

Safe OCL

Denis Nikiforov

September 13, 2023

### Abstract

The theory is a formalization of the OCL type system, its abstract syntax and expression typing rules [1]. The theory does not define a concrete syntax and a semantics. In contrast to Featherweight OCL [2], it is based on a deep embedding approach. The type system is defined from scratch, it is not based on the Isabelle HOL type system.

The Safe OCL distinguishes nullable and non-nullable types. Also the theory gives a formal definition of safe navigation operations [3]. The Safe OCL typing rules are much stricter than rules given in the OCL specification. It allows one to catch more errors on a type checking phase.

The type theory presented is four-layered: classes, basic types, generic types, errorable types. We introduce the following new types: non-nullable types ( $\tau[I]$ ), nullable types ( $\tau[?]$ ), *OclSuper*. *OclSuper* is a supertype of all other types (basic types, collections, tuples). This type allows us to define a total supremum function, so types form an upper semilattice. It allows us to define rich expression typing rules in an elegant manner.

The Preliminaries Section of the theory defines a number of helper lemmas for transitive closures and tuples. It defines also a generic object model independent from OCL. It allows one to use the theory as a reference for formalization of analogous languages.

# Contents

<b>1</b>	<b>Preliminaries</b>	<b>7</b>
1.1	Errorable	7
1.2	Transitive Closures	7
1.2.1	Basic Properties	8
1.2.2	Helper Lemmas	8
1.2.3	Transitive Closure Preservation	8
1.2.4	Transitive Closure Reflection	9
1.3	Finite Maps	9
1.3.1	Helper Lemmas	10
1.3.2	Merge Operation	10
1.3.3	Acyclicity	11
1.3.4	Transitive Closures	11
1.3.5	Size Calculation	13
1.3.6	Code Setup	14
1.4	Tuples	14
1.4.1	Definitions	14
1.4.2	Helper Lemmas	14
1.4.3	Basic Properties	15
1.4.4	Transitive Closures	16
1.4.5	Code Setup	17
1.5	Object Model	18
1.5.1	Type Synonyms	18
1.5.2	Attributes	19
1.5.3	Association Ends	19
1.5.4	Association Classes	21
1.5.5	Association Class Ends	21
1.5.6	Operations	22
1.5.7	Literals	23
1.5.8	Definition	23
1.5.9	Properties	24
1.5.10	Code Setup	25

<b>2</b>	<b>Basic Types</b>	<b>29</b>
2.1	Definition . . . . .	29
2.2	Partial Order of Basic Types . . . . .	30
2.2.1	Strict Introduction Rules . . . . .	30
2.2.2	Strict Elimination Rules . . . . .	31
2.2.3	Properties . . . . .	32
2.2.4	Non-Strict Introduction Rules . . . . .	33
2.2.5	Non-Strict Elimination Rules . . . . .	33
2.2.6	Simplification Rules . . . . .	34
2.3	Upper Semilattice of Basic Types . . . . .	35
2.4	Code Setup . . . . .	36
<b>3</b>	<b>Types</b>	<b>37</b>
3.1	Definition . . . . .	37
3.2	Constructors Bijectivity on Transitive Closures . . . . .	39
3.3	Partial Order of Types . . . . .	40
3.3.1	Strict Introduction Rules . . . . .	40
3.3.2	Strict Elimination Rules . . . . .	41
3.3.3	Properties . . . . .	42
3.3.4	Non-Strict Introduction Rules . . . . .	43
3.3.5	Non-Strict Elimination Rules . . . . .	44
3.3.6	Simplification Rules . . . . .	45
3.4	Upper Semilattice of Types . . . . .	46
3.5	Helper Relations . . . . .	47
3.6	Determinism . . . . .	49
3.7	Code Setup . . . . .	49
<b>4</b>	<b>Abstract Syntax</b>	<b>53</b>
4.1	Preliminaries . . . . .	53
4.2	Standard Library Operations . . . . .	53
4.3	Expressions . . . . .	55
<b>5</b>	<b>Object Model</b>	<b>59</b>
<b>6</b>	<b>Typing</b>	<b>61</b>
6.1	Operations Typing . . . . .	61
6.1.1	Metaclass Operations . . . . .	61
6.1.2	Type Operations . . . . .	61
6.1.3	OclSuper Operations . . . . .	62
6.1.4	OclAny Operations . . . . .	62
6.1.5	Boolean Operations . . . . .	63
6.1.6	Numeric Operations . . . . .	63
6.1.7	String Operations . . . . .	65
6.1.8	Collection Operations . . . . .	65

6.1.9	Coercions	68
6.1.10	Simplification Rules	69
6.1.11	Determinism	69
6.2	Expressions Typing	71
6.3	Elimination Rules	76
6.4	Simplification Rules	77
6.5	Determinism	78
6.6	Code Setup	79
<b>7</b>	<b>Normalization</b>	<b>81</b>
7.1	Normalization Rules	81
7.2	Elimination Rules	85
7.3	Simplification Rules	86
7.4	Determinism	86
7.5	Normalized Expressions Typing	87
7.6	Code Setup	87
<b>8</b>	<b>Examples</b>	<b>89</b>
8.1	Classes	89
8.2	Object Model	91
8.3	Simplification Rules	93
8.4	Basic Types	94
8.4.1	Positive Cases	94
8.4.2	Negative Cases	94
8.5	Types	95
8.5.1	Positive Cases	95
8.5.2	Negative Cases	95
8.6	Typing	95
8.6.1	Positive Cases	95
8.6.2	Negative Cases	98
8.7	Code	98
8.7.1	Positive Cases	98
8.7.2	Negative Cases	99



# Chapter 1

## Preliminaries

### 1.1 Errorable

```
theory Errorable
  imports Main
begin

notation bot (  $\perp$  )

typedef 'a errorable (  $\perp$  [21] 21) = UNIV :: 'a option set <proof>

definition errorable :: 'a  $\Rightarrow$  'a errorable (  $\perp$  [1000] 1000) where
  errorable x = Abs-errorable (Some x)

instantiation errorable :: (type) bot
begin
definition  $\perp \equiv$  Abs-errorable None
instance <proof>
end

free-constructors case-errorable for
  errorable
|  $\perp$  :: 'a errorable
  <proof>

copy-bnf 'a errorable

end
```

### 1.2 Transitive Closures

```
theory Transitive-Closure-Ext
  imports HOL-Library.FuncSet
begin
```

### 1.2.1 Basic Properties

$R^{++}$  is a transitive closure of a relation  $R$ .  $R^{**}$  is a reflexive transitive closure of a relation  $R$ .

A function  $f$  is surjective on  $R^{++}$  iff for any two elements in the range of  $f$ , related through  $R^{++}$ , all their intermediate elements belong to the range of  $f$ .

**abbreviation** *surj-on-trancl*  $R f \equiv$   
 $(\forall x y z. R^{++} (f x) y \longrightarrow R y (f z) \longrightarrow y \in \text{range } f)$

A function  $f$  is bijective on  $R^{++}$  iff it is injective and surjective on  $R^{++}$ .

**abbreviation** *bij-on-trancl*  $R f \equiv \text{inj } f \wedge \text{surj-on-trancl } R f$

### 1.2.2 Helper Lemmas

**lemma** *rtranclp-eq-rtranclp* [iff]:  
 $(\lambda x y. P x y \vee x = y)^{**} = P^{**}$   
 ⟨proof⟩

**lemma** *tranclp-eq-rtranclp* [iff]:  
 $(\lambda x y. P x y \vee x = y)^{++} = P^{**}$   
 ⟨proof⟩

**lemma** *rtranclp-eq-rtranclp'* [iff]:  
 $(\lambda x y. P x y \wedge x \neq y)^{**} = P^{**}$   
 ⟨proof⟩

**lemma** *tranclp-tranclp-to-tranclp-r*:  
**assumes**  $(\bigwedge x y z. R^{++} x y \Longrightarrow R y z \Longrightarrow P x \Longrightarrow P z \Longrightarrow P y)$   
**assumes**  $R^{++} x y$  **and**  $R^{++} y z$   
**assumes**  $P x$  **and**  $P z$   
**shows**  $P y$   
 ⟨proof⟩

### 1.2.3 Transitive Closure Preservation

A function  $f$  preserves  $R^{++}$  if it preserves  $R$ .

The proof was derived from the accepted answer on the website Stack Overflow that is available at <https://stackoverflow.com/a/52573551/632199> and provided with the permission of the author of the answer.

**lemma** *preserve-tranclp*:  
**assumes**  $\bigwedge x y. R x y \Longrightarrow S (f x) (f y)$   
**assumes**  $R^{++} x y$   
**shows**  $S^{++} (f x) (f y)$   
 ⟨proof⟩

A function  $f$  preserves  $R^{**}$  if it preserves  $R$ .



**lemma** *preserve-rtranclp*:

$$\begin{aligned} (\bigwedge x y. R x y \implies S (f x) (f y)) &\implies \\ R^{**} x y \implies S^{**} (f x) (f y) & \\ \langle proof \rangle & \end{aligned}$$

If one needs to prove that  $(f x)$  and  $(g y)$  are related through  $S^{**}$  then one can use the previous lemma and add a one more step from  $(f y)$  to  $(g y)$ .

**lemma** *preserve-rtranclp'*:

$$\begin{aligned} (\bigwedge x y. R x y \implies S (f x) (f y)) &\implies \\ (\bigwedge y. S (f y) (g y)) \implies & \\ R^{**} x y \implies S^{**} (f x) (g y) & \\ \langle proof \rangle & \end{aligned}$$

**lemma** *preserve-rtranclp''*:

$$\begin{aligned} (\bigwedge x y. R x y \implies S (f x) (f y)) &\implies \\ (\bigwedge y. S (f y) (g y)) \implies & \\ R^{**} x y \implies S^{++} (f x) (g y) & \\ \langle proof \rangle & \end{aligned}$$

### 1.2.4 Transitive Closure Reflection

A function  $f$  reflects  $S^{++}$  if it reflects  $S$  and is bijective on  $S^{++}$ .

The proof was derived from the accepted answer on the website Stack Overflow that is available at <https://stackoverflow.com/a/52573551/632199> and provided with the permission of the author of the answer.

**lemma** *reflect-tranclp*:

**assumes** *refl-f*:  $\bigwedge x y. S (f x) (f y) \implies R x y$   
**assumes** *bij-f*: *bij-on-trancl*  $S f$   
**assumes** *prem*:  $S^{++} (f x) (f y)$   
**shows**  $R^{++} x y$   
 $\langle proof \rangle$

A function  $f$  reflects  $S^{**}$  if it reflects  $S$  and is bijective on  $S^{++}$ .

**lemma** *reflect-rtranclp*:

$$\begin{aligned} (\bigwedge x y. S (f x) (f y) \implies R x y) &\implies \\ \text{bij-on-trancl } S f &\implies \\ S^{**} (f x) (f y) \implies R^{**} x y & \\ \langle proof \rangle & \end{aligned}$$

**end**

## 1.3 Finite Maps

**theory** *Finite-Map-Ext*

**imports** *HOL-Library.Finite-Map*

**begin**

**type-notation**  $fmap$  (  $(- \rightarrow_f /-)$  [22, 21] 21)

**nonterminal**  $fmaplets$  and  $fmaplet$

**syntax**

$$\begin{aligned} -fmaplet &:: ['a, 'a] \Rightarrow fmaplet && (- / \mapsto_f / -) \\ -fmaplets &:: ['a, 'a] \Rightarrow fmaplet && (- / [\mapsto_f] / -) \\ &:: fmaplet \Rightarrow fmaplets && (-) \\ -FMaplets &:: [fmaplet, fmaplets] \Rightarrow fmaplets && (-, / -) \\ -FMapUpd &:: ['a \rightarrow 'b, fmaplets] \Rightarrow 'a \rightarrow 'b && (- / '(-) [900, 0] 900) \\ -FMap &:: fmaplets \Rightarrow 'a \rightarrow 'b && (1[-]) \end{aligned}$$

**syntax (ASCII)**

$$\begin{aligned} -fmaplet &:: ['a, 'a] \Rightarrow fmaplet && (- / | \rightarrow f / -) \\ -fmaplets &:: ['a, 'a] \Rightarrow fmaplet && (- / [| \rightarrow f] / -) \end{aligned}$$

**translations**

$$\begin{aligned} -FMapUpd\ m\ (-FMaplets\ xy\ ms) &\quad \Rightarrow -FMapUpd\ (-FMapUpd\ m\ xy)\ ms \\ -FMapUpd\ m\ (-fmaplet\ x\ y) &\quad \Rightarrow CONST\ fmapupd\ x\ y\ m \\ -FMap\ ms &\quad \Rightarrow -FMapUpd\ (CONST\ fmempty)\ ms \\ -FMap\ (-FMaplets\ ms1\ ms2) &\quad \Leftarrow -FMapUpd\ (-FMap\ ms1)\ ms2 \\ -FMaplets\ ms1\ (-FMaplets\ ms2\ ms3) &\quad \Leftarrow -FMaplets\ (-FMaplets\ ms1\ ms2)\ ms3 \end{aligned}$$

### 1.3.1 Helper Lemmas

**lemma**  $fmrel\text{-on-fset-fmdom}$ :

$$\begin{aligned} fmrel\text{-on-fset}\ (fmdom\ ym)\ f\ xm\ ym &\implies \\ k\ |\in|\ fmdom\ ym &\implies \\ k\ |\in|\ fmdom\ xm & \\ \langle proof \rangle & \end{aligned}$$

### 1.3.2 Merge Operation

**definition**  $fmmerge\ f\ xm\ ym \equiv$

$$\begin{aligned} &fmap\text{-of-list}\ (map \\ &(\lambda k.\ (k,\ f\ (the\ (fmlookup\ xm\ k))\ (the\ (fmlookup\ ym\ k)))) \\ &(sorted\text{-list-of-fset}\ (fmdom\ xm\ |\cap|\ fmdom\ ym))) \end{aligned}$$

**lemma**  $fmdom\text{-fmmerge}$  [simp]:

$$\begin{aligned} fmdom\ (fmmerge\ g\ xm\ ym) &= fmdom\ xm\ |\cap|\ fmdom\ ym \\ \langle proof \rangle & \end{aligned}$$

**lemma**  $fmmerge\text{-commut}$ :

$$\begin{aligned} \text{assumes } \bigwedge x\ y.\ x \in fmran'\ xm &\implies f\ x\ y = f\ y\ x \\ \text{shows } fmmerge\ f\ xm\ ym &= fmmerge\ f\ ym\ xm \\ \langle proof \rangle & \end{aligned}$$

**lemma**  $fmrel\text{-on-fset-fmmerge1}$  [intro]:

$$\text{assumes } \bigwedge x\ y\ z.\ z \in fmran'\ zm \implies f\ x\ z \implies f\ y\ z \implies f\ (g\ x\ y)\ z$$

**assumes**  $fmrel\text{-}on\text{-}fset (fmdom\ zm) f\ xm\ zm$   
**assumes**  $fmrel\text{-}on\text{-}fset (fmdom\ zm) f\ ym\ zm$   
**shows**  $fmrel\text{-}on\text{-}fset (fmdom\ zm) f (fmmerge\ g\ xm\ ym)\ zm$   
 <proof>

