at-pre-nullstrict [simp] : (null).salary@pre = invalid
by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def)
lemma dot_Person_SALARY-strict [simp] : (invalid).salary = invalid
by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def)
lemma dot_Person_SALARY-at-pre-strict [simp] : (invalid).salary@pre = invalid
by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def)
```

4.8.4. Representation in States

```
lemma dot_Person_BOSS-def-mono:  $\tau \models \delta(X . boss) \implies \tau \models \delta(X)$ 
  apply(case-tac  $\tau \models (X \triangleq invalid)$ , insert StrongEq-L-subst2[where  $P = (\lambda x. (\delta (x . boss)))$ ] and  $\tau = \tau$  and  $x = X$  and  $y = invalid$ ], simp add: foundation16')
  apply(case-tac  $\tau \models (X \triangleq null)$ , insert StrongEq-L-subst2[where  $P = (\lambda x. (\delta (x . boss)))$ ] and  $\tau = \tau$  and  $x = X$  and  $y = null$ ], simp add: foundation16')
by(simp add: defined-split)
```

```
lemma repr-boss:
  assumes  $A : \tau \models \delta(x . boss)$ 
  shows is-represented-in-state in-post-state (x . boss) Person  $\tau$ 
  apply(insert A[simplified foundation16])
  apply(THEN dot_Person_BOSS-def-mono, simplified foundation16)
  unfolding is-represented-in-state-def
  dot_Person_BOSS-def eval-extract-def select_Person_BOSS-def in-post-state-def
  oops
```

```
lemma repr-bossX :
  assumes  $A : \tau \models \delta(x . boss)$ 
  shows  $\tau \models ((Person . allInstances()) \rightarrow includes_{Set}(x . boss))$ 
  oops
```

4.9. A Little Infra-structure on Example States

The example we are defining in this section comes from the figure 4.2.

```
definition OclInt1000 (<1000>) where OclInt1000 = ( $\lambda . . \underline{\underline{1000}}$ )
definition OclInt1200 (<1200>) where OclInt1200 = ( $\lambda . . \underline{\underline{1200}}$ )
```

definition $OclInt1300$ ($\langle 1300 \rangle$) **where** $OclInt1300 = (\lambda - . \llbracket 1300 \rrbracket)$
definition $OclInt1800$ ($\langle 1800 \rangle$) **where** $OclInt1800 = (\lambda - . \llbracket 1800 \rrbracket)$
definition $OclInt2600$ ($\langle 2600 \rangle$) **where** $OclInt2600 = (\lambda - . \llbracket 2600 \rrbracket)$
definition $OclInt2900$ ($\langle 2900 \rangle$) **where** $OclInt2900 = (\lambda - . \llbracket 2900 \rrbracket)$
definition $OclInt3200$ ($\langle 3200 \rangle$) **where** $OclInt3200 = (\lambda - . \llbracket 3200 \rrbracket)$
definition $OclInt3500$ ($\langle 3500 \rangle$) **where** $OclInt3500 = (\lambda - . \llbracket 3500 \rrbracket)$

definition $oid0 \equiv 0$
definition $oid1 \equiv 1$
definition $oid2 \equiv 2$
definition $oid3 \equiv 3$
definition $oid4 \equiv 4$
definition $oid5 \equiv 5$
definition $oid6 \equiv 6$
definition $oid7 \equiv 7$
definition $oid8 \equiv 8$

definition $person1 \equiv mk_{Person} \text{ } oid0 \llbracket 1300 \rrbracket$
definition $person2 \equiv mk_{Person} \text{ } oid1 \llbracket 1800 \rrbracket$
definition $person3 \equiv mk_{Person} \text{ } oid2 \text{ } None$
definition $person4 \equiv mk_{Person} \text{ } oid3 \llbracket 2900 \rrbracket$
definition $person5 \equiv mk_{Person} \text{ } oid4 \llbracket 3500 \rrbracket$
definition $person6 \equiv mk_{Person} \text{ } oid5 \llbracket 2500 \rrbracket$
definition $person7 \equiv mk_{OclAny} \text{ } oid6 \llbracket 3200 \rrbracket$
definition $person8 \equiv mk_{OclAny} \text{ } oid7 \text{ } None$
definition $person9 \equiv mk_{Person} \text{ } oid8 \llbracket 0 \rrbracket$ **definition**

$\sigma_1 \equiv (\text{heap} = \text{Map.empty}(oid0 \mapsto in_{Person} (mk_{Person} \text{ } oid0 \llbracket 1000 \rrbracket),$
 $oid1 \mapsto in_{Person} (mk_{Person} \text{ } oid1 \llbracket 1200 \rrbracket),$
 ~~$oid2 \mapsto in_{Person} (mk_{Person} \text{ } oid2 \llbracket 1500 \rrbracket),$~~
 $oid3 \mapsto in_{Person} (mk_{Person} \text{ } oid3 \llbracket 2600 \rrbracket),$
 $oid4 \mapsto in_{Person} \text{ } person5,$
 $oid5 \mapsto in_{Person} (mk_{Person} \text{ } oid5 \llbracket 2300 \rrbracket),$
 ~~$oid6 \mapsto in_{Person} (mk_{Person} \text{ } oid6 \llbracket 2500 \rrbracket),$~~
 ~~$oid7 \mapsto in_{Person} (mk_{Person} \text{ } oid7 \llbracket 2700 \rrbracket),$~~
 $oid8 \mapsto in_{Person} \text{ } person9),$
 $assocs = \text{Map.empty}(oid_{Person} \text{ } BOSS \mapsto [[oid0],[oid1]], [[oid3],[oid4]], [[oid5],[oid3]]))$

definition

$\sigma_1' \equiv (\text{heap} = \text{Map.empty}(oid0 \mapsto in_{Person} \text{ } person1,$
 $oid1 \mapsto in_{Person} \text{ } person2,$
 $oid2 \mapsto in_{Person} \text{ } person3,$
 $oid3 \mapsto in_{Person} \text{ } person4,$
 ~~$oid4 \mapsto in_{Person} \text{ } person5,$~~
 $oid5 \mapsto in_{Person} \text{ } person6,$
 $oid6 \mapsto in_{OclAny} \text{ } person7,$
 $oid7 \mapsto in_{OclAny} \text{ } person8,$
 $oid8 \mapsto in_{Person} \text{ } person9),$
 $assocs = \text{Map.empty}(oid_{Person} \text{ } BOSS \mapsto [[oid0],[oid1]], [[oid1],[oid1]], [[oid5],[oid6]], [[oid6],[oid6]]))$

definition $\sigma_0 \equiv (\text{heap} = \text{Map.empty}, \text{assocs} = \text{Map.empty})$

lemma *basic- τ -wff*: $WFF(\sigma_1, \sigma_1')$

by (*auto simp*: $WFF\text{-def}$ $\sigma_1\text{-def}$ $\sigma_1'\text{-def}$)

$oid0\text{-def}$ $oid1\text{-def}$ $oid2\text{-def}$ $oid3\text{-def}$ $oid4\text{-def}$ $oid5\text{-def}$ $oid6\text{-def}$ $oid7\text{-def}$ $oid8\text{-def}$
 $oid\text{-of-}\mathfrak{A}\text{-def}$ $oid\text{-of-type}_{Person}\text{-def}$ $oid\text{-of-type}_{OclAny}\text{-def}$

*person1-def person2-def person3-def person4-def
 person5-def person6-def person7-def person8-def person9-def*

lemma [*simp,code-unfold*]: $\text{dom}(\text{heap } \sigma_1) = \{\text{oid0}, \text{oid1}, \text{oid2}, \text{oid3}, \text{oid4}, \text{oid5}, \text{oid6}, \text{oid7}, \text{oid8}\}$
by(*auto simp: σ_1 -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: σ_1' -def*)

definition $X_{\text{Person1}} :: \text{Person} \equiv \lambda \cdot \cdot \perp \text{person1} \perp \perp$

definition $X_{\text{Person2}} :: \text{Person} \equiv \lambda \cdot \cdot \perp \text{person2} \perp \perp$

definition $X_{\text{Person3}} :: \text{Person} \equiv \lambda \cdot \cdot \perp \text{person3} \perp \perp$

definition $X_{\text{Person4}} :: \text{Person} \equiv \lambda \cdot \cdot \perp \text{person4} \perp \perp$

definition $X_{\text{Person5}} :: \text{Person} \equiv \lambda \cdot \cdot \perp \text{person5} \perp \perp$

definition $X_{\text{Person6}} :: \text{Person} \equiv \lambda \cdot \cdot \perp \text{person6} \perp \perp$

definition $X_{\text{Person7}} :: \text{OclAny} \equiv \lambda \cdot \cdot \perp \text{person7} \perp \perp$

definition $X_{\text{Person8}} :: \text{OclAny} \equiv \lambda \cdot \cdot \perp \text{person8} \perp \perp$

definition $X_{\text{Person9}} :: \text{Person} \equiv \lambda \cdot \cdot \perp \text{person9} \perp \perp$

lemma [*code-unfold*]: $((x :: \text{Person}) \doteq y) = \text{StrictRefEq}_{\text{Object}} x y$ **by**(*simp only: StrictRefEq_{Object-Person}*)

lemma [*code-unfold*]: $((x :: \text{OclAny}) \doteq y) = \text{StrictRefEq}_{\text{Object}} x y$ **by**(*simp only: StrictRefEq_{Object-OclAny}*)

lemmas [*simp,code-unfold*] =

OclAsType_{OclAny-OclAny}

OclAsType_{OclAny-Person}

OclAsType_{Person-OclAny}

OclAsType_{Person-Person}

OclIsTypeOf_{OclAny-OclAny}

OclIsTypeOf_{OclAny-Person}

OclIsTypeOf_{Person-OclAny}

OclIsTypeOf_{Person-Person}

OclIsKindOf_{OclAny-OclAny}

OclIsKindOf_{OclAny-Person}

OclIsKindOf_{Person-OclAny}

OclIsKindOf_{Person-Person} **Assert** $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{\text{Person1}} \cdot \text{salary} <> 1000)$

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{\text{Person1}} \cdot \text{salary} \doteq 1300)$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{\text{Person1}} \cdot \text{salary}@pre \doteq 1000)$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{\text{Person1}} \cdot \text{salary}@pre <> 1300)$

lemma $(\sigma_1, \sigma_1') \models (X_{\text{Person1}} \cdot \text{oclIsMaintained}())$

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 $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models ((X_{\text{Person1}} \cdot \text{oclAsType}(\text{OclAny}) \cdot \text{oclAsType}(\text{Person})) \doteq X_{\text{Person1}})$

by(*rule up-down-cast-Person-OclAny-Person', simp add: X_{Person1} -def*)

Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{\text{Person1}} \cdot \text{oclIsTypeOf}(\text{Person}))$

Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models \text{not}(X_{\text{Person1}} \cdot \text{oclIsTypeOf}(\text{OclAny}))$

Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{\text{Person1}} \cdot \text{oclIsKindOf}(\text{Person}))$

Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{\text{Person1}} \cdot \text{oclIsKindOf}(\text{OclAny}))$

Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models \text{not}(X_{\text{Person1}} \cdot \text{oclAsType}(\text{OclAny}) \cdot \text{oclIsTypeOf}(\text{OclAny}))$

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{\text{Person2}} \cdot \text{salary} \doteq 1800)$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{\text{Person2}} \cdot \text{salary}@pre \doteq 1200)$

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*)

Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models (X_{Person3} .salary \doteq null)$
Assert $\bigwedge_{s_{post}} . (s_{pre}, s_{post}) \models not(v(X_{Person3} .salary@pre))$
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*)

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*)

Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models not(v(X_{Person5} .salary))$
Assert $\bigwedge_{s_{post}} . (s_{pre}, s_{post}) \models (X_{Person5} .salary@pre \doteq 3500)$

lemma $(\sigma_1, \sigma_1') \models (X_{Person5} .oclIsDeleted())$
by(*simp add: OclNot-def OclValid-def OclIsDeleted-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*)

Assert $\bigwedge_{s_{pre} s_{post}} . (s_{pre}, s_{post}) \models v(X_{Person7} .oclAsType(Person))$

lemma $\bigwedge_{s_{pre} s_{post}} . (s_{pre}, s_{post}) \models ((X_{Person7} .oclAsType(Person) .oclAsType(OclAny) .oclAsType(Person)) \doteq (X_{Person7} .oclAsType(Person)))$
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*)

Assert $\bigwedge_{s_{pre} s_{post}} . (s_{pre}, s_{post}) \models (X_{Person8} <> X_{Person7})$
Assert $\bigwedge_{s_{pre} s_{post}} . (s_{pre}, s_{post}) \models not(v(X_{Person8} .oclAsType(Person)))$
Assert $\bigwedge_{s_{pre} s_{post}} . (s_{pre}, s_{post}) \models (X_{Person8} .oclIsTypeOf(OclAny))$
Assert $\bigwedge_{s_{pre} s_{post}} . (s_{pre}, s_{post}) \models not(X_{Person8} .oclIsTypeOf(Person))$
Assert $\bigwedge_{s_{pre} s_{post}} . (s_{pre}, s_{post}) \models not(X_{Person8} .oclIsKindOf(Person))$
Assert $\bigwedge_{s_{pre} s_{post}} . (s_{pre}, s_{post}) \models (X_{Person8} .oclIsKindOf(OclAny))$

lemma σ -modifiedonly: $(\sigma_1, \sigma_1') \models (Set\{ X_{Person1} .oclAsType(OclAny)$

```

, XPerson2 .oclAsType(OclAny)
//XPerson3.oclAsType(OclAny)
, XPerson4 .oclAsType(OclAny)
//XPerson5.oclAsType(OclAny)
, XPerson6 .oclAsType(OclAny)
//XPerson7.oclAsType(OclAny)
//XPerson8.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: OclIncluding-rep-set mtSet-rep-set null-option-def bot-option-def)
apply(simp add: oid-of-option-def oid-of-typeOclAny-def, clarsimp)
apply(simp add:  $\sigma_1$ -def  $\sigma_1'$ -def
oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)
done

```

lemma $(\sigma_1, \sigma_1') \models ((X_{Person9} @pre (\lambda x. _OclAsType_{Person} \mathcal{A} x)) \triangleq X_{Person9})$
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}- \mathcal{A} -def)

lemma $(\sigma_1, \sigma_1') \models ((X_{Person9} @post (\lambda x. _OclAsType_{Person} \mathcal{A} x)) \triangleq X_{Person9})$
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}- \mathcal{A} -def)

lemma $(\sigma_1, \sigma_1') \models (((X_{Person9} .oclAsType(OclAny)) @pre (\lambda x. _OclAsType_{OclAny} \mathcal{A} x)) \triangleq$
 $((X_{Person9} .oclAsType(OclAny)) @post (\lambda x. _OclAsType_{OclAny} \mathcal{A} x)))$

proof –

have including₄ : $\bigwedge a b c d \tau$.
Set $\{\lambda\tau. _a, \lambda\tau. _b, \lambda\tau. _c, \lambda\tau. _d\} \tau = Abs-Set_{base} _ \{_a, _b, _c, _d\} _$
apply(subst abs-rep-simp^[symmetric], simp)
apply(simp add: OclIncluding-rep-set mtSet-rep-set)
by(rule arg-cong[of - - $\lambda x. (Abs-Set_{base}(_ x))$], auto)

have excluding₁: $\bigwedge S a b c d e \tau$.
 $(\lambda _ . Abs-Set_{base} _ \{_a, _b, _c, _d\} _) -> excluding_{Set}(\lambda\tau. _e) \tau =$
 $Abs-Set_{base} _ \{_a, _b, _c, _d\} - \{_e\} _$
apply(simp add: UML-Set.OclExcluding-def)
apply(simp add: defined-def OclValid-def false-def true-def
bot-fun-def bot-Set_{base}-def null-fun-def null-Set_{base}-def)
apply(rule conjI)
apply(rule impI, subst (asm) Abs-Set_{base}-inject) **apply**(simp add: bot-option-def)+
apply(rule conjI)
apply(rule impI, subst (asm) Abs-Set_{base}-inject) **apply**(simp add: bot-option-def null-option-def)+
apply(subst Abs-Set_{base}-inverse, simp add: bot-option-def, simp)
done

show ?thesis

apply(rule framing[**where** X = Set{ X_{Person1} .oclAsType(OclAny)
, X_{Person2} .oclAsType(OclAny)
//XPerson3.oclAsType(OclAny)
, X_{Person4} .oclAsType(OclAny)

```

//XPerson5//OclAsType(OclAny)
, XPerson6 .oclAsType(OclAny)
//XPerson6//OclAsType(OclAny)
//XPerson7//OclAsType(OclAny)
//XPerson8//OclAsType(OclAny)
//XPerson9//OclAsType(OclAny)}
apply(cut-tac  $\sigma$ -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 UML-Set.OclExcluding.cp0,
  subst (asm) cp-OclIsModifiedOnly, simp add: including4 excluding1)

