



HAL
open science

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

Achim D. Brucker, Frédéric Tuong, Burkhart Wolff

► **To cite this version:**

Achim D. Brucker, Frédéric Tuong, Burkhart Wolff. Featherweight OCL: A Proposal for a Machine-Checked Formal Semantics for OCL 2.5. 15th International Workshop on OCL and Textual Modeling co-located with 18th International Conference on Model Driven Engineering Languages and Systems, Oct 2015, Karlsruhe, Germany. pp.199. hal-01213440

HAL Id: hal-01213440

<https://inria.hal.science/hal-01213440>

Submitted on 11 Oct 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Featherweight OCL

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

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

October 11, 2015

*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	41
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)	43
1.3.2. The Construction of the Set Type	44
1.3.3. The Construction of the Bag Type	44
1.3.4. The Construction of the Sequence Type	45
1.3.5. Discussion: The Representation of UML/OCL Types in Featherweight OCL	45
2. Formalization II: OCL Terms and Library Operations	47
2.1. The Operations of the Boolean Type and the OCL Logic	47
2.1.1. Basic Constants	47
2.1.2. Validity and Definedness	47

2.1.3.	The Equalities of OCL	49
	Definition	50
	Fundamental Predicates on Strong Equality	50
2.1.4.	Logical Connectives and their Universal Properties	51
2.1.5.	A Standard Logical Calculus for OCL	55
	Global vs. Local Judgements	55
	Local Validity and Meta-logic	56
	Local Judgements and Strong Equality	59
2.1.6.	OCL's if then else endif	60
2.1.7.	Fundamental Predicates on Basic Types: Strict (Referential) Equality	61
2.1.8.	Laws to Establish Definedness (δ -closure)	61
2.1.9.	A Side-calculus for Constant Terms	62
2.2.	Property Profiles for OCL Operators via Isabelle Locales	64
2.2.1.	Property Profiles for Monadic Operators	64
2.2.2.	Property Profiles for Single	65
2.2.3.	Property Profiles for Binary Operators	65
2.2.4.	Fundamental Predicates on Basic Types: Strict (Referential) Equality	68
2.2.5.	Test Statements on Boolean Operations.	69
2.3.	Basic Type Void: Operations	69
2.3.1.	Fundamental Properties on Voids: Strict Equality	70
	Definition	70
2.3.2.	Basic Void Constants	70
2.3.3.	Validity and Definedness Properties	70
2.3.4.	Test Statements	70
2.4.	Basic Type Integer: Operations	71
2.4.1.	Fundamental Predicates on Integers: Strict Equality	71
2.4.2.	Basic Integer Constants	71
2.4.3.	Validity and Definedness Properties	71
2.4.4.	Arithmetical Operations	72
	Definition	72
	Basic Properties	73
	Execution with Invalid or Null or Zero as Argument	73
2.4.5.	Test Statements	73
2.5.	Basic Type Real: Operations	74
2.5.1.	Fundamental Predicates on Reals: Strict Equality	74
2.5.2.	Basic Real Constants	75
2.5.3.	Validity and Definedness Properties	75
2.5.4.	Arithmetical Operations	75
	Definition	75
	Basic Properties	76
	Execution with Invalid or Null or Zero as Argument	77
2.5.5.	Test Statements	77
2.6.	Basic Type String: Operations	78
2.6.1.	Fundamental Properties on Strings: Strict Equality	78
2.6.2.	Basic String Constants	78
2.6.3.	Validity and Definedness Properties	78
2.6.4.	String Operations	79
	Definition	79
	Basic Properties	79
2.6.5.	Test Statements	79
2.7.	Collection Type Pairs: Operations	80
2.7.1.	Semantic Properties of the Type Constructor	80
2.7.2.	Fundamental Properties of Strict Equality	80

2.7.3.	Standard Operations Definitions	80
	Definition: Pair Constructor	81
	Definition: First	81
	Definition: Second	81
2.7.4.	Logical Properties	81
2.7.5.	Algebraic Execution Properties	81
2.7.6.	Test Statements	81
2.8.	Collection Type Bag: Operations	82
2.8.1.	As a Motivation for the (infinite) Type Construction: Type-Extensions as Bags	82
2.8.2.	Basic Properties of the Bag Type	84
2.8.3.	Definition: Strict Equality	84
2.8.4.	Constants: mtBag	85
2.8.5.	Definition: Including	85
2.8.6.	Definition: Excluding	85
2.8.7.	Definition: Includes	86
2.8.8.	Definition: Excludes	86
2.8.9.	Definition: Size	86
2.8.10.	Definition: IsEmpty	86
2.8.11.	Definition: NotEmpty	86
2.8.12.	Definition: Any	86
2.8.13.	Definition: Forall	87
2.8.14.	Definition: Exists	87
2.8.15.	Definition: Iterate	87
2.8.16.	Definition: Select	87
2.8.17.	Definition: Reject	88
2.8.18.	Definition: IncludesAll	88
2.8.19.	Definition: ExcludesAll	88
2.8.20.	Definition: Union	88
2.8.21.	Definition: Intersection	88
2.8.22.	Definition: Count	89
2.8.23.	Definition (future operators)	89
2.8.24.	Test Statements	89
2.9.	Collection Type Set: Operations	90
2.9.1.	As a Motivation for the (infinite) Type Construction: Type-Extensions as Sets	90
2.9.2.	Basic Properties of the Set Type	91
2.9.3.	Definition: Strict Equality	92
2.9.4.	Constants: mtSet	92
2.9.5.	Definition: Including	93
2.9.6.	Definition: Excluding	93
2.9.7.	Definition: Includes	93
2.9.8.	Definition: Excludes	93
2.9.9.	Definition: Size	94
2.9.10.	Definition: IsEmpty	94
2.9.11.	Definition: NotEmpty	94
2.9.12.	Definition: Any	94
2.9.13.	Definition: Forall	94
2.9.14.	Definition: Exists	94
2.9.15.	Definition: Iterate	95
2.9.16.	Definition: Select	95
2.9.17.	Definition: Reject	95
2.9.18.	Definition: IncludesAll	95
2.9.19.	Definition: ExcludesAll	95
2.9.20.	Definition: Union	96

2.9.21.	Definition: Intersection	96
2.9.22.	Definition (future operators)	96
2.9.23.	Logical Properties	96
2.9.24.	Execution Laws with Invalid or Null or Infinite Set as Argument	98
	Context Passing	100
	Const	101
2.9.25.	General Algebraic Execution Rules	101
	Execution Rules on Including	101
	Execution Rules on Excluding	102
	Execution Rules on Includes	104
	Execution Rules on Excludes	105
	Execution Rules on Size	105
	Execution Rules on IsEmpty	105
	Execution Rules on NotEmpty	105
	Execution Rules on Any	106
	Execution Rules on Forall	106
	Execution Rules on Exists	106
	Execution Rules on Iterate	106
	Execution Rules on Select	107
	Execution Rules on Reject	107
	Execution Rules Combining Previous Operators	107
2.9.26.	Test Statements	109
2.10.	Collection Type Sequence: Operations	110
2.10.1.	Basic Properties of the Sequence Type	110
2.10.2.	Definition: Strict Equality	110
2.10.3.	Constants: mtSequence	111
2.10.4.	Definition: Prepend	111
2.10.5.	Definition: Including	111
2.10.6.	Definition: Excluding	112
2.10.7.	Definition: Append	112
2.10.8.	Definition: Union	112
2.10.9.	Definition: At	112
2.10.10.	Definition: First	112
2.10.11.	Definition: Last	113
2.10.12.	Definition: Iterate	113
2.10.13.	Definition: Forall	113
2.10.14.	Definition: Exists	113
2.10.15.	Definition: Collect	113
2.10.16.	Definition: Select	114
2.10.17.	Definition: Size	114
2.10.18.	Definition: IsEmpty	114
2.10.19.	Definition: NotEmpty	114
2.10.20.	Definition: Any	114
2.10.21.	Definition (future operators)	114
2.10.22.	Logical Properties	114
2.10.23.	Execution Laws with Invalid or Null as Argument	114
	Context Passing	115
	Const	115
2.10.24.	General Algebraic Execution Rules	115
	Execution Rules on Iterate	115
2.10.25.	Test Statements	115
2.11.	Miscellaneous Stuff	116
2.11.1.	Definition: asBoolean	116

2.11.2.	Definition: asInteger	116
2.11.3.	Definition: asReal	117
2.11.4.	Definition: asPair	117
2.11.5.	Definition: asSet	117
2.11.6.	Definition: asSequence	118
2.11.7.	Definition: asBag	118
2.11.8.	Collection Types	118
2.11.9.	Test Statements	118
3.	Formalization III: UML/OCL constructs: State Operations and Objects	121
3.1.	Introduction: States over Typed Object Universes	121
3.1.1.	Fundamental Properties on Objects: Core Referential Equality	121
	Definition	121
	Strictness and context passing	121
3.1.2.	Logic and Algebraic Layer on Object	122
	Validity and Definedness Properties	122
	Symmetry	122
	Behavior vs StrongEq	122
3.2.	Operations on Object	123
3.2.1.	Initial States (for testing and code generation)	123
3.2.2.	OclAllInstances	123
	OclAllInstances (@post)	125
	OclAllInstances (@pre)	127
	@post or @pre	128
3.2.3.	OclIsNew, OclIsDeleted, OclIsMaintained, OclIsAbsent	128
3.2.4.	OclIsModifiedOnly	129
	Definition	129
	Execution with Invalid or Null or Null Element as Argument	129
	Context Passing	130
3.2.5.	OclSelf	130
3.2.6.	Framing Theorem	130
3.2.7.	Miscellaneous	131
3.3.	Accessors on Object	131
3.3.1.	Definition	131
3.3.2.	Validity and Definedness Properties	131
4.	Example: The Employee Analysis Model	141
4.1.	Introduction	141
4.1.1.	Outlining the Example	141
4.2.	Example Data-Universe and its Infrastructure	142
4.3.	Instantiation of the Generic Strict Equality	143
4.4.	OclAsType	143
4.4.1.	Definition	143
4.4.2.	Context Passing	144
4.4.3.	Execution with Invalid or Null as Argument	144
4.5.	OclIsTypeOf	145
4.5.1.	Definition	145
4.5.2.	Context Passing	146
4.5.3.	Execution with Invalid or Null as Argument	146
4.5.4.	Up Down Casting	147
4.6.	OclIsKindOf	147
4.6.1.	Definition	147
4.6.2.	Context Passing	148

4.6.3.	Execution with Invalid or Null as Argument	148
4.6.4.	Up Down Casting	149
4.7.	OclAllInstances	149
4.7.1.	OclIsTypeOf	149
4.7.2.	OclIsKindOf	150
4.8.	The Accessors (any, boss, salary)	151
4.8.1.	Definition (of the association Employee-Boss)	151
4.8.2.	Context Passing	153
4.8.3.	Execution with Invalid or Null as Argument	154
4.8.4.	Representation in States	154
4.9.	A Little Infra-structure on Example States	155
4.10.	OCL Part: Invariant	159
4.11.	OCL Part: The Contract of a Recursive Query	160
4.12.	OCL Part: The Contract of a User-defined Method	161
5.	Example: The Employee Design Model	163
5.1.	Introduction	163
5.1.1.	Outlining the Example	163
5.2.	Example Data-Universe and its Infrastructure	163
5.3.	Instantiation of the Generic Strict Equality	164
5.4.	OclAsType	165
5.4.1.	Definition	165
5.4.2.	Context Passing	166
5.4.3.	Execution with Invalid or Null as Argument	166
5.5.	OclIsTypeOf	167
5.5.1.	Definition	167
5.5.2.	Context Passing	167
5.5.3.	Execution with Invalid or Null as Argument	168
5.5.4.	Up Down Casting	169
5.6.	OclIsKindOf	169
5.6.1.	Definition	169
5.6.2.	Context Passing	170
5.6.3.	Execution with Invalid or Null as Argument	170
5.6.4.	Up Down Casting	171
5.7.	OclAllInstances	171
5.7.1.	OclIsTypeOf	171
5.7.2.	OclIsKindOf	172
5.8.	The Accessors (any, boss, salary)	173
5.8.1.	Definition	173
5.8.2.	Context Passing	174
5.8.3.	Execution with Invalid or Null as Argument	175
5.8.4.	Representation in States	176
5.9.	A Little Infra-structure on Example States	176
5.10.	OCL Part: Invariant	180
5.11.	OCL Part: The Contract of a Recursive Query	182
II.	Conclusion	183
6.	Conclusion	185
6.1.	Lessons Learned and Contributions	185
6.2.	Lessons Learned	186
6.3.	Conclusion and Future Work	186

III. Appendix

193

A. The OCL And Featherweight OCL Syntax

195

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.,

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

(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]

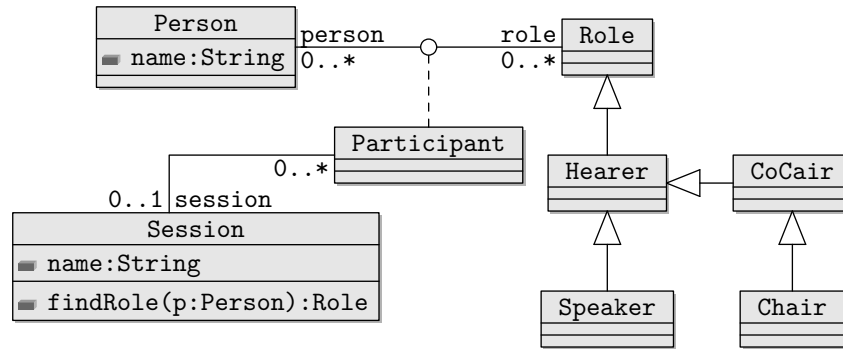


Figure 0.1.: A simple UML class model representing a conference system for organizing conference sessions: persons can participate, in different roles, in a session.

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 0.1 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 $()$ bool, $()$ nat and $()$ 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 $'\alpha \Rightarrow '\alpha \Rightarrow \text{bool}$, 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:

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

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:

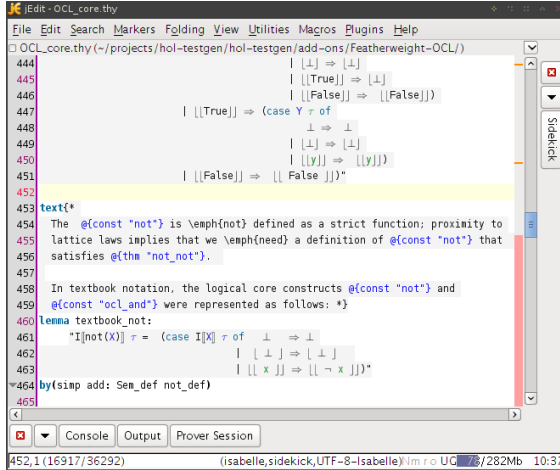
$$\begin{aligned} \text{fun } \text{ins} &:: [\alpha :: \text{linorder}, \alpha \text{ list}] \Rightarrow \alpha \text{ list} \\ \text{where } \text{ins } x \, [] &= [x] \\ \text{ins } x \, (y\#ys) &= \text{if } x < y \text{ then } x\#y\#ys \text{ else } y\#(\text{ins } x \, ys) \end{aligned} \quad (0.12)$$

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

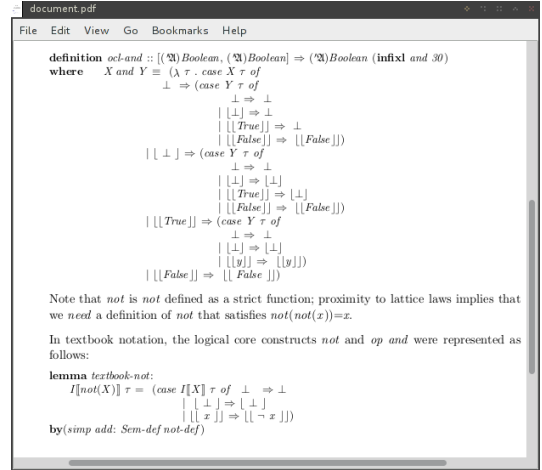
The internal (non-recursive) constant definition for these operations is quite involved; however, the logical compiler will finally derive all the equations in the statements above from this definition and make them available for automated simplification.

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 0.2.: 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., $\text{not (not } x) = x$.

Figure 0.2 illustrates this approach: Figure 0.2a shows the jEdit-based development environment of Isabelle with an excerpt of one of the core theories of Featherweight OCL. Figure 0.2b 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:

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

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:

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

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:

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

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

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:

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

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:

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

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 $x \ll y$ 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:

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

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 $\tau \models x > 0 \text{ or } 3 * y > x * x$ into the equivalent formula $\tau \models x > 0 \vee \tau \models 3 * y > x * x$ and thus internalize the OCL-logic into a classical (and more tool-conform) logic. This works—for the price of a potential, but due to the usually “rich” δ -closures of invariants rare—exponential blow-up of the formula for all OCL formulas.

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: $C_i := (oid \times A_{i_1} \times \dots \times A_{i_k})$ where the types A_{i_1}, \dots, A_{i_k} are the attribute types (including inherited attributes) with class types substituted by oid . The function OidOf projects the first component, the oid , out of an object representation. Then the object universe will be constructed by the type definition:

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

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

Example instances of this scheme—outlining a compiler—can be found in Chapter 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

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

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

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

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:

$$\text{definition select}_a f = (\lambda \text{ mk}_C \text{ oid } \dots \perp \dots \text{ C}_{\text{ext}} \Rightarrow \text{null} \quad | \quad \text{mk}_C \text{ oid } \dots \perp a \dots \text{ C}_{\text{ext}} \Rightarrow f(\lambda x _ \perp x _) a) \quad (0.21)$$

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 Section 0.3.6. If `n ≤ 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

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

where \mathbf{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 \mathbf{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:

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

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:

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

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 Transcendental
keywords Assert :: thy-decl
and Assert-local :: thy-decl
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:

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

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

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

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

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

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

```
definition Sem :: 'a ⇒ 'a (I $\llbracket \_ \rrbracket$ )
where I $\llbracket x \rrbracket$  ≡ 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 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 :  $\exists x. x \neq bot$ 
```

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

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: (bot::'a option)  $\equiv$  (None::'a option)
  instance <proof>
end
```

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

```
instantiation fun :: (type,bot) bot
begin
  definition bot-fun-def: bot  $\equiv$  ( $\lambda x. bot$ )
  instance <proof>
end
```

```
instantiation fun :: (type,null) null
begin
  definition null-fun-def: (null::'a  $\Rightarrow$  'b:null)  $\equiv$  ( $\lambda x. null$ )
  instance <proof>
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 $(\mathcal{A})state =$
 $heap :: oid \rightarrow \mathcal{A}$
 $assocs :: oid \rightarrow ((oid\ list)\ list)\ list$

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

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

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

class *object* = **fixes** $oid-of :: 'a \Rightarrow oid$

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

typ $\mathcal{A} :: object$

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

instantiation $option :: (object)object$
begin
definition *oid-of-option-def*: $oid-of\ x = oid-of\ (the\ x)$
instance $\langle proof \rangle$
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 \mathfrak{A}) to an arbitrary null-type (i. e., containing at least a distinguished *null* and *invalid* element).

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

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

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

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

lemma *textbook-invalid*: $I[\text{invalid}]\tau = \text{bot}$
<proof>

Note that the definition :

definition *null* :: $(\mathfrak{A}, \alpha::\text{null}) \text{ val}$
where $\text{null} \equiv \lambda \tau. \text{null}$

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

lemma *textbook-null-fun*: $I[\text{null}::(\mathfrak{A}, \alpha::\text{null}) \text{ val}] \tau = (\text{null}::(\alpha::\text{null}))$
<proof>

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 $\langle\langle \text{bool} \rangle_{\perp}\rangle_{\perp}$, i. e. the Boolean base type:

type-synonym $\text{Boolean}_{\text{base}} = \text{bool option option}$
type-synonym $(\mathfrak{A})\text{Boolean} = (\mathfrak{A}, \text{Boolean}_{\text{base}}) \text{ val}$

Because of the previous class definitions, Isabelle type-inference establishes that $\mathfrak{A} \text{ Boolean}$ 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 $\text{Integer}_{\text{base}} = \text{int option option}$
type-synonym $(\mathfrak{A})\text{Integer} = (\mathfrak{A}, \text{Integer}_{\text{base}}) \text{ val}$

type-synonym $\text{String}_{\text{base}} = \text{string option option}$
type-synonym $(\mathfrak{A})\text{String} = (\mathfrak{A}, \text{String}_{\text{base}}) \text{ val}$

type-synonym $\text{Real}_{\text{base}} = \text{real option option}$
type-synonym $(\mathfrak{A})\text{Real} = (\mathfrak{A}, \text{Real}_{\text{base}}) \text{ val}$

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 Voidbase = {X::unit option option. X = bot  $\vee$  X = null } <proof>
```

```
type-synonym ('A) Void = ('A, Voidbase) 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 ' α *Set* just by (' \mathcal{A} , (' α *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 ('α, 'β) Pairbase = {X::('α::null  $\times$  'β::null) option option.  

                                    X = bot  $\vee$  X = null  $\vee$  (fst $\top$ X $\top$   $\neq$  bot  $\wedge$  snd $\top$ X $\top$   $\neq$  bot)}
```

<proof>

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 $\llcorner(\perp, a)\llcorner$ or $\llcorner(b, \perp)\llcorner$. The type constructor *Pair*{*x*,*y*} to be defined later will identify these with *invalid*.

```
instantiation Pairbase :: (null,null)bot
```

```
begin
```

```
  definition bot-Pairbase-def: (bot-class.bot :: ('a::null, 'b::null) Pairbase)  $\equiv$  Abs-Pairbase None
```

```
  instance <proof>
```

```
end
```

```
instantiation Pairbase :: (null,null)null
```

```
begin
```

```
  definition null-Pairbase-def: (null::('a::null, 'b::null) Pairbase)  $\equiv$  Abs-Pairbase  $\llcorner$  None  $\llcorner$ 
```

```

instance ⟨proof⟩
end

```

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

```

type-synonym ('ℳ, 'α, 'β) Pair = ('ℳ, ('α, 'β) Pairbase) val
type-notation Pairbase (Pair'(-, '-))

```

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 'α Setbase = {X :: ('α :: null) set option option. X = bot ∨ X = null ∨ (∀ x ∈ ⊔X⊔. x ≠ bot)}
⟨proof⟩

```

```

instantiation Setbase :: (null)bot
begin

```

```

definition bot-Setbase-def: (bot :: ('a :: null) Setbase) ≡ Abs-Setbase None

```

```

instance ⟨proof⟩
end

```

```

instantiation Setbase :: (null)null
begin

```

```

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

```

```

instance ⟨proof⟩
end

```

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

```

type-synonym ('ℳ, 'α) Set = ('ℳ, 'α Setbase) val
type-notation Setbase (Set'(-))

```

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 'α Bagbase = {X :: ('α :: null ⇒ nat) option option. X = bot ∨ X = null ∨ ⊔X⊔ bot = 0 }
⟨proof⟩

```

```

instantiation Bagbase :: (null)bot
begin

```

```

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

```

```
instance ⟨proof⟩
end
```

```
instantiation Bagbase :: (null)null
begin
```

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

```
instance ⟨proof⟩
end
```

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

```
type-synonym ('A, 'α) Bag = ('A, 'α 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 $'\alpha$ $Sequence_{base}$. It is shown that this type “fits” indeed into the abstract type interface discussed in the previous section.

```
typedef 'α Sequencebase = {X::('α::null) list option option.
                        X = bot ∨ X = null ∨ (∀ x ∈ set ⌈X⌉. x ≠ bot)}
⟨proof⟩
```

```
instantiation Sequencebase :: (null)bot
begin
```

```
definition bot-Sequencebase-def: (bot::('a::null) Sequencebase) ≡ Abs-Sequencebase None
```

```
instance ⟨proof⟩
end
```

```
instantiation Sequencebase :: (null)null
begin
```

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

```
instance ⟨proof⟩
end
```

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

```
type-synonym ('A, 'α) Sequence = ('A, 'α Sequencebase) val
type-notation Sequencebase (Sequence'(-'))
```

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 resenation 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:

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

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

Table 1.1.: Correspondance between OCL types and HOL types

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 -> T' were mapped to functions in HOL, where T and T' were mapped to the valuations for them, and
4. the arguments of type constructors Set(T) remain corresponding HOL base-types.

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

$\langle ML \rangle$

end

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) = (λ τ. ⊥)
⟨proof⟩
```

```
lemma null-Boolean-def : (null::('A)Boolean) = (λ τ. ⊥⊥)
⟨proof⟩
```

```
definition true :: ('A)Boolean
where true ≡ λ τ. ⊥True⊥
```

```
definition false :: ('A)Boolean
where false ≡ λ τ. ⊥False⊥
```

```
lemma bool-split-0: X τ = invalid τ ∨ X τ = null τ ∨
X τ = true τ ∨ X τ = false τ
⟨proof⟩
```

```
lemma [simp]: false (a, b) = ⊥False⊥
⟨proof⟩
```

```
lemma [simp]: true (a, b) = ⊥True⊥
⟨proof⟩
```

```
lemma textbook-true: I[[true]] τ = ⊥True⊥
⟨proof⟩
```

```
lemma textbook-false: I[[false]] τ = ⊥False⊥
⟨proof⟩
```

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:

```
definition valid :: ('A, 'a::null)val ⇒ ('A)Boolean (v - [100]100)
where v X ≡ λ τ. if X τ = bot τ then false τ else true τ
```


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

Table 2.1.: Basic semantic constant definitions of the logic

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

<proof>

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

<proof>

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

<proof>

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

<proof> **lemma** *cp-valid*: $(v\ X) \tau = (v\ (\lambda \cdot X\ \tau)) \tau$

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

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

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

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

<proof>

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

<proof>

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

<proof>

lemma *defined4[simp]*: $\delta\ false = true$

<proof>

lemma *defined5[simp]*: $\delta\ \delta\ X = true$

<proof>

lemma *defined6[simp]*: $\delta\ v\ X = true$

<proof>

lemma *valid5[simp]*: $v\ v\ X = true$

<proof>

lemma *valid6[simp]*: $v\ \delta\ X = true$

<proof> **lemma** *cp-defined*: $(\delta\ X) \tau = (\delta\ (\lambda \cdot X\ \tau)) \tau$

<proof>

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

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

<proof>

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

<proof>

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:

$$s = t \implies P(s) = P(t) \quad (2.1)$$

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

From this follow already elementary properties like:

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

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

Fundamental Predicates on Strong Equality

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

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

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

lemma *StrongEq-trans-strong* [*simp*]:
assumes $A: (X \triangleq Y) = true$
and $B: (Y \triangleq Z) = true$
shows $(X \triangleq Z) = true$
 $\langle proof \rangle$

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

lemma *StrongEq-subst* :
assumes $cp: \bigwedge X. P(X)\tau = P(\lambda -. X \tau)\tau$
and $eq: (X \triangleq Y)\tau = true \tau$
shows $(P X \triangleq P Y)\tau = true \tau$
 $\langle proof \rangle$

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

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

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

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* :: $(\mathfrak{A})Boolean \Rightarrow (\mathfrak{A})Boolean (not)$

where $not X \equiv \lambda \tau . case X \tau of$
 $\quad \perp \quad \Rightarrow \perp$
 $\quad | \perp \perp \perp \quad \Rightarrow \perp \perp \perp$
 $\quad | \perp x \perp \quad \Rightarrow \perp \neg x \perp$

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

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

lemma *OclNot2[simp]*: $not null = null$
 $\langle proof \rangle$

lemma *OclNot3[simp]*: $not true = false$
 $\langle proof \rangle$

lemma *OclNot4[simp]*: $not false = true$
 $\langle proof \rangle$

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

lemma *OclNot-inject*: $\bigwedge x y. not x = not y \implies x = y$

<proof>

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

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

$$\begin{aligned} & \quad \underline{\perp} \Rightarrow \underline{\text{False}}_{\perp} \\ & \quad | \underline{\perp} \Rightarrow (\text{case } Y \tau \text{ of} \\ & \quad \quad \underline{\text{False}}_{\perp} \Rightarrow \underline{\text{False}}_{\perp} \\ & \quad \quad | - \Rightarrow \underline{\perp}) \\ & \quad | \underline{\perp}_{\perp} \Rightarrow (\text{case } Y \tau \text{ of} \\ & \quad \quad \underline{\text{False}}_{\perp} \Rightarrow \underline{\text{False}}_{\perp} \\ & \quad \quad | \underline{\perp} \Rightarrow \underline{\perp} \\ & \quad \quad | - \Rightarrow \underline{\perp}_{\perp}) \\ & \quad | \underline{\text{True}}_{\perp} \Rightarrow Y \tau \end{aligned}$$

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

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

lemma *textbook-OclNot*:

$$\begin{aligned} I[\text{not}(X)] \tau = (\text{case } I[X] \tau \text{ of } & \underline{\perp} \Rightarrow \underline{\perp} \\ & | \underline{\perp}_{\perp} \Rightarrow \underline{\perp}_{\perp} \\ & | \underline{x}_{\perp} \Rightarrow \underline{\neg x}_{\perp}) \end{aligned}$$

<proof>

lemma *textbook-OclAnd*:

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

<proof>

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

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

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

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

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

<proof>

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

<proof>

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

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

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

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

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

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

lemma *OclAnd6[simp]*: $(\text{null and false}) = \text{false}$
<proof>

lemma *OclAnd7[simp]*: $(\text{null and null}) = \text{null}$
<proof>

lemma *OclAnd8[simp]*: $(\text{null and invalid}) = \text{invalid}$
<proof>

lemma *OclAnd9[simp]*: $(\text{false and true}) = \text{false}$
<proof>

lemma *OclAnd10[simp]*: $(\text{false and false}) = \text{false}$
<proof>

lemma *OclAnd11[simp]*: $(\text{false and null}) = \text{false}$
<proof>

lemma *OclAnd12[simp]*: $(\text{false and invalid}) = \text{false}$
<proof>

lemma *OclAnd13[simp]*: $(\text{true and true}) = \text{true}$
<proof>

lemma *OclAnd14[simp]*: $(\text{true and false}) = \text{false}$
<proof>

lemma *OclAnd15[simp]*: $(\text{true and null}) = \text{null}$
<proof>

lemma *OclAnd16[simp]*: $(\text{true and invalid}) = \text{invalid}$
<proof>

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

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

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

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

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

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

<proof>

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *OclImplies-true1*[simp]: $(\text{true implies } X) = X$
 ⟨proof⟩

lemma *OclImplies-true2*[simp]: $(X \text{ implies true}) = \text{true}$
 ⟨proof⟩

lemma *OclImplies-false1*[simp]: $(\text{false implies } X) = \text{true}$
 ⟨proof⟩

2.1.5. A Standard Logical Calculus for OCL

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

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

syntax *OclNonValid* :: $[('a)st, ('a)Boolean] \Rightarrow \text{bool} ((1(-)/ \not\models (-)) 50)$

translations $\tau \not\models P \equiv \neg(\tau \models P)$

Global vs. Local Judgements

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

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

lemma *transform2*: $(P = Q) \implies ((\tau \models P) = (\tau \models Q))$
 ⟨proof⟩

lemma *transform2-rev*: $\forall \tau. (\tau \models \delta P) \wedge (\tau \models \delta Q) \wedge (\tau \models P) = (\tau \models Q) \implies P = Q$
 ⟨proof⟩

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

lemma

assumes $H : P = \text{true} \implies Q = \text{true}$

shows $\tau \models P \implies \tau \models Q$

⟨proof⟩

Local Validity and Meta-logic

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

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

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

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

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

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

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

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

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

lemma *foundation6*:
 $\tau \models P \implies \tau \models \delta P$
<proof>

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

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

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

lemma *foundation8*:
 $(\tau \models \delta x) \vee (\tau \models (x \triangleq \text{invalid})) \vee (\tau \models (x \triangleq \text{null}))$
<proof>

lemma *foundation9*:
 $\tau \models \delta x \implies (\tau \models \text{not } x) = (\neg(\tau \models x))$

<proof>

lemma foundation9':

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

<proof>

lemma foundation9'':

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

<proof>

lemma foundation10:

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

<proof>

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

<proof>

lemma foundation11:

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

<proof>

lemma foundation12:

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

lemma *foundation18''*: $(\tau \models (v X)) = (\neg(\tau \models (X \triangleq \text{invalid})))$
<proof>

lemma *foundation20*: $\tau \models (\delta X) \implies \tau \models v X$
<proof>

lemma *foundation21*: $(\text{not } A \triangleq \text{not } B) = (A \triangleq B)$
<proof>

lemma *foundation22*: $(\tau \models (X \triangleq Y)) = (X \tau = Y \tau)$
<proof>

lemma *foundation23*: $(\tau \models P) = (\tau \models (\lambda \cdot . P \tau))$
<proof>

lemma *foundation24*: $(\tau \models \text{not}(X \triangleq Y)) = (X \tau \neq Y \tau)$
<proof>

lemma *foundation25*: $\tau \models P \implies \tau \models (P \text{ or } Q)$
<proof>

lemma *foundation25'*: $\tau \models Q \implies \tau \models (P \text{ or } Q)$
<proof>

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*
<proof>

lemma *foundation27*: $\tau \models A \implies (\tau \models A \text{ implies } B) = (\tau \models B)$
<proof>

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

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

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

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

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

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

Local Judgements and Strong Equality

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

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

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

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* :: $((\mathcal{A}, \alpha) \text{ val} \Rightarrow (\mathcal{A}, \beta) \text{ val}) \Rightarrow \text{bool}$
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*: $\bigwedge \tau. cp\ P \implies \tau \models (x \triangleq y) \implies \tau \models (P\ x \triangleq P\ y)$
 ⟨proof⟩

lemma *StrongEq-L-subst2*:
 $\bigwedge \tau. cp\ P \implies \tau \models (x \triangleq y) \implies \tau \models (P\ x) \implies \tau \models (P\ y)$
 ⟨proof⟩

lemma *StrongEq-L-subst2-rev*: $\tau \models y \triangleq x \implies cp\ P \implies \tau \models P\ x \implies \tau \models P\ y$
 ⟨proof⟩

lemma *StrongEq-L-subst3*:
assumes *cp*: $cp\ P$
and *eq*: $\tau \models (x \triangleq y)$
shows $(\tau \models P\ x) = (\tau \models P\ y)$
 ⟨proof⟩

lemma *StrongEq-L-subst3-rev*:
assumes *eq*: $\tau \models (x \triangleq y)$
and *cp*: $cp\ P$
shows $(\tau \models P\ x) = (\tau \models P\ y)$
 ⟨proof⟩

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]*
 ⟨proof⟩

lemma *cpI1*:
 $(\forall X \tau. f\ X\ \tau = f(\lambda-. X\ \tau)\ \tau) \implies cp\ P \implies cp(\lambda X. f\ (P\ X))$
 ⟨proof⟩

lemma *cpI2*:
 $(\forall X\ Y\ \tau. f\ X\ Y\ \tau = f(\lambda-. X\ \tau)(\lambda-. Y\ \tau)\ \tau) \implies$
 $cp\ P \implies cp\ Q \implies cp(\lambda X. f\ (P\ X)\ (Q\ X))$
 ⟨proof⟩

lemma *cpI3*:

$(\forall X Y Z \tau. f X Y Z \tau = f(\lambda-. X \tau)(\lambda-. Y \tau)(\lambda-. Z \tau) \tau) \implies$
 $cp P \implies cp Q \implies cp R \implies cp(\lambda X. f (P X) (Q X) (R X))$
 ⟨proof⟩

lemma *cpI4*:

$(\forall W X Y Z \tau. f W X Y Z \tau = f(\lambda-. W \tau)(\lambda-. X \tau)(\lambda-. Y \tau)(\lambda-. Z \tau) \tau) \implies$
 $cp P \implies cp Q \implies cp R \implies cp S \implies cp(\lambda X. f (P X) (Q X) (R X) (S X))$
 ⟨proof⟩

lemma *cpI5*:

$(\forall V W X Y Z \tau. f V W X Y Z \tau = f(\lambda-. V \tau) (\lambda-. W \tau)(\lambda-. X \tau)(\lambda-. Y \tau)(\lambda-. Z \tau) \tau) \implies$
 $cp N \implies cp P \implies cp Q \implies cp R \implies cp S \implies cp(\lambda X. f (N X) (P X) (Q X) (R X) (S X))$
 ⟨proof⟩

lemma *cp-const* : $cp(\lambda-. c)$

⟨proof⟩

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

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

2.1.6. OCL's if then else endif

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

where $(if C then B_1 else B_2 endif) = (\lambda \tau. if (\delta C) \tau = true \tau$
 $then (if (C \tau) = true \tau$
 $then B_1 \tau$
 $else B_2 \tau)$
 $else invalid \tau)$

lemma *cp-OclIf*: $((if C then B_1 else B_2 endif) \tau =$
 $(if (\lambda -. C \tau) then (\lambda -. B_1 \tau) else (\lambda -. B_2 \tau) endif) \tau)$

⟨proof⟩**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*]: $(if invalid then B_1 else B_2 endif) = invalid$

⟨proof⟩

lemma *OclIf-null* [*simp*]: $(if null then B_1 else B_2 endif) = invalid$

⟨proof⟩

lemma *OclIf-true* [*simp*]: $(if true then B_1 else B_2 endif) = B_1$

⟨proof⟩

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

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

lemma *OclIf-false* [simp]: $(\text{if false then } B_1 \text{ else } B_2 \text{ endif}) = B_2$
 ⟨proof⟩

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

lemma *OclIf-idem1*[simp]: $(\text{if } \delta X \text{ then } A \text{ else } A \text{ endif}) = A$
 ⟨proof⟩

lemma *OclIf-idem2*[simp]: $(\text{if } v X \text{ then } A \text{ else } A \text{ endif}) = A$
 ⟨proof⟩

lemma *OclNot-if*[simp]:
 $\text{not}(\text{if } P \text{ then } C \text{ else } E \text{ endif}) = (\text{if } P \text{ then not } C \text{ else not } E \text{ endif})$
 ⟨proof⟩

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

with term "not" we can express the notation:

syntax
 $\text{notequal} \quad :: ('A)\text{Boolean} \Rightarrow ('A)\text{Boolean} \Rightarrow ('A)\text{Boolean}$ (**infix** $\langle \rangle$ 40)

translations
 $a \langle \rangle b == \text{CONST } \text{OclNot}(a \doteq b)$

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 $\tau \models P \implies \tau \models \delta P$ — the following facts:

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

lemma *OclNot-contrapos-nn*:
assumes $A: \tau \models \delta A$
assumes $B: \tau \models \text{not } B$
assumes $C: \tau \models A \implies \tau \models B$
shows $\tau \models \text{not } A$
 ⟨proof⟩

2.1.9. A Side-calculus for Constant Terms

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

lemma *const-charn*: $const\ X \implies X\ \tau = X\ \tau'$
<proof>

lemma *const-subst*:

assumes *const-X*: $const\ X$
and *const-Y*: $const\ Y$
and *eq*: $X\ \tau = Y\ \tau$
and *cp-P*: $cp\ P$
and *pp*: $P\ Y\ \tau = P\ Y\ \tau'$
shows $P\ X\ \tau = P\ X\ \tau'$
<proof>

lemma *const-imply2* :

assumes $\bigwedge \tau \tau'. P\ \tau = P\ \tau' \implies Q\ \tau = Q\ \tau'$
shows $const\ P \implies const\ Q$
<proof>

lemma *const-imply3* :

assumes $\bigwedge \tau \tau'. P\ \tau = P\ \tau' \implies Q\ \tau = Q\ \tau' \implies R\ \tau = R\ \tau'$
shows $const\ P \implies const\ Q \implies const\ R$
<proof>

lemma *const-imply4* :

assumes $\bigwedge \tau \tau'. P\ \tau = P\ \tau' \implies Q\ \tau = Q\ \tau' \implies R\ \tau = R\ \tau' \implies S\ \tau = S\ \tau'$
shows $const\ P \implies const\ Q \implies const\ R \implies const\ S$
<proof>

lemma *const-lam* : $const\ (\lambda\cdot\ e)$

<proof>

lemma *const-true[simp]* : $const\ true$

<proof>

lemma *const-false[simp]* : $const\ false$

<proof>

lemma *const-null[simp]* : $const\ null$

<proof>

lemma *const-invalid [simp]*: $const\ invalid$

<proof>

lemma *const-bot[simp]* : $const\ bot$

<proof>

lemma *const-defined* :

assumes $const\ X$
shows $const\ (\delta\ X)$
<proof>

lemma *const-valid* :
 assumes *const X*
 shows *const (v X)*
 \langle *proof* \rangle

lemma *const-OclAnd* :
 assumes *const X*
 assumes *const X'*
 shows *const (X and X')*
 \langle *proof* \rangle

lemma *const-OclNot* :
 assumes *const X*
 shows *const (not X)*
 \langle *proof* \rangle

lemma *const-OclOr* :
 assumes *const X*
 assumes *const X'*
 shows *const (X or X')*
 \langle *proof* \rangle

lemma *const-OclImplies* :
 assumes *const X*
 assumes *const X'*
 shows *const (X implies X')*
 \langle *proof* \rangle

lemma *const-StrongEq*:
 assumes *const X*
 assumes *const X'*
 shows *const(X \triangleq X')*
 \langle *proof* \rangle

lemma *const-OclIf* :
 assumes *const B*
 and *const C1*
 and *const C2*
 shows *const (if B then C1 else C2 endif)*
 \langle *proof* \rangle

lemma *const-OclValid1*:
 assumes *const x*
 shows $(\tau \models \delta x) = (\tau' \models \delta x)$
 \langle *proof* \rangle

lemma *const-OclValid2*:
 assumes *const x*
 shows $(\tau \models v x) = (\tau' \models v x)$
 \langle *proof* \rangle

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

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

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

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

Miscellaneous: Overloading the syntax of “bottom”

notation *bot* (\perp)

end

theory *UML-PropertyProfiles*
imports *UML-Logic*
begin

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* :: ($\mathfrak{A}, \alpha::\text{null}$)*val* \Rightarrow ($\mathfrak{A}, \beta::\text{null}$)*val*
fixes *g*
assumes *def-scheme*: (*f x*) \equiv $\lambda \tau$. *if* (δ *x*) τ = *true* τ *then g* (*x* τ) *else invalid* τ
begin

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

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

lemma *cp0* : *f X* τ = *f* ($\lambda _.$ *X* τ) τ
<proof>

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

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 def-scheme:  $(f\ x) \equiv \lambda \tau. \text{if } (v\ x)\ \tau = \text{true}\ \tau \text{ then } g\ (x\ \tau) \text{ else } \text{invalid}\ \tau$ 
begin
  lemma strict[simp,code-unfold]:  $f\ \text{invalid} = \text{invalid}$ 
     $\langle \text{proof} \rangle$ 

  lemma cp0 :  $f\ X\ \tau = f\ (\lambda \cdot. X\ \tau)\ \tau$ 
     $\langle \text{proof} \rangle$ 

  lemma cp[simp,code-unfold] :  $cp\ P \Longrightarrow cp\ (\lambda X. f\ (P\ X))$ 
     $\langle \text{proof} \rangle$ 

```

end

```

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

```

```

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

```

end

```

locale profile-mono0 = profile-mono-scheme-defined +
  assumes 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 profile-mono0 < profile-monoa
 $\langle \text{proof} \rangle$ 

```

context *profile-mono0*

begin

```

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

```

```

  lemma def-valid-then-def:  $v(f\ x) = (\delta(f\ x))$ 
     $\langle \text{proof} \rangle$ 

```

end

2.2.2. Property Profiles for Single

```

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

```

2.2.3. Property Profiles for Binary Operators

```

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

```

```

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

```

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

```

locale profile-bin-scheme =
  fixes dx:: ('A,'a::null)val ⇒ 'A Boolean
  fixes dy:: ('A,'b::null)val ⇒ 'A Boolean
  fixes f::('A,'a::null)val ⇒ ('A,'b::null)val ⇒ ('A,'c::null)val
  fixes g
  assumes dx' : profile-single dx
  assumes dy' : profile-single dy
  assumes dx-dy-homo[simp,code-unfold]: cp (f X) ⇒
    cp (λx. f x Y) ⇒
    f X invalid = invalid ⇒
    f invalid Y = invalid ⇒
    (¬ (τ ⊨ dx X) ∨ ¬ (τ ⊨ dy Y)) ⇒
    τ ⊨ (δ f X Y ≐ (dx X and dy Y))
  assumes def-scheme''[simplified]: bin f g dx dy X Y
  assumes 1: τ ⊨ dx X ⇒ τ ⊨ dy Y ⇒ τ ⊨ δ f X Y
begin
  interpretation dx : profile-single dx <proof>
  interpretation dy : profile-single dy <proof>

  lemma strict1[simp,code-unfold]: f invalid y = invalid
  <proof>

  lemma strict2[simp,code-unfold]: f x invalid = invalid
  <proof>

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

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

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

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

  lemma defined-args-valid: (τ ⊨ δ (f x y)) = ((τ ⊨ dx x) ∧ (τ ⊨ dy y))
  <proof>

  lemma const[simp,code-unfold] :
    assumes C1 :const X and C2 : const Y
    shows      const(f X Y)
  <proof>
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 dy:: ('A,'b::null)val ⇒ 'A Boolean
  fixes f::('A,'a::null)val ⇒ ('A,'b::null)val ⇒ ('A,'c::null)val
  fixes g