**lemma**  $fmrel\text{-}on\text{-}fset\text{-}fmmerge2$  [intro]:  
**assumes**  $\bigwedge x\ y. x \in fmran'\ xm \implies f\ x\ (g\ x\ y)$   
**shows**  $fmrel\text{-}on\text{-}fset (fmdom\ ym) f\ xm\ (fmmerge\ g\ xm\ ym)$   
 <proof>

### 1.3.3 Acyclicity

**abbreviation**  $acyclic\text{-}on\ xs\ r \equiv (\forall x. x \in xs \longrightarrow (x, x) \notin r^+)$

**abbreviation**  $acyclicP\text{-}on\ xs\ r \equiv acyclic\text{-}on\ xs\ \{(x, y). r\ x\ y\}$

**lemma**  $fmrel\text{-}acyclic$ :  
 $acyclicP\text{-}on (fmran'\ xm)\ R \implies$   
 $fmrel\ R^{++}\ xm\ ym \implies$   
 $fmrel\ R\ ym\ xm \implies$   
 $xm = ym$   
 <proof>

**lemma**  $fmrel\text{-}acyclic'$ :  
**assumes**  $acyclicP\text{-}on (fmran'\ ym)\ R$   
**assumes**  $fmrel\ R^{++}\ xm\ ym$   
**assumes**  $fmrel\ R\ ym\ xm$   
**shows**  $xm = ym$   
 <proof>

**lemma**  $fmrel\text{-}on\text{-}fset\text{-}acyclic$ :  
 $acyclicP\text{-}on (fmran'\ xm)\ R \implies$   
 $fmrel\text{-}on\text{-}fset (fmdom\ ym)\ R^{++}\ xm\ ym \implies$   
 $fmrel\text{-}on\text{-}fset (fmdom\ xm)\ R\ ym\ xm \implies$   
 $xm = ym$   
 <proof>

**lemma**  $fmrel\text{-}on\text{-}fset\text{-}acyclic'$ :  
 $acyclicP\text{-}on (fmran'\ ym)\ R \implies$   
 $fmrel\text{-}on\text{-}fset (fmdom\ ym)\ R^{++}\ xm\ ym \implies$   
 $fmrel\text{-}on\text{-}fset (fmdom\ xm)\ R\ ym\ xm \implies$   
 $xm = ym$   
 <proof>

### 1.3.4 Transitive Closures

**lemma**  $fmrel\text{-}trans$ :  
 $(\bigwedge x\ y\ z. x \in fmran'\ xm \implies P\ x\ y \implies Q\ y\ z \implies R\ x\ z) \implies$   
 $fmrel\ P\ xm\ ym \implies fmrel\ Q\ ym\ zm \implies fmrel\ R\ xm\ zm$   
 <proof>

**lemma** *fmrel-on-fset-trans*:

$$(\wedge x y z. x \in \text{fmran}' xm \implies P x y \implies Q y z \implies R x z) \implies \\ \text{fmrel-on-fset } (\text{fmdom } ym) P xm ym \implies \\ \text{fmrel-on-fset } (\text{fmdom } zm) Q ym zm \implies \\ \text{fmrel-on-fset } (\text{fmdom } zm) R xm zm \\ \langle \text{proof} \rangle$$

**lemma** *trancl-to-fmrel*:

$$(\text{fmrel } f)^{++} xm ym \implies \text{fmrel } f^{++} xm ym \\ \langle \text{proof} \rangle$$

**lemma** *fmrel-trancl-fmdom-eq*:

$$(\text{fmrel } f)^{++} xm ym \implies \text{fmdom } xm = \text{fmdom } ym \\ \langle \text{proof} \rangle$$

The proof was derived from the accepted answer on the website Stack Overflow that is available at <https://stackoverflow.com/a/53585232/632199> and provided with the permission of the author of the answer.

**lemma** *fmupd-fmdrop*:

$$\text{fmlookup } xm k = \text{Some } x \implies \\ xm = \text{fmupd } k x (\text{fmdrop } k xm) \\ \langle \text{proof} \rangle$$

**lemma** *fmap-eqdom-Cons1*:

**assumes** *fmlookup*  $xm\ i = \text{None}$   
**and** *fmdom*  $(\text{fmupd } i\ x\ xm) = \text{fmdom } ym$   
**and** *fmrel*  $R (\text{fmupd } i\ x\ xm)\ ym$   
**shows**  $(\exists z\ zm. \text{fmlookup } zm\ i = \text{None} \wedge ym = (\text{fmupd } i\ z\ zm) \wedge \\ R\ x\ z \wedge \text{fmrel } R\ xm\ zm)$

$\langle \text{proof} \rangle$

The proof was derived from the accepted answer on the website Stack Overflow that is available at <https://stackoverflow.com/a/53585232/632199> and provided with the permission of the author of the answer.

**lemma** *fmap-eqdom-induct* [*consumes 2, case-names nil step*]:

**assumes** *R*: *fmrel*  $R\ xm\ ym$   
**and** *dom-eq*: *fmdom*  $xm = \text{fmdom } ym$   
**and** *nil*:  $P (\text{fmap-of-list } []) (\text{fmap-of-list } [])$   
**and** *step*:  
 $\wedge x\ xm\ y\ ym\ i. \\ \llbracket R\ x\ y; \text{fmrel } R\ xm\ ym; \text{fmdom } xm = \text{fmdom } ym; P\ xm\ ym \rrbracket \implies \\ P (\text{fmupd } i\ x\ xm) (\text{fmupd } i\ y\ ym)$

**shows**  $P\ xm\ ym$

$\langle \text{proof} \rangle$

The proof was derived from the accepted answer on the website Stack Overflow that is available at <https://stackoverflow.com/a/53585232/632199> and provided with the permission of the author of the answer.

**lemma** *fmrel-to-rtrancl*:  
**assumes** *as-r*: *reflp r*  
**and** *rel-rpp-xm-ym*: *fmrel r\*\* xm ym*  
**shows**  $(fmrel\ r)^{**}\ xm\ ym$   
*<proof>*

The proof was derived from the accepted answer on the website Stack Overflow that is available at <https://stackoverflow.com/a/53585232/632199> and provided with the permission of the author of the answer.

**lemma** *fmrel-to-trancl*:  
**assumes** *reflp r*  
**and** *fmrel r++ xm ym*  
**shows**  $(fmrel\ r)^{++}\ xm\ ym$   
*<proof>*

**lemma** *fmrel-tranclp-induct*:  
 $fmrel\ r^{++}\ a\ b \implies$   
 $reflp\ r \implies$   
 $(\bigwedge y. fmrel\ r\ a\ y \implies P\ y) \implies$   
 $(\bigwedge y\ z. (fmrel\ r)^{++}\ a\ y \implies fmrel\ r\ y\ z \implies P\ y \implies P\ z) \implies P\ b$   
*<proof>*

**lemma** *fmrel-converse-tranclp-induct*:  
 $fmrel\ r^{++}\ a\ b \implies$   
 $reflp\ r \implies$   
 $(\bigwedge y. fmrel\ r\ y\ b \implies P\ y) \implies$   
 $(\bigwedge y\ z. fmrel\ r\ y\ z \implies fmrel\ r^{++}\ z\ b \implies P\ z \implies P\ y) \implies P\ a$   
*<proof>*

**lemma** *fmrel-tranclp-trans-induct*:  
 $fmrel\ r^{++}\ a\ b \implies$   
 $reflp\ r \implies$   
 $(\bigwedge x\ y. fmrel\ r\ x\ y \implies P\ x\ y) \implies$   
 $(\bigwedge x\ y\ z. fmrel\ r^{++}\ x\ y \implies P\ x\ y \implies fmrel\ r^{++}\ y\ z \implies P\ y\ z \implies P\ x\ z) \implies$   
 $P\ a\ b$   
*<proof>*

### 1.3.5 Size Calculation

The contents of the subsection was derived from the accepted answer on the website Stack Overflow that is available at <https://stackoverflow.com/a/53244203/632199> and provided with the permission of the author of the answer.

**abbreviation** *tcf*  $\equiv (\lambda v::('a \times nat). (\lambda r::nat. snd\ v + r))$

**interpretation** *tcf*: *comp-fun-commute tcf*  
*<proof>*

**lemma** *ffold-rec-exp*:  
**assumes**  $k \in | \text{fmdom } x$   
**and**  $ky = (k, \text{the } (\text{fmlookup } (\text{fmmap } f \ x) \ k))$   
**shows**  $\text{ffold } \text{tcf } 0 \ (\text{fset-of-fmap } (\text{fmmap } f \ x)) =$   
 $\text{tcf } ky \ (\text{ffold } \text{tcf } 0 \ ((\text{fset-of-fmap } (\text{fmmap } f \ x)) \ |- \ \{|ky\}))$   
*<proof>*

**lemma** *elem-le-ffold* [*intro*]:  
 $k \in | \text{fmdom } x \implies$   
 $f \ (\text{the } (\text{fmlookup } x \ k)) < \text{Suc } (\text{ffold } \text{tcf } 0 \ (\text{fset-of-fmap } (\text{fmmap } f \ x)))$   
*<proof>*

**lemma** *elem-le-ffold'* [*intro*]:  
 $z \in \text{fmran}' \ x \implies$   
 $f \ z < \text{Suc } (\text{ffold } \text{tcf } 0 \ (\text{fset-of-fmap } (\text{fmmap } f \ x)))$   
*<proof>*

### 1.3.6 Code Setup

**abbreviation** *fmmerge-fun*  $f \ xm \ ym \equiv$   
 $\text{fmap-of-list } (\text{map}$   
 $(\lambda k. \text{if } k \in | \text{fmdom } xm \wedge k \in | \text{fmdom } ym$   
 $\text{then } (k, f \ (\text{the } (\text{fmlookup } xm \ k)) \ (\text{the } (\text{fmlookup } ym \ k)))$   
 $\text{else } (k, \text{undefined}))$   
 $(\text{sorted-list-of-fset } (\text{fmdom } xm \ |\cap| \ \text{fmdom } ym)))$

**lemma** *fmmerge-fun-simp* [*code-abbrev*, *simp*]:  
 $\text{fmmerge-fun } f \ xm \ ym = \text{fmmerge } f \ xm \ ym$   
*<proof>*

**end**

## 1.4 Tuples

**theory** *Tuple*  
**imports** *Finite-Map-Ext Transitive-Closure-Ext*  
**begin**

### 1.4.1 Definitions

**abbreviation** *subtuple*  $f \ xm \ ym \equiv \text{fmrel-on-fset } (\text{fmdom } ym) \ f \ xm \ ym$

**abbreviation** *strict-subtuple*  $f \ xm \ ym \equiv \text{subtuple } f \ xm \ ym \wedge xm \neq ym$

### 1.4.2 Helper Lemmas

**lemma** *fmrel-to-subtuple*:  
 $\text{fmrel } r \ xm \ ym \implies \text{subtuple } r \ xm \ ym$   
*<proof>*

**lemma** *subtuple-eq-fmrel-fmrestrict-fset*:

$subtuple\ r\ xm\ ym = fmrel\ r\ (fmrestrict-fset\ (fmdom\ ym)\ xm)\ ym$   
 ⟨proof⟩

**lemma** *subtuple-fmdom*:

$subtuple\ f\ xm\ ym \implies$   
 $subtuple\ g\ ym\ xm \implies$   
 $fmdom\ xm = fmdom\ ym$   
 ⟨proof⟩

### 1.4.3 Basic Properties

**lemma** *subtuple-refl*:

$reflp\ R \implies subtuple\ R\ xm\ xm$   
 ⟨proof⟩

**lemma** *subtuple-mono* [mono]:

$(\bigwedge x\ y. x \in fmran'\ xm \implies y \in fmran'\ ym \implies f\ x\ y \longrightarrow g\ x\ y) \implies$   
 $subtuple\ f\ xm\ ym \longrightarrow subtuple\ g\ xm\ ym$   
 ⟨proof⟩

**lemma** *strict-subtuple-mono* [mono]:

$(\bigwedge x\ y. x \in fmran'\ xm \implies y \in fmran'\ ym \implies f\ x\ y \longrightarrow g\ x\ y) \implies$   
 $strict-subtuple\ f\ xm\ ym \longrightarrow strict-subtuple\ g\ xm\ ym$   
 ⟨proof⟩

**lemma** *subtuple-antisym*:

**assumes**  $subtuple\ (\lambda x\ y. f\ x\ y \vee x = y)\ xm\ ym$   
**assumes**  $subtuple\ (\lambda x\ y. f\ x\ y \wedge \neg f\ y\ x \vee x = y)\ ym\ xm$   
**shows**  $xm = ym$   
 ⟨proof⟩

**lemma** *strict-subtuple-antisym*:

$strict-subtuple\ (\lambda x\ y. f\ x\ y \vee x = y)\ xm\ ym \implies$   
 $strict-subtuple\ (\lambda x\ y. f\ x\ y \wedge \neg f\ y\ x \vee x = y)\ ym\ xm \implies False$   
 ⟨proof⟩

**lemma** *subtuple-acyclic*:

**assumes**  $acyclicP-on\ (fmran'\ xm)\ P$   
**assumes**  $subtuple\ (\lambda x\ y. P\ x\ y \vee x = y)^{++}\ xm\ ym$   
**assumes**  $subtuple\ (\lambda x\ y. P\ x\ y \vee x = y)\ ym\ xm$   
**shows**  $xm = ym$   
 ⟨proof⟩

**lemma** *subtuple-acyclic'*:

**assumes**  $acyclicP-on\ (fmran'\ ym)\ P$   
**assumes**  $subtuple\ (\lambda x\ y. P\ x\ y \vee x = y)^{++}\ xm\ ym$   
**assumes**  $subtuple\ (\lambda x\ y. P\ x\ y \vee x = y)\ ym\ xm$

**shows**  $xm = ym$   
 ⟨proof⟩

**lemma** *subtuple-acyclic''*:  
 $acyclicP\text{-on } (f\text{mran}' ym) R \implies$   
 $subtuple R^{**} xm ym \implies$   
 $subtuple R ym xm \implies$   
 $xm = ym$   
 ⟨proof⟩

**lemma** *strict-subtuple-trans*:  
 $acyclicP\text{-on } (f\text{mran}' xm) P \implies$   
 $strict\text{-subtuple } (\lambda x y. P x y \vee x = y)^{++} xm ym \implies$   
 $strict\text{-subtuple } (\lambda x y. P x y \vee x = y) ym zm \implies$   
 $strict\text{-subtuple } (\lambda x y. P x y \vee x = y)^{++} xm zm$   
 ⟨proof⟩

**lemma** *strict-subtuple-trans'*:  
 $acyclicP\text{-on } (f\text{mran}' zm) P \implies$   
 $strict\text{-subtuple } (\lambda x y. P x y \vee x = y) xm ym \implies$   
 $strict\text{-subtuple } (\lambda x y. P x y \vee x = y)^{++} ym zm \implies$   
 $strict\text{-subtuple } (\lambda x y. P x y \vee x = y)^{++} xm zm$   
 ⟨proof⟩

**lemma** *strict-subtuple-trans''*:  
 $acyclicP\text{-on } (f\text{mran}' zm) R \implies$   
 $strict\text{-subtuple } R xm ym \implies$   
 $strict\text{-subtuple } R^{**} ym zm \implies$   
 $strict\text{-subtuple } R^{**} xm zm$   
 ⟨proof⟩