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: OclIncluding-rep-set mtSet-rep-set
  oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)
apply(simp add: StrictRefEqObject-def oid-of-option-def oid-of-typeOclAny-def OclNot-def OclValid-def
  null-option-def bot-option-def)

done
qed

lemma perm- $\sigma_1'$  :  $\sigma_1' = (\text{heap} = \text{Map.empty}$ 
  (oid8  $\mapsto$  inPerson person9,
  oid7  $\mapsto$  inOclAny person8,
  oid6  $\mapsto$  inOclAny person7,
  oid5  $\mapsto$  inPerson person6,
  oid4
  oid3  $\mapsto$  inPerson person4,
  oid2  $\mapsto$  inPerson person3,
  oid1  $\mapsto$  inPerson person2,
  oid0  $\mapsto$  inPerson person1)
  , assocs = assocs  $\sigma_1'$ )

proof –
note P = fun-upd-twist
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) P, simp)
apply(subst (2) P, simp) apply(subst (1) P, simp)
apply(subst (3) P, simp) apply(subst (2) P, simp) apply(subst (1) P, simp)
apply(subst (4) P, simp) apply(subst (3) P, simp) apply(subst (2) P, simp) apply(subst (1) P, simp)
apply(subst (5) P, simp) apply(subst (4) P, simp) apply(subst (3) P, simp) apply(subst (2) P, simp)
apply(subst (1) P, simp)
apply(subst (6) P, simp) apply(subst (5) P, simp) apply(subst (4) P, simp) apply(subst (3) P, simp)
apply(subst (2) P, simp) apply(subst (1) P, simp)
apply(subst (7) P, simp) apply(subst (6) P, simp) apply(subst (5) P, simp) apply(subst (4) P, simp)
apply(subst (3) P, simp) apply(subst (2) P, simp) apply(subst (1) P, simp)
by(simp)
qed

declare const-ss [simp]

```

lemma $\bigwedge \sigma_1$.

$(\sigma_1, \sigma_1') \models (Person .allInstances() \doteq Set\{ X_{Person1}, X_{Person2}, X_{Person3}, X_{Person4}, X_{Person5}, X_{Person6}, X_{Person7} .oclAsType(Person), X_{Person8}, X_{Person9} \})$

apply(subst perm- σ_1')

apply(simp only: oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def

$X_{Person1}$ -def $X_{Person2}$ -def $X_{Person3}$ -def $X_{Person4}$ -def

$X_{Person5}$ -def $X_{Person6}$ -def $X_{Person7}$ -def $X_{Person8}$ -def $X_{Person9}$ -def

person7-def)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- \mathcal{A} -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule OclIncluding-cong, simp, simp)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- \mathcal{A} -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule OclIncluding-cong, simp, simp)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- \mathcal{A} -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule OclIncluding-cong, simp, simp)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- \mathcal{A} -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule OclIncluding-cong, simp, simp)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- \mathcal{A} -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule OclIncluding-cong, simp, simp)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- \mathcal{A} -def, simp, rule const-StrictRefEqSet-including, simp, simp, rule OclIncluding-cong, simp, simp)

apply(subst state-update-vs-allInstances-at-post-ntc, simp, simp add: OclAsTypePerson- \mathcal{A} -def

person8-def, simp, rule const-StrictRefEqSet-including,

simp, simp, simp)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- \mathcal{A} -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule OclIncluding-cong, simp, simp)

apply(rule state-update-vs-allInstances-at-post-empty)

by(simp-all add: OclAsTypePerson- \mathcal{A} -def)

lemma $\bigwedge \sigma_1$.

$(\sigma_1, \sigma_1') \models (OclAny .allInstances() \doteq Set\{ X_{Person1} .oclAsType(OclAny), X_{Person2} .oclAsType(OclAny), X_{Person3} .oclAsType(OclAny), X_{Person4} .oclAsType(OclAny), X_{Person5}, X_{Person6} .oclAsType(OclAny), X_{Person7}, X_{Person8}, X_{Person9} .oclAsType(OclAny) \})$

apply(subst perm- σ_1')

apply(simp only: oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def

$X_{Person1}$ -def $X_{Person2}$ -def $X_{Person3}$ -def $X_{Person4}$ -def $X_{Person5}$ -def $X_{Person6}$ -def $X_{Person7}$ -def

$X_{Person8}$ -def $X_{Person9}$ -def

person1-def person2-def person3-def person4-def person5-def person6-def person9-def)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypeOclAny- \mathcal{A} -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule OclIncluding-cong, simp, simp)+

apply(rule state-update-vs-allInstances-at-post-empty)

by(simp-all add: OclAsTypeOclAny- \mathcal{A} -def)

end

theory

Analysis-OCL

imports

Analysis-UML

begin

4.10. OCL Part: Invariant

These recursive predicates can be defined conservatively by greatest fix-point constructions—automatically. See [4, 6] for details. For the purpose of this example, we state them as axioms here.

```

context Person
  inv label : self .boss <> null implies (self .salary \<le>
((self .boss) .salary))

```

definition $Person\text{-}label_{inv} :: Person \Rightarrow Boolean$
where $Person\text{-}label_{inv}(self) \equiv$
 $(self .boss <> null \text{ implies } (self .salary \leq_{int} ((self .boss) .salary)))$

definition $Person\text{-}label_{invATpre} :: Person \Rightarrow Boolean$
where $Person\text{-}label_{invATpre}(self) \equiv$
 $(self .boss@pre <> null \text{ implies } (self .salary@pre \leq_{int} ((self .boss@pre) .salary@pre)))$

definition $Person\text{-}label_{globalinv} :: Boolean$
where $Person\text{-}label_{globalinv} \equiv (Person .allInstances() \rightarrow \text{forAll}_{Set}(x \mid Person\text{-}label_{inv}(x)) \text{ and}$
 $(Person .allInstances@pre() \rightarrow \text{forAll}_{Set}(x \mid Person\text{-}label_{invATpre}(x))))$

lemma $\tau \models \delta(X .boss) \implies \tau \models Person .allInstances() \rightarrow \text{includes}_{Set}(X .boss) \wedge$
 $\tau \models Person .allInstances() \rightarrow \text{includes}_{Set}(X)$

oops

lemma $REC\text{-}pre : \tau \models Person\text{-}label_{globalinv}$
 $\implies \tau \models Person .allInstances() \rightarrow \text{includes}_{Set}(X) \text{ — } X \text{ represented object in state}$
 $\implies \exists REC. \tau \models REC(X) \triangleq (Person\text{-}label_{inv}(X) \text{ and } (X .boss <> null \text{ implies } REC(X .boss)))$

oops

This allows to state a predicate:

axiomatization $inv_{Person\text{-}label} :: Person \Rightarrow Boolean$

where $inv_{Person\text{-}label}\text{-}def:$

$(\tau \models Person .allInstances() \rightarrow \text{includes}_{Set}(self)) \implies$
 $(\tau \models (inv_{Person\text{-}label}(self) \triangleq (self .boss <> null \text{ implies}$
 $(self .salary \leq_{int} ((self .boss) .salary)) \text{ and}$
 $inv_{Person\text{-}label}(self .boss))))$

axiomatization $inv_{Person\text{-}labelATpre} :: Person \Rightarrow Boolean$

where $inv_{Person\text{-}labelATpre}\text{-}def:$

$(\tau \models Person .allInstances@pre() \rightarrow \text{includes}_{Set}(self)) \implies$
 $(\tau \models (inv_{Person\text{-}labelATpre}(self) \triangleq (self .boss@pre <> null \text{ implies}$
 $(self .salary@pre \leq_{int} ((self .boss@pre) .salary@pre)) \text{ and}$
 $inv_{Person\text{-}labelATpre}(self .boss@pre))))$

lemma $inv\text{-}1 :$

$(\tau \models Person .allInstances() \rightarrow \text{includes}_{Set}(self)) \implies$
 $(\tau \models inv_{Person\text{-}label}(self) = ((\tau \models (self .boss \doteq null)) \vee$
 $(\tau \models (self .boss <> null) \wedge$
 $\tau \models ((self .salary) \leq_{int} (self .boss .salary)) \wedge$
 $\tau \models (inv_{Person\text{-}label}(self .boss))))))$

oops

lemma $inv\text{-}2 :$

$(\tau \models Person .allInstances@pre() \rightarrow \text{includes}_{Set}(self)) \implies$
 $(\tau \models inv_{Person\text{-}labelATpre}(self) = ((\tau \models (self .boss@pre \doteq null)) \vee$

$$\begin{aligned}
& (\tau \models (\text{self}.\text{boss}@pre \langle \rangle \text{null}) \wedge \\
& (\tau \models (\text{self}.\text{boss}@pre.\text{salary}@pre \leq_{int} \text{self}.\text{salary}@pre)) \wedge \\
& (\tau \models (\text{inv}_{\text{Person-labelATpre}}(\text{self}.\text{boss}@pre))))
\end{aligned}$$

oops

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 => (A)st => bool where
  (τ ⊨ (δ self)) => ((τ ⊨ (self.boss ≐ null)) ∨
    (τ ⊨ (self.boss <> null) ∧ (τ ⊨ (self.boss.salary ≤int self.salary)) ∧
    ((inv(self.boss))τ)))
  => (inv self τ)

```

4.11. OCL Part: The Contract of a Recursive Query

The original specification of a recursive query :

```

context Person :: contents() : Set(Integer)
pre: true
post: result = if self.boss = null
              then Set{i}
              else self.boss.contents()->including(i)
endif

```

For the case of recursive queries, we use at present just axiomatizations:

axiomatization contents :: Person => Set-Integer (⟨(1(-).contents'())⟩ 50)

where contents-def:

```

(self.contents()) = (λ τ. SOME res. let res = λ -. res in
  if τ ⊨ (δ self)
  then ((τ ⊨ true) ∧
    (τ ⊨ res ≐ if (self.boss ≐ null)
      then (Set{self.salary})
      else (self.boss.contents()
        ->includingSet(self.salary))
    endif))
  else τ ⊨ res ≐ invalid)

```

and cp0-contents:(X.contents()) τ = ((λ-. X τ).contents()) τ

interpretation contents : contract0 contents λ self. true

```

λ self res. res ≐ if (self.boss ≐ null)
  then (Set{self.salary})
  else (self.boss.contents()
    ->includingSet(self.salary))
endif

```

proof (unfold-locales)

show $\bigwedge \text{self } \tau. \text{true } \tau = \text{true } \tau$ **by** auto

next

show $\bigwedge \text{self}. \forall \sigma \sigma' \sigma''. ((\sigma, \sigma') \models \text{true}) = ((\sigma, \sigma'') \models \text{true})$ **by** auto

next

```

show  $\bigwedge \text{self}. \text{self}.\text{contents}() \equiv$ 
  λ τ. SOME res. let res = λ -. res in
  if τ ⊨ (δ self)
  then ((τ ⊨ true) ∧
    (τ ⊨ res ≐ if (self.boss ≐ null)
      then (Set{self.salary})
      else (self.boss.contents()
        ->includingSet(self.salary))
    endif))

```

```

                                ->includingSet(self .salary))
                                endif))
                                else  $\tau \models res \triangleq invalid$ 
by(auto simp: contents-def )
next
have A: $\bigwedge self \tau. ((\lambda-. self \tau) .boss \doteq null) \tau = (\lambda-. (self .boss \doteq null) \tau) \tau$ 
by (metis (no-types) StrictRefEqObject-Person cp-StrictRefEqObject cp-dotPersonBOSS)
have B: $\bigwedge self \tau. (\lambda-. Set\{(\lambda-. self \tau) .salary\} \tau) = (\lambda-. Set\{self .salary\} \tau)$ 
    apply(subst UML-Set.OclIncluding.cp0)
    apply(subst (2) UML-Set.OclIncluding.cp0)
    apply(subst (2) Analysis-UML.cp-dotPersonSALARY) by simp
have C: $\bigwedge self \tau. ((\lambda-. self \tau) .boss .contents() ->includingSet((\lambda-. self \tau) .salary) \tau) =$ 
    (self .boss .contents() ->includingSet(self .salary) \tau)
    apply(subst UML-Set.OclIncluding.cp0) apply(subst (2) UML-Set.OclIncluding.cp0)
    apply(subst (2) Analysis-UML.cp-dotPersonSALARY)
    apply(subst cp0-contents) apply(subst (2) cp0-contents)
    apply(subst (2) cp-dotPersonBOSS) by simp
show  $\bigwedge self res \tau.$ 
    (res  $\triangleq$  if (self .boss)  $\doteq$  null then Set{self .salary}
    else self .boss .contents() ->includingSet(self .salary) endif)  $\tau =$ 
    (( $\lambda-. res \tau) \triangleq$  if ( $\lambda-. self \tau) .boss \doteq$  null then Set{( $\lambda-. self \tau) .salary$ }
    else( $\lambda-. self \tau) .boss .contents() ->includingSet(( $\lambda-. self \tau) .salary$ ) endif)  $\tau$ )
apply(subst cp-StrongEq)
apply(subst (2) cp-StrongEq)
apply(subst cp-OclIf)
apply(subst (2) cp-OclIf)
by(simp add: A B C)
qed$ 
```

Specializing $\llbracket cp E; \tau \models \delta self; \tau \models true; \tau \models POST' self; \bigwedge res. (res \triangleq \text{if } self.boss \doteq \text{null then } Set\{self.salary\} \text{ else } self.boss.contents() ->including_{Set}(self.salary) \text{ endif}) = (POST' self \text{ and } (res \triangleq BODY self)) \rrbracket \implies (\tau \models E (self.contents())) = (\tau \models E (BODY self))$, one gets the following more practical rewrite rule that is amenable to symbolic evaluation:

theorem *unfold-contents* :

```

assumes cp E
and  $\tau \models \delta self$ 
shows  $(\tau \models E (self .contents())) =$ 
    ( $\tau \models E$  (if self .boss  $\doteq$  null
    then Set{self .salary}
    else self .boss .contents() ->includingSet(self .salary) endif))
by(rule contents.unfold2[of - - -  $\lambda X. true$ ], simp-all add: assms)

```

Since we have only one interpretation function, we need the corresponding operation on the pre-state:

consts *contentsATpre* :: *Person* \Rightarrow *Set-Integer* ($\langle (1(-).contents@pre'()) \rangle$ 50)

axiomatization where *contentsATpre-def*:

```

(self).contents@pre() = ( $\lambda \tau.$ 
  SOME res. let res =  $\lambda -. res$  in
  if  $\tau \models (\delta self)$ 
  then (( $\tau \models true$ )  $\wedge$   $\text{--- pre}$ 
    ( $\tau \models (res \triangleq$  if (self).boss@pre  $\doteq$  null  $\text{--- post}$ 
    then Set{(self).salary@pre}
    else (self).boss@pre .contents@pre()
    ->includingSet(self .salary@pre)
    endif)))
  else  $\tau \models res \triangleq invalid$ )
and cp0-contents-at-pre:(X .contents@pre())  $\tau = ((\lambda-. X \tau) .contents@pre()) \tau$ 

```

```

interpretation contentsATpre : contract0 contentsATpre λ self. true
  λ self res. res ≐ if (self .boss@pre ≐ null)
    then (Set{self .salary@pre})
    else (self .boss@pre .contents@pre()
      ->includingSet(self .salary@pre))
    endif

proof (unfold-locales)
  show ∧self τ. true τ = true τ by auto
next
  show ∧self. ∀σ σ' σ''. ((σ, σ') ⊨ true) = ((σ, σ'') ⊨ true) by auto
next
  show ∧self. self .contents@pre() ≐
    λτ. SOME res. let res = λ -. res in
      if τ ⊨ δ self
      then τ ⊨ true ∧
        τ ⊨ res ≐ (if self .boss@pre ≐ null then Set{self .salary@pre}
          else self .boss@pre .contents@pre()->includingSet(self .salary@pre)
        endif)
      else τ ⊨ res ≐ invalid
    by(auto simp: contentsATpre-def)
next
  have A: ∧self τ. ((λ-. self τ) .boss@pre ≐ null) τ = (λ-. (self .boss@pre ≐ null) τ) τ
  by (metis StrictRefEqObject-Person cp-StrictRefEqObject cp-dotPersonBOSS-at-pre)
  have B: ∧self τ. (λ-. Set{(λ-. self τ) .salary@pre}) τ = (λ-. Set{self .salary@pre}) τ
    apply(subst UML-Set.OclIncluding.cp0)
    apply(subst (2) UML-Set.OclIncluding.cp0)
    apply(subst (2) Analysis-UML.cp-dotPersonSALARJ-at-pre) by simp
  have C: ∧self τ. ((λ-. self τ).boss@pre .contents@pre()->includingSet((λ-. self τ).salary@pre) τ) =
    (self .boss@pre .contents@pre() ->includingSet(self .salary@pre) τ)
    apply(subst UML-Set.OclIncluding.cp0) apply(subst (2) UML-Set.OclIncluding.cp0)
    apply(subst (2) Analysis-UML.cp-dotPersonSALARJ-at-pre)
    apply(subst cp0-contents-at-pre) apply(subst (2) cp0-contents-at-pre)
    apply(subst (2) cp-dotPersonBOSS-at-pre) by simp
  show ∧self res τ.
    (res ≐ if (self .boss@pre) ≐ null then Set{self .salary@pre}
      else self .boss@pre .contents@pre()->includingSet(self .salary@pre) endif) τ =
    ((λ-. res τ) ≐ if (λ-. self τ) .boss@pre ≐ null then Set{(λ-. self τ) .salary@pre}
      else(λ-. self τ) .boss@pre .contents@pre()->includingSet((λ-. self τ) .salary@pre)
    endif) τ
    apply(subst cp-StrongEq)
    apply(subst (2) cp-StrongEq)
    apply(subst cp-OclIf)
    apply(subst (2) cp-OclIf)
    by(simp add: A B C)
  qed

```

Again, we derive via *contents.unfold2* a Knaster-Tarski like Fixpoint rule that is amenable to symbolic evaluation:

```

theorem unfold-contentsATpre :
  assumes cp E
  and τ ⊨ δ self
  shows (τ ⊨ E (self .contents@pre())) =
    (τ ⊨ E (if self .boss@pre ≐ null
      then Set{self .salary@pre}
      else self .boss@pre .contents@pre()->includingSet(self .salary@pre) endif))
by(rule contentsATpre.unfold2[of - - λ X. true], simp-all add: assms)

```

Note that these @pre variants on methods are only available on queries, i. e., operations without side-effect.