```

```

assumes  $d_y$  : profile-single  $d_y$ 
assumes  $d_y$ -homo[simp,code-unfold]:  $cp (f X) \implies$ 
   $f X \text{ invalid} = \text{invalid} \implies$ 
   $\neg \tau \models d_y Y \implies$ 
   $\tau \models \delta f X Y \triangleq (\delta X \text{ and } d_y Y)$ 
assumes def-scheme[simplified]: bin f g defined  $d_y X Y$ 
assumes def-body':  $\bigwedge x y \tau. x \neq \text{bot} \implies x \neq \text{null} \implies (d_y y) \tau = \text{true} \tau \implies g x (y \tau) \neq \text{bot} \wedge g x (y \tau) \neq$ 
null
begin
  lemma strict3[simp,code-unfold]:  $f \text{ null } y = \text{invalid}$ 
  <proof>
end

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

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

sublocale profile-bind-d < profile-bin-scheme-defined defined
<proof>

locale profile-bind-v =
  fixes  $f :: ('A, 'a :: \text{null}) \text{val} \Rightarrow ('A, 'b :: \text{null}) \text{val} \Rightarrow ('A, 'c :: \text{null}) \text{val}$ 
  fixes  $g$ 
  assumes def-scheme[simplified]: bin f g defined valid  $X Y$ 
  assumes def-body:  $\bigwedge x y. x \neq \text{bot} \implies x \neq \text{null} \implies y \neq \text{bot} \implies g x y \neq \text{bot} \wedge g x y \neq \text{null}$ 

sublocale profile-bind-v < profile-bin-scheme-defined valid
<proof>

locale profile-binStrongEq-v-v =
  fixes  $f :: ('A, 'a :: \text{null}) \text{val} \Rightarrow ('A, 'b :: \text{null}) \text{val} \Rightarrow ('A) \text{ Boolean}$ 
  assumes def-scheme[simplified]: bin' f StrongEq valid valid  $X Y$ 

sublocale profile-binStrongEq-v-v < profile-bin-scheme valid valid  $f \lambda x y. \sqcup x = y \sqcup$ 
<proof>

context profile-binStrongEq-v-v
begin
  lemma idem[simp,code-unfold]:  $f \text{ null } \text{null} = \text{true}$ 
  <proof>

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

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

```

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

lemma *sym* : $\tau \models (f\ x\ y) \implies \tau \models (f\ y\ x)$
 ⟨proof⟩

lemma *symmetric* : $(f\ x\ y) = (f\ y\ x)$
 ⟨proof⟩

lemma *trans* : $\tau \models (f\ x\ y) \implies \tau \models (f\ y\ z) \implies \tau \models (f\ x\ z)$
 ⟨proof⟩

lemma *StrictRefEq-vs-StrongEq*: $\tau \models (v\ x) \implies \tau \models (v\ y) \implies (\tau \models ((f\ x\ y) \triangleq (x \triangleq y)))$
 ⟨proof⟩

end

locale *profile-bin_{v-v}* =
 fixes $f :: ('\mathfrak{A}, '\alpha :: \text{null}) \text{val} \implies ('\mathfrak{A}, '\beta :: \text{null}) \text{val} \implies (''\mathfrak{A}, '\gamma :: \text{null}) \text{val}$
 fixes g
 assumes *def-scheme[simplified]*: $\text{bin}\ f\ g\ \text{valid}\ \text{valid}\ X\ Y$
 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*
 ⟨proof⟩

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 $'\mathfrak{A}\ \text{Boolean}$, it is just the strict extension of the logical equality:

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

which implies elementary properties like:

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

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

lemma *null-non-false* [*simp,code-unfold*]: $(\text{null} \doteq \text{false}) = \text{false}$
 ⟨proof⟩

lemma *null-non-true* [*simp,code-unfold*]: $(\text{null} \doteq \text{true}) = \text{false}$
 ⟨proof⟩

lemma *false-non-null* [simp,code-unfold]:(*false* \doteq *null*) = *false*
 ⟨*proof*⟩

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

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 *StrictRefEq_{Boolean}* : *profile-bin_{StrongEq^{-v-v}}* λ *x y*. (*x*::('A)Boolean) \doteq *y*
 ⟨*proof*⟩

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* ⟨*proof*⟩

lemma (*invalid* \doteq *true*) = *invalid* ⟨*proof*⟩

lemma (*false* \doteq *invalid*) = *invalid* ⟨*proof*⟩

lemma (*true* \doteq *invalid*) = *invalid* ⟨*proof*⟩

lemma ((*invalid*::('A)Boolean) \doteq *invalid*) = *invalid* ⟨*proof*⟩

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 $\tau \models v(\textit{true})$

Assert $\tau \models \delta(\textit{false})$

Assert $\tau \not\models \delta(\textit{null})$

Assert $\tau \not\models \delta(\textit{invalid})$

Assert $\tau \models v((\textit{null}::('A)Boolean))$

Assert $\tau \not\models v(\textit{invalid})$

Assert $\tau \models (\textit{true and true})$

Assert $\tau \models (\textit{true and true} \triangleq \textit{true})$

Assert $\tau \models ((\textit{null or null}) \triangleq \textit{null})$

Assert $\tau \models ((\textit{null or null}) \doteq \textit{null})$

Assert $\tau \models ((\textit{true} \triangleq \textit{false}) \triangleq \textit{false})$

Assert $\tau \models ((\textit{invalid} \triangleq \textit{false}) \triangleq \textit{false})$

Assert $\tau \models ((\textit{invalid} \doteq \textit{false}) \triangleq \textit{invalid})$

Assert $\tau \models (\textit{true} \langle \rangle \textit{false})$

Assert $\tau \models (\textit{false} \langle \rangle \textit{true})$

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 $\langle\langle \textit{unit} \rangle_{\perp}\rangle_{\perp}$, however the cardinal of this type is more than two, so it would have the cost to consider

Some *None* and *Some* (*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⟩
end

```

```

instantiation Voidbase :: null
begin
  definition null-Void-def: (null::Voidbase) ≡ Abs-Voidbase ⊥ None ⊥
  instance ⟨proof⟩
end

```

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

```

defs (overloaded) StrictRefEqVoid[code-unfold] :
  (x::( $\mathfrak{A}$ ) Void) ≐ y ≡ λ τ. if (v x) τ = true τ ∧ (v y) τ = true τ
    then (x ≐ y) τ
    else invalid τ

```

Property proof in terms of *profile-bin*_{StrongEq^v-v}

```

interpretation StrictRefEqVoid : profile-binStrongEqv-v λ x y. (x::( $\mathfrak{A}$ ) Void) ≐ y
  ⟨proof⟩

```

2.3.2. Basic Void Constants

2.3.3. Validity and Definedness Properties

```

lemma δ(null::( $\mathfrak{A}$ ) Void) = false ⟨proof⟩
lemma v(null::( $\mathfrak{A}$ ) Void) = true ⟨proof⟩

```

```

lemma [simp,code-unfold]: δ (λ-. Abs-Voidbase None) = false
  ⟨proof⟩

```

```

lemma [simp,code-unfold]: v (λ-. Abs-Voidbase None) = false
  ⟨proof⟩

```

```

lemma [simp,code-unfold]: δ (λ-. Abs-Voidbase ⊥ None ⊥) = false
  ⟨proof⟩

```

```

lemma [simp,code-unfold]: v (λ-. Abs-Voidbase ⊥ None ⊥) = true
  ⟨proof⟩

```

2.3.4. Test Statements

```

Assert τ ⊨ ((null::( $\mathfrak{A}$ ) Void) ≐ 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:

```

defs (overloaded) StrictRefEqInteger[code-unfold] :
  (x::( $\mathcal{A}$ )Integer)  $\doteq$  y  $\equiv$   $\lambda$   $\tau$ . if (v x)  $\tau$  = true  $\tau$   $\wedge$  (v y)  $\tau$  = true  $\tau$ 
    then (x  $\hat{=}$  y)  $\tau$ 
    else invalid  $\tau$ 

```

Property proof in terms of *profile-bin*_{StrongEq^v-v}

```

interpretation StrictRefEqInteger : profile-binStrongEqv-v  $\lambda$  x y. (x::( $\mathcal{A}$ )Integer)  $\doteq$  y
  <proof>

```

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 OclInt0 ::( $\mathcal{A}$ )Integer (0) where 0 = ( $\lambda$  - .  $\underline{\underline{0}}$ ::int $\mathcal{A}$ )
definition OclInt1 ::( $\mathcal{A}$ )Integer (1) where 1 = ( $\lambda$  - .  $\underline{\underline{1}}$ ::int $\mathcal{A}$ )
definition OclInt2 ::( $\mathcal{A}$ )Integer (2) where 2 = ( $\lambda$  - .  $\underline{\underline{2}}$ ::int $\mathcal{A}$ )

```

Etc.

```

definition OclInt3 ::( $\mathcal{A}$ )Integer (3) where 3 = ( $\lambda$  - .  $\underline{\underline{3}}$ ::int $\mathcal{A}$ )
definition OclInt4 ::( $\mathcal{A}$ )Integer (4) where 4 = ( $\lambda$  - .  $\underline{\underline{4}}$ ::int $\mathcal{A}$ )
definition OclInt5 ::( $\mathcal{A}$ )Integer (5) where 5 = ( $\lambda$  - .  $\underline{\underline{5}}$ ::int $\mathcal{A}$ )
definition OclInt6 ::( $\mathcal{A}$ )Integer (6) where 6 = ( $\lambda$  - .  $\underline{\underline{6}}$ ::int $\mathcal{A}$ )
definition OclInt7 ::( $\mathcal{A}$ )Integer (7) where 7 = ( $\lambda$  - .  $\underline{\underline{7}}$ ::int $\mathcal{A}$ )
definition OclInt8 ::( $\mathcal{A}$ )Integer (8) where 8 = ( $\lambda$  - .  $\underline{\underline{8}}$ ::int $\mathcal{A}$ )
definition OclInt9 ::( $\mathcal{A}$ )Integer (9) where 9 = ( $\lambda$  - .  $\underline{\underline{9}}$ ::int $\mathcal{A}$ )
definition OclInt10 ::( $\mathcal{A}$ )Integer (10) where 10 = ( $\lambda$  - .  $\underline{\underline{10}}$ ::int $\mathcal{A}$ )

```

2.4.3. Validity and Definedness Properties

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma [simp,code-unfold]:  $\delta$  0 = true <proof>

```

```

lemma [simp,code-unfold]: v 0 = true <proof>

```

```

lemma [simp,code-unfold]:  $\delta$  1 = true <proof>

```

```

lemma [simp,code-unfold]: v 1 = true <proof>

```


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

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** $+_{int}$ 40)

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

interpretation $OclAdd_{Integer} : \text{profile-bin}_{d-d} \text{ op } +_{int} \lambda x y. \llbracket x \tau \rrbracket + \llbracket y \tau \rrbracket$
 $\langle \text{proof} \rangle$

definition $OclMinus_{Integer} :: ('A)Integer \Rightarrow ('A)Integer \Rightarrow ('A)Integer$ (**infix** $-_{int}$ 41)

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

interpretation $OclMinus_{Integer} : \text{profile-bin}_{d-d} \text{ op } -_{int} \lambda x y. \llbracket x \tau \rrbracket - \llbracket y \tau \rrbracket$
 $\langle \text{proof} \rangle$

definition $OclMult_{Integer} :: ('A)Integer \Rightarrow ('A)Integer \Rightarrow ('A)Integer$ (**infix** $*_{int}$ 45)

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

interpretation $OclMult_{Integer} : \text{profile-bin}_{d-d} \text{ op } *_{int} \lambda x y. \llbracket x \tau \rrbracket * \llbracket y \tau \rrbracket$
 $\langle \text{proof} \rangle$

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** div_{int} 45)

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

definition $OclModulus_{Integer} :: ('A)Integer \Rightarrow ('A)Integer \Rightarrow ('A)Integer$ (**infix** mod_{int} 45)

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

definition $OclLess_{Integer} :: ('A)Integer \Rightarrow ('A)Integer \Rightarrow ('A)Boolean$ (**infix** $<_{int}$ 35)

where $x <_{int} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
 then $\perp^{\ulcorner x \urcorner} \tau^{\lrcorner} < \ulcorner y \urcorner \tau^{\lrcorner}$
 else *invalid* τ

interpretation $OclLess_{Integer} : \text{profile-bin}_{d-d} \text{ op } <_{int} \lambda x y. \perp^{\ulcorner x \urcorner} < \ulcorner y \urcorner$
 $\langle \text{proof} \rangle$

definition $OclLe_{Integer} :: ('A)Integer \Rightarrow ('A)Integer \Rightarrow ('A)Boolean$ (**infix** \leq_{int} 35)

where $x \leq_{int} y \equiv \lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (\delta y) \tau = \text{true } \tau$
 then $\perp^{\ulcorner x \urcorner} \tau^{\lrcorner} \leq \ulcorner y \urcorner \tau^{\lrcorner}$
 else *invalid* τ

interpretation $OclLe_{Integer} : \text{profile-bin}_{d-d} \text{ op } \leq_{int} \lambda x y. \perp^{\ulcorner x \urcorner} \leq \ulcorner y \urcorner$
 $\langle \text{proof} \rangle$

Basic Properties

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

Execution with Invalid or Null or Zero as Argument

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

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

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 $integer\text{-non-null}$ [*simp*]: $((\lambda \cdot. \perp^{\ulcorner n \urcorner}) \doteq (\text{null} :: ('A)Integer)) = \text{false}$
 $\langle \text{proof} \rangle$

lemma $null\text{-non-integer}$ [*simp*]: $((\text{null} :: ('A)Integer) \doteq (\lambda \cdot. \perp^{\ulcorner n \urcorner})) = \text{false}$
 $\langle \text{proof} \rangle$

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

```

lemma null-non-OclInt8 [simp,code-unfold]: (null  $\doteq$  8) = false  $\langle$ proof $\rangle$ 
lemma OclInt9-non-null [simp,code-unfold]: (9  $\doteq$  null) = false  $\langle$ proof $\rangle$ 
lemma null-non-OclInt9 [simp,code-unfold]: (null  $\doteq$  9) = false  $\langle$ proof $\rangle$ 

```

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  $\tau \models v \ (null::('A)Integer)$ 
Assert  $\tau \models (invalid \doteq invalid)$ 
Assert  $\tau \models (null \doteq null)$ 
Assert  $\tau \models (\mathbf{4} \doteq \mathbf{4})$ 
Assert  $\tau \not\models (\mathbf{9} \doteq \mathbf{10})$ 
Assert  $\tau \not\models (invalid \doteq \mathbf{10})$ 
Assert  $\tau \not\models (null \doteq \mathbf{10})$ 
Assert  $\tau \not\models (invalid \doteq (invalid::('A)Integer))$ 
Assert  $\tau \not\models v \ (invalid \doteq (invalid::('A)Integer))$ 
Assert  $\tau \not\models (invalid <> (invalid::('A)Integer))$ 
Assert  $\tau \not\models v \ (invalid <> (invalid::('A)Integer))$ 
Assert  $\tau \models (null \doteq (null::('A)Integer))$ 
Assert  $\tau \models (null \doteq (null::('A)Integer))$ 
Assert  $\tau \models (\mathbf{4} \doteq \mathbf{4})$ 
Assert  $\tau \not\models (\mathbf{4} <> \mathbf{4})$ 
Assert  $\tau \not\models (\mathbf{4} \doteq \mathbf{10})$ 
Assert  $\tau \models (\mathbf{4} <> \mathbf{10})$ 
Assert  $\tau \not\models (\mathbf{0} <_{int} null)$ 
Assert  $\tau \not\models (\delta \ (\mathbf{0} <_{int} null))$ 

```

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 *'A Boolean*-case as strict extension of the strong equality:

```

defs (overloaded) StrictRefEqReal [code-unfold] :
  (x::('A)Real)  $\doteq$  y  $\equiv$   $\lambda \tau. \text{if } (v \ x) \ \tau = \text{true } \tau \wedge (v \ y) \ \tau = \text{true } \tau$ 
    then (x  $\doteq$  y)  $\tau$ 
    else invalid  $\tau$ 

```

Property proof in terms of *profile-bin_{StrongEq-v-v}*

interpretation $StrictRefEq_{Real} : profile-bin_{StrongEq-v-v} \lambda x y. (x::('A)Real) \doteq y$
 ⟨proof⟩

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 $OclReal0 :: ('A)Real$ **(0.0)** **where** $0.0 = (\lambda - . \underline{0}::real_{\perp})$
definition $OclReal1 :: ('A)Real$ **(1.0)** **where** $1.0 = (\lambda - . \underline{1}::real_{\perp})$
definition $OclReal2 :: ('A)Real$ **(2.0)** **where** $2.0 = (\lambda - . \underline{2}::real_{\perp})$

Etc.

definition $OclReal3 :: ('A)Real$ **(3.0)** **where** $3.0 = (\lambda - . \underline{3}::real_{\perp})$
definition $OclReal4 :: ('A)Real$ **(4.0)** **where** $4.0 = (\lambda - . \underline{4}::real_{\perp})$
definition $OclReal5 :: ('A)Real$ **(5.0)** **where** $5.0 = (\lambda - . \underline{5}::real_{\perp})$
definition $OclReal6 :: ('A)Real$ **(6.0)** **where** $6.0 = (\lambda - . \underline{6}::real_{\perp})$
definition $OclReal7 :: ('A)Real$ **(7.0)** **where** $7.0 = (\lambda - . \underline{7}::real_{\perp})$
definition $OclReal8 :: ('A)Real$ **(8.0)** **where** $8.0 = (\lambda - . \underline{8}::real_{\perp})$
definition $OclReal9 :: ('A)Real$ **(9.0)** **where** $9.0 = (\lambda - . \underline{9}::real_{\perp})$
definition $OclReal10 :: ('A)Real$ **(10.0)** **where** $10.0 = (\lambda - . \underline{10}::real_{\perp})$
definition $OclRealpi :: ('A)Real$ **(π)** **where** $\pi = (\lambda - . \underline{pi}_{\perp})$

2.5.3. Validity and Definedness Properties

lemma $\delta(null::('A)Real) = false$ ⟨proof⟩
lemma $v(null::('A)Real) = true$ ⟨proof⟩

lemma $[simp,code-unfold]: \delta(\lambda - . \underline{n}_{\perp}) = true$
 ⟨proof⟩

lemma $[simp,code-unfold]: v(\lambda - . \underline{n}_{\perp}) = true$
 ⟨proof⟩

lemma $[simp,code-unfold]: \delta 0.0 = true$ ⟨proof⟩
lemma $[simp,code-unfold]: v 0.0 = true$ ⟨proof⟩
lemma $[simp,code-unfold]: \delta 1.0 = true$ ⟨proof⟩
lemma $[simp,code-unfold]: v 1.0 = true$ ⟨proof⟩
lemma $[simp,code-unfold]: \delta 2.0 = true$ ⟨proof⟩
lemma $[simp,code-unfold]: v 2.0 = true$ ⟨proof⟩
lemma $[simp,code-unfold]: \delta 6.0 = true$ ⟨proof⟩
lemma $[simp,code-unfold]: v 6.0 = true$ ⟨proof⟩
lemma $[simp,code-unfold]: \delta 8.0 = true$ ⟨proof⟩
lemma $[simp,code-unfold]: v 8.0 = true$ ⟨proof⟩
lemma $[simp,code-unfold]: \delta 9.0 = true$ ⟨proof⟩
lemma $[simp,code-unfold]: v 9.0 = true$ ⟨proof⟩

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 $OclAdd_{Real} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Real$ (**infix** $+_{real}$ 40)
where $x +_{real} y \equiv \lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (\delta y) \tau = true \tau$
 then $\lfloor \lceil x \rceil \tau \rceil + \lceil y \rceil \tau \rfloor$
 else *invalid* τ

interpretation $OclAdd_{Real} : \text{profile-bin}_{d-d} \text{ op } +_{real} \lambda x y. \lfloor \lceil x \rceil + \lceil y \rceil \rfloor$
 ⟨proof⟩

definition $OclMinus_{Real} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Real$ (**infix** $-_{real}$ 41)
where $x -_{real} y \equiv \lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (\delta y) \tau = true \tau$
 then $\lfloor \lceil x \rceil \tau \rceil - \lceil y \rceil \tau \rfloor$
 else *invalid* τ

interpretation $OclMinus_{Real} : \text{profile-bin}_{d-d} \text{ op } -_{real} \lambda x y. \lfloor \lceil x \rceil - \lceil y \rceil \rfloor$
 ⟨proof⟩

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

interpretation $OclMult_{Real} : \text{profile-bin}_{d-d} \text{ op } *_{real} \lambda x y. \lfloor \lceil x \rceil * \lceil y \rceil \rfloor$
 ⟨proof⟩

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

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

definition $mod\text{-float } a \ b = a - real \ (floor \ (a / b)) * b$

definition $OclModulus_{Real} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Real$ (**infix** mod_{real} 45)
where $x mod_{real} y \equiv \lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (\delta y) \tau = true \tau$
 then *if* $y \tau \neq OclReal0 \tau$ then $\lfloor mod\text{-float } \lceil x \rceil \tau \rceil \lceil y \rceil \tau \rfloor$ else *invalid* τ
 else *invalid* τ

definition $OclLess_{Real} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Boolean$ (**infix** $<_{real}$ 35)
where $x <_{real} y \equiv \lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (\delta y) \tau = true \tau$
 then $\lfloor \lceil x \rceil \tau \rceil < \lceil y \rceil \tau \rfloor$
 else *invalid* τ

interpretation $OclLess_{Real} : \text{profile-bin}_{d-d} \text{ op } <_{real} \lambda x y. \lfloor \lceil x \rceil < \lceil y \rceil \rfloor$
 ⟨proof⟩

definition $OclLe_{Real} :: ('A)Real \Rightarrow ('A)Real \Rightarrow ('A)Boolean$ (**infix** \leq_{real} 35)
where $x \leq_{real} y \equiv \lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (\delta y) \tau = true \tau$
 then $\lfloor \lceil x \rceil \tau \rceil \leq \lceil y \rceil \tau \rfloor$
 else *invalid* τ

interpretation $OclLe_{Real} : \text{profile-bin}_{d-d} \text{ op } \leq_{real} \lambda x y. \lfloor \lceil x \rceil \leq \lceil y \rceil \rfloor$
 ⟨proof⟩

Basic Properties

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

Execution with Invalid or Null or Zero as Argument

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

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

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

lemma *real-non-null* [*simp*]: (($\lambda-. _ \sqcup n \sqcup$) \doteq ($\text{null}::(\mathfrak{A})\text{Real}$)) = *false*
<proof>

lemma *null-non-real* [*simp*]: (($\text{null}::(\mathfrak{A})\text{Real}$) \doteq ($\lambda-. _ \sqcup n \sqcup$)) = *false*
<proof>

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

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

Elementary computations on Real

Assert $\tau \models \mathbf{1.0} <> \mathbf{2.0}$
Assert $\tau \models \mathbf{2.0} <> \mathbf{1.0}$
Assert $\tau \models \mathbf{2.0} \doteq \mathbf{2.0}$

Assert $\tau \models v \mathbf{4.0}$
Assert $\tau \models \delta \mathbf{4.0}$
Assert $\tau \models v (\text{null}::(\mathfrak{A})\text{Real})$
Assert $\tau \models (\text{invalid} \triangleq \text{invalid})$
Assert $\tau \models (\text{null} \triangleq \text{null})$
Assert $\tau \models (\mathbf{4.0} \triangleq \mathbf{4.0})$
Assert $\tau \not\models (\mathbf{9.0} \triangleq \mathbf{10.0})$

```

Assert  $\tau \not\models (\text{invalid} \hat{=} \mathbf{10.0})$ 
Assert  $\tau \not\models (\text{null} \hat{=} \mathbf{10.0})$ 
Assert  $\tau \not\models (\text{invalid} \hat{=} (\text{invalid}::(\mathfrak{A})\text{Real}))$ 
Assert  $\tau \not\models v (\text{invalid} \hat{=} (\text{invalid}::(\mathfrak{A})\text{Real}))$ 
Assert  $\tau \not\models (\text{invalid} <> (\text{invalid}::(\mathfrak{A})\text{Real}))$ 
Assert  $\tau \not\models v (\text{invalid} <> (\text{invalid}::(\mathfrak{A})\text{Real}))$ 
Assert  $\tau \models (\text{null} \hat{=} (\text{null}::(\mathfrak{A})\text{Real}))$ 
Assert  $\tau \models (\text{null} \hat{=} (\text{null}::(\mathfrak{A})\text{Real}))$ 
Assert  $\tau \models (\mathbf{4.0} \hat{=} \mathbf{4.0})$ 
Assert  $\tau \not\models (\mathbf{4.0} <> \mathbf{4.0})$ 
Assert  $\tau \not\models (\mathbf{4.0} \hat{=} \mathbf{10.0})$ 
Assert  $\tau \models (\mathbf{4.0} <> \mathbf{10.0})$ 
Assert  $\tau \not\models (\mathbf{0.0} <_{\text{real}} \text{null})$ 
Assert  $\tau \not\models (\delta (\mathbf{0.0} <_{\text{real}} \text{null}))$ 

```

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 \mathfrak{A} Boolean-case as strict extension of the strong equality:

```

defs (overloaded) StrictRefEqString[code-unfold] :
  ( $x::(\mathfrak{A})\text{String} \hat{=} y \equiv \lambda \tau. \text{if } (v\ x) \tau = \text{true } \tau \wedge (v\ y) \tau = \text{true } \tau$ 
    then  $(x \hat{=} y) \tau$ 
    else  $\text{invalid } \tau$ )

```

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

```

interpretation StrictRefEqString : profile-binStrongEq-v-v  $\lambda x\ y. (x::(\mathfrak{A})\text{String} \hat{=} y)$ 
  <proof>

```

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 :: ( $\mathfrak{A}$ )String (a)   where a =  $(\lambda \_ . \_ \llcorner''a''\lrcorner)$ 
definition OclStringb :: ( $\mathfrak{A}$ )String (b)   where b =  $(\lambda \_ . \_ \llcorner''b''\lrcorner)$ 
definition OclStringc :: ( $\mathfrak{A}$ )String (c)   where c =  $(\lambda \_ . \_ \llcorner''c''\lrcorner)$ 

```

Etc.

2.6.3. Validity and Definedness Properties

```

lemma  $\delta(\text{null}::(\mathfrak{A})\text{String}) = \text{false}$  <proof>

```

```

lemma  $v(\text{null}::(\mathfrak{A})\text{String}) = \text{true}$  <proof>

```

```

lemma [simp,code-unfold]:  $\delta(\lambda \_ . \_ \llcorner n \lrcorner) = \text{true}$ 

```

<proof>

lemma [*simp,code-unfold*]: $v (\lambda \cdot \perp n \perp) = true$
<proof>

lemma [*simp,code-unfold*]: $\delta a = true$ *<proof>*

lemma [*simp,code-unfold*]: $v a = true$ *<proof>*

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 $OclAdd_{String} :: ('A)String \Rightarrow ('A)String \Rightarrow ('A)String$ (**infix** $+_{string}$ 40)

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

interpretation $OclAdd_{String} : \text{profile-bin}_{d-d} \text{ op } +_{string} \lambda x y. \perp\text{concat } [\ulcorner x \urcorner, \ulcorner y \urcorner \rrbracket_{\perp}$
<proof>

Basic Properties

lemma $OclAdd_{String}\text{-not-commute}: \exists X Y. (X +_{string} Y) \neq (Y +_{string} X)$
<proof>

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 $\tau \models v (null::('A)String)$

Assert $\tau \models (invalid \triangleq invalid)$

Assert $\tau \models (null \triangleq null)$

Assert $\tau \models (a \triangleq a)$

Assert $\tau \models (a \triangleq b)$

Assert $\tau \models (invalid \triangleq b)$

Assert $\tau \models (null \triangleq b)$

Assert $\tau \models (invalid \doteq (invalid::('A)String))$

Assert $\tau \models v (invalid \doteq (invalid::('A)String))$

Assert $\tau \models (invalid <> (invalid::('A)String))$

Assert $\tau \models v (invalid <> (invalid::('A)String))$

Assert $\tau \models (null \doteq (null::('A)String))$


```

Assert  $\tau \models (\text{null} \doteq (\text{null}::('A)\text{String}) )$ 
Assert  $\tau \models (b \doteq b)$ 
Assert  $\tau \models (b \langle \rangle b)$ 
Assert  $\tau \models (b \doteq c)$ 
Assert  $\tau \models (b \langle \rangle c)$ 

```

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 $A[\text{simp}]: \text{Rep-Pair}_{base} x \neq \text{None} \implies \text{Rep-Pair}_{base} x \neq \text{null} \implies (\text{fst } \ulcorner \text{Rep-Pair}_{base} x \urcorner) \neq \text{bot}$
 $\langle \text{proof} \rangle$

lemma $A'[\text{simp}]: x \neq \text{bot} \implies x \neq \text{null} \implies (\text{fst } \ulcorner \text{Rep-Pair}_{base} x \urcorner) \neq \text{bot}$
 $\langle \text{proof} \rangle$

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

lemma $B'[\text{simp}]: x \neq \text{bot} \implies x \neq \text{null} \implies (\text{snd } \ulcorner \text{Rep-Pair}_{base} x \urcorner) \neq \text{bot}$
 $\langle \text{proof} \rangle$

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:

```

defs (overloaded) StrictRefEqPair :
   $((x::('A, 'alpha::\text{null}, 'beta::\text{null})\text{Pair}) \doteq y) \equiv (\lambda \tau. \text{if } (v x) \tau = \text{true} \wedge (v y) \tau = \text{true} \tau$ 
     $\text{then } (x \doteq y) \tau$ 
     $\text{else } \text{invalid } \tau)$ 

```

Property proof in terms of *profile-bin_{StrongEq-v-v}*

interpretation $\text{StrictRefEqPair} : \text{profile-bin}_{\text{StrongEq-v-v}} \lambda x y. (x::('A, 'alpha::\text{null}, 'beta::\text{null})\text{Pair}) \doteq y$
 $\langle \text{proof} \rangle$

2.7.3. Standard Operations Definitions

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

Definition: Pair Constructor

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

interpretation $OclPair : \text{profile-bin}_{v-v}$
 $OclPair \lambda x y. Abs\text{-}Pair_{base} \llcorner (x, y) \llcorner$
 $\langle \text{proof} \rangle$

Definition: First

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

interpretation $OclFirst : \text{profile-mono}_d$ $OclFirst \lambda x. \text{fst } \ulcorner Rep\text{-}Pair_{base} (x) \urcorner$
 $\langle \text{proof} \rangle$

Definition: Second

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

interpretation $OclSecond : \text{profile-mono}_d$ $OclSecond \lambda x. \text{snd } \ulcorner Rep\text{-}Pair_{base} (x) \urcorner$
 $\langle \text{proof} \rangle$

2.7.4. Logical Properties

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

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

2.7.5. Algebraic Execution Properties

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

lemma proj2-exec [simp, code-unfold] : $Pair\{X, Y\} . Second () = (\text{if } (v X) \text{ then } Y \text{ else } \text{invalid } \text{endif})$
 $\langle \text{proof} \rangle$

2.7.6. Test Statements

instantiation $Pair_{base} :: (\text{equal}, \text{equal}) \text{equal}$

begin

definition $HOL.\text{equal } k l \longleftrightarrow (k :: ('a :: \text{equal}, 'b :: \text{equal}) \text{Pair}_{base}) = l$

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

end

lemma *equal-Pair_{base}-code* [code]:

$HOL.equal\ k\ (l::('a::\{equal,null\},'b::\{equal,null\})Pair_{base}) \longleftrightarrow Rep-Pair_{base}\ k = Rep-Pair_{base}\ l$
 ⟨proof⟩

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

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* (\perp)

2.8. Collection Type Bag: Operations

definition *Rep-Bag-base'* $x = \{(x0, y). y < {}^{\top}Rep-Bag_{base}\ x^{\top} x0\}$

definition *Rep-Bag-base* $x\ \tau = \{(x0, y). y < {}^{\top}Rep-Bag_{base}\ (x\ \tau)^{\top} x0\}$

definition *Rep-Set-base* $x\ \tau = fst\ '\{(x0, y). y < {}^{\top}Rep-Bag_{base}\ (x\ \tau)^{\top} x0\}$

definition *ApproxEq* (**infixl** $\cong 30$)

where $X \cong Y \equiv \lambda\ \tau. \perp Rep-Set-base\ X\ \tau = Rep-Set-base\ Y\ \tau \perp$

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 $Integer :: (\mathfrak{A}, Integer_{base}) Bag$
where $Integer \equiv (\lambda \tau. (Abs-Bag_{base} \circ Some \circ Some) (\lambda None \Rightarrow 0 \mid Some\ None \Rightarrow 0 \mid - \Rightarrow 1))$

definition $Integer_{null} :: (\mathfrak{A}, Integer_{base}) Bag$
where $Integer_{null} \equiv (\lambda \tau. (Abs-Bag_{base} \circ Some \circ Some) (\lambda None \Rightarrow 0 \mid - \Rightarrow 1))$

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

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

This allows the theorems:
 $\tau \models \delta x \implies \tau \models (Integer \text{-} \> includes_{Bag}(x)) \quad \tau \models \delta x \implies \tau \models Integer \quad \triangleq$
 $(Integer \text{-} \> including_{Bag}(x))$
and
 $\tau \models v x \implies \tau \models (Integer_{null} \text{-} \> includes_{Bag}(x)) \quad \tau \models v x \implies \tau \models Integer_{null} \quad \triangleq$
 $(Integer_{null} \text{-} \> including_{Bag}(x))$
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 $Void_{null} :: (\mathfrak{A}, Void_{base}) Bag$
where $Void_{null} \equiv (\lambda \tau. (Abs-Bag_{base} \circ Some \circ Some) (\lambda x. \text{if } x = Abs-Void_{base} (Some\ None) \text{ then } 1 \text{ else } 0))$

definition $Void_{empty} :: (\mathfrak{A}, Void_{base}) Bag$
where $Void_{empty} \equiv (\lambda \tau. (Abs-Bag_{base} \circ Some \circ Some) (\lambda -. 0))$

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

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

lemma assumes $\tau \models \delta (V :: (\mathfrak{A}, Void_{base}) Bag)$
shows $\tau \models V \cong Void_{null} \vee \tau \models V \cong Void_{empty}$
 $\langle proof \rangle$

definition $Boolean :: (\mathfrak{A}, Boolean_{base}) Bag$
where $Boolean \equiv (\lambda \tau. (Abs-Bag_{base} \circ Some \circ Some) (\lambda None \Rightarrow 0 \mid Some\ None \Rightarrow 0 \mid - \Rightarrow 1))$

definition $Boolean_{null} :: (\mathfrak{A}, Boolean_{base}) Bag$
where $Boolean_{null} \equiv (\lambda \tau. (Abs-Bag_{base} \circ Some \circ Some) (\lambda None \Rightarrow 0 \mid - \Rightarrow 1))$

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

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

definition $String :: (\mathfrak{A}, String_{base}) Bag$
where $String \equiv (\lambda \tau. (Abs-Bag_{base} \circ Some \circ Some) (\lambda None \Rightarrow 0 \mid Some\ None \Rightarrow 0 \mid - \Rightarrow 1))$

definition $String_{null} :: (\mathfrak{A}, String_{base}) Bag$
where $String_{null} \equiv (\lambda \tau. (Abs-Bag_{base} \circ Some \circ Some) (\lambda None \Rightarrow 0 \mid - \Rightarrow 1))$

lemma $String\text{-}defined : \delta String = true$

<proof>

lemma *String_{null}-defined* : δ *String_{null}* = true

<proof>

definition *Real* :: (\mathfrak{A} , *Real_{base}*) *Bag*

where *Real* \equiv (λ τ . (*Abs-Bag_{base}* *o Some o Some*) (λ *None* \Rightarrow 0 | *Some None* \Rightarrow 0 | - \Rightarrow 1))

definition *Real_{null}* :: (\mathfrak{A} , *Real_{base}*) *Bag*

where *Real_{null}* \equiv (λ τ . (*Abs-Bag_{base}* *o Some o Some*) (λ *None* \Rightarrow 0 | - \Rightarrow 1))

lemma *Real-defined* : δ *Real* = true

<proof>

lemma *Real_{null}-defined* : δ *Real_{null}* = true

<proof>

2.8.2. Basic Properties of the Bag Type

Every element in a defined bag is valid.

lemma *Bag-inv-lemma*: $\tau \models (\delta X) \Longrightarrow \ulcorner \text{Rep-Bag}_{base} (X \tau) \urcorner \text{bot} = 0$

<proof>

lemma *Bag-inv-lemma'* :

assumes *x-def* : $\tau \models \delta X$

and *e-mem* : $\ulcorner \text{Rep-Bag}_{base} (X \tau) \urcorner e \geq 1$

shows $\tau \models v (\lambda \cdot e)$

<proof>

lemma *abs-rep-simp'* :

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

shows *Abs-Bag_{base}* $\llcorner \ulcorner \text{Rep-Bag}_{base} (S \tau) \urcorner \llcorner = S \tau$

<proof>

lemma *invalid-bag-OclNot-defined* [*simp,code-unfold*]: $\delta(\text{invalid}::(\mathfrak{A}, \alpha::\text{null}) \text{Bag}) = \text{false}$ *<proof>*

lemma *null-bag-OclNot-defined* [*simp,code-unfold*]: $\delta(\text{null}::(\mathfrak{A}, \alpha::\text{null}) \text{Bag}) = \text{false}$

<proof>

lemma *invalid-bag-valid* [*simp,code-unfold*]: $v(\text{invalid}::(\mathfrak{A}, \alpha::\text{null}) \text{Bag}) = \text{false}$

<proof>

lemma *null-bag-valid* [*simp,code-unfold*]: $v(\text{null}::(\mathfrak{A}, \alpha::\text{null}) \text{Bag}) = \text{true}$

<proof>

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

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:

defs (overloaded) *StrictRefEq_{Bag}* :

$(x::(\mathfrak{A}, \alpha::\text{null}) \text{Bag}) \doteq y \equiv \lambda \tau$. if $(v x) \tau = \text{true} \wedge (v y) \tau = \text{true}$ τ
then $(x \doteq y) \tau$
else *invalid* τ

One might object here that for the case of objects, this is an empty definition. The answer is no, we will restrain later on states and objects such that any object has its oid stored inside the object (so the ref, under which an object can be referenced in the store will 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 $StrictRefEq_{Bag} : profile-bin_{StrongEq}^{-v-v} \lambda x y. (x::('A, 'a)::null) Bag) \doteq y$
 ⟨proof⟩

2.8.4. Constants: mtBag

definition $mtBag::('A, 'a)::null) Bag (Bag\{\})$
where $Bag\{\} \equiv (\lambda \tau. Abs-Bag_{base} \perp \lambda \tau. 0::nat_{\perp})$

lemma $mtBag-defined[simp, code-unfold]: \delta(Bag\{\}) = true$
 ⟨proof⟩

lemma $mtBag-valid[simp, code-unfold]: v(Bag\{\}) = true$
 ⟨proof⟩

lemma $mtBag-rep-bag: \ulcorner Rep-Bag_{base} (Bag\{\} \tau) \urcorner = (\lambda \tau. 0)$
 ⟨proof⟩ **lemma** $[simp, code-unfold]: const Bag\{\}$
 ⟨proof⟩

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

2.8.5. Definition: Including

definition $OclIncluding :: [('A, 'a)::null) Bag, ('A, 'a) val] \Rightarrow ('A, 'a) Bag$
where $OclIncluding x y = (\lambda \tau. if (\delta x) \tau = true \tau \wedge (v y) \tau = true \tau$
 $then Abs-Bag_{base} \perp \ulcorner Rep-Bag_{base} (x \tau) \urcorner$
 $((y \tau) := \ulcorner Rep-Bag_{base} (x \tau) \urcorner (y \tau) + 1)$
 $else invalid \tau)$

notation $OclIncluding (->including_{Bag} '(-))$

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⟩

syntax

$-OclFinbag :: args \Rightarrow ('A, 'a)::null) Bag (Bag\{-})$

translations

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

$Bag\{x\} == CONST OclIncluding (Bag\{\}) x$

2.8.6. Definition: Excluding

definition $OclExcluding :: [('A, 'a)::null) Bag, ('A, 'a) val] \Rightarrow ('A, 'a) Bag$
where $OclExcluding x y = (\lambda \tau. if (\delta x) \tau = true \tau \wedge (v y) \tau = true \tau$
 $then Abs-Bag_{base} \perp \ulcorner Rep-Bag_{base} (x \tau) \urcorner ((y \tau) := 0::nat)_{\perp}$
 $else invalid \tau)$

notation $OclExcluding (->excluding_{Bag} '(-))$

interpretation *OclExcluding: profile-bin_{d-v} OclExcluding*
 $\lambda x y. \text{Abs-Bag}_{base} \perp \ulcorner \text{Rep-Bag}_{base}(x) \urcorner (y := 0 :: nat) \perp$
 ⟨proof⟩

2.8.7. Definition: Includes

definition *OclIncludes* :: $[(\mathfrak{A}, \alpha :: null) \text{Bag}, (\mathfrak{A}, \alpha) \text{val}] \Rightarrow \mathfrak{A} \text{ Boolean}$
where $\text{OclIncludes } x y = (\lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge (v y) \tau = \text{true } \tau$
 $\text{then } \perp \ulcorner \text{Rep-Bag}_{base}(x \tau) \urcorner (y \tau) > 0 \perp$
 $\text{else } \perp)$

notation *OclIncludes* $(--> \text{includes}_{Bag} '())$

interpretation *OclIncludes : profile-bin_{d-v} OclIncludes* $\lambda x y. \perp \ulcorner \text{Rep-Bag}_{base} x \urcorner y > 0 \perp$
 ⟨proof⟩

2.8.8. Definition: Excludes

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

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* $\lambda x y. \perp \ulcorner \text{Rep-Bag}_{base} x \urcorner y \leq 0 \perp$
 ⟨proof⟩

2.8.9. Definition: Size

definition *OclSize* :: $(\mathfrak{A}, \alpha :: null) \text{Bag} \Rightarrow \mathfrak{A} \text{ Integer}$
where $\text{OclSize } x = (\lambda \tau. \text{if } (\delta x) \tau = \text{true } \tau \wedge \text{finite}(\text{Rep-Bag-base } x \tau)$
 $\text{then } \perp \text{int}(\text{card}(\text{Rep-Bag-base } x \tau)) \perp$
 $\text{else } \perp)$

notation
OclSize $(--> \text{size}_{Bag} '())$

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* :: $(\mathfrak{A}, \alpha :: null) \text{Bag} \Rightarrow \mathfrak{A} \text{ Boolean}$
where $\text{OclIsEmpty } x = ((v x \text{ and not } (\delta x)) \text{ or } ((\text{OclSize } x) \doteq \mathbf{0}))$
notation *OclIsEmpty* $(--> \text{isEmpty}_{Bag} '())$

2.8.11. Definition: NotEmpty

definition *OclNotEmpty* :: $(\mathfrak{A}, \alpha :: null) \text{Bag} \Rightarrow \mathfrak{A} \text{ Boolean}$
where $\text{OclNotEmpty } x = \text{not}(\text{OclIsEmpty } x)$
notation *OclNotEmpty* $(--> \text{notEmpty}_{Bag} '())$

2.8.12. Definition: Any

definition *OclANY* :: $[(\mathfrak{A}, \alpha :: null) \text{Bag}] \Rightarrow (\mathfrak{A}, \alpha) \text{ val}$
where $\text{OclANY } x = (\lambda \tau. \text{if } (v x) \tau = \text{true } \tau$
 $\text{then if } (\delta x \text{ and } \text{OclNotEmpty } x) \tau = \text{true } \tau$

then SOME $y. y \in (\text{Rep-Set-base } x \ \tau)$
 else null τ
 else \perp)

notation $\text{OclANY} \ (\rightarrow \text{any}_{\text{Bag}} '())$

2.8.13. Definition: Forall

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

definition $\text{OclForall} \ :: [('A, 'a::\text{null}) \text{Bag}, ('A, 'a) \text{val} \Rightarrow ('A) \text{Boolean}] \Rightarrow 'A \ \text{Boolean}$
where $\text{OclForall } S \ P = (\lambda \ \tau. \text{if } (\delta \ S) \ \tau = \text{true } \tau$
 then $\text{if } (\exists x \in \text{Rep-Set-base } S \ \tau. P \ (\lambda \cdot x) \ \tau = \text{false } \tau)$
 then $\text{false } \tau$
 else $\text{if } (\exists x \in \text{Rep-Set-base } S \ \tau. P \ (\lambda \cdot x) \ \tau = \text{invalid } \tau)$
 then $\text{invalid } \tau$
 else $\text{if } (\exists x \in \text{Rep-Set-base } S \ \tau. P \ (\lambda \cdot x) \ \tau = \text{null } \tau)$
 then $\text{null } \tau$
 else $\text{true } \tau$
 else \perp)

syntax

$\text{-OclForallBag} \ :: [('A, 'a::\text{null}) \ \text{Bag}, \text{id}, ('A) \ \text{Boolean}] \Rightarrow 'A \ \text{Boolean} \ (\rightarrow \text{forAll}_{\text{Bag}} '(-|-'))$

translations

$X \rightarrow \text{forAll}_{\text{Bag}} (x \mid P) == \text{CONST UML-Bag.OclForall } X \ (\%x. P)$

2.8.14. Definition: Exists

Like OclForall , OclExists is also not strict.

definition $\text{OclExists} \ :: [('A, 'a::\text{null}) \ \text{Bag}, ('A, 'a) \ \text{val} \Rightarrow ('A) \ \text{Boolean}] \Rightarrow 'A \ \text{Boolean}$
where $\text{OclExists } S \ P = \text{not}(\text{UML-Bag.OclForall } S \ (\lambda \ X. \text{not } (P \ X)))$

syntax

$\text{-OclExistBag} \ :: [('A, 'a::\text{null}) \ \text{Bag}, \text{id}, ('A) \ \text{Boolean}] \Rightarrow 'A \ \text{Boolean} \ (\rightarrow \text{exists}_{\text{Bag}} '(-|-'))$

translations

$X \rightarrow \text{exists}_{\text{Bag}} (x \mid P) == \text{CONST UML-Bag.OclExists } X \ (\%x. P)$