**lemma** *strict-subtuple-trans'''*:  
 $acyclicP\text{-on } (f\text{mran}' zm) P \implies$   
 $strict\text{-subtuple } (\lambda x y. P x y \vee x = y) xm ym \implies$   
 $strict\text{-subtuple } (\lambda x y. P x y \vee x = y)^{**} ym zm \implies$   
 $strict\text{-subtuple } (\lambda x y. P x y \vee x = y)^{**} xm zm$   
 ⟨proof⟩

**lemma** *subtuple-fmmerge2 [intro]*:  
 $(\bigwedge x y. x \in f\text{mran}' xm \implies f x (g x y)) \implies$   
 $subtuple f xm (f\text{mmerge } g xm ym)$   
 ⟨proof⟩

#### 1.4.4 Transitive Closures

**lemma** *trancl-to-subtuple*:  
 $(subtuple r)^{++} xm ym \implies$   
 $subtuple r^{++} xm ym$   
 ⟨proof⟩



**lemma** *rtrancl-to-subtuple*:  
 $(\text{subtuple } r)^{**} \text{ } xm \text{ } ym \implies$   
 $\text{subtuple } r^{**} \text{ } xm \text{ } ym$   
 ⟨proof⟩

**lemma** *fmrel-to-subtuple-trancl*:  
 $\text{reflp } r \implies$   
 $(\text{fmrel } r)^{++} (\text{fmrestrict-fset } (\text{fmdom } ym) \text{ } xm) \text{ } ym \implies$   
 $(\text{subtuple } r)^{++} \text{ } xm \text{ } ym$   
 ⟨proof⟩

**lemma** *subtuple-to-trancl*:  
 $\text{reflp } r \implies$   
 $\text{subtuple } r^{++} \text{ } xm \text{ } ym \implies$   
 $(\text{subtuple } r)^{++} \text{ } xm \text{ } ym$   
 ⟨proof⟩

**lemma** *trancl-to-strict-subtuple*:  
 $\text{acyclicP-on } (\text{fmrans' } ym) \text{ } R \implies$   
 $(\text{strict-subtuple } R)^{++} \text{ } xm \text{ } ym \implies$   
 $\text{strict-subtuple } R^{**} \text{ } xm \text{ } ym$   
 ⟨proof⟩

**lemma** *trancl-to-strict-subtuple'*:  
 $\text{acyclicP-on } (\text{fmrans' } ym) \text{ } R \implies$   
 $(\text{strict-subtuple } (\lambda x y. R \text{ } x \text{ } y \vee x = y))^{++} \text{ } xm \text{ } ym \implies$   
 $\text{strict-subtuple } (\lambda x y. R \text{ } x \text{ } y \vee x = y)^{**} \text{ } xm \text{ } ym$   
 ⟨proof⟩

**lemma** *subtuple-rtranclp-intro*:  
**assumes**  $\bigwedge xm \text{ } ym. R \text{ } (f \text{ } xm) \text{ } (f \text{ } ym) \implies \text{subtuple } R \text{ } xm \text{ } ym$   
**and**  $\text{bij-on-trancl } R \text{ } f$   
**and**  $R^{**} \text{ } (f \text{ } xm) \text{ } (f \text{ } ym)$   
**shows**  $\text{subtuple } R^{**} \text{ } xm \text{ } ym$   
 ⟨proof⟩

**lemma** *strict-subtuple-rtranclp-intro*:  
**assumes**  $\bigwedge xm \text{ } ym. R \text{ } (f \text{ } xm) \text{ } (f \text{ } ym) \implies$   
 $\text{strict-subtuple } (\lambda x y. R \text{ } x \text{ } y \vee x = y) \text{ } xm \text{ } ym$   
**and**  $\text{bij-on-trancl } R \text{ } f$   
**and**  $\text{acyclicP-on } (\text{fmrans' } ym) \text{ } R$   
**and**  $R^{++} \text{ } (f \text{ } xm) \text{ } (f \text{ } ym)$   
**shows**  $\text{strict-subtuple } R^{**} \text{ } xm \text{ } ym$   
 ⟨proof⟩

### 1.4.5 Code Setup

**abbreviation**  $\text{subtuple-fun } f \text{ } xm \text{ } ym \equiv$

$fBall$  ( $fmdom\ ym$ ) ( $\lambda x. rel-option\ f\ (fmlookup\ xm\ x)\ (fmlookup\ ym\ x)$ )

**abbreviation**  $strict-subtuple-fun\ f\ xm\ ym \equiv$   
 $subtuple-fun\ f\ xm\ ym \wedge xm \neq ym$

**lemma**  $subtuple-fun-simp$  [*code-abbrev*, *simp*]:  
 $subtuple-fun\ f\ xm\ ym = subtuple\ f\ xm\ ym$   
*<proof>*

**lemma**  $strict-subtuple-fun-simp$  [*code-abbrev*, *simp*]:  
 $strict-subtuple-fun\ f\ xm\ ym = strict-subtuple\ f\ xm\ ym$   
*<proof>*

end

## 1.5 Object Model

**theory** *Object-Model*  
**imports** *HOL-Library.Extended-Nat Finite-Map-Ext*  
**begin**

The section defines a generic object model.

### 1.5.1 Type Synonyms

**type-synonym**  $attr = String.literal$   
**type-synonym**  $assoc = String.literal$   
**type-synonym**  $role = String.literal$   
**type-synonym**  $oper = String.literal$   
**type-synonym**  $param = String.literal$   
**type-synonym**  $elit = String.literal$

**datatype**  $param-dir = In \mid Out \mid InOut$

**type-synonym**  $'c\ assoc-end = 'c \times nat \times enat \times bool \times bool$   
**type-synonym**  $'t\ param-spec = param \times 't \times param-dir$   
**type-synonym**  $('t, 'e)\ oper-spec =$   
 $oper \times 't \times 't\ param-spec\ list \times 't \times bool \times 'e\ option$

**definition**  $assoc-end-class :: 'c\ assoc-end \Rightarrow 'c \equiv fst$

**definition**  $assoc-end-min :: 'c\ assoc-end \Rightarrow nat \equiv fst \circ snd$

**definition**  $assoc-end-max :: 'c\ assoc-end \Rightarrow enat \equiv fst \circ snd \circ snd$

**definition**  $assoc-end-ordered :: 'c\ assoc-end \Rightarrow bool \equiv fst \circ snd \circ snd \circ snd$

**definition**  $assoc-end-unique :: 'c\ assoc-end \Rightarrow bool \equiv snd \circ snd \circ snd \circ snd$

**definition**  $oper-name :: ('t, 'e)\ oper-spec \Rightarrow oper \equiv fst$

**definition**  $oper-context :: ('t, 'e)\ oper-spec \Rightarrow 't \equiv fst \circ snd$

**definition**  $oper-params :: ('t, 'e)\ oper-spec \Rightarrow 't\ param-spec\ list \equiv fst \circ snd \circ snd$

**definition**  $oper\text{-}result :: ('t, 'e) oper\text{-}spec \Rightarrow 't \equiv fst \circ snd \circ snd \circ snd$

**definition**  $oper\text{-}static :: ('t, 'e) oper\text{-}spec \Rightarrow bool \equiv fst \circ snd \circ snd \circ snd \circ snd$

**definition**  $oper\text{-}body :: ('t, 'e) oper\text{-}spec \Rightarrow 'e option \equiv snd \circ snd \circ snd \circ snd \circ snd$

**definition**  $param\text{-}name :: 't param\text{-}spec \Rightarrow param \equiv fst$

**definition**  $param\text{-}type :: 't param\text{-}spec \Rightarrow 't \equiv fst \circ snd$

**definition**  $param\text{-}dir :: 't param\text{-}spec \Rightarrow param\text{-}dir \equiv snd \circ snd$

**definition**  $oper\text{-}in\text{-}params op \equiv$   
 $filter (\lambda p. param\text{-}dir p = In \vee param\text{-}dir p = InOut) (oper\text{-}params op)$

**definition**  $oper\text{-}out\text{-}params op \equiv$   
 $filter (\lambda p. param\text{-}dir p = Out \vee param\text{-}dir p = InOut) (oper\text{-}params op)$

### 1.5.2 Attributes

**inductive**  $owned\text{-}attribute'$  **where**

$C \in | fmdom\ attributes \implies$   
 $fmlookup\ attributes\ C = Some\ attr_{sc} \implies$   
 $fmlookup\ attr_{sc}\ attr = Some\ \tau \implies$   
 $owned\text{-}attribute'\ attributes\ C\ attr\ \tau$

**inductive**  $attribute\text{-}not\text{-}closest$  **where**

$owned\text{-}attribute'\ attributes\ \mathcal{D}'\ attr\ \tau' \implies$   
 $C \leq \mathcal{D}' \implies$   
 $\mathcal{D}' < \mathcal{D} \implies$   
 $attribute\text{-}not\text{-}closest\ attributes\ C\ attr\ \mathcal{D}\ \tau$

**inductive**  $closest\text{-}attribute$  **where**

$owned\text{-}attribute'\ attributes\ \mathcal{D}\ attr\ \tau \implies$   
 $C \leq \mathcal{D} \implies$   
 $\neg attribute\text{-}not\text{-}closest\ attributes\ C\ attr\ \mathcal{D}\ \tau \implies$   
 $closest\text{-}attribute\ attributes\ C\ attr\ \mathcal{D}\ \tau$

**inductive**  $closest\text{-}attribute\text{-}not\text{-}unique$  **where**

$closest\text{-}attribute\ attributes\ C\ attr\ \mathcal{D}'\ \tau' \implies$   
 $\mathcal{D} \neq \mathcal{D}' \vee \tau \neq \tau' \implies$   
 $closest\text{-}attribute\text{-}not\text{-}unique\ attributes\ C\ attr\ \mathcal{D}\ \tau$

**inductive**  $unique\text{-}closest\text{-}attribute$  **where**

$closest\text{-}attribute\ attributes\ C\ attr\ \mathcal{D}\ \tau \implies$   
 $\neg closest\text{-}attribute\text{-}not\text{-}unique\ attributes\ C\ attr\ \mathcal{D}\ \tau \implies$   
 $unique\text{-}closest\text{-}attribute\ attributes\ C\ attr\ \mathcal{D}\ \tau$

### 1.5.3 Association Ends

**inductive**  $role\text{-}refer\text{-}class$  **where**

$role \in | fmdom\ ends \implies$   
 $fmlookup\ ends\ role = Some\ end \implies$

*assoc-end-class end = C*  $\implies$   
*role-refer-class ends C role*

**inductive association-ends' where**

*C | $\in$ | classes*  $\implies$   
*assoc | $\in$ | fmdom associations*  $\implies$   
*fmlookup associations assoc = Some ends*  $\implies$   
*role-refer-class ends C from*  $\implies$   
*role | $\in$ | fmdom ends*  $\implies$   
*fmlookup ends role = Some end*  $\implies$   
*role  $\neq$  from*  $\implies$   
*association-ends' classes associations C from role end*

**inductive association-ends-not-unique' where**

*association-ends' classes associations C from role end<sub>1</sub>*  $\implies$   
*association-ends' classes associations C from role end<sub>2</sub>*  $\implies$   
*end<sub>1</sub>  $\neq$  end<sub>2</sub>*  $\implies$   
*association-ends-not-unique' classes associations*

**inductive owned-association-end' where**

*association-ends' classes associations C from role end*  $\implies$   
*owned-association-end' classes associations C None role end*  
 | *association-ends' classes associations C from role end*  $\implies$   
*owned-association-end' classes associations C (Some from) role end*

**inductive association-end-not-closest where**

*owned-association-end' classes associations D' from role end'*  $\implies$   
*C  $\leq$  D'*  $\implies$   
*D' < D*  $\implies$   
*association-end-not-closest classes associations C from role D end*

**inductive closest-association-end where**

*owned-association-end' classes associations D from role end*  $\implies$   
*C  $\leq$  D*  $\implies$   
 $\neg$  *association-end-not-closest classes associations C from role D end*  $\implies$   
*closest-association-end classes associations C from role D end*

**inductive closest-association-end-not-unique where**

*closest-association-end classes associations C from role D' end'*  $\implies$   
*D  $\neq$  D'  $\vee$  end  $\neq$  end'*  $\implies$   
*closest-association-end-not-unique classes associations C from role D end*

**inductive unique-closest-association-end where**

*closest-association-end classes associations C from role D end*  $\implies$   
 $\neg$  *closest-association-end-not-unique classes associations C from role D end*  $\implies$   
*unique-closest-association-end classes associations C from role D end*

### 1.5.4 Association Classes

**inductive** *referred-by-association-class''* **where**

*fmlookup association-classes A = Some assoc*  $\implies$

*fmlookup associations assoc = Some ends*  $\implies$

*role-refer-class ends C from*  $\implies$

*referred-by-association-class'' association-classes associations C from A*

**inductive** *referred-by-association-class'* **where**

*referred-by-association-class'' association-classes associations C from A*  $\implies$

*referred-by-association-class' association-classes associations C None A*

| *referred-by-association-class'' association-classes associations C from A*  $\implies$

*referred-by-association-class' association-classes associations C (Some from) A*

**inductive** *association-class-not-closest* **where**

*referred-by-association-class' association-classes associations D' from A*  $\implies$

*C ≤ D'*  $\implies$

*D' < D*  $\implies$

*association-class-not-closest association-classes associations C from A D*

**inductive** *closest-association-class* **where**

*referred-by-association-class' association-classes associations D from A*  $\implies$

*C ≤ D*  $\implies$

$\neg$  *association-class-not-closest association-classes associations C from A D*  $\implies$

*closest-association-class association-classes associations C from A D*

**inductive** *closest-association-class-not-unique* **where**

*closest-association-class association-classes associations C from A D'*  $\implies$

*D ≠ D'*  $\implies$

*closest-association-class-not-unique*

*association-classes associations C from A D*

**inductive** *unique-closest-association-class* **where**

*closest-association-class association-classes associations C from A D*  $\implies$

$\neg$  *closest-association-class-not-unique*

*association-classes associations C from A D*  $\implies$

*unique-closest-association-class association-classes associations C from A D*

### 1.5.5 Association Class Ends

**inductive** *association-class-end'* **where**

*fmlookup association-classes A = Some assoc*  $\implies$

*fmlookup associations assoc = Some ends*  $\implies$

*fmlookup ends role = Some end*  $\implies$

*association-class-end' association-classes associations A role end*

**inductive** *association-class-end-not-unique* **where**

*association-class-end' association-classes associations A role end'*  $\implies$

*end ≠ end'*  $\implies$

*association-class-end-not-unique association-classes associations A role end*

**inductive** *unique-association-class-end* **where**  
*association-class-end' association-classes associations A role end*  $\implies$   
 $\neg$  *association-class-end-not-unique*  
*association-classes associations A role end*  $\implies$   
*unique-association-class-end association-classes associations A role end*

### 1.5.6 Operations

**inductive** *any-operation'* **where**  
*op | $\in$ | fset-of-list operations*  $\implies$   
*oper-name op = name*  $\implies$   
 $\tau \leq$  *oper-context op*  $\implies$   
*list-all2* ( $\lambda\sigma$  *param.  $\sigma \leq$  param-type param*)  $\pi$  (*oper-in-params op*)  $\implies$   
*any-operation' operations  $\tau$  name  $\pi$  op*