4.12. OCL Part: The Contract of a User-defined Method

The example specification in high-level OCL input syntax reads as follows:

```
context Person::insert(x:Integer)
pre: true
post: contents():Set(Integer)
contents() = contents@pre()->including(x)
```

This boils down to:

definition *insert* :: *Person* \Rightarrow *Integer* \Rightarrow *Void* ($\langle(1(-).insert'(-))\rangle$ 50)
where *self* .*insert*(*x*) \equiv
 $(\lambda \tau. \text{SOME } res. \text{let } res = \lambda -. \text{res in}$
 $\text{if } (\tau \models (\delta \text{ self})) \wedge (\tau \models v \ x)$
 $\text{then } (\tau \models \text{true} \wedge$
 $\text{ } (\tau \models ((\text{self}).\text{contents}()) \triangleq (\text{self}).\text{contents}@pre()->\text{including}_{Set}(x)))$
 $\text{else } \tau \models res \triangleq \text{invalid})$

The semantic consequences of this definition were computed inside this locale interpretation:

interpretation *insert* : *contract1 insert* λ *self* *x*. *true*
 λ *self* *x* *res*. $((\text{self} .\text{contents}()) \triangleq$
 $(\text{self} .\text{contents}@pre()->\text{including}_{Set}(x)))$
apply *unfold-locales* **apply**(*auto simp:insert-def*)
apply(*subst cp-StrongEq*) **apply**(*subst (2) cp-StrongEq*)
apply(*subst contents.cp0*)
apply(*subst UML-Set.OclIncluding.cp0*)
apply(*subst (2) UML-Set.OclIncluding.cp0*)
apply(*subst contentsATpre.cp0*)
by(*simp*)

The result of this locale interpretation for our *Analysis-OCL.insert* contract is the following set of properties, which serves as basis for automated deduction on them:

end

Name	Theorem
<i>insert.strict0</i>	$(invalid.insert(X)) = invalid$
<i>insert.nullstrict0</i>	$(null.insert(X)) = invalid$
<i>insert.strict1</i>	$(self.insert(invalid)) = invalid$
<i>insert.cpPRE</i>	$true \tau = true \tau$
<i>insert.cpPOST</i>	$(self.contents() \triangleq self.contents@pre() \rightarrow including_{Set}(a1.0)) \tau = (\lambda-. self \tau.contents() \triangleq \lambda-. self \tau.contents@pre() \rightarrow including_{Set}(\lambda-. a1.0 \tau)) \tau$
<i>insert.cp-pre</i>	$\llbracket cp \ self'; \ cp \ a1' \rrbracket \implies cp \ (\lambda X. \ true)$
<i>insert.cp-post</i>	$\llbracket cp \ self'; \ cp \ a1'; \ cp \ res' \rrbracket \implies cp \ (\lambda X. \ self' \ X.contents() \triangleq self' \ X.contents@pre() \rightarrow including_{Set}(a1' \ X))$
<i>insert.cp</i>	$\llbracket cp \ self'; \ cp \ a1'; \ cp \ res' \rrbracket \implies cp \ (\lambda X. \ self' \ X.insert(a1' \ X))$
<i>insert.cp0</i>	$(self.insert(a1.0)) \tau = (\lambda-. self \ \tau.insert(\lambda-. a1.0 \ \tau)) \ \tau$
<i>insert.def-scheme</i>	$self.insert(a1.0) \equiv \lambda\tau. \ SOME \ res. \ let \ res = \lambda-. \ res \ in \ if \ \tau \models \delta \ self \ \wedge \ \tau \models v \ a1.0 \ then \ \tau \models true \ \wedge \ \tau \models self.contents() \triangleq self.contents@pre() \rightarrow including_{Set}(a1.0) \ else \ \tau \models res \triangleq invalid$
<i>insert.unfold</i>	$\llbracket cp \ E; \ \tau \models \delta \ self \ \wedge \ \tau \models v \ a1.0; \ \tau \models true; \ \exists \ res. \ \tau \models self.contents() \triangleq self.contents@pre() \rightarrow including_{Set}(a1.0); \ \bigwedge \ res. \ \tau \models self.contents() \triangleq self.contents@pre() \rightarrow including_{Set}(a1.0) \implies \tau \models E \ (\lambda-. \ res) \rrbracket \implies \tau \models E \ (self.insert(a1.0))$
<i>insert.unfold2</i>	$\llbracket cp \ E; \ \tau \models \delta \ self \ \wedge \ \tau \models v \ a1.0; \ \tau \models true; \ \tau \models POST' \ self \ a1.0; \ \bigwedge \ res. \ (self.contents() \triangleq self.contents@pre() \rightarrow including_{Set}(a1.0)) = (POST' \ self \ a1.0 \ and \ (res \triangleq BODY \ self \ a1.0)) \rrbracket \implies (\tau \models E \ (self.insert(a1.0))) = (\tau \models E \ (BODY \ self \ a1.0))$

Table 4.1.: Semantic properties resulting from a user-defined operation contract.

5. Example: The Employee Design Model

```
theory
  Design-UML
imports
  ../../../../UML-Main
begin
```

5.1. Introduction

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

Such generic function or “compiler” can be implemented in Isabelle on the ML level. This has been done, for a semantics following the open-world assumption, for UML 2.0 in [4, 7]. In this paper, we follow another approach for UML 2.4: we define the concepts of the compilation informally, and present a concrete example which is verified in Isabelle/HOL.

5.1.1. Outlining the Example

We are presenting here a “design-model” of the (slightly modified) example Figure 7.3, page 20 of the OCL standard [32]. To be precise, this theory contains the formalization of the data-part covered by the UML class model (see Figure 5.1):

This means that the association (attached to the association class **EmployeeRanking**) with the association ends **boss** and **employees** is implemented by the attribute **boss** and the operation **employees** (to be discussed in the OCL part captured by the subsequent theory).

5.2. Example Data-Universe and its Infrastructure

Ideally, the following is generated automatically from a UML class model.

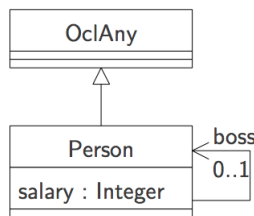


Figure 5.1.: A simple UML class model drawn from Figure 7.3, page 20 of [32].

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 OclAny will be used as instance for all respective 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
```

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 - - ⇒ oid)
  instance ..
end
```

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

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

5.3. Instantiation of the Generic Strict Equality

We instantiate the referential equality on *Person* and *OclAny*

```

overloading StrictRefEq ≡ StrictRefEq :: [Person,Person] ⇒ Boolean
begin
  definition StrictRefEqObject-Person : (x::Person) ≐ y ≡ StrictRefEqObject x y
end

```

```

overloading StrictRefEq ≡ StrictRefEq :: [OclAny,OclAny] ⇒ Boolean
begin
  definition StrictRefEqObject-OclAny : (x::OclAny) ≐ y ≡ StrictRefEqObject x y
end

```

```

lemmas cps23 =
  cp-StrictRefEqObject [of x::Person y::Person τ,
    simplified StrictRefEqObject-Person[symmetric]]
  cp-intro(9) [of P::Person ⇒ Person Q::Person ⇒ Person,
    simplified StrictRefEqObject-Person[symmetric]]
  StrictRefEqObject-def [of x::Person y::Person,
    simplified StrictRefEqObject-Person[symmetric]]
  StrictRefEqObject-defargs [of - x::Person y::Person,
    simplified StrictRefEqObject-Person[symmetric]]
  StrictRefEqObject-strict1
    [of x::Person,
    simplified StrictRefEqObject-Person[symmetric]]
  StrictRefEqObject-strict2
    [of x::Person,
    simplified StrictRefEqObject-Person[symmetric]]
for x y τ P Q

```

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

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

5.4. OclAsType

5.4.1. Definition

```

consts OclAsTypeOclAny :: 'α ⇒ OclAny (⟨(-) .oclAsType'(OclAny)⟩)
consts OclAsTypePerson :: 'α ⇒ Person (⟨(-) .oclAsType'(Person)⟩)

```

```

definition OclAsTypeOclAny-ℳ = (λu. ⊔ case u of inOclAny a ⇒ a
  | inPerson (mkPerson oid a b) ⇒ mkOclAny oid ⊔(a,b)⊔)

```

```

lemma OclAsTypeOclAny-ℳ-some: OclAsTypeOclAny-ℳ x ≠ None
by(simp add: OclAsTypeOclAny-ℳ-def)

```

```

overloading OclAsTypeOclAny ≡ OclAsTypeOclAny :: OclAny ⇒ OclAny
begin
  definition OclAsTypeOclAny-OclAny:
    (X::OclAny) .oclAsType(OclAny) ≡ X
end

```

```

overloading OclAsTypeOclAny ≡ OclAsTypeOclAny :: Person ⇒ OclAny
begin
  definition OclAsTypeOclAny-Person:
    (X::Person) .oclAsType(OclAny) ≡

```

```

      (λτ. case X τ of
        ⊥ ⇒ invalid τ
        | ⊥⊥ ⇒ null τ
        | ⊥⊥ mkPerson oid a b⊥ ⇒ ⊥⊥ (mkOclAny oid ⊥(a,b)⊥)⊥)
end

definition OclAsTypePerson- $\mathfrak{A}$  =
  (λu. case u of inPerson p ⇒ ⊥p
    | inOclAny (mkOclAny oid ⊥(a,b)⊥) ⇒ ⊥mkPerson oid a b⊥
    | - ⇒ None)

overloading OclAsTypePerson ≡ OclAsTypePerson :: OclAny ⇒ Person
begin
  definition OclAsTypePerson-OclAny:
    (X::OclAny) .oclAsType(Person) ≡
      (λτ. case X τ of
        ⊥ ⇒ invalid τ
        | ⊥⊥ ⇒ null τ
        | ⊥⊥ mkOclAny oid ⊥⊥ ⇒ invalid τ — down-cast exception
        | ⊥⊥ mkOclAny oid ⊥(a,b)⊥ ⇒ ⊥⊥ mkPerson oid a b⊥)
end

overloading OclAsTypePerson ≡ OclAsTypePerson :: Person ⇒ Person
begin
  definition OclAsTypePerson-Person:
    (X::Person) .oclAsType(Person) ≡ X
endlemmas [simp] =
  OclAsTypeOclAny-OclAny
  OclAsTypePerson-Person