2.8.15. Definition: Iterate

definition $\text{OclIterate} \ :: [('A, 'a::\text{null}) \ \text{Bag}, ('A, 'b::\text{null}) \ \text{val},$
 $(('A, 'a) \ \text{val} \Rightarrow ('A, 'b) \ \text{val}) \Rightarrow ('A, 'b) \ \text{val}] \Rightarrow ('A, 'b) \ \text{val}$
where $\text{OclIterate } S \ A \ F = (\lambda \ \tau. \text{if } (\delta \ S) \ \tau = \text{true } \tau \wedge (v \ A) \ \tau = \text{true } \tau \wedge \text{finite } (\text{Rep-Bag-base } S \ \tau)$
 then $\text{Finite-Set.fold } (F \ o \ (\lambda a \ \tau. a) \ o \ \text{fst}) \ A \ (\text{Rep-Bag-base } S \ \tau) \ \tau$
 else \perp)

syntax

$\text{-OclIterateBag} \ :: [('A, 'a::\text{null}) \ \text{Bag}, \text{idt}, \text{idt}, 'a, 'b] \Rightarrow ('A, 'b) \ \text{val}$
 $(\rightarrow \text{iterate}_{\text{Bag}} '(-;=- | -'))$

translations

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

2.8.16. Definition: Select

definition $\text{OclSelect} \ :: [('A, 'a::\text{null}) \ \text{Bag}, ('A, 'a) \ \text{val} \Rightarrow ('A) \ \text{Boolean}] \Rightarrow ('A, 'a) \ \text{Bag}$
where $\text{OclSelect } S \ P = (\lambda \ \tau. \text{if } (\delta \ S) \ \tau = \text{true } \tau$
 then $\text{if } (\exists x \in \text{Rep-Set-base } S \ \tau. P \ (\lambda \cdot x) \ \tau = \text{invalid } \tau)$
 then $\text{invalid } \tau$
 else $\text{Abs-Bag}_{\text{base}} \ \perp \ \lambda x.$
 let $n = \ulcorner \text{Rep-Bag}_{\text{base}} (S \ \tau) \urcorner x$ in
 if $n = 0 \mid P \ (\lambda \cdot x) \ \tau = \text{false } \tau$ then

$$\begin{array}{l} 0 \\ \text{else} \\ n_{\perp} \\ \text{else invalid } \tau \end{array}$$

syntax

$\text{-OclSelectBag} :: [(\mathfrak{A}, \alpha :: \text{null}) \text{Bag}, \text{id}, (\mathfrak{A}) \text{Boolean}] \Rightarrow \mathfrak{A} \text{ Boolean} \quad ((-) \rightarrow \text{select}_{\text{Bag}}'(-))$

translations

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

2.8.17. Definition: Reject

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

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

syntax

$\text{-OclRejectBag} :: [(\mathfrak{A}, \alpha :: \text{null}) \text{Bag}, \text{id}, (\mathfrak{A}) \text{Boolean}] \Rightarrow \mathfrak{A} \text{ Boolean} \quad ((-) \rightarrow \text{reject}_{\text{Bag}}'(-))$

translations

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

2.8.18. Definition: IncludesAll

definition $\text{OclIncludesAll} :: [(\mathfrak{A}, \alpha :: \text{null}) \text{Bag}, (\mathfrak{A}, \alpha) \text{Bag}] \Rightarrow \mathfrak{A} \text{ Boolean}$

where $\text{OclIncludesAll } x \ y = (\lambda \tau. \text{ if } (\delta \ x) \ \tau = \text{true } \tau \wedge (\delta \ y) \ \tau = \text{true } \tau \\ \text{ then } \perp \text{Rep-Bag-base } y \ \tau \subseteq \text{Rep-Bag-base } x \ \tau \perp \\ \text{ else } \perp)$

notation $\text{OclIncludesAll } (-) \rightarrow \text{includesAll}_{\text{Bag}}'(-)$

interpretation $\text{OclIncludesAll} : \text{profile-bin}_{d-d} \ \text{OclIncludesAll } \lambda x \ y. \perp \text{Rep-Bag-base}' \ y \subseteq \text{Rep-Bag-base}' \ x \perp$
 $\langle \text{proof} \rangle$

2.8.19. Definition: ExcludesAll

definition $\text{OclExcludesAll} :: [(\mathfrak{A}, \alpha :: \text{null}) \text{Bag}, (\mathfrak{A}, \alpha) \text{Bag}] \Rightarrow \mathfrak{A} \text{ Boolean}$

where $\text{OclExcludesAll } x \ y = (\lambda \tau. \text{ if } (\delta \ x) \ \tau = \text{true } \tau \wedge (\delta \ y) \ \tau = \text{true } \tau \\ \text{ then } \perp \text{Rep-Bag-base } y \ \tau \cap \text{Rep-Bag-base } x \ \tau = \{ \} \perp \\ \text{ else } \perp)$

notation $\text{OclExcludesAll } (-) \rightarrow \text{excludesAll}_{\text{Bag}}'(-)$

interpretation $\text{OclExcludesAll} : \text{profile-bin}_{d-d} \ \text{OclExcludesAll } \lambda x \ y. \perp \text{Rep-Bag-base}' \ y \cap \text{Rep-Bag-base}' \ x = \{ \} \perp$
 $\langle \text{proof} \rangle$

2.8.20. Definition: Union

definition $\text{OclUnion} :: [(\mathfrak{A}, \alpha :: \text{null}) \text{Bag}, (\mathfrak{A}, \alpha) \text{Bag}] \Rightarrow (\mathfrak{A}, \alpha) \text{Bag}$

where $\text{OclUnion } x \ y = (\lambda \tau. \text{ if } (\delta \ x) \ \tau = \text{true } \tau \wedge (\delta \ y) \ \tau = \text{true } \tau \\ \text{ then } \text{Abs-Bag}_{\text{base}} \perp \lambda X. \text{ } \overline{\text{Rep-Bag}_{\text{base}}} (x \ \tau) \overline{\text{Rep-Bag}_{\text{base}}} (y \ \tau) \perp \\ \text{ else invalid } \tau)$

notation $\text{OclUnion } (-) \rightarrow \text{union}_{\text{Bag}}'(-)$

interpretation $\text{OclUnion} :$

$\text{profile-bin}_{d-d} \ \text{OclUnion } \lambda x \ y. \text{Abs-Bag}_{\text{base}} \perp \lambda X. \text{ } \overline{\text{Rep-Bag}_{\text{base}}} x \overline{\text{Rep-Bag}_{\text{base}}} y \perp$

$\langle \text{proof} \rangle$

2.8.21. Definition: Intersection

definition $\text{OclIntersection} :: [(\mathfrak{A}, \alpha :: \text{null}) \text{Bag}, (\mathfrak{A}, \alpha) \text{Bag}] \Rightarrow (\mathfrak{A}, \alpha) \text{Bag}$

where $OclIntersection\ x\ y = (\lambda\ \tau.\ \text{if } (\delta\ x)\ \tau = \text{true } \tau \wedge (\delta\ y)\ \tau = \text{true } \tau$
 $\text{then } Abs\text{-}Bag_{base} \perp \lambda\ X.\ \min\ (\ulcorner Rep\text{-}Bag_{base}\ (x\ \tau) \urcorner X)$
 $\quad\quad\quad (\ulcorner Rep\text{-}Bag_{base}\ (y\ \tau) \urcorner X) \perp$
 $\text{else } \perp)$

notation $OclIntersection(\rightarrow intersection_{Bag}('))$

interpretation $OclIntersection$:

$profile\text{-}bin_{d-d}\ OclIntersection\ \lambda x\ y.\ Abs\text{-}Bag_{base}\ \perp\ \lambda\ X.\ \min\ (\ulcorner Rep\text{-}Bag_{base}\ x \urcorner X)$
 $\quad\quad\quad (\ulcorner Rep\text{-}Bag_{base}\ y \urcorner X) \perp$

$\langle proof \rangle$

2.8.22. Definition: Count

definition $OclCount :: [(\mathcal{A}, \alpha :: null)\ Bag, (\mathcal{A}, \alpha)\ val] \Rightarrow (\mathcal{A})\ Integer$

where $OclCount\ x\ y = (\lambda\ \tau.\ \text{if } (\delta\ x)\ \tau = \text{true } \tau \wedge (\delta\ y)\ \tau = \text{true } \tau$
 $\text{then } \perp int(\ulcorner Rep\text{-}Bag_{base}\ (x\ \tau) \urcorner (y\ \tau)) \perp$
 $\text{else } invalid\ \tau)$

notation $OclCount(\rightarrow count_{Bag}('))$

interpretation $OclCount : profile\text{-}bin_{d-d}\ OclCount\ \lambda x\ y.\ \perp int(\ulcorner Rep\text{-}Bag_{base}\ x \urcorner y) \perp$

$\langle proof \rangle$

2.8.23. Definition (future operators)

consts

$OclSum :: (\mathcal{A}, \alpha :: null)\ Bag \Rightarrow \mathcal{A}\ Integer$

notation $OclSum(\rightarrow sum_{Bag}('))$

2.8.24. Test Statements

instantiation $Bag_{base} :: (equal)\ equal$

begin

definition $HOL.equal\ k\ l \longleftrightarrow (k :: ('a :: equal)\ Bag_{base}) = l$

instance $\langle proof \rangle$

end

lemma $equal\text{-}Bag_{base}\text{-}code\ [code]:$

$HOL.equal\ k\ (l :: ('a :: \{equal, null\})\ Bag_{base}) \longleftrightarrow Rep\text{-}Bag_{base}\ k = Rep\text{-}Bag_{base}\ l$

$\langle proof \rangle$

Assert $\tau \models (Bag\{\} \doteq Bag\{\})$

end

theory $UML\text{-}Set$

imports $../basic\text{-}types/UML\text{-}Void$

$../basic\text{-}types/UML\text{-}Boolean$

$../basic\text{-}types/UML\text{-}Integer$

$../basic\text{-}types/UML\text{-}String$

$../basic\text{-}types/UML\text{-}Real$

begin

no-notation *None* (\perp)

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 $Integer :: ('A, Integer_{base}) Set$

where $Integer \equiv (\lambda \tau. (Abs-Set_{base} \circ Some \circ Some) ((Some \circ Some) ' (UNIV::int set)))$

definition $Integer_{null} :: ('A, Integer_{base}) Set$

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

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

<proof>

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

<proof>

This allows the theorems:

$\tau \models \delta x \implies \tau \models (Integer \text{-} > includes_{Set}(x)) \quad \tau \models \delta x \implies \tau \models Integer \triangleq (Integer \text{-} > including_{Set}(x))$

and

$\tau \models v x \implies \tau \models (Integer_{null} \text{-} > includes_{Set}(x)) \quad \tau \models v x \implies \tau \models Integer_{null} \triangleq (Integer_{null} \text{-} > including_{Set}(x))$

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 $Void_{null} :: ('A, Void_{base}) Set$

where $Void_{null} \equiv (\lambda \tau. (Abs-Set_{base} \circ Some \circ Some) \{Abs-Void_{base} (Some None)\})$

definition $Void_{empty} :: ('A, Void_{base}) Set$

where $Void_{empty} \equiv (\lambda \tau. (Abs-Set_{base} \circ Some \circ Some) \{\})$

lemma $Void_{null}\text{-}defined : \delta Void_{null} = true$

<proof>

lemma $Void_{empty}\text{-}defined : \delta Void_{empty} = true$

<proof>

lemma assumes $\tau \models \delta (V :: ('A, Void_{base}) Set)$

shows $\tau \models V \triangleq \text{Void}_{\text{null}} \vee \tau \models V \triangleq \text{Void}_{\text{empty}}$
 ⟨proof⟩

definition $\text{Boolean} :: ('A, \text{Boolean}_{\text{base}}) \text{Set}$
where $\text{Boolean} \equiv (\lambda \tau. (\text{Abs-Set}_{\text{base}} \circ \text{Some} \circ \text{Some}) ((\text{Some} \circ \text{Some}) ' (\text{UNIV}::\text{bool set})))$

definition $\text{Boolean}_{\text{null}} :: ('A, \text{Boolean}_{\text{base}}) \text{Set}$
where $\text{Boolean}_{\text{null}} \equiv (\lambda \tau. (\text{Abs-Set}_{\text{base}} \circ \text{Some} \circ \text{Some}) (\text{Some} ' (\text{UNIV}::\text{bool option set})))$

lemma $\text{Boolean-defined} : \delta \text{ Boolean} = \text{true}$
 ⟨proof⟩

lemma $\text{Boolean}_{\text{null}}\text{-defined} : \delta \text{ Boolean}_{\text{null}} = \text{true}$
 ⟨proof⟩

definition $\text{String} :: ('A, \text{String}_{\text{base}}) \text{Set}$
where $\text{String} \equiv (\lambda \tau. (\text{Abs-Set}_{\text{base}} \circ \text{Some} \circ \text{Some}) ((\text{Some} \circ \text{Some}) ' (\text{UNIV}::\text{string set})))$

definition $\text{String}_{\text{null}} :: ('A, \text{String}_{\text{base}}) \text{Set}$
where $\text{String}_{\text{null}} \equiv (\lambda \tau. (\text{Abs-Set}_{\text{base}} \circ \text{Some} \circ \text{Some}) (\text{Some} ' (\text{UNIV}::\text{string option set})))$

lemma $\text{String-defined} : \delta \text{ String} = \text{true}$
 ⟨proof⟩

lemma $\text{String}_{\text{null}}\text{-defined} : \delta \text{ String}_{\text{null}} = \text{true}$
 ⟨proof⟩

definition $\text{Real} :: ('A, \text{Real}_{\text{base}}) \text{Set}$
where $\text{Real} \equiv (\lambda \tau. (\text{Abs-Set}_{\text{base}} \circ \text{Some} \circ \text{Some}) ((\text{Some} \circ \text{Some}) ' (\text{UNIV}::\text{real set})))$

definition $\text{Real}_{\text{null}} :: ('A, \text{Real}_{\text{base}}) \text{Set}$
where $\text{Real}_{\text{null}} \equiv (\lambda \tau. (\text{Abs-Set}_{\text{base}} \circ \text{Some} \circ \text{Some}) (\text{Some} ' (\text{UNIV}::\text{real option set})))$

lemma $\text{Real-defined} : \delta \text{ Real} = \text{true}$
 ⟨proof⟩

lemma $\text{Real}_{\text{null}}\text{-defined} : \delta \text{ Real}_{\text{null}} = \text{true}$
 ⟨proof⟩

2.9.2. Basic Properties of the Set Type

Every element in a defined set is valid.

lemma $\text{Set-inv-lemma} : \tau \models (\delta X) \implies \forall x \in {}^{\top}\text{Rep-Set}_{\text{base}} (X \tau)^{\top}. x \neq \text{bot}$
 ⟨proof⟩

lemma $\text{Set-inv-lemma}' :$
assumes $x\text{-def} : \tau \models \delta X$
and $e\text{-mem} : e \in {}^{\top}\text{Rep-Set}_{\text{base}} (X \tau)^{\top}$
shows $\tau \models v (\lambda \cdot. e)$
 ⟨proof⟩

lemma $\text{abs-rep-simp}' :$
assumes $S\text{-all-def} : \tau \models \delta S$
shows $\text{Abs-Set}_{\text{base}} \sqsubseteq {}^{\top}\text{Rep-Set}_{\text{base}} (S \tau)^{\top} \sqsubseteq S \tau$
 ⟨proof⟩

lemma $S\text{-lift}' :$

assumes S -all-def : $(\tau :: \mathfrak{A} \text{ st}) \models \delta S$
shows $\exists S'. (\lambda a (-::\mathfrak{A} \text{ st}). a) \text{ ' } \ulcorner \text{Rep-Set}_{base} (S \tau) \urcorner = (\lambda a (-::\mathfrak{A} \text{ st}). \ulcorner a \urcorner) \text{ ' } S'$
 $\langle \text{proof} \rangle$

lemma *invalid-set-OclNot-defined* [simp,code-unfold]: $\delta(\text{invalid}::(\mathfrak{A},\alpha::\text{null}) \text{ Set}) = \text{false}$ $\langle \text{proof} \rangle$

lemma *null-set-OclNot-defined* [simp,code-unfold]: $\delta(\text{null}::(\mathfrak{A},\alpha::\text{null}) \text{ Set}) = \text{false}$
 $\langle \text{proof} \rangle$

lemma *invalid-set-valid* [simp,code-unfold]: $v(\text{invalid}::(\mathfrak{A},\alpha::\text{null}) \text{ Set}) = \text{false}$
 $\langle \text{proof} \rangle$

lemma *null-set-valid* [simp,code-unfold]: $v(\text{null}::(\mathfrak{A},\alpha::\text{null}) \text{ Set}) = \text{true}$
 $\langle \text{proof} \rangle$

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

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:

defs (overloaded) *StrictRefEqSet* :
 $(x::(\mathfrak{A},\alpha::\text{null})\text{Set}) \doteq y \equiv \lambda \tau. \text{if } (v \ x) \ \tau = \text{true} \ \tau \wedge (v \ y) \ \tau = \text{true} \ \tau$
 $\text{then } (x \hat{=} y) \ \tau$
 $\text{else } \text{invalid} \ \tau$

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

Property proof in terms of *profile-bin_{StrongEq^{v-v}}*

interpretation *StrictRefEqSet* : *profile-bin_{StrongEq^{v-v}}* $\lambda x y. (x::(\mathfrak{A},\alpha::\text{null})\text{Set}) \doteq y$
 $\langle \text{proof} \rangle$

2.9.4. Constants: mtSet

definition *mtSet*:: $(\mathfrak{A},\alpha::\text{null}) \text{ Set} \ (\text{Set}\{\})$
where $\text{Set}\{\} \equiv (\lambda \tau. \text{Abs-Set}_{base} \ \ulcorner \{\} \urcorner :: \alpha \ \text{set}_{\ulcorner})$

lemma *mtSet-defined*[simp,code-unfold]: $\delta(\text{Set}\{\}) = \text{true}$
 $\langle \text{proof} \rangle$

lemma *mtSet-valid*[simp,code-unfold]: $v(\text{Set}\{\}) = \text{true}$
 $\langle \text{proof} \rangle$

lemma *mtSet-rep-set*: $\ulcorner \text{Rep-Set}_{base} (\text{Set}\{\}) \ \tau \urcorner = \{\}$
 $\langle \text{proof} \rangle$

lemma [simp,code-unfold]: *const* $\text{Set}\{\}$
 $\langle \text{proof} \rangle$

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 $OclIncluding x y = (\lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (v y) \tau = true \tau$
 $\quad \text{then } Abs-Set_{base} \perp \ulcorner Rep-Set_{base} (x \tau) \urcorner \cup \{y \tau\} \perp$
 $\quad \text{else } invalid \tau)$
notation $OclIncluding \ (-\>including_{Set} '(-'))$

interpretation $OclIncluding : profile-bin_{d-v} OclIncluding \lambda x y. Abs-Set_{base} \perp \ulcorner Rep-Set_{base} x \urcorner \cup \{y\} \perp$
 $\langle proof \rangle$

syntax

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

translations

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

2.9.6. Definition: Excluding

definition $OclExcluding :: [(\mathfrak{A}, \alpha :: null) Set, (\mathfrak{A}, \alpha) val] \Rightarrow (\mathfrak{A}, \alpha) Set$
where $OclExcluding x y = (\lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (v y) \tau = true \tau$
 $\quad \text{then } Abs-Set_{base} \perp \ulcorner Rep-Set_{base} (x \tau) \urcorner - \{y \tau\} \perp$
 $\quad \text{else } \perp)$
notation $OclExcluding \ (-\>excluding_{Set} '(-'))$

lemma $OclExcluding-inv: (x :: Set('b :: \{null\})) \neq \perp \implies x \neq null \implies y \neq \perp \implies$
 $\quad \perp \ulcorner Rep-Set_{base} x \urcorner - \{y\} \perp \in \{X. X = bot \vee X = null \vee (\forall x \in \ulcorner X \urcorner. x \neq bot)\}$
 $\langle proof \rangle$

interpretation $OclExcluding : profile-bin_{d-v} OclExcluding \lambda x y. Abs-Set_{base} \perp \ulcorner Rep-Set_{base} x \urcorner - \{y\} \perp$
 $\langle proof \rangle$

2.9.7. Definition: Includes

definition $OclIncludes :: [(\mathfrak{A}, \alpha :: null) Set, (\mathfrak{A}, \alpha) val] \Rightarrow \mathfrak{A} Boolean$
where $OclIncludes x y = (\lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (v y) \tau = true \tau$
 $\quad \text{then } \perp (y \tau) \in \ulcorner Rep-Set_{base} (x \tau) \urcorner$
 $\quad \text{else } \perp)$
notation $OclIncludes \ (-\>includes_{Set} '(-'))$

interpretation $OclIncludes : profile-bin_{d-v} OclIncludes \lambda x y. \perp y \in \ulcorner Rep-Set_{base} x \urcorner$
 $\langle proof \rangle$

2.9.8. Definition: Excludes

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

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 $OclExcludes : profile-bin_{d-v} OclExcludes \lambda x y. \perp y \notin \ulcorner Rep-Set_{base} x \urcorner$
 $\langle proof \rangle$

2.9.9. Definition: Size

definition $OclSize$ $:: ('A, 'a::null) Set \Rightarrow 'A Integer$
where $OclSize x = (\lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge finite(\ulcorner Rep-Set_{base} (x \tau) \urcorner)$
 $\text{then } \perp \text{int}(\text{card } \ulcorner Rep-Set_{base} (x \tau) \urcorner) \perp$
 $\text{else } \perp)$

notation $OclSize$ $(-->size_{Set}'('))$

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 $OclIsEmpty$ $:: ('A, 'a::null) Set \Rightarrow 'A Boolean$
where $OclIsEmpty x = ((v x \text{ and not } (\delta x)) \text{ or } ((OclSize x) \doteq 0))$
notation $OclIsEmpty$ $(-->isEmpty_{Set}'('))$

2.9.11. Definition: NotEmpty

definition $OclNotEmpty$ $:: ('A, 'a::null) Set \Rightarrow 'A Boolean$
where $OclNotEmpty x = not(OclIsEmpty x)$
notation $OclNotEmpty$ $(-->notEmpty_{Set}'('))$

2.9.12. Definition: Any

definition $OclANY$ $:: [('A, 'a::null) Set] \Rightarrow ('A, 'a) val$
where $OclANY x = (\lambda \tau. \text{if } (v x) \tau = true \tau$
 $\text{then if } (\delta x \text{ and } OclNotEmpty x) \tau = true \tau$
 $\text{then } SOME y. y \in \ulcorner Rep-Set_{base} (x \tau) \urcorner$
 $\text{else } null \tau$
 $\text{else } \perp)$

notation $OclANY$ $(-->any_{Set}'('))$

2.9.13. Definition: Forall

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

definition $OclForall$ $:: [('A, 'a::null) Set, ('A, 'a) val \Rightarrow ('A) Boolean] \Rightarrow 'A Boolean$
where $OclForall S P = (\lambda \tau. \text{if } (\delta S) \tau = true \tau$
 $\text{then if } (\exists x \in \ulcorner Rep-Set_{base} (S \tau) \urcorner. P(\lambda -. x) \tau = false \tau)$
 $\text{then } false \tau$
 $\text{else if } (\exists x \in \ulcorner Rep-Set_{base} (S \tau) \urcorner. P(\lambda -. x) \tau = invalid \tau)$
 $\text{then } invalid \tau$
 $\text{else if } (\exists x \in \ulcorner Rep-Set_{base} (S \tau) \urcorner. P(\lambda -. x) \tau = null \tau)$
 $\text{then } null \tau$
 $\text{else } true \tau$
 $\text{else } \perp)$

syntax

$-OclForallSet :: [('A, 'a::null) Set, id, ('A) Boolean] \Rightarrow 'A Boolean \quad ((-)>forAll_{Set}'(-|'))$

translations

$X->forAll_{Set}(x | P) == CONST UML-Set.OclForall X (\%x. P)$

2.9.14. Definition: Exists

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

definition $OclExists$ $:: [('A, 'a::null) Set, ('A, 'a) val \Rightarrow ('A) Boolean] \Rightarrow 'A Boolean$

where $OclExists S P = not(UML-Set.OclForall S (\lambda X. not (P X)))$

syntax

$-OclExistSet :: [(\mathfrak{A}, 'alpha::null) Set, id, (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean \quad ((-)\rightarrow exists_{Set} '(-|-'))$

translations

$X\rightarrow exists_{Set}(x | P) == CONST UML-Set.OclExists X (\%x. P)$

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 $OclIterate S A F = (\lambda \tau. \text{if } (\delta S) \tau = true \tau \wedge (v A) \tau = true \tau \wedge finite^{\Gamma} Rep-Set_{base} (S \tau)^{\Gamma}$
 $\text{then } (Finite-Set.fold (F) (A) ((\lambda a \tau. a) \text{ ' }^{\Gamma} Rep-Set_{base} (S \tau)^{\Gamma})) \tau$
 $\text{else } \perp)$

syntax

$-OclIterateSet :: [(\mathfrak{A}, 'alpha::null) Set, idt, idt, 'alpha, 'beta] \Rightarrow (\mathfrak{A}, 'gamma) val$
 $(- \rightarrow iterates_{Set} '(-|-|-))$

translations

$X\rightarrow iterates_{Set}(a; x = A | P) == CONST OclIterate X A (\%a. (\% x. P))$

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 $OclSelect S P = (\lambda \tau. \text{if } (\delta S) \tau = true \tau$
 $\text{then if } (\exists x \in^{\Gamma} Rep-Set_{base} (S \tau)^{\Gamma}. P(\lambda -. x) \tau = invalid \tau)$
 $\text{then } invalid \tau$
 $\text{else } Abs-Set_{base} \sqcup \{x \in^{\Gamma} Rep-Set_{base} (S \tau)^{\Gamma}. P(\lambda -. x) \tau \neq false \tau\} \sqcup$
 $\text{else } invalid \tau)$

syntax

$-OclSelectSet :: [(\mathfrak{A}, 'alpha::null) Set, id, (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean \quad ((-)\rightarrow select_{Set} '(-|-'))$

translations

$X\rightarrow select_{Set}(x | P) == CONST OclSelect X (\% x. P)$

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

$-OclRejectSet :: [(\mathfrak{A}, 'alpha::null) Set, id, (\mathfrak{A}) Boolean] \Rightarrow \mathfrak{A} Boolean \quad ((-)\rightarrow reject_{Set} '(-|-'))$

translations

$X\rightarrow reject_{Set}(x | P) == CONST OclReject X (\% x. P)$

2.9.18. Definition: IncludesAll

definition $OclIncludesAll :: [(\mathfrak{A}, 'alpha::null) Set, (\mathfrak{A}, 'alpha) Set] \Rightarrow \mathfrak{A} Boolean$

where $OclIncludesAll x y = (\lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (\delta y) \tau = true \tau$
 $\text{then } \sqcup^{\Gamma} Rep-Set_{base} (y \tau)^{\Gamma} \subseteq^{\Gamma} Rep-Set_{base} (x \tau)^{\Gamma} \sqcup$
 $\text{else } \perp)$

notation $OclIncludesAll (-\rightarrow includesAll_{Set} '(-|-'))$

interpretation $OclIncludesAll : profile-bin_d-d OclIncludesAll \lambda x y. \sqcup^{\Gamma} Rep-Set_{base} y^{\Gamma} \subseteq^{\Gamma} Rep-Set_{base} x^{\Gamma} \sqcup$
 $\langle proof \rangle$

2.9.19. Definition: ExcludesAll

definition $OclExcludesAll :: [(\mathfrak{A}, 'alpha::null) Set, (\mathfrak{A}, 'alpha) Set] \Rightarrow \mathfrak{A} Boolean$

where $OclExcludesAll x y = (\lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (\delta y) \tau = true \tau$
 $\text{then } \sqcup^{\Gamma} Rep-Set_{base} (y \tau)^{\Gamma} \cap^{\Gamma} Rep-Set_{base} (x \tau)^{\Gamma} = \{\} \sqcup$

$else \perp$)

notation $OclExcludesAll$ ($-->excludesAll_{Set}'(-)$)

interpretation $OclExcludesAll$: $profile-bin_{d-d}$ $OclExcludesAll \lambda x y. \perp^{\ulcorner Rep-Set_{base} \urcorner} y^{\urcorner} \cap \ulcorner Rep-Set_{base} \urcorner x^{\urcorner} = \{\perp\}$
 $\langle proof \rangle$

2.9.20. Definition: Union

definition $OclUnion$:: $[('A, 'a::null) Set, ('A, 'a) Set] \Rightarrow ('A, 'a) Set$
where $OclUnion x y = (\lambda \tau. \text{ if } (\delta x) \tau = true \tau \wedge (\delta y) \tau = true \tau$
 $\text{ then } Abs-Set_{base} \ulcorner Rep-Set_{base} (y \tau)^{\urcorner} \cup \ulcorner Rep-Set_{base} (x \tau)^{\urcorner}$
 $\text{ else } \perp$)

notation $OclUnion$ ($-->union_{Set}'(-)$)

lemma $OclUnion-inv$: $(x::Set('b::\{null\})) \neq \perp \implies x \neq null \implies y \neq \perp \implies y \neq null \implies$
 $\ulcorner Rep-Set_{base} y^{\urcorner} \cup \ulcorner Rep-Set_{base} x^{\urcorner} \in \{X. X = bot \vee X = null \vee (\forall x \in \ulcorner X^{\urcorner}. x \neq bot)\}$
 $\langle proof \rangle$

interpretation $OclUnion$: $profile-bin_{d-d}$ $OclUnion \lambda x y. Abs-Set_{base} \ulcorner Rep-Set_{base} y^{\urcorner} \cup \ulcorner Rep-Set_{base} x^{\urcorner}$
 $\langle proof \rangle$

2.9.21. Definition: Intersection

definition $OclIntersection$:: $[('A, 'a::null) Set, ('A, 'a) Set] \Rightarrow ('A, 'a) Set$
where $OclIntersection x y = (\lambda \tau. \text{ if } (\delta x) \tau = true \tau \wedge (\delta y) \tau = true \tau$
 $\text{ then } Abs-Set_{base} \ulcorner Rep-Set_{base} (y \tau)^{\urcorner}$
 $\quad \cap \ulcorner Rep-Set_{base} (x \tau)^{\urcorner}$
 $\text{ else } \perp$)

notation $OclIntersection$ ($-->intersection_{Set}'(-)$)

lemma $OclIntersection-inv$: $(x::Set('b::\{null\})) \neq \perp \implies x \neq null \implies y \neq \perp \implies y \neq null \implies$
 $\ulcorner Rep-Set_{base} y^{\urcorner} \cap \ulcorner Rep-Set_{base} x^{\urcorner} \in \{X. X = bot \vee X = null \vee (\forall x \in \ulcorner X^{\urcorner}. x \neq bot)\}$
 $\langle proof \rangle$

interpretation $OclIntersection$: $profile-bin_{d-d}$ $OclIntersection \lambda x y. Abs-Set_{base} \ulcorner Rep-Set_{base} y^{\urcorner} \cap$
 $\ulcorner Rep-Set_{base} x^{\urcorner}$
 $\langle proof \rangle$

2.9.22. Definition (future operators)

consts

$OclCount$:: $[('A, 'a::null) Set, ('A, 'a) Set] \Rightarrow 'A$ Integer
 $OclSum$:: $('A, 'a::null) Set \Rightarrow 'A$ Integer

notation $OclCount$ ($-->count_{Set}'(-)$)

notation $OclSum$ ($-->sum_{Set}'(-)$)

2.9.23. Logical Properties

$OclIncluding$

lemma $OclIncluding-valid-args-valid$:

$(\tau \models v(X \rightarrow including_{Set}(x))) = ((\tau \models (\delta X)) \wedge (\tau \models (v x)))$
 $\langle proof \rangle$

lemma $OclIncluding-valid-args-valid''[simp, code-unfold]$:

$v(X \rightarrow including_{Set}(x)) = ((\delta X) \text{ and } (v x))$

<proof>

etc. etc.

OclExcluding

lemma *OclExcluding-valid-args-valid*:

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

<proof>

lemma *OclExcluding-valid-args-valid'*[simp,code-unfold]:

$v(X \rightarrow \text{excluding}_{Set}(x)) = ((\delta X) \text{ and } (v x))$

<proof>

OclIncludes

lemma *OclIncludes-valid-args-valid*:

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

<proof>

lemma *OclIncludes-valid-args-valid'*[simp,code-unfold]:

$v(X \rightarrow \text{includes}_{Set}(x)) = ((\delta X) \text{ and } (v x))$

<proof>

OclExcludes

lemma *OclExcludes-valid-args-valid*:

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

<proof>

lemma *OclExcludes-valid-args-valid'*[simp,code-unfold]:

$v(X \rightarrow \text{excludes}_{Set}(x)) = ((\delta X) \text{ and } (v x))$

<proof>

OclSize

lemma *OclSize-defined-args-valid*: $\tau \models \delta (X \rightarrow \text{size}_{Set}()) \implies \tau \models \delta X$

<proof>

lemma *OclSize-infinite*:

assumes *non-finite*: $\tau \models \text{not}(\delta(S \rightarrow \text{size}_{Set}()))$

shows $(\tau \models \text{not}(\delta(S))) \vee \neg \text{finite } \ulcorner \text{Rep-Set}_{base} (S \tau) \urcorner$

<proof>

lemma $\tau \models \delta X \implies \neg \text{finite } \ulcorner \text{Rep-Set}_{base} (X \tau) \urcorner \implies \neg \tau \models \delta (X \rightarrow \text{size}_{Set}())$

<proof>

lemma *size-defined*:

assumes *X-finite*: $\bigwedge \tau. \text{finite } \ulcorner \text{Rep-Set}_{base} (X \tau) \urcorner$

shows $\delta (X \rightarrow \text{size}_{Set}()) = \delta X$

<proof>

lemma *size-defined'*:

assumes *X-finite*: $\text{finite } \ulcorner \text{Rep-Set}_{base} (X \tau) \urcorner$

shows $(\tau \models \delta (X \rightarrow \text{size}_{Set}())) = (\tau \models \delta X)$

<proof>

OclIsEmpty

lemma *OclIsEmpty-defined-args-valid*: $\tau \models \delta (X \rightarrow \text{isEmpty}_{Set}()) \implies \tau \models v X$

<proof>

lemma $\tau \models \delta (\text{null} \rightarrow \text{isEmpty}_{\text{Set}}())$
 ⟨proof⟩

lemma *OclIsEmpty-infinite*: $\tau \models \delta X \implies \neg \text{finite } \ulcorner \text{Rep-Set}_{\text{base}} (X \ \tau) \urcorner \implies \neg \tau \models \delta (X \rightarrow \text{isEmpty}_{\text{Set}}())$
 ⟨proof⟩

OclNotEmpty

lemma *OclNotEmpty-defined-args-valid*: $\tau \models \delta (X \rightarrow \text{notEmpty}_{\text{Set}}()) \implies \tau \models v X$
 ⟨proof⟩

lemma $\tau \models \delta (\text{null} \rightarrow \text{notEmpty}_{\text{Set}}())$
 ⟨proof⟩

lemma *OclNotEmpty-infinite*: $\tau \models \delta X \implies \neg \text{finite } \ulcorner \text{Rep-Set}_{\text{base}} (X \ \tau) \urcorner \implies \neg \tau \models \delta (X \rightarrow \text{notEmpty}_{\text{Set}}())$
 ⟨proof⟩

lemma *OclNotEmpty-has-elt* : $\tau \models \delta X \implies$
 $\tau \models X \rightarrow \text{notEmpty}_{\text{Set}}() \implies$
 $\exists e. e \in \ulcorner \text{Rep-Set}_{\text{base}} (X \ \tau) \urcorner$
 ⟨proof⟩

OclANY

lemma *OclANY-defined-args-valid*: $\tau \models \delta (X \rightarrow \text{anySet}()) \implies \tau \models \delta X$
 ⟨proof⟩

lemma $\tau \models \delta X \implies \tau \models X \rightarrow \text{isEmpty}_{\text{Set}}() \implies \neg \tau \models \delta (X \rightarrow \text{anySet}())$
 ⟨proof⟩

lemma *OclANY-valid-args-valid*:
 $(\tau \models v(X \rightarrow \text{anySet}())) = (\tau \models v X)$
 ⟨proof⟩

lemma *OclANY-valid-args-valid'*[simp,code-unfold]:
 $v(X \rightarrow \text{anySet}()) = (v X)$
 ⟨proof⟩

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]: $(\text{invalid} \rightarrow \text{size}_{\text{Set}}()) = \text{invalid}$
 ⟨proof⟩

lemma *OclSize-null*[simp,code-unfold]: $(\text{null} \rightarrow \text{size}_{\text{Set}}()) = \text{invalid}$
 ⟨proof⟩

OclIsEmpty

lemma *OclIsEmpty-invalid*[simp,code-unfold]: $(\text{invalid} \rightarrow \text{isEmpty}_{\text{Set}}()) = \text{invalid}$
 ⟨proof⟩

lemma *OclIsEmpty-null*[simp,code-unfold]: $(\text{null} \rightarrow \text{isEmpty}_{\text{Set}}()) = \text{true}$

<proof>

OclNotEmpty

lemma *OclNotEmpty-invalid*[simp,code-unfold]:(*invalid*→*notEmptySet*()) = *invalid*
<proof>

lemma *OclNotEmpty-null*[simp,code-unfold]:(*null*→*notEmptySet*()) = *false*
<proof>

OclANY

lemma *OclANY-invalid*[simp,code-unfold]:(*invalid*→*anySet*()) = *invalid*
<proof>

lemma *OclANY-null*[simp,code-unfold]:(*null*→*anySet*()) = *null*
<proof>

OclForall

lemma *OclForall-invalid*[simp,code-unfold]:*invalid*→*forAllSet*(*a* | *P a*) = *invalid*
<proof>

lemma *OclForall-null*[simp,code-unfold]:*null*→*forAllSet*(*a* | *P a*) = *invalid*
<proof>

OclExists

lemma *OclExists-invalid*[simp,code-unfold]:*invalid*→*existsSet*(*a* | *P a*) = *invalid*
<proof>

lemma *OclExists-null*[simp,code-unfold]:*null*→*existsSet*(*a* | *P a*) = *invalid*
<proof>

OclIterate

lemma *OclIterate-invalid*[simp,code-unfold]:*invalid*→*iterateSet*(*a*; *x = A* | *P a x*) = *invalid*
<proof>

lemma *OclIterate-null*[simp,code-unfold]:*null*→*iterateSet*(*a*; *x = A* | *P a x*) = *invalid*
<proof>

lemma *OclIterate-invalid-args*[simp,code-unfold]:*S*→*iterateSet*(*a*; *x = invalid* | *P a x*) = *invalid*
<proof>

An open question is this ...

lemma *S*→*iterateSet*(*a*; *x = null* | *P a x*) = *invalid*
<proof>

lemma *OclIterate-infinite*:

assumes *non-finite*: $\tau \models \text{not}(\delta(S \rightarrow \text{size}_{Set}()))$

shows (*OclIterate S A F*) $\tau = \text{invalid } \tau$

<proof>

OclSelect

lemma *OclSelect-invalid*[simp,code-unfold]:*invalid*→*selectSet*(*a* | *P a*) = *invalid*
<proof>

lemma *OclSelect-null*[simp,code-unfold]:*null*→*selectSet*(*a* | *P a*) = *invalid*
<proof>

OclReject

lemma *OclReject-invalid*[simp,code-unfold]: $invalid \rightarrow reject_{Set}(a \mid P a) = invalid$
<proof>

lemma *OclReject-null*[simp,code-unfold]: $null \rightarrow reject_{Set}(a \mid P a) = invalid$
<proof>

Context Passing

lemma *cp-OclIncludes1*:
 $(X \rightarrow includes_{Set}(x)) \tau = (X \rightarrow includes_{Set}(\lambda \cdot. x \tau)) \tau$
<proof>

lemma *cp-OclSize*: $X \rightarrow size_{Set}() \tau = ((\lambda \cdot. X \tau) \rightarrow size_{Set}()) \tau$
<proof>

lemma *cp-OclIsEmpty*: $X \rightarrow isEmpty_{Set}() \tau = ((\lambda \cdot. X \tau) \rightarrow isEmpty_{Set}()) \tau$
<proof>

lemma *cp-OclNotEmpty*: $X \rightarrow notEmpty_{Set}() \tau = ((\lambda \cdot. X \tau) \rightarrow notEmpty_{Set}()) \tau$
<proof>

lemma *cp-OclANY*: $X \rightarrow any_{Set}() \tau = ((\lambda \cdot. X \tau) \rightarrow any_{Set}()) \tau$
<proof>

lemma *cp-OclForall*:
 $(S \rightarrow forall_{Set}(x \mid P x)) \tau = ((\lambda \cdot. S \tau) \rightarrow forall_{Set}(x \mid P (\lambda \cdot. x \tau))) \tau$
<proof>

lemma *cp-OclForall1* [simp,intro!]:
 $cp S \implies cp (\lambda X. ((S X) \rightarrow forall_{Set}(x \mid P x)))$
<proof>

lemma
 $cp (\lambda X St x. P (\lambda \tau. x) X St) \implies cp S \implies cp (\lambda X. (S X) \rightarrow forall_{Set}(x \mid P x X))$
<proof>

lemma
 $cp S \implies$
 $(\bigwedge x. cp(P x)) \implies$
 $cp(\lambda X. ((S X) \rightarrow forall_{Set}(x \mid P x X)))$
<proof>

lemma *cp-OclExists*:
 $(S \rightarrow exists_{Set}(x \mid P x)) \tau = ((\lambda \cdot. S \tau) \rightarrow exists_{Set}(x \mid P (\lambda \cdot. x \tau))) \tau$
<proof>

lemma *cp-OclExists1* [simp,intro!]:
 $cp S \implies cp (\lambda X. ((S X) \rightarrow exists_{Set}(x \mid P x)))$
<proof>

lemma *cp-OclIterate*:
 $(X \rightarrow iterate_{Set}(a; x = A \mid P a x)) \tau =$

$((\lambda -. X \tau) \rightarrow \text{iterate}_{\text{Set}}(a; x = A \mid P a x)) \tau$
 ⟨proof⟩

lemma *cp-OclSelect*: $(X \rightarrow \text{select}_{\text{Set}}(a \mid P a)) \tau =$
 $((\lambda -. X \tau) \rightarrow \text{select}_{\text{Set}}(a \mid P a)) \tau$
 ⟨proof⟩

lemma *cp-OclReject*: $(X \rightarrow \text{reject}_{\text{Set}}(a \mid P a)) \tau = ((\lambda -. X \tau) \rightarrow \text{reject}_{\text{Set}}(a \mid P a)) \tau$
 ⟨proof⟩

lemmas *cp-intro''_{Set}*[*intro!*,*simp*,*code-unfold*] =
 $\text{cp-OclSize} \quad [\text{THEN allI}[\text{THEN allI}[\text{THEN cpI1}], \text{of OclSize}]]$
 $\text{cp-OclIsEmpty} \quad [\text{THEN allI}[\text{THEN allI}[\text{THEN cpI1}], \text{of OclIsEmpty}]]$
 $\text{cp-OclNotEmpty} \quad [\text{THEN allI}[\text{THEN allI}[\text{THEN cpI1}], \text{of OclNotEmpty}]]$
 $\text{cp-OclANY} \quad [\text{THEN allI}[\text{THEN allI}[\text{THEN cpI1}], \text{of OclANY}]]$

Const

lemma *const-OclIncluding*[*simp*,*code-unfold*] :
assumes *const-x* : *const x*
and *const-S* : *const S*
shows *const* ($S \rightarrow \text{including}_{\text{Set}}(x)$)
 ⟨proof⟩

2.9.25. General Algebraic Execution Rules

Execution Rules on Including

lemma *OclIncluding-finite-rep-set* :
assumes *X-def* : $\tau \models \delta X$
and *x-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⟩

lemma *OclIncluding-rep-set*:
assumes *S-def*: $\tau \models \delta S$
shows $\ulcorner \text{Rep-Set}_{\text{base}} (S \rightarrow \text{including}_{\text{Set}}(\lambda \cdot. \ulcorner x \urcorner) \tau) \urcorner = \text{insert } \ulcorner x \urcorner \ulcorner \text{Rep-Set}_{\text{base}} (S \tau) \urcorner$
 ⟨proof⟩

lemma *OclIncluding-notempty-rep-set*:
assumes *X-def*: $\tau \models \delta X$
and *a-val*: $\tau \models v a$
shows $\ulcorner \text{Rep-Set}_{\text{base}} (X \rightarrow \text{including}_{\text{Set}}(a) \tau) \urcorner \neq \{\}$
 ⟨proof⟩

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⟩

lemma *OclIncluding-includes*:
assumes $\tau \models X \rightarrow \text{includes}_{\text{Set}}(x)$
shows $\tau \models X \rightarrow \text{including}_{\text{Set}}(x) \triangleq X$
 ⟨proof⟩

lemma *OclIncluding-commute0* :
assumes *S-def* : $\tau \models \delta S$
and *i-val* : $\tau \models v i$

and $j\text{-val} : \tau \models v j$
shows $\tau \models ((S \text{ :: } (\mathfrak{A}, 'a::\text{null}) \text{ Set}) \rightarrow \text{including}_{\text{Set}}(i) \rightarrow \text{including}_{\text{Set}}(j)) \triangleq$
 $(S \rightarrow \text{including}_{\text{Set}}(j) \rightarrow \text{including}_{\text{Set}}(i))$
 <proof>

lemma *OclIncluding-commute[simp,code-unfold]*:
 $((S \text{ :: } (\mathfrak{A}, 'a::\text{null}) \text{ Set}) \rightarrow \text{including}_{\text{Set}}(i) \rightarrow \text{including}_{\text{Set}}(j)) = (S \rightarrow \text{including}_{\text{Set}}(j) \rightarrow \text{including}_{\text{Set}}(i))$
 <proof>

Execution Rules on Excluding

lemma *OclExcluding-finite-rep-set* :
assumes $X\text{-def} : \tau \models \delta X$
and $x\text{-val} : \tau \models v x$
shows $\text{finite } \ulcorner \text{Rep-Set}_{\text{base}} (X \rightarrow \text{excluding}_{\text{Set}}(x) \tau) \urcorner = \text{finite } \ulcorner \text{Rep-Set}_{\text{base}} (X \tau) \urcorner$
 <proof>

lemma *OclExcluding-rep-set*:
assumes $S\text{-def} : \tau \models \delta S$
shows $\ulcorner \text{Rep-Set}_{\text{base}} (S \rightarrow \text{excluding}_{\text{Set}}(\lambda \cdot \underline{x}_{\perp}) \tau) \urcorner = \ulcorner \text{Rep-Set}_{\text{base}} (S \tau) \urcorner - \{\underline{x}_{\perp}\}$
 <proof>

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>

lemma *OclExcluding-excludes*:
assumes $\tau \models X \rightarrow \text{excludes}_{\text{Set}}(x)$
shows $\tau \models X \rightarrow \text{excluding}_{\text{Set}}(x) \triangleq X$
 <proof>

lemma *OclExcluding-cha0[simp]*:
assumes $\text{val-}x:\tau \models (v x)$
shows $\tau \models ((\text{Set}\{\} \rightarrow \text{excluding}_{\text{Set}}(x)) \triangleq \text{Set}\{\})$
 <proof>