**inductive** *operation'* **where**  
*any-operation' operations  $\tau$  name  $\pi$  op*  $\implies$   
 $\neg$  *oper-static op*  $\implies$   
*operation' operations  $\tau$  name  $\pi$  op*

**inductive** *operation-not-unique* **where**  
*operation' operations  $\tau$  name  $\pi$  oper'*  $\implies$   
*oper  $\neq$  oper'*  $\implies$   
*operation-not-unique operations  $\tau$  name  $\pi$  oper*

**inductive** *unique-operation* **where**  
*operation' operations  $\tau$  name  $\pi$  oper*  $\implies$   
 $\neg$  *operation-not-unique operations  $\tau$  name  $\pi$  oper*  $\implies$   
*unique-operation operations  $\tau$  name  $\pi$  oper*

**inductive** *operation-defined'* **where**  
*unique-operation operations  $\tau$  name  $\pi$  oper*  $\implies$   
*operation-defined' operations  $\tau$  name  $\pi$*

**inductive** *static-operation'* **where**  
*any-operation' operations  $\tau$  name  $\pi$  op*  $\implies$   
*oper-static op*  $\implies$   
*static-operation' operations  $\tau$  name  $\pi$  op*

**inductive** *static-operation-not-unique* **where**  
*static-operation' operations  $\tau$  name  $\pi$  oper'*  $\implies$   
*oper  $\neq$  oper'*  $\implies$   
*static-operation-not-unique operations  $\tau$  name  $\pi$  oper*

**inductive** *unique-static-operation* **where**  
*static-operation' operations  $\tau$  name  $\pi$  oper*  $\implies$   
 $\neg$  *static-operation-not-unique operations  $\tau$  name  $\pi$  oper*  $\implies$   
*unique-static-operation operations  $\tau$  name  $\pi$  oper*

**inductive** *static-operation-defined'* **where**

*unique-static-operation operations*  $\tau$  *name*  $\pi$  *oper*  $\implies$   
*static-operation-defined' operations*  $\tau$  *name*  $\pi$

### 1.5.7 Literals

**inductive** *has-literal'* **where**

*fmlookup literals*  $e = \text{Some } lits \implies$   
*lit*  $|\in|$  *lits*  $\implies$   
*has-literal' literals*  $e$  *lit*

### 1.5.8 Definition

**locale** *object-model* =

**fixes** *classes*  $:: 'a :: \text{semilattice-sup fset}$   
**and** *attributes*  $:: 'a \rightarrow_f \text{attr} \rightarrow_f 't :: \text{order}$   
**and** *associations*  $:: \text{assoc} \rightarrow_f \text{role} \rightarrow_f 'a \text{ assoc-end}$   
**and** *association-classes*  $:: 'a \rightarrow_f \text{assoc}$   
**and** *operations*  $:: ('t, 'e) \text{ oper-spec list}$   
**and** *literals*  $:: 'n \rightarrow_f \text{elit fset}$

**assumes** *assoc-end-min-less-eq-max*:

*assoc*  $|\in|$  *fmdom associations*  $\implies$   
*fmlookup associations* *assoc* = *Some ends*  $\implies$   
*role*  $|\in|$  *fmdom ends*  $\implies$   
*fmlookup ends* *role* = *Some end*  $\implies$   
*assoc-end-min end*  $\leq$  *assoc-end-max end*

**assumes** *association-ends-unique*:

*association-ends' classes associations*  $\mathcal{C}$  *from role* *end*<sub>1</sub>  $\implies$   
*association-ends' classes associations*  $\mathcal{C}$  *from role* *end*<sub>2</sub>  $\implies$  *end*<sub>1</sub> = *end*<sub>2</sub>

**begin**

**abbreviation** *owned-attribute*  $\equiv$

*owned-attribute' attributes*

**abbreviation** *attribute*  $\equiv$

*unique-closest-attribute attributes*

**abbreviation** *association-ends*  $\equiv$

*association-ends' classes associations*

**abbreviation** *owned-association-end*  $\equiv$

*owned-association-end' classes associations*

**abbreviation** *association-end*  $\equiv$

*unique-closest-association-end classes associations*

**abbreviation** *referred-by-association-class*  $\equiv$

*unique-closest-association-class association-classes associations*

**abbreviation** *association-class-end*  $\equiv$   
*unique-association-class-end association-classes associations*

**abbreviation** *operation*  $\equiv$   
*unique-operation operations*

**abbreviation** *operation-defined*  $\equiv$   
*operation-defined' operations*

**abbreviation** *static-operation*  $\equiv$   
*unique-static-operation operations*

**abbreviation** *static-operation-defined*  $\equiv$   
*static-operation-defined' operations*

**abbreviation** *has-literal*  $\equiv$   
*has-literal' literals*

**end**

**declare** *operation-defined'.simps* [*simp*]  
**declare** *static-operation-defined'.simps* [*simp*]

**declare** *has-literal'.simps* [*simp*]

### 1.5.9 Properties

**lemma** (**in** *object-model*) *attribute-det*:  
*attribute*  $\mathcal{C}$  *attr*  $\mathcal{D}_1$   $\tau_1 \implies$   
*attribute*  $\mathcal{C}$  *attr*  $\mathcal{D}_2$   $\tau_2 \implies \mathcal{D}_1 = \mathcal{D}_2 \wedge \tau_1 = \tau_2$   
*<proof>*

**lemma** (**in** *object-model*) *attribute-self-or-inherited*:  
*attribute*  $\mathcal{C}$  *attr*  $\mathcal{D}$   $\tau \implies \mathcal{C} \leq \mathcal{D}$   
*<proof>*

**lemma** (**in** *object-model*) *attribute-closest*:  
*attribute*  $\mathcal{C}$  *attr*  $\mathcal{D}$   $\tau \implies$   
*owned-attribute*  $\mathcal{D}'$  *attr*  $\tau \implies$   
 $\mathcal{C} \leq \mathcal{D}' \implies \neg \mathcal{D}' < \mathcal{D}$   
*<proof>*

**lemma** (**in** *object-model*) *association-end-det*:  
*association-end*  $\mathcal{C}$  *from role*  $\mathcal{D}_1$  *end* $_1 \implies$   
*association-end*  $\mathcal{C}$  *from role*  $\mathcal{D}_2$  *end* $_2 \implies \mathcal{D}_1 = \mathcal{D}_2 \wedge \text{end}_1 = \text{end}_2$   
*<proof>*

**lemma** (**in** *object-model*) *association-end-self-or-inherited*:





```

i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ o ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ o ⇒ bool,
i ⇒ i ⇒ o ⇒ i ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ o ⇒ i ⇒ i ⇒ o ⇒ bool,
i ⇒ i ⇒ o ⇒ i ⇒ o ⇒ i ⇒ bool,
i ⇒ i ⇒ o ⇒ i ⇒ o ⇒ o ⇒ bool,
i ⇒ i ⇒ o ⇒ o ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ o ⇒ o ⇒ i ⇒ o ⇒ bool,
i ⇒ i ⇒ o ⇒ o ⇒ o ⇒ i ⇒ bool,
i ⇒ i ⇒ o ⇒ o ⇒ o ⇒ o ⇒ bool) association-ends'
⟨proof⟩

```

**code-pred** *association-ends-not-unique'* ⟨*proof*⟩

**code-pred** (*modes*:

```

i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ o ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ o ⇒ bool,
i ⇒ i ⇒ o ⇒ i ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ o ⇒ i ⇒ i ⇒ o ⇒ bool,
i ⇒ i ⇒ o ⇒ i ⇒ o ⇒ i ⇒ bool,
i ⇒ i ⇒ o ⇒ i ⇒ o ⇒ o ⇒ bool,
i ⇒ i ⇒ o ⇒ o ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ o ⇒ o ⇒ i ⇒ o ⇒ bool,
i ⇒ i ⇒ o ⇒ o ⇒ o ⇒ i ⇒ bool,
i ⇒ i ⇒ o ⇒ o ⇒ o ⇒ o ⇒ bool) owned-association-end' ⟨proof⟩

```

**code-pred** (*modes*:

```

i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ o ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ o ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ o ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ o ⇒ bool,

```

```

i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ o ⇒ o ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ i ⇒ o ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ o ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ o ⇒ o ⇒ bool) closest-association-end ⟨proof⟩

```

**code-pred** (*modes*:

```

i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ i ⇒ i ⇒ bool) closest-association-end-not-unique
⟨proof⟩

```

**code-pred** (*modes*:

```

i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ bool) unique-closest-association-end ⟨proof⟩

```

**code-pred** *unique-closest-association-class* ⟨*proof*⟩

**code-pred** *association-class-end'* ⟨*proof*⟩

**code-pred** *association-class-end-not-unique* ⟨*proof*⟩

**code-pred** *unique-association-class-end* ⟨*proof*⟩

**declare** *any-operation'*.intros[folded *Predicate-Compile.contains-def*, *code-pred-intro*]

**code-pred** [*show-modes*] *any-operation'*  
 ⟨*proof*⟩

**code-pred** (*modes*:

```

i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool) operation' ⟨proof⟩

```

**code-pred** (*modes*:

```

i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool) operation-not-unique ⟨proof⟩

```

**code-pred** (*modes*:

```

i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool) unique-operation ⟨proof⟩

```

**code-pred** *operation-defined'* ⟨*proof*⟩

**code-pred** (*modes*:

```

i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool) static-operation' ⟨proof⟩

```

**code-pred** (*modes*:

```
i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool) static-operation-not-unique ⟨proof⟩
```

```
code-pred (modes:
```

```
  i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool,
```

```
  i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool) unique-static-operation ⟨proof⟩
```

```
code-pred static-operation-defined' ⟨proof⟩
```

```
declare has-literal'.intros[folded Predicate-Compile.contains-def, code-pred-intro]
```

```
code-pred (modes:
```

```
  i ⇒ i ⇒ i ⇒ bool, i ⇒ i ⇒ o ⇒ bool) has-literal'
```

```
  ⟨proof⟩
```

```
end
```

# Chapter 2

## Basic Types

```
theory OCL-Basic-Types
  imports Main HOL-Library.FSet HOL-Library.Phantom-Type
begin
```

### 2.1 Definition

Basic types are parameterized over classes.

```
type-synonym 'a enum = ('a, String.literal) phantom
```

```
type-synonym elit = String.literal
```

```
datatype ('a :: order) basic-type =
```

```
  OclAny
| OclVoid
| Boolean
| Real
| Integer
| UnlimitedNatural
| String
| ObjectType 'a ( $\langle - \rangle_{\mathcal{T}}$  [0] 1000)
| Enum 'a enum
```

```
inductive basic-subtype (infix  $\sqsubseteq_B$  65) where
```

```
  OclVoid  $\sqsubseteq_B$  Boolean
| OclVoid  $\sqsubseteq_B$  UnlimitedNatural
| OclVoid  $\sqsubseteq_B$  String
| OclVoid  $\sqsubseteq_B$   $\langle \mathcal{C} \rangle_{\mathcal{T}}$ 
| OclVoid  $\sqsubseteq_B$  Enum  $\mathcal{E}$ 

| UnlimitedNatural  $\sqsubseteq_B$  Integer
| Integer  $\sqsubseteq_B$  Real
|  $\mathcal{C} < \mathcal{D} \implies \langle \mathcal{C} \rangle_{\mathcal{T}} \sqsubseteq_B \langle \mathcal{D} \rangle_{\mathcal{T}}$ 

| Boolean  $\sqsubseteq_B$  OclAny
| Real  $\sqsubseteq_B$  OclAny
```

```
| String  $\sqsubset_B$  OclAny
|  $\langle C \rangle_{\mathcal{T}}$   $\sqsubset_B$  OclAny
| Enum  $\mathcal{E}$   $\sqsubset_B$  OclAny
```

**declare** *basic-subtype.intros* [*intro!*]

```
inductive-cases basic-subtype-x-OclAny [elim!]:  $\tau \sqsubset_B$  OclAny
inductive-cases basic-subtype-x-OclVoid [elim!]:  $\tau \sqsubset_B$  OclVoid
inductive-cases basic-subtype-x-Boolean [elim!]:  $\tau \sqsubset_B$  Boolean
inductive-cases basic-subtype-x-Real [elim!]:  $\tau \sqsubset_B$  Real
inductive-cases basic-subtype-x-Integer [elim!]:  $\tau \sqsubset_B$  Integer
inductive-cases basic-subtype-x-UnlimitedNatural [elim!]:  $\tau \sqsubset_B$  UnlimitedNatural
```

```
inductive-cases basic-subtype-x-String [elim!]:  $\tau \sqsubset_B$  String
inductive-cases basic-subtype-x-ObjectType [elim!]:  $\tau \sqsubset_B$   $\langle C \rangle_{\mathcal{T}}$ 
inductive-cases basic-subtype-x-Enum [elim!]:  $\tau \sqsubset_B$  Enum  $\mathcal{E}$ 
```

```
inductive-cases basic-subtype-OclAny-x [elim!]: OclAny  $\sqsubset_B$   $\sigma$ 
inductive-cases basic-subtype-ObjectType-x [elim!]:  $\langle C \rangle_{\mathcal{T}}$   $\sqsubset_B$   $\sigma$ 
```

**lemma** *basic-subtype-asym*:

```
 $\tau \sqsubset_B \sigma \implies \sigma \sqsubset_B \tau \implies \text{False}$ 
<proof>
```

## 2.2 Partial Order of Basic Types

**instantiation** *basic-type* :: (*order*) *order*  
**begin**

**definition** ( $<$ )  $\equiv$  *basic-subtype*<sup>++</sup>

**definition** ( $\leq$ )  $\equiv$  *basic-subtype*<sup>\*\*</sup>

### 2.2.1 Strict Introduction Rules

**lemma** *type-less-x-OclAny-intro* [*intro*]:

```
 $\tau \neq \text{OclAny} \implies \tau < \text{OclAny}$ 
<proof>
```

**lemma** *type-less-OclVoid-x-intro* [*intro*]:

```
 $\tau \neq \text{OclVoid} \implies \text{OclVoid} < \tau$ 
<proof>
```

**lemma** *type-less-x-Real-intro* [*intro*]:

```
 $\tau = \text{UnlimitedNatural} \implies \tau < \text{Real}$ 
 $\tau = \text{Integer} \implies \tau < \text{Real}$ 
<proof>
```

**lemma** *type-less-x-Integer-intro* [*intro*]:

```
 $\tau = \text{UnlimitedNatural} \implies \tau < \text{Integer}$ 
```

*<proof>*

**lemma** *type-less-x-ObjectType-intro* [*intro*]:

$\tau = \langle \mathcal{C} \rangle_{\mathcal{T}} \implies \mathcal{C} < \mathcal{D} \implies \tau < \langle \mathcal{D} \rangle_{\mathcal{T}}$   
*<proof>*

### 2.2.2 Strict Elimination Rules

**lemma** *type-less-x-OclAny* [*elim!*]:

$\tau < OclAny \implies$   
 $(\tau = OclVoid \implies P) \implies$   
 $(\tau = Boolean \implies P) \implies$   
 $(\tau = Integer \implies P) \implies$   
 $(\tau = UnlimitedNatural \implies P) \implies$   
 $(\tau = Real \implies P) \implies$   
 $(\tau = String \implies P) \implies$   
 $(\bigwedge \mathcal{E}. \tau = Enum \ \mathcal{E} \implies P) \implies$   
 $(\bigwedge \mathcal{C}. \tau = \langle \mathcal{C} \rangle_{\mathcal{T}} \implies P) \implies P$   
*<proof>*