```

5.4.2. Context Passing

```

lemma cp-OclAsTypeOclAny-Person-Person: cp P ⇒ cp(λX. (P (X::Person)::Person) .oclAsType(OclAny))
by(rule cpI1, simp-all add: OclAsTypeOclAny-Person)
lemma cp-OclAsTypeOclAny-OclAny-OclAny: cp P ⇒ cp(λX. (P (X::OclAny)::OclAny) .oclAsType(OclAny))
by(rule cpI1, simp-all add: OclAsTypeOclAny-OclAny)
lemma cp-OclAsTypePerson-Person-Person: cp P ⇒ cp(λX. (P (X::Person)::Person) .oclAsType(Person))
by(rule cpI1, simp-all add: OclAsTypePerson-Person)
lemma cp-OclAsTypePerson-OclAny-OclAny: cp P ⇒ cp(λX. (P (X::OclAny)::OclAny) .oclAsType(Person))
by(rule cpI1, simp-all add: OclAsTypePerson-OclAny)

lemma cp-OclAsTypeOclAny-Person-OclAny: cp P ⇒ cp(λX. (P (X::Person)::OclAny) .oclAsType(OclAny))
by(rule cpI1, simp-all add: OclAsTypeOclAny-OclAny)
lemma cp-OclAsTypeOclAny-OclAny-Person: cp P ⇒ cp(λX. (P (X::OclAny)::Person) .oclAsType(OclAny))
by(rule cpI1, simp-all add: OclAsTypeOclAny-Person)
lemma cp-OclAsTypePerson-Person-OclAny: cp P ⇒ cp(λX. (P (X::Person)::OclAny) .oclAsType(Person))
by(rule cpI1, simp-all add: OclAsTypePerson-OclAny)
lemma cp-OclAsTypePerson-OclAny-Person: cp P ⇒ cp(λX. (P (X::OclAny)::Person) .oclAsType(Person))
by(rule cpI1, simp-all add: OclAsTypePerson-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

5.4.3. 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*)

5.5. OclIsTypeOf

5.5.1. Definition

consts *OclIsTypeOfOclAny* :: ' α \Rightarrow Boolean ($\langle(-).oclIsTypeOf'(OclAny)\rangle$)
consts *OclIsTypeOfPerson* :: ' α \Rightarrow Boolean ($\langle(-).oclIsTypeOf'(Person)\rangle$)

overloading *OclIsTypeOfOclAny* \equiv *OclIsTypeOfOclAny* :: *OclAny* \Rightarrow Boolean

begin

definition *OclIsTypeOfOclAny-OclAny*:

(*X*::*OclAny*) .*oclIsTypeOf*(*OclAny*) \equiv
($\lambda\tau$. *case X* τ *of*
 $\perp \Rightarrow$ *invalid* τ
 $\lfloor \perp \rfloor \Rightarrow$ *true* τ — *invalid* ??
 $\lfloor \perp mk_{OclAny} oid \perp \rfloor \Rightarrow$ *true* τ
 $\lfloor \perp mk_{OclAny} oid \lfloor \perp \rfloor \rfloor \Rightarrow$ *false* τ)

end

lemma *OclIsTypeOfOclAny-OclAny'*:

(*X*::*OclAny*) .*oclIsTypeOf*(*OclAny*) =
($\lambda\tau$. *if* $\tau \models v X$ *then* (*case X* τ *of*
 $\lfloor \perp \rfloor \Rightarrow$ *true* τ — *invalid* ??
 $\lfloor \perp mk_{OclAny} oid \perp \rfloor \Rightarrow$ *true* τ
 $\lfloor \perp mk_{OclAny} oid \lfloor \perp \rfloor \rfloor \Rightarrow$ *false* τ)
else *invalid* τ)

apply(*rule ext*, *simp add*: *OclIsTypeOfOclAny-OclAny*)

by(*case-tac* $\tau \models v X$, *auto simp*: *foundation18' bot-option-def*)

interpretation *OclIsTypeOfOclAny-OclAny* :

profile-mono-schemeV

OclIsTypeOfOclAny::*OclAny* \Rightarrow Boolean

λX . (*case X* *of*

$\lfloor None \rfloor \Rightarrow$ $\lfloor True \rfloor$ — *invalid* ??

$\lfloor \perp mk_{OclAny} oid None \rfloor \Rightarrow$ $\lfloor True \rfloor$

$\lfloor \perp mk_{OclAny} oid \lfloor \perp \rfloor \rfloor \Rightarrow$ $\lfloor False \rfloor$)

apply(*unfold-locales*, *simp add*: *atomize-eq*, *rule ext*)

by(*auto simp*: *OclIsTypeOfOclAny-OclAny'* *OclValid-def true-def false-def*
split: *option.split typeOclAny.split*)

overloading $OclIsTypeOf_{OclAny} \equiv OclIsTypeOf_{OclAny} :: Person \Rightarrow Boolean$
begin
definition $OclIsTypeOf_{OclAny}-Person$:
 $(X::Person) .oclIsTypeOf(OclAny) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \ \text{of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \ \perp \Rightarrow \text{true } \tau \quad \text{--- invalid ??}$
 $\quad | \ _ \Rightarrow \text{false } \tau) \quad \text{--- must have actual type } Person \ \text{otherwise}$
end

overloading $OclIsTypeOf_{Person} \equiv OclIsTypeOf_{Person} :: OclAny \Rightarrow Boolean$
begin
definition $OclIsTypeOf_{Person}-OclAny$:
 $(X::OclAny) .oclIsTypeOf(Person) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \ \text{of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \ \perp \Rightarrow \text{true } \tau$
 $\quad | \ \underline{\perp} \text{mk}_{OclAny} \ \text{oid} \ \perp \Rightarrow \text{false } \tau$
 $\quad | \ \underline{\perp} \text{mk}_{OclAny} \ \text{oid} \ _ \Rightarrow \text{true } \tau)$
end

overloading $OclIsTypeOf_{Person} \equiv OclIsTypeOf_{Person} :: Person \Rightarrow Boolean$
begin
definition $OclIsTypeOf_{Person}-Person$:
 $(X::Person) .oclIsTypeOf(Person) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \ \text{of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \ _ \Rightarrow \text{true } \tau)$
end

5.5.2. Context Passing

lemma $cp-OclIsTypeOf_{OclAny}-Person-Person$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::Person)::Person).oclIsTypeOf(OclAny))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny}-Person$)
lemma $cp-OclIsTypeOf_{OclAny}-OclAny-OclAny$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::OclAny).oclIsTypeOf(OclAny))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny}-OclAny$)
lemma $cp-OclIsTypeOf_{Person}-Person-Person$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::Person)::Person).oclIsTypeOf(Person))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{Person}-Person$)
lemma $cp-OclIsTypeOf_{Person}-OclAny-OclAny$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::OclAny).oclIsTypeOf(Person))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{Person}-OclAny$)

lemma $cp-OclIsTypeOf_{OclAny}-Person-OclAny$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::Person)::OclAny).oclIsTypeOf(OclAny))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny}-OclAny$)
lemma $cp-OclIsTypeOf_{OclAny}-OclAny-Person$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::Person).oclIsTypeOf(OclAny))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{OclAny}-Person$)
lemma $cp-OclIsTypeOf_{Person}-Person-OclAny$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::Person)::OclAny).oclIsTypeOf(Person))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{Person}-OclAny$)
lemma $cp-OclIsTypeOf_{Person}-OclAny-Person$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::Person).oclIsTypeOf(Person))$
by(rule $cpI1$, simp-all add: $OclIsTypeOf_{Person}-Person$)

lemmas [simp] =
 $cp-OclIsTypeOf_{OclAny}-Person-Person$
 $cp-OclIsTypeOf_{OclAny}-OclAny-OclAny$
 $cp-OclIsTypeOf_{Person}-Person-Person$
 $cp-OclIsTypeOf_{Person}-OclAny-OclAny$

cp-OclIsTypeOf_{OclAny}-Person-OclAny
cp-OclIsTypeOf_{OclAny}-OclAny-Person
cp-OclIsTypeOf_{Person}-Person-OclAny
cp-OclIsTypeOf_{Person}-OclAny-Person

5.5.3. Execution with Invalid or Null as Argument

lemma *OclIsTypeOf_{OclAny}-OclAny-strict1* [simp]:
 (invalid::*OclAny*) .*oclIsTypeOf*(*OclAny*) = *invalid*
by(rule *ext*, simp add: *null-fun-def null-option-def bot-option-def invalid-def*
OclIsTypeOf_{OclAny}-OclAny)
lemma *OclIsTypeOf_{OclAny}-OclAny-strict2* [simp]:
 (null::*OclAny*) .*oclIsTypeOf*(*OclAny*) = *true*
by(rule *ext*, simp add: *null-fun-def null-option-def bot-option-def invalid-def*
OclIsTypeOf_{OclAny}-OclAny)
lemma *OclIsTypeOf_{OclAny}-Person-strict1* [simp]:
 (invalid::*Person*) .*oclIsTypeOf*(*OclAny*) = *invalid*
by(rule *ext*, simp add: *null-fun-def null-option-def bot-option-def invalid-def*
OclIsTypeOf_{OclAny}-Person)
lemma *OclIsTypeOf_{OclAny}-Person-strict2* [simp]:
 (null::*Person*) .*oclIsTypeOf*(*OclAny*) = *true*
by(rule *ext*, simp add: *null-fun-def null-option-def bot-option-def invalid-def*
OclIsTypeOf_{OclAny}-Person)
lemma *OclIsTypeOf_{Person}-OclAny-strict1* [simp]:
 (invalid::*OclAny*) .*oclIsTypeOf*(*Person*) = *invalid*
by(rule *ext*, simp add: *null-fun-def null-option-def bot-option-def invalid-def*
OclIsTypeOf_{Person}-OclAny)
lemma *OclIsTypeOf_{Person}-OclAny-strict2* [simp]:
 (null::*OclAny*) .*oclIsTypeOf*(*Person*) = *true*
by(rule *ext*, simp add: *null-fun-def null-option-def bot-option-def invalid-def*
OclIsTypeOf_{Person}-OclAny)
lemma *OclIsTypeOf_{Person}-Person-strict1* [simp]:
 (invalid::*Person*) .*oclIsTypeOf*(*Person*) = *invalid*
by(rule *ext*, simp add: *null-fun-def null-option-def bot-option-def invalid-def*
OclIsTypeOf_{Person}-Person)
lemma *OclIsTypeOf_{Person}-Person-strict2* [simp]:
 (null::*Person*) .*oclIsTypeOf*(*Person*) = *true*
by(rule *ext*, simp add: *null-fun-def null-option-def bot-option-def invalid-def*
OclIsTypeOf_{Person}-Person)

5.5.4. Up Down Casting

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

lemma *down-cast-type*:
assumes *isOclAny*: $\tau \models (X::\text{OclAny}) .\text{oclIsTypeOf}(\text{OclAny})$
and *non-null*: $\tau \models (\delta X)$
shows $\tau \models (X .\text{oclAsType}(\text{Person})) \triangleq \text{invalid}$
using *isOclAny non-null*
apply(auto simp : *bot-fun-def null-fun-def null-option-def bot-option-def invalid-def*
OclAsType_{OclAny}-Person OclAsType_{Person}-OclAny foundation22 foundation16)

split: option.split typeOclAny.split typePerson.split
by(*simp add: OclIsTypeOfOclAny-OclAny OclValid-def false-def true-def*)

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

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

lemma *up-down-cast-Person-OclAny-Person* [*simp*]:
shows $((X :: \text{Person}) . \text{oclAsType}(\text{OclAny}) . \text{oclAsType}(\text{Person}) = X)$
apply(*rule ext, rename-tac τ*)
apply(*rule foundation22[THEN iffD1]*)
apply(*case-tac $\tau \models (\delta X)$, simp add: up-down-cast*)
apply(*simp add: defined-split, 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*

5.6. OclIsKindOf

5.6.1. Definition

consts *OclIsKindOfOclAny* :: $'\alpha \Rightarrow \text{Boolean } (\langle(-) . \text{oclIsKindOf}'(\text{OclAny}')\rangle)$
consts *OclIsKindOfPerson* :: $'\alpha \Rightarrow \text{Boolean } (\langle(-) . \text{oclIsKindOf}'(\text{Person}')\rangle)$

overloading *OclIsKindOfOclAny* \equiv *OclIsKindOfOclAny* :: *OclAny* \Rightarrow *Boolean*
begin

definition *OclIsKindOfOclAny-OclAny*:
 $(X :: \text{OclAny}) . \text{oclIsKindOf}(\text{OclAny}) \equiv$
 $(\lambda\tau . \text{case } X \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | - \Rightarrow \text{true } \tau)$

end

overloading $OclIsKindOf_{OclAny} \equiv OclIsKindOf_{OclAny} :: Person \Rightarrow Boolean$

begin

definition $OclIsKindOf_{OclAny}-Person:$

$$(X::Person) .oclIsKindOf(OclAny) \equiv$$

$$(\lambda\tau. \text{case } X \ \tau \ \text{of}$$

$$\quad \perp \Rightarrow \text{invalid } \tau$$

$$\quad | _ \Rightarrow \text{true } \tau)$$

end

overloading $OclIsKindOf_{Person} \equiv OclIsKindOf_{Person} :: OclAny \Rightarrow Boolean$

begin

definition $OclIsKindOf_{Person}-OclAny:$

$$(X::OclAny) .oclIsKindOf(Person) \equiv$$

$$(\lambda\tau. \text{case } X \ \tau \ \text{of}$$

$$\quad \perp \Rightarrow \text{invalid } \tau$$

$$\quad | \perp \Rightarrow \text{true } \tau$$

$$\quad | \sqsubseteq mk_{OclAny} \ \text{oid } \perp \sqcup \Rightarrow \text{false } \tau$$

$$\quad | \sqsubseteq mk_{OclAny} \ \text{oid } \sqcup \sqcup \Rightarrow \text{true } \tau)$$

end

overloading $OclIsKindOf_{Person} \equiv OclIsKindOf_{Person} :: Person \Rightarrow Boolean$

begin

definition $OclIsKindOf_{Person}-Person:$

$$(X::Person) .oclIsKindOf(Person) \equiv$$

$$(\lambda\tau. \text{case } X \ \tau \ \text{of}$$

$$\quad \perp \Rightarrow \text{invalid } \tau$$

$$\quad | _ \Rightarrow \text{true } \tau)$$

end

5.6.2. Context Passing

lemma $cp-OclIsKindOf_{OclAny}-Person-Person: cp \ P \Longrightarrow cp(\lambda X.(P(X::Person)::Person).oclIsKindOf(OclAny))$

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

lemma $cp-OclIsKindOf_{OclAny}-OclAny-OclAny: cp \ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::OclAny).oclIsKindOf(OclAny))$

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

lemma $cp-OclIsKindOf_{Person}-Person-Person: cp \ P \Longrightarrow cp(\lambda X.(P(X::Person)::Person).oclIsKindOf(Person))$

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

lemma $cp-OclIsKindOf_{Person}-OclAny-OclAny: cp \ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::OclAny).oclIsKindOf(Person))$

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

lemma $cp-OclIsKindOf_{OclAny}-Person-OclAny: cp \ P \Longrightarrow cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(OclAny))$

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

lemma $cp-OclIsKindOf_{OclAny}-OclAny-Person: cp \ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(OclAny))$

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

lemma $cp-OclIsKindOf_{Person}-Person-OclAny: cp \ P \Longrightarrow cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(Person))$

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

lemma $cp-OclIsKindOf_{Person}-OclAny-Person: cp \ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(Person))$

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

lemmas [simp] =

$cp-OclIsKindOf_{OclAny}-Person-Person$

$cp-OclIsKindOf_{OclAny}-OclAny-OclAny$

$cp-OclIsKindOf_{Person}-Person-Person$

$cp-OclIsKindOf_{Person}-OclAny-OclAny$

$cp\text{-}OclIsKindOf_{OclAny}\text{-}Person\text{-}OclAny$
 $cp\text{-}OclIsKindOf_{OclAny}\text{-}OclAny\text{-}Person$
 $cp\text{-}OclIsKindOf_{Person}\text{-}Person\text{-}OclAny$
 $cp\text{-}OclIsKindOf_{Person}\text{-}OclAny\text{-}Person$

5.6.3. Execution with Invalid or Null as Argument

lemma $OclIsKindOf_{OclAny}\text{-}OclAny\text{-}strict1$ [simp] : $(invalid::OclAny) .oclIsKindOf(OclAny) = invalid$
by(rule ext, simp add: invalid-def bot-option-def
 $OclIsKindOf_{OclAny}\text{-}OclAny$)
lemma $OclIsKindOf_{OclAny}\text{-}OclAny\text{-}strict2$ [simp] : $(null::OclAny) .oclIsKindOf(OclAny) = true$
by(rule ext, simp add: null-fun-def null-option-def
 $OclIsKindOf_{OclAny}\text{-}OclAny$)
lemma $OclIsKindOf_{OclAny}\text{-}Person\text{-}strict1$ [simp] : $(invalid::Person) .oclIsKindOf(OclAny) = invalid$
by(rule ext, simp add: bot-option-def invalid-def
 $OclIsKindOf_{OclAny}\text{-}Person$)
lemma $OclIsKindOf_{OclAny}\text{-}Person\text{-}strict2$ [simp] : $(null::Person) .oclIsKindOf(OclAny) = true$
by(rule ext, simp add: null-fun-def null-option-def bot-option-def
 $OclIsKindOf_{OclAny}\text{-}Person$)
lemma $OclIsKindOf_{Person}\text{-}OclAny\text{-}strict1$ [simp] : $(invalid::OclAny) .oclIsKindOf(Person) = invalid$
by(rule ext, simp add: null-fun-def null-option-def bot-option-def invalid-def
 $OclIsKindOf_{Person}\text{-}OclAny$)
lemma $OclIsKindOf_{Person}\text{-}OclAny\text{-}strict2$ [simp] : $(null::OclAny) .oclIsKindOf(Person) = true$
by(rule ext, simp add: null-fun-def null-option-def bot-option-def invalid-def
 $OclIsKindOf_{Person}\text{-}OclAny$)
lemma $OclIsKindOf_{Person}\text{-}Person\text{-}strict1$ [simp] : $(invalid::Person) .oclIsKindOf(Person) = invalid$
by(rule ext, simp add: null-fun-def null-option-def bot-option-def invalid-def
 $OclIsKindOf_{Person}\text{-}Person$)
lemma $OclIsKindOf_{Person}\text{-}Person\text{-}strict2$ [simp] : $(null::Person) .oclIsKindOf(Person) = true$
by(rule ext, simp add: null-fun-def null-option-def bot-option-def invalid-def
 $OclIsKindOf_{Person}\text{-}Person$)

5.6.4. Up Down Casting

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
 $OclIsKindOf_{OclAny}\text{-}Person$ 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 invalid-def
 $OclAsType_{OclAny}\text{-}Person$ $OclAsType_{Person}\text{-}OclAny$ foundation22 foundation16
split: *option.split type_{OclAny}.split type_{Person}.split*)
by(simp add: $OclIsKindOf_{Person}\text{-}OclAny$ $OclValid\text{-}def$ *false-def true-def*)

5.7. OclAllInstances

To denote OCL-types occurring in OCL expressions syntactically—as, for example, as “argument” of $oclAllInstances()$ —we use the inverses of the injection functions into the object universes; we show that this is sufficient “characterization.”

definition $Person \equiv OclAsType_{Person}\text{-}\mathfrak{A}$
definition $OclAny \equiv OclAsType_{OclAny}\text{-}\mathfrak{A}$
lemmas $[simp] = Person\text{-}def\ OclAny\text{-}def$

lemma $OclAllInstances\text{-}generic_{OclAny}\text{-}exec: OclAllInstances\text{-}generic\ pre\text{-}post\ OclAny =$
 $(\lambda\tau. Abs\text{-}Set_{base} \perp\!\!\!\perp Some\ 'OclAny\ 'ran\ (heap\ (pre\text{-}post\ \tau)) \perp\!\!\!\perp)$

proof –
let $?S1 = \lambda\tau. OclAny\ 'ran\ (heap\ (pre\text{-}post\ \tau))$
let $?S2 = \lambda\tau. ?S1\ \tau - \{None\}$
have $B : \bigwedge\tau. ?S2\ \tau \subseteq ?S1\ \tau$ **by** *auto*
have $C : \bigwedge\tau. ?S1\ \tau \subseteq ?S2\ \tau$ **by** $(auto\ simp: OclAsType_{OclAny}\text{-}\mathfrak{A}\text{-}some)$

show $?thesis$ **by** $(insert\ equalityI[OF\ B\ C],\ simp)$
qed

lemma $OclAllInstances\text{-}at\text{-}post_{OclAny}\text{-}exec: OclAny.\ allInstances() =$
 $(\lambda\tau. Abs\text{-}Set_{base} \perp\!\!\!\perp Some\ 'OclAny\ 'ran\ (heap\ (snd\ \tau)) \perp\!\!\!\perp)$
unfolding $OclAllInstances\text{-}at\text{-}post\text{-}def$
by $(rule\ OclAllInstances\text{-}generic_{OclAny}\text{-}exec)$

lemma $OclAllInstances\text{-}at\text{-}pre_{OclAny}\text{-}exec: OclAny.\ allInstances@pre() =$
 $(\lambda\tau. Abs\text{-}Set_{base} \perp\!\!\!\perp Some\ 'OclAny\ 'ran\ (heap\ (fst\ \tau)) \perp\!\!\!\perp)$
unfolding $OclAllInstances\text{-}at\text{-}pre\text{-}def$
by $(rule\ OclAllInstances\text{-}generic_{OclAny}\text{-}exec)$

5.7.1. OclIsTypeOf

lemma $OclAny\text{-}allInstances\text{-}generic\ oclIsTypeOf_{OclAny}1:$
assumes $[simp]: \bigwedge x. pre\text{-}post\ (x, x) = x$