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 $\tau \models ((S \text{ :: } (\mathfrak{A}, 'a::\text{null}) \text{ Set}) \rightarrow \text{excluding}_{\text{Set}}(i) \rightarrow \text{excluding}_{\text{Set}}(j)) \triangleq$
 $(S \rightarrow \text{excluding}_{\text{Set}}(j) \rightarrow \text{excluding}_{\text{Set}}(i))$
 <proof>

lemma *OclExcluding-commute[simp,code-unfold]*:
 $((S \text{ :: } (\mathfrak{A}, 'a::\text{null}) \text{ Set}) \rightarrow \text{excluding}_{\text{Set}}(i) \rightarrow \text{excluding}_{\text{Set}}(j)) = (S \rightarrow \text{excluding}_{\text{Set}}(j) \rightarrow \text{excluding}_{\text{Set}}(i))$
 <proof>

lemma *OclExcluding-cha0-exec[simp,code-unfold]*:
 $(\text{Set}\{\} \rightarrow \text{excluding}_{\text{Set}}(x)) = (\text{if } (v x) \text{ then } \text{Set}\{\} \text{ else } \text{invalid endif})$
 <proof>

lemma *OclExcluding-cha1*:
assumes $\text{def-}X:\tau \models (\delta X)$

and $val\text{-}x:\tau \models (v\ x)$
and $val\text{-}y:\tau \models (v\ y)$
and $neq : \tau \models not(x \triangleq y)$
shows $\tau \models ((X \rightarrow including_{Set}(x)) \rightarrow excluding_{Set}(y)) \triangleq ((X \rightarrow excluding_{Set}(y)) \rightarrow including_{Set}(x))$
 <proof>

lemma *OclExcluding-charn2*:

assumes $def\text{-}X:\tau \models (\delta\ X)$
and $val\text{-}x:\tau \models (v\ x)$
shows $\tau \models (((X \rightarrow including_{Set}(x)) \rightarrow excluding_{Set}(x)) \triangleq (X \rightarrow excluding_{Set}(x)))$
 <proof>

theorem *OclExcluding-charn3*: $((X \rightarrow including_{Set}(x)) \rightarrow excluding_{Set}(x)) = (X \rightarrow excluding_{Set}(x))$
 <proof>

One would like a generic theorem of the form:

lemma *OclExcluding_charn_exec*:

$"(X \rightarrow including_{Set}(x::('A, 'a::null)val)) \rightarrow excluding_{Set}(y)) =$
 $(if\ \delta\ X\ then\ if\ x \doteq y$
 $\quad then\ X \rightarrow excluding_{Set}(y)$
 $\quad else\ X \rightarrow excluding_{Set}(y) \rightarrow including_{Set}(x)$
 $\quad endif$
 $else\ invalid\ endif)"$

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

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

lemma *OclExcluding-charn-exec*:

assumes $strict1: (invalid \doteq y) = invalid$
and $strict2: (x \doteq invalid) = invalid$
and $StrictRefEq\text{-}valid\text{-}args\text{-}valid: \bigwedge (x::('A, 'a::null)val)\ y\ \tau.$
 $(\tau \models \delta\ (x \doteq y)) = ((\tau \models (v\ x)) \wedge (\tau \models v\ y))$
and $cp\text{-}StrictRefEq: \bigwedge (X::('A, 'a::null)val)\ Y\ \tau. (X \doteq Y)\ \tau = ((\lambda\cdot. X\ \tau) \doteq (\lambda\cdot. Y\ \tau))\ \tau$
and $StrictRefEq\text{-}vs\text{-}StrongEq: \bigwedge (x::('A, 'a::null)val)\ y\ \tau.$
 $\tau \models v\ x \implies \tau \models v\ y \implies (\tau \models ((x \doteq y) \triangleq (x \triangleq y)))$
shows $(X \rightarrow including_{Set}(x::('A, 'a::null)val)) \rightarrow excluding_{Set}(y)) =$
 $(if\ \delta\ X\ then\ if\ x \doteq y$
 $\quad then\ X \rightarrow excluding_{Set}(y)$
 $\quad else\ X \rightarrow excluding_{Set}(y) \rightarrow including_{Set}(x)$
 $\quad endif$
 $else\ invalid\ endif)$
 <proof>

schematic-lemma *OclExcluding-charn-execInteger[simp,code-unfold]*: $?X$
 <proof>

schematic-lemma *OclExcluding-charn-exec_{Boolean}*[simp,code-unfold]: ?X
 ⟨proof⟩

schematic-lemma *OclExcluding-charn-exec_{Set}*[simp,code-unfold]: ?X
 ⟨proof⟩

Execution Rules on Includes

lemma *OclIncludes-charn0*[simp]:
assumes *val-x:τ* ⊨ (v x)
shows τ ⊨ not(*Set*{}->*includes_{Set}*(x))
 ⟨proof⟩

lemma *OclIncludes-charn0'*[simp,code-unfold]:
Set{}->*includes_{Set}*(x) = (if v x then false else invalid endif)
 ⟨proof⟩

lemma *OclIncludes-charn1*:
assumes *def-X:τ* ⊨ (δ X)
assumes *val-x:τ* ⊨ (v x)
shows τ ⊨ (X->*including_{Set}*(x)->*includes_{Set}*(x))
 ⟨proof⟩

lemma *OclIncludes-charn2*:
assumes *def-X:τ* ⊨ (δ X)
and *val-x:τ* ⊨ (v x)
and *val-y:τ* ⊨ (v y)
and *neq* :τ ⊨ not(x ≐ y)
shows τ ⊨ (X->*including_{Set}*(x)->*includes_{Set}*(y)) ≐ (X->*includes_{Set}*(y))
 ⟨proof⟩

Here is again a generic theorem similar as above.

lemma *OclIncludes-execute-generic*:
assumes *strict1*: (invalid ≐ y) = invalid
and *strict2*: (x ≐ invalid) = invalid
and *cp-StrictRefEq*: $\bigwedge (X::('A,'a::null)val) Y \tau. (X \doteq Y) \tau = ((\lambda-. X \tau) \doteq (\lambda-. Y \tau)) \tau$
and *StrictRefEq-vs-StrongEq*: $\bigwedge (x::('A,'a::null)val) y \tau. \tau \models v x \implies \tau \models v y \implies (\tau \models ((x \doteq y) \triangleq (x \triangleq y)))$
shows
 (X->*including_{Set}*(x::('A,'a::null)val)->*includes_{Set}*(y)) =
 (if δ X then if x ≐ y then true else X->*includes_{Set}*(y) endif else invalid endif)
 ⟨proof⟩

schematic-lemma *OclIncludes-execute_{Integer}*[simp,code-unfold]: ?X
 ⟨proof⟩

schematic-lemma *OclIncludes-execute_{Boolean}*[simp,code-unfold]: ?X
 ⟨proof⟩

schematic-lemma *OclIncludes-execute_{Set}*[simp,code-unfold]: ?X

<proof>

lemma *OclIncludes-including-generic* :

assumes *OclIncludes-execute-generic* [simp] : $\bigwedge X x y.$
 $(X \rightarrow \text{including}_{Set}(x::('A, 'a::\text{null})\text{val}) \rightarrow \text{includes}_{Set}(y)) =$
 (if δX then if $x \doteq y$ then true else $X \rightarrow \text{includes}_{Set}(y)$ 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>

lemmas *OclIncludes-includingInteger* =

OclIncludes-including-generic[OF *OclIncludes-executeInteger 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>

Execution Rules on Size

lemma [simp,code-unfold]: $Set\{\} \rightarrow \text{size}_{Set}() = \mathbf{0}$

<proof>

lemma *OclSize-including-exec*[simp,code-unfold]:

$((X \rightarrow \text{including}_{Set}(x)) \rightarrow \text{size}_{Set}()) = (\text{if } \delta X \text{ and } v x \text{ then}$
 $X \rightarrow \text{size}_{Set}() +_{int} \text{if } X \rightarrow \text{includes}_{Set}(x) \text{ then } \mathbf{0} \text{ else } \mathbf{1} \text{ endif}$
 else
 invalid
 endif)

<proof>

Execution Rules on IsEmpty

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

<proof>

lemma *OclIsEmpty-including* [simp]:

assumes *X-def*: $\tau \models \delta X$
 and *X-finite*: $\text{finite } \ulcorner \text{Rep-Set}_{base} (X \tau) \urcorner$
 and *a-val*: $\tau \models v a$
shows $X \rightarrow \text{including}_{Set}(a) \rightarrow \text{isEmpty}_{Set}() \tau = \text{false } \tau$
<proof>

Execution Rules on NotEmpty

lemma [simp,code-unfold]: $Set\{\} \rightarrow \text{notEmpty}_{Set}() = \text{false}$

<proof>

lemma *OclNotEmpty-including* [simp,code-unfold]:

assumes *X-def*: $\tau \models \delta X$
 and *X-finite*: $\text{finite } \ulcorner \text{Rep-Set}_{base} (X \tau) \urcorner$
 and *a-val*: $\tau \models v a$

shows $X \rightarrow \text{including}_{Set}(a) \rightarrow \text{notEmpty}_{Set}() \tau = \text{true} \tau$
 ⟨proof⟩

Execution Rules on Any

lemma $[\text{simp}, \text{code-unfold}]$: $Set\{\} \rightarrow \text{any}_{Set}() = \text{null}$
 ⟨proof⟩

lemma $OclANY\text{-singleton-exec}[\text{simp}, \text{code-unfold}]$:
 $(Set\{\} \rightarrow \text{including}_{Set}(a)) \rightarrow \text{any}_{Set}() = a$
 ⟨proof⟩

Execution Rules on Forall

lemma $OclForall\text{-mtSet-exec}[\text{simp}, \text{code-unfold}]$: $((Set\{\}) \rightarrow \text{forAll}_{Set}(z \mid P(z))) = \text{true}$
 ⟨proof⟩

The following rule is a main theorem of our approach: From a denotational definition that assures consistency, but may be — as in the case of the $OclForall X P$ — dauntingly complex, we derive operational rules that can serve as a gold-standard for operational execution, since they may be evaluated in whatever situation and according to whatever strategy. In the case of $OclForall X P$, the operational rule gives immediately a way to evaluation in any finite (in terms of conventional OCL: denotable) set, although the rule also holds for the infinite case:

$Integer_{null} \rightarrow \text{forAll}_{Set}(x \mid Integer_{null} \rightarrow \text{forAll}_{Set}(y \mid x +_{int} y \stackrel{\Delta}{=} y +_{int} x))$
 or even:
 $Integer \rightarrow \text{forAll}_{Set}(x \mid Integer \rightarrow \text{forAll}_{Set}(y \mid x +_{int} y \stackrel{\dot{=}}{=} y +_{int} x))$
 are valid OCL statements in any context τ .

theorem $OclForall\text{-including-exec}[\text{simp}, \text{code-unfold}]$:
assumes $cp0 : cp P$
shows $((S \rightarrow \text{including}_{Set}(x)) \rightarrow \text{forAll}_{Set}(z \mid P(z))) = (\text{if } \delta S \text{ and } v x$
 $\text{then } P x \text{ and } (S \rightarrow \text{forAll}_{Set}(z \mid P(z)))$
 else invalid
 $\text{endif})$

⟨proof⟩

Execution Rules on Exists

lemma $OclExists\text{-mtSet-exec}[\text{simp}, \text{code-unfold}]$:
 $((Set\{\}) \rightarrow \text{exists}_{Set}(z \mid P(z))) = \text{false}$
 ⟨proof⟩

lemma $OclExists\text{-including-exec}[\text{simp}, \text{code-unfold}]$:
assumes $cp : cp P$
shows $((S \rightarrow \text{including}_{Set}(x)) \rightarrow \text{exists}_{Set}(z \mid P(z))) = (\text{if } \delta S \text{ and } v x$
 $\text{then } P x \text{ or } (S \rightarrow \text{exists}_{Set}(z \mid P(z)))$
 else invalid
 $\text{endif})$

⟨proof⟩

Execution Rules on Iterate

lemma $OclIterate\text{-empty}[\text{simp}, \text{code-unfold}]$: $((Set\{\}) \rightarrow \text{iterate}_{Set}(a; x = A \mid P a x)) = A$
 ⟨proof⟩

In particular, this does hold for $A = \text{null}$.

lemma $OclIterate\text{-including}$:
assumes $S\text{-finite}$: $\tau \models \delta(S \rightarrow \text{size}_{Set}())$

and $F\text{-valid-arg}$: $(v A) \tau = (v (F a A)) \tau$
and $F\text{-commute}$: $\text{comp-fun-commute } F$
and $F\text{-cp}$: $\bigwedge x y \tau. F x y \tau = F (\lambda -. x \tau) y \tau$
shows $((S \rightarrow \text{including}_{Set}(a)) \rightarrow \text{iterate}_{Set}(a; x = A \mid F a x)) \tau =$
 $((S \rightarrow \text{excluding}_{Set}(a)) \rightarrow \text{iterate}_{Set}(a; x = F a A \mid F a x)) \tau$
 $\langle \text{proof} \rangle$

Execution Rules on Select

lemma $OclSelect\text{-}mtSet\text{-}exec[simp,code\text{-}unfold]$: $OclSelect\ mtSet\ P = mtSet$
 $\langle \text{proof} \rangle$

definition $OclSelect\text{-}body :: - \Rightarrow - \Rightarrow - \Rightarrow ('a, 'a\ option\ option)\ Set$
 $\equiv (\lambda P x\ acc. \text{if } P\ x \doteq \text{false then } acc\ \text{else } acc \rightarrow \text{including}_{Set}(x)\ \text{endif})$

theorem $OclSelect\text{-}including\text{-}exec[simp,code\text{-}unfold]$:

assumes $P\text{-cp} : cp\ P$
shows $OclSelect\ (X \rightarrow \text{including}_{Set}(y))\ P = OclSelect\text{-}body\ P\ y\ (OclSelect\ (X \rightarrow \text{excluding}_{Set}(y))\ P)$
 $(\text{is } - = ?select)$
 $\langle \text{proof} \rangle$

Execution Rules on Reject

lemma $OclReject\text{-}mtSet\text{-}exec[simp,code\text{-}unfold]$: $OclReject\ mtSet\ P = mtSet$
 $\langle \text{proof} \rangle$

lemma $OclReject\text{-}including\text{-}exec[simp,code\text{-}unfold]$:

assumes $P\text{-cp} : cp\ P$
shows $OclReject\ (X \rightarrow \text{including}_{Set}(y))\ P = OclSelect\text{-}body\ (not\ o\ P)\ y\ (OclReject\ (X \rightarrow \text{excluding}_{Set}(y))\ P)$
 $\langle \text{proof} \rangle$

Execution Rules Combining Previous Operators

OclIncluding

lemma $OclIncluding\text{-}idem0$:

assumes $\tau \models \delta\ S$
and $\tau \models v\ i$
shows $\tau \models (S \rightarrow \text{including}_{Set}(i)) \rightarrow \text{including}_{Set}(i) \triangleq (S \rightarrow \text{including}_{Set}(i))$
 $\langle \text{proof} \rangle$

theorem $OclIncluding\text{-}idem[simp,code\text{-}unfold]$: $((S :: ('a, 'a::null)\ Set) \rightarrow \text{including}_{Set}(i)) \rightarrow \text{including}_{Set}(i) =$
 $(S \rightarrow \text{including}_{Set}(i))$

$\langle \text{proof} \rangle$

OclExcluding

lemma $OclExcluding\text{-}idem0$:

assumes $\tau \models \delta\ S$
and $\tau \models v\ i$
shows $\tau \models (S \rightarrow \text{excluding}_{Set}(i)) \rightarrow \text{excluding}_{Set}(i) \triangleq (S \rightarrow \text{excluding}_{Set}(i))$
 $\langle \text{proof} \rangle$

theorem $OclExcluding\text{-}idem[simp,code\text{-}unfold]$: $((S \rightarrow \text{excluding}_{Set}(i)) \rightarrow \text{excluding}_{Set}(i)) =$
 $(S \rightarrow \text{excluding}_{Set}(i))$

$\langle \text{proof} \rangle$

OclIncludes

lemma *OclIncludes-any*[*simp,code-unfold*]:
 $X \rightarrow \text{includes}_{Set}(X \rightarrow \text{any}_{Set}()) = (\text{if } \delta X \text{ then}$
 $\quad \text{if } \delta (X \rightarrow \text{size}_{Set}()) \text{ then not}(X \rightarrow \text{isEmpty}_{Set}())$
 $\quad \text{else } X \rightarrow \text{includes}_{Set}(\text{null}) \text{ endif}$
 $\text{else invalid endif})$

<proof>

OclSize

lemma [*simp,code-unfold*]: $\delta (Set\{\} \rightarrow \text{size}_{Set}()) = \text{true}$
<proof>

lemma [*simp,code-unfold*]: $\delta ((X \rightarrow \text{including}_{Set}(x)) \rightarrow \text{size}_{Set}()) = (\delta(X \rightarrow \text{size}_{Set}()) \text{ and } v(x))$
<proof>

lemma [*simp,code-unfold*]: $\delta ((X \rightarrow \text{excluding}_{Set}(x)) \rightarrow \text{size}_{Set}()) = (\delta(X \rightarrow \text{size}_{Set}()) \text{ and } v(x))$
<proof>

lemma [*simp*]:
assumes $X\text{-finite}: \bigwedge \tau. \text{finite } \ulcorner \text{Rep-Set}_{base} (X \tau) \urcorner$
shows $\delta ((X \rightarrow \text{including}_{Set}(x)) \rightarrow \text{size}_{Set}()) = (\delta(X) \text{ and } v(x))$
<proof>

OclForall

lemma *OclForall-rep-set-false*:
assumes $\tau \models \delta X$
shows $(\text{OclForall } X P \tau = \text{false } \tau) = (\exists x \in \ulcorner \text{Rep-Set}_{base} (X \tau) \urcorner. P (\lambda \tau. x) \tau = \text{false } \tau)$
<proof>

lemma *OclForall-rep-set-true*:
assumes $\tau \models \delta X$
shows $(\tau \models \text{OclForall } X P) = (\forall x \in \ulcorner \text{Rep-Set}_{base} (X \tau) \urcorner. \tau \models P (\lambda \tau. x))$
<proof>

lemma *OclForall-includes* :
assumes $x\text{-def} : \tau \models \delta x$
and $y\text{-def} : \tau \models \delta y$
shows $(\tau \models \text{OclForall } x (\text{OclIncludes } y)) = (\ulcorner \text{Rep-Set}_{base} (x \tau) \urcorner \subseteq \ulcorner \text{Rep-Set}_{base} (y \tau) \urcorner)$
<proof>

lemma *OclForall-not-includes* :
assumes $x\text{-def} : \tau \models \delta x$
and $y\text{-def} : \tau \models \delta y$
shows $(\text{OclForall } x (\text{OclIncludes } y) \tau = \text{false } \tau) = (\neg \ulcorner \text{Rep-Set}_{base} (x \tau) \urcorner \subseteq \ulcorner \text{Rep-Set}_{base} (y \tau) \urcorner)$
<proof>

lemma *OclForall-iterate*:
assumes $S\text{-finite}: \text{finite } \ulcorner \text{Rep-Set}_{base} (S \tau) \urcorner$
shows $S \rightarrow \text{forAll}_{Set}(x \mid P x) \tau = (S \rightarrow \text{iterates}_{Set}(x; \text{acc} = \text{true} \mid \text{acc and } P x)) \tau$
<proof>

lemma *OclForall-cong*:
assumes $\bigwedge x. x \in \ulcorner \text{Rep-Set}_{base} (X \tau) \urcorner \implies \tau \models P (\lambda \tau. x) \implies \tau \models Q (\lambda \tau. x)$
assumes $P: \tau \models \text{OclForall } X P$
shows $\tau \models \text{OclForall } X Q$
<proof>

lemma *OclForall-cong'*:

assumes $\bigwedge x. x \in {}^\top \text{Rep-Set}_{base} (X \ \tau)^\top \implies \tau \models P (\lambda \tau. x) \implies \tau \models Q (\lambda \tau. x) \implies \tau \models R (\lambda \tau. x)$

assumes $P: \tau \models \text{OclForall } X \ P$

assumes $Q: \tau \models \text{OclForall } X \ Q$

shows $\tau \models \text{OclForall } X \ R$

<proof>

Strict Equality

lemma *StrictRefEqSet-defined* :

assumes $x\text{-def}: \tau \models \delta \ x$

assumes $y\text{-def}: \tau \models \delta \ y$

shows $((x::(\mathfrak{A}, \alpha::\text{null})\text{Set}) \doteq y) \ \tau =$

$(x \rightarrow \text{forAll}_{Set}(z) \ y \rightarrow \text{includes}_{Set}(z)) \ \text{and} \ (y \rightarrow \text{forAll}_{Set}(z) \ x \rightarrow \text{includes}_{Set}(z))) \ \tau$

<proof>

lemma *StrictRefEqSet-exec[simp,code-unfold]* :

$((x::(\mathfrak{A}, \alpha::\text{null})\text{Set}) \doteq y) =$

$(\text{if } \delta \ x \ \text{then} \ (\text{if } \delta \ y$

$\text{then} \ ((x \rightarrow \text{forAll}_{Set}(z) \ y \rightarrow \text{includes}_{Set}(z)) \ \text{and} \ (y \rightarrow \text{forAll}_{Set}(z) \ x \rightarrow \text{includes}_{Set}(z))))$

$\text{else if } v \ y$

$\text{then false} \ (* \ x' \rightarrow \text{includes} = \text{null} \ *)$

else invalid

endif

$\text{endif})$

$\text{else if } v \ x \ (* \ \text{null} = ??? \ *)$

$\text{then if } v \ y \ \text{then not}(\delta \ y) \ \text{else invalid endif}$

else invalid

endif

$\text{endif})$

<proof>

lemma *StrictRefEqSet-L-subst1* : $cp \ P \implies \tau \models v \ x \implies \tau \models v \ y \implies \tau \models v \ P \ x \implies \tau \models v \ P \ y \implies$

$\tau \models (x::(\mathfrak{A}, \alpha::\text{null})\text{Set}) \doteq y \implies \tau \models (P \ x :: (\mathfrak{A}, \alpha::\text{null})\text{Set}) \doteq P \ y$

<proof>

lemma *OclIncluding-cong'* :

shows $\tau \models \delta \ s \implies \tau \models \delta \ t \implies \tau \models v \ x \implies$

$\tau \models ((s::(\mathfrak{A}, \alpha::\text{null})\text{Set}) \doteq t) \implies \tau \models (s \rightarrow \text{including}_{Set}(x) \doteq (t \rightarrow \text{including}_{Set}(x)))$

<proof>

lemma *OclIncluding-cong* : $\bigwedge (s::(\mathfrak{A}, \alpha::\text{null})\text{Set}) \ t \ x \ y \ \tau. \ \tau \models \delta \ t \implies \tau \models v \ y \implies$

$\tau \models s \doteq t \implies x = y \implies \tau \models s \rightarrow \text{including}_{Set}(x) \doteq (t \rightarrow \text{including}_{Set}(y))$

<proof>

lemma *const-StrictRefEqSet-empty* : $\text{const } X \implies \text{const } (X \doteq \text{Set}\{\})$

<proof>

lemma *const-StrictRefEqSet-including* :

$\text{const } a \implies \text{const } S \implies \text{const } X \implies \text{const } (X \doteq S \rightarrow \text{including}_{Set}(a))$

<proof>

2.9.26. Test Statements

Assert $(\tau \models (\text{Set}\{\lambda \cdot. \underline{\underline{x}}\} \doteq \text{Set}\{\lambda \cdot. \underline{\underline{x}}\}))$

Assert $(\tau \models (\text{Set}\{\lambda \cdot. \underline{x}\} \doteq \text{Set}\{\lambda \cdot. \underline{x}\}))$

```

instantiation Set_base :: (equal)equal
begin
  definition HOL.equal k l  $\longleftrightarrow$  (k::('a::equal)Set_base) = l
  instance ⟨proof⟩
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
  ⟨proof⟩

Assert  $\tau \models (\text{Set}\{\} \doteq \text{Set}\{\})$ 
Assert  $\tau \models (\text{Set}\{\mathbf{1},\mathbf{2}\} \doteq \text{Set}\{\} \rightarrow \text{including}_{\text{Set}}(\mathbf{2}) \rightarrow \text{including}_{\text{Set}}(\mathbf{1}))$ 
Assert  $\tau \models (\text{Set}\{\mathbf{1},\text{invalid},\mathbf{2}\} \doteq \text{invalid})$ 
Assert  $\tau \models (\text{Set}\{\mathbf{1},\mathbf{2}\} \rightarrow \text{including}_{\text{Set}}(\text{null}) \doteq \text{Set}\{\text{null},\mathbf{1},\mathbf{2}\})$ 
Assert  $\tau \models (\text{Set}\{\mathbf{1},\mathbf{2}\} \rightarrow \text{including}_{\text{Set}}(\text{null}) \doteq \text{Set}\{\mathbf{1},\mathbf{2},\text{null}\})$ 

```

end

```

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

```

```

no-notation None ( $\perp$ )

```

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:  $\tau \models (\delta X) \implies \forall x \in \text{set } \ulcorner \text{Rep-Sequence}_{\text{base}} (X \ \tau) \urcorner. x \neq \text{bot}$ 
  ⟨proof⟩

```

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:

```

defs (overloaded) StrictRefEqSeq :
  ((x::('a,'α::null)Sequence)  $\doteq$  y)  $\equiv$  ( $\lambda \tau. \text{if } (v \ x) \ \tau = \text{true} \ \tau \wedge (v \ y) \ \tau = \text{true} \ \tau$ 
    then (x  $\doteq$  y) $\tau$ 
    else invalid  $\tau$ )

```

One might object here that for the case of objects, this is an empty definition. The answer is no, we will restrain later on states and objects such that any object has its oid stored inside the object (so the ref, under which an object can be referenced in the store will be represented in the object itself). For

such well-formed stores that satisfy this invariant (the WFF-invariant), the referential equality and the strong equality—and therefore the strict equality on sequences in the sense above—coincides.

Property proof in terms of $profile-bin_{StrongEq^{-v^{-v}}}$

interpretation $StrictRefEqSeq : profile-bin_{StrongEq^{-v^{-v}}} \lambda x y. (x :: ('\mathfrak{A}, '\alpha :: null) Sequence) \doteq y$
 ⟨proof⟩

2.10.3. Constants: mtSequence

definition $mtSequence :: ('\mathfrak{A}, '\alpha :: null) Sequence (Sequence\{\})$
where $Sequence\{\} \equiv (\lambda \tau. Abs-Sequence_{base} \perp \perp :: '\alpha list_{\perp})$

lemma $mtSequence-defined[simp, code-unfold]: \delta(Sequence\{\}) = true$
 ⟨proof⟩

lemma $mtSequence-valid[simp, code-unfold]: v(Sequence\{\}) = true$
 ⟨proof⟩

lemma $mtSequence-rep-set: \ulcorner Rep-Sequence_{base} (Sequence\{\}) \tau \urcorner = []$
 ⟨proof⟩ **lemma** $[simp, code-unfold]: const Sequence\{\}$
 ⟨proof⟩

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

2.10.4. Definition: Prepend

definition $OclPrepend :: [('\mathfrak{A}, '\alpha :: null) Sequence, ('\mathfrak{A}, '\alpha) val] \Rightarrow ('\mathfrak{A}, '\alpha) Sequence$
where $OclPrepend x y = (\lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (v y) \tau = true \tau$
 $\text{then } Abs-Sequence_{base} \perp \perp (y \tau) \# \ulcorner Rep-Sequence_{base} (x \tau) \urcorner$
 $\text{else } invalid \tau)$
notation $OclPrepend \ (->prepend_{Seq} '(-'))$

interpretation $OclPrepend: profile-bin_{d^{-v}} OclPrepend \lambda x y. Abs-Sequence_{base} \perp \perp y \# \ulcorner Rep-Sequence_{base} x \urcorner$
 ⟨proof⟩

syntax
 $-OclFinsequence :: args \Rightarrow ('\mathfrak{A}, 'a :: null) Sequence (Sequence\{-\})$

translations
 $Sequence\{x, xs\} == CONST OclPrepend (Sequence\{xs\}) x$
 $Sequence\{x\} == CONST OclPrepend (Sequence\{\}) x$

2.10.5. Definition: Including

definition $OclIncluding :: [('\mathfrak{A}, '\alpha :: null) Sequence, ('\mathfrak{A}, '\alpha) val] \Rightarrow ('\mathfrak{A}, '\alpha) Sequence$
where $OclIncluding x y = (\lambda \tau. \text{if } (\delta x) \tau = true \tau \wedge (v y) \tau = true \tau$
 $\text{then } Abs-Sequence_{base} \perp \perp \ulcorner Rep-Sequence_{base} (x \tau) \urcorner @ [y \tau] \perp$
 $\text{else } invalid \tau)$
notation $OclIncluding \ (->including_{Seq} '(-'))$

interpretation $OclIncluding :$
 $profile-bin_{d^{-v}} OclIncluding \lambda x y. Abs-Sequence_{base} \perp \perp \ulcorner Rep-Sequence_{base} x \urcorner @ [y] \perp$
 ⟨proof⟩

lemma $[simp, code-unfold] : (Sequence\{\} \rightarrow including_{Seq}(a)) = (Sequence\{\} \rightarrow prepend_{Seq}(a))$
 ⟨proof⟩

| $x \# - \Rightarrow x$

else invalid τ)

notation $OclFirst$ ($->first_{Seq}'(-)$)

2.10.11. Definition: Last

definition $OclLast$:: [$(\mathfrak{A}, \alpha :: null)$ Sequence] \Rightarrow (\mathfrak{A}, α) val

where $OclLast$ $x = (\lambda \tau. \text{if } (\delta x) \tau = true \tau \text{ then}$
if $\ulcorner Rep-Sequence_{base} (x \tau) \urcorner = []$ then
invalid τ
else
last $\ulcorner Rep-Sequence_{base} (x \tau) \urcorner$
else invalid τ)

notation $OclLast$ ($->last_{Seq}'(-)$)

2.10.12. Definition: Iterate

definition $OclIterate$:: [$(\mathfrak{A}, \alpha :: null)$ Sequence, $(\mathfrak{A}, \beta :: null)$ val,
 $(\mathfrak{A}, \alpha) val \Rightarrow (\mathfrak{A}, \beta) val \Rightarrow (\mathfrak{A}, \beta) val \Rightarrow (\mathfrak{A}, \beta) val$

where $OclIterate$ $S A F = (\lambda \tau. \text{if } (\delta S) \tau = true \tau \wedge (v A) \tau = true \tau$
then (foldr (F) (map $(\lambda a \tau. a) \ulcorner Rep-Sequence_{base} (S \tau) \urcorner$))(A) τ
else \perp)

syntax

- $OclIterateSeq$:: [$(\mathfrak{A}, \alpha :: null)$ Sequence, idt, idt, α, β] \Rightarrow (\mathfrak{A}, γ) val
($->iterate_{Seq}'(-;=- | -)$)

translations

$X->iterate_{Seq}(a; x = A | P) == CONST OclIterate X A (\%a. (\% x. P))$

2.10.13. Definition: Forall

definition $OclForall$:: [$(\mathfrak{A}, \alpha :: null)$ Sequence, $(\mathfrak{A}, \alpha) val \Rightarrow (\mathfrak{A}) Boolean$] \Rightarrow $\mathfrak{A} Boolean$

where $OclForall$ $S P = (S->iterate_{Seq}(b; x = true | x \text{ and } (P b)))$

syntax

- $OclForallSeq$:: [$(\mathfrak{A}, \alpha :: null)$ Sequence, $id, (\mathfrak{A}) Boolean$] \Rightarrow $\mathfrak{A} Boolean$ ($(-)->forAll_{Seq}'(-|-)$)

translations

$X->forAll_{Seq}(x | P) == CONST UML-Sequence.OclForall X (\%x. P)$

2.10.14. Definition: Exists

definition $OclExists$:: [$(\mathfrak{A}, \alpha :: null)$ Sequence, $(\mathfrak{A}, \alpha) val \Rightarrow (\mathfrak{A}) Boolean$] \Rightarrow $\mathfrak{A} Boolean$

where $OclExists$ $S P = (S->iterate_{Seq}(b; x = false | x \text{ or } (P b)))$

syntax

- $OclExistsSeq$:: [$(\mathfrak{A}, \alpha :: null)$ Sequence, $id, (\mathfrak{A}) Boolean$] \Rightarrow $\mathfrak{A} Boolean$ ($(-)->exists_{Seq}'(-|-)$)

translations

$X->exists_{Seq}(x | P) == CONST OclExists X (\%x. P)$

2.10.15. Definition: Collect

definition $OclCollect$:: [$(\mathfrak{A}, \alpha :: null)$ Sequence, $(\mathfrak{A}, \alpha) val \Rightarrow (\mathfrak{A}, \beta) val \Rightarrow (\mathfrak{A}, \beta :: null)$ Sequence

where $OclCollect$ $S P = (S->iterate_{Seq}(b; x = Sequence\{\} | x->prepend_{Seq}(P b)))$

syntax

- $OclCollectSeq$:: [$(\mathfrak{A}, \alpha :: null)$ Sequence, $id, (\mathfrak{A}) Boolean$] \Rightarrow $\mathfrak{A} Boolean$ ($(-)->collect_{Seq}'(-|-)$)

translations

$X->collect_{Seq}(x | P) == CONST OclCollect X (\%x. P)$

2.10.16. Definition: Select

definition *OclSelect* :: [$(\mathfrak{A}, \alpha::\text{null})$ Sequence, $(\mathfrak{A}, \alpha) \text{val} \Rightarrow (\mathfrak{A}) \text{Boolean}$] $\Rightarrow (\mathfrak{A}, \alpha::\text{null})$ Sequence
where *OclSelect* $S P =$
 $(S \rightarrow \text{iterate}_{Seq}(b; x = \text{Sequence}\{\} \mid \text{if } P \text{ b then } x \rightarrow \text{prepend}_{Seq}(b) \text{ else } x \text{ endif}))$

syntax

$\text{-OclSelectSeq} :: [(\mathfrak{A}, \alpha::\text{null}) \text{ Sequence}, \text{id}, (\mathfrak{A}) \text{Boolean}] \Rightarrow \mathfrak{A} \text{ Boolean } ((-)\rightarrow \text{select}_{Seq}'(-))$

translations

$X \rightarrow \text{select}_{Seq}(x \mid P) == \text{CONST UML-Sequence.OclSelect } X \text{ } (\%x. P)$

2.10.17. Definition: Size

definition *OclSize* :: [$(\mathfrak{A}, \alpha::\text{null})$ Sequence] $\Rightarrow (\mathfrak{A})$ Integer $((-)\rightarrow \text{size}_{Seq}'('))$
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* $(\rightarrow \text{isEmpty}_{Seq}'('))$

2.10.19. Definition: NotEmpty

definition *OclNotEmpty* :: $(\mathfrak{A}, \alpha::\text{null})$ Sequence $\Rightarrow \mathfrak{A}$ Boolean
where *OclNotEmpty* $x = \text{not}(\text{OclIsEmpty } x)$
notation *OclNotEmpty* $(\rightarrow \text{notEmpty}_{Seq}'('))$

2.10.20. Definition: Any

definition *OclANY* $x = (\lambda \tau.$
 $\text{if } x \ \tau = \text{invalid } \tau \text{ then}$
 \perp
 else
 $\text{case drop (drop (Rep-Sequence}_{base} (x \ \tau))) \text{ of } [] \Rightarrow \perp$
 $\mid l \Rightarrow \text{hd } l)$
notation *OclANY* $(\rightarrow \text{any}_{Seq}'('))$

2.10.21. Definition (future operators)

consts

OclCount :: [$(\mathfrak{A}, \alpha::\text{null})$ Sequence, (\mathfrak{A}, α) Sequence] $\Rightarrow \mathfrak{A}$ Integer

OclSum :: $(\mathfrak{A}, \alpha::\text{null})$ Sequence $\Rightarrow \mathfrak{A}$ Integer

notation *OclCount* $(\rightarrow \text{count}_{Seq}'('))$

notation *OclSum* $(\rightarrow \text{sum}_{Seq}'('))$

2.10.22. Logical Properties

2.10.23. Execution Laws with Invalid or Null as Argument

OclIterate

lemma *OclIterate-invalid[simp,code-unfold]:invalid* $\rightarrow \text{iterate}_{Seq}(a; x = A \mid P \ a \ x) = \text{invalid}$

<proof>

lemma *OclIterate-null*[*simp,code-unfold*]: $\text{null} \rightarrow \text{iterate}_{Seq}(a; x = A \mid P a x) = \text{invalid}$
<proof>

lemma *OclIterate-invalid-args*[*simp,code-unfold*]: $S \rightarrow \text{iterate}_{Seq}(a; x = \text{invalid} \mid P a x) = \text{invalid}$
<proof>

Context Passing

lemma *cp-OclIncluding*:
 $(X \rightarrow \text{including}_{Seq}(x)) \tau = ((\lambda -. X \tau) \rightarrow \text{including}_{Seq}(\lambda -. x \tau)) \tau$
<proof>

lemma *cp-OclIterate*:
 $(X \rightarrow \text{iterate}_{Seq}(a; x = A \mid P a x)) \tau =$
 $((\lambda -. X \tau) \rightarrow \text{iterate}_{Seq}(a; x = A \mid P a x)) \tau$
<proof>

lemmas *cp-intro''_{Seq}*[*intro!,simp,code-unfold*] =
cp-OclIncluding [THEN *allU*[THEN *allU*[THEN *allU*[THEN *cpI2*]], of *OclIncluding*]]

Const

2.10.24. General Algebraic Execution Rules

Execution Rules on Iterate

lemma *OclIterate-empty*[*simp,code-unfold*]: $\text{Sequence}\{\} \rightarrow \text{iterate}_{Seq}(a; x = A \mid P a x) = A$
<proof>

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 $(S \rightarrow \text{including}_{Seq}(a)) \rightarrow \text{iterate}_{Seq}(b; x = A \mid P b x) = S \rightarrow \text{iterate}_{Seq}(b; x = P a A \mid P b x)$
<proof>

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 $(S \rightarrow \text{prepend}_{Seq}(a)) \rightarrow \text{iterate}_{Seq}(b; x = A \mid P b x) = P a (S \rightarrow \text{iterate}_{Seq}(b; x = A \mid P b x))$
<proof>

2.10.25. Test Statements

instantiation *Sequence_{base}* :: (*equal*)*equal*

begin

definition *HOL.equal* $k l \longleftrightarrow (k::('a::\text{equal})\text{Sequence}_{base}) = l$

instance *<proof>*

end

lemma *equal-Sequence_{base}-code* [*code*]:
 $\text{HOL.equal } k (l::('a::\{\text{equal},\text{null}\})\text{Sequence}_{base}) \longleftrightarrow \text{Rep-Sequence}_{base} k = \text{Rep-Sequence}_{base} l$

<proof>

Assert $\tau \models (\text{Sequence}\{\} \doteq \text{Sequence}\{\})$
Assert $\tau \models (\text{Sequence}\{\mathbf{1},\mathbf{2}\} \doteq \text{Sequence}\{\} \rightarrow \text{prepend}_{\text{Seq}}(\mathbf{2}) \rightarrow \text{prepend}_{\text{Seq}}(\mathbf{1}))$
Assert $\tau \models (\text{Sequence}\{\mathbf{1},\text{invalid},\mathbf{2}\} \doteq \text{invalid})$
Assert $\tau \models (\text{Sequence}\{\mathbf{1},\mathbf{2}\} \rightarrow \text{prepend}_{\text{Seq}}(\text{null}) \doteq \text{Sequence}\{\text{null},\mathbf{1},\mathbf{2}\})$
Assert $\tau \models (\text{Sequence}\{\mathbf{1},\mathbf{2}\} \rightarrow \text{including}_{\text{Seq}}(\text{null}) \doteq \text{Sequence}\{\mathbf{1},\mathbf{2},\text{null}\})$

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 $\text{OclAsBoolean}_{\text{Int}} :: ('A) \text{Integer} \Rightarrow ('A) \text{Boolean} ((-) \rightarrow \text{oclAsType}_{\text{Int}}('(\text{Boolean}')))$

where $\text{OclAsBoolean}_{\text{Int}} X = (\lambda\tau. \text{if } (\delta X) \tau = \text{true } \tau$
 $\text{then } \perp^{\ulcorner} X \tau^{\urcorner} \neq 0_{\perp}$
 $\text{else } \text{invalid } \tau)$

interpretation $\text{OclAsBoolean}_{\text{Int}} : \text{profile-mono}_d \text{OclAsBoolean}_{\text{Int}} \lambda x. \perp^{\ulcorner} x^{\urcorner} \neq 0_{\perp}$
<proof>

definition $\text{OclAsBoolean}_{\text{Real}} :: ('A) \text{Real} \Rightarrow ('A) \text{Boolean} ((-) \rightarrow \text{oclAsType}_{\text{Real}}('(\text{Boolean}')))$

where $\text{OclAsBoolean}_{\text{Real}} X = (\lambda\tau. \text{if } (\delta X) \tau = \text{true } \tau$
 $\text{then } \perp^{\ulcorner} X \tau^{\urcorner} \neq 0_{\perp}$
 $\text{else } \text{invalid } \tau)$

interpretation $\text{OclAsBoolean}_{\text{Real}} : \text{profile-mono}_d \text{OclAsBoolean}_{\text{Real}} \lambda x. \perp^{\ulcorner} x^{\urcorner} \neq 0_{\perp}$
<proof>

2.11.2. Definition: asInteger

definition $\text{OclAsInteger}_{\text{Real}} :: ('A) \text{Real} \Rightarrow ('A) \text{Integer} ((-) \rightarrow \text{oclAsType}_{\text{Real}}('(\text{Integer}')))$

where $\text{OclAsInteger}_{\text{Real}} X = (\lambda\tau. \text{if } (\delta X) \tau = \text{true } \tau$
 $\text{then } \perp^{\ulcorner} \text{floor } \tau^{\urcorner}$

else invalid τ)

interpretation $OclAsInteger_{Real} : profile\text{-}mono_d OclAsInteger_{Real} \lambda x. \sqcup floor \ulcorner x \urcorner \sqcup$
 ⟨proof⟩

2.11.3. Definition: asReal

definition $OclAsReal_{Int} :: ('A) Integer \Rightarrow ('A) Real ((-) \rightarrow oclAsType_{Int}('Real'))$
where $OclAsReal_{Int} X = (\lambda \tau. \text{if } (\delta X) \tau = true \tau$
 then $\sqcup real\text{-}of\text{-}int \ulcorner X \urcorner \tau \sqcup$
 else invalid τ)

interpretation $OclAsReal_{Int} : profile\text{-}mono_d OclAsReal_{Int} \lambda x. \sqcup real\text{-}of\text{-}int \ulcorner x \urcorner \sqcup$
 ⟨proof⟩

lemma *Integer-subtype-of-Real:*

assumes $\tau \models \delta X$

shows $\tau \models X \rightarrow oclAsType_{Int}(Real) \rightarrow oclAsType_{Real}(Integer) \triangleq X$

⟨proof⟩

2.11.4. Definition: asPair

definition $OclAsPair_{Seq} :: [('A, 'A::null) Sequence] \Rightarrow ('A, 'A::null, 'A::null) Pair ((-) \rightarrow asPair_{Seq}('))$
where $OclAsPair_{Seq} S = (\text{if } S \rightarrow size_{Seq}() \doteq 2$
 then $Pair\{S \rightarrow at_{Seq}(0), S \rightarrow at_{Seq}(1)\}$
 else invalid
 endif)

definition $OclAsPair_{Set} :: [('A, 'A::null) Set] \Rightarrow ('A, 'A::null, 'A::null) Pair ((-) \rightarrow asPair_{Set}('))$
where $OclAsPair_{Set} S = (\text{if } S \rightarrow size_{Set}() \doteq 2$
 then let $v = S \rightarrow any_{Set}()$ in
 $Pair\{v, S \rightarrow excluding_{Set}(v) \rightarrow any_{Set}()\}$
 else invalid
 endif)

definition $OclAsPair_{Bag} :: [('A, 'A::null) Bag] \Rightarrow ('A, 'A::null, 'A::null) Pair ((-) \rightarrow asPair_{Bag}('))$
where $OclAsPair_{Bag} S = (\text{if } S \rightarrow size_{Bag}() \doteq 2$
 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 $OclAsSet_{Seq} :: [('A, 'A::null) Sequence] \Rightarrow ('A, 'A) Set ((-) \rightarrow asSet_{Seq}('))$
where $OclAsSet_{Seq} S = (S \rightarrow iterate_{Seq}(b; x = Set\{\} \mid x \rightarrow including_{Set}(b)))$

definition $OclAsSet_{Pair} :: [('A, 'A::null, 'A::null) Pair] \Rightarrow ('A, 'A) Set ((-) \rightarrow asSet_{Pair}('))$
where $OclAsSet_{Pair} S = Set\{S .First(), S .Second()\}$

definition $OclAsSet_{Bag} :: ('A, 'A::null) Bag \Rightarrow ('A, 'A) Set ((-) \rightarrow asSet_{Bag}('))$
where $OclAsSet_{Bag} S = (\lambda \tau. \text{if } (\delta S) \tau = true \tau$
 then $Abs\text{-}Set_{base} \ulcorner Rep\text{-}Set\text{-}base S \urcorner \tau \sqcup$
 else if $(v S) \tau = true \tau$ then null τ
 else invalid τ)

2.11.6. Definition: asSequence

definition $OclAsSeq_{Set} :: [(\mathfrak{A}, \alpha :: null) Set] \Rightarrow (\mathfrak{A}, \alpha) Sequence ((-) \rightarrow asSequence_{Set} '())$
where $OclAsSeq_{Set} S = (S \rightarrow iterate_{Set}(b; x = Sequence\{\} \mid x \rightarrow including_{Seq}(b)))$

definition $OclAsSeq_{Bag} :: [(\mathfrak{A}, \alpha :: null) Bag] \Rightarrow (\mathfrak{A}, \alpha) Sequence ((-) \rightarrow asSequence_{Bag} '())$
where $OclAsSeq_{Bag} S = (S \rightarrow iterate_{Bag}(b; x = Sequence\{\} \mid x \rightarrow including_{Seq}(b)))$

definition $OclAsSeq_{Pair} :: [(\mathfrak{A}, \alpha :: null, \alpha' :: null) Pair] \Rightarrow (\mathfrak{A}, \alpha) Sequence ((-) \rightarrow asSequence_{Pair} '())$
where $OclAsSeq_{Pair} S = Sequence\{S .First(), S .Second()\}$

2.11.7. Definition: asBag

definition $OclAsBag_{Seq} :: [(\mathfrak{A}, \alpha :: null) Sequence] \Rightarrow (\mathfrak{A}, \alpha) Bag ((-) \rightarrow asBag_{Seq} '())$
where $OclAsBag_{Seq} S = (\lambda \tau. Abs-Bag_{base} \perp \lambda s. \text{if list-ex } (op = s) \text{ } \ulcorner Rep-Sequence_{base} (S \tau) \urcorner \text{ then } 1 \text{ else } 0_{\perp})$

definition $OclAsBag_{Set} :: [(\mathfrak{A}, \alpha :: null) Set] \Rightarrow (\mathfrak{A}, \alpha) Bag ((-) \rightarrow asBag_{Set} '())$
where $OclAsBag_{Set} S = (\lambda \tau. Abs-Bag_{base} \perp \lambda s. \text{if } s \in \ulcorner Rep-Set_{base} (S \tau) \urcorner \text{ then } 1 \text{ else } 0_{\perp})$

lemma assumes $\tau \models \delta (S \rightarrow size_{Set}())$
shows $OclAsBag_{Set} S = (S \rightarrow iterate_{Set}(b; x = Bag\{\} \mid x \rightarrow including_{Bag}(b)))$
 $\langle proof \rangle$

definition $OclAsBag_{Pair} :: [(\mathfrak{A}, \alpha :: null, \alpha' :: null) Pair] \Rightarrow (\mathfrak{A}, \alpha) Bag ((-) \rightarrow asBag_{Pair} '())$
where $OclAsBag_{Pair} S = Bag\{S .First(), S .Second()\}$

2.11.8. Collection Types

lemmas $cp\text{-intro}'' [intro!, simp, code-unfold] =$
 $cp\text{-intro}'$

$cp\text{-intro}''_{Set}$
 $cp\text{-intro}''_{Seq}$

2.11.9. Test Statements

lemma $syntax\text{-test}: Set\{2, 1\} = (Set\{\} \rightarrow including_{Set}(1) \rightarrow including_{Set}(2))$
 $\langle proof \rangle$

Here is an example of a nested collection.

lemma $semantic\text{-test}2:$
assumes $H: (Set\{2\} \doteq null) = (false :: (\mathfrak{A}) Boolean)$
shows $(\tau :: (\mathfrak{A}) st) \models (Set\{Set\{2\}, null\} \rightarrow includes_{Set}(null))$
 $\langle proof \rangle$

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

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

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

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

Elementary computations on Sets.

declare *OclSelect-body-def* [*simp*]

Assert $\neg (\tau \models v(\text{invalid}::('A, 'a::\text{null}) \text{Set}))$
Assert $\tau \models v(\text{null}::('A, 'a::\text{null}) \text{Set})$
Assert $\neg (\tau \models \delta(\text{null}::('A, 'a::\text{null}) \text{Set}))$
Assert $\tau \models v(\text{Set}\{\})$
Assert $\tau \models v(\text{Set}\{\text{Set}\{\mathbf{2}\}, \text{null}\})$
Assert $\tau \models \delta(\text{Set}\{\text{Set}\{\mathbf{2}\}, \text{null}\})$
Assert $\tau \models (\text{Set}\{\mathbf{2}, \mathbf{1}\} \rightarrow \text{includes}_{\text{Set}}(\mathbf{1}))$
Assert $\neg (\tau \models (\text{Set}\{\mathbf{2}\} \rightarrow \text{includes}_{\text{Set}}(\mathbf{1})))$
Assert $\neg (\tau \models (\text{Set}\{\mathbf{2}, \mathbf{1}\} \rightarrow \text{includes}_{\text{Set}}(\text{null})))$
Assert $\tau \models (\text{Set}\{\mathbf{2}, \text{null}\} \rightarrow \text{includes}_{\text{Set}}(\text{null}))$
Assert $\tau \models (\text{Set}\{\text{null}, \mathbf{2}\} \rightarrow \text{includes}_{\text{Set}}(\text{null}))$

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

Assert $\tau \models ((\text{Set}\{\mathbf{2}, \mathbf{1}\}) \rightarrow \text{forAll}_{\text{Set}}(z \mid \mathbf{0} <_{\text{int}} z))$
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 $\neg (\tau \models (\text{Set}\{\text{true}, \text{true}\} \doteq \text{Set}\{\text{false}\}))$
Assert $\neg (\tau \models (\text{Set}\{\mathbf{2}\} \doteq \text{Set}\{\mathbf{1}\}))$
Assert $\tau \models (\text{Set}\{\mathbf{2}, \text{null}, \mathbf{2}\} \doteq \text{Set}\{\text{null}, \mathbf{2}\})$
Assert $\tau \models (\text{Set}\{\mathbf{1}, \text{null}, \mathbf{2}\} \langle \rangle \text{Set}\{\text{null}, \mathbf{2}\})$
Assert $\tau \models (\text{Set}\{\text{Set}\{\mathbf{2}, \text{null}\}\} \doteq \text{Set}\{\text{Set}\{\text{null}, \mathbf{2}\}\})$
Assert $\tau \models (\text{Set}\{\text{Set}\{\mathbf{2}, \text{null}\}\} \langle \rangle \text{Set}\{\text{Set}\{\text{null}, \mathbf{2}\}, \text{null}\})$
Assert $\tau \models (\text{Set}\{\text{null}\} \rightarrow \text{selects}_{\text{Set}}(x \mid \text{not } x) \doteq \text{Set}\{\text{null}\})$
Assert $\tau \models (\text{Set}\{\text{null}\} \rightarrow \text{rejects}_{\text{Set}}(x \mid \text{not } x) \doteq \text{Set}\{\text{null}\})$

lemma *const* (*Set*{*Set*{**2**, *null*}, *invalid*}) *<proof>*

Elementary computations on Sequences.

Assert $\neg (\tau \models v(\text{invalid}::('A, 'a::\text{null}) \text{Sequence}))$
Assert $\tau \models v(\text{null}::('A, 'a::\text{null}) \text{Sequence})$
Assert $\neg (\tau \models \delta(\text{null}::('A, 'a::\text{null}) \text{Sequence}))$
Assert $\tau \models v(\text{Sequence}\{\})$

lemma *const* (*Sequence*{*Sequence*{**2**, *null*}, *invalid*}) *<proof>**<ML>**<ML>***end**

3. Formalization III: UML/OCL constructs: State Operations and Objects