**lemma** *type-less-x-OclVoid* [*elim!*]:

$\tau < OclVoid \implies P$   
*<proof>*

**lemma** *type-less-x-Boolean* [*elim!*]:

$\tau < Boolean \implies$   
 $(\tau = OclVoid \implies P) \implies P$   
*<proof>*

**lemma** *type-less-x-Real* [*elim!*]:

$\tau < Real \implies$   
 $(\tau = OclVoid \implies P) \implies$   
 $(\tau = UnlimitedNatural \implies P) \implies$   
 $(\tau = Integer \implies P) \implies P$   
*<proof>*

**lemma** *type-less-x-Integer* [*elim!*]:

$\tau < Integer \implies$   
 $(\tau = OclVoid \implies P) \implies$   
 $(\tau = UnlimitedNatural \implies P) \implies P$   
*<proof>*

**lemma** *type-less-x-UnlimitedNatural* [*elim!*]:

$\tau < UnlimitedNatural \implies$   
 $(\tau = OclVoid \implies P) \implies P$   
*<proof>*

**lemma** *type-less-x-String* [*elim!*]:

$\tau < String \implies$

$(\tau = \text{OclVoid} \implies P) \implies P$   
 $\langle \text{proof} \rangle$

**lemma** *type-less-x-ObjectType* [elim!]:

$\tau < \langle \mathcal{D} \rangle_{\mathcal{T}} \implies$   
 $(\tau = \text{OclVoid} \implies P) \implies$   
 $(\wedge \mathcal{C}. \tau = \langle \mathcal{C} \rangle_{\mathcal{T}} \implies \mathcal{C} < \mathcal{D} \implies P) \implies P$   
 $\langle \text{proof} \rangle$

**lemma** *type-less-x-Enum* [elim!]:

$\tau < \text{Enum } \mathcal{E} \implies$   
 $(\tau = \text{OclVoid} \implies P) \implies P$   
 $\langle \text{proof} \rangle$

### 2.2.3 Properties

**lemma** *basic-subtype-irrefl*:

$\tau < \tau \implies \text{False}$   
**for**  $\tau :: 'a \text{ basic-type}$   
 $\langle \text{proof} \rangle$

**lemma** *transclp-less-basic-type*:

$(\tau, \sigma) \in \{(\tau, \sigma). \tau \sqsubset_B \sigma\}^+ \iff \tau < \sigma$   
 $\langle \text{proof} \rangle$

**lemma** *basic-subtype-acyclic*:

*acyclicP basic-subtype*  
 $\langle \text{proof} \rangle$

**lemma** *less-le-not-le-basic-type*:

$\tau < \sigma \iff \tau \leq \sigma \wedge \neg \sigma \leq \tau$   
**for**  $\tau \sigma :: 'a \text{ basic-type}$   
 $\langle \text{proof} \rangle$

**lemma** *antisym-basic-type*:

$\tau \leq \sigma \implies \sigma \leq \tau \implies \tau = \sigma$   
**for**  $\tau \sigma :: 'a \text{ basic-type}$   
 $\langle \text{proof} \rangle$

**lemma** *order-refl-basic-type* [iff]:

$\tau \leq \tau$   
**for**  $\tau :: 'a \text{ basic-type}$   
 $\langle \text{proof} \rangle$

**instance**

$\langle \text{proof} \rangle$

**end**



### 2.2.4 Non-Strict Introduction Rules

**lemma** *type-less-eq-x-OclAny-intro* [intro]:

$$\tau \leq \text{OclAny}$$

*<proof>*

**lemma** *type-less-eq-OclVoid-x-intro* [intro]:

$$\text{OclVoid} \leq \tau$$

*<proof>*

**lemma** *type-less-eq-x-Real-intro* [intro]:

$$\tau = \text{UnlimitedNatural} \implies \tau \leq \text{Real}$$

$$\tau = \text{Integer} \implies \tau \leq \text{Real}$$

*<proof>*

**lemma** *type-less-eq-x-Integer-intro* [intro]:

$$\tau = \text{UnlimitedNatural} \implies \tau \leq \text{Integer}$$

*<proof>*

**lemma** *type-less-eq-x-ObjectType-intro* [intro]:

$$\tau = \langle \mathcal{C} \rangle_{\mathcal{T}} \implies \mathcal{C} \leq \mathcal{D} \implies \tau \leq \langle \mathcal{D} \rangle_{\mathcal{T}}$$

*<proof>*

### 2.2.5 Non-Strict Elimination Rules

**lemma** *type-less-eq-x-OclAny* [elim!]:

$$\tau \leq \text{OclAny} \implies$$

$$(\tau = \text{OclVoid} \implies P) \implies$$

$$(\tau = \text{OclAny} \implies P) \implies$$

$$(\tau = \text{Boolean} \implies P) \implies$$

$$(\tau = \text{Integer} \implies P) \implies$$

$$(\tau = \text{UnlimitedNatural} \implies P) \implies$$

$$(\tau = \text{Real} \implies P) \implies$$

$$(\tau = \text{String} \implies P) \implies$$

$$(\bigwedge \mathcal{E}. \tau = \text{Enum } \mathcal{E} \implies P) \implies$$

$$(\bigwedge \mathcal{C}. \tau = \langle \mathcal{C} \rangle_{\mathcal{T}} \implies P) \implies P$$

*<proof>*

**lemma** *type-less-eq-x-OclVoid* [elim!]:

$$\tau \leq \text{OclVoid} \implies$$

$$(\tau = \text{OclVoid} \implies P) \implies P$$

*<proof>*

**lemma** *type-less-eq-x-Boolean* [elim!]:

$$\tau \leq \text{Boolean} \implies$$

$$(\tau = \text{OclVoid} \implies P) \implies$$

$$(\tau = \text{Boolean} \implies P) \implies P$$

*<proof>*

**lemma** *type-less-eq-x-Real* [elim!]:

$$\begin{aligned}
&\tau \leq \mathit{Real} \implies \\
&(\tau = \mathit{OclVoid} \implies P) \implies \\
&(\tau = \mathit{UnlimitedNatural} \implies P) \implies \\
&(\tau = \mathit{Integer} \implies P) \implies \\
&(\tau = \mathit{Real} \implies P) \implies P \\
&\langle \mathit{proof} \rangle
\end{aligned}$$

**lemma** *type-less-eq-x-Integer* [elim!]:

$$\begin{aligned}
&\tau \leq \mathit{Integer} \implies \\
&(\tau = \mathit{OclVoid} \implies P) \implies \\
&(\tau = \mathit{UnlimitedNatural} \implies P) \implies \\
&(\tau = \mathit{Integer} \implies P) \implies P \\
&\langle \mathit{proof} \rangle
\end{aligned}$$

**lemma** *type-less-eq-x-UnlimitedNatural* [elim!]:

$$\begin{aligned}
&\tau \leq \mathit{UnlimitedNatural} \implies \\
&(\tau = \mathit{OclVoid} \implies P) \implies \\
&(\tau = \mathit{UnlimitedNatural} \implies P) \implies P \\
&\langle \mathit{proof} \rangle
\end{aligned}$$

**lemma** *type-less-eq-x-String* [elim!]:

$$\begin{aligned}
&\tau \leq \mathit{String} \implies \\
&(\tau = \mathit{OclVoid} \implies P) \implies \\
&(\tau = \mathit{String} \implies P) \implies P \\
&\langle \mathit{proof} \rangle
\end{aligned}$$

**lemma** *type-less-eq-x-ObjectType* [elim!]:

$$\begin{aligned}
&\tau \leq \langle \mathcal{D} \rangle_{\tau} \implies \\
&(\tau = \mathit{OclVoid} \implies P) \implies \\
&(\bigwedge \mathcal{C}. \tau = \langle \mathcal{C} \rangle_{\tau} \implies \mathcal{C} \leq \mathcal{D} \implies P) \implies P \\
&\langle \mathit{proof} \rangle
\end{aligned}$$

**lemma** *type-less-eq-x-Enum* [elim!]:

$$\begin{aligned}
&\tau \leq \mathit{Enum} \mathcal{E} \implies \\
&(\tau = \mathit{OclVoid} \implies P) \implies \\
&(\tau = \mathit{Enum} \mathcal{E} \implies P) \implies P \\
&\langle \mathit{proof} \rangle
\end{aligned}$$

## 2.2.6 Simplification Rules

**lemma** *basic-type-less-left-simps* [simp]:

$$\begin{aligned}
&\mathit{OclAny} < \sigma = \mathit{False} \\
&\mathit{OclVoid} < \sigma = (\sigma \neq \mathit{OclVoid}) \\
&\mathit{Boolean} < \sigma = (\sigma = \mathit{OclAny}) \\
&\mathit{Real} < \sigma = (\sigma = \mathit{OclAny}) \\
&\mathit{Integer} < \sigma = (\sigma = \mathit{OclAny} \vee \sigma = \mathit{Real}) \\
&\mathit{UnlimitedNatural} < \sigma = (\sigma = \mathit{OclAny} \vee \sigma = \mathit{Real} \vee \sigma = \mathit{Integer}) \\
&\mathit{String} < \sigma = (\sigma = \mathit{OclAny}) \\
&\mathit{ObjectType} \mathcal{C} < \sigma = (\exists \mathcal{D}. \sigma = \mathit{OclAny} \vee \sigma = \mathit{ObjectType} \mathcal{D} \wedge \mathcal{C} < \mathcal{D})
\end{aligned}$$

*Enum*  $\mathcal{E} < \sigma = (\sigma = \text{OclAny})$   
 ⟨proof⟩

**lemma** *basic-type-less-right-simps* [simp]:

$\tau < \text{OclAny} = (\tau \neq \text{OclAny})$   
 $\tau < \text{OclVoid} = \text{False}$   
 $\tau < \text{Boolean} = (\tau = \text{OclVoid})$   
 $\tau < \text{Real} = (\tau = \text{Integer} \vee \tau = \text{UnlimitedNatural} \vee \tau = \text{OclVoid})$   
 $\tau < \text{Integer} = (\tau = \text{UnlimitedNatural} \vee \tau = \text{OclVoid})$   
 $\tau < \text{UnlimitedNatural} = (\tau = \text{OclVoid})$   
 $\tau < \text{String} = (\tau = \text{OclVoid})$   
 $\tau < \text{ObjectType } \mathcal{D} = (\exists \mathcal{C}. \tau = \text{ObjectType } \mathcal{C} \wedge \mathcal{C} < \mathcal{D} \vee \tau = \text{OclVoid})$   
 $\tau < \text{Enum } \mathcal{E} = (\tau = \text{OclVoid})$   
 ⟨proof⟩

## 2.3 Upper Semilattice of Basic Types

**notation** *sup* (infixl  $\sqcup$  65)

**instantiation** *basic-type* :: (semilattice-sup) semilattice-sup  
 begin

**fun** *sup-basic-type* **where**

$\langle \mathcal{C} \rangle_{\mathcal{T}} \sqcup \sigma = (\text{case } \sigma \text{ of } \text{OclVoid} \Rightarrow \langle \mathcal{C} \rangle_{\mathcal{T}} \mid \langle \mathcal{D} \rangle_{\mathcal{T}} \Rightarrow \langle \mathcal{C} \sqcup \mathcal{D} \rangle_{\mathcal{T}} \mid - \Rightarrow \text{OclAny})$   
 $\mid \tau \sqcup \sigma = (\text{if } \tau \leq \sigma \text{ then } \sigma \text{ else } (\text{if } \sigma \leq \tau \text{ then } \tau \text{ else } \text{OclAny}))$

**lemma** *sup-ge1-ObjectType*:

$\langle \mathcal{C} \rangle_{\mathcal{T}} \leq \langle \mathcal{C} \rangle_{\mathcal{T}} \sqcup \sigma$   
 ⟨proof⟩

**lemma** *sup-ge1-basic-type*:

$\tau \leq \tau \sqcup \sigma$   
**for**  $\tau \sigma :: 'a \text{ basic-type}$   
 ⟨proof⟩

**lemma** *sup-commut-basic-type*:

$\tau \sqcup \sigma = \sigma \sqcup \tau$   
**for**  $\tau \sigma :: 'a \text{ basic-type}$   
 ⟨proof⟩

**lemma** *sup-least-basic-type*:

$\tau \leq \varrho \Longrightarrow \sigma \leq \varrho \Longrightarrow \tau \sqcup \sigma \leq \varrho$   
**for**  $\tau \sigma \varrho :: 'a \text{ basic-type}$   
 ⟨proof⟩

**instance**

⟨proof⟩

end

## 2.4 Code Setup

**code-pred** *basic-subtype*  $\langle proof \rangle$

```

fun basic-subtype-fun :: 'a::order basic-type  $\Rightarrow$  'a basic-type  $\Rightarrow$  bool where
  basic-subtype-fun OclAny  $\sigma = False$ 
| basic-subtype-fun OclVoid  $\sigma = (\sigma \neq OclVoid)$ 
| basic-subtype-fun Boolean  $\sigma = (\sigma = OclAny)$ 
| basic-subtype-fun Real  $\sigma = (\sigma = OclAny)$ 
| basic-subtype-fun Integer  $\sigma = (\sigma = Real \vee \sigma = OclAny)$ 
| basic-subtype-fun UnlimitedNatural  $\sigma = (\sigma = Integer \vee \sigma = Real \vee \sigma = OclAny)$ 

| basic-subtype-fun String  $\sigma = (\sigma = OclAny)$ 
| basic-subtype-fun  $\langle C \rangle_{\mathcal{T}}$   $\sigma = (case \sigma$ 
  of  $\langle D \rangle_{\mathcal{T}} \Rightarrow C < D$ 
  | OclAny  $\Rightarrow True$ 
  | -  $\Rightarrow False)$ 
| basic-subtype-fun (Enum -)  $\sigma = (\sigma = OclAny)$ 

```

**lemma** *less-basic-type-code* [*code*]:

$\langle < \rangle = \text{basic-subtype-fun}$   
 $\langle proof \rangle$

**lemma** *less-eq-basic-type-code* [*code*]:

$\langle \leq \rangle = (\lambda x y. \text{basic-subtype-fun } x y \vee x = y)$   
 $\langle proof \rangle$

end

# Chapter 3

## Types

```
theory OCL-Types
  imports OCL-Basic-Types Errorable Tuple
begin
```

### 3.1 Definition

Types are parameterized over classes.

```
type-synonym telem = String.literal
```

```
datatype (plugins del: size) 'a type =
  OclSuper
| Required 'a basic-type ( -[1] [1000] 1000)
| Optional 'a basic-type ( -[?] [1000] 1000)
| Collection 'a type
| Set 'a type
| OrderedSet 'a type
| Bag 'a type
| Sequence 'a type
| Tuple telem  $\rightarrow_f$  'a type
```