shows $\exists\tau. (\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ OclAny) \text{-}> forAll_{Set}(X|X.\ oclIsTypeOf(OclAny))))$
apply $(rule\text{-}tac\ x = \tau_0\ \mathbf{in}\ exI,\ simp\ add: \tau_0\text{-}def\ OclValid\text{-}def\ del: OclAllInstances\text{-}generic\text{-}def)$
apply $(simp\ only: assms\ UML\text{-}Set.\ OclForall\text{-}def\ refl\ if\text{-}True$
 $OclAllInstances\text{-}generic\text{-}defined[simplified\ OclValid\text{-}def])$
apply $(simp\ only: OclAllInstances\text{-}generic\text{-}def)$
apply $(subst\ (1\ 2\ 3)\ Abs\text{-}Set_{base}\text{-}inverse,\ simp\ add: bot\text{-}option\text{-}def)$
by $(simp\ add: OclIsTypeOf_{OclAny}\text{-}OclAny)$

lemma $OclAny\text{-}allInstances\text{-}at\text{-}post\ oclIsTypeOf_{OclAny}1:$
 $\exists\tau. (\tau \models (OclAny.\ allInstances() \text{-}> forAll_{Set}(X|X.\ oclIsTypeOf(OclAny))))$
unfolding $OclAllInstances\text{-}at\text{-}post\text{-}def$
by $(rule\ OclAny\text{-}allInstances\text{-}generic\ oclIsTypeOf_{OclAny}1,\ simp)$

lemma $OclAny\text{-}allInstances\text{-}at\text{-}pre\ oclIsTypeOf_{OclAny}1:$
 $\exists\tau. (\tau \models (OclAny.\ allInstances@pre() \text{-}> forAll_{Set}(X|X.\ oclIsTypeOf(OclAny))))$
unfolding $OclAllInstances\text{-}at\text{-}pre\text{-}def$
by $(rule\ OclAny\text{-}allInstances\text{-}generic\ oclIsTypeOf_{OclAny}1,\ simp)$

lemma $OclAny\text{-}allInstances\text{-}generic\ oclIsTypeOf_{OclAny}2:$
assumes $[simp]: \bigwedge x. pre\text{-}post\ (x, x) = x$
shows $\exists\tau. (\tau \models not\ ((OclAllInstances\text{-}generic\ pre\text{-}post\ OclAny) \text{-}> forAll_{Set}(X|X.\ oclIsTypeOf(OclAny))))$
proof – **fix** oid **let** $?t0 = (\text{heap} = \text{Map.empty}(oid \mapsto in_{OclAny}\ (mk_{OclAny}\ oid\ \perp_a)),$
 $assoc = \text{Map.empty})$ **show** $?thesis$
apply $(rule\text{-}tac\ x = (?t0,\ ?t0)\ \mathbf{in}\ exI,\ simp\ add: OclValid\text{-}def\ del: OclAllInstances\text{-}generic\text{-}def)$
apply $(simp\ only: UML\text{-}Set.\ OclForall\text{-}def\ refl\ if\text{-}True$
 $OclAllInstances\text{-}generic\text{-}defined[simplified\ OclValid\text{-}def])$
apply $(simp\ only: OclAllInstances\text{-}generic\text{-}def\ OclAsType_{OclAny}\text{-}\mathfrak{A}\text{-}def)$
apply $(subst\ (1\ 2\ 3)\ Abs\text{-}Set_{base}\text{-}inverse,\ simp\ add: bot\text{-}option\text{-}def)$

by(simp add: OclIsTypeOf_{OclAny}-OclAny OclNot-def OclAny-def)
qed

lemma OclAny-allInstances-at-post-oclIsTypeOf_{OclAny}2:
 $\exists \tau. (\tau \models \text{not } (\text{OclAny} . \text{allInstances}()) \rightarrow \text{forAll}_{\text{Set}}(X|X . \text{oclIsTypeOf}(\text{OclAny})))$
unfolding OclAllInstances-at-post-def
by(rule OclAny-allInstances-generic-oclIsTypeOf_{OclAny}2, simp)

lemma OclAny-allInstances-at-pre-oclIsTypeOf_{OclAny}2:
 $\exists \tau. (\tau \models \text{not } (\text{OclAny} . \text{allInstances}@pre() \rightarrow \text{forAll}_{\text{Set}}(X|X . \text{oclIsTypeOf}(\text{OclAny}))))$
unfolding OclAllInstances-at-pre-def
by(rule OclAny-allInstances-generic-oclIsTypeOf_{OclAny}2, simp)

lemma Person-allInstances-generic-oclIsTypeOf_{Person}:
 $\tau \models ((\text{OclAllInstances-generic pre-post Person}) \rightarrow \text{forAll}_{\text{Set}}(X|X . \text{oclIsTypeOf}(\text{Person})))$
apply(simp add: OclValid-def del: OclAllInstances-generic-def)
apply(simp only: UML-Set.OclForall-def refl if-True
OclAllInstances-generic-defined[simplified OclValid-def])
apply(simp only: OclAllInstances-generic-def)
apply(subst (1 2 3) Abs-Set_{base}-inverse, simp add: bot-option-def)
by(simp add: OclIsTypeOf_{Person}-Person)

lemma Person-allInstances-at-post-oclIsTypeOf_{Person}:
 $\tau \models (\text{Person} . \text{allInstances}()) \rightarrow \text{forAll}_{\text{Set}}(X|X . \text{oclIsTypeOf}(\text{Person})))$
unfolding OclAllInstances-at-post-def
by(rule Person-allInstances-generic-oclIsTypeOf_{Person})

lemma Person-allInstances-at-pre-oclIsTypeOf_{Person}:
 $\tau \models (\text{Person} . \text{allInstances}@pre()) \rightarrow \text{forAll}_{\text{Set}}(X|X . \text{oclIsTypeOf}(\text{Person})))$
unfolding OclAllInstances-at-pre-def
by(rule Person-allInstances-generic-oclIsTypeOf_{Person})

5.7.2. OclIsKindOf

lemma OclAny-allInstances-generic-oclIsKindOf_{OclAny}:
 $\tau \models ((\text{OclAllInstances-generic pre-post OclAny}) \rightarrow \text{forAll}_{\text{Set}}(X|X . \text{oclIsKindOf}(\text{OclAny})))$
apply(simp add: OclValid-def del: OclAllInstances-generic-def)
apply(simp only: UML-Set.OclForall-def refl if-True
OclAllInstances-generic-defined[simplified OclValid-def])
apply(simp only: OclAllInstances-generic-def)
apply(subst (1 2 3) Abs-Set_{base}-inverse, simp add: bot-option-def)
by(simp add: OclIsKindOf_{OclAny}-OclAny)

lemma OclAny-allInstances-at-post-oclIsKindOf_{OclAny}:
 $\tau \models (\text{OclAny} . \text{allInstances}()) \rightarrow \text{forAll}_{\text{Set}}(X|X . \text{oclIsKindOf}(\text{OclAny})))$
unfolding OclAllInstances-at-post-def
by(rule OclAny-allInstances-generic-oclIsKindOf_{OclAny})

lemma OclAny-allInstances-at-pre-oclIsKindOf_{OclAny}:
 $\tau \models (\text{OclAny} . \text{allInstances}@pre()) \rightarrow \text{forAll}_{\text{Set}}(X|X . \text{oclIsKindOf}(\text{OclAny})))$
unfolding OclAllInstances-at-pre-def
by(rule OclAny-allInstances-generic-oclIsKindOf_{OclAny})

lemma Person-allInstances-generic-oclIsKindOf_{OclAny}:
 $\tau \models ((\text{OclAllInstances-generic pre-post Person}) \rightarrow \text{forAll}_{\text{Set}}(X|X . \text{oclIsKindOf}(\text{OclAny})))$
apply(simp add: OclValid-def del: OclAllInstances-generic-def)
apply(simp only: UML-Set.OclForall-def refl if-True)

$OclAllInstances-generic-defined[simplified\ OclValid-def]$
apply(*simp only*: $OclAllInstances-generic-def$)
apply(*subst* (1 2 3) $Abs-Set_{base-inverse}$, *simp add*: $bot-option-def$)
by(*simp add*: $OclIsKindOf_{OclAny-Person}$)

lemma $Person-allInstances-at-post-oclIsKindOf_{OclAny}$:
 $\tau \models (Person.allInstances() \rightarrow_{forAllSet(X|X.oclIsKindOf(OclAny))})$
unfolding $OclAllInstances-at-post-def$
by(*rule* $Person-allInstances-generic-oclIsKindOf_{OclAny}$)

lemma $Person-allInstances-at-pre-oclIsKindOf_{OclAny}$:
 $\tau \models (Person.allInstances@pre() \rightarrow_{forAllSet(X|X.oclIsKindOf(OclAny))})$
unfolding $OclAllInstances-at-pre-def$
by(*rule* $Person-allInstances-generic-oclIsKindOf_{OclAny}$)

lemma $Person-allInstances-generic-oclIsKindOf_{Person}$:
 $\tau \models ((OclAllInstances-generic\ pre-post\ Person) \rightarrow_{forAllSet(X|X.oclIsKindOf(Person))})$
apply(*simp add*: $OclValid-def\ del: OclAllInstances-generic-def$)
apply(*simp only*: $UML-Set.OclForall-def\ refl\ if-True$
 $OclAllInstances-generic-defined[simplified\ OclValid-def]$)
apply(*simp only*: $OclAllInstances-generic-def$)
apply(*subst* (1 2 3) $Abs-Set_{base-inverse}$, *simp add*: $bot-option-def$)
by(*simp add*: $OclIsKindOf_{Person-Person}$)

lemma $Person-allInstances-at-post-oclIsKindOf_{Person}$:
 $\tau \models (Person.allInstances() \rightarrow_{forAllSet(X|X.oclIsKindOf(Person))})$
unfolding $OclAllInstances-at-post-def$
by(*rule* $Person-allInstances-generic-oclIsKindOf_{Person}$)

lemma $Person-allInstances-at-pre-oclIsKindOf_{Person}$:
 $\tau \models (Person.allInstances@pre() \rightarrow_{forAllSet(X|X.oclIsKindOf(Person))})$
unfolding $OclAllInstances-at-pre-def$
by(*rule* $Person-allInstances-generic-oclIsKindOf_{Person}$)

5.8. The Accessors (any, boss, salary)

Should be generated entirely from a class-diagram.

5.8.1. Definition

definition $eval-extract :: (\mathfrak{A}, ('a::object)\ option\ option)\ val$
 $\Rightarrow (oid \Rightarrow (\mathfrak{A}, 'c::null)\ val)$
 $\Rightarrow (\mathfrak{A}, 'c::null)\ val$
where $eval-extract\ X\ f = (\lambda\ \tau.\ case\ X\ \tau\ of$
 $\quad \perp \Rightarrow invalid\ \tau \quad \text{--- exception propagation}$
 $\quad | \ \lrcorner \ \perp \ \lrcorner \Rightarrow invalid\ \tau \quad \text{--- dereferencing null pointer}$
 $\quad | \ \lrcorner \ obj \ \lrcorner \Rightarrow f\ (oid-of\ obj)\ \tau)$

definition $deref-oid_{Person} :: (\mathfrak{A}\ state \times \mathfrak{A}\ state \Rightarrow \mathfrak{A}\ state)$
 $\Rightarrow (type_{Person} \Rightarrow (\mathfrak{A}, 'c::null)\ val)$
 $\Rightarrow oid$
 $\Rightarrow (\mathfrak{A}, 'c::null)\ val$
where $deref-oid_{Person}\ fst-snd\ f\ oid = (\lambda\ \tau.\ case\ (heap\ (fst-snd\ \tau))\ oid\ of$
 $\quad | \ \lrcorner \ in_{Person}\ obj \ \lrcorner \Rightarrow f\ obj\ \tau$
 $\quad | \ - \Rightarrow invalid\ \tau)$

definition $deref-oid_{OclAny} :: (\mathfrak{A} \text{ state} \times \mathfrak{A} \text{ state} \Rightarrow \mathfrak{A} \text{ state})$
 $\Rightarrow (type_{OclAny} \Rightarrow (\mathfrak{A}, 'c::null)val)$
 $\Rightarrow oid$
 $\Rightarrow (\mathfrak{A}, 'c::null)val$

where $deref-oid_{OclAny} \text{ fst-snd } f \text{ oid} = (\lambda \tau. \text{ case } (heap \text{ (fst-snd } \tau)) \text{ oid of}$
 $\quad \perp \text{ in}_{OclAny} \text{ obj} _ \Rightarrow f \text{ obj } \tau$
 $\quad | _ \Rightarrow \text{ invalid } \tau)$

pointer undefined in state or not referencing a type conform object representation

definition $select_{OclAny} \mathcal{ANY} f = (\lambda X. \text{ case } X \text{ of}$
 $\quad (mk_{OclAny} \text{ - } \perp) \Rightarrow null$
 $\quad | (mk_{OclAny} \text{ - } _ \text{ any}__) \Rightarrow f (\lambda x \text{ - } _ \text{ x}__) \text{ any})$

definition $select_{Person} \mathcal{BOSS} f = (\lambda X. \text{ case } X \text{ of}$
 $\quad (mk_{Person} \text{ - - } \perp) \Rightarrow null$ — object contains null pointer
 $\quad | (mk_{Person} \text{ - - } _ \text{ boss}__) \Rightarrow f (\lambda x \text{ - } _ \text{ x}__) \text{ boss})$

definition $select_{Person} \mathcal{SALARY} f = (\lambda X. \text{ case } X \text{ of}$
 $\quad (mk_{Person} \text{ - } \perp \text{ -}) \Rightarrow null$
 $\quad | (mk_{Person} \text{ - } _ \text{ salary}__) \Rightarrow f (\lambda x \text{ - } _ \text{ x}__) \text{ salary})$

definition $in\text{-pre}\text{-state} = \text{fst}$

definition $in\text{-post}\text{-state} = \text{snd}$

definition $reconst\text{-basetype} = (\lambda \text{ convert } x. \text{ convert } x)$

definition $dot_{OclAny} \mathcal{ANY} :: OclAny \Rightarrow \text{ - } (\langle (1(-).any) \rangle 50)$
where $(X).any = \text{eval-extract } X$
 $(deref-oid_{OclAny} \text{ in-post-state}$
 $\quad (select_{OclAny} \mathcal{ANY}$
 $\quad \quad reconst\text{-basetype}))$

definition $dot_{Person} \mathcal{BOSS} :: Person \Rightarrow Person (\langle (1(-).boss) \rangle 50)$
where $(X).boss = \text{eval-extract } X$
 $(deref-oid_{Person} \text{ in-post-state}$
 $\quad (select_{Person} \mathcal{BOSS}$
 $\quad \quad (deref-oid_{Person} \text{ in-post-state})))$

definition $dot_{Person} \mathcal{SALARY} :: Person \Rightarrow Integer (\langle (1(-).salary) \rangle 50)$
where $(X).salary = \text{eval-extract } X$
 $(deref-oid_{Person} \text{ in-post-state}$
 $\quad (select_{Person} \mathcal{SALARY}$
 $\quad \quad reconst\text{-basetype}))$

definition $dot_{OclAny} \mathcal{ANY}\text{-at-pre} :: OclAny \Rightarrow \text{ - } (\langle (1(-).any@pre) \rangle 50)$
where $(X).any@pre = \text{eval-extract } X$
 $(deref-oid_{OclAny} \text{ in-pre-state}$
 $\quad (select_{OclAny} \mathcal{ANY}$
 $\quad \quad reconst\text{-basetype}))$

definition $dot_{Person} \mathcal{BOSS}\text{-at-pre} :: Person \Rightarrow Person (\langle (1(-).boss@pre) \rangle 50)$

where $(X).boss@pre = eval-extract\ X$
 $(deref-oid_{Person}\ in-pre-state$
 $(select_{Person}\ BOSS$
 $(deref-oid_{Person}\ in-pre-state)))$

definition $dot_{Person}\ SALARY-at-pre:: Person \Rightarrow Integer\ (\langle(1(-).salary@pre)\rangle\ 50)$

where $(X).salary@pre = eval-extract\ X$
 $(deref-oid_{Person}\ in-pre-state$
 $(select_{Person}\ SALARY$
 $reconst-basetype))$

lemmas $dot-accessor =$
 $dot_{OclAny}\ ANY-def$
 $dot_{Person}\ BOSS-def$
 $dot_{Person}\ SALARY-def$
 $dot_{OclAny}\ ANY-at-pre-def$
 $dot_{Person}\ BOSS-at-pre-def$
 $dot_{Person}\ SALARY-at-pre-def$

5.8.2. Context Passing

lemmas $[simp] = eval-extract-def$

lemma $cp-dot_{OclAny}\ ANY: ((X).any)\ \tau = ((\lambda-. X\ \tau).any)\ \tau$ **by** $(simp\ add: dot-accessor)$
lemma $cp-dot_{Person}\ BOSS: ((X).boss)\ \tau = ((\lambda-. X\ \tau).boss)\ \tau$ **by** $(simp\ add: dot-accessor)$
lemma $cp-dot_{Person}\ SALARY: ((X).salary)\ \tau = ((\lambda-. X\ \tau).salary)\ \tau$ **by** $(simp\ add: dot-accessor)$

lemma $cp-dot_{OclAny}\ ANY-at-pre: ((X).any@pre)\ \tau = ((\lambda-. X\ \tau).any@pre)\ \tau$ **by** $(simp\ add: dot-accessor)$
lemma $cp-dot_{Person}\ BOSS-at-pre: ((X).boss@pre)\ \tau = ((\lambda-. X\ \tau).boss@pre)\ \tau$ **by** $(simp\ add: dot-accessor)$
lemma $cp-dot_{Person}\ SALARY-at-pre: ((X).salary@pre)\ \tau = ((\lambda-. X\ \tau).salary@pre)\ \tau$ **by** $(simp\ add: dot-accessor)$

lemmas $cp-dot_{OclAny}\ ANY-I [simp, intro!]=$
 $cp-dot_{OclAny}\ ANY[THEN\ allI[THEN\ allI],$
 $of\ \lambda\ X\ -. X\ \lambda - \tau.\ \tau, THEN\ cpI1]$

lemmas $cp-dot_{OclAny}\ ANY-at-pre-I [simp, intro!]=$
 $cp-dot_{OclAny}\ ANY-at-pre[THEN\ allI[THEN\ allI],$
 $of\ \lambda\ X\ -. X\ \lambda - \tau.\ \tau, THEN\ cpI1]$

lemmas $cp-dot_{Person}\ BOSS-I [simp, intro!]=$
 $cp-dot_{Person}\ BOSS[THEN\ allI[THEN\ allI],$
 $of\ \lambda\ X\ -. X\ \lambda - \tau.\ \tau, THEN\ cpI1]$

lemmas $cp-dot_{Person}\ BOSS-at-pre-I [simp, intro!]=$
 $cp-dot_{Person}\ BOSS-at-pre[THEN\ allI[THEN\ allI],$
 $of\ \lambda\ X\ -. X\ \lambda - \tau.\ \tau, THEN\ cpI1]$

lemmas $cp-dot_{Person}\ SALARY-I [simp, intro!]=$
 $cp-dot_{Person}\ SALARY[THEN\ allI[THEN\ allI],$
 $of\ \lambda\ X\ -. X\ \lambda - \tau.\ \tau, THEN\ cpI1]$

lemmas $cp-dot_{Person}\ SALARY-at-pre-I [simp, intro!]=$
 $cp-dot_{Person}\ SALARY-at-pre[THEN\ allI[THEN\ allI],$
 $of\ \lambda\ X\ -. X\ \lambda - \tau.\ \tau, THEN\ cpI1]$

5.8.3. Execution with Invalid or Null as Argument

lemma $dot_{OclAny}\ ANY-nullstrict [simp]: (null).any = invalid$
by $(rule\ ext, simp\ add: dot-accessor\ null-fun-def\ null-option-def\ bot-option-def\ invalid-def)$

lemma $dot_{OclAny} \mathcal{ANY}$ -at-pre-nullstrict [simp] : (null).any@pre = invalid
by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def)
lemma $dot_{OclAny} \mathcal{ANY}$ -strict [simp] : (invalid).any = invalid
by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def)
lemma $dot_{OclAny} \mathcal{ANY}$ -at-pre-strict [simp] : (invalid).any@pre = invalid
by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def)

lemma $dot_{Person} \mathcal{BOSS}$ -nullstrict [simp]: (null).boss = invalid
by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def)
lemma $dot_{Person} \mathcal{BOSS}$ -at-pre-nullstrict [simp] : (null).boss@pre = invalid
by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def)
lemma $dot_{Person} \mathcal{BOSS}$ -strict [simp] : (invalid).boss = invalid
by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def)
lemma $dot_{Person} \mathcal{BOSS}$ -at-pre-strict [simp] : (invalid).boss@pre = invalid
by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def)

lemma $dot_{Person} \mathcal{SALARY}$ -nullstrict [simp]: (null).salary = invalid
by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def)
lemma $dot_{Person} \mathcal{SALARY}$ -at-pre-nullstrict [simp] : (null).salary@pre = invalid
by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def)
lemma $dot_{Person} \mathcal{SALARY}$ -strict [simp] : (invalid).salary = invalid
by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def)
lemma $dot_{Person} \mathcal{SALARY}$ -at-pre-strict [simp] : (invalid).salary@pre = invalid
by(rule ext, simp add: dot-accessor null-fun-def null-option-def bot-option-def invalid-def)

5.8.4. Representation in States

lemma $dot_{Person} \mathcal{BOSS}$ -def-mono: $\tau \models \delta(X .boss) \implies \tau \models \delta(X)$
apply(case-tac $\tau \models (X \triangleq invalid)$, insert StrongEq-L-subst2[**where** $P = (\lambda x. (\delta (x .boss)))$ and $\tau = \tau$ and $x = X$ and $y = invalid$], simp add: foundation16')
apply(case-tac $\tau \models (X \triangleq null)$, insert StrongEq-L-subst2[**where** $P = (\lambda x. (\delta (x .boss)))$ and $\tau = \tau$ and $x = X$ and $y = null$], simp add: foundation16')
by(simp add: defined-split)

lemma repr-boss:
assumes $A : \tau \models \delta(x .boss)$
shows is-represented-in-state in-post-state (x .boss) Person τ
apply(insert A[simplified foundation16])
 A [THEN $dot_{Person} \mathcal{BOSS}$ -def-mono, simplified foundation16])
unfolding is-represented-in-state-def
 $dot_{Person} \mathcal{BOSS}$ -def eval-extract-def select $_{Person} \mathcal{BOSS}$ -def in-post-state-def
by(auto simp: deref-oid $_{Person}$ -def bot-fun-def bot-option-def null-option-def null-fun-def invalid-def
OclAsType $_{Person}$ - \mathcal{A} -def image-def ran-def
split: type $_{Person}$.split option.split \mathcal{A} .split)

lemma repr-bossX :
assumes $A : \tau \models \delta(x .boss)$
shows $\tau \models ((Person .allInstances()) \rightarrow includes_{Set}(x .boss))$
proof –
have $B : \bigwedge S f. (x .boss) \tau \in (Some \text{ ' } f \text{ ' } S) \implies$
 $(x .boss) \tau \in (Some \text{ ' } (f \text{ ' } S - \{None\}))$
apply(auto simp: image-def ran-def, metis)
by(insert A[simplified foundation16], simp add: null-option-def bot-option-def)
show ?thesis
apply(insert repr-boss[OF A] OclAllInstances-at-post-defined[**where** $H = Person$ and $\tau = \tau$])