```
theory UML-State
imports UML-Library
begin
```

```
no-notation None ( $\perp$ )
```

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 :: (' $\mathfrak{A}$ , 'a::{object,null})val  $\Rightarrow$  (' $\mathfrak{A}$ , 'a)val  $\Rightarrow$  (' $\mathfrak{A}$ )Boolean
where   StrictRefEqObject x y
         $\equiv$   $\lambda \tau$ . if (v x)  $\tau = true$   $\wedge$  (v y)  $\tau = true$   $\wedge$ 
            then if x  $\tau = null$   $\vee$  y  $\tau = null$ 
                then  $\perp$ x  $\tau = null$   $\wedge$  y  $\tau = null$   $\perp$ 
                else  $\perp$ (oid-of (x  $\tau$ )) = (oid-of (y  $\tau$ ))  $\perp$ 
            else invalid  $\tau$ 
```

Strictness and context passing

```
lemma StrictRefEqObject-strict1[simp,code-unfold] :
(StrictRefEqObject x invalid) = invalid
<proof>
```

```
lemma StrictRefEqObject-strict2[simp,code-unfold] :
(StrictRefEqObject invalid x) = invalid
<proof>
```

```
lemma cp-StrictRefEqObject:
(StrictRefEqObject x y  $\tau$ ) = (StrictRefEqObject ( $\lambda \cdot$ . x  $\tau$ ) ( $\lambda \cdot$ . y  $\tau$ ))  $\tau$ 
<proof>lemmas cp0-StrictRefEqObject= cp-StrictRefEqObject[THEN allI[THEN allI[THEN allI[THEN cpI2]],
of StrictRefEqObject]]
```

```
lemmas cp-intro''[intro!,simp,code-unfold] =
cp-intro''
cp-StrictRefEqObject[THEN allI[THEN allI[THEN allI[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:*

$\tau \models (\text{StrictRefEq}_{\text{Object}}\ x\ (y::(\mathfrak{A}, 'a::\{\text{null}, \text{object}\})\text{val})) \implies (\tau \models (v\ x)) \wedge (\tau \models (v\ y))$
 $\langle \text{proof} \rangle$

lemma *defined-StrictRefEqObject-I:*

assumes $\text{val-}x : \tau \models v\ x$
assumes $\text{val-}x : \tau \models v\ y$
shows $\tau \models \delta\ (\text{StrictRefEq}_{\text{Object}}\ x\ y)$
 $\langle \text{proof} \rangle$

lemma *StrictRefEqObject-def-homo :*

$\delta(\text{StrictRefEq}_{\text{Object}}\ x\ (y::(\mathfrak{A}, 'a::\{\text{null}, \text{object}\})\text{val})) = ((v\ x)\ \text{and}\ (v\ y))$
 $\langle \text{proof} \rangle$

Symmetry

lemma *StrictRefEqObject-sym :*

assumes $x\text{-val} : \tau \models v\ x$
shows $\tau \models \text{StrictRefEq}_{\text{Object}}\ x\ x$
 $\langle \text{proof} \rangle$

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 $WFF: WFF\ \tau$
and $\text{valid-}x: \tau \models (v\ x)$
and $\text{valid-}y: \tau \models (v\ y)$
and $x\text{-present-}pre: x\ \tau \in \text{ran}\ (\text{heap}(\text{fst } \tau))$
and $y\text{-present-}pre: y\ \tau \in \text{ran}\ (\text{heap}(\text{fst } \tau))$

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 $(\tau \models (\text{StrictRefEq}_{\text{Object}} x y)) = (\tau \models (x \triangleq y))$
<proof>

theorem *StrictRefEq_{Object}-vs-StrongEq'*:

assumes *WFF*: $\text{WFF } \tau$

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 $(\tau \models (\text{StrictRefEq}_{\text{Object}} x y)) = (\tau \models (x \triangleq y))$
<proof>

So, if two object descriptions live in the same state (both pre or post), the referential equality on objects implies in a WFF state the logical equality.

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* :: $((('A::\text{object}) \text{st} \Rightarrow 'A \text{state}) \Rightarrow ('A::\text{object} \rightarrow 'a) \Rightarrow$
 $('A, 'a \text{ option option}) \text{Set}$

where *OclAllInstances-generic* *fst-snd* $H =$
 $(\lambda \tau. \text{Abs-Set}_{\text{base}} \perp \perp \text{Some } ('(H \text{ 'ran } (\text{heap} (\text{fst-snd } \tau))) - \{ \text{None} \}) \perp \perp)$

lemma *OclAllInstances-generic-defined*: $\tau \models \delta (\text{OclAllInstances-generic } \text{pre-post } H)$
<proof>

lemma *OclAllInstances-generic-init-empty*:

assumes *[simp]*: $\bigwedge x. \text{pre-post } (x, x) = x$

shows $\tau_0 \models \text{OclAllInstances-generic } \text{pre-post } H \triangleq \text{Set}\{\}$

<proof>

lemma *represented-generic-objects-nonnul*:

assumes $A: \tau \models ((\text{OclAllInstances-generic } \text{pre-post } (H::('A::\text{object} \rightarrow 'a))) \rightarrow \text{includes}_{\text{Set}}(x))$

shows $\tau \models \text{not}(x \triangleq \text{null})$

<proof>

lemma *represented-generic-objects-defined*:

assumes $A: \tau \models ((\text{OclAllInstances-generic } \text{pre-post } (H::('A::\text{object} \rightarrow 'a))) \rightarrow \text{includes}_{\text{Set}}(x))$

shows $\tau \models \delta (\text{OclAllInstances-generic } \text{pre-post } H) \wedge \tau \models \delta x$

$\langle proof \rangle$

One way to establish the actual presence of an object representation in a state is:

definition *is-represented-in-state* $fst\text{-}snd\ x\ H\ \tau = (x\ \tau \in (Some\ o\ H)\ 'ran\ (heap\ (fst\text{-}snd\ \tau)))$

lemma *represented-generic-objects-in-state*:

assumes $A: \tau \models (OclAllInstances\text{-}generic\ pre\text{-}post\ H) \text{-} \> includes_{Set}(x)$

shows *is-represented-in-state* $pre\text{-}post\ x\ H\ \tau$

$\langle proof \rangle$

lemma *state-update-vs-allInstances-generic-empty*:

assumes $[simp]: \bigwedge a. pre\text{-}post\ (mk\ a) = a$

shows $(mk\ (\!|heap=empty, assoc= A\!)) \models OclAllInstances\text{-}generic\ pre\text{-}post\ Type \doteq Set\{\}$

$\langle proof \rangle$

Here comes a couple of operational rules that allow to infer the value of `oclAllInstances` from the context τ . These rules are a special-case in the sense that they are the only rules that relate statements with *different* τ 's. For that reason, new concepts like "constant contexts P" are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

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

assumes $[simp]: \bigwedge a. pre\text{-}post\ (mk\ a) = a$

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

and $Type\ Object \neq None$

shows $(OclAllInstances\text{-}generic\ pre\text{-}post\ Type)$

$(mk\ (\!|heap=\sigma'(oid \mapsto Object), assoc= A\!))$

$=$

$((OclAllInstances\text{-}generic\ pre\text{-}post\ Type) \text{-} \> including_{Set}(\lambda \cdot. \perp\ drop\ (Type\ Object)\ \perp))$

$(mk\ (\!|heap=\sigma', assoc= A\!))$

$\langle proof \rangle$

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

assumes $[simp]: \bigwedge a. pre\text{-}post\ (mk\ a) = a$

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

and $Type\ Object \neq None$

shows $(OclAllInstances\text{-}generic\ pre\text{-}post\ Type)$

$(mk\ (\!|heap=\sigma'(oid \mapsto Object), assoc= A\!))$

$=$

$((\lambda \cdot. (OclAllInstances\text{-}generic\ pre\text{-}post\ Type)$

$(mk\ (\!|heap=\sigma', assoc= A\!)) \text{-} \> including_{Set}(\lambda \cdot. \perp\ drop\ (Type\ Object)\ \perp))$

$(mk\ (\!|heap=\sigma'(oid \mapsto Object), assoc= A\!))$

$\langle proof \rangle$

lemma *state-update-vs-allInstances-generic-noincluding'*:

assumes $[simp]: \bigwedge a. pre\text{-}post\ (mk\ a) = a$

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

and $Type\ Object = None$

shows $(OclAllInstances\text{-}generic\ pre\text{-}post\ Type)$

$(mk\ (\!|heap=\sigma'(oid \mapsto Object), assoc= A\!))$

$=$

$(OclAllInstances\text{-}generic\ pre\text{-}post\ Type)$

$(mk\ (\!|heap=\sigma', assoc= A\!))$

$\langle proof \rangle$

theorem *state-update-vs-allInstances-generic-ntc*:
assumes [simp]: $\bigwedge a. \text{pre-post } (mk\ a) = a$
assumes *oid-def*: $oid \notin \text{dom } \sigma'$
and *non-type-conform*: $\text{Type Object} = \text{None}$
and *cp-ctxt*: $cp\ P$
and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P\ X)$
shows $(mk\ (\!|heap=\sigma'(oid \mapsto \text{Object}), assoc s=A|) \models P\ (\text{OclAllInstances-generic pre-post Type})) =$
 $(mk\ (\!|heap=\sigma', assoc s=A|) \models P\ (\text{OclAllInstances-generic pre-post Type}))$
(is $(? \tau \models P\ ? \varphi) = (? \tau' \models P\ ? \varphi)$
 $\langle \text{proof} \rangle$

theorem *state-update-vs-allInstances-generic-tc*:
assumes [simp]: $\bigwedge a. \text{pre-post } (mk\ a) = a$
assumes *oid-def*: $oid \notin \text{dom } \sigma'$
and *type-conform*: $\text{Type Object} \neq \text{None}$
and *cp-ctxt*: $cp\ P$
and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P\ X)$
shows $(mk\ (\!|heap=\sigma'(oid \mapsto \text{Object}), assoc s=A|) \models P\ (\text{OclAllInstances-generic pre-post Type})) =$
 $(mk\ (\!|heap=\sigma', assoc s=A|) \models P\ ((\text{OclAllInstances-generic pre-post Type})$
 $\quad \rightarrow \text{including}_{\text{Set}}(\lambda \cdot \cdot \cdot _[(\text{Type Object})])))$
(is $(? \tau \models P\ ? \varphi) = (? \tau' \models P\ ? \varphi')$
 $\langle \text{proof} \rangle$

declare *OclAllInstances-generic-def* [simp]

OclAllInstances (@post)

definition *OclAllInstances-at-post* :: $(\mathfrak{A} :: \text{object} \rightarrow 'a) \Rightarrow (\mathfrak{A}, 'a \text{ option option}) \text{ Set}$
 $(\cdot \cdot \cdot \text{allInstances}'(\cdot))$

where *OclAllInstances-at-post* = *OclAllInstances-generic snd*

lemma *OclAllInstances-at-post-defined*: $\tau \models \delta\ (H \cdot \cdot \cdot \text{allInstances}())$
 $\langle \text{proof} \rangle$

lemma $\tau_0 \models H \cdot \cdot \cdot \text{allInstances}() \triangleq \text{Set}\{\}$
 $\langle \text{proof} \rangle$

lemma *represented-at-post-objects-nonnull*:
assumes $A: \tau \models (((H :: (\mathfrak{A} :: \text{object} \rightarrow 'a)). \text{allInstances}()) \rightarrow \text{includes}_{\text{Set}}(x))$
shows $\tau \models \text{not}(x \triangleq \text{null})$
 $\langle \text{proof} \rangle$

lemma *represented-at-post-objects-defined*:
assumes $A: \tau \models (((H :: (\mathfrak{A} :: \text{object} \rightarrow 'a)). \text{allInstances}()) \rightarrow \text{includes}_{\text{Set}}(x))$
shows $\tau \models \delta\ (H \cdot \cdot \cdot \text{allInstances}()) \wedge \tau \models \delta\ x$
 $\langle \text{proof} \rangle$

One way to establish the actual presence of an object representation in a state is:

lemma
assumes $A: \tau \models H \cdot \cdot \cdot \text{allInstances}() \rightarrow \text{includes}_{\text{Set}}(x)$
shows *is-represented-in-state snd x H τ*
 $\langle \text{proof} \rangle$

lemma *state-update-vs-allInstances-at-post-empty*:
shows $(\sigma, (\!|heap=\text{empty}, assoc s=A|) \models \text{Type} \cdot \cdot \cdot \text{allInstances}() \triangleq \text{Set}\{\})$

<proof>

Here comes a couple of operational rules that allow to infer the value of `oclAllInstances` from the context τ . These rules are a special-case in the sense that they are the only rules that relate statements with *different* τ 's. For that reason, new concepts like “constant contexts P” are necessary (for which we do not elaborate an own theory for reasons of space limitations; in examples, we will prove resulting constraints straight forward by hand).

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

assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and $\text{Type Object} \neq \text{None}$
shows $(\text{Type .allInstances}())$
 $(\sigma, (\!| \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \!|))$
 $=$
 $((\text{Type .allInstances}()) \text{-> includingSet}(\lambda \cdot \perp \perp \text{ drop } (\text{Type Object}) \perp))$
 $(\sigma, (\!| \text{heap} = \sigma', \text{assocs} = A \!|))$

<proof>

lemma *state-update-vs-allInstances-at-post-including:*

assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and $\text{Type Object} \neq \text{None}$
shows $(\text{Type .allInstances}())$
 $(\sigma, (\!| \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \!|))$
 $=$
 $((\lambda \cdot (\text{Type .allInstances}())$
 $(\sigma, (\!| \text{heap} = \sigma', \text{assocs} = A \!|))) \text{-> includingSet}(\lambda \cdot \perp \perp \text{ drop } (\text{Type Object}) \perp))$
 $(\sigma, (\!| \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \!|))$

<proof>

lemma *state-update-vs-allInstances-at-post-noincluding'*:

assumes $\bigwedge x. \sigma' \text{ oid} = \text{Some } x \implies x = \text{Object}$
and $\text{Type Object} = \text{None}$
shows $(\text{Type .allInstances}())$
 $(\sigma, (\!| \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \!|))$
 $=$
 $(\text{Type .allInstances}())$
 $(\sigma, (\!| \text{heap} = \sigma', \text{assocs} = A \!|))$

<proof>

theorem *state-update-vs-allInstances-at-post-ntc:*

assumes *oid-def:* $\text{oid} \notin \text{dom } \sigma'$
and *non-type-conform:* $\text{Type Object} = \text{None}$
and *cp-ctxt:* $\text{cp } P$
and *const-ctxt:* $\bigwedge X. \text{const } X \implies \text{const } (P X)$
shows $((\sigma, (\!| \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \!|)) \models (P(\text{Type .allInstances}()))) =$
 $((\sigma, (\!| \text{heap} = \sigma', \text{assocs} = A \!|)) \models (P(\text{Type .allInstances}())))$

<proof>

theorem *state-update-vs-allInstances-at-post-tc:*

assumes *oid-def:* $\text{oid} \notin \text{dom } \sigma'$
and *type-conform:* $\text{Type Object} \neq \text{None}$
and *cp-ctxt:* $\text{cp } P$
and *const-ctxt:* $\bigwedge X. \text{const } X \implies \text{const } (P X)$
shows $((\sigma, (\!| \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \!|)) \models (P(\text{Type .allInstances}()))) =$
 $((\sigma, (\!| \text{heap} = \sigma', \text{assocs} = A \!|)) \models (P((\text{Type .allInstances}())))$

$\rightarrow \text{including}_{Set}(\lambda \cdot \cdot \cdot \lfloor \text{Type Object} \rfloor))$

$\langle \text{proof} \rangle$

OclAllInstances (@pre)

definition $\text{OclAllInstances-at-pre} :: ('\mathfrak{A} :: \text{object} \rightarrow '\alpha) \Rightarrow ('\mathfrak{A}, '\alpha \text{ option option}) \text{ Set}$
 $(\cdot \cdot \cdot \text{allInstances@pre}'())$

where $\text{OclAllInstances-at-pre} = \text{OclAllInstances-generic fst}$

lemma $\text{OclAllInstances-at-pre-defined}: \tau \models \delta (H \cdot \text{allInstances@pre}())$

$\langle \text{proof} \rangle$

lemma $\tau_0 \models H \cdot \text{allInstances@pre}() \hat{=} \text{Set}\{\}$

$\langle \text{proof} \rangle$

lemma $\text{represented-at-pre-objects-nonnull}$:

assumes $A: \tau \models (((H::(''\mathfrak{A}::\text{object} \rightarrow '\alpha)).\text{allInstances@pre}()) \rightarrow \text{includes}_{Set}(x))$

shows $\tau \models \text{not}(x \hat{=} \text{null})$

$\langle \text{proof} \rangle$

lemma $\text{represented-at-pre-objects-defined}$:

assumes $A: \tau \models (((H::(''\mathfrak{A}::\text{object} \rightarrow '\alpha)).\text{allInstances@pre}()) \rightarrow \text{includes}_{Set}(x))$

shows $\tau \models \delta (H \cdot \text{allInstances@pre}()) \wedge \tau \models \delta x$

$\langle \text{proof} \rangle$

One way to establish the actual presence of an object representation in a state is:

lemma

assumes $A: \tau \models H \cdot \text{allInstances@pre}() \rightarrow \text{includes}_{Set}(x)$

shows $\text{is-represented-in-state fst } x \text{ } H \tau$

$\langle \text{proof} \rangle$

lemma $\text{state-update-vs-allInstances-at-pre-empty}$:

shows $(\lfloor \text{heap} = \text{empty}, \text{assocs} = A \rfloor, \sigma) \models \text{Type} \cdot \text{allInstances@pre}() \hat{=} \text{Set}\{\}$

$\langle \text{proof} \rangle$

Here comes a couple of operational rules that allow to infer the value of $\text{oclAllInstances@pre}$ 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 $\text{state-update-vs-allInstances-at-pre-including}'$:

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

and $\text{Type Object} \neq \text{None}$

shows $(\text{Type} \cdot \text{allInstances@pre}())$

$(\lfloor \text{heap} = \sigma'(\text{oid} \mapsto \text{Object}), \text{assocs} = A \rfloor, \sigma)$

$=$

$((\text{Type} \cdot \text{allInstances@pre}()) \rightarrow \text{including}_{Set}(\lambda \cdot \cdot \cdot \lfloor \text{drop (Type Object)} \rfloor))$

$(\lfloor \text{heap} = \sigma', \text{assocs} = A \rfloor, \sigma)$

$\langle \text{proof} \rangle$

lemma $\text{state-update-vs-allInstances-at-pre-including}$:

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

and $\text{Type Object} \neq \text{None}$

shows $(\text{Type} \cdot \text{allInstances@pre}())$

$$\begin{aligned}
& ((\text{heap}=\sigma'(oid \mapsto \text{Object}), \text{assoc}=\text{A}), \sigma) \\
& = \\
& ((\lambda \cdot (\text{Type} . \text{allInstances}@pre()) \\
& \quad ((\text{heap}=\sigma', \text{assoc}=\text{A}), \sigma)) \rightarrow \text{including}_{\text{Set}}(\lambda \cdot \perp \text{ drop } (\text{Type } \text{Object}) \perp)) \\
& ((\text{heap}=\sigma'(oid \mapsto \text{Object}), \text{assoc}=\text{A}), \sigma) \\
\langle \text{proof} \rangle
\end{aligned}$$

lemma *state-update-vs-allInstances-at-pre-noincluding'*:

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

and $\text{Type } \text{Object} = \text{None}$

shows $(\text{Type} . \text{allInstances}@pre())$
 $((\text{heap}=\sigma'(oid \mapsto \text{Object}), \text{assoc}=\text{A}), \sigma)$
 $=$
 $(\text{Type} . \text{allInstances}@pre())$
 $((\text{heap}=\sigma', \text{assoc}=\text{A}), \sigma)$

$\langle \text{proof} \rangle$

theorem *state-update-vs-allInstances-at-pre-ntc*:

assumes *oid-def*: $oid \notin \text{dom } \sigma'$

and *non-type-conform*: $\text{Type } \text{Object} = \text{None}$

and *cp-ctxt*: $cp \ P$

and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P \ X)$

shows $((\text{heap}=\sigma'(oid \mapsto \text{Object}), \text{assoc}=\text{A}), \sigma) \models (P(\text{Type} . \text{allInstances}@pre())) =$
 $((\text{heap}=\sigma', \text{assoc}=\text{A}), \sigma) \models (P(\text{Type} . \text{allInstances}@pre()))$

$\langle \text{proof} \rangle$

theorem *state-update-vs-allInstances-at-pre-tc*:

assumes *oid-def*: $oid \notin \text{dom } \sigma'$

and *type-conform*: $\text{Type } \text{Object} \neq \text{None}$

and *cp-ctxt*: $cp \ P$

and *const-ctxt*: $\bigwedge X. \text{const } X \implies \text{const } (P \ X)$

shows $((\text{heap}=\sigma'(oid \mapsto \text{Object}), \text{assoc}=\text{A}), \sigma) \models (P(\text{Type} . \text{allInstances}@pre())) =$
 $((\text{heap}=\sigma', \text{assoc}=\text{A}), \sigma) \models (P((\text{Type} . \text{allInstances}@pre())$
 $\rightarrow \text{including}_{\text{Set}}(\lambda \cdot \perp (\text{Type } \text{Object}) \perp))$

$\langle \text{proof} \rangle$

@post or @pre

theorem *StrictRefEqObject-vs-StrongEq''*:

assumes *WFF*: $WFF \ \tau$

and *valid-x*: $\tau \models (v \ (x :: (\mathfrak{A}::\text{object}, \alpha::\text{object option option})val))$

and *valid-y*: $\tau \models (v \ y)$

and *oid-preserve*: $\bigwedge x. x \in \text{ran } (\text{heap}(\text{fst } \tau)) \vee x \in \text{ran } (\text{heap}(\text{snd } \tau)) \implies$
 $\text{oid-of } (H \ x) = \text{oid-of } x$

and *xy-together*: $\tau \models ((H . \text{allInstances}() \rightarrow \text{includes}_{\text{Set}}(x) \text{ and } H . \text{allInstances}() \rightarrow \text{includes}_{\text{Set}}(y)) \text{ or}$
 $(H . \text{allInstances}@pre() \rightarrow \text{includes}_{\text{Set}}(x) \text{ and } H . \text{allInstances}@pre() \rightarrow \text{includes}_{\text{Set}}(y)))$

shows $(\tau \models (\text{StrictRefEq}_{\text{Object}} \ x \ y)) = (\tau \models (x \hat{=} \ y))$

$\langle \text{proof} \rangle$

3.2.3. OclIsNew, OclIsDeleted, OclIsMaintained, OclIsAbsent

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

where $X . \text{oclIsNew}() \equiv (\lambda \tau . \text{if } (\delta \ X) \ \tau = \text{true} \ \tau$

$\text{then } \perp \text{oid-of } (X \ \tau) \notin \text{dom}(\text{heap}(\text{fst } \tau)) \wedge$
 $\text{oid-of } (X \ \tau) \in \text{dom}(\text{heap}(\text{snd } \tau)) \perp$

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} ', ' α ::{null,object})val \Rightarrow (' \mathfrak{A} ')Boolean ((-).oclIsDeleted'('))
where `X .oclIsDeleted()` \equiv ($\lambda\tau$. if (δ X) $\tau = true$ τ
then $\perp_{oid-of}(X \tau) \in dom(heap(fst \tau)) \wedge$
 $oid-of(X \tau) \notin dom(heap(snd \tau))_{\perp}$
else invalid τ)

definition `OclIsMaintained`:: (' \mathfrak{A} ', ' α ::{null,object})val \Rightarrow (' \mathfrak{A} ')Boolean((-).oclIsMaintained'('))
where `X .oclIsMaintained()` \equiv ($\lambda\tau$. if (δ X) $\tau = true$ τ
then $\perp_{oid-of}(X \tau) \in dom(heap(fst \tau)) \wedge$
 $oid-of(X \tau) \in dom(heap(snd \tau))_{\perp}$
else invalid τ)

definition `OclIsAbsent`:: (' \mathfrak{A} ', ' α ::{null,object})val \Rightarrow (' \mathfrak{A} ')Boolean ((-).oclIsAbsent'('))
where `X .oclIsAbsent()` \equiv ($\lambda\tau$. if (δ X) $\tau = true$ τ
then $\perp_{oid-of}(X \tau) \notin dom(heap(fst \tau)) \wedge$
 $oid-of(X \tau) \notin dom(heap(snd \tau))_{\perp}$
else invalid τ)

lemma `state-split` : $\tau \models \delta X \implies$
 $\tau \models (X .oclIsNew()) \vee \tau \models (X .oclIsDeleted()) \vee$
 $\tau \models (X .oclIsMaintained()) \vee \tau \models (X .oclIsAbsent())$

<proof>

lemma `notNew-vs-others` : $\tau \models \delta X \implies$
 $(\neg \tau \models (X .oclIsNew())) = (\tau \models (X .oclIsDeleted()) \vee$
 $\tau \models (X .oclIsMaintained()) \vee \tau \models (X .oclIsAbsent()))$

<proof>

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` :: (' \mathfrak{A} ::object, ' α ::{null,object})Set \Rightarrow ' \mathfrak{A} ' Boolean
 $(-\>oclIsModifiedOnly'('))$
where `X \rightarrow oclIsModifiedOnly()` \equiv ($\lambda(\sigma, \sigma')$.
let $X' = (oid-of \text{ ' } \ulcorner Rep-Set_{base}(X(\sigma, \sigma')) \urcorner)$;
 $S = ((dom(heap \sigma) \cap dom(heap \sigma')) - X')$
in if (δ X) $(\sigma, \sigma') = true$ $(\sigma, \sigma') \wedge (\forall x \in \ulcorner Rep-Set_{base}(X(\sigma, \sigma')) \urcorner. x \neq null)$
then $\perp_{\forall} x \in S. (heap \sigma) x = (heap \sigma') x_{\perp}$
else invalid (σ, σ'))

Execution with Invalid or Null or Null Element as Argument

lemma `invalid \rightarrow oclIsModifiedOnly()` = invalid
<proof>

lemma `null \rightarrow oclIsModifiedOnly()` = invalid
<proof>

lemma

assumes $X\text{-null} : \tau \models X \rightarrow \text{includes}_{Set}(\text{null})$
shows $\tau \models X \rightarrow \text{oclIsModifiedOnly}() \triangleq \text{invalid}$
<proof>

Context Passing

lemma $cp\text{-OclIsModifiedOnly} : X \rightarrow \text{oclIsModifiedOnly}() \tau = (\lambda \cdot X \tau) \rightarrow \text{oclIsModifiedOnly}() \tau$
<proof>

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

definition $OclSelf\text{-at}\text{-pre} :: ('A :: \text{object}, 'a :: \{\text{null}, \text{object}\})\ \text{val} \Rightarrow$
 $('A \Rightarrow 'a) \Rightarrow$
 $('A :: \text{object}, 'a :: \{\text{null}, \text{object}\})\ \text{val } ((-)\text{@pre}(-))$

where $x\ \text{@pre}\ H = OclSelf\ x\ H\ fst$

definition $OclSelf\text{-at}\text{-post} :: ('A :: \text{object}, 'a :: \{\text{null}, \text{object}\})\ \text{val} \Rightarrow$
 $('A \Rightarrow 'a) \Rightarrow$
 $('A :: \text{object}, 'a :: \{\text{null}, \text{object}\})\ \text{val } ((-)\text{@post}(-))$

where $x\ \text{@post}\ H = OclSelf\ x\ H\ snd$

3.2.6. Framing Theorem

lemma *all-oid-diff*:

assumes $def\text{-}x : \tau \models \delta\ x$

assumes $def\text{-}X : \tau \models \delta\ X$

assumes $def\text{-}X' : \bigwedge x. x \in \lceil \text{Rep}\text{-Set}_{base}(X\ \tau) \rceil \implies x \neq \text{null}$

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

shows $(\tau \models X \rightarrow \text{forAll}_{Set}(a\ | \ P\ a)) = (oid\text{-of } (x\ \tau) \notin oid\text{-of } \lceil \text{Rep}\text{-Set}_{base}(X\ \tau) \rceil)$

<proof>

theorem *framing*:

assumes $\text{modifies}\text{clause} : \tau \models (X \rightarrow \text{excluding}_{Set}(x)) \rightarrow \text{oclIsModifiedOnly}()$

and $oid\text{-is}\text{-typerepr} : \tau \models X \rightarrow \text{forAll}_{Set}(a\ | \ \text{not } (\text{StrictRefEq}_{Object}\ x\ a))$

shows $\tau \models (x\ \text{@pre}\ P \triangleq x\ \text{@post}\ P)$

<proof>

As corollary, the framing property can be expressed with only the strong equality as comparison operator.

theorem *framing'*:

assumes $wff : WFF\ \tau$

assumes $\text{modifies}\text{clause} : \tau \models (X \rightarrow \text{excluding}_{Set}(x)) \rightarrow \text{oclIsModifiedOnly}()$

and $oid\text{-is}\text{-typerepr} : \tau \models X \rightarrow \text{forAll}_{Set}(a\ | \ \text{not } (x \triangleq a))$

and $oid\text{-preserve} : \bigwedge x. x \in \text{ran } (\text{heap}(fst\ \tau)) \vee x \in \text{ran } (\text{heap}(snd\ \tau)) \implies$
 $oid\text{-of } (H\ x) = oid\text{-of } x$

and $xy\text{-together}$:

$\tau \models X \rightarrow \text{forAll}_{Set}(y \mid (H . \text{allInstances}() \rightarrow \text{includes}_{Set}(x) \text{ and } H . \text{allInstances}() \rightarrow \text{includes}_{Set}(y)) \text{ or } (H . \text{allInstances}@pre() \rightarrow \text{includes}_{Set}(x) \text{ and } H . \text{allInstances}@pre() \rightarrow \text{includes}_{Set}(y)))$
shows $\tau \models (x @pre P \triangleq (x @post P))$
 <proof>

3.2.7. Miscellaneous

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

lemma *pre-post-old*: $\tau \models (x . \text{oclIsDeleted}()) \implies \neg (\tau \models v(x @pre H1)) \wedge \neg (\tau \models v(x @post H2))$
 <proof>

lemma *pre-post-absent*: $\tau \models (x . \text{oclIsAbsent}()) \implies \neg (\tau \models v(x @pre H1)) \wedge \neg (\tau \models v(x @post H2))$
 <proof>

lemma *pre-post-maintained*: $(\tau \models v(x @pre H1) \vee \tau \models v(x @post H2)) \implies \tau \models (x . \text{oclIsMaintained}())$
 <proof>

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

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

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*): (' \mathfrak{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 $s = \text{select-object}_{Set} f s\text{-set}$ in

if $s \rightarrow \text{size}_{Set}() \triangleq \mathbf{1}$ then

$s \rightarrow \text{any}_{Set}()$

else

\perp

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 = ($\lambda(a,b). \text{OclPair} (f1 a) (f2 b)$)*

3.3.2. Validity and Definedness Properties

lemma *select-fold-exec_{Seq}*:

assumes *list-all* ($\lambda f. (\tau \models v f)$) l

shows $\ulcorner \text{Rep-Sequence}_{base} (\text{foldl } \text{UML-Sequence.OclIncluding } \text{Sequence}\{ \} l \tau) \urcorner = \text{List.map } (\lambda f. f \tau) l$

<proof>

lemma *select-fold-exec_{Set}*:
assumes *list-all* ($\lambda f. (\tau \models v f)$) *l*
shows $\lceil \text{Rep-Set}_{base} (\text{foldl } \text{UML-Set.OclIncluding } \text{Set}\{ \} l \tau) \rceil = \text{set } (\text{List.map } (\lambda f. f \tau) l)$
 $\langle \text{proof} \rangle$

lemma *fold-val-elem_{Seq}*:
assumes $\tau \models v (\text{foldl } \text{UML-Sequence.OclIncluding } \text{Sequence}\{ \} (\text{List.map } (f p) s\text{-set}))$
shows *list-all* ($\lambda x. (\tau \models v (f p x))$) *s-set*
 $\langle \text{proof} \rangle$

lemma *fold-val-elem_{Set}*:
assumes $\tau \models v (\text{foldl } \text{UML-Set.OclIncluding } \text{Set}\{ \} (\text{List.map } (f p) s\text{-set}))$
shows *list-all* ($\lambda x. (\tau \models v (f p x))$) *s-set*
 $\langle \text{proof} \rangle$

lemma *select-object-any-defined_{Seq}*:
assumes *def-sel*: $\tau \models \delta (\text{select-object-any}_{Seq} f s\text{-set})$
shows *s-set* $\neq []$
 $\langle \text{proof} \rangle$

lemma
assumes *def-sel*: $\tau \models \delta (\text{select-object-any}0_{Set} f s\text{-set})$
shows *s-set* $\neq []$
 $\langle \text{proof} \rangle$

lemma *select-object-any-defined_{Set}*:
assumes *def-sel*: $\tau \models \delta (\text{select-object-any}_{Set} f s\text{-set})$
shows *s-set* $\neq []$
 $\langle \text{proof} \rangle$

lemma *select-object-any-exec0_{Seq}*:
assumes *def-sel*: $\tau \models \delta (\text{select-object-any}_{Seq} f s\text{-set})$
shows $\tau \models (\text{select-object-any}_{Seq} f s\text{-set} \hat{=} f (\text{hd } s\text{-set}))$
 $\langle \text{proof} \rangle$

lemma *select-object-any-exec_{Seq}*:
assumes *def-sel*: $\tau \models \delta (\text{select-object-any}_{Seq} f s\text{-set})$
shows $\exists e. \text{List.member } s\text{-set } e \wedge (\tau \models (\text{select-object-any}_{Seq} f s\text{-set} \hat{=} f e))$
 $\langle \text{proof} \rangle$

lemma
assumes *def-sel*: $\tau \models \delta (\text{select-object-any}0_{Set} f s\text{-set})$
shows $\exists e. \text{List.member } s\text{-set } e \wedge (\tau \models (\text{select-object-any}0_{Set} f s\text{-set} \hat{=} f e))$
 $\langle \text{proof} \rangle$

lemma *select-object-any-exec_{Set}*:
assumes *def-sel*: $\tau \models \delta (\text{select-object-any}_{Set} f s\text{-set})$
shows $\exists e. \text{List.member } s\text{-set } e \wedge (\tau \models (\text{select-object-any}_{Set} f s\text{-set} \hat{=} f e))$
 $\langle \text{proof} \rangle$

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} , ' $\alpha 0$::*null*)*val* \Rightarrow
 '*b* \Rightarrow
 ('' \mathfrak{A} , '*res*::*null*)*val*

fixes *PRE*

fixes *POST*

assumes *def-scheme'*: *f self x* \equiv ($\lambda \tau$. *SOME res. let res =* $\lambda \cdot$. *res in*
 if ($\tau \models (\delta \text{ self}) \wedge f\text{-v } x \tau$
 then ($\tau \models \text{PRE self } x$) \wedge
 ($\tau \models \text{POST self } x \text{ res}$)
 else $\tau \models \text{res} \triangleq \text{invalid}$)

assumes *all-post'*: $\forall \sigma \sigma' \sigma''$. ($(\sigma, \sigma') \models \text{PRE self } x$) = ($(\sigma, \sigma'') \models \text{PRE self } x$)

assumes *cp_{PRE}'*: *PRE (self) x* τ = *PRE* ($\lambda \cdot$. *self* τ) (*f-lam* *x* τ) τ

assumes *cp_{POST}'*: *POST (self) x (res)* τ = *POST* ($\lambda \cdot$. *self* τ) (*f-lam* *x* τ) ($\lambda \cdot$. *res* τ) τ

assumes *f-v-val*: $\bigwedge a1$. *f-v* (*f-lam* *a1* τ) τ = *f-v* *a1* τ

begin

lemma *strict0* [*simp*]: *f invalid X* = *invalid*

<proof>

lemma *nullstrict0* [*simp*]: *f null X* = *invalid*

<proof>

lemma *cp0* : *f self a1* τ = *f* ($\lambda \cdot$. *self* τ) (*f-lam* *a1* τ) τ

<proof>

theorem *unfold'* :

assumes *context-ok*: *cp E*

and *args-def-or-valid*: ($\tau \models \delta \text{ self}$) \wedge *f-v a1* τ

and *pre-satisfied*: $\tau \models \text{PRE self } a1$

and *post-satisfiable*: $\exists \text{res}$. ($\tau \models \text{POST self } a1 (\lambda \cdot$. *res*))

and *sat-for-sols-post*: ($\bigwedge \text{res}$. $\tau \models \text{POST self } a1 (\lambda \cdot$. *res*) $\implies \tau \models E (\lambda \cdot$. *res*))

shows $\tau \models E(f \text{ self } a1)$

<proof>

lemma *unfold2'* :

assumes *context-ok*: *cp E*

and *args-def-or-valid*: ($\tau \models \delta \text{ self}$) \wedge (*f-v a1* τ)

and *pre-satisfied*: $\tau \models \text{PRE self } a1$

and *postsplit-satisfied*: $\tau \models \text{POST}' \text{ self } a1$

and *post-decomposable* : $\bigwedge \text{res}$. (*POST self a1 res*) =
 (*POST'* *self a1*) *and* (*res* \triangleq (*BODY self a1*)))

shows ($\tau \models E(f \text{ self } a1)$) = ($\tau \models E(\text{BODY self } a1)$)

<proof>

end

locale *contract0* =

fixes *f* :: (' \mathfrak{A} , ' $\alpha 0$::*null*)*val* \Rightarrow
 ('' \mathfrak{A} , '*res*::*null*)*val*