We define the *OclInvalid* type separately, because we do not need types like *Set(OclInvalid)* in the theory. The *OclVoid[1]* type is not equal to *OclInvalid*. For example, *Set(OclVoid[1])* could theoretically be a valid type of the following expression:

```
Set{null}->excluding(null)
```

```
definition OclInvalid :: 'a type⊥ ≡ ⊥
```

```
instantiation type :: (type) size
begin
```

```
primrec size-type :: 'a type ⇒ nat where
  size-type OclSuper = 0
| size-type (Required τ) = 0
```

```

| size-type (Optional  $\tau$ ) = 0
| size-type (Collection  $\tau$ ) = Suc (size-type  $\tau$ )
| size-type (Set  $\tau$ ) = Suc (size-type  $\tau$ )
| size-type (OrderedSet  $\tau$ ) = Suc (size-type  $\tau$ )
| size-type (Bag  $\tau$ ) = Suc (size-type  $\tau$ )
| size-type (Sequence  $\tau$ ) = Suc (size-type  $\tau$ )
| size-type (Tuple  $\pi$ ) = Suc (ffold tcf 0 (fset-of-fmap (fmmap size-type  $\pi$ )))

```

**instance**  $\langle proof \rangle$

**end**

**inductive** *subtype* :: 'a::order type  $\Rightarrow$  'a type  $\Rightarrow$  bool (**infix**  $\sqsubset$  65) **where**

```

   $\tau \sqsubset_B \sigma \Longrightarrow \tau[1] \sqsubset \sigma[1]$ 
|  $\tau \sqsubset_B \sigma \Longrightarrow \tau[?] \sqsubset \sigma[?]$ 
|  $\tau[1] \sqsubset \tau[?]$ 
| OclAny[?]  $\sqsubset$  OclSuper

|  $\tau \sqsubset \sigma \Longrightarrow$  Collection  $\tau \sqsubset$  Collection  $\sigma$ 
|  $\tau \sqsubset \sigma \Longrightarrow$  Set  $\tau \sqsubset$  Set  $\sigma$ 
|  $\tau \sqsubset \sigma \Longrightarrow$  OrderedSet  $\tau \sqsubset$  OrderedSet  $\sigma$ 
|  $\tau \sqsubset \sigma \Longrightarrow$  Bag  $\tau \sqsubset$  Bag  $\sigma$ 
|  $\tau \sqsubset \sigma \Longrightarrow$  Sequence  $\tau \sqsubset$  Sequence  $\sigma$ 
| Set  $\tau \sqsubset$  Collection  $\tau$ 
| OrderedSet  $\tau \sqsubset$  Collection  $\tau$ 
| Bag  $\tau \sqsubset$  Collection  $\tau$ 
| Sequence  $\tau \sqsubset$  Collection  $\tau$ 
| Collection OclSuper  $\sqsubset$  OclSuper

| strict-subtuple ( $\lambda\tau \sigma. \tau \sqsubset \sigma \vee \tau = \sigma$ )  $\pi \xi \Longrightarrow$ 
  Tuple  $\pi \sqsubset$  Tuple  $\xi$ 
| Tuple  $\pi \sqsubset$  OclSuper

```

**declare** *subtype.intros* [intro!]

```

inductive-cases subtype-x-OclSuper [elim!]:  $\tau \sqsubset$  OclSuper
inductive-cases subtype-x-Required [elim!]:  $\tau \sqsubset \sigma[1]$ 
inductive-cases subtype-x-Optional [elim!]:  $\tau \sqsubset \sigma[?]$ 
inductive-cases subtype-x-Collection [elim!]:  $\tau \sqsubset$  Collection  $\sigma$ 
inductive-cases subtype-x-Set [elim!]:  $\tau \sqsubset$  Set  $\sigma$ 
inductive-cases subtype-x-OrderedSet [elim!]:  $\tau \sqsubset$  OrderedSet  $\sigma$ 
inductive-cases subtype-x-Bag [elim!]:  $\tau \sqsubset$  Bag  $\sigma$ 
inductive-cases subtype-x-Sequence [elim!]:  $\tau \sqsubset$  Sequence  $\sigma$ 
inductive-cases subtype-x-Tuple [elim!]:  $\tau \sqsubset$  Tuple  $\pi$ 

```

```

inductive-cases subtype-OclSuper-x [elim!]: OclSuper  $\sqsubset$   $\sigma$ 
inductive-cases subtype-Collection-x [elim!]: Collection  $\tau \sqsubset$   $\sigma$ 

```

**lemma** *subtype-asym*:

$\tau \sqsubset \sigma \implies \sigma \sqsubset \tau \implies \text{False}$   
 ⟨proof⟩

## 3.2 Constructors Bijectivity on Transitive Closures

**lemma** *Required-bij-on-trancl* [simp]:  
*bij-on-trancl subtype Required*  
 ⟨proof⟩

**lemma** *not-subtype-Optional-Required*:  
 $\text{subtype}^{++} \tau[\?] \sigma \implies \sigma = \varrho[1] \implies P$   
 ⟨proof⟩

**lemma** *Optional-bij-on-trancl* [simp]:  
*bij-on-trancl subtype Optional*  
 ⟨proof⟩

**lemma** *subtype-tranclp-Collection-x*:  
 $\text{subtype}^{++} (\text{Collection } \tau) \sigma \implies$   
 $(\bigwedge \varrho. \sigma = \text{Collection } \varrho \implies \text{subtype}^{++} \tau \varrho \implies P) \implies$   
 $(\sigma = \text{OclSuper} \implies P) \implies P$   
 ⟨proof⟩

**lemma** *Collection-bij-on-trancl* [simp]:  
*bij-on-trancl subtype Collection*  
 ⟨proof⟩

**lemma** *Set-bij-on-trancl* [simp]:  
*bij-on-trancl subtype Set*  
 ⟨proof⟩

**lemma** *OrderedSet-bij-on-trancl* [simp]:  
*bij-on-trancl subtype OrderedSet*  
 ⟨proof⟩

**lemma** *Bag-bij-on-trancl* [simp]:  
*bij-on-trancl subtype Bag*  
 ⟨proof⟩

**lemma** *Sequence-bij-on-trancl* [simp]:  
*bij-on-trancl subtype Sequence*  
 ⟨proof⟩

**lemma** *Tuple-bij-on-trancl* [simp]:  
*bij-on-trancl subtype Tuple*  
 ⟨proof⟩

### 3.3 Partial Order of Types

**instantiation**  $type :: (order) order$   
**begin**

**definition**  $(<) \equiv subtype^{++}$

**definition**  $(\leq) \equiv subtype^{**}$

#### 3.3.1 Strict Introduction Rules

**lemma** *type-less-x-Required-intro* [intro]:  
 $\tau = \varrho[1] \implies \varrho < \sigma \implies \tau < \sigma[1]$   
 ⟨proof⟩

**lemma** *type-less-x-Optional-intro* [intro]:  
 $\tau = \varrho[1] \implies \varrho \leq \sigma \implies \tau < \sigma[?]$   
 $\tau = \varrho[?] \implies \varrho < \sigma \implies \tau < \sigma[?]$   
 ⟨proof⟩

**lemma** *type-less-x-Collection-intro* [intro]:  
 $\tau = Collection \varrho \implies \varrho < \sigma \implies \tau < Collection \sigma$   
 $\tau = Set \varrho \implies \varrho \leq \sigma \implies \tau < Collection \sigma$   
 $\tau = OrderedSet \varrho \implies \varrho \leq \sigma \implies \tau < Collection \sigma$   
 $\tau = Bag \varrho \implies \varrho \leq \sigma \implies \tau < Collection \sigma$   
 $\tau = Sequence \varrho \implies \varrho \leq \sigma \implies \tau < Collection \sigma$   
 ⟨proof⟩

**lemma** *type-less-x-Set-intro* [intro]:  
 $\tau = Set \varrho \implies \varrho < \sigma \implies \tau < Set \sigma$   
 ⟨proof⟩

**lemma** *type-less-x-OrderedSet-intro* [intro]:  
 $\tau = OrderedSet \varrho \implies \varrho < \sigma \implies \tau < OrderedSet \sigma$   
 ⟨proof⟩

**lemma** *type-less-x-Bag-intro* [intro]:  
 $\tau = Bag \varrho \implies \varrho < \sigma \implies \tau < Bag \sigma$   
 ⟨proof⟩

**lemma** *type-less-x-Sequence-intro* [intro]:  
 $\tau = Sequence \varrho \implies \varrho < \sigma \implies \tau < Sequence \sigma$   
 ⟨proof⟩

**lemma** *fun-or-eq-refl* [intro]:  
 $reflp (\lambda x y. f x y \vee x = y)$   
 ⟨proof⟩

**lemma** *type-less-x-Tuple-intro* [intro]:  
**assumes**  $\tau = Tuple \pi$   
**and** *strict-subtuple*  $(\leq) \pi \xi$



**shows**  $\tau < \text{Tuple } \xi$   
 $\langle \text{proof} \rangle$

**lemma** *type-less-x-OclSuper-intro* [intro]:  
 $\tau \neq \text{OclSuper} \implies \tau < \text{OclSuper}$   
 $\langle \text{proof} \rangle$

### 3.3.2 Strict Elimination Rules

**lemma** *type-less-x-Required* [elim!]:  
**assumes**  $\tau < \sigma[1]$   
**and**  $\bigwedge \varrho. \tau = \varrho[1] \implies \varrho < \sigma \implies P$   
**shows**  $P$   
 $\langle \text{proof} \rangle$

**lemma** *type-less-x-Optional* [elim!]:  
 $\tau < \sigma[?] \implies$   
 $(\bigwedge \varrho. \tau = \varrho[1] \implies \varrho \leq \sigma \implies P) \implies$   
 $(\bigwedge \varrho. \tau = \varrho[?] \implies \varrho < \sigma \implies P) \implies P$   
 $\langle \text{proof} \rangle$

**lemma** *type-less-x-Collection* [elim!]:  
 $\tau < \text{Collection } \sigma \implies$   
 $(\bigwedge \varrho. \tau = \text{Collection } \varrho \implies \varrho < \sigma \implies P) \implies$   
 $(\bigwedge \varrho. \tau = \text{Set } \varrho \implies \varrho \leq \sigma \implies P) \implies$   
 $(\bigwedge \varrho. \tau = \text{OrderedSet } \varrho \implies \varrho \leq \sigma \implies P) \implies$   
 $(\bigwedge \varrho. \tau = \text{Bag } \varrho \implies \varrho \leq \sigma \implies P) \implies$   
 $(\bigwedge \varrho. \tau = \text{Sequence } \varrho \implies \varrho \leq \sigma \implies P) \implies P$   
 $\langle \text{proof} \rangle$

**lemma** *type-less-x-Set* [elim!]:  
**assumes**  $\tau < \text{Set } \sigma$   
**and**  $\bigwedge \varrho. \tau = \text{Set } \varrho \implies \varrho < \sigma \implies P$   
**shows**  $P$   
 $\langle \text{proof} \rangle$

**lemma** *type-less-x-OrderedSet* [elim!]:  
**assumes**  $\tau < \text{OrderedSet } \sigma$   
**and**  $\bigwedge \varrho. \tau = \text{OrderedSet } \varrho \implies \varrho < \sigma \implies P$   
**shows**  $P$   
 $\langle \text{proof} \rangle$

**lemma** *type-less-x-Bag* [elim!]:  
**assumes**  $\tau < \text{Bag } \sigma$   
**and**  $\bigwedge \varrho. \tau = \text{Bag } \varrho \implies \varrho < \sigma \implies P$   
**shows**  $P$   
 $\langle \text{proof} \rangle$

**lemma** *type-less-x-Sequence* [elim!]:

**assumes**  $\tau < \text{Sequence } \sigma$   
**and**  $\bigwedge \varrho. \tau = \text{Sequence } \varrho \implies \varrho < \sigma \implies P$   
**shows**  $P$   
 ⟨proof⟩

We will be able to remove the acyclicity assumption only after we prove that the subtype relation is acyclic.

**lemma** *type-less-x-Tuple'*:  
**assumes**  $\tau < \text{Tuple } \xi$   
**and** *acyclicP-on (fmran'  $\xi$ ) subtype*  
**and**  $\bigwedge \pi. \tau = \text{Tuple } \pi \implies \text{strict-subtuple } (\leq) \pi \xi \implies P$   
**shows**  $P$   
 ⟨proof⟩

**lemma** *type-less-x-OclSuper [elim!]*:  
 $\tau < \text{OclSuper} \implies (\tau \neq \text{OclSuper} \implies P) \implies P$   
 ⟨proof⟩

### 3.3.3 Properties

**lemma** *subtype-irrefl*:  
 $\tau < \tau \implies \text{False}$   
**for**  $\tau :: 'a \text{ type}$   
 ⟨proof⟩

**lemma** *subtype-acyclic*:  
*acyclicP subtype*  
 ⟨proof⟩

**lemma** *less-le-not-le-type*:  
 $\tau < \sigma \iff \tau \leq \sigma \wedge \neg \sigma \leq \tau$   
**for**  $\tau \sigma :: 'a \text{ type}$   
 ⟨proof⟩

**lemma** *order-refl-type [iff]*:  
 $\tau \leq \tau$   
**for**  $\tau :: 'a \text{ type}$   
 ⟨proof⟩

**lemma** *order-trans-type*:  
 $\tau \leq \sigma \implies \sigma \leq \varrho \implies \tau \leq \varrho$   
**for**  $\tau \sigma \varrho :: 'a \text{ type}$   
 ⟨proof⟩

**lemma** *antisym-type*:  
 $\tau \leq \sigma \implies \sigma \leq \tau \implies \tau = \sigma$   
**for**  $\tau \sigma :: 'a \text{ type}$   
 ⟨proof⟩

**instance**

*<proof>*

**end**

### 3.3.4 Non-Strict Introduction Rules

**lemma** *type-less-eq-x-Required-intro* [intro]:

$\tau = \varrho[1] \implies \varrho \leq \sigma \implies \tau \leq \sigma[1]$

*<proof>*

**lemma** *type-less-eq-x-Optional-intro* [intro]:

$\tau = \varrho[1] \implies \varrho \leq \sigma \implies \tau \leq \sigma[?]$

$\tau = \varrho[?] \implies \varrho \leq \sigma \implies \tau \leq \sigma[?]$

*<proof>*

**lemma** *type-less-eq-x-Collection-intro* [intro]:

$\tau = \text{Collection } \varrho \implies \varrho \leq \sigma \implies \tau \leq \text{Collection } \sigma$

$\tau = \text{Set } \varrho \implies \varrho \leq \sigma \implies \tau \leq \text{Collection } \sigma$

$\tau = \text{OrderedSet } \varrho \implies \varrho \leq \sigma \implies \tau \leq \text{Collection } \sigma$

$\tau = \text{Bag } \varrho \implies \varrho \leq \sigma \implies \tau \leq \text{Collection } \sigma$

$\tau = \text{Sequence } \varrho \implies \varrho \leq \sigma \implies \tau \leq \text{Collection } \sigma$

*<proof>*

**lemma** *type-less-eq-x-Set-intro* [intro]:

$\tau = \text{Set } \varrho \implies \varrho \leq \sigma \implies \tau \leq \text{Set } \sigma$

*<proof>*

**lemma** *type-less-eq-x-OrderedSet-intro* [intro]:

$\tau = \text{OrderedSet } \varrho \implies \varrho \leq \sigma \implies \tau \leq \text{OrderedSet } \sigma$

*<proof>*

**lemma** *type-less-eq-x-Bag-intro* [intro]:

$\tau = \text{Bag } \varrho \implies \varrho \leq \sigma \implies \tau \leq \text{Bag } \sigma$