```

oid1 ↦ in_Person (mk_Person oid1 1200 None),
oid2
oid3 ↦ in_Person (mk_Person oid3 2600 oid4),
oid4 ↦ in_Person person5,
oid5 ↦ in_Person (mk_Person oid5 2300 oid3),
oid6
oid7
oid8 ↦ in_Person person9),
assocs = Map.empty

```

definition

```

σ1' ≡ (| heap = Map.empty(oid0 ↦ in_Person person1,
oid1 ↦ in_Person person2,
oid2 ↦ in_Person person3,
oid3 ↦ in_Person person4,
oid4
oid5 ↦ in_Person person6,
oid6 ↦ in_OclAny person7,
oid7 ↦ in_OclAny person8,
oid8 ↦ in_Person person9),
assocs = Map.empty
|)

```

definition σ₀ ≡ (| heap = Map.empty, assocs = Map.empty |)

lemma *basic-τ-wff*: WFF(σ₁,σ₁')

by(*auto simp*: WFF-def σ₁-def σ₁'-def

```

oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def
oid-of-ℳ-def oid-of-type_Person-def oid-of-type_OclAny-def
person1-def person2-def person3-def person4-def
person5-def person6-def person7-def person8-def person9-def)

```

lemma [*simp,code-unfold*]: dom (heap σ₁) = {oid0,oid1,~~oid2~~,oid3,oid4,oid5,~~oid6~~,~~oid7~~,oid8}

by(*auto simp*: σ₁-def)

lemma [*simp,code-unfold*]: dom (heap σ₁') = {oid0,oid1,oid2,oid3,~~oid4~~,oid5,oid6,oid7,oid8}

by(*auto simp*: σ₁'-def)**definition** X_{Person1} :: Person ≡ λ ·.⊥ person1 ⊥

definition X_{Person2} :: Person ≡ λ ·.⊥ person2 ⊥

definition X_{Person3} :: Person ≡ λ ·.⊥ person3 ⊥

definition X_{Person4} :: Person ≡ λ ·.⊥ person4 ⊥

definition X_{Person5} :: Person ≡ λ ·.⊥ person5 ⊥

definition X_{Person6} :: Person ≡ λ ·.⊥ person6 ⊥

definition X_{Person7} :: OclAny ≡ λ ·.⊥ person7 ⊥

definition X_{Person8} :: OclAny ≡ λ ·.⊥ person8 ⊥

definition X_{Person9} :: Person ≡ λ ·.⊥ person9 ⊥

lemma [*code-unfold*]: ((x::Person) ≐ y) = StrictRefEq_{Object} x y **by**(*simp only*: StrictRefEq_{Object}-Person)

lemma [*code-unfold*]: ((x::OclAny) ≐ y) = StrictRefEq_{Object} x y **by**(*simp only*: StrictRefEq_{Object}-OclAny)

lemmas [*simp,code-unfold*] =

OclAsType_{OclAny}-OclAny

OclAsType_{OclAny}-Person

OclAsType_{Person}-OclAny

OclAsType_{Person}-Person

OclIsTypeOf_{OclAny}-OclAny

OclIsTypeOf_{OclAny}-Person

OclIsTypeOf_{Person}-OclAny
OclIsTypeOf_{Person}-Person

OclIsKindOf_{OclAny}-OclAny
OclIsKindOf_{OclAny}-Person
OclIsKindOf_{Person}-OclAny
OclIsKindOf_{Person}-Person

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .salary \langle \rangle 1000)$

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .salary \doteq 1300)$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .salary@pre \doteq 1000)$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .salary@pre \langle \rangle 1300)$

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .boss \langle \rangle X_{Person1})$

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .boss .salary \doteq 1800)$

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .boss .boss \langle \rangle X_{Person1})$

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person1} .boss .boss \doteq X_{Person2})$

Assert $(\sigma_1, \sigma_1') \models (X_{Person1} .boss@pre .salary \doteq 1800)$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .boss@pre .salary@pre \doteq 1200)$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .boss@pre .salary@pre \langle \rangle 1800)$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .boss@pre \doteq X_{Person2})$

Assert $(\sigma_1, \sigma_1') \models (X_{Person1} .boss@pre .boss \doteq X_{Person2})$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person1} .boss@pre .boss@pre \doteq null)$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models \text{not}(v(X_{Person1} .boss@pre .boss@pre .boss@pre))$

lemma $(\sigma_1, \sigma_1') \models (X_{Person1} .oclIsMaintained())$

by(*simp add: OclValid-def OclIsMaintained-def*

σ₁-def σ₁'-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*)

Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person1} .oclIsTypeOf(Person))$

Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models \text{not}(X_{Person1} .oclIsTypeOf(OclAny))$

Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person1} .oclIsKindOf(Person))$

Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models (X_{Person1} .oclIsKindOf(OclAny))$

Assert $\bigwedge_{s_{pre} s_{post}} \cdot (s_{pre}, s_{post}) \models \text{not}(X_{Person1} .oclAsType(OclAny) .oclIsTypeOf(OclAny))$

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person2} .salary \doteq 1800)$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person2} .salary@pre \doteq 1200)$

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person2} .boss \doteq X_{Person2})$

Assert $(\sigma_1, \sigma_1') \models (X_{Person2} .boss .salary@pre \doteq 1200)$

Assert $(\sigma_1, \sigma_1') \models (X_{Person2} .boss .boss@pre \doteq null)$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person2} .boss@pre \doteq null)$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{Person2} .boss@pre \langle \rangle X_{Person2})$

Assert $(\sigma_1, \sigma_1') \models (X_{Person2} .boss@pre \langle \rangle (X_{Person2} .boss))$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models \text{not}(v(X_{Person2} .boss@pre .boss))$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models \text{not}(v(X_{Person2} .boss@pre .salary@pre))$

lemma $(\sigma_1, \sigma_1') \models (X_{Person2} .oclIsMaintained())$

by(*simp add: OclValid-def OclIsMaintained-def σ₁-def σ₁'-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)

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person3} .salary \doteq null)$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models \text{not}(v(X_{Person3} .salary@pre))$

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{Person3} .boss \doteq null)$

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models \text{not}(v(X_{Person3} .boss .salary))$

Assert $\wedge_{s_{post}. (\sigma_1, s_{post})} \models \text{not}(v(X_{Person3} .boss@pre))$
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-typePerson-def*)

Assert $\wedge_{s_{post}. (\sigma_1, s_{post})} \models (X_{Person4} .boss@pre \doteq X_{Person5})$
Assert $(\sigma_1, \sigma_1') \models \text{not}(v(X_{Person4} .boss@pre .salary))$
Assert $\wedge_{s_{post}. (\sigma_1, s_{post})} \models (X_{Person4} .boss@pre .salary@pre \doteq \mathbf{3500})$
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-typePerson-def*)

Assert $\wedge_{s_{pre}. (\sigma_1, s_{pre})} \models \text{not}(v(X_{Person5} .salary))$
Assert $\wedge_{s_{post}. (\sigma_1, s_{post})} \models (X_{Person5} .salary@pre \doteq \mathbf{3500})$
Assert $\wedge_{s_{pre}. (\sigma_1, s_{pre})} \models \text{not}(v(X_{Person5} .boss))$
lemma $(\sigma_1, \sigma_1') \models (X_{Person5} .oclIsDeleted())$
by(*simp add: OclNot-def OclValid-def OclIsDeleted-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-typePerson-def*)

Assert $\wedge_{s_{pre}. (\sigma_1, s_{pre})} \models \text{not}(v(X_{Person6} .boss .salary@pre))$
Assert $\wedge_{s_{post}. (\sigma_1, s_{post})} \models (X_{Person6} .boss@pre \doteq X_{Person4})$
Assert $(\sigma_1, \sigma_1') \models (X_{Person6} .boss@pre .salary \doteq \mathbf{2900})$
Assert $\wedge_{s_{post}. (\sigma_1, s_{post})} \models (X_{Person6} .boss@pre .salary@pre \doteq \mathbf{2600})$
Assert $\wedge_{s_{post}. (\sigma_1, s_{post})} \models (X_{Person6} .boss@pre .boss@pre \doteq X_{Person5})$
lemma $(\sigma_1, \sigma_1') \models (X_{Person6} .oclIsMaintained())$
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-typePerson-def*)

Assert $\wedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models v(X_{Person7} .oclAsType(Person))$
Assert $\wedge_{s_{post}. (\sigma_1, s_{post})} \models \text{not}(v(X_{Person7} .oclAsType(Person) .boss@pre))$
lemma $\wedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models ((X_{Person7} .oclAsType(Person) .oclAsType(OclAny) .oclAsType(Person)) \doteq (X_{Person7} .oclAsType(Person)))$
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-typeOclAny-def*)

Assert $\wedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models (X_{Person8} <> X_{Person7})$
Assert $\wedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models \text{not}(v(X_{Person8} .oclAsType(Person)))$
Assert $\wedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models (X_{Person8} .oclIsTypeOf(OclAny))$
Assert $\wedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models \text{not}(X_{Person8} .oclIsTypeOf(Person))$
Assert $\wedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models \text{not}(X_{Person8} .oclIsKindOf(Person))$
Assert $\wedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models (X_{Person8} .oclIsKindOf(OclAny))$

lemma σ -modifiedonly: $(\sigma_1, \sigma_1') \models (\text{Set}\{ X_{Person1} .oclAsType(OclAny) , X_{Person2} .oclAsType(OclAny) \})$


```

, XPerson6 .oclAsType(OclAny)
//XPerson7 .oclAsType(OclAny)
//XPerson8 .oclAsType(OclAny)
//XPerson9 .oclAsType(OclAny)})
apply(cut-tac  $\sigma$ -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 UML-Set.OclExcluding.cp0,
  subst (asm) cp-OclIsModifiedOnly, simp add: including4 excluding1)

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: OclIncluding-rep-set mtSet-rep-set
  oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def)
apply(simp add: StrictRefEqObject-def oid-of-option-def oid-of-typeOclAny-def OclNot-def OclValid-def
  null-option-def bot-option-def)

done
qed

lemma perm- $\sigma_1'$  :  $\sigma_1' = \langle \text{heap} = \text{Map.empty}$ 
  (oid8  $\mapsto$  inPerson person9,
  oid7  $\mapsto$  inOclAny person8,
  oid6  $\mapsto$  inOclAny person7,
  oid5  $\mapsto$  inPerson person6,
  oid4
  oid3  $\mapsto$  inPerson person4,
  oid2  $\mapsto$  inPerson person3,
  oid1  $\mapsto$  inPerson person2,
  oid0  $\mapsto$  inPerson person1)
  , assocs = assocs  $\sigma_1'$   $\rangle$ 

proof –
note P = fun-upd-twist
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) P, simp)
apply(subst (2) P, simp) apply(subst (1) P, simp)
apply(subst (3) P, simp) apply(subst (2) P, simp) apply(subst (1) P, simp)
apply(subst (4) P, simp) apply(subst (3) P, simp) apply(subst (2) P, simp) apply(subst (1) P, simp)
apply(subst (5) P, simp) apply(subst (4) P, simp) apply(subst (3) P, simp) apply(subst (2) P, simp)
apply(subst (1) P, simp)
apply(subst (6) P, simp) apply(subst (5) P, simp) apply(subst (4) P, simp) apply(subst (3) P, simp)
apply(subst (2) P, simp) apply(subst (1) P, simp)
apply(subst (7) P, simp) apply(subst (6) P, simp) apply(subst (5) P, simp) apply(subst (4) P, simp)
apply(subst (3) P, simp) apply(subst (2) P, simp) apply(subst (1) P, simp)
by(simp)
qed

declare const-ss [simp]

lemma  $\wedge\sigma_1$ .

```

$(\sigma_1, \sigma_1') \models (Person .allInstances() \doteq Set\{ X_{Person1}, X_{Person2}, X_{Person3}, X_{Person4}, X_{Person5}, X_{Person6}, X_{Person7} .oclAsType(Person), X_{Person8}, X_{Person9} \})$

apply(subst perm- σ_1')

apply(simp only: oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def

$X_{Person1}$ -def $X_{Person2}$ -def $X_{Person3}$ -def $X_{Person4}$ -def

$X_{Person5}$ -def $X_{Person6}$ -def $X_{Person7}$ -def $X_{Person8}$ -def $X_{Person9}$ -def

person7-def)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- \mathcal{A} -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule OclIncluding-cong, simp, simp)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- \mathcal{A} -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule OclIncluding-cong, simp, simp)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- \mathcal{A} -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule OclIncluding-cong, simp, simp)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- \mathcal{A} -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule OclIncluding-cong, simp, simp)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- \mathcal{A} -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule OclIncluding-cong, simp, simp)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- \mathcal{A} -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule OclIncluding-cong, simp, simp)

apply(subst state-update-vs-allInstances-at-post-ntc, simp, simp add: OclAsTypePerson- \mathcal{A} -def

person8-def, simp, rule const-StrictRefEqSet-including,

simp, simp, simp)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypePerson- \mathcal{A} -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule OclIncluding-cong, simp, simp)

apply(rule state-update-vs-allInstances-at-post-empty)

by(simp-all add: OclAsTypePerson- \mathcal{A} -def)

lemma $\bigwedge \sigma_1$.

$(\sigma_1, \sigma_1') \models (OclAny .allInstances() \doteq Set\{ X_{Person1} .oclAsType(OclAny), X_{Person2} .oclAsType(OclAny),$

$X_{Person3} .oclAsType(OclAny), X_{Person4} .oclAsType(OclAny)$

$X_{Person5} .oclAsType(OclAny),$

$X_{Person6} .oclAsType(OclAny), X_{Person7}, X_{Person8}, X_{Person9} .oclAsType(OclAny) \})$

apply(subst perm- σ_1')

apply(simp only: oid0-def oid1-def oid2-def oid3-def oid4-def oid5-def oid6-def oid7-def oid8-def

$X_{Person1}$ -def $X_{Person2}$ -def $X_{Person3}$ -def $X_{Person4}$ -def $X_{Person5}$ -def $X_{Person6}$ -def $X_{Person7}$ -def

$X_{Person8}$ -def $X_{Person9}$ -def

person1-def person2-def person3-def person4-def person5-def person6-def person9-def)

apply(subst state-update-vs-allInstances-at-post-tc, simp, simp add: OclAsTypeOclAny- \mathcal{A} -def, simp, rule const-StrictRefEqSet-including, simp, simp, simp, rule OclIncluding-cong, simp, simp)+

apply(rule state-update-vs-allInstances-at-post-empty)

by(simp-all add: OclAsTypeOclAny- \mathcal{A} -def)

end

theory

Design-OCL

imports

Design-UML

begin

5.10. OCL Part: Invariant

These recursive predicates can be defined conservatively by greatest fix-point constructions—automatically. See [4, 6] for details. For the purpose of this example, we state them as axioms here.