```

fixes PRE
fixes POST
assumes def-scheme:  $f \text{ self} \equiv (\lambda \tau. \text{SOME } res. \text{let } res = \lambda -. \text{res in}$ 
     $\text{if } (\tau \models (\delta \text{ self}))$ 
     $\text{then } (\tau \models \text{PRE self}) \wedge$ 
     $(\tau \models \text{POST self } res)$ 
     $\text{else } \tau \models res \triangleq \text{invalid}$ 
assumes all-post:  $\forall \sigma \sigma' \sigma''. ((\sigma, \sigma') \models \text{PRE self}) = ((\sigma, \sigma'') \models \text{PRE self})$ 

assumes cpPRE:  $\text{PRE (self) } \tau = \text{PRE } (\lambda -. \text{self } \tau) \tau$ 

assumes cpPOST:  $\text{POST (self) (res) } \tau = \text{POST } (\lambda -. \text{self } \tau) (\lambda -. \text{res } \tau) \tau$ 

sublocale contract0 < contract-scheme  $\lambda -. \text{True } \lambda x -. x \lambda x -. f x \lambda x -. \text{PRE } x \lambda x -. \text{POST } x$ 
  <proof>

context contract0
begin
  lemma cp-pre:  $cp \text{ self}' \implies cp (\lambda X. \text{PRE (self}' X) )$ 
  <proof>

  lemma cp-post:  $cp \text{ self}' \implies cp \text{ res}' \implies cp (\lambda X. \text{POST (self}' X) (res' X))$ 
  <proof>

  lemma cp [simp]:  $cp \text{ self}' \implies cp \text{ res}' \implies cp (\lambda X. f (\text{self}' X) )$ 
  <proof>

  lemmas unfold = unfold'[simplified]

  lemma unfold2 :
    assumes  $cp E$ 
    and  $(\tau \models \delta \text{ self})$ 
    and  $\tau \models \text{PRE self}$ 
    and  $\tau \models \text{POST}' \text{ self}$ 
    and  $\bigwedge res. (\text{POST self } res) =$ 
       $((\text{POST}' \text{ self}) \text{ and } (res \triangleq (\text{BODY self})))$ 
    shows  $(\tau \models E(f \text{ self})) = (\tau \models E(\text{BODY self}))$ 
    <proof>

end

locale contract1 =
  fixes f ::  $(\alpha, \alpha 0 :: \text{null}) \text{val} \Rightarrow$ 
     $(\alpha, \alpha 1 :: \text{null}) \text{val} \Rightarrow$ 
     $(\alpha, \text{res} :: \text{null}) \text{val}$ 
  fixes PRE
  fixes POST
  assumes def-scheme:  $f \text{ self } a1 \equiv$ 
     $(\lambda \tau. \text{SOME } res. \text{let } res = \lambda -. \text{res in}$ 
     $\text{if } (\tau \models (\delta \text{ self})) \wedge (\tau \models v \text{ a1})$ 
     $\text{then } (\tau \models \text{PRE self } a1) \wedge$ 
     $(\tau \models \text{POST self } a1 \text{ res})$ 
     $\text{else } \tau \models res \triangleq \text{invalid}$ 
  assumes all-post:  $\forall \sigma \sigma' \sigma''. ((\sigma, \sigma') \models \text{PRE self } a1) = ((\sigma, \sigma'') \models \text{PRE self } a1)$ 

  assumes cpPRE:  $\text{PRE (self) (a1) } \tau = \text{PRE } (\lambda -. \text{self } \tau) (\lambda -. \text{a1 } \tau) \tau$ 

```

assumes $cp_{POST}: POST (self) (a1) (res) \tau = POST (\lambda -. self \tau)(\lambda -. a1 \tau) (\lambda -. res \tau) \tau$
sublocale $contract1 < contract-scheme \lambda a1 \tau. (\tau \models v a1) \lambda a1 \tau. (\lambda -. a1 \tau)$
 <proof>
context $contract1$
begin
lemma $strict1[simp]: f self invalid = invalid$
 <proof>
lemma $defined-mono : \tau \models v(f Y Z) \implies (\tau \models \delta Y) \wedge (\tau \models v Z)$
 <proof>
lemma $cp-pre: cp self' \implies cp a1' \implies cp (\lambda X. PRE (self' X) (a1' X))$
 <proof>
lemma $cp-post: cp self' \implies cp a1' \implies cp res'$
 $\implies cp (\lambda X. POST (self' X) (a1' X) (res' X))$
 <proof>
lemma $cp [simp]: cp self' \implies cp a1' \implies cp res' \implies cp (\lambda X. f (self' X) (a1' X))$
 <proof>
lemmas $unfold = unfold'$
lemmas $unfold2 = unfold2'$
end

locale $contract2 =$
fixes $f :: ('A, 'a0::null)val \Rightarrow$
 $('A, 'a1::null)val \Rightarrow ('A, 'a2::null)val \Rightarrow$
 $('A, 'res::null)val$
fixes PRE
fixes $POST$
assumes $def-scheme: f self a1 a2 \equiv$
 $(\lambda \tau. SOME res. let res = \lambda -. res in$
 $if (\tau \models (\delta self)) \wedge (\tau \models v a1) \wedge (\tau \models v a2)$
 $then (\tau \models PRE self a1 a2) \wedge$
 $(\tau \models POST self a1 a2 res)$
 $else \tau \models res \triangleq invalid)$
assumes $all-post: \forall \sigma \sigma' \sigma''. ((\sigma, \sigma') \models PRE self a1 a2) = ((\sigma, \sigma'') \models PRE self a1 a2)$
assumes $cp_{PRE}: PRE (self) (a1) (a2) \tau = PRE (\lambda -. self \tau) (\lambda -. a1 \tau) (\lambda -. a2 \tau) \tau$
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 $contract2 < contract-scheme \lambda(a1, a2) \tau. (\tau \models v a1) \wedge (\tau \models v a2)$
 $\lambda(a1, a2) \tau. (\lambda -. a1 \tau, \lambda -. a2 \tau)$
 $(\lambda x (a, b). f x a b)$
 $(\lambda x (a, b). PRE x a b)$
 $(\lambda x (a, b). POST x a b)$
 <proof>

context $contract2$
begin
lemma $strict0'[simp] : f invalid X Y = invalid$

<proof>

lemma *nullstrict0*[simp]: $f \text{ null } X \ Y = \text{invalid}$
<proof>

lemma *strict1*[simp]: $f \text{ self } \text{invalid } Y = \text{invalid}$
<proof>

lemma *strict2*[simp]: $f \text{ self } X \ \text{invalid} = \text{invalid}$
<proof>

lemma *defined-mono* : $\tau \models v(f \ X \ Y \ Z) \implies (\tau \models \delta \ X) \wedge (\tau \models v \ Y) \wedge (\tau \models v \ Z)$
<proof>

lemma *cp-pre*: $cp \ \text{self}' \implies cp \ a1' \implies cp \ a2' \implies cp \ (\lambda X. \text{PRE} \ (\text{self}' \ X) \ (a1' \ X) \ (a2' \ X))$
<proof>

lemma *cp-post*: $cp \ \text{self}' \implies cp \ a1' \implies cp \ a2' \implies cp \ \text{res}'$
 $\implies cp \ (\lambda X. \text{POST} \ (\text{self}' \ X) \ (a1' \ X) \ (a2' \ X) \ (\text{res}' \ X))$
<proof>

lemma *cp0'* : $f \ \text{self} \ a1 \ a2 \ \tau = f \ (\lambda -. \ \text{self} \ \tau) \ (\lambda -. \ a1 \ \tau) \ (\lambda -. \ a2 \ \tau) \ \tau$
<proof>

lemma *cp* [simp]: $cp \ \text{self}' \implies cp \ a1' \implies cp \ a2' \implies cp \ \text{res}'$
 $\implies cp \ (\lambda X. \ f \ (\text{self}' \ X) \ (a1' \ X) \ (a2' \ X))$
<proof>

theorem *unfold* :

assumes $cp \ E$
and $(\tau \models \delta \ \text{self}) \wedge (\tau \models v \ a1) \wedge (\tau \models v \ a2)$
and $\tau \models \text{PRE} \ \text{self} \ a1 \ a2$
and $\exists \text{res}. (\tau \models \text{POST} \ \text{self} \ a1 \ a2 \ (\lambda -. \ \text{res}))$
and $(\bigwedge \text{res}. \tau \models \text{POST} \ \text{self} \ a1 \ a2 \ (\lambda -. \ \text{res}) \implies \tau \models E \ (\lambda -. \ \text{res}))$
shows $\tau \models E(f \ \text{self} \ a1 \ a2)$
<proof>

lemma *unfold2* :

assumes $cp \ E$
and $(\tau \models \delta \ \text{self}) \wedge (\tau \models v \ a1) \wedge (\tau \models v \ a2)$
and $\tau \models \text{PRE} \ \text{self} \ a1 \ a2$
and $\tau \models \text{POST}' \ \text{self} \ a1 \ a2$
and $\bigwedge \text{res}. (\text{POST} \ \text{self} \ a1 \ a2 \ \text{res}) =$
 $((\text{POST}' \ \text{self} \ a1 \ a2) \ \text{and} \ (\text{res} \triangleq (\text{BODY} \ \text{self} \ a1 \ a2)))$
shows $(\tau \models E(f \ \text{self} \ a1 \ a2)) = (\tau \models E(\text{BODY} \ \text{self} \ a1 \ a2))$
<proof>

end

locale *contract3* =

fixes $f :: ('A, 'a0::\text{null}) \text{val} \Rightarrow$
 $('A, 'a1::\text{null}) \text{val} \Rightarrow$
 $('A, 'a2::\text{null}) \text{val} \Rightarrow$
 $('A, 'a3::\text{null}) \text{val} \Rightarrow$
 $('A, 'res::\text{null}) \text{val}$
fixes PRE
fixes POST
assumes $\text{def-scheme}: f \ \text{self} \ a1 \ a2 \ a3 \equiv$

$(\lambda \tau. \text{SOME } res. \text{ let } res = \lambda -. res \text{ in}$
 $\text{ if } (\tau \models (\delta \text{ self})) \wedge (\tau \models v \ a1) \wedge (\tau \models v \ a2) \wedge (\tau \models v \ a3)$
 $\text{ then } (\tau \models \text{PRE self } a1 \ a2 \ a3) \wedge$
 $(\tau \models \text{POST self } a1 \ a2 \ a3 \ res)$
 $\text{ else } \tau \models res \triangleq \text{invalid})$

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*: $\text{PRE (self) (a1) (a2) (a3) } \tau = \text{PRE } (\lambda -. \text{self } \tau) (\lambda -. a1 \ \tau) (\lambda -. a2 \ \tau) (\lambda -. a3 \ \tau) \ \tau$

assumes *cpPOST*: $\bigwedge res. \text{POST (self) (a1) (a2) (a3) (res) } \tau =$
 $\text{POST } (\lambda -. \text{self } \tau) (\lambda -. a1 \ \tau) (\lambda -. a2 \ \tau) (\lambda -. a3 \ \tau) (\lambda -. res \ \tau) \ \tau$

sublocale *contract3* < *contract-scheme* $\lambda(a1, a2, a3) \tau. (\tau \models v \ a1) \wedge (\tau \models v \ a2) \wedge (\tau \models v \ a3)$
 $\lambda(a1, a2, a3) \tau. (\lambda -. a1 \ \tau, \lambda -. a2 \ \tau, \lambda -. a3 \ \tau)$
 $(\lambda x \ (a, b, c). f \ x \ a \ b \ c)$
 $(\lambda x \ (a, b, c). \text{PRE } x \ a \ b \ c)$
 $(\lambda x \ (a, b, c). \text{POST } x \ a \ b \ c)$

<proof>

context *contract3*

begin

lemma *strict0'[simp]*: $f \ \text{invalid } X \ Y \ Z = \text{invalid}$
<proof>

lemma *nullstrict0'[simp]*: $f \ \text{null } X \ Y \ Z = \text{invalid}$
<proof>

lemma *strict1[simp]*: $f \ \text{self } \text{invalid } Y \ Z = \text{invalid}$
<proof>

lemma *strict2[simp]*: $f \ \text{self } X \ \text{invalid } Z = \text{invalid}$
<proof>

lemma *defined-mono*: $\tau \models v(f \ W \ X \ Y \ Z) \implies (\tau \models \delta \ W) \wedge (\tau \models v \ X) \wedge (\tau \models v \ Y) \wedge (\tau \models v \ Z)$
<proof>

lemma *cp-pre*: $cp \ \text{self}' \implies cp \ a1' \implies cp \ a2' \implies cp \ a3'$
 $\implies cp \ (\lambda X. \text{PRE (self}' \ X) (a1' \ X) (a2' \ X) (a3' \ X))$
<proof>

lemma *cp-post*: $cp \ \text{self}' \implies cp \ a1' \implies cp \ a2' \implies cp \ a3' \implies cp \ \text{res}'$
 $\implies cp \ (\lambda X. \text{POST (self}' \ X) (a1' \ X) (a2' \ X) (a3' \ X) (res' \ X))$
<proof>

lemma *cp0'*: $f \ \text{self } a1 \ a2 \ a3 \ \tau = f \ (\lambda -. \text{self } \tau) (\lambda -. a1 \ \tau) (\lambda -. a2 \ \tau) (\lambda -. a3 \ \tau) \ \tau$
<proof>

lemma *cp [simp]*: $cp \ \text{self}' \implies cp \ a1' \implies cp \ a2' \implies cp \ a3' \implies cp \ \text{res}'$
 $\implies cp \ (\lambda X. f \ (\text{self}' \ X) (a1' \ X) (a2' \ X) (a3' \ X))$
<proof>

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 $\tau \models \text{PRE self } a1 \ a2 \ a3$
and $\exists res. (\tau \models \text{POST self } a1 \ a2 \ a3 \ (\lambda -. res))$

and $(\bigwedge res. \tau \models POST\ self\ a1\ a2\ a3\ (\lambda -. res) \implies \tau \models E\ (\lambda -. res))$
shows $\tau \models E(f\ self\ a1\ a2\ a3)$
 $\langle proof \rangle$

lemma *unfold2* :

assumes $cp\ E$
and $(\tau \models \delta\ self) \wedge (\tau \models v\ a1) \wedge (\tau \models v\ a2) \wedge (\tau \models v\ a3)$
and $\tau \models PRE\ self\ a1\ a2\ a3$
and $\tau \models POST'\ self\ a1\ a2\ a3$
and $\bigwedge res. (POST\ self\ a1\ a2\ a3\ res) =$
 $((POST'\ self\ a1\ a2\ a3)\ and\ (res \triangleq (BODY\ self\ a1\ a2\ a3)))$
shows $(\tau \models E(f\ self\ a1\ a2\ a3)) = (\tau \models E(BODY\ self\ a1\ a2\ a3))$
 $\langle proof \rangle$

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 *subst = subst1 subst2 subst4 [THEN iffD2] subst4*
thm *subst*
 $\langle ML \rangle$

lemma *test1* : $\tau \models A \implies \tau \models (A\ and\ B \triangleq B)$
 $\langle proof \rangle$

lemma *test2* : $\tau \models A \implies \tau \models (A\ and\ B \triangleq B)$
 $\langle proof \rangle$

lemma *test3* : $\tau \models A \implies \tau \models (A \text{ and } A)$
<proof>

lemma *test4* : $\tau \models \text{not } A \implies \tau \models (A \text{ and } B \triangleq \text{false})$
<proof>

lemma *test5* : $\tau \models (A \triangleq \text{null}) \implies \tau \models (B \triangleq \text{null}) \implies \neg (\tau \models (A \text{ and } B))$
<proof>

lemma *test6* : $\tau \models \text{not } A \implies \neg (\tau \models (A \text{ and } B))$
<proof>

lemma *test7* : $\neg (\tau \models (v \ A)) \implies \tau \models (\text{not } B) \implies \neg (\tau \models (A \text{ and } B))$
<proof>

lemma *X*: $\neg (\tau \models (\text{invalid and } B))$
<proof>

lemma *X'*: $\neg (\tau \models (\text{invalid and } B))$
<proof>

lemma *Y*: $\neg (\tau \models (\text{null and } B))$
<proof>

lemma *Z*: $\neg (\tau \models (\text{false and } B))$
<proof>

lemma *Z'*: $(\tau \models (\text{true and } B)) = (\tau \models B)$
<proof>

end

theory *UML-Main*
imports *UML-Contracts UML-Tools*

begin

end

4. Example: The Employee Analysis Model

```
theory
  Analysis-UML
imports
  ../../../../src/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).

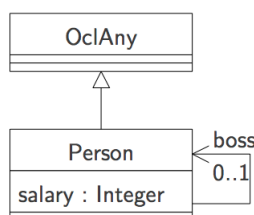


Figure 4.1.: A simple UML class model drawn from Figure 7.3, page 20 of [32].

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

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

```
instantiation  $\mathfrak{A}$  :: 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  $\langle$ proof $\rangle$ 
end
```

4.3. Instantiation of the Generic Strict Equality

We instantiate the referential equality on *Person* and *OclAny*

```

defs(overloaded) StrictRefEqObject-Person : (x::Person) ≐ y ≡ StrictRefEqObject x y
defs(overloaded) StrictRefEqObject-OclAny : (x::OclAny) ≐ y ≡ StrictRefEqObject x y

```

```

lemmas 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 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
<proof>

```

```

defs (overloaded) OclAsTypeOclAny-OclAny:
  (X::OclAny) .oclAsType(OclAny) ≡ X

```

```

defs (overloaded) OclAsTypeOclAny-Person:
  (X::Person) .oclAsType(OclAny) ≡
    (λτ. case X τ of
      ⊥ ⇒ invalid τ
      | ⊔⊥ ⇒ null τ
      | ⊔mkPerson oid a⊔ ⇒ ⊔ (mkOclAny oid ⊔a⊔)⊔)

```

```

definition OclAsTypePerson-ℳ =
  (λu. case u of inPerson p ⇒ ⊔p⊔
    | inOclAny (mkOclAny oid ⊔a⊔) ⇒ ⊔mkPerson oid a⊔
    | - ⇒ None)

```



```

defs (overloaded) OclAsTypePerson-OclAny:
  (X::OclAny) .oclAsType(Person) ≡
    (λτ. case X τ of
      ⊥ ⇒ invalid τ
      | ⊥⊥ ⇒ null τ
      | ⊥mkOclAny oid ⊥⊥ ⇒ invalid τ (* down-cast exception *)
      | ⊥mkOclAny oid ⊥a⊥ ⇒ ⊥mkPerson oid a⊥)

```

```

defs (overloaded) OclAsTypePerson-Person:
  (X::Person) .oclAsType(Person) ≡ X 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))
⟨proof⟩

```

```

lemma cp-OclAsTypeOclAny-OclAny-OclAny: cp P ⇒ cp(λX. (P (X::OclAny)::OclAny) .oclAsType(OclAny))
⟨proof⟩

```

```

lemma cp-OclAsTypePerson-Person-Person: cp P ⇒ cp(λX. (P (X::Person)::Person) .oclAsType(Person))
⟨proof⟩

```

```

lemma cp-OclAsTypePerson-OclAny-OclAny: cp P ⇒ cp(λX. (P (X::OclAny)::OclAny) .oclAsType(Person))
⟨proof⟩

```

```

lemma cp-OclAsTypeOclAny-Person-OclAny: cp P ⇒ cp(λX. (P (X::Person)::OclAny) .oclAsType(OclAny))
⟨proof⟩

```

```

lemma cp-OclAsTypeOclAny-OclAny-Person: cp P ⇒ cp(λX. (P (X::OclAny)::Person) .oclAsType(OclAny))
⟨proof⟩

```

```

lemma cp-OclAsTypePerson-Person-OclAny: cp P ⇒ cp(λX. (P (X::Person)::OclAny) .oclAsType(Person))
⟨proof⟩

```

```

lemma cp-OclAsTypePerson-OclAny-Person: cp P ⇒ cp(λX. (P (X::OclAny)::Person) .oclAsType(Person))
⟨proof⟩

```

```

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 ⟨proof⟩

```

```

lemma OclAsTypeOclAny-OclAny-nullstrict : (null::OclAny) .oclAsType(OclAny) = null ⟨proof⟩

```

```

lemma OclAsTypeOclAny-Person-strict[simp] : (invalid::Person) .oclAsType(OclAny) = invalid
⟨proof⟩

```

```

lemma OclAsTypeOclAny-Person-nullstrict[simp] : (null::Person) .oclAsType(OclAny) = null
⟨proof⟩

```

```

lemma OclAsTypePerson-OclAny-strict[simp] : (invalid::OclAny) .oclAsType(Person) = invalid
⟨proof⟩

```

```

lemma OclAsTypePerson-OclAny-nullstrict[simp] : (null::OclAny) .oclAsType(Person) = null
⟨proof⟩

```

```

lemma OclAsTypePerson-Person-strict : (invalid::Person) .oclAsType(Person) = invalid ⟨proof⟩

```

lemma *OclAsTypePerson-Person-nullstrict* : (null::Person) .oclAsType(Person) = null ⟨proof⟩

4.5. OclIsTypeOf

4.5.1. Definition

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

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

defs (overloaded) *OclIsTypeOfOclAny-OclAny*:

(X::OclAny) .oclIsTypeOf(OclAny) ≡
 (λτ. case X τ of
 ⊥ ⇒ invalid τ
 | ⊥_⊥ ⇒ true τ (* invalid ?? *)
 | ⊥_{mkOclAny} oid ⊥_⊥ ⇒ true τ
 | ⊥_{mkOclAny} oid ⊥_⊥ ⇒ false τ)

lemma *OclIsTypeOfOclAny-OclAny'*:

(X::OclAny) .oclIsTypeOf(OclAny) =
 (λ τ. if τ ⊨ v X then (case X τ of
 ⊥_⊥ ⇒ true τ (* invalid ?? *)
 | ⊥_{mkOclAny} oid ⊥_⊥ ⇒ true τ
 | ⊥_{mkOclAny} oid ⊥_⊥ ⇒ false τ)
 else invalid τ)

⟨proof⟩

interpretation *OclIsTypeOfOclAny-OclAny* :

profile-mono-schemeV

OclIsTypeOfOclAny::OclAny ⇒ Boolean

λ X. (case X of

 ⊥_{None} ⇒ ⊥_{True} (* invalid ?? *)
 | ⊥_{mkOclAny} oid None_⊥ ⇒ ⊥_{True}
 | ⊥_{mkOclAny} oid ⊥_⊥ ⇒ ⊥_{False})

⟨proof⟩

defs (overloaded) *OclIsTypeOfOclAny-Person*:

(X::Person) .oclIsTypeOf(OclAny) ≡
 (λτ. case X τ of
 ⊥ ⇒ invalid τ
 | ⊥_⊥ ⇒ true τ (* invalid ?? *)
 | ⊥₋ ⇒ false τ)

defs (overloaded) *OclIsTypeOfPerson-OclAny*:

(X::OclAny) .oclIsTypeOf(Person) ≡
 (λτ. case X τ of
 ⊥ ⇒ invalid τ
 | ⊥_⊥ ⇒ true τ
 | ⊥_{mkOclAny} oid ⊥_⊥ ⇒ false τ
 | ⊥_{mkOclAny} oid ⊥_⊥ ⇒ true τ)

defs (overloaded) *OclIsTypeOfPerson-Person*:

(X::Person) .oclIsTypeOf(Person) ≡
 (λτ. case X τ of
 ⊥ ⇒ invalid τ
 | - ⇒ true τ)

4.5.2. Context Passing

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

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

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

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

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

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

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

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

lemmas $[simp] =$

$cp\text{-}OclIsTypeOf_{OclAny}\text{-}Person\text{-}Person$
 $cp\text{-}OclIsTypeOf_{OclAny}\text{-}OclAny\text{-}OclAny$
 $cp\text{-}OclIsTypeOf_{Person}\text{-}Person\text{-}Person$
 $cp\text{-}OclIsTypeOf_{Person}\text{-}OclAny\text{-}OclAny$

$cp\text{-}OclIsTypeOf_{OclAny}\text{-}Person\text{-}OclAny$
 $cp\text{-}OclIsTypeOf_{OclAny}\text{-}OclAny\text{-}Person$
 $cp\text{-}OclIsTypeOf_{Person}\text{-}Person\text{-}OclAny$
 $cp\text{-}OclIsTypeOf_{Person}\text{-}OclAny\text{-}Person$

4.5.3. Execution with Invalid or Null as Argument

lemma $OclIsTypeOf_{OclAny}\text{-}OclAny\text{-}strict1$ $[simp]$:
 $(invalid::OclAny).oclIsTypeOf(OclAny) = invalid$
 $\langle proof \rangle$

lemma $OclIsTypeOf_{OclAny}\text{-}OclAny\text{-}strict2$ $[simp]$:
 $(null::OclAny).oclIsTypeOf(OclAny) = true$
 $\langle proof \rangle$

lemma $OclIsTypeOf_{OclAny}\text{-}Person\text{-}strict1$ $[simp]$:
 $(invalid::Person).oclIsTypeOf(OclAny) = invalid$
 $\langle proof \rangle$

lemma $OclIsTypeOf_{OclAny}\text{-}Person\text{-}strict2$ $[simp]$:
 $(null::Person).oclIsTypeOf(OclAny) = true$
 $\langle proof \rangle$

lemma $OclIsTypeOf_{Person}\text{-}OclAny\text{-}strict1$ $[simp]$:
 $(invalid::OclAny).oclIsTypeOf(Person) = invalid$
 $\langle proof \rangle$

lemma $OclIsTypeOf_{Person}\text{-}OclAny\text{-}strict2$ $[simp]$:
 $(null::OclAny).oclIsTypeOf(Person) = true$
 $\langle proof \rangle$

lemma $OclIsTypeOf_{Person}\text{-}Person\text{-}strict1$ $[simp]$:
 $(invalid::Person).oclIsTypeOf(Person) = invalid$
 $\langle proof \rangle$

lemma $OclIsTypeOf_{Person}\text{-}Person\text{-}strict2$ $[simp]$:
 $(null::Person).oclIsTypeOf(Person) = true$
 $\langle proof \rangle$

4.5.4. Up Down Casting

lemma *actualType-larger-staticType*:

assumes *isdef*: $\tau \models (\delta X)$

shows $\tau \models (X::Person) .oclIsTypeOf(OclAny) \triangleq false$

<proof>

lemma *down-cast-type*:

assumes *isOclAny*: $\tau \models (X::OclAny) .oclIsTypeOf(OclAny)$

and *non-null*: $\tau \models (\delta X)$

shows $\tau \models (X .oclAsType(Person)) \triangleq invalid$

<proof>

lemma *down-cast-type'*:

assumes *isOclAny*: $\tau \models (X::OclAny) .oclIsTypeOf(OclAny)$

and *non-null*: $\tau \models (\delta X)$

shows $\tau \models not (v (X .oclAsType(Person)))$

<proof>

lemma *up-down-cast* :

assumes *isdef*: $\tau \models (\delta X)$

shows $\tau \models ((X::Person) .oclAsType(OclAny) .oclAsType(Person)) \triangleq X$

<proof>

lemma *up-down-cast-Person-OclAny-Person* [*simp*]:

shows $((X::Person) .oclAsType(OclAny) .oclAsType(Person) = X)$

<proof>

lemma *up-down-cast-Person-OclAny-Person'*:

assumes $\tau \models v X$

shows $\tau \models (((X :: Person) .oclAsType(OclAny) .oclAsType(Person)) \doteq X)$

<proof>

lemma *up-down-cast-Person-OclAny-Person''*:

assumes $\tau \models v (X :: Person)$

shows $\tau \models (X .oclIsTypeOf(Person) \text{ implies } (X .oclAsType(OclAny) .oclAsType(Person)) \doteq X)$

<proof>

4.6. OclIsKindOf

4.6.1. Definition

consts *OclIsKindOf_{OclAny}* :: $'\alpha \Rightarrow Boolean ((-).oclIsKindOf'(OclAny'))$

consts *OclIsKindOf_{Person}* :: $'\alpha \Rightarrow Boolean ((-).oclIsKindOf'(Person'))$

defs (**overloaded**) *OclIsKindOf_{OclAny-OclAny}*:

$(X::OclAny) .oclIsKindOf(OclAny) \equiv$

$(\lambda\tau. \text{ case } X \text{ of}$
 $\quad \perp \Rightarrow invalid \ \tau$
 $\quad | _ \Rightarrow true \ \tau)$

defs (**overloaded**) *OclIsKindOf_{OclAny-Person}*:

$(X::Person) .oclIsKindOf(OclAny) \equiv$

$(\lambda\tau. \text{ case } X \text{ of}$
 $\quad \perp \Rightarrow invalid \ \tau$
 $\quad | _ \Rightarrow true \ \tau)$

defs (overloaded) $OclIsKindOf_{Person-OclAny}$:
 $(X::OclAny) .oclIsKindOf(Person) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \perp \Rightarrow \text{true } \tau$
 $\quad | \perp mk_{OclAny} \text{ oid } \perp \perp \Rightarrow \text{false } \tau$
 $\quad | \perp mk_{OclAny} \text{ oid } \perp \perp \Rightarrow \text{true } \tau)$

defs (overloaded) $OclIsKindOf_{Person-Person}$:
 $(X::Person) .oclIsKindOf(Person) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | - \Rightarrow \text{true } \tau)$

4.6.2. Context Passing

lemma $cp-OclIsKindOf_{OclAny-Person-Person}$: $cp \ P \implies cp(\lambda X.(P(X::Person)::Person).oclIsKindOf(OclAny))$
 $\langle \text{proof} \rangle$

lemma $cp-OclIsKindOf_{OclAny-OclAny-OclAny}$: $cp \ P \implies cp(\lambda X.(P(X::OclAny)::OclAny).oclIsKindOf(OclAny))$
 $\langle \text{proof} \rangle$

lemma $cp-OclIsKindOf_{Person-Person-Person}$: $cp \ P \implies cp(\lambda X.(P(X::Person)::Person).oclIsKindOf(Person))$
 $\langle \text{proof} \rangle$

lemma $cp-OclIsKindOf_{Person-OclAny-OclAny}$: $cp \ P \implies cp(\lambda X.(P(X::OclAny)::OclAny).oclIsKindOf(Person))$
 $\langle \text{proof} \rangle$

lemma $cp-OclIsKindOf_{OclAny-Person-OclAny}$: $cp \ P \implies cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(OclAny))$
 $\langle \text{proof} \rangle$

lemma $cp-OclIsKindOf_{OclAny-OclAny-Person}$: $cp \ P \implies cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(OclAny))$
 $\langle \text{proof} \rangle$

lemma $cp-OclIsKindOf_{Person-Person-OclAny}$: $cp \ P \implies cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(Person))$
 $\langle \text{proof} \rangle$

lemma $cp-OclIsKindOf_{Person-OclAny-Person}$: $cp \ P \implies cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(Person))$
 $\langle \text{proof} \rangle$

lemmas [simp] =

$cp-OclIsKindOf_{OclAny-Person-Person}$
 $cp-OclIsKindOf_{OclAny-OclAny-OclAny}$
 $cp-OclIsKindOf_{Person-Person-Person}$
 $cp-OclIsKindOf_{Person-OclAny-OclAny}$

$cp-OclIsKindOf_{OclAny-Person-OclAny}$
 $cp-OclIsKindOf_{OclAny-OclAny-Person}$
 $cp-OclIsKindOf_{Person-Person-OclAny}$
 $cp-OclIsKindOf_{Person-OclAny-Person}$

4.6.3. Execution with Invalid or Null as Argument

lemma $OclIsKindOf_{OclAny-OclAny-strict1}$ [simp]: $(invalid::OclAny) .oclIsKindOf(OclAny) = \text{invalid}$
 $\langle \text{proof} \rangle$

lemma $OclIsKindOf_{OclAny-OclAny-strict2}$ [simp]: $(null::OclAny) .oclIsKindOf(OclAny) = \text{true}$
 $\langle \text{proof} \rangle$

lemma $OclIsKindOf_{OclAny-Person-strict1}$ [simp]: $(invalid::Person) .oclIsKindOf(OclAny) = \text{invalid}$
 $\langle \text{proof} \rangle$

lemma $OclIsKindOf_{OclAny-Person-strict2}$ [simp]: $(null::Person) .oclIsKindOf(OclAny) = \text{true}$

$\langle \text{proof} \rangle$
lemma $OclIsKindOf_{Person-OclAny-strict1}[simp]: (invalid::OclAny) .oclIsKindOf(Person) = invalid$
 $\langle \text{proof} \rangle$
lemma $OclIsKindOf_{Person-OclAny-strict2}[simp]: (null::OclAny) .oclIsKindOf(Person) = true$
 $\langle \text{proof} \rangle$
lemma $OclIsKindOf_{Person-Person-strict1}[simp]: (invalid::Person) .oclIsKindOf(Person) = invalid$
 $\langle \text{proof} \rangle$
lemma $OclIsKindOf_{Person-Person-strict2}[simp]: (null::Person) .oclIsKindOf(Person) = true$
 $\langle \text{proof} \rangle$

4.6.4. Up Down Casting

lemma $actualKind-larger-staticKind:$
assumes $isdef: \tau \models (\delta X)$
shows $\tau \models ((X::Person) .oclIsKindOf(OclAny) \triangleq true)$
 $\langle \text{proof} \rangle$

lemma $down-cast-kind:$
assumes $isOclAny: \neg (\tau \models ((X::OclAny).oclIsKindOf(Person)))$
and $non-null: \tau \models (\delta X)$
shows $\tau \models ((X .oclAsType(Person)) \triangleq invalid)$
 $\langle \text{proof} \rangle$

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 $Person \equiv OclAsType_{Person-\mathfrak{A}}$
definition $OclAny \equiv OclAsType_{OclAny-\mathfrak{A}}$
lemmas $[simp] = Person-def\ OclAny-def$

lemma $OclAllInstances-generic_{OclAny-exec}: OclAllInstances-generic\ pre-post\ OclAny =$
 $(\lambda\tau. Abs-Set_{base} \perp\!\!\!\perp Some\ 'OclAny'\ ran\ (heap\ (pre-post\ \tau)) \perp\!\!\!\perp)$
 $\langle \text{proof} \rangle$

lemma $OclAllInstances-at-post_{OclAny-exec}: OclAny .allInstances() =$
 $(\lambda\tau. Abs-Set_{base} \perp\!\!\!\perp Some\ 'OclAny'\ ran\ (heap\ (snd\ \tau)) \perp\!\!\!\perp)$
 $\langle \text{proof} \rangle$

lemma $OclAllInstances-at-pre_{OclAny-exec}: OclAny .allInstances@pre() =$
 $(\lambda\tau. Abs-Set_{base} \perp\!\!\!\perp Some\ 'OclAny'\ ran\ (heap\ (fst\ \tau)) \perp\!\!\!\perp)$
 $\langle \text{proof} \rangle$

4.7.1. OclIsTypeOf

lemma $OclAny-allInstances-generic-oclIsTypeOf_{OclAny1}:$
assumes $[simp]: \bigwedge x. pre-post\ (x, x) = x$
shows $\exists \tau. (\tau \models ((OclAllInstances-generic\ pre-post\ OclAny) \rightarrow forAll_{Set}(X|X .oclIsTypeOf(OclAny))))$
 $\langle \text{proof} \rangle$

lemma $OclAny-allInstances-at-post-oclIsTypeOf_{OclAny1}:$
 $\exists \tau. (\tau \models (OclAny .allInstances() \rightarrow forAll_{Set}(X|X .oclIsTypeOf(OclAny))))$
 $\langle \text{proof} \rangle$

lemma $OclAny-allInstances-at-pre-oclIsTypeOf_{OclAny1}:$

$\exists \tau. (\tau \models (\text{OclAny} . \text{allInstances@pre}() \rightarrow \text{forAllSet}(X|X . \text{oclIsTypeOf}(\text{OclAny}))))$
 ⟨proof⟩

lemma *OclAny-allInstances-generic-oclIsTypeOf_{OclAny}2*:

assumes [*simp*]: $\bigwedge x. \text{pre-post}(x, x) = x$

shows $\exists \tau. (\tau \models \text{not} ((\text{OclAllInstances-generic pre-post OclAny}) \rightarrow \text{forAllSet}(X|X . \text{oclIsTypeOf}(\text{OclAny}))))$
 ⟨proof⟩

lemma *OclAny-allInstances-at-post-oclIsTypeOf_{OclAny}2*:

$\exists \tau. (\tau \models \text{not} (\text{OclAny} . \text{allInstances}() \rightarrow \text{forAllSet}(X|X . \text{oclIsTypeOf}(\text{OclAny}))))$
 ⟨proof⟩

lemma *OclAny-allInstances-at-pre-oclIsTypeOf_{OclAny}2*:

$\exists \tau. (\tau \models \text{not} (\text{OclAny} . \text{allInstances@pre}() \rightarrow \text{forAllSet}(X|X . \text{oclIsTypeOf}(\text{OclAny}))))$
 ⟨proof⟩

lemma *Person-allInstances-generic-oclIsTypeOf_{Person}*:

$\tau \models ((\text{OclAllInstances-generic pre-post Person}) \rightarrow \text{forAllSet}(X|X . \text{oclIsTypeOf}(\text{Person})))$
 ⟨proof⟩

lemma *Person-allInstances-at-post-oclIsTypeOf_{Person}*:

$\tau \models (\text{Person} . \text{allInstances}() \rightarrow \text{forAllSet}(X|X . \text{oclIsTypeOf}(\text{Person})))$
 ⟨proof⟩

lemma *Person-allInstances-at-pre-oclIsTypeOf_{Person}*:

$\tau \models (\text{Person} . \text{allInstances@pre}() \rightarrow \text{forAllSet}(X|X . \text{oclIsTypeOf}(\text{Person})))$
 ⟨proof⟩

4.7.2. OclIsKindOf

lemma *OclAny-allInstances-generic-oclIsKindOf_{OclAny}*:

$\tau \models ((\text{OclAllInstances-generic pre-post OclAny}) \rightarrow \text{forAllSet}(X|X . \text{oclIsKindOf}(\text{OclAny})))$
 ⟨proof⟩

lemma *OclAny-allInstances-at-post-oclIsKindOf_{OclAny}*:

$\tau \models (\text{OclAny} . \text{allInstances}() \rightarrow \text{forAllSet}(X|X . \text{oclIsKindOf}(\text{OclAny})))$
 ⟨proof⟩

lemma *OclAny-allInstances-at-pre-oclIsKindOf_{OclAny}*:

$\tau \models (\text{OclAny} . \text{allInstances@pre}() \rightarrow \text{forAllSet}(X|X . \text{oclIsKindOf}(\text{OclAny})))$
 ⟨proof⟩

lemma *Person-allInstances-generic-oclIsKindOf_{OclAny}*:

$\tau \models ((\text{OclAllInstances-generic pre-post Person}) \rightarrow \text{forAllSet}(X|X . \text{oclIsKindOf}(\text{OclAny})))$
 ⟨proof⟩

lemma *Person-allInstances-at-post-oclIsKindOf_{OclAny}*:

$\tau \models (\text{Person} . \text{allInstances}() \rightarrow \text{forAllSet}(X|X . \text{oclIsKindOf}(\text{OclAny})))$
 ⟨proof⟩

lemma *Person-allInstances-at-pre-oclIsKindOf_{OclAny}*:

$\tau \models (\text{Person} . \text{allInstances@pre}() \rightarrow \text{forAllSet}(X|X . \text{oclIsKindOf}(\text{OclAny})))$
 ⟨proof⟩

lemma *Person-allInstances-generic-oclIsKindOf_{Person}*:

$\tau \models ((\text{OclAllInstances-generic pre-post Person}) \rightarrow \text{forAllSet}(X|X . \text{oclIsKindOf}(\text{Person})))$
 ⟨proof⟩

lemma *Person-allInstances-at-post-oclIsKindOf_{Person}*:
 $\tau \models (Person \ .allInstances() \rightarrow \text{forAll}_{Set}(X|X \ .oclIsKindOf(Person)))$
<proof>

lemma *Person-allInstances-at-pre-oclIsKindOf_{Person}*:
 $\tau \models (Person \ .allInstances@pre() \rightarrow \text{forAll}_{Set}(X|X \ .oclIsKindOf(Person)))$
<proof>

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) val
 \Rightarrow (oid \Rightarrow (' \mathcal{A} , 'c::null) val)
 \Rightarrow (' \mathcal{A} , 'c::null) val

where *eval-extract* $X f = (\lambda \tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau \quad (* \text{ exception propagation } *)$
 $\quad | \perp \ \perp \ _ \Rightarrow \text{invalid } \tau \quad (* \text{ dereferencing null pointer } *)$
 $\quad | \perp \ \text{obj} \ _ \Rightarrow f \ (\text{oid-of } \text{obj}) \ \tau)$

definition *choose2-1* = *fst*

definition *choose2-2* = *snd*

definition *List-flatten* = $(\lambda l. (\text{foldl } ((\lambda \text{acc}. (\lambda l. (\text{foldl } ((\lambda \text{acc}. (\lambda l. (\text{Cons } l) (\text{acc})))))) (\text{acc}) ((\text{rev } l)))))) (\text{Nil})$
 $((\text{rev } l))))))$

definition *deref-assocs₂* :: (' \mathcal{A} state \times ' \mathcal{A} state \Rightarrow ' \mathcal{A} state)
 \Rightarrow (oid list list \Rightarrow oid list \times oid list)
 \Rightarrow oid
 \Rightarrow (oid list \Rightarrow (' \mathcal{A} , 'f) val)
 \Rightarrow oid
 \Rightarrow (' \mathcal{A} , 'f::null) val

where *deref-assocs₂* *pre-post to-from assoc-oid f oid* =
 $(\lambda \tau. \text{case } (\text{assocs } (\text{pre-post } \tau)) \ \text{assoc-oid} \ \text{of}$
 $\quad \perp \ S \ _ \Rightarrow f \ (\text{List-flatten } (\text{map } (\text{choose2-2} \circ \text{to-from})$
 $\quad \quad (\text{filter } (\lambda p. \text{List.member } (\text{choose2-1 } (\text{to-from } p)) \ \text{oid}) \ S)))$
 $\quad \quad \tau$
 $\quad | - \ \Rightarrow \text{invalid } \tau)$

The *pre-post*-parameter is configured with *fst* or *snd*, the *to-from*-parameter either with the identity *id* or the following combinator *switch*:

definition *switch₂₋₁* = $(\lambda [x,y] \Rightarrow (x,y))$

definition *switch₂₋₂* = $(\lambda [x,y] \Rightarrow (y,x))$

definition *switch₃₋₁* = $(\lambda [x,y,z] \Rightarrow (x,y))$

definition $switch_{3-2} = (\lambda[x,y,z] \Rightarrow (x,z))$

definition $switch_{3-3} = (\lambda[x,y,z] \Rightarrow (y,x))$

definition $switch_{3-4} = (\lambda[x,y,z] \Rightarrow (y,z))$

definition $switch_{3-5} = (\lambda[x,y,z] \Rightarrow (z,x))$

definition $switch_{3-6} = (\lambda[x,y,z] \Rightarrow (z,y))$

definition $deref-oid_{Person} :: (\mathfrak{A} \text{ state} \times \mathfrak{A} \text{ state} \Rightarrow \mathfrak{A} \text{ state})$

$\Rightarrow (type_{Person} \Rightarrow (\mathfrak{A}, 'c::null)val)$

$\Rightarrow oid$

$\Rightarrow (\mathfrak{A}, 'c::null)val$

where $deref-oid_{Person} \text{ fst-snd } f \text{ oid} = (\lambda\tau. \text{ case } (heap \text{ (fst-snd } \tau)) \text{ oid of}$

$\quad \sqsubset in_{Person} \text{ obj } \sqsupset \Rightarrow f \text{ obj } \tau$
 $\quad | - \quad \Rightarrow \text{ invalid } \tau)$

definition $deref-oid_{OclAny} :: (\mathfrak{A} \text{ state} \times \mathfrak{A} \text{ state} \Rightarrow \mathfrak{A} \text{ state})$

$\Rightarrow (type_{OclAny} \Rightarrow (\mathfrak{A}, 'c::null)val)$

$\Rightarrow oid$

$\Rightarrow (\mathfrak{A}, 'c::null)val$

where $deref-oid_{OclAny} \text{ fst-snd } f \text{ oid} = (\lambda\tau. \text{ case } (heap \text{ (fst-snd } \tau)) \text{ oid of}$

$\quad \sqsubset in_{OclAny} \text{ obj } \sqsupset \Rightarrow f \text{ obj } \tau$
 $\quad | - \quad \Rightarrow \text{ invalid } \tau)$

pointer undefined in state or not referencing a type conform object representation

definition $select_{OclAny} \mathcal{ANY} f = (\lambda X. \text{ case } X \text{ of}$

$\quad (mk_{OclAny} - \perp) \Rightarrow \text{ null}$

$\quad | (mk_{OclAny} - \sqsubset any \sqsupset) \Rightarrow f (\lambda x -. \sqsubset x \sqsupset) \text{ any})$

definition $select_{Person} \mathcal{BOSS} f = \text{select-object mtSet UML-Set.OclIncluding UML-Set.OclANY } (f (\lambda x -. \sqsubset x \sqsupset))$

definition $select_{Person} \mathcal{SALARY} f = (\lambda X. \text{ case } X \text{ of}$

$\quad (mk_{Person} - \perp) \Rightarrow \text{ null}$

$\quad | (mk_{Person} - \sqsubset salary \sqsupset) \Rightarrow f (\lambda x -. \sqsubset x \sqsupset) \text{ salary})$

definition $deref-assocs_2 \mathcal{BOSS} \text{ fst-snd } f = (\lambda mk_{Person} \text{ oid} -. \Rightarrow$

$\text{deref-assocs}_2 \text{ fst-snd } switch_{2-1} \text{ oid}_{Person} \mathcal{BOSS} f \text{ oid})$

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 $dot_{OclAny} \mathcal{ANY} :: OclAny \Rightarrow - \ ((1(-).any) \ 50)$

where $(X).any = \text{eval-extract } X$

$(deref-oid_{OclAny} \text{ in-post-state}$

$\text{ (select}_{OclAny} \mathcal{ANY}$

$\text{ reconst-basetype)})$

definition $dot_{Person} \mathcal{BOSS} :: Person \Rightarrow Person \ ((1(-).boss) \ 50)$

where $(X).boss = \text{eval-extract } X$

$(deref-oid_{Person} \text{ in-post-state}$

$\text{ (deref-assocs}_2 \mathcal{BOSS} \text{ in-post-state}$

$\text{ (select}_{Person} \mathcal{BOSS}$

(*deref-oid_{Person} in-post-state*))))

definition *dot_{Person}SALARY* :: *Person* ⇒ *Integer* ((1(-).salary) 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* ⇒ - ((1(-).any@pre) 50)

where (*X*).any@pre = *eval-extract X*
 (*deref-oid_{OclAny} in-pre-state*
 (*select_{OclAny}ANY*
reconst-basetype))

definition *dot_{Person}BOSS-at-pre*:: *Person* ⇒ *Person* ((1(-).boss@pre) 50)

where (*X*).boss@pre = *eval-extract X*
 (*deref-oid_{Person} in-pre-state*
 (*deref-assocs₂BOSS in-pre-state*
 (*select_{Person}BOSS*
 (*deref-oid_{Person} in-pre-state*))))

definition *dot_{Person}SALARY-at-pre*:: *Person* ⇒ *Integer* ((1(-).salary@pre) 50)

where (*X*).salary@pre = *eval-extract X*
 (*deref-oid_{Person} in-pre-state*
 (*select_{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

4.8.2. Context Passing

lemmas [*simp*] = *eval-extract-def*

lemma *cp-dot_{OclAny}ANY*: ((*X*).any) τ = ((λ-. *X* τ).any) τ ⟨*proof*⟩

lemma *cp-dot_{Person}BOSS*: ((*X*).boss) τ = ((λ-. *X* τ).boss) τ ⟨*proof*⟩

lemma *cp-dot_{Person}SALARY*: ((*X*).salary) τ = ((λ-. *X* τ).salary) τ ⟨*proof*⟩

lemma *cp-dot_{OclAny}ANY-at-pre*: ((*X*).any@pre) τ = ((λ-. *X* τ).any@pre) τ ⟨*proof*⟩

lemma *cp-dot_{Person}BOSS-at-pre*: ((*X*).boss@pre) τ = ((λ-. *X* τ).boss@pre) τ ⟨*proof*⟩

lemma *cp-dot_{Person}SALARY-at-pre*: ((*X*).salary@pre) τ = ((λ-. *X* τ).salary@pre) τ ⟨*proof*⟩

lemmas *cp-dot_{OclAny}ANY-I* [*simp, intro!*] =

cp-dot_{OclAny}ANY[*THEN all*[*THEN all*],
 of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

lemmas *cp-dot_{OclAny}ANY-at-pre-I* [*simp, intro!*] =

cp-dot_{OclAny}ANY-at-pre[*THEN all*[*THEN all*],
 of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

lemmas *cp-dot_{Person}BOSS-I* [*simp, intro!*] =

cp-dot_{Person}BOSS[*THEN all*[*THEN all*],
 of λ *X* -. *X* λ - τ. τ, *THEN cpI1*]

lemmas *cp-dotPersonBOSS-at-pre-I* [*simp, intro!*]=
cp-dotPersonBOSS-at-pre[*THEN all*[*THEN all*],
of $\lambda X \cdot X \lambda \tau. \tau$, *THEN cpI1*]

lemmas *cp-dotPersonSALARY-I* [*simp, intro!*]=
cp-dotPersonSALARY[*THEN all*[*THEN all*],
of $\lambda X \cdot X \lambda \tau. \tau$, *THEN cpI1*]

lemmas *cp-dotPersonSALARY-at-pre-I* [*simp, intro!*]=
cp-dotPersonSALARY-at-pre[*THEN all*[*THEN all*],
of $\lambda X \cdot X \lambda \tau. \tau$, *THEN cpI1*]

4.8.3. Execution with Invalid or Null as Argument

lemma *dotOclAnyANY-nullstrict* [*simp*]: *(null).any = invalid*
<proof>

lemma *dotOclAnyANY-at-pre-nullstrict* [*simp*] : *(null).any@pre = invalid*
<proof>

lemma *dotOclAnyANY-strict* [*simp*] : *(invalid).any = invalid*
<proof>

lemma *dotOclAnyANY-at-pre-strict* [*simp*] : *(invalid).any@pre = invalid*
<proof>

lemma *dotPersonBOSS-nullstrict* [*simp*]: *(null).boss = invalid*
<proof>

lemma *dotPersonBOSS-at-pre-nullstrict* [*simp*] : *(null).boss@pre = invalid*
<proof>

lemma *dotPersonBOSS-strict* [*simp*] : *(invalid).boss = invalid*
<proof>

lemma *dotPersonBOSS-at-pre-strict* [*simp*] : *(invalid).boss@pre = invalid*
<proof>

lemma *dotPersonSALARY-nullstrict* [*simp*]: *(null).salary = invalid*
<proof>

lemma *dotPersonSALARY-at-pre-nullstrict* [*simp*] : *(null).salary@pre = invalid*
<proof>

lemma *dotPersonSALARY-strict* [*simp*] : *(invalid).salary = invalid*
<proof>

lemma *dotPersonSALARY-at-pre-strict* [*simp*] : *(invalid).salary@pre = invalid*
<proof>

4.8.4. Representation in States

lemma *dotPersonBOSS-def-mono*: $\tau \models \delta(X \text{ . boss}) \implies \tau \models \delta(X)$
<proof>

lemma *repr-boss*:

assumes *A* : $\tau \models \delta(x \text{ . boss})$

shows *is-represented-in-state in-post-state* (*x . boss*) *Person* τ
<proof>

lemma *repr-bossX* :

assumes *A*: $\tau \models \delta(x \text{ . boss})$

shows $\tau \models ((\text{Person} \text{ . allInstances}()) \text{ -> includes}_{\text{Set}}(x \text{ . boss}))$
<proof>

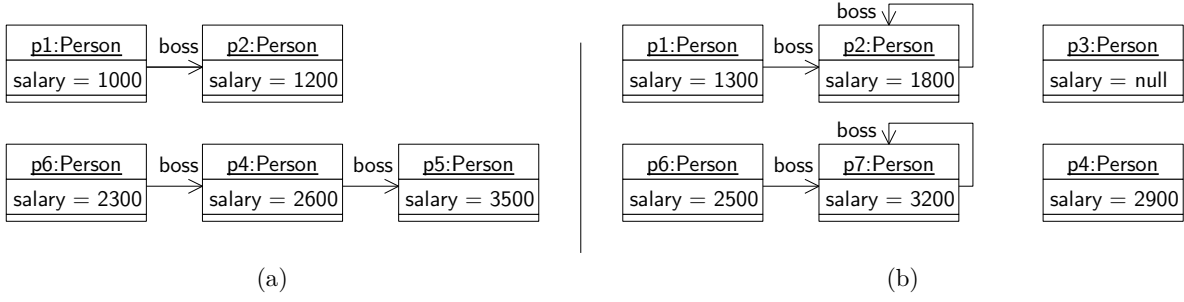


Figure 4.2.: (a) pre-state σ_1 and (b) post-state σ'_1 .