*<proof>*

**lemma** *type-less-eq-x-Sequence-intro* [intro]:

$\tau = \text{Sequence } \varrho \implies \varrho \leq \sigma \implies \tau \leq \text{Sequence } \sigma$

*<proof>*

**lemma** *type-less-eq-x-Tuple-intro* [intro]:

$\tau = \text{Tuple } \pi \implies \text{subtuple } (\leq) \pi \xi \implies \tau \leq \text{Tuple } \xi$

*<proof>*

**lemma** *type-less-eq-x-OclSuper-intro* [intro]:

$\tau \leq \text{OclSuper}$

*<proof>*

### 3.3.5 Non-Strict Elimination Rules

**lemma** *type-less-eq-x-Required* [elim!]:

$$\begin{aligned} & \tau \leq \sigma[1] \implies \\ & (\bigwedge \varrho. \tau = \varrho[1] \implies \varrho \leq \sigma \implies P) \implies P \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *type-less-eq-x-Optional* [elim!]:

$$\begin{aligned} & \tau \leq \sigma[?] \implies \\ & (\bigwedge \varrho. \tau = \varrho[1] \implies \varrho \leq \sigma \implies P) \implies \\ & (\bigwedge \varrho. \tau = \varrho[?] \implies \varrho \leq \sigma \implies P) \implies P \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *type-less-eq-x-Collection* [elim!]:

$$\begin{aligned} & \tau \leq \text{Collection } \sigma \implies \\ & (\bigwedge \varrho. \tau = \text{Set } \varrho \implies \varrho \leq \sigma \implies P) \implies \\ & (\bigwedge \varrho. \tau = \text{OrderedSet } \varrho \implies \varrho \leq \sigma \implies P) \implies \\ & (\bigwedge \varrho. \tau = \text{Bag } \varrho \implies \varrho \leq \sigma \implies P) \implies \\ & (\bigwedge \varrho. \tau = \text{Sequence } \varrho \implies \varrho \leq \sigma \implies P) \implies \\ & (\bigwedge \varrho. \tau = \text{Collection } \varrho \implies \varrho \leq \sigma \implies P) \implies P \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *type-less-eq-x-Set* [elim!]:

$$\begin{aligned} & \tau \leq \text{Set } \sigma \implies \\ & (\bigwedge \varrho. \tau = \text{Set } \varrho \implies \varrho \leq \sigma \implies P) \implies P \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *type-less-eq-x-OrderedSet* [elim!]:

$$\begin{aligned} & \tau \leq \text{OrderedSet } \sigma \implies \\ & (\bigwedge \varrho. \tau = \text{OrderedSet } \varrho \implies \varrho \leq \sigma \implies P) \implies P \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *type-less-eq-x-Bag* [elim!]:

$$\begin{aligned} & \tau \leq \text{Bag } \sigma \implies \\ & (\bigwedge \varrho. \tau = \text{Bag } \varrho \implies \varrho \leq \sigma \implies P) \implies P \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *type-less-eq-x-Sequence* [elim!]:

$$\begin{aligned} & \tau \leq \text{Sequence } \sigma \implies \\ & (\bigwedge \varrho. \tau = \text{Sequence } \varrho \implies \varrho \leq \sigma \implies P) \implies P \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *type-less-x-Tuple* [elim!]:

$$\begin{aligned} & \tau < \text{Tuple } \xi \implies \\ & (\bigwedge \pi. \tau = \text{Tuple } \pi \implies \text{strict-subtuple } (\leq) \pi \xi \implies P) \implies P \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *type-less-eq-x-Tuple* [elim!]:

$$\begin{aligned} & \tau \leq \text{Tuple } \xi \implies \\ & (\bigwedge \pi. \tau = \text{Tuple } \pi \implies \text{subtuple } (\leq) \pi \xi \implies P) \implies P \end{aligned}$$

*<proof>*

### 3.3.6 Simplification Rules

**lemma** *type-less-left-simps* [*simp*]:

*OclSuper* <  $\sigma$  = *False*

$\varrho[1]$  <  $\sigma$  = ( $\exists v.$

$\sigma = \textit{OclSuper} \vee$

$\sigma = v[1] \wedge \varrho < v \vee$

$\sigma = v[?] \wedge \varrho \leq v$ )

$\varrho[?] < \sigma$  = ( $\exists v.$

$\sigma = \textit{OclSuper} \vee$

$\sigma = v[?] \wedge \varrho < v$ )

*Collection*  $\tau < \sigma$  = ( $\exists \varphi.$

$\sigma = \textit{OclSuper} \vee$

$\sigma = \textit{Collection} \varphi \wedge \tau < \varphi$ )

*Set*  $\tau < \sigma$  = ( $\exists \varphi.$

$\sigma = \textit{OclSuper} \vee$

$\sigma = \textit{Collection} \varphi \wedge \tau \leq \varphi \vee$

$\sigma = \textit{Set} \varphi \wedge \tau < \varphi$ )

*OrderedSet*  $\tau < \sigma$  = ( $\exists \varphi.$

$\sigma = \textit{OclSuper} \vee$

$\sigma = \textit{Collection} \varphi \wedge \tau \leq \varphi \vee$

$\sigma = \textit{OrderedSet} \varphi \wedge \tau < \varphi$ )

*Bag*  $\tau < \sigma$  = ( $\exists \varphi.$

$\sigma = \textit{OclSuper} \vee$

$\sigma = \textit{Collection} \varphi \wedge \tau \leq \varphi \vee$

$\sigma = \textit{Bag} \varphi \wedge \tau < \varphi$ )

*Sequence*  $\tau < \sigma$  = ( $\exists \varphi.$

$\sigma = \textit{OclSuper} \vee$

$\sigma = \textit{Collection} \varphi \wedge \tau \leq \varphi \vee$

$\sigma = \textit{Sequence} \varphi \wedge \tau < \varphi$ )

*Tuple*  $\pi < \sigma$  = ( $\exists \xi.$

$\sigma = \textit{OclSuper} \vee$

$\sigma = \textit{Tuple} \xi \wedge \textit{strict-subtuple} (\leq) \pi \xi$ )

*<proof>*

**lemma** *type-less-right-simps* [*simp*]:

$\tau < \textit{OclSuper} = (\tau \neq \textit{OclSuper})$

$\tau < v[1] = (\exists \varrho. \tau = \varrho[1] \wedge \varrho < v)$

$\tau < v[?] = (\exists \varrho. \tau = \varrho[1] \wedge \varrho \leq v \vee \tau = \varrho[?] \wedge \varrho < v)$

$\tau < \textit{Collection} \sigma = (\exists \varphi.$

$\tau = \textit{Collection} \varphi \wedge \varphi < \sigma \vee$

$\tau = \textit{Set} \varphi \wedge \varphi \leq \sigma \vee$

$\tau = \textit{OrderedSet} \varphi \wedge \varphi \leq \sigma \vee$

$\tau = \textit{Bag} \varphi \wedge \varphi \leq \sigma \vee$

$\tau = \textit{Sequence} \varphi \wedge \varphi \leq \sigma$ )

$\tau < \textit{Set} \sigma = (\exists \varphi. \tau = \textit{Set} \varphi \wedge \varphi < \sigma)$

$\tau < \textit{OrderedSet} \sigma = (\exists \varphi. \tau = \textit{OrderedSet} \varphi \wedge \varphi < \sigma)$

$$\begin{aligned} \tau < Bag \ \sigma &= (\exists \varphi. \tau = Bag \ \varphi \wedge \varphi < \sigma) \\ \tau < Sequence \ \sigma &= (\exists \varphi. \tau = Sequence \ \varphi \wedge \varphi < \sigma) \\ \tau < Tuple \ \xi &= (\exists \pi. \tau = Tuple \ \pi \wedge \text{strict-subtuple} (\leq) \ \pi \ \xi) \\ &\langle \text{proof} \rangle \end{aligned}$$

### 3.4 Upper Semilattice of Types

**instantiation** *type* :: (*semilattice-sup*) *semilattice-sup*  
**begin**

**fun** *sup-type* **where**

```

  OclSuper  $\sqcup$   $\sigma$  = OclSuper
| Required  $\tau$   $\sqcup$   $\sigma$  = (case  $\sigma$ 
  of  $\varrho[1] \Rightarrow (\tau \sqcup \varrho)[1]$ 
    |  $\varrho[?] \Rightarrow (\tau \sqcup \varrho)[?]$ 
    | -  $\Rightarrow$  OclSuper)
| Optional  $\tau$   $\sqcup$   $\sigma$  = (case  $\sigma$ 
  of  $\varrho[1] \Rightarrow (\tau \sqcup \varrho)[?]$ 
    |  $\varrho[?] \Rightarrow (\tau \sqcup \varrho)[?]$ 
    | -  $\Rightarrow$  OclSuper)
| Collection  $\tau$   $\sqcup$   $\sigma$  = (case  $\sigma$ 
  of Collection  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
    | Set  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
    | OrderedSet  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
    | Bag  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
    | Sequence  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
    | -  $\Rightarrow$  OclSuper)
| Set  $\tau$   $\sqcup$   $\sigma$  = (case  $\sigma$ 
  of Collection  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
    | Set  $\varrho \Rightarrow$  Set ( $\tau \sqcup \varrho$ )
    | OrderedSet  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
    | Bag  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
    | Sequence  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
    | -  $\Rightarrow$  OclSuper)
| OrderedSet  $\tau$   $\sqcup$   $\sigma$  = (case  $\sigma$ 
  of Collection  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
    | Set  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
    | OrderedSet  $\varrho \Rightarrow$  OrderedSet ( $\tau \sqcup \varrho$ )
    | Bag  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
    | Sequence  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
    | -  $\Rightarrow$  OclSuper)
| Bag  $\tau$   $\sqcup$   $\sigma$  = (case  $\sigma$ 
  of Collection  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
    | Set  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
    | OrderedSet  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
    | Bag  $\varrho \Rightarrow$  Bag ( $\tau \sqcup \varrho$ )
    | Sequence  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
    | -  $\Rightarrow$  OclSuper)
| Sequence  $\tau$   $\sqcup$   $\sigma$  = (case  $\sigma$ 

```

```

of Collection  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
| Set  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
| OrderedSet  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
| Bag  $\varrho \Rightarrow$  Collection ( $\tau \sqcup \varrho$ )
| Sequence  $\varrho \Rightarrow$  Sequence ( $\tau \sqcup \varrho$ )
| -  $\Rightarrow$  OclSuper
| Tuple  $\pi \sqcup \sigma =$  (case  $\sigma$ 
of Tuple  $\xi \Rightarrow$  Tuple (fmerge-fun ( $\sqcup$ )  $\pi \xi$ )
| -  $\Rightarrow$  OclSuper)

```

**lemma** *sup-ge1-type*:

```

 $\tau \leq \tau \sqcup \sigma$ 
for  $\tau \sigma :: 'a$  type
<proof>

```

**lemma** *sup-commut-type*:

```

 $\tau \sqcup \sigma = \sigma \sqcup \tau$ 
for  $\tau \sigma :: 'a$  type
<proof>

```

**lemma** *sup-least-type*:

```

 $\tau \leq \varrho \Longrightarrow \sigma \leq \varrho \Longrightarrow \tau \sqcup \sigma \leq \varrho$ 
for  $\tau \sigma \varrho :: 'a$  type
<proof>

```

**instance**

```

<proof>

```

**end**

## 3.5 Helper Relations

**abbreviation** *between* ( $-/ = \text{---}$  [51, 51, 51] 50) **where**

```

 $x = y - z \equiv y \leq x \wedge x \leq z$ 

```

**inductive** *element-type* **where**

```

element-type (Collection  $\tau$ )  $\tau$ 
| element-type (Set  $\tau$ )  $\tau$ 
| element-type (OrderedSet  $\tau$ )  $\tau$ 
| element-type (Bag  $\tau$ )  $\tau$ 
| element-type (Sequence  $\tau$ )  $\tau$ 

```

**lemma** *element-type-alt-simps*:

```

element-type  $\tau \sigma =$ 
(Collection  $\sigma = \tau \vee$ 
Set  $\sigma = \tau \vee$ 
OrderedSet  $\sigma = \tau \vee$ 
Bag  $\sigma = \tau \vee$ 
Sequence  $\sigma = \tau$ )

```

*<proof>*

**inductive** *update-element-type* **where**

```

  update-element-type (Collection -)  $\tau$  (Collection  $\tau$ )
| update-element-type (Set -)  $\tau$  (Set  $\tau$ )
| update-element-type (OrderedSet -)  $\tau$  (OrderedSet  $\tau$ )
| update-element-type (Bag -)  $\tau$  (Bag  $\tau$ )
| update-element-type (Sequence -)  $\tau$  (Sequence  $\tau$ )

```

**inductive** *to-unique-collection* **where**

```

  to-unique-collection (Collection  $\tau$ ) (Set  $\tau$ )
| to-unique-collection (Set  $\tau$ ) (Set  $\tau$ )
| to-unique-collection (OrderedSet  $\tau$ ) (OrderedSet  $\tau$ )
| to-unique-collection (Bag  $\tau$ ) (Set  $\tau$ )
| to-unique-collection (Sequence  $\tau$ ) (OrderedSet  $\tau$ )

```

**inductive** *to-nonunique-collection* **where**

```

  to-nonunique-collection (Collection  $\tau$ ) (Bag  $\tau$ )
| to-nonunique-collection (Set  $\tau$ ) (Bag  $\tau$ )
| to-nonunique-collection (OrderedSet  $\tau$ ) (Sequence  $\tau$ )
| to-nonunique-collection (Bag  $\tau$ ) (Bag  $\tau$ )
| to-nonunique-collection (Sequence  $\tau$ ) (Sequence  $\tau$ )

```

**inductive** *to-ordered-collection* **where**

```

  to-ordered-collection (Collection  $\tau$ ) (Sequence  $\tau$ )
| to-ordered-collection (Set  $\tau$ ) (OrderedSet  $\tau$ )
| to-ordered-collection (OrderedSet  $\tau$ ) (OrderedSet  $\tau$ )
| to-ordered-collection (Bag  $\tau$ ) (Sequence  $\tau$ )
| to-ordered-collection (Sequence  $\tau$ ) (Sequence  $\tau$ )

```

**fun** *to-single-type* **where**

```

  to-single-type OclSuper = OclSuper
| to-single-type  $\tau[1] = \tau[1]$ 
| to-single-type  $\tau[?] = \tau[?]$ 
| to-single-type (Collection  $\tau$ ) = to-single-type  $\tau$ 
| to-single-type (Set  $\tau$ ) = to-single-type  $\tau$ 
| to-single-type (OrderedSet  $\tau$ ) = to-single-type  $\tau$ 
| to-single-type (Bag  $\tau$ ) = to-single-type  $\tau$ 
| to-single-type (Sequence  $\tau$ ) = to-single-type  $\tau$ 
| to-single-type (Tuple  $\pi$ ) = Tuple  $\pi$ 

```

**fun** *to-required-type* **where**

```

  to-required-type  $\tau[1] = \tau[1]$ 
| to-required-type  $\tau[?] = \tau[1]$ 
| to-required-type  $\tau = \tau$ 

```

**fun** *to-optional-type-nested* **where**

```

  to-optional-type-nested OclSuper = OclSuper
| to-optional-type-nested  $\tau[1] = \tau[?]$ 