```

context Person
  inv label : self .boss <> null implies (self .salary \<le>
    ((self .boss) .salary))

```

definition $Person\text{-}label_{inv} :: Person \Rightarrow Boolean$
where $Person\text{-}label_{inv} (self) \equiv$
 $(self .boss <> null \text{ implies } (self .salary \leq_{int} ((self .boss) .salary)))$

definition $Person\text{-}label_{invATpre} :: Person \Rightarrow Boolean$
where $Person\text{-}label_{invATpre} (self) \equiv$
 $(self .boss@pre <> null \text{ implies } (self .salary@pre \leq_{int} ((self .boss@pre) .salary@pre)))$

definition $Person\text{-}label_{globalinv} :: Boolean$
where $Person\text{-}label_{globalinv} \equiv (Person .allInstances() \rightarrow \text{forAll}_{Set}(x \mid Person\text{-}label_{inv} (x)) \text{ and}$
 $(Person .allInstances@pre() \rightarrow \text{forAll}_{Set}(x \mid Person\text{-}label_{invATpre} (x))))$

lemma $\tau \models \delta (X .boss) \implies \tau \models Person .allInstances() \rightarrow \text{includes}_{Set}(X .boss) \wedge$
 $\tau \models Person .allInstances() \rightarrow \text{includes}_{Set}(X)$

oops

lemma $REC\text{-}pre : \tau \models Person\text{-}label_{globalinv}$
 $\implies \tau \models Person .allInstances() \rightarrow \text{includes}_{Set}(X) \text{ — } X \text{ represented object in state}$
 $\implies \exists REC. \tau \models REC(X) \triangleq (Person\text{-}label_{inv} (X) \text{ and } (X .boss <> null \text{ implies } REC(X .boss)))$

oops

This allows to state a predicate:

axiomatization $inv_{Person\text{-}label} :: Person \Rightarrow Boolean$
where $inv_{Person\text{-}label}\text{-}def:$
 $(\tau \models Person .allInstances() \rightarrow \text{includes}_{Set}(self)) \implies$
 $(\tau \models (inv_{Person\text{-}label}(self) \triangleq (self .boss <> null \text{ implies}$
 $(self .salary \leq_{int} ((self .boss) .salary)) \text{ and}$
 $inv_{Person\text{-}label}(self .boss))))$

axiomatization $inv_{Person\text{-}labelATpre} :: Person \Rightarrow Boolean$
where $inv_{Person\text{-}labelATpre}\text{-}def:$
 $(\tau \models Person .allInstances@pre() \rightarrow \text{includes}_{Set}(self)) \implies$
 $(\tau \models (inv_{Person\text{-}labelATpre}(self) \triangleq (self .boss@pre <> null \text{ implies}$
 $(self .salary@pre \leq_{int} ((self .boss@pre) .salary@pre)) \text{ and}$
 $inv_{Person\text{-}labelATpre}(self .boss@pre))))$

lemma $inv\text{-}1 :$
 $(\tau \models Person .allInstances() \rightarrow \text{includes}_{Set}(self)) \implies$
 $(\tau \models inv_{Person\text{-}label}(self) = ((\tau \models (self .boss \doteq null)) \vee$
 $(\tau \models (self .boss <> null) \wedge$
 $\tau \models ((self .salary) \leq_{int} (self .boss .salary)) \wedge$
 $\tau \models (inv_{Person\text{-}label}(self .boss))))))$

oops

lemma $inv\text{-}2 :$
 $(\tau \models Person .allInstances@pre() \rightarrow \text{includes}_{Set}(self)) \implies$
 $(\tau \models inv_{Person\text{-}labelATpre}(self) = ((\tau \models (self .boss@pre \doteq null)) \vee$
 $(\tau \models (self .boss@pre <> null) \wedge$

$$\begin{aligned}
& (\tau \models (self .boss@pre .salary@pre \leq_{int} self .salary@pre)) \wedge \\
& (\tau \models (inv_{Person-labelATpre}(self .boss@pre))))
\end{aligned}$$

oops

A very first attempt to characterize the axiomatization by an inductive definition - this can not be the last word since too weak (should be equality!)

coinductive *inv* :: *Person* \Rightarrow (\mathfrak{A})*st* \Rightarrow *bool* **where**
 $(\tau \models (\delta self)) \Rightarrow ((\tau \models (self .boss \doteq null)) \vee$
 $(\tau \models (self .boss <> null) \wedge (\tau \models (self .boss .salary \leq_{int} self .salary)) \wedge$
 $(inv(self .boss)\tau)))$
 $\Rightarrow (inv self \tau)$

5.11. OCL Part: The Contract of a Recursive Query

This part is analogous to the Analysis Model and skipped here.

end

Part II.

Conclusion

6. Conclusion

6.1. Lessons Learned and Contributions

We provided a typed and type-safe shallow embedding of the core of UML [30, 31] and OCL [32]. Shallow embedding means that types of OCL were mapped by the embedding one-to-one to types in Isabelle/HOL [27]. We followed the usual methodology to build up the theory uniquely by conservative extensions of all operators in a denotational style and to derive logical and algebraic (execution) rules from them; thus, we can guarantee the logical consistency of the library and instances of the class model construction. The class models were given a closed-world interpretation as object-oriented datatype theories, as long as it follows the described methodology.¹ Moreover, all derived execution rules are by construction type-safe (which would be an issue, if we had chosen to use an object universe construction in Zermelo-Fraenkel set theory as an alternative approach to subtyping.). In more detail, our theory gives answers and concrete solutions to a number of open major issues for the UML/OCL standardization:

1. the role of the two exception elements `invalid` and `null`, the former usually assuming strict evaluation while the latter ruled by non-strict evaluation.
2. the functioning of the resulting four-valued logic, together with safe rules (for example `foundation9` – `foundation12` in Section 2.1.5) that allow a reduction to two-valued reasoning as required for many automated provers. The resulting logic still enjoys the rules of a strong Kleene Logic in the spirit of the Amsterdam Manifesto [18].
3. the complicated life resulting from the two necessary equalities: the standard’s “strict weak referential equality” as default (written \doteq throughout this document) and the strong equality (written \triangleq), which follows the logical Leibniz principle that “equals can be replaced by equals.” Which is not necessarily the case if `invalid` or objects of different states are involved.
4. a type-safe representation of objects and a clarification of the old idea of a one-to-one correspondence between object representations and object-id’s, which became a state invariant.
5. a simple concept of state-framing via the novel operator `_->oclIsModifiedOnly()` and its consequences for strong and weak equality.
6. a semantic view on subtyping clarifying the role of static and dynamic type (aka *apparent* and *actual* type in Java terminology), and its consequences for casts, dynamic type-tests, and static types.
7. a semantic view on path expressions, that clarify the role of `invalid` and `null` as well as the tricky issues related to de-referentiation in pre- and post state.
8. an optional extension of the OCL semantics by *infinite* sets that provide means to represent “the set of potential objects or values” to state properties over them (this will be an important feature if OCL is intended to become a full-blown code annotation language in the spirit of JML [25] for semi-automated code verification, and has been considered desirable in the Aachen Meeting [14]).

¹Our two examples of `Employee_AnalysisModel` and `Employee_DesignModel` (see Chapter 4 and Figure 0.3.8 as well as Chapter 5 and Figure 0.3.8) sketch how this construction can be captured by an automated process; its implementation is described elsewhere.

Moreover, we managed to make our theory in large parts executable, which allowed us to include mechanically checked value-statements that capture numerous corner-cases relevant for OCL implementors. Among many minor issues, we thus pin-pointed the behavior of `null` in collections as well as in casts and the desired `isKindOf`-semantics of `allInstances()`.

6.2. Lessons Learned

While our paper and pencil arguments, given in [12], 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 [34] or SMT-solvers like Z3 [19] completely impractical. Concretely, if the expression `not(null)` is defined `invalid` (as was the case in prior versions of the standard [32]), then standard involution does not hold, i.e., `not(not(A)) = A` does not hold universally. Similarly, if `null and null` is `invalid`, then not even idempotence `X and X = X` holds. We strongly argue in favor of a lattice-like organization, where `null` represents “more information” than `invalid` and the logical operators are monotone with respect to this semantical “information ordering.”

A similar experience with prior paper and pencil arguments was our investigation of the object-oriented data-models, in particular path-expressions [15]. The final presentation is again essentially correct, but the technical details concerning exception handling lead finally to a continuation-passing style of the (in future generated) definitions for accessors, casts and tests. Apparently, OCL semantics (as many other “real” programming and specification languages) is meanwhile too complex to be treated by informal arguments solely.

Featherweight OCL makes several minor deviations from the standard and showed how the previous constructions can be made correct and consistent, and the DNF-normalization as well as δ -closure laws (necessary for a transition into a two-valued presentation of OCL specifications ready for interpretation in SMT solvers (see [13] for details)) are valid in Featherweight OCL.

6.3. Conclusion and Future Work

Featherweight OCL concentrates on formalizing the semantics of a core subset of OCL in general and in particular on formalizing the consequences of a four-valued logic (i.e., OCL versions that support, besides the truth values `true` and `false` also the two exception values `invalid` and `null`).

In the following, we outline the following future extensions to use Featherweight OCL for a concrete fully fledged tool for OCL. There are essentially five extensions necessary:

- development of a compiler that compiles a textual or CASE tool representation (e.g., using XMI or the textual syntax of the USE tool [33]) of class models into an object-oriented data type theory automatically.
- Full support of OCL standard syntax in a front-end parser; Such a parser could also generate the necessary casts as well as 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 [13]. 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 both an integration of fast constraint solvers such as Z3 as well as test-case generation scenarios as described in [13].
- a setup in Featherweight OCL of the Nitpick animator [3]. It remains to be shown that the standard, Kodkod [34] based animator in Isabelle can give a similar quality of animation as the OCLexec Tool [24]

- a code-generator setup for Featherweight OCL for Isabelle’s code generator. For example, the Isabelle code generator supports the generation of F#, which would allow to use OCL specifications for testing arbitrary .net-based applications.

The first two extensions are sufficient to provide a formal proof environment for OCL 2.5 similar to HOL-OCL while the remaining extensions are geared towards increasing the degree of proof automation and usability as well as providing a tool-supported test methodology for UML/OCL.

Our work shows that developing a machine-checked formal semantics of recent OCL standards still reveals significant inconsistencies—even though this type of research is not new. In fact, we started our work already with the 1.x series of OCL. The reasons for this ongoing consistency problems of OCL standard are manifold. For example, the consequences of adding an additional exception value to OCL 2.2 are widespread across the whole language and many of them are also quite subtle. Here, a machine-checked formal semantics is of great value, as one is forced to formalize all details and subtleties. Moreover, the standardization process of the OMG, in which standards (e. g., the UML infrastructure and the OCL standard) that need to be aligned closely are developed quite independently, are prone to ad-hoc changes that attempt to align these standards. And, even worse, updating a standard document by voting on the acceptance (or rejection) of isolated text changes does not help either. Here, a tool for the editor of the standard that helps to check the consistency of the whole standard after each and every modifications can be of great value as well.

Bibliography

- [1] P. B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Kluwer Academic Publishers, Dordrecht, 2nd edition, 2002. ISBN 1-402-00763-9.
- [2] C. Barrett and C. Tinelli. Cvc3. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 2007. ISBN 978-3-540-73367-6. doi: 10.1007/978-3-540-73368-3_34.
- [3] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. C. Paulson, editors, *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer-Verlag, 2010. ISBN 978-3-642-14051-8. doi: 10.1007/978-3-642-14052-5_11.
- [4] A. D. Brucker. *An Interactive Proof Environment for Object-oriented Specifications*. PhD thesis, ETH Zurich, Mar. 2007. URL <http://www.brucker.ch/bibliography/abstract/brucker-interactive-2007>. ETH Dissertation No. 17097.
- [5] A. D. Brucker and B. Wolff. A proposal for a formal OCL semantics in Isabelle/HOL. In V. A. Carreño, C. A. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, number 2410 in *Lecture Notes in Computer Science*, pages 99–114. Springer-Verlag, Heidelberg, 2002. ISBN 3-540-44039-9. doi: 10.1007/3-540-45685-6_8. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-proposal-2002>.
- [6] A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-ocl-book-2006>.
- [7] A. D. Brucker and B. Wolff. An extensible encoding of object-oriented data models in hol. *Journal of Automated Reasoning*, 41:219–249, 2008. ISSN 0168-7433. doi: 10.1007/s10817-008-9108-3. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-extensible-2008-b>.
- [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, Heidelberg, 2008. doi: 10.1007/978-3-540-78743-3_8. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-ocl-2008>.
- [9] A. D. Brucker and B. Wolff. Semantics, calculi, and analysis for object-oriented specifications. *Acta Informatica*, 46(4):255–284, July 2009. ISSN 0001-5903. doi: 10.1007/s00236-009-0093-8. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-semantics-2009>.
- [10] A. D. Brucker, J. Doser, and B. Wolff. Semantic issues of OCL: Past, present, and future. *Electronic Communications of the EASST*, 5, 2006. ISSN 1863-2122. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-semantic-2006-b>.
- [11] A. D. Brucker, J. Doser, and B. Wolff. A model transformation semantics and analysis methodology for SecureUML. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS 2006: Model Driven Engineering Languages and Systems*, number 4199 in *Lecture Notes in Computer Science*, pages 306–320. Springer-Verlag, Heidelberg, 2006. doi: 10.1007/11880240_22. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-transformation-2006>. An extended version of this paper is available as ETH Technical Report, no. 524.

- [12] A. D. Brucker, M. P. Krieger, and B. Wolff. Extending OCL with null-references. In S. Gosh, editor, *Models in Software Engineering*, number 6002 in Lecture Notes in Computer Science, pages 261–275. Springer-Verlag, Heidelberg, 2009. doi: 10.1007/978-3-642-12261-3_25. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-ocl-null-2009>. Selected best papers from all satellite events of the MoDELS 2009 conference.
- [13] A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff. A specification-based test case generation method for UML/OCL. In J. Dingel and A. Solberg, editors, *MoDELS Workshops*, number 6627 in Lecture Notes in Computer Science, pages 334–348. Springer-Verlag, Heidelberg, 2010. ISBN 978-3-642-21209-3. doi: 10.1007/978-3-642-21210-9_33. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-ocl-testing-2010>. Selected best papers from all satellite events of the MoDELS 2010 conference. Workshop on OCL and Textual Modelling.
- [14] A. D. Brucker, D. Chiorean, T. Clark, B. Demuth, M. Gogolla, D. Plotnikov, B. Rumpe, E. D. Willink, and B. Wolff. Report on the Aachen OCL meeting. In J. Cabot, M. Gogolla, I. Rath, and E. Willink, editors, *Proceedings of the MODELS 2013 OCL Workshop (OCL 2013)*, volume 1092 of *CEUR Workshop Proceedings*, pages 103–111. CEUR-WS.org, 2013. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-summary-aachen-2013>.
- [15] A. D. Brucker, D. Longuet, F. Tuong, and B. Wolff. On the semantics of object-oriented data structures and path expressions. In J. Cabot, M. Gogolla, I. Ráth, and E. D. Willink, editors, *Proceedings of the MODELS 2013 OCL Workshop (OCL 2013)*, volume 1092 of *CEUR Workshop Proceedings*, pages 23–32. CEUR-WS.org, 2013. URL <http://www.brucker.ch/bibliography/abstract/brucker.ea-path-expressions-2013>. An extended version of this paper is available as LRI Technical Report 1565.
- [16] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.
- [17] T. Clark and J. Warmer, editors. *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *Lecture Notes in Computer Science*, Heidelberg, 2002. Springer-Verlag. ISBN 3-540-43169-1.