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 = ( $\lambda$  . .  $\_1000\_$ )
definition OclInt1200 (1200) where OclInt1200 = ( $\lambda$  . .  $\_1200\_$ )
definition OclInt1300 (1300) where OclInt1300 = ( $\lambda$  . .  $\_1300\_$ )
definition OclInt1800 (1800) where OclInt1800 = ( $\lambda$  . .  $\_1800\_$ )
definition OclInt2600 (2600) where OclInt2600 = ( $\lambda$  . .  $\_2600\_$ )
definition OclInt2900 (2900) where OclInt2900 = ( $\lambda$  . .  $\_2900\_$ )
definition OclInt3200 (3200) where OclInt3200 = ( $\lambda$  . .  $\_3200\_$ )
definition OclInt3500 (3500) where OclInt3500 = ( $\lambda$  . .  $\_3500\_$ )

```

```

definition oid0  $\equiv$  0
definition oid1  $\equiv$  1
definition oid2  $\equiv$  2
definition oid3  $\equiv$  3
definition oid4  $\equiv$  4
definition oid5  $\equiv$  5
definition oid6  $\equiv$  6
definition oid7  $\equiv$  7
definition oid8  $\equiv$  8

```

```

definition person1  $\equiv$  mkPerson oid0  $\_1300\_$ 
definition person2  $\equiv$  mkPerson oid1  $\_1800\_$ 
definition person3  $\equiv$  mkPerson oid2 None
definition person4  $\equiv$  mkPerson oid3  $\_2900\_$ 
definition person5  $\equiv$  mkPerson oid4  $\_3500\_$ 
definition person6  $\equiv$  mkPerson oid5  $\_2500\_$ 
definition person7  $\equiv$  mkOclAny oid6  $\_3200\_$ 
definition person8  $\equiv$  mkOclAny oid7 None
definition person9  $\equiv$  mkPerson oid8  $\_0\_$ 

```

```

definition
   $\sigma_1$   $\equiv$  ( heap = empty(oid0  $\mapsto$  inPerson (mkPerson oid0  $\_1000\_$ ))
    (oid1  $\mapsto$  inPerson (mkPerson oid1  $\_1200\_$ ))
    (*oid2*)
    (oid3  $\mapsto$  inPerson (mkPerson oid3  $\_2600\_$ ))
    (oid4  $\mapsto$  inPerson person5)
    (oid5  $\mapsto$  inPerson (mkPerson oid5  $\_2300\_$ ))
    (*oid6*)
    (*oid7*)
    (oid8  $\mapsto$  inPerson person9),
    assoc = empty(oidPersonBOSS  $\mapsto$  [[oid0],[oid1]],[oid3],[oid4],[oid5],[oid3]]))

```

definition

$$\sigma_1' \equiv (\text{heap} = \text{empty}(\text{oid0} \mapsto \text{in}_{\text{Person}} \text{person1})$$

$$\quad (\text{oid1} \mapsto \text{in}_{\text{Person}} \text{person2})$$

$$\quad (\text{oid2} \mapsto \text{in}_{\text{Person}} \text{person3})$$

$$\quad (\text{oid3} \mapsto \text{in}_{\text{Person}} \text{person4})$$

$$\quad (*\text{oid4}*)$$

$$\quad (\text{oid5} \mapsto \text{in}_{\text{Person}} \text{person6})$$

$$\quad (\text{oid6} \mapsto \text{in}_{\text{OclAny}} \text{person7})$$

$$\quad (\text{oid7} \mapsto \text{in}_{\text{OclAny}} \text{person8})$$

$$\quad (\text{oid8} \mapsto \text{in}_{\text{Person}} \text{person9}),$$

$$\text{assocs} = \text{empty}(\text{oid}_{\text{Person}} \text{BOSS} \mapsto [[[\text{oid0}],[\text{oid1}]],[[\text{oid1}],[\text{oid1}]],[[\text{oid5}],[\text{oid6}]],[[\text{oid6}],[\text{oid6}]]]) \text{)}$$

definition $\sigma_0 \equiv (\text{heap} = \text{empty}, \text{assocs} = \text{empty} \text{)}$

lemma *basic- τ -wff*: $WFF(\sigma_1, \sigma_1')$

<proof>

lemma [*simp, code-unfold*]: $\text{dom}(\text{heap } \sigma_1) = \{\text{oid0}, \text{oid1}, (*, \text{oid2}*)\text{oid3}, \text{oid4}, \text{oid5}(*, \text{oid6}, \text{oid7}*), \text{oid8}\}$

<proof>

lemma [*simp, code-unfold*]: $\text{dom}(\text{heap } \sigma_1') = \{\text{oid0}, \text{oid1}, \text{oid2}, \text{oid3}, (*, \text{oid4}*)\text{oid5}, \text{oid6}, \text{oid7}, \text{oid8}\}$

<proof> **definition** $X_{\text{Person}1} :: \text{Person} \equiv \lambda \cdot \cdot_{\perp} \text{person1} \ \perp$

definition $X_{\text{Person}2} :: \text{Person} \equiv \lambda \cdot \cdot_{\perp} \text{person2} \ \perp$

definition $X_{\text{Person}3} :: \text{Person} \equiv \lambda \cdot \cdot_{\perp} \text{person3} \ \perp$

definition $X_{\text{Person}4} :: \text{Person} \equiv \lambda \cdot \cdot_{\perp} \text{person4} \ \perp$

definition $X_{\text{Person}5} :: \text{Person} \equiv \lambda \cdot \cdot_{\perp} \text{person5} \ \perp$

definition $X_{\text{Person}6} :: \text{Person} \equiv \lambda \cdot \cdot_{\perp} \text{person6} \ \perp$

definition $X_{\text{Person}7} :: \text{OclAny} \equiv \lambda \cdot \cdot_{\perp} \text{person7} \ \perp$

definition $X_{\text{Person}8} :: \text{OclAny} \equiv \lambda \cdot \cdot_{\perp} \text{person8} \ \perp$

definition $X_{\text{Person}9} :: \text{Person} \equiv \lambda \cdot \cdot_{\perp} \text{person9} \ \perp$

lemma [*code-unfold*]: $((x :: \text{Person}) \doteq y) = \text{StrictRefEq}_{\text{Object}} \ x \ y$ *<proof>*

lemma [*code-unfold*]: $((x :: \text{OclAny}) \doteq y) = \text{StrictRefEq}_{\text{Object}} \ x \ y$ *<proof>*

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** $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{\text{Person}1} . \text{salary} \ \langle \rangle \ \mathbf{1000})$

Assert $\bigwedge_{s_{pre}} \cdot (s_{pre}, \sigma_1') \models (X_{\text{Person}1} . \text{salary} \ \doteq \ \mathbf{1300})$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{\text{Person}1} . \text{salary}@pre \ \doteq \ \mathbf{1000})$

Assert $\bigwedge_{s_{post}} \cdot (\sigma_1, s_{post}) \models (X_{\text{Person}1} . \text{salary}@pre \ \langle \rangle \ \mathbf{1300})$

lemma $(\sigma_1, \sigma_1') \models (X_{\text{Person}1} . \text{oclIsMaintained}())$

<proof>

lemma $\bigwedge_{s_{pre} \ s_{post}} \cdot (s_{pre}, s_{post}) \models ((X_{\text{Person}1} . \text{oclAsType}(\text{OclAny}) . \text{oclAsType}(\text{Person})) \doteq X_{\text{Person}1})$

$\langle proof \rangle$
Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models (X_{Person1} .oclIsTypeOf(Person))$
Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models not(X_{Person1} .oclIsTypeOf(OclAny))$
Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models (X_{Person1} .oclIsKindOf(Person))$
Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models (X_{Person1} .oclIsKindOf(OclAny))$
Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models not(X_{Person1} .oclAsType(OclAny) .oclIsTypeOf(OclAny))$

Assert $\bigwedge_{s_{pre} . (\sigma_1, \sigma_1')} \models (X_{Person2} .salary \doteq 1800)$
Assert $\bigwedge_{s_{post}. (\sigma_1, s_{post})} \models (X_{Person2} .salary@pre \doteq 1200)$

lemma $(\sigma_1, \sigma_1') \models (X_{Person2} .oclIsMaintained())$
 $\langle proof \rangle$

Assert $\bigwedge_{s_{pre} . (\sigma_1, \sigma_1')} \models (X_{Person3} .salary \doteq null)$
Assert $\bigwedge_{s_{post}. (\sigma_1, s_{post})} \models not(v(X_{Person3} .salary@pre))$
lemma $(\sigma_1, \sigma_1') \models (X_{Person3} .oclIsNew())$
 $\langle proof \rangle$

lemma $(\sigma_1, \sigma_1') \models (X_{Person4} .oclIsMaintained())$
 $\langle proof \rangle$

Assert $\bigwedge_{s_{pre} . (\sigma_1, \sigma_1')} \models not(v(X_{Person5} .salary))$
Assert $\bigwedge_{s_{post}. (\sigma_1, s_{post})} \models (X_{Person5} .salary@pre \doteq 3500)$

lemma $(\sigma_1, \sigma_1') \models (X_{Person5} .oclIsDeleted())$
 $\langle proof \rangle$

lemma $(\sigma_1, \sigma_1') \models (X_{Person6} .oclIsMaintained())$
 $\langle proof \rangle$

Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models v(X_{Person7} .oclAsType(Person))$

lemma $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models ((X_{Person7} .oclAsType(Person) .oclAsType(OclAny) .oclAsType(Person)) \doteq (X_{Person7} .oclAsType(Person)))$

$\langle proof \rangle$

lemma $(\sigma_1, \sigma_1') \models (X_{Person7} .oclIsNew())$
 $\langle proof \rangle$

Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models (X_{Person8} <> X_{Person7})$
Assert $\bigwedge_{s_{pre} s_{post}. (s_{pre}, s_{post})} \models not(v(X_{Person8} .oclAsType(Person)))$
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: $(\sigma_1, \sigma_1') \models (Set\{ X_{Person1} .oclAsType(OclAny) , X_{Person2} .oclAsType(OclAny) (*, X_{Person3} .oclAsType(OclAny))*$

```

    , XPerson4 .oclAsType(OclAny)
    (*, XPerson5 .oclAsType(OclAny)*)
    , XPerson6 .oclAsType(OclAny)
    (*, XPerson7 .oclAsType(OclAny)*)
    (*, XPerson8 .oclAsType(OclAny)*)
    (*, XPerson9 .oclAsType(OclAny)*)} -> oclIsModifiedOnly()
⟨proof⟩

```

lemma $(\sigma_1, \sigma_1') \models ((X_{Person9} \text{ @pre } (\lambda x. _OclAsType_{Person}\text{-}\mathfrak{A} \ x)) \triangleq X_{Person9})$
⟨proof⟩

lemma $(\sigma_1, \sigma_1') \models ((X_{Person9} \text{ @post } (\lambda x. _OclAsType_{Person}\text{-}\mathfrak{A} \ x)) \triangleq X_{Person9})$
⟨proof⟩

lemma $(\sigma_1, \sigma_1') \models (((X_{Person9} \text{ .oclAsType}(OclAny)) \text{ @pre } (\lambda x. _OclAsType_{OclAny}\text{-}\mathfrak{A} \ x)) \triangleq$
 $(X_{Person9} \text{ .oclAsType}(OclAny)) \text{ @post } (\lambda x. _OclAsType_{OclAny}\text{-}\mathfrak{A} \ x))$
⟨proof⟩

lemma $perm\text{-}\sigma_1' : \sigma_1' = (\text{ heap} = \text{ empty}$
 $(oid8 \mapsto in_{Person} \ person9)$
 $(oid7 \mapsto in_{OclAny} \ person8)$
 $(oid6 \mapsto in_{OclAny} \ person7)$
 $(oid5 \mapsto in_{Person} \ person6)$
 $(*oid4*)$
 $(oid3 \mapsto in_{Person} \ person4)$
 $(oid2 \mapsto in_{Person} \ person3)$
 $(oid1 \mapsto in_{Person} \ person2)$
 $(oid0 \mapsto in_{Person} \ person1)$
 $, \text{ assoc} = \text{ assoc} \ \sigma_1' \)$
⟨proof⟩

declare *const-ss* [*simp*]

lemma $\bigwedge \sigma_1.$
 $(\sigma_1, \sigma_1') \models (Person \ .allInstances() \doteq Set\{ X_{Person1}, X_{Person2}, X_{Person3}, X_{Person4}(*, X_{Person5*}),$
 $X_{Person6},$
 $X_{Person7} \ .oclAsType(Person)(* , X_{Person8*}), X_{Person9} \})$
⟨proof⟩

lemma $\bigwedge \sigma_1.$
 $(\sigma_1, \sigma_1') \models (OclAny \ .allInstances() \doteq Set\{ X_{Person1} \ .oclAsType(OclAny), X_{Person2} \ .oclAsType(OclAny),$
 $X_{Person3} \ .oclAsType(OclAny), X_{Person4} \ .oclAsType(OclAny)$
 $(* , X_{Person5*}), X_{Person6} \ .oclAsType(OclAny),$
 $X_{Person7}, X_{Person8}, X_{Person9} \ .oclAsType(OclAny) \})$
⟨proof⟩

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 $Person\text{-}label_{globalinv} \equiv (Person .allInstances() \text{->} \text{forAll}_{Set}(x \mid Person\text{-}label_{inv}(x)) \text{ and}$
 $(Person .allInstances@pre() \text{->} \text{forAll}_{Set}(x \mid Person\text{-}label_{invATpre}(x))))$

lemma $\tau \models \delta(X .boss) \implies \tau \models Person .allInstances() \text{->} \text{includes}_{Set}(X .boss) \wedge$
 $\tau \models Person .allInstances() \text{->} \text{includes}_{Set}(X)$
 <proof>

lemma $REC\text{-}pre : \tau \models Person\text{-}label_{globalinv}$
 $\implies \tau \models Person .allInstances() \text{->} \text{includes}_{Set}(X) (* X \text{ represented object in state } *)$
 $\implies \exists REC. \tau \models REC(X) \triangleq (Person\text{-}label_{inv}(X) \text{ and } (X .boss <> null \text{ implies } REC(X .boss)))$
 <proof>

This allows to state a predicate:

axiomatization $inv_{Person\text{-}label} :: Person \Rightarrow Boolean$
where $inv_{Person\text{-}label}\text{-}def:$
 $(\tau \models Person .allInstances() \text{->} \text{includes}_{Set}(self)) \implies$
 $(\tau \models (inv_{Person\text{-}label}(self) \triangleq (self .boss <> null \text{ implies}$
 $(self .salary \leq_{int} ((self .boss) .salary)) \text{ and}$
 $inv_{Person\text{-}label}(self .boss))))$

axiomatization $inv_{Person\text{-}labelATpre} :: Person \Rightarrow Boolean$
where $inv_{Person\text{-}labelATpre}\text{-}def:$
 $(\tau \models Person .allInstances@pre() \text{->} \text{includes}_{Set}(self)) \implies$
 $(\tau \models (inv_{Person\text{-}labelATpre}(self) \triangleq (self .boss@pre <> null \text{ implies}$
 $(self .salary@pre \leq_{int} ((self .boss@pre) .salary@pre)) \text{ and}$
 $inv_{Person\text{-}labelATpre}(self .boss@pre))))$

lemma $inv\text{-}1 :$
 $(\tau \models Person .allInstances() \text{->} \text{includes}_{Set}(self)) \implies$
 $(\tau \models inv_{Person\text{-}label}(self) = ((\tau \models (self .boss \doteq null)) \vee$
 $(\tau \models (self .boss <> null) \wedge$
 $\tau \models ((self .salary) \leq_{int} (self .boss .salary)) \wedge$
 $\tau \models (inv_{Person\text{-}label}(self .boss))))))$
 <proof>

lemma *inv-2* :

$$\begin{aligned}
& (\tau \models \text{Person} . \text{allInstances@pre}() \rightarrow \text{includes}_{\text{Set}}(\text{self})) \implies \\
& \quad (\tau \models \text{inv}_{\text{Person-labelATpre}}(\text{self}) = ((\tau \models (\text{self} . \text{boss@pre} \doteq \text{null})) \vee \\
& \quad \quad (\tau \models (\text{self} . \text{boss@pre} <> \text{null})) \wedge \\
& \quad \quad (\tau \models (\text{self} . \text{boss@pre} . \text{salary@pre} \leq_{\text{int}} \text{self} . \text{salary@pre})) \wedge \\
& \quad \quad (\tau \models (\text{inv}_{\text{Person-labelATpre}}(\text{self} . \text{boss@pre})))))) \\
& \langle \text{proof} \rangle
\end{aligned}$$

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**

$$\begin{aligned}
& (\tau \models (\delta \text{ self})) \implies ((\tau \models (\text{self} . \text{boss} \doteq \text{null})) \vee \\
& \quad (\tau \models (\text{self} . \text{boss} <> \text{null})) \wedge (\tau \models (\text{self} . \text{boss} . \text{salary} \leq_{\text{int}} \text{self} . \text{salary})) \wedge \\
& \quad ((\text{inv}(\text{self} . \text{boss}))\tau)) \\
& \implies (\text{inv self } \tau)
\end{aligned}$$

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* \Rightarrow *Set-Integer* ((*1*(-).*contents'*(')) 50)

where *contents-def*:

$$\begin{aligned}
& (\text{self} . \text{contents}()) = (\lambda \tau . \text{SOME } \text{res} . \text{let } \text{res} = \lambda - . \text{res in} \\
& \quad \text{if } \tau \models (\delta \text{ self}) \\
& \quad \text{then } ((\tau \models \text{true}) \wedge \\
& \quad \quad (\tau \models \text{res} \triangleq \text{if } (\text{self} . \text{boss} \doteq \text{null}) \\
& \quad \quad \quad \text{then } (\text{Set}\{\text{self} . \text{salary}\}) \\
& \quad \quad \quad \text{else } (\text{self} . \text{boss} . \text{contents}() \\
& \quad \quad \quad \quad \rightarrow \text{including}_{\text{Set}}(\text{self} . \text{salary})) \\
& \quad \quad \quad \text{endif})) \\
& \quad \text{else } \tau \models \text{res} \triangleq \text{invalid})
\end{aligned}$$

and *cp0-contents*:(*X* .*contents*()) $\tau = ((\lambda - . X \tau) . \text{contents}()) \tau$

interpretation *contents* : *contract0 contents* λ *self* . *true*

$$\begin{aligned}
& \lambda \text{ self } \text{res} . \text{res} \triangleq \text{if } (\text{self} . \text{boss} \doteq \text{null}) \\
& \quad \text{then } (\text{Set}\{\text{self} . \text{salary}\}) \\
& \quad \text{else } (\text{self} . \text{boss} . \text{contents}() \\
& \quad \quad \rightarrow \text{including}_{\text{Set}}(\text{self} . \text{salary})) \\
& \quad \text{endif}
\end{aligned}$$

$\langle \text{proof} \rangle$

Specializing $\llbracket \text{cp } E; \tau \models \delta \text{ self}; \tau \models \text{true}; \tau \models \text{POST}' \text{ self}; \bigwedge \text{res} . (\text{res} \triangleq \text{if } \text{self} . \text{boss} \doteq \text{null} \text{ then } \text{Set}\{\text{self} . \text{salary}\} \text{ else } \text{self} . \text{boss} . \text{contents}() \rightarrow \text{including}_{\text{Set}}(\text{self} . \text{salary}) \text{ endif}) = (\text{POST}' \text{ self and } (\text{res} \triangleq \text{BODY self})) \rrbracket \implies (\tau \models E (\text{self} . \text{contents}())) = (\tau \models E (\text{BODY self}))$, one gets the following more practical rewrite rule that is amenable to symbolic evaluation:

theorem *unfold-contents* :

assumes *cp E*

```

and     $\tau \models \delta \text{ self}$ 
shows ( $\tau \models E (\text{self} .\text{contents}())$ ) =
        ( $\tau \models E (\text{if } \text{self} .\text{boss} \doteq \text{null}$ 
           $\text{then } \text{Set}\{\text{self} .\text{salary}\}$ 
           $\text{else } \text{self} .\text{boss} .\text{contents}() \text{-> including}_{\text{Set}}(\text{self} .\text{salary}) \text{ endif})$ )
<proof>

```

Since we have only one interpretation function, we need the corresponding operation on the pre-state:

```

consts contentsATpre :: Person  $\Rightarrow$  Set-Integer (( $1(-).\text{contents}@pre'()$ ) 50)

```

axiomatization where *contentsATpre-def*:

```

(self).contents@pre() = ( $\lambda \tau$ .
  SOME res. let res =  $\lambda -$ . res in
    if  $\tau \models (\delta \text{ self})$ 
    then (( $\tau \models \text{true}$ )  $\wedge$  (* pre *)
      ( $\tau \models (\text{res} \triangleq \text{if } (\text{self} .\text{boss}@pre \doteq \text{null}) \text{ (* post *)}$ 
         $\text{then } \text{Set}\{(\text{self} .\text{salary}@pre)\}$ 
         $\text{else } (\text{self} .\text{boss}@pre .\text{contents}@pre()$ 
           $\text{-> including}_{\text{Set}}(\text{self} .\text{salary}@pre)$ 
           $\text{endif})$ ))
    else  $\tau \models \text{res} \triangleq \text{invalid}$ )
and cp0-contents-at-pre:(X .contents@pre())  $\tau$  = (( $\lambda -$ . X  $\tau$ ) .contents@pre())  $\tau$ 

```

```

interpretation contentsATpre : contract0 contentsATpre  $\lambda \text{ self}$ . true
   $\lambda \text{ self } \text{res}$ . res  $\triangleq \text{if } (\text{self} .\text{boss}@pre \doteq \text{null})$ 
    then (Set{self .salary@pre})
    else (self .boss@pre .contents@pre()
       $\text{-> including}_{\text{Set}}(\text{self} .\text{salary}@pre)$ )
    endif
<proof>

```

Again, we derive via *contents.unfold2* a Knaster-Tarski like Fixpoint rule that is amenable to symbolic evaluation:

```

theorem unfold-contentsATpre :
assumes cp E
and     $\tau \models \delta \text{ self}$ 
shows ( $\tau \models E (\text{self} .\text{contents}@pre())$ ) =
        ( $\tau \models E (\text{if } \text{self} .\text{boss}@pre \doteq \text{null}$ 
           $\text{then } \text{Set}\{\text{self} .\text{salary}@pre\}$ 
           $\text{else } \text{self} .\text{boss}@pre .\text{contents}@pre() \text{-> including}_{\text{Set}}(\text{self} .\text{salary}@pre) \text{ endif})$ )
<proof>

```

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()  $\text{-> including}$ (x)

```

This boils down to:

```

definition insert :: Person  $\Rightarrow$  Integer  $\Rightarrow$  Void (( $1(-).\text{insert}'(-)$ ) 50)

```

where $self.insert(x) \equiv$
 $(\lambda \tau. SOME \ res. \ let \ res = \lambda -. \ res \ in$
 $\quad if \ (\tau \models (\delta \ self)) \wedge \ (\tau \models v \ x)$
 $\quad then \ (\tau \models true \wedge$
 $\quad \quad (\tau \models ((self).contents()) \triangleq (self).contents@pre()->including_{Set}(x)))$
 $\quad else \ \tau \models res \triangleq invalid)$

The semantic consequences of this definition were computed inside this locale interpretation:

interpretation $insert : contract1 \ insert \ \lambda \ self \ x. \ true$
 $\lambda \ self \ x \ res. \ ((self \ .contents()) \triangleq$
 $\quad (self \ .contents@pre()->including_{Set}(x)))$
 $\langle proof \rangle$

The result of this locale interpretation for our *Analysis-OCL.insert* contract is the following set of properties, which serves as basis for automated deduction on them:

Name	Theorem
$insert.strict0$	$(invalid.insert(X)) = invalid$
$insert.nullstrict0$	$(null.insert(X)) = invalid$
$insert.strict1$	$(self.insert(invalid)) = invalid$
$insert.cpPRE$	$true \ \tau = true \ \tau$
$insert.cpPOST$	$(self.contents() \triangleq self.contents@pre()->including_{Set}(a1.0)) \ \tau = (\lambda -. \ self$ $\tau.contents() \triangleq \lambda -. \ self \ \tau.contents@pre()->including_{Set}(\lambda -. \ a1.0 \ \tau)) \ \tau$
$insert.cp-pre$	$\llbracket cp \ self'; \ cp \ a1 \rrbracket \implies cp \ (\lambda X. \ true)$
$insert.cp-post$	$\llbracket cp \ self'; \ cp \ a1'; \ cp \ res \rrbracket \implies cp \ (\lambda X. \ self' \ X.contents() \triangleq self'$ $X.contents@pre()->including_{Set}(a1' \ X))$
$insert.cp$	$\llbracket cp \ self'; \ cp \ a1'; \ cp \ res \rrbracket \implies cp \ (\lambda X. \ self' \ X.insert(a1' \ X))$
$insert.cp0$	$(self.insert(a1.0)) \ \tau = (\lambda -. \ self \ \tau.insert(\lambda -. \ a1.0 \ \tau)) \ \tau$
$insert.def-scheme$	$self.insert(a1.0) \equiv \lambda \tau. \ SOME \ res. \ let \ res = \lambda -. \ res \ in \ if \ \tau \models \delta \ self \wedge \ \tau \models v$ $\quad a1.0 \ then \ \tau \models true \wedge \ \tau \models self.contents() \triangleq$ $\quad self.contents@pre()->including_{Set}(a1.0) \ else \ \tau \models res \triangleq invalid$
$insert.unfold$	$\llbracket cp \ E; \ \tau \models \delta \ self \wedge \ \tau \models v \ a1.0; \ \tau \models true; \ \exists \ res. \ \tau \models self.contents() \triangleq$ $\quad self.contents@pre()->including_{Set}(a1.0); \ \bigwedge res. \ \tau \models self.contents() \triangleq$ $\quad self.contents@pre()->including_{Set}(a1.0) \implies \tau \models E \ (\lambda -. \ res) \rrbracket \implies \tau \models E$ $\quad (self.insert(a1.0))$
$insert.unfold2$	$\llbracket cp \ E; \ \tau \models \delta \ self \wedge \ \tau \models v \ a1.0; \ \tau \models true; \ \tau \models POST' \ self \ a1.0; \ \bigwedge res.$ $\quad (self.contents() \triangleq self.contents@pre()->including_{Set}(a1.0)) = (POST' \ self$ $\quad a1.0 \ and \ (res \triangleq BODY \ self \ a1.0)) \rrbracket \implies (\tau \models E \ (self.insert(a1.0))) = (\tau \models E$ $\quad (BODY \ self \ a1.0))$

Table 4.1.: Semantic properties resulting from a user-defined operation contract.

end

5. Example: The Employee Design Model

```
theory
  Design-UML
imports
  ../../../../src/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.

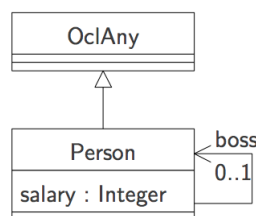


Figure 5.1.: A simple UML class model drawn from Figure 7.3, page 20 of [32].

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 <proof>
end
```

```
instantiation typeOclAny :: object
begin
  definition oid-of-typeOclAny-def: oid-of x = (case x of mkOclAny oid - ⇒ oid)
  instance <proof>
end
```

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

5.3. Instantiation of the Generic Strict Equality

We instantiate the referential equality on *Person* and *OclAny*

defs(overloaded) $StrictRefEq_{Object-Person} : (x::Person) \doteq y \equiv StrictRefEq_{Object} x y$
defs(overloaded) $StrictRefEq_{Object-OclAny} : (x::OclAny) \doteq y \equiv StrictRefEq_{Object} x y$

lemmas $cps23 =$
 $cp-StrictRefEq_{Object}$ [of $x::Person y::Person \tau$,
 $simplified\ StrictRefEq_{Object-Person}[symmetric]$]
 $cp-intro(9)$ [of $P::Person \Rightarrow PersonQ::Person \Rightarrow Person$,
 $simplified\ StrictRefEq_{Object-Person}[symmetric]$]
 $StrictRefEq_{Object-def}$ [of $x::Person y::Person$,
 $simplified\ StrictRefEq_{Object-Person}[symmetric]$]
 $StrictRefEq_{Object-defargs}$ [of $- x::Person y::Person$,
 $simplified\ StrictRefEq_{Object-Person}[symmetric]$]
 $StrictRefEq_{Object-strict1}$
[of $x::Person$,
 $simplified\ StrictRefEq_{Object-Person}[symmetric]$]
 $StrictRefEq_{Object-strict2}$
[of $x::Person$,
 $simplified\ StrictRefEq_{Object-Person}[symmetric]$]

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

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

5.4. OclAsType

5.4.1. Definition

consts $OclAsType_{OclAny} :: 'a \Rightarrow OclAny ((-) .oclAsType'(OclAny'))$
consts $OclAsType_{Person} :: 'a \Rightarrow Person ((-) .oclAsType'(Person'))$

definition $OclAsType_{OclAny}\text{-}\mathfrak{A} = (\lambda u. \text{case } u \text{ of } in_{OclAny} a \Rightarrow a$
 $| in_{Person} (mk_{Person} oid a b) \Rightarrow mk_{OclAny} oid \lfloor(a,b)\rfloor)$

lemma $OclAsType_{OclAny}\text{-}\mathfrak{A}\text{-some}: OclAsType_{OclAny}\text{-}\mathfrak{A} x \neq None$

(proof)

defs (overloaded) $OclAsType_{OclAny}\text{-}OclAny:$
 $(X::OclAny) .oclAsType(OclAny) \equiv X$

defs (overloaded) $OclAsType_{OclAny}\text{-}Person:$
 $(X::Person) .oclAsType(OclAny) \equiv$
 $(\lambda \tau. \text{case } X \tau \text{ of}$
 $\perp \Rightarrow invalid \tau$
 $| \lfloor \perp \rfloor \Rightarrow null \tau$
 $| \lfloor mk_{Person} oid a b \rfloor \Rightarrow \lfloor mk_{OclAny} oid \lfloor(a,b)\rfloor \rfloor)$

definition $OclAsType_{Person}\text{-}\mathfrak{A} =$
 $(\lambda u. \text{case } u \text{ of } in_{Person} p \Rightarrow \lfloor p \rfloor$
 $| in_{OclAny} (mk_{OclAny} oid \lfloor(a,b)\rfloor) \Rightarrow \lfloor mk_{Person} oid a b \rfloor$
 $| - \Rightarrow None)$

defs (overloaded) $OclAsType_{Person}\text{-}OclAny:$
 $(X::OclAny) .oclAsType(Person) \equiv$
 $(\lambda \tau. \text{case } X \tau \text{ of}$

$$\begin{aligned}
& \perp \Rightarrow \text{invalid } \tau \\
& | \lfloor \perp \rfloor \Rightarrow \text{null } \tau \\
& | \lfloor \text{mk}_{OclAny} \text{ oid } \perp \rfloor \Rightarrow \text{invalid } \tau \quad (* \text{ down-cast exception } *) \\
& | \lfloor \text{mk}_{OclAny} \text{ oid } \lfloor (a,b) \rfloor \rfloor \Rightarrow \lfloor \text{mk}_{Person} \text{ oid } a \ b \rfloor
\end{aligned}$$

defs (overloaded) $OclAsType_{Person-Person}$:
 $(X::Person) .oclAsType(Person) \equiv X$ **lemmas [simp] =**
 $OclAsType_{OclAny-OclAny}$
 $OclAsType_{Person-Person}$

5.4.2. Context Passing

lemma $cp-OclAsType_{OclAny-Person-Person}$: $cp \ P \Rightarrow cp(\lambda X. (P \ (X::Person)::Person) .oclAsType(OclAny))$
 $\langle \text{proof} \rangle$

lemma $cp-OclAsType_{OclAny-OclAny-OclAny}$: $cp \ P \Rightarrow cp(\lambda X. (P \ (X::OclAny)::OclAny) .oclAsType(OclAny))$
 $\langle \text{proof} \rangle$

lemma $cp-OclAsType_{Person-Person-Person}$: $cp \ P \Rightarrow cp(\lambda X. (P \ (X::Person)::Person) .oclAsType(Person))$
 $\langle \text{proof} \rangle$

lemma $cp-OclAsType_{Person-OclAny-OclAny}$: $cp \ P \Rightarrow cp(\lambda X. (P \ (X::OclAny)::OclAny) .oclAsType(Person))$
 $\langle \text{proof} \rangle$

lemma $cp-OclAsType_{OclAny-Person-OclAny}$: $cp \ P \Rightarrow cp(\lambda X. (P \ (X::Person)::OclAny) .oclAsType(OclAny))$
 $\langle \text{proof} \rangle$

lemma $cp-OclAsType_{OclAny-OclAny-Person}$: $cp \ P \Rightarrow cp(\lambda X. (P \ (X::OclAny)::Person) .oclAsType(OclAny))$
 $\langle \text{proof} \rangle$

lemma $cp-OclAsType_{Person-Person-OclAny}$: $cp \ P \Rightarrow cp(\lambda X. (P \ (X::Person)::OclAny) .oclAsType(Person))$
 $\langle \text{proof} \rangle$

lemma $cp-OclAsType_{Person-OclAny-Person}$: $cp \ P \Rightarrow cp(\lambda X. (P \ (X::OclAny)::Person) .oclAsType(Person))$
 $\langle \text{proof} \rangle$

lemmas [simp] =
 $cp-OclAsType_{OclAny-Person-Person}$
 $cp-OclAsType_{OclAny-OclAny-OclAny}$
 $cp-OclAsType_{Person-Person-Person}$
 $cp-OclAsType_{Person-OclAny-OclAny}$

 $cp-OclAsType_{OclAny-Person-OclAny}$
 $cp-OclAsType_{OclAny-OclAny-Person}$
 $cp-OclAsType_{Person-Person-OclAny}$
 $cp-OclAsType_{Person-OclAny-Person}$

5.4.3. Execution with Invalid or Null as Argument

lemma $OclAsType_{OclAny-OclAny-strict}$: $(\text{invalid}::OclAny) .oclAsType(OclAny) = \text{invalid}$ $\langle \text{proof} \rangle$

lemma $OclAsType_{OclAny-OclAny-nullstrict}$: $(\text{null}::OclAny) .oclAsType(OclAny) = \text{null}$ $\langle \text{proof} \rangle$

lemma $OclAsType_{OclAny-Person-strict[simp]}$: $(\text{invalid}::Person) .oclAsType(OclAny) = \text{invalid}$
 $\langle \text{proof} \rangle$

lemma $OclAsType_{OclAny-Person-nullstrict[simp]}$: $(\text{null}::Person) .oclAsType(OclAny) = \text{null}$
 $\langle \text{proof} \rangle$

lemma $OclAsType_{Person-OclAny-strict[simp]}$: $(\text{invalid}::OclAny) .oclAsType(Person) = \text{invalid}$
 $\langle \text{proof} \rangle$

lemma $OclAsType_{Person-OclAny-nullstrict[simp]}$: $(\text{null}::OclAny) .oclAsType(Person) = \text{null}$
 $\langle \text{proof} \rangle$

lemma $OclAsType_{Person-Person-strict}$: $(\text{invalid}::Person) .oclAsType(Person) = \text{invalid}$ $\langle \text{proof} \rangle$

lemma $OclAsType_{Person-Person-nullstrict}$: $(\text{null}::Person) .oclAsType(Person) = \text{null}$ $\langle \text{proof} \rangle$

5.5. OclIsTypeOf

5.5.1. Definition

consts $OclIsTypeOf_{OclAny} :: 'a \Rightarrow Boolean ((-).oclIsTypeOf'(OclAny'))$
consts $OclIsTypeOf_{Person} :: 'a \Rightarrow Boolean ((-).oclIsTypeOf'(Person'))$

defs (overloaded) $OclIsTypeOf_{OclAny-OclAny}$:
 $(X::OclAny) .oclIsTypeOf(OclAny) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \ \text{of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \perp_{\perp} \Rightarrow \text{true } \tau \ (* \ \text{invalid } ?? \ *)$
 $\quad | \perp_{mk_{OclAny} \ oid} \perp_{\perp} \Rightarrow \text{true } \tau$
 $\quad | \perp_{mk_{OclAny} \ oid} \perp_{\perp} \Rightarrow \text{false } \tau)$

lemma $OclIsTypeOf_{OclAny-OclAny}'$:
 $(X::OclAny) .oclIsTypeOf(OclAny) =$
 $(\lambda\tau. \text{if } \tau \models v \ X \ \text{then } (\text{case } X \ \tau \ \text{of}$
 $\quad \perp_{\perp} \Rightarrow \text{true } \tau \ (* \ \text{invalid } ?? \ *)$
 $\quad | \perp_{mk_{OclAny} \ oid} \perp_{\perp} \Rightarrow \text{true } \tau$
 $\quad | \perp_{mk_{OclAny} \ oid} \perp_{\perp} \Rightarrow \text{false } \tau)$
 $\text{else } \text{invalid } \tau)$
 $\langle \text{proof} \rangle$

interpretation $OclIsTypeOf_{OclAny-OclAny}$:
 $\text{profile-mono-schemeV}$
 $OclIsTypeOf_{OclAny}::OclAny \Rightarrow Boolean$
 $\lambda X. (\text{case } X \ \text{of}$
 $\quad \perp_{None} \Rightarrow \perp_{True_{\perp}} \ (* \ \text{invalid } ?? \ *)$
 $\quad | \perp_{mk_{OclAny} \ oid} \perp_{None} \Rightarrow \perp_{True_{\perp}}$
 $\quad | \perp_{mk_{OclAny} \ oid} \perp_{\perp} \Rightarrow \perp_{False_{\perp}})$
 $\langle \text{proof} \rangle$

defs (overloaded) $OclIsTypeOf_{OclAny-Person}$:
 $(X::Person) .oclIsTypeOf(OclAny) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \ \text{of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \perp_{\perp} \Rightarrow \text{true } \tau \ (* \ \text{invalid } ?? \ *)$
 $\quad | \perp_{\perp} \Rightarrow \text{false } \tau)$

defs (overloaded) $OclIsTypeOf_{Person-OclAny}$:
 $(X::OclAny) .oclIsTypeOf(Person) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \ \text{of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \perp_{\perp} \Rightarrow \text{true } \tau$
 $\quad | \perp_{mk_{OclAny} \ oid} \perp_{\perp} \Rightarrow \text{false } \tau$
 $\quad | \perp_{mk_{OclAny} \ oid} \perp_{\perp} \Rightarrow \text{true } \tau)$

defs (overloaded) $OclIsTypeOf_{Person-Person}$:
 $(X::Person) .oclIsTypeOf(Person) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \ \text{of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | - \Rightarrow \text{true } \tau)$

5.5.2. Context Passing

lemma $cp\text{-}OclIsTypeOf_{OclAny-Person-Person}$: $cp \ P \Longrightarrow cp(\lambda X. (P(X::Person)::Person).oclIsTypeOf(OclAny))$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

lemmas [simp] =

$cp\text{-}OclIsTypeOf_{OclAny}\text{-}Person\text{-}Person$

$cp\text{-}OclIsTypeOf_{OclAny}\text{-}OclAny\text{-}OclAny$

$cp\text{-}OclIsTypeOf_{Person}\text{-}Person\text{-}Person$

$cp\text{-}OclIsTypeOf_{Person}\text{-}OclAny\text{-}OclAny$

$cp\text{-}OclIsTypeOf_{OclAny}\text{-}Person\text{-}OclAny$

$cp\text{-}OclIsTypeOf_{OclAny}\text{-}OclAny\text{-}Person$

$cp\text{-}OclIsTypeOf_{Person}\text{-}Person\text{-}OclAny$

$cp\text{-}OclIsTypeOf_{Person}\text{-}OclAny\text{-}Person$

5.5.3. Execution with Invalid or Null as Argument

lemma $OclIsTypeOf_{OclAny}\text{-}OclAny\text{-}strict1$ [simp]:
 $(invalid::OclAny).oclIsTypeOf(OclAny) = invalid$
 ⟨proof⟩

lemma $OclIsTypeOf_{OclAny}\text{-}OclAny\text{-}strict2$ [simp]:
 $(null::OclAny).oclIsTypeOf(OclAny) = true$
 ⟨proof⟩

lemma $OclIsTypeOf_{OclAny}\text{-}Person\text{-}strict1$ [simp]:
 $(invalid::Person).oclIsTypeOf(OclAny) = invalid$
 ⟨proof⟩

lemma $OclIsTypeOf_{OclAny}\text{-}Person\text{-}strict2$ [simp]:
 $(null::Person).oclIsTypeOf(OclAny) = true$
 ⟨proof⟩

lemma $OclIsTypeOf_{Person}\text{-}OclAny\text{-}strict1$ [simp]:
 $(invalid::OclAny).oclIsTypeOf(Person) = invalid$
 ⟨proof⟩

lemma $OclIsTypeOf_{Person}\text{-}OclAny\text{-}strict2$ [simp]:
 $(null::OclAny).oclIsTypeOf(Person) = true$
 ⟨proof⟩

lemma $OclIsTypeOf_{Person}\text{-}Person\text{-}strict1$ [simp]:
 $(invalid::Person).oclIsTypeOf(Person) = invalid$
 ⟨proof⟩

lemma $OclIsTypeOf_{Person}\text{-}Person\text{-}strict2$ [simp]:
 $(null::Person).oclIsTypeOf(Person) = true$
 ⟨proof⟩

5.5.4. Up Down Casting

lemma *actualType-larger-staticType*:

assumes *isdef*: $\tau \models (\delta X)$

shows $\tau \models (X::Person) .oclIsTypeOf(OclAny) \triangleq false$

<proof>

lemma *down-cast-type*:

assumes *isOclAny*: $\tau \models (X::OclAny) .oclIsTypeOf(OclAny)$

and *non-null*: $\tau \models (\delta X)$

shows $\tau \models (X .oclAsType(Person)) \triangleq invalid$

<proof>

lemma *down-cast-type'*:

assumes *isOclAny*: $\tau \models (X::OclAny) .oclIsTypeOf(OclAny)$

and *non-null*: $\tau \models (\delta X)$

shows $\tau \models not (v (X .oclAsType(Person)))$

<proof>

lemma *up-down-cast* :

assumes *isdef*: $\tau \models (\delta X)$

shows $\tau \models ((X::Person) .oclAsType(OclAny) .oclAsType(Person)) \triangleq X$

<proof>

lemma *up-down-cast-Person-OclAny-Person* [*simp*]:

shows $((X::Person) .oclAsType(OclAny) .oclAsType(Person) = X)$

<proof>

lemma *up-down-cast-Person-OclAny-Person'*:

assumes $\tau \models v X$

shows $\tau \models (((X :: Person) .oclAsType(OclAny) .oclAsType(Person)) \doteq X)$

<proof>

lemma *up-down-cast-Person-OclAny-Person''*:

assumes $\tau \models v (X :: Person)$

shows $\tau \models (X .oclIsTypeOf(Person) \text{ implies } (X .oclAsType(OclAny) .oclAsType(Person)) \doteq X)$

<proof>

5.6. OclIsKindOf

5.6.1. Definition

consts *OclIsKindOf_{OclAny}* :: $'\alpha \Rightarrow Boolean ((-).oclIsKindOf'(OclAny'))$

consts *OclIsKindOf_{Person}* :: $'\alpha \Rightarrow Boolean ((-).oclIsKindOf'(Person'))$

defs (**overloaded**) *OclIsKindOf_{OclAny-OclAny}*:

$(X::OclAny) .oclIsKindOf(OclAny) \equiv$

$(\lambda\tau. \text{ case } X \tau \text{ of}$
 $\quad \perp \Rightarrow invalid \tau$
 $\quad | - \Rightarrow true \tau)$

defs (**overloaded**) *OclIsKindOf_{OclAny-Person}*:

$(X::Person) .oclIsKindOf(OclAny) \equiv$

$(\lambda\tau. \text{ case } X \tau \text{ of}$
 $\quad \perp \Rightarrow invalid \tau$
 $\quad | - \Rightarrow true \tau)$

defs (overloaded) $OclIsKindOf_{Person-OclAny}$:
 $(X::OclAny) .oclIsKindOf(Person) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | \perp \Rightarrow \text{true } \tau$
 $\quad | \perp mk_{OclAny} \text{ oid } \perp \perp \Rightarrow \text{false } \tau$
 $\quad | \perp mk_{OclAny} \text{ oid } \perp \perp \Rightarrow \text{true } \tau)$

defs (overloaded) $OclIsKindOf_{Person-Person}$:
 $(X::Person) .oclIsKindOf(Person) \equiv$
 $(\lambda\tau. \text{case } X \ \tau \text{ of}$
 $\quad \perp \Rightarrow \text{invalid } \tau$
 $\quad | - \Rightarrow \text{true } \tau)$

5.6.2. Context Passing

lemma $cp-OclIsKindOf_{OclAny-Person-Person}$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::Person)::Person).oclIsKindOf(OclAny))$
 $\langle \text{proof} \rangle$

lemma $cp-OclIsKindOf_{OclAny-OclAny-OclAny}$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::OclAny).oclIsKindOf(OclAny))$
 $\langle \text{proof} \rangle$

lemma $cp-OclIsKindOf_{Person-Person-Person}$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::Person)::Person).oclIsKindOf(Person))$
 $\langle \text{proof} \rangle$

lemma $cp-OclIsKindOf_{Person-OclAny-OclAny}$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::OclAny).oclIsKindOf(Person))$
 $\langle \text{proof} \rangle$

lemma $cp-OclIsKindOf_{OclAny-Person-OclAny}$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(OclAny))$
 $\langle \text{proof} \rangle$

lemma $cp-OclIsKindOf_{OclAny-OclAny-Person}$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(OclAny))$
 $\langle \text{proof} \rangle$

lemma $cp-OclIsKindOf_{Person-Person-OclAny}$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::Person)::OclAny).oclIsKindOf(Person))$
 $\langle \text{proof} \rangle$

lemma $cp-OclIsKindOf_{Person-OclAny-Person}$: $cp \ P \Longrightarrow cp(\lambda X.(P(X::OclAny)::Person).oclIsKindOf(Person))$
 $\langle \text{proof} \rangle$

lemmas [simp] =

$cp-OclIsKindOf_{OclAny-Person-Person}$
 $cp-OclIsKindOf_{OclAny-OclAny-OclAny}$
 $cp-OclIsKindOf_{Person-Person-Person}$
 $cp-OclIsKindOf_{Person-OclAny-OclAny}$

$cp-OclIsKindOf_{OclAny-Person-OclAny}$
 $cp-OclIsKindOf_{OclAny-OclAny-Person}$
 $cp-OclIsKindOf_{Person-Person-OclAny}$
 $cp-OclIsKindOf_{Person-OclAny-Person}$

5.6.3. Execution with Invalid or Null as Argument

lemma $OclIsKindOf_{OclAny-OclAny-strict1}$ [simp]: $(invalid::OclAny) .oclIsKindOf(OclAny) = \text{invalid}$
 $\langle \text{proof} \rangle$

lemma $OclIsKindOf_{OclAny-OclAny-strict2}$ [simp]: $(null::OclAny) .oclIsKindOf(OclAny) = \text{true}$
 $\langle \text{proof} \rangle$

lemma $OclIsKindOf_{OclAny-Person-strict1}$ [simp]: $(invalid::Person) .oclIsKindOf(OclAny) = \text{invalid}$
 $\langle \text{proof} \rangle$

lemma $OclIsKindOf_{OclAny-Person-strict2}$ [simp]: $(null::Person) .oclIsKindOf(OclAny) = \text{true}$

$\langle \text{proof} \rangle$
lemma $OclIsKindOf_{Person-OclAny-strict1}[simp]: (invalid::OclAny) .oclIsKindOf(Person) = invalid$
 $\langle \text{proof} \rangle$
lemma $OclIsKindOf_{Person-OclAny-strict2}[simp]: (null::OclAny) .oclIsKindOf(Person) = true$
 $\langle \text{proof} \rangle$
lemma $OclIsKindOf_{Person-Person-strict1}[simp]: (invalid::Person) .oclIsKindOf(Person) = invalid$
 $\langle \text{proof} \rangle$
lemma $OclIsKindOf_{Person-Person-strict2}[simp]: (null::Person) .oclIsKindOf(Person) = true$
 $\langle \text{proof} \rangle$

5.6.4. Up Down Casting

lemma $actualKind-larger-staticKind:$
assumes $isdef: \tau \models (\delta X)$
shows $\tau \models ((X::Person) .oclIsKindOf(OclAny) \triangleq true)$
 $\langle \text{proof} \rangle$

lemma $down-cast-kind:$
assumes $isOclAny: \neg (\tau \models ((X::OclAny).oclIsKindOf(Person)))$
and $non-null: \tau \models (\delta X)$
shows $\tau \models ((X .oclAsType(Person)) \triangleq invalid)$
 $\langle \text{proof} \rangle$

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 $Person \equiv OclAsType_{Person-\mathfrak{A}}$
definition $OclAny \equiv OclAsType_{OclAny-\mathfrak{A}}$
lemmas $[simp] = Person-def\ OclAny-def$

lemma $OclAllInstances-generic_{OclAny-exec}: OclAllInstances-generic\ pre-post\ OclAny =$
 $(\lambda\tau. Abs-Set_{base} \perp\!\!\!\perp Some\ 'OclAny'\ ran\ (heap\ (pre-post\ \tau)) \perp\!\!\!\perp)$
 $\langle \text{proof} \rangle$

lemma $OclAllInstances-at-post_{OclAny-exec}: OclAny .allInstances() =$
 $(\lambda\tau. Abs-Set_{base} \perp\!\!\!\perp Some\ 'OclAny'\ ran\ (heap\ (snd\ \tau)) \perp\!\!\!\perp)$
 $\langle \text{proof} \rangle$

lemma $OclAllInstances-at-pre_{OclAny-exec}: OclAny .allInstances@pre() =$
 $(\lambda\tau. Abs-Set_{base} \perp\!\!\!\perp Some\ 'OclAny'\ ran\ (heap\ (fst\ \tau)) \perp\!\!\!\perp)$
 $\langle \text{proof} \rangle$

5.7.1. OclIsTypeOf

lemma $OclAny-allInstances-generic-oclIsTypeOf_{OclAny}1:$
assumes $[simp]: \bigwedge x. pre-post\ (x, x) = x$
shows $\exists \tau. (\tau \models ((OclAllInstances-generic\ pre-post\ OclAny) \rightarrow forAll_{Set}(X|X .oclIsTypeOf(OclAny))))$
 $\langle \text{proof} \rangle$

lemma $OclAny-allInstances-at-post-oclIsTypeOf_{OclAny}1:$
 $\exists \tau. (\tau \models (OclAny .allInstances() \rightarrow forAll_{Set}(X|X .oclIsTypeOf(OclAny))))$
 $\langle \text{proof} \rangle$

lemma $OclAny-allInstances-at-pre-oclIsTypeOf_{OclAny}1:$

$\exists \tau. (\tau \models (OclAny .allInstances@pre() \rightarrow forAll_{Set}(X|X .oclIsTypeOf(OclAny))))$
 $\langle proof \rangle$

lemma *OclAny-allInstances-generic-oclIsTypeOf_{OclAny}2*:

assumes [*simp*]: $\bigwedge x. pre\text{-}post(x, x) = x$

shows $\exists \tau. (\tau \models not((OclAllInstances\text{-}generic\ pre\text{-}post\ OclAny) \rightarrow forAll_{Set}(X|X .oclIsTypeOf(OclAny))))$
 $\langle proof \rangle$

lemma *OclAny-allInstances-at-post-oclIsTypeOf_{OclAny}2*:

$\exists \tau. (\tau \models not(OclAny .allInstances() \rightarrow forAll_{Set}(X|X .oclIsTypeOf(OclAny))))$
 $\langle proof \rangle$

lemma *OclAny-allInstances-at-pre-oclIsTypeOf_{OclAny}2*:

$\exists \tau. (\tau \models not(OclAny .allInstances@pre() \rightarrow forAll_{Set}(X|X .oclIsTypeOf(OclAny))))$
 $\langle proof \rangle$

lemma *Person-allInstances-generic-oclIsTypeOf_{Person}*:

$\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ Person) \rightarrow forAll_{Set}(X|X .oclIsTypeOf(Person)))$
 $\langle proof \rangle$

lemma *Person-allInstances-at-post-oclIsTypeOf_{Person}*:

$\tau \models (Person .allInstances() \rightarrow forAll_{Set}(X|X .oclIsTypeOf(Person)))$
 $\langle proof \rangle$

lemma *Person-allInstances-at-pre-oclIsTypeOf_{Person}*:

$\tau \models (Person .allInstances@pre() \rightarrow forAll_{Set}(X|X .oclIsTypeOf(Person)))$
 $\langle proof \rangle$

5.7.2. OclIsKindOf

lemma *OclAny-allInstances-generic-oclIsKindOf_{OclAny}*:

$\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ OclAny) \rightarrow forAll_{Set}(X|X .oclIsKindOf(OclAny)))$
 $\langle proof \rangle$

lemma *OclAny-allInstances-at-post-oclIsKindOf_{OclAny}*:

$\tau \models (OclAny .allInstances() \rightarrow forAll_{Set}(X|X .oclIsKindOf(OclAny)))$
 $\langle proof \rangle$

lemma *OclAny-allInstances-at-pre-oclIsKindOf_{OclAny}*:

$\tau \models (OclAny .allInstances@pre() \rightarrow forAll_{Set}(X|X .oclIsKindOf(OclAny)))$
 $\langle proof \rangle$

lemma *Person-allInstances-generic-oclIsKindOf_{OclAny}*:

$\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ Person) \rightarrow forAll_{Set}(X|X .oclIsKindOf(OclAny)))$
 $\langle proof \rangle$

lemma *Person-allInstances-at-post-oclIsKindOf_{OclAny}*:

$\tau \models (Person .allInstances() \rightarrow forAll_{Set}(X|X .oclIsKindOf(OclAny)))$
 $\langle proof \rangle$

lemma *Person-allInstances-at-pre-oclIsKindOf_{OclAny}*:

$\tau \models (Person .allInstances@pre() \rightarrow forAll_{Set}(X|X .oclIsKindOf(OclAny)))$
 $\langle proof \rangle$

lemma *Person-allInstances-generic-oclIsKindOf_{Person}*:

$\tau \models ((OclAllInstances\text{-}generic\ pre\text{-}post\ Person) \rightarrow forAll_{Set}(X|X .oclIsKindOf(Person)))$
 $\langle proof \rangle$

lemma *Person-allInstances-at-post-oclIsKindOf_{Person}*:
 $\tau \models (Person \ .allInstances() \rightarrow forAll_{Set}(X|X \ .oclIsKindOf(Person)))$
<proof>

lemma *Person-allInstances-at-pre-oclIsKindOf_{Person}*:
 $\tau \models (Person \ .allInstances@pre() \rightarrow forAll_{Set}(X|X \ .oclIsKindOf(Person)))$
<proof>

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* $X \ f = (\lambda \tau. \ case \ X \ \tau \ of$
 $\quad \perp \Rightarrow \text{invalid } \tau \quad (* \ \text{exception propagation} \ *)$
 $\quad | \ \perp \ \perp \Rightarrow \text{invalid } \tau \quad (* \ \text{dereferencing null pointer} \ *)$
 $\quad | \ \perp \ \text{obj} \ \perp \Rightarrow f \ (oid\text{-of } \text{obj}) \ \tau$

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}* $fst\text{-snd} \ f \ oid = (\lambda \tau. \ case \ (heap \ (fst\text{-snd} \ \tau)) \ oid \ of$
 $\quad | \ \perp \ in_{Person} \ \text{obj} \ \perp \Rightarrow f \ \text{obj} \ \tau$
 $\quad | \ - \ \Rightarrow \text{invalid } \tau$

definition *deref-oid_{OclAny}* :: ($\mathfrak{A} \ state \times \ \mathfrak{A} \ state \Rightarrow \ \mathfrak{A} \ state$)
 $\Rightarrow (type_{OclAny} \Rightarrow (\mathfrak{A}, 'c::null) \ val)$
 $\Rightarrow oid$
 $\Rightarrow (\mathfrak{A}, 'c::null) \ val$
where *deref-oid_{OclAny}* $fst\text{-snd} \ f \ oid = (\lambda \tau. \ case \ (heap \ (fst\text{-snd} \ \tau)) \ oid \ of$
 $\quad | \ \perp \ in_{OclAny} \ \text{obj} \ \perp \Rightarrow f \ \text{obj} \ \tau$
 $\quad | \ - \ \Rightarrow \text{invalid } \tau$

pointer undefined in state or not referencing a type conform object representation

definition *select_{OclAny}ANY* $f = (\lambda X. \ case \ X \ of$
 $\quad (mk_{OclAny} \ - \ \perp) \Rightarrow null$
 $\quad | \ (mk_{OclAny} \ - \ \perp \ any) \Rightarrow f \ (\lambda x \ -. \ \perp \ x) \ any)$

definition *select_{Person}BOSS* $f = (\lambda X. \ case \ X \ of$
 $\quad (mk_{Person} \ - \ - \ \perp) \Rightarrow null \quad (* \ \text{object contains null pointer} \ *)$
 $\quad | \ (mk_{Person} \ - \ - \ \perp \ boss) \Rightarrow f \ (\lambda x \ -. \ \perp \ x) \ boss)$

definition *select_{Person}SALARY* $f = (\lambda X. \ case \ X \ of$
 $\quad (mk_{Person} \ - \ \perp \ -) \Rightarrow null$

| (mk_{Person} - salary) ⇒ f (λx . (x) salary)

definition *in-pre-state* = fst

definition *in-post-state* = snd

definition *reconst-basetype* = (λ convert x. convert x)

definition *dot_{OclAny}ANY* :: *OclAny* ⇒ - ((1(-).any) 50)

where (X).any = eval-extract X
 (deref-oid_{OclAny} in-post-state
 (select_{OclAny}ANY
 reconst-basetype))

definition *dot_{Person}BOSS* :: *Person* ⇒ *Person* ((1(-).boss) 50)

where (X).boss = eval-extract X
 (deref-oid_{Person} in-post-state
 (select_{Person}BOSS
 (deref-oid_{Person} in-post-state)))

definition *dot_{Person}SALARY* :: *Person* ⇒ *Integer* ((1(-).salary) 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* ⇒ - ((1(-).any@pre) 50)

where (X).any@pre = eval-extract X
 (deref-oid_{OclAny} in-pre-state
 (select_{OclAny}ANY
 reconst-basetype))

definition *dot_{Person}BOSS-at-pre* :: *Person* ⇒ *Person* ((1(-).boss@pre) 50)

where (X).boss@pre = eval-extract X
 (deref-oid_{Person} in-pre-state
 (select_{Person}BOSS
 (deref-oid_{Person} in-pre-state)))

definition *dot_{Person}SALARY-at-pre* :: *Person* ⇒ *Integer* ((1(-).salary@pre) 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) τ = ((λ-. X τ).any) τ ⟨proof⟩

lemma *cp-dot_{Person}BOSS*: $((X).boss) \tau = ((\lambda-. X \tau).boss) \tau$ *<proof>*
lemma *cp-dot_{Person}SALARY*: $((X).salary) \tau = ((\lambda-. X \tau).salary) \tau$ *<proof>*

lemma *cp-dot_{OclAny}ANY-at-pre*: $((X).any@pre) \tau = ((\lambda-. X \tau).any@pre) \tau$ *<proof>*
lemma *cp-dot_{Person}BOSS-at-pre*: $((X).boss@pre) \tau = ((\lambda-. X \tau).boss@pre) \tau$ *<proof>*
lemma *cp-dot_{Person}SALARY-at-pre*: $((X).salary@pre) \tau = ((\lambda-. X \tau).salary@pre) \tau$ *<proof>*

lemmas *cp-dot_{OclAny}ANY-I* [*simp, intro!*]=
cp-dot_{OclAny}ANY[*THEN all*[*THEN all*],
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 all*[*THEN all*],
of $\lambda X -. X \lambda - \tau. \tau$, *THEN cpI1*]

lemmas *cp-dot_{Person}BOSS-I* [*simp, intro!*]=
cp-dot_{Person}BOSS[*THEN all*[*THEN all*],
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 all*[*THEN all*],
of $\lambda X -. X \lambda - \tau. \tau$, *THEN cpI1*]

lemmas *cp-dot_{Person}SALARY-I* [*simp, intro!*]=
cp-dot_{Person}SALARY[*THEN all*[*THEN all*],
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 all*[*THEN all*],
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*]: $(null).any = invalid$
<proof>
lemma *dot_{OclAny}ANY-at-pre-nullstrict* [*simp*]: $(null).any@pre = invalid$
<proof>
lemma *dot_{OclAny}ANY-strict* [*simp*]: $(invalid).any = invalid$
<proof>
lemma *dot_{OclAny}ANY-at-pre-strict* [*simp*]: $(invalid).any@pre = invalid$
<proof>

lemma *dot_{Person}BOSS-nullstrict* [*simp*]: $(null).boss = invalid$
<proof>
lemma *dot_{Person}BOSS-at-pre-nullstrict* [*simp*]: $(null).boss@pre = invalid$
<proof>
lemma *dot_{Person}BOSS-strict* [*simp*]: $(invalid).boss = invalid$
<proof>
lemma *dot_{Person}BOSS-at-pre-strict* [*simp*]: $(invalid).boss@pre = invalid$
<proof>

lemma *dot_{Person}SALARY-nullstrict* [*simp*]: $(null).salary = invalid$
<proof>
lemma *dot_{Person}SALARY-at-pre-nullstrict* [*simp*]: $(null).salary@pre = invalid$
<proof>
lemma *dot_{Person}SALARY-strict* [*simp*]: $(invalid).salary = invalid$
<proof>
lemma *dot_{Person}SALARY-at-pre-strict* [*simp*]: $(invalid).salary@pre = invalid$

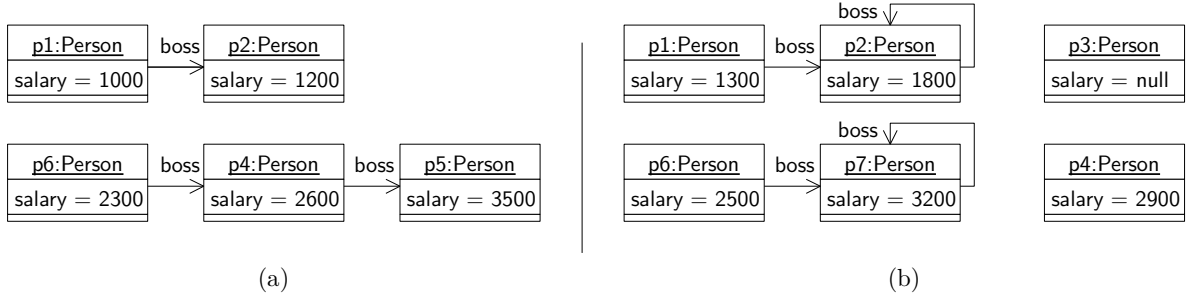


Figure 5.2.: (a) pre-state σ_1 and (b) post-state σ'_1 .

<proof>

5.8.4. Representation in States

lemma $\text{dot}_{Person} \mathcal{BOSS}\text{-def-mono} : \tau \models \delta(X \text{ .boss}) \implies \tau \models \delta(X)$
<proof>

lemma *repr-boss*:

assumes $A : \tau \models \delta(x \text{ .boss})$

shows *is-represented-in-state in-post-state* $(x \text{ .boss})$ *Person* τ
<proof>

lemma *repr-bossX* :

assumes $A : \tau \models \delta(x \text{ .boss})$

shows $\tau \models ((Person \text{ .allInstances}()) \text{ -> includes}_{Set}(x \text{ .boss}))$
<proof>

5.9. A Little Infra-structure on Example States

The example we are defining in this section comes from the figure 5.2.

definition *OclInt1000* (**1000**) **where** $OclInt1000 = (\lambda \text{ . } \llbracket 1000 \rrbracket)$

definition *OclInt1200* (**1200**) **where** $OclInt1200 = (\lambda \text{ . } \llbracket 1200 \rrbracket)$

definition *OclInt1300* (**1300**) **where** $OclInt1300 = (\lambda \text{ . } \llbracket 1300 \rrbracket)$

definition *OclInt1800* (**1800**) **where** $OclInt1800 = (\lambda \text{ . } \llbracket 1800 \rrbracket)$

definition *OclInt2600* (**2600**) **where** $OclInt2600 = (\lambda \text{ . } \llbracket 2600 \rrbracket)$

definition *OclInt2900* (**2900**) **where** $OclInt2900 = (\lambda \text{ . } \llbracket 2900 \rrbracket)$

definition *OclInt3200* (**3200**) **where** $OclInt3200 = (\lambda \text{ . } \llbracket 3200 \rrbracket)$

definition *OclInt3500* (**3500**) **where** $OclInt3500 = (\lambda \text{ . } \llbracket 3500 \rrbracket)$

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* $\equiv mk_{Person} \text{ oid0 } \llbracket 1300 \rrbracket \llbracket oid1 \rrbracket$

definition *person2* $\equiv mk_{Person} \text{ oid1 } \llbracket 1800 \rrbracket \llbracket oid1 \rrbracket$

definition *person3* $\equiv mk_{Person} \text{ oid2 } None \ None$

OclAsTypePerson-OclAny
OclAsTypePerson-Person

OclIsTypeOfOclAny-OclAny
OclIsTypeOfOclAny-Person
OclIsTypeOfPerson-OclAny
OclIsTypeOfPerson-Person

OclIsKindOfOclAny-OclAny
OclIsKindOfOclAny-Person
OclIsKindOfPerson-OclAny

OclIsKindOfPerson-Person **Assert** $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models (X_{Person1} .salary <> 1000)$
Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models (X_{Person1} .salary \doteq 1300)$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models (X_{Person1} .salary@pre \doteq 1000)$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models (X_{Person1} .salary@pre <> 1300)$
Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models (X_{Person1} .boss <> X_{Person1})$
Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models (X_{Person1} .boss .salary \doteq 1800)$
Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models (X_{Person1} .boss .boss <> X_{Person1})$
Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models (X_{Person1} .boss .boss \doteq X_{Person2})$
Assert $(\sigma_1, \sigma_1') \models (X_{Person1} .boss@pre .salary \doteq 1800)$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models (X_{Person1} .boss@pre .salary@pre \doteq 1200)$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models (X_{Person1} .boss@pre .salary@pre <> 1800)$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models (X_{Person1} .boss@pre \doteq X_{Person2})$
Assert $(\sigma_1, \sigma_1') \models (X_{Person1} .boss@pre .boss \doteq X_{Person2})$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models (X_{Person1} .boss@pre .boss@pre \doteq null)$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models not(v(X_{Person1} .boss@pre .boss@pre .boss@pre))$

lemma $(\sigma_1, \sigma_1') \models (X_{Person1} .oclIsMaintained())$
<proof>

lemma $\bigwedge_{s_{pre} s_{post}} . (s_{pre}, s_{post}) \models ((X_{Person1} .oclAsType(OclAny) .oclAsType(Person)) \doteq X_{Person1})$
<proof>

Assert $\bigwedge_{s_{pre} s_{post}} . (s_{pre}, s_{post}) \models (X_{Person1} .oclIsTypeOf(Person))$
Assert $\bigwedge_{s_{pre} s_{post}} . (s_{pre}, s_{post}) \models not(X_{Person1} .oclIsTypeOf(OclAny))$
Assert $\bigwedge_{s_{pre} s_{post}} . (s_{pre}, s_{post}) \models (X_{Person1} .oclIsKindOf(Person))$
Assert $\bigwedge_{s_{pre} s_{post}} . (s_{pre}, s_{post}) \models (X_{Person1} .oclIsKindOf(OclAny))$
Assert $\bigwedge_{s_{pre} s_{post}} . (s_{pre}, s_{post}) \models not(X_{Person1} .oclAsType(OclAny) .oclIsTypeOf(OclAny))$

Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models (X_{Person2} .salary \doteq 1800)$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models (X_{Person2} .salary@pre \doteq 1200)$
Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models (X_{Person2} .boss \doteq X_{Person2})$
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 $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models (X_{Person2} .boss@pre \doteq null)$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models (X_{Person2} .boss@pre <> X_{Person2})$
Assert $(\sigma_1, \sigma_1') \models (X_{Person2} .boss@pre <> (X_{Person2} .boss))$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models not(v(X_{Person2} .boss@pre .boss))$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models not(v(X_{Person2} .boss@pre .salary@pre))$
lemma $(\sigma_1, \sigma_1') \models (X_{Person2} .oclIsMaintained())$
<proof>

Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models (X_{Person3} .salary \doteq null)$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models not(v(X_{Person3} .salary@pre))$
Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models (X_{Person3} .boss \doteq null)$
Assert $\bigwedge_{s_{pre}} . (s_{pre}, \sigma_1') \models not(v(X_{Person3} .boss .salary))$
Assert $\bigwedge_{s_{post}} . (\sigma_1, s_{post}) \models not(v(X_{Person3} .boss@pre))$

lemma $(\sigma_1, \sigma_1') \models (X_{Person3} .oclIsNew())$
 ⟨proof⟩

Assert $\bigwedge_{s_{post}} (\sigma_1, s_{post}) \models (X_{Person4} .boss@pre \doteq X_{Person5})$
Assert $\bigwedge_{s_{post}} (\sigma_1, \sigma_1') \models not(v(X_{Person4} .boss@pre .salary))$
Assert $\bigwedge_{s_{post}} (\sigma_1, s_{post}) \models (X_{Person4} .boss@pre .salary@pre \doteq \mathbf{3500})$
lemma $(\sigma_1, \sigma_1') \models (X_{Person4} .oclIsMaintained())$
 ⟨proof⟩

Assert $\bigwedge_{s_{pre}} (\sigma_1, s_{pre}) \models not(v(X_{Person5} .salary))$
Assert $\bigwedge_{s_{post}} (\sigma_1, s_{post}) \models (X_{Person5} .salary@pre \doteq \mathbf{3500})$
Assert $\bigwedge_{s_{pre}} (\sigma_1, s_{pre}) \models not(v(X_{Person5} .boss))$
lemma $(\sigma_1, \sigma_1') \models (X_{Person5} .oclIsDeleted())$
 ⟨proof⟩

Assert $\bigwedge_{s_{pre}} (\sigma_1, s_{pre}) \models not(v(X_{Person6} .boss .salary@pre))$
Assert $\bigwedge_{s_{post}} (\sigma_1, s_{post}) \models (X_{Person6} .boss@pre \doteq X_{Person4})$
Assert $(\sigma_1, \sigma_1') \models (X_{Person6} .boss@pre .salary \doteq \mathbf{2900})$
Assert $\bigwedge_{s_{post}} (\sigma_1, s_{post}) \models (X_{Person6} .boss@pre .salary@pre \doteq \mathbf{2600})$
Assert $\bigwedge_{s_{post}} (\sigma_1, s_{post}) \models (X_{Person6} .boss@pre .boss@pre \doteq X_{Person5})$
lemma $(\sigma_1, \sigma_1') \models (X_{Person6} .oclIsMaintained())$
 ⟨proof⟩

Assert $\bigwedge_{s_{pre} s_{post}} (\sigma_1, s_{pre}, s_{post}) \models v(X_{Person7} .oclAsType(Person))$
Assert $\bigwedge_{s_{post}} (\sigma_1, s_{post}) \models not(v(X_{Person7} .oclAsType(Person) .boss@pre))$
lemma $\bigwedge_{s_{pre} s_{post}} (\sigma_1, s_{pre}, s_{post}) \models ((X_{Person7} .oclAsType(Person) .oclAsType(OclAny) .oclAsType(Person)) \doteq (X_{Person7} .oclAsType(Person)))$
 ⟨proof⟩
lemma $(\sigma_1, \sigma_1') \models (X_{Person7} .oclIsNew())$
 ⟨proof⟩

Assert $\bigwedge_{s_{pre} s_{post}} (\sigma_1, s_{pre}, s_{post}) \models (X_{Person8} <> X_{Person7})$
Assert $\bigwedge_{s_{pre} s_{post}} (\sigma_1, s_{pre}, s_{post}) \models not(v(X_{Person8} .oclAsType(Person)))$
Assert $\bigwedge_{s_{pre} s_{post}} (\sigma_1, s_{pre}, s_{post}) \models (X_{Person8} .oclIsTypeOf(OclAny))$
Assert $\bigwedge_{s_{pre} s_{post}} (\sigma_1, s_{pre}, s_{post}) \models not(X_{Person8} .oclIsTypeOf(Person))$
Assert $\bigwedge_{s_{pre} s_{post}} (\sigma_1, s_{pre}, s_{post}) \models not(X_{Person8} .oclIsKindOf(Person))$
Assert $\bigwedge_{s_{pre} s_{post}} (\sigma_1, s_{pre}, s_{post}) \models (X_{Person8} .oclIsKindOf(OclAny))$

lemma σ -modifiedonly: $(\sigma_1, \sigma_1') \models (Set\{ X_{Person1} .oclAsType(OclAny) , X_{Person2} .oclAsType(OclAny) (*, X_{Person3} .oclAsType(OclAny)* , X_{Person4} .oclAsType(OclAny) (*, X_{Person5} .oclAsType(OclAny)* , X_{Person6} .oclAsType(OclAny) (*, X_{Person7} .oclAsType(OclAny)* (*, X_{Person8} .oclAsType(OclAny)* (*, X_{Person9} .oclAsType(OclAny)*}\} \rightarrow oclIsModifiedOnly())$
 ⟨proof⟩

lemma $(\sigma_1, \sigma_1') \models ((X_{Person9} @pre (\lambda x. _OclAsType_{Person-2} x)) \doteq X_{Person9})$
 ⟨proof⟩

lemma $(\sigma_1, \sigma_1') \models ((X_{Person9} \text{ @post } (\lambda x. _OclAsType_{Person}\text{-}\mathfrak{A} \ x)) \triangleq X_{Person9})$
 $\langle \text{proof} \rangle$

lemma $(\sigma_1, \sigma_1') \models (((X_{Person9} \text{ .oclAsType}(OclAny)) \text{ @pre } (\lambda x. _OclAsType_{OclAny}\text{-}\mathfrak{A} \ x)) \triangleq$
 $((X_{Person9} \text{ .oclAsType}(OclAny)) \text{ @post } (\lambda x. _OclAsType_{OclAny}\text{-}\mathfrak{A} \ x)))$
 $\langle \text{proof} \rangle$

lemma $\text{perm-}\sigma_1' : \sigma_1' = (\!| \text{ heap} = \text{empty}$
 $(oid8 \mapsto \text{in}_{Person} \ \text{person9})$
 $(oid7 \mapsto \text{in}_{OclAny} \ \text{person8})$
 $(oid6 \mapsto \text{in}_{OclAny} \ \text{person7})$
 $(oid5 \mapsto \text{in}_{Person} \ \text{person6})$
 $(*oid4*)$
 $(oid3 \mapsto \text{in}_{Person} \ \text{person4})$
 $(oid2 \mapsto \text{in}_{Person} \ \text{person3})$
 $(oid1 \mapsto \text{in}_{Person} \ \text{person2})$
 $(oid0 \mapsto \text{in}_{Person} \ \text{person1})$
 $\left. , \text{assoc} = \text{assoc} \ \sigma_1' \right|)$

$\langle \text{proof} \rangle$

declare *const-ss* [*simp*]

lemma $\bigwedge \sigma_1.$
 $(\sigma_1, \sigma_1') \models (\text{Person} \text{ .allInstances}()) \doteq \text{Set}\{ X_{Person1}, X_{Person2}, X_{Person3}, X_{Person4}(*, X_{Person5*}),$
 $X_{Person6},$
 $X_{Person7} \text{ .oclAsType}(\text{Person})(*, X_{Person8*}), X_{Person9} \}$
 $\langle \text{proof} \rangle$

lemma $\bigwedge \sigma_1.$
 $(\sigma_1, \sigma_1') \models (\text{OclAny} \text{ .allInstances}()) \doteq \text{Set}\{ X_{Person1} \text{ .oclAsType}(\text{OclAny}), X_{Person2} \text{ .oclAsType}(\text{OclAny}),$
 $X_{Person3} \text{ .oclAsType}(\text{OclAny}), X_{Person4} \text{ .oclAsType}(\text{OclAny})$
 $(*, X_{Person5*}), X_{Person6} \text{ .oclAsType}(\text{OclAny}),$
 $X_{Person7}, X_{Person8}, X_{Person9} \text{ .oclAsType}(\text{OclAny}) \}$
 $\langle \text{proof} \rangle$

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 $\text{Person-label}_{inv} :: \text{Person} \Rightarrow \text{Boolean}$

where $Person\text{-}label_{inv}(self) \equiv$
 $(self.boss \langle \rangle null \text{ implies } (self.salary \leq_{int} ((self.boss).salary)))$

definition $Person\text{-}label_{invATpre} :: Person \Rightarrow Boolean$

where $Person\text{-}label_{invATpre}(self) \equiv$
 $(self.boss@pre \langle \rangle null \text{ implies } (self.salary@pre \leq_{int} ((self.boss@pre).salary@pre)))$

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 $\tau \models \delta(X.boss) \implies \tau \models Person.allInstances() \rightarrow \text{includes}_{Set}(X.boss) \wedge$
 $\tau \models Person.allInstances() \rightarrow \text{includes}_{Set}(X)$

<proof>

lemma $REC\text{-}pre : \tau \models Person\text{-}label_{globalinv}$

$\implies \tau \models Person.allInstances() \rightarrow \text{includes}_{Set}(X) (* X \text{ represented object in state } *)$
 $\implies \exists REC. \tau \models REC(X) \triangleq (Person\text{-}label_{inv}(X) \text{ and } (X.boss \langle \rangle null \text{ implies } REC(X.boss)))$

<proof>

This allows to state a predicate:

axiomatization $inv_{Person\text{-}label} :: Person \Rightarrow Boolean$

where $inv_{Person\text{-}label}\text{-}def:$

$(\tau \models Person.allInstances() \rightarrow \text{includes}_{Set}(self)) \implies$
 $(\tau \models (inv_{Person\text{-}label}(self) \triangleq (self.boss \langle \rangle null \text{ implies}$
 $(self.salary \leq_{int} ((self.boss).salary)) \text{ and}$
 $inv_{Person\text{-}label}(self.boss))))$

axiomatization $inv_{Person\text{-}labelATpre} :: Person \Rightarrow Boolean$

where $inv_{Person\text{-}labelATpre}\text{-}def:$

$(\tau \models Person.allInstances@pre() \rightarrow \text{includes}_{Set}(self)) \implies$
 $(\tau \models (inv_{Person\text{-}labelATpre}(self) \triangleq (self.boss@pre \langle \rangle null \text{ implies}$
 $(self.salary@pre \leq_{int} ((self.boss@pre).salary@pre)) \text{ and}$
 $inv_{Person\text{-}labelATpre}(self.boss@pre))))$

lemma $inv\text{-}1 :$

$(\tau \models Person.allInstances() \rightarrow \text{includes}_{Set}(self)) \implies$
 $(\tau \models inv_{Person\text{-}label}(self) = ((\tau \models (self.boss \doteq null)) \vee$
 $(\tau \models (self.boss \langle \rangle null) \wedge$
 $\tau \models ((self.salary) \leq_{int} (self.boss.salary)) \wedge$
 $\tau \models (inv_{Person\text{-}label}(self.boss))))$

<proof>

lemma $inv\text{-}2 :$

$(\tau \models Person.allInstances@pre() \rightarrow \text{includes}_{Set}(self)) \implies$
 $(\tau \models inv_{Person\text{-}labelATpre}(self) = ((\tau \models (self.boss@pre \doteq null)) \vee$
 $(\tau \models (self.boss@pre \langle \rangle null) \wedge$
 $(\tau \models (self.boss@pre.salary@pre \leq_{int} self.salary@pre)) \wedge$
 $(\tau \models (inv_{Person\text{-}labelATpre}(self.boss@pre))))$

<proof>

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 ⇒ (ℳ)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)

```

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 http://dx.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>
	<code>_ * _</code>	$op *_{real}$	<i>UML-Real.OclMult_{Real}</i>

Continued on next page

	OCL	Featherweight OCL	Logical Constant
	-		
	- / -		
	- .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>
	5	5	<i>UML-Integer.OclInt5</i>

Continued on next page

	OCL	Featherweight OCL	Logical Constant
	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>			
	<code>a</code>	<code>a</code>	<i>UML-String.OclStringa</i>
	<code>b</code>	<code>b</code>	<i>UML-String.OclStringb</i>
	<code>c</code>	<code>c</code>	<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>	
	<code>_ = _</code>	<i>op ≐</i>	<i>UML-Logic.StrongEq</i>
Set and Iterators on Set	<code>Set (_)</code>	<i>Set(type⁰)</i>	<i>UML-Types.Set_{base} 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>	
<code>_ ->flatten()</code>			

Continued on next page

	OCL	Featherweight OCL	Logical Constant
	<code>_ ->selectByKind(_)</code>		
	<code>_ ->selectByType(_)</code>		
	<code>_ ->reject(_ _)</code>	<code>_ ->rejectSet(id _)</code>	<i>OclRejectSet</i>
	<code>_ ->select(_ _)</code>	<code>_ ->selectSet(id _)</code>	<i>OclSelectSet</i>
	<code>_ ->iterate(_ ; _ = _ _)</code>	<code>_ ->iterateSet(idt⁰ ; idt⁰ = any⁰ any⁰)</code>	<i>OclIterateSet</i>
	<code>_ ->exists(_ _)</code>	<code>_ ->existsSet(id _)</code>	<i>OclExistSet</i>
	<code>_ ->forall(_ _)</code>	<code>_ ->forallSet(id _)</code>	<i>OclForallSet</i>
	<code>_ ->asSequence()</code>	<code>_ ->asSequenceSet()</code>	<i>UML-Library.OclAsSeqSet</i>
	<code>_ ->asBag()</code>	<code>_ ->asBagSet()</code>	<i>UML-Library.OclAsBagSet</i>
	<code>_ ->asPair()</code>	<code>_ ->asPairSet()</code>	<i>UML-Library.OclAsPairSet</i>
	<code>_ ->sum()</code>	<code>_ ->sumSet()</code>	<i>UML-Set.OclSum</i>
	<code>_ ->excludesAll(_)</code>	<code>_ ->excludesAllSet(_)</code>	<i>UML-Set.OclExcludesAll</i>
	<code>_ ->includesAll(_)</code>	<code>_ ->includesAllSet(_)</code>	<i>UML-Set.OclIncludesAll</i>
	<code>_ ->any()</code>	<code>_ ->anySet()</code>	<i>UML-Set.OclANY</i>
	<code>_ ->notEmpty()</code>	<code>_ ->notEmptySet()</code>	<i>UML-Set.OclNotEmpty</i>
	<code>_ ->isEmpty()</code>	<code>_ ->isEmptySet()</code>	<i>UML-Set.OclIsEmpty</i>
	<code>_ ->size()</code>	<code>_ ->sizeSet()</code>	<i>UML-Set.OclSize</i>
	<code>_ ->excludes(_)</code>	<code>_ ->excludesSet(_)</code>	<i>UML-Set.OclExcludes</i>
	<code>_ ->includes(_)</code>	<code>_ ->includesSet(_)</code>	<i>UML-Set.OclIncludes</i>
Sequence and Iterators on Sequence	<code>Sequence(_)</code>	<code>Sequence(type⁰)</code>	<i>UML-Types.Sequencebase 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>_ ->anySeq()</code>	<i>UML-Sequence.OclANY</i>
	<code>_ ->notEmpty()</code>	<code>_ ->notEmptySeq()</code>	<i>UML-Sequence.OclNotEmpty</i>
	<code>_ ->isEmpty()</code>	<code>_ ->isEmptySeq()</code>	<i>UML-Sequence.OclIsEmpty</i>
	<code>_ ->size()</code>	<code>_ ->sizeSeq()</code>	<i>UML-Sequence.OclSize</i>
	<code>_ ->select(_ _)</code>	<code>_ ->selectSeq(id _)</code>	<i>OclSelectSeq</i>
	<code>_ ->collect(_ _)</code>	<code>_ ->collectSeq(id _)</code>	<i>OclCollectSeq</i>
	<code>_ ->exists(_ _)</code>	<code>_ ->existsSeq(id _)</code>	<i>OclExistSeq</i>
	<code>_ ->forall(_ _)</code>	<code>_ ->forallSeq(id _)</code>	<i>OclForallSeq</i>
	<code>_ ->iterate(_ ; _ : _ = _ _)</code>	<code>_ ->iterateSeq(idt⁰ ; idt⁰ = any⁰ any⁰)</code>	<i>OclIterateSeq</i>
	<code>_ ->last()</code>	<code>_ ->lastSeq(_)</code>	<i>UML-Sequence.OclLast</i>
	<code>_ ->first()</code>	<code>_ ->firstSeq(_)</code>	<i>UML-Sequence.OclFirst</i>
	<code>_ ->at(_)</code>	<code>_ ->atSeq(_)</code>	<i>UML-Sequence.OclAt</i>
	<code>_ ->union(_)</code>	<code>_ ->unionSeq(_)</code>	<i>UML-Sequence.OclUnion</i>
	<code>_ ->append(_)</code>	<code>_ ->appendSeq(_)</code>	<i>UML-Sequence.OclAppend</i>
	<code>_ ->excluding(_)</code>	<code>_ ->excludingSeq(_)</code>	<i>UML-Sequence.OclExcluding</i>
	<code>_ ->including(_)</code>	<code>_ ->includingSeq(_)</code>	<i>UML-Sequence.OclIncluding</i>
	<code>_ ->prepend(_)</code>	<code>_ ->prependSeq(_)</code>	<i>UML-Sequence.OclPrepend</i>
<code>_ ->asSet()</code>	<code>_ ->asSetSeq()</code>	<i>UML-Library.OclAsSetSeq</i>	
<code>_ ->asBag()</code>	<code>_ ->asBagSeq()</code>	<i>UML-Library.OclAsBagSeq</i>	
<code>_ ->asPair()</code>	<code>_ ->asPairSeq()</code>	<i>UML-Library.OclAsPairSeq</i>	
Bags and Iterators on Bag	<code>Bag(_)</code>	<code>Bag(type⁰)</code>	<i>UML-Types.Bagbase 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>_ ->sumBag()</code>	<i>UML-Bag.OclSum</i>
	<code>_ ->count(_)</code>	<code>_ ->countBag(_)</code>	<i>UML-Bag.OclCount</i>
	<code>_ ->intersection(_)</code>	<code>_ ->intersectionBag(_)</code>	<i>UML-Bag.OclIntersection</i>
	<code>_ ->union(_)</code>	<code>_ ->unionBag(_)</code>	<i>UML-Bag.OclUnion</i>

Continued on next page

OCL	Featherweight OCL	Logical Constant	
<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}(id __)</code>	<i>OclRejectBag</i>	
<code>_ ->select(_ _)</code>	<code>__ ->select_{Bag}(id __)</code>	<i>OclSelectBag</i>	
<code>_ ->iterate(_ ; _ = _ _)</code>	<code>__ ->iterate_{Bag}(idt⁰ ; idt⁰ = <i>any</i>⁰ <i>any</i>⁰)</code>	<i>OclIterateBag</i>	
<code>_ ->exists(_ _)</code>	<code>__ ->exists_{Bag}(id __)</code>	<i>OclExistBag</i>	
<code>_ ->forall(_ _)</code>	<code>__ ->forall_{Bag}(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(<i>type</i>⁰ , <i>type</i>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>	<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>	