- [18] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The amsterdam manifesto on OCL. In Clark and Warmer [17], pages 115–149. ISBN 3-540-43169-1.
- [19] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-78799-0. doi: 10.1007/978-3-540-78800-3_24.
- [20] M. Gogolla and M. Richters. Expressing UML class diagrams properties with OCL. In Clark and Warmer [17], pages 85–114. ISBN 3-540-43169-1.
- [21] F. Haftmann and M. Wenzel. Constructive type classes in isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 160–174. Springer, 2006. ISBN 978-3-540-74463-4. doi: 10.1007/978-3-540-74464-1_11. URL https://doi.org/10.1007/978-3-540-74464-1_11.
- [22] A. Hamie, F. Civello, J. Howse, S. Kent, and R. Mitchell. Reflections on the Object Constraint Language. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language. «UML» '98: Beyond the Notation*, volume 1618 of *Lecture Notes in Computer Science*, pages 162–172, Heidelberg, 1998. Springer-Verlag. ISBN 3-540-66252-9. doi: 10.1007/b72309.
- [23] P. Kosiuczenko. Specification of invariability in OCL. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *Lecture Notes in Computer Science*, pages 676–691, Heidelberg, 2006. Springer-Verlag. ISBN 978-3-540-45772-5. doi: 10.1007/11880240_47.

- [24] M. P. Krieger, A. Knapp, and B. Wolff. Generative programming and component engineering. In E. Visser and J. Järvi, editors, *International Conference on Generative Programming and Component Engineering (GPCE 2010)*, pages 53–62. ACM, Oct. 2010. ISBN 978-1-4503-0154-1.
- [25] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML reference manual (revision 1.2), Feb. 2007. Available from <http://www.jmlspecs.org>.
- [26] L. Mandel and M. V. Cengarle. On the expressive power of OCL. In J. M. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems (FM)*, volume 1708 of *Lecture Notes in Computer Science*, pages 854–874, Heidelberg, 1999. Springer-Verlag. ISBN 3-540-66587-0.
- [27] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2002. doi: 10.1007/3-540-45949-9.
- [28] Object Management Group. Object constraint language specification (version 1.1), Sept. 1997. Available as OMG document ad/97-08-08.
- [29] Object Management Group. UML 2.0 OCL specification, Apr. 2006. Available as OMG document formal/06-05-01.
- [30] Object Management Group. UML 2.4.1: Infrastructure specification, Aug. 2011. Available as OMG document formal/2011-08-05.
- [31] Object Management Group. UML 2.4.1: Superstructure specification, Aug. 2011. Available as OMG document formal/2011-08-06.
- [32] Object Management Group. UML 2.3.1 OCL specification, Feb. 2012. Available as OMG document formal/2012-01-01.
- [33] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.
- [34] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-71208-4. doi: 10.1007/978-3-540-71209-1_49.
- [35] M. Wenzel and B. Wolff. Building formal method tools in the Isabelle/Isar framework. In K. Schneider and J. Brandt, editors, *TPHOLS 2007*, number 4732 in *Lecture Notes in Computer Science*, pages 352–367. Springer-Verlag, Heidelberg, 2007. doi: 10.1007/978-3-540-74591-4_26.
- [36] M. M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, TU München, München, Feb. 2002. URL <http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html>.

Part III.
Appendix

A. The OCL And Featherweight OCL Syntax

Table A.1.: Comparison of different concrete syntax variants for OCL

	OCL	Featherweight OCL	Logical Constant
OclAny	<code>_ = _</code>	$op \triangleq$	<i>UML-Logic.StrongEq</i>
	<code>_ <> _</code>	$op <>$	<i>notequal</i>
	<code>_ ->oclAsSet(_)</code>		
	<code>_ .oclIsNew()</code>	$__ .oclIsNew()$	<i>UML-State.OclIsNew</i>
	<code>not (_ ->oclIsUndefined())</code>	$\delta_ $	<i>UML-Logic.defined</i>
	<code>not (_ ->oclIsInvalid())</code>	$v_ $	<i>UML-Logic.valid</i>
	<code>_ ->oclAsType(_)</code>		
	<code>_ ->oclIsTypeOf(_)</code>		
	<code>_ ->oclIsKindOf(_)</code>		
	<code>_ ->oclIsInState(_)</code>		
	<code>_ ->oclType()</code>		
	<code>_ ->oclLocale()</code>		
OclVoid	<code>_ = _</code>	$op \triangleq$	<i>UML-Logic.StrongEq</i>
	<code>_ <> _</code>	$op <>$	<i>notequal</i>
	<code>_ ->oclAsSet(_)</code>		
	<code>_ .oclIsNew()</code>	$__ .oclIsNew()$	<i>UML-State.OclIsNew</i>
	<code>not (_ ->oclIsUndefined())</code>	$\delta_ $	<i>UML-Logic.defined</i>
	<code>not (_ ->oclIsInvalid())</code>	$v_ $	<i>UML-Logic.valid</i>
	<code>_ ->oclAsType(_)</code>		
	<code>_ ->oclIsTypeOf(_)</code>		
	<code>_ ->oclIsKindOf(_)</code>		
	<code>_ ->oclIsInState(_)</code>		
	<code>_ ->oclType()</code>		
	<code>_ ->oclLocale()</code>		
OclInvalid	<code>_ = _</code>	$op \triangleq$	<i>UML-Logic.StrongEq</i>
	<code>_ <> _</code>	$op <>$	<i>notequal</i>
	<code>_ ->oclAsSet(_)</code>		
	<code>_ .oclIsNew()</code>	$__ .oclIsNew()$	<i>UML-State.OclIsNew</i>
	<code>not (_ ->oclIsUndefined())</code>	$\delta_ $	<i>UML-Logic.defined</i>
	<code>not (_ ->oclIsInvalid())</code>	$v_ $	<i>UML-Logic.valid</i>
	<code>_ ->oclAsType(_)</code>		
	<code>_ ->oclIsTypeOf(_)</code>		
	<code>_ ->oclIsKindOf(_)</code>		
	<code>_ ->oclIsInState(_)</code>		
	<code>_ ->oclType()</code>		
	<code>_ ->oclLocale()</code>		
Real	<code>_ + _</code>	$op +_{real}$	<i>UML-Real.OclAdd_{Real}</i>
	<code>_ - _</code>	$op -_{real}$	<i>UML-Real.OclMinus_{Real}</i>

Continued on next page

	OCL	Featherweight OCL	Logical Constant
	- * -	$op *_{real}$	<i>UML-Real.OclMult_{Real}</i>
	- / -		
	- .abs()		
	- .floor()		
	- .round()		
	- .max()		
	- .min()		
	- < -	$op <_{real}$	<i>UML-Real.OclLess_{Real}</i>
	- > -		
	- <= -	$op \leq_{real}$	<i>UML-Real.OclLe_{Real}</i>
	- >= -		
	- .toString()		
	- .div(_)	$op div_{real}$	<i>UML-Real.OclDivision_{Real}</i>
	- .mod(_)	$op mod_{real}$	<i>UML-Real.OclModulus_{Real}</i>
	- ->oclAsType(Integer)	$_ \rightarrow oclAsType_{Real}(Integer)$	<i>UML-Library.OclAsInteger_{Real}</i>
	- ->oclAsType(Boolean)	$_ \rightarrow oclAsType_{Real}(Boolean)$	<i>UML-Library.OclAsBoolean_{Real}</i>
Real Literals	0.0	0.0	<i>UML-Real.OclReal0</i>
	1.0	1.0	<i>UML-Real.OclReal1</i>
	2.0	2.0	<i>UML-Real.OclReal2</i>
	3.0	3.0	<i>UML-Real.OclReal3</i>
	4.0	4.0	<i>UML-Real.OclReal4</i>
	5.0	5.0	<i>UML-Real.OclReal5</i>
	6.0	6.0	<i>UML-Real.OclReal6</i>
	7.0	7.0	<i>UML-Real.OclReal7</i>
	8.0	8.0	<i>UML-Real.OclReal8</i>
	9.0	9.0	<i>UML-Real.OclReal9</i>
	10.0	10.0	<i>UML-Real.OclReal10</i>
	π	<i>UML-Real.OclRealpi</i>	
Integer	- - -	$op -_{int}$	<i>UML-Integer.OclMinus_{Integer}</i>
	- + -	$op +_{int}$	<i>UML-Integer.OclAdd_{Integer}</i>
	- -		
	- * -	$op *_{int}$	<i>UML-Integer.OclMult_{Integer}</i>
	- / -		
	- .abs()		
	- div (_)	$op div_{int}$	<i>UML-Integer.OclDivision_{Integer}</i>
	- mod (_)	$op mod_{int}$	<i>UML-Integer.OclModulus_{Integer}</i>
	- .max()		
	- .min()		
	- .toString()		
- < -	$op <_{int}$	<i>UML-Integer.OclLess_{Integer}</i>	
- <= -	$op \leq_{int}$	<i>UML-Integer.OclLe_{Integer}</i>	
- ->oclAsType(Real)	$_ \rightarrow oclAsType_{Int}(Real)$	<i>UML-Library.OclAsReal_{Int}</i>	
- ->oclAsType(Boolean)	$_ \rightarrow oclAsType_{Int}(Boolean)$	<i>UML-Library.OclAsBoolean_{Int}</i>	
Integer Literals	0	0	<i>UML-Integer.OclInt0</i>
	1	1	<i>UML-Integer.OclInt1</i>
	2	2	<i>UML-Integer.OclInt2</i>
	3	3	<i>UML-Integer.OclInt3</i>
	4	4	<i>UML-Integer.OclInt4</i>

Continued on next page

	OCL	Featherweight OCL	Logical Constant
	5	5	<i>UML-Integer.OclInt5</i>
	6	6	<i>UML-Integer.OclInt6</i>
	7	7	<i>UML-Integer.OclInt7</i>
	8	8	<i>UML-Integer.OclInt8</i>
	9	9	<i>UML-Integer.OclInt9</i>
	10	10	<i>UML-Integer.OclInt10</i>
String and String Literals	<code>_ + _</code>	<i>op +string</i>	<i>UML-String.OclAddString</i>
	<code>_ .size()</code>		
	<code>_ .concat(_)</code>		
	<code>_ .substring(_ , _)</code>		
	<code>_ .toInteger()</code>		
	<code>_ .toReal()</code>		
	<code>_ .toUpperCase()</code>		
	<code>_ .toLowerCase()</code>		
	<code>_ .indexOf()</code>		
	<code>_ .equalsIgnoreCase(_)</code>		
	<code>_ .at(_)</code>		
	<code>_ .characters()</code>		
	<code>_ .toBoolean()</code>		
	<code>_ < _</code>		
	<code>_ > _</code>		
	<code>_ <= _</code>		
<code>_ >= _</code>			
a	a	<i>UML-String.OclStringa</i>	
b	b	<i>UML-String.OclStringb</i>	
c	c	<i>UML-String.OclStringc</i>	
Boolean and Core Logic	<code>_ or _</code>	<i>op or</i>	<i>UML-Logic.OclOr</i>
	<code>_ xor _</code>		
	<code>_ and _</code>	<i>op and</i>	<i>UML-Logic.OclAnd</i>
	<code>not _</code>	<i>not</i>	<i>UML-Logic.OclNot</i>
	<code>_ implies _</code>	<i>op implies</i>	<i>UML-Logic.OclImplies</i>
	<code>_ .toString()</code>		
	<code>if _ then _ else _ endif</code>	<i>if _ then _ else _ endif</i>	<i>UML-Logic.OclIf</i>
	<code>_ = _</code>	<i>op ≐</i>	<i>UML-Logic.StrictRefEq</i>
	<code>_ <> _</code>	<i>op <></i>	<i>notequal</i>
		<code>__ ≠ __</code>	<i>OclNonValid</i>
	<code>__ ⊨ __</code>	<i>UML-Logic.OclValid</i>	
	<i>op ≐</i>	<i>UML-Logic.StrongEq</i>	
Set and Iterators on Set	<code>Set (_)</code>	<i>Set(type⁰)</i>	<i>UML-Types.Setbase type</i>
	<code>Set{}</code>	<i>Set{}</i>	<i>UML-Set.mtSet</i>
	<code>Set{ _ }</code>	<i>Set{ args⁰ }</i>	<i>OclFinset</i>
	<code>_ ->union(_)</code>	<code>__ ->union_{Set}(__)</code>	<i>UML-Set.OclUnion</i>
	<code>_ = _</code>	<i>op ≐</i>	<i>UML-Logic.StrongEq</i>
	<code>_ ->intersection(_)</code>	<code>__ ->intersection_{Set}(__)</code>	<i>UML-Set.OclIntersection</i>
	<code>_ - _</code>		
	<code>_ ->including(_)</code>	<code>__ ->including_{Set}(__)</code>	<i>UML-Set.OclIncluding</i>
	<code>_ ->excluding(_)</code>	<code>__ ->excluding_{Set}(__)</code>	<i>UML-Set.OclExcluding</i>
	<code>_ ->symmetricDifference(_)</code>		
<code>_ ->count(_)</code>	<code>__ ->count_{Set}(__)</code>	<i>UML-Set.OclCount</i>	

Continued on next page

	OCL	Featherweight OCL	Logical Constant
	<code>_ ->flatten()</code>		
	<code>_ ->selectByKind(_)</code>		
	<code>_ ->selectByType(_)</code>		
	<code>_ ->reject(_ _)</code>	<code>_ ->reject_{Set}(\boxed{id} _)</code>	<i>OclRejectSet</i>
	<code>_ ->select(_ _)</code>	<code>_ ->select_{Set}(\boxed{id} _)</code>	<i>OclSelectSet</i>
	<code>_ ->iterate(_ ; _ = _ _)</code>	<code>_ ->iterate_{Set}(idt^0 ; $idt^0 = any^0$ any^0)</code>	<i>OclIterateSet</i>
	<code>_ ->exists(_ _)</code>	<code>_ ->exists_{Set}(\boxed{id} _)</code>	<i>OclExistSet</i>
	<code>_ ->forAll(_ _)</code>	<code>_ ->forAll_{Set}(\boxed{id} _)</code>	<i>OclForallSet</i>
	<code>_ ->asSequence()</code>	<code>_ ->asSequence_{Set}()</code>	<i>UML-Library.OclAsSeqSet</i>
	<code>_ ->asBag()</code>	<code>_ ->asBag_{Set}()</code>	<i>UML-Library.OclAsBagSet</i>
	<code>_ ->asPair()</code>	<code>_ ->asPair_{Set}()</code>	<i>UML-Library.OclAsPairSet</i>
	<code>_ ->sum()</code>	<code>_ ->sum_{Set}()</code>	<i>UML-Set.OclSum</i>
	<code>_ ->excludesAll(_)</code>	<code>_ ->excludesAll_{Set}(_)</code>	<i>UML-Set.OclExcludesAll</i>
	<code>_ ->includesAll(_)</code>	<code>_ ->includesAll_{Set}(_)</code>	<i>UML-Set.OclIncludesAll</i>
	<code>_ ->any()</code>	<code>_ ->any_{Set}()</code>	<i>UML-Set.OclANY</i>
	<code>_ ->notEmpty()</code>	<code>_ ->notEmpty_{Set}()</code>	<i>UML-Set.OclNotEmpty</i>
	<code>_ ->isEmpty()</code>	<code>_ ->isEmpty_{Set}()</code>	<i>UML-Set.OclIsEmpty</i>
	<code>_ ->size()</code>	<code>_ ->size_{Set}()</code>	<i>UML-Set.OclSize</i>
	<code>_ ->excludes(_)</code>	<code>_ ->excludes_{Set}(_)</code>	<i>UML-Set.OclExcludes</i>
	<code>_ ->includes(_)</code>	<code>_ ->includes_{Set}(_)</code>	<i>UML-Set.OclIncludes</i>
Sequence and Iterators on Sequence	<code>Sequence(_)</code>	<code>Sequence(type⁰)</code>	<i>UML-Types.Sequence_{base} type</i>
	<code>Sequence{}</code>	<code>Sequence{}</code>	<i>UML-Sequence.mtSequence</i>
	<code>Sequence{ _ }</code>	<code>Sequence{ args⁰ }</code>	<i>OclFinsequence</i>
	<code>_ ->any()</code>	<code>_ ->any_{Seq}()</code>	<i>UML-Sequence.OclANY</i>
	<code>_ ->notEmpty()</code>	<code>_ ->notEmpty_{Seq}()</code>	<i>UML-Sequence.OclNotEmpty</i>
	<code>_ ->isEmpty()</code>	<code>_ ->isEmpty_{Seq}()</code>	<i>UML-Sequence.OclIsEmpty</i>
	<code>_ ->size()</code>	<code>_ ->size_{Seq}()</code>	<i>UML-Sequence.OclSize</i>
	<code>_ ->select(_ _)</code>	<code>_ ->select_{Seq}(\boxed{id} _)</code>	<i>OclSelectSeq</i>
	<code>_ ->collect(_ _)</code>	<code>_ ->collect_{Seq}(\boxed{id} _)</code>	<i>OclCollectSeq</i>
	<code>_ ->exists(_ _)</code>	<code>_ ->exists_{Seq}(\boxed{id} _)</code>	<i>OclExistSeq</i>
	<code>_ ->forAll(_ _)</code>	<code>_ ->forAll_{Seq}(\boxed{id} _)</code>	<i>OclForallSeq</i>
	<code>_ ->iterate(_ ; _ : _ = _ _)</code>	<code>_ ->iterate_{Seq}(idt^0 ; $idt^0 = any^0$ any^0)</code>	<i>OclIterateSeq</i>
	<code>_ ->last()</code>	<code>_ ->last_{Seq}(_)</code>	<i>UML-Sequence.OclLast</i>
	<code>_ ->first()</code>	<code>_ ->first_{Seq}(_)</code>	<i>UML-Sequence.OclFirst</i>
	<code>_ ->at(_)</code>	<code>_ ->at_{Seq}(_)</code>	<i>UML-Sequence.OclAt</i>
	<code>_ ->union(_)</code>	<code>_ ->union_{Seq}(_)</code>	<i>UML-Sequence.OclUnion</i>
	<code>_ ->append(_)</code>	<code>_ ->append_{Seq}(_)</code>	<i>UML-Sequence.OclAppend</i>
	<code>_ ->excluding(_)</code>	<code>_ ->excluding_{Seq}(_)</code>	<i>UML-Sequence.OclExcluding</i>
	<code>_ ->including(_)</code>	<code>_ ->including_{Seq}(_)</code>	<i>UML-Sequence.OclIncluding</i>
	<code>_ ->prepend(_)</code>	<code>_ ->prepend_{Seq}(_)</code>	<i>UML-Sequence.OclPrepend</i>
<code>_ ->asSet()</code>	<code>_ ->asSet_{Seq}()</code>	<i>UML-Library.OclAsSetSeq</i>	
<code>_ ->asBag()</code>	<code>_ ->asBag_{Seq}()</code>	<i>UML-Library.OclAsBagSeq</i>	
<code>_ ->asPair()</code>	<code>_ ->asPair_{Seq}()</code>	<i>UML-Library.OclAsPairSeq</i>	
Bag and Iterators on Bag	<code>Bag(_)</code>	<code>Bag(type⁰)</code>	<i>UML-Types.Bag_{base} type</i>
	<code>Bag{}</code>	<code>Bag{}</code>	<i>UML-Bag.mtBag</i>
	<code>Bag{ _ }</code>	<code>Bag{ args⁰ }</code>	<i>OclFinbag</i>
	<code>_ ->sum()</code>	<code>_ ->sum_{Bag}()</code>	<i>UML-Bag.OclSum</i>
	<code>_ ->count(_)</code>	<code>_ ->count_{Bag}(_)</code>	<i>UML-Bag.OclCount</i>
	<code>_ ->intersection(_)</code>	<code>_ ->intersection_{Bag}(_)</code>	<i>UML-Bag.OclIntersection</i>

Continued on next page

	OCL	Featherweight OCL	Logical Constant
	<code>_ ->union(_)</code>	<code>__ ->union_{Bag}(__)</code>	<i>UML-Bag.OclUnion</i>
	<code>_ ->excludesAll(_)</code>	<code>__ ->excludesAll_{Bag}(__)</code>	<i>UML-Bag.OclExcludesAll</i>
	<code>_ ->includesAll(_)</code>	<code>__ ->includesAll_{Bag}(__)</code>	<i>UML-Bag.OclIncludesAll</i>
	<code>_ ->reject(_ _)</code>	<code>__ ->reject_{Bag}(\boxed{id} __)</code>	<i>OclRejectBag</i>
	<code>_ ->select(_ _)</code>	<code>__ ->select_{Bag}(\boxed{id} __)</code>	<i>OclSelectBag</i>
	<code>_ ->iterate(_ ; _ = _ _)</code>	<code>__ ->iterate_{Bag}(idt^0 ; $idt^0 = any^0$ any^0)</code>	<i>OclIterateBag</i>
	<code>_ ->exists(_ _)</code>	<code>__ ->exists_{Bag}(\boxed{id} __)</code>	<i>OclExistBag</i>
	<code>_ ->forall(_ _)</code>	<code>__ ->forall_{Bag}(\boxed{id} __)</code>	<i>OclForallBag</i>
	<code>_ ->any()</code>	<code>__ ->any_{Bag}()</code>	<i>UML-Bag.OclANY</i>
	<code>_ ->notEmpty()</code>	<code>__ ->notEmpty_{Bag}()</code>	<i>UML-Bag.OclNotEmpty</i>
	<code>_ ->isEmpty()</code>	<code>__ ->isEmpty_{Bag}()</code>	<i>UML-Bag.OclIsEmpty</i>
	<code>_ ->size()</code>	<code>__ ->size_{Bag}()</code>	<i>UML-Bag.OclSize</i>
	<code>_ ->excludes(_)</code>	<code>__ ->excludes_{Bag}(__)</code>	<i>UML-Bag.OclExcludes</i>
	<code>_ ->includes(_)</code>	<code>__ ->includes_{Bag}(__)</code>	<i>UML-Bag.OclIncludes</i>
	<code>_ ->excluding(_)</code>	<code>__ ->excluding_{Bag}(__)</code>	<i>UML-Bag.OclExcluding</i>
	<code>_ ->including(_)</code>	<code>__ ->including_{Bag}(__)</code>	<i>UML-Bag.OclIncluding</i>
	<code>_ ->asSet()</code>	<code>__ ->asSet_{Bag}()</code>	<i>UML-Library.OclAsSet_{Bag}</i>
	<code>_ ->asSeq()</code>	<code>__ ->asSeq_{Bag}()</code>	<i>UML-Library.OclAsSeq_{Bag}</i>
	<code>_ ->asPair()</code>	<code>__ ->asPair_{Bag}()</code>	<i>UML-Library.OclAsPair_{Bag}</i>
Pair		<code>Pair($type^0$, $type^0$)</code>	<i>UML-Types.Pair_{base} type</i>
		<code>Pair{ __ , __ }</code>	<i>UML-Pair.OclPair</i>
		<code>__ .Second()</code>	<i>UML-Pair.OclSecond</i>
		<code>__ .First()</code>	<i>UML-Pair.OclFirst</i>
	<code>_ ->asSequence()</code>	<code>__ ->asSequence_{Pair}()</code>	<i>UML-Library.OclAsSeq_{Pair}</i>
	<code>_ ->asSet()</code>	<code>__ ->asSet_{Pair}()</code>	<i>UML-Library.OclAsSet_{Pair}</i>
State Access		<code>__ .allInstances()</code>	<i>UML-State.OclAllInstances-at-post</i>
		<code>__ .allInstances@pre()</code>	<i>UML-State.OclAllInstances-at-pre</i>
		<code>__ .oclIsDeleted()</code>	<i>UML-State.OclIsDeleted</i>
		<code>__ .oclIsMaintained()</code>	<i>UML-State.OclIsMaintained</i>
		<code>__ .oclIsAbsent()</code>	<i>UML-State.OclIsAbsent</i>
		<code>__ ->oclIsModifiedOnly()</code>	<i>UML-State.OclIsModifiedOnly</i>
	<code>_ @pre _</code>	<code>__ @pre __</code>	<i>UML-State.OclSelf-at-pre</i>
		<code>__ @post __</code>	<i>UML-State.OclSelf-at-post</i>