```



```

| to-optional-type-nested  $\tau[?] = \tau[?]$ 
| to-optional-type-nested (Collection  $\tau$ ) = Collection (to-optional-type-nested  $\tau$ )
| to-optional-type-nested (Set  $\tau$ ) = Set (to-optional-type-nested  $\tau$ )
| to-optional-type-nested (OrderedSet  $\tau$ ) = OrderedSet (to-optional-type-nested  $\tau$ )

| to-optional-type-nested (Bag  $\tau$ ) = Bag (to-optional-type-nested  $\tau$ )
| to-optional-type-nested (Sequence  $\tau$ ) = Sequence (to-optional-type-nested  $\tau$ )
| to-optional-type-nested (Tuple  $\pi$ ) = Tuple (fmap to-optional-type-nested  $\pi$ )

```

### 3.6 Determinism

**lemma** *element-type-det*:

```

element-type  $\tau$   $\sigma_1 \implies$ 
element-type  $\tau$   $\sigma_2 \implies \sigma_1 = \sigma_2$ 
⟨proof⟩

```

**lemma** *update-element-type-det*:

```

update-element-type  $\tau$   $\sigma$   $\varrho_1 \implies$ 
update-element-type  $\tau$   $\sigma$   $\varrho_2 \implies \varrho_1 = \varrho_2$ 
⟨proof⟩

```

**lemma** *to-unique-collection-det*:

```

to-unique-collection  $\tau$   $\sigma_1 \implies$ 
to-unique-collection  $\tau$   $\sigma_2 \implies \sigma_1 = \sigma_2$ 
⟨proof⟩

```

**lemma** *to-nonunique-collection-det*:

```

to-nonunique-collection  $\tau$   $\sigma_1 \implies$ 
to-nonunique-collection  $\tau$   $\sigma_2 \implies \sigma_1 = \sigma_2$ 
⟨proof⟩

```

**lemma** *to-ordered-collection-det*:

```

to-ordered-collection  $\tau$   $\sigma_1 \implies$ 
to-ordered-collection  $\tau$   $\sigma_2 \implies \sigma_1 = \sigma_2$ 
⟨proof⟩

```

### 3.7 Code Setup

**code-pred** *subtype* ⟨proof⟩

**function** *subtype-fun* :: 'a::order type  $\Rightarrow$  'a type  $\Rightarrow$  bool **where**

```

subtype-fun OclSuper - = False
| subtype-fun (Required  $\tau$ )  $\sigma$  = (case  $\sigma$ 
of OclSuper  $\Rightarrow$  True
|  $\varrho[1]$   $\Rightarrow$  basic-subtype-fun  $\tau$   $\varrho$ 
|  $\varrho[?]$   $\Rightarrow$  basic-subtype-fun  $\tau$   $\varrho \vee \tau = \varrho$ 
| -  $\Rightarrow$  False)
| subtype-fun (Optional  $\tau$ )  $\sigma$  = (case  $\sigma$ 

```

```

of OclSuper  $\Rightarrow$  True
|  $\varrho[?]$   $\Rightarrow$  basic-subtype-fun  $\tau$   $\varrho$ 
| -  $\Rightarrow$  False)
| subtype-fun (Collection  $\tau$ )  $\sigma =$  (case  $\sigma$ 
of OclSuper  $\Rightarrow$  True
| Collection  $\varrho \Rightarrow$  subtype-fun  $\tau$   $\varrho$ 
| -  $\Rightarrow$  False)
| subtype-fun (Set  $\tau$ )  $\sigma =$  (case  $\sigma$ 
of OclSuper  $\Rightarrow$  True
| Collection  $\varrho \Rightarrow$  subtype-fun  $\tau$   $\varrho \vee \tau = \varrho$ 
| Set  $\varrho \Rightarrow$  subtype-fun  $\tau$   $\varrho$ 
| -  $\Rightarrow$  False)
| subtype-fun (OrderedSet  $\tau$ )  $\sigma =$  (case  $\sigma$ 
of OclSuper  $\Rightarrow$  True
| Collection  $\varrho \Rightarrow$  subtype-fun  $\tau$   $\varrho \vee \tau = \varrho$ 
| OrderedSet  $\varrho \Rightarrow$  subtype-fun  $\tau$   $\varrho$ 
| -  $\Rightarrow$  False)
| subtype-fun (Bag  $\tau$ )  $\sigma =$  (case  $\sigma$ 
of OclSuper  $\Rightarrow$  True
| Collection  $\varrho \Rightarrow$  subtype-fun  $\tau$   $\varrho \vee \tau = \varrho$ 
| Bag  $\varrho \Rightarrow$  subtype-fun  $\tau$   $\varrho$ 
| -  $\Rightarrow$  False)
| subtype-fun (Sequence  $\tau$ )  $\sigma =$  (case  $\sigma$ 
of OclSuper  $\Rightarrow$  True
| Collection  $\varrho \Rightarrow$  subtype-fun  $\tau$   $\varrho \vee \tau = \varrho$ 
| Sequence  $\varrho \Rightarrow$  subtype-fun  $\tau$   $\varrho$ 
| -  $\Rightarrow$  False)
| subtype-fun (Tuple  $\pi$ )  $\sigma =$  (case  $\sigma$ 
of OclSuper  $\Rightarrow$  True
| Tuple  $\xi \Rightarrow$  strict-subtuple-fun ( $\lambda\tau \sigma.$  subtype-fun  $\tau$   $\sigma \vee \tau = \sigma$ )  $\pi$   $\xi$ 
| -  $\Rightarrow$  False)
⟨proof⟩
termination
⟨proof⟩

```

**lemma** *less-type-code* [code]:

( $<$ ) = subtype-fun  
⟨proof⟩

**lemma** *less-eq-type-code* [code]:

( $\leq$ ) = ( $\lambda x y.$  subtype-fun  $x$   $y \vee x = y$ )  
⟨proof⟩

**code-pred** *element-type* ⟨proof⟩

**code-pred** *update-element-type* ⟨proof⟩

**code-pred** *to-unique-collection* ⟨proof⟩

**code-pred** *to-nonunique-collection* ⟨proof⟩

**code-pred** *to-ordered-collection* ⟨proof⟩

**end**



## Chapter 4

# Abstract Syntax

```
theory OCL-Syntax  
  imports Complex-Main Object-Model OCL-Types  
begin
```

### 4.1 Preliminaries

```
type-synonym vname = String.literal  
type-synonym 'a env = vname  $\rightarrow_f$  'a
```

In OCL  $1 + \infty = \perp$ . So we do not use *enat* and define the new data type.

```
typedef unat = UNIV :: nat option set  $\langle$ proof $\rangle$ 
```

```
definition unat x  $\equiv$  Abs-unat (Some x)
```

```
instantiation unat :: infinity
```

```
begin
```

```
definition  $\infty$   $\equiv$  Abs-unat None
```

```
instance  $\langle$ proof $\rangle$ 
```

```
end
```

```
free-constructors cases-unat for
```

```
  unat  
|  $\infty$  :: unat  
   $\langle$ proof $\rangle$ 
```

### 4.2 Standard Library Operations

```
datatype metaop = AllInstancesOp
```

```
datatype typeop = OclAsTypeOp | OclIsTypeOfOp | OclIsKindOfOp  
| SelectByKindOp | SelectByTypeOp
```

**datatype** *super-binop* = *EqualOp* | *NotEqualOp*

**datatype** *any-unop* = *OclAsSetOp* | *OclIsNewOp*  
| *OclIsUndefinedOp* | *OclIsInvalidOp* | *OclLocaleOp* | *ToStringOp*

**datatype** *boolean-unop* = *NotOp*

**datatype** *boolean-binop* = *AndOp* | *OrOp* | *XorOp* | *ImpliesOp*

**datatype** *numeric-unop* = *UMinusOp* | *AbsOp* | *FloorOp* | *RoundOp* | *ToIntegerOp*

**datatype** *numeric-binop* = *PlusOp* | *MinusOp* | *MultOp* | *DivideOp*

| *DivOp* | *ModOp* | *MaxOp* | *MinOp*

| *LessOp* | *LessEqOp* | *GreaterOp* | *GreaterEqOp*

**datatype** *string-unop* = *SizeOp* | *ToUpperCaseOp* | *ToLowerCaseOp* | *Character-*  
*sOp*

| *ToBooleanOp* | *ToIntegerOp* | *ToRealOp*

**datatype** *string-binop* = *ConcatOp* | *IndexOfOp* | *EqualsIgnoreCaseOp* | *AtOp*

| *LessOp* | *LessEqOp* | *GreaterOp* | *GreaterEqOp*

**datatype** *string-ternop* = *SubstringOp*

**datatype** *collection-unop* = *CollectionSizeOp* | *IsEmptyOp* | *NotEmptyOp*

| *CollectionMaxOp* | *CollectionMinOp* | *SumOp*

| *AsSetOp* | *AsOrderedSetOp* | *AsSequenceOp* | *AsBagOp* | *FlattenOp*

| *FirstOp* | *LastOp* | *ReverseOp*

**datatype** *collection-binop* = *IncludesOp* | *ExcludesOp*

| *CountOp* | *IncludesAllOp* | *ExcludesAllOp* | *ProductOp*

| *UnionOp* | *IntersectionOp* | *SetMinusOp* | *SymmetricDifferenceOp*

| *IncludingOp* | *ExcludingOp*

| *AppendOp* | *PrependOp* | *CollectionAtOp* | *CollectionIndexOfOp*

**datatype** *collection-ternop* = *InsertAtOp* | *SubOrderedSetOp* | *SubSequenceOp*

**type-synonym** *unop* = *any-unop* + *boolean-unop* + *numeric-unop* + *string-unop*  
+ *collection-unop*

**declare** [[*coercion Inl* :: *any-unop* ⇒ *unop*]]

**declare** [[*coercion Inr* ◦ *Inl* :: *boolean-unop* ⇒ *unop*]]

**declare** [[*coercion Inr* ◦ *Inr* ◦ *Inl* :: *numeric-unop* ⇒ *unop*]]

**declare** [[*coercion Inr* ◦ *Inr* ◦ *Inr* ◦ *Inl* :: *string-unop* ⇒ *unop*]]

**declare** [[*coercion Inr* ◦ *Inr* ◦ *Inr* ◦ *Inr* :: *collection-unop* ⇒ *unop*]]

**type-synonym** *binop* = *super-binop* + *boolean-binop* + *numeric-binop* + *string-binop*  
+ *collection-binop*

**declare** [[*coercion Inl* :: *super-binop* ⇒ *binop*]]

**declare** [[*coercion Inr* ◦ *Inl* :: *boolean-binop* ⇒ *binop*]]

**declare** [[*coercion Inr* ◦ *Inr* ◦ *Inl* :: *numeric-binop* ⇒ *binop*]]

**declare** [[*coercion Inr* ◦ *Inr* ◦ *Inr* ◦ *Inl* :: *string-binop* ⇒ *binop*]]

**declare** [[*coercion Inr* ◦ *Inr* ◦ *Inr* ◦ *Inr* :: *collection-binop* ⇒ *binop*]]

```
type-synonym ternop = string-ternop + collection-ternop
```

```
declare [[coercion Inl :: string-ternop ⇒ ternop ]]
declare [[coercion Inr :: collection-ternop ⇒ ternop ]]
```

```
type-synonym op = unop + binop + ternop + oper
```

```
declare [[coercion Inl ◦ Inl :: any-unop ⇒ op ]]
declare [[coercion Inl ◦ Inr ◦ Inl :: boolean-unop ⇒ op ]]
declare [[coercion Inl ◦ Inr ◦ Inr ◦ Inl :: numeric-unop ⇒ op ]]
declare [[coercion Inl ◦ Inr ◦ Inr ◦ Inr ◦ Inl :: string-unop ⇒ op ]]
declare [[coercion Inl ◦ Inr ◦ Inr ◦ Inr ◦ Inr :: collection-unop ⇒ op ]]
```

```
declare [[coercion Inr ◦ Inl ◦ Inl :: super-binop ⇒ op ]]
declare [[coercion Inr ◦ Inl ◦ Inr ◦ Inl :: boolean-binop ⇒ op ]]
declare [[coercion Inr ◦ Inl ◦ Inr ◦ Inr ◦ Inl :: numeric-binop ⇒ op ]]
declare [[coercion Inr ◦ Inl ◦ Inr ◦ Inr ◦ Inr ◦ Inl :: string-binop ⇒ op ]]
declare [[coercion Inr ◦ Inl ◦ Inr ◦ Inr ◦ Inr ◦ Inr :: collection-binop ⇒ op ]]
```

```
declare [[coercion Inr ◦ Inr ◦ Inl ◦ Inl :: string-ternop ⇒ op ]]
declare [[coercion Inr ◦ Inr ◦ Inl ◦ Inr :: collection-ternop ⇒ op ]]
```

```
declare [[coercion Inr ◦ Inr ◦ Inr :: oper ⇒ op ]]
```

```
datatype iterator = AnyIter | ClosureIter | CollectIter | CollectNestedIter
| ExistsIter | ForAllIter | OneIter | IsUniqueIter
| SelectIter | RejectIter | SortedByIter
```

## 4.3 Expressions

```
datatype collection-literal-kind =
  CollectionKind | SetKind | OrderedSetKind | BagKind | SequenceKind
```

A call kind could be defined as two boolean values (*is-arrow-call*, *is-safe-call*). Also we could derive *is-arrow-call* value automatically based on an operation kind. However, it is much easier and more natural to use the following enumeration.

```
datatype call-kind = DotCall | ArrowCall | SafeDotCall | SafeArrowCall
```

We do not define a *Classifier* type (a type of all types), because it will add unnecessary complications to the theory. So we have to define type operations as a pure syntactic constructs. We do not define *Type* expressions either.

We do not define *InvalidLiteral*, because it allows us to exclude *OclInvalid* type from typing rules. It simplifies the types system.

Please take a note that for *AssociationEnd* and *AssociationClass* call expressions one can specify an optional role of a source class (*from-role*). It differs from the OCL specification, which allows one to specify a role of a

destination class. However, the latter one does not allow one to determine uniquely a set of linked objects, for example, in a ternary self relation.

```

datatype 'a expr =
  Literal 'a literal-expr
| Let (var : vname) (var-type : 'a type option) (init-expr : 'a expr)
  (body-expr : 'a expr)
| Var (var : vname)
| If (if-expr : 'a expr) (then-expr : 'a expr) (else-expr : 'a expr)
| MetaOperationCall (type : 'a type) metaop
| StaticOperationCall (type : 'a type) oper (args : 'a expr list)
| Call (source : 'a expr) (kind : call-kind) 'a call-expr
and 'a literal-expr =
  NullLiteral
| BooleanLiteral (boolean-symbol : bool)
| RealLiteral (real-symbol : real)
| IntegerLiteral (integer-symbol : int)
| UnlimitedNaturalLiteral (unlimited-natural-symbol : unat)
| StringLiteral (string-symbol : string)
| EnumLiteral (enum-type : 'a enum) (enum-literal : elit)
| CollectionLiteral (kind : collection-literal-kind)
  (parts : 'a collection-literal-part-expr list)
| TupleLiteral (tuple-elements : (telem × 'a type option × 'a expr) list)
and 'a collection-literal-part-expr =
  CollectionItem (item : 'a expr)
| CollectionRange (first : 'a expr) (last : 'a expr)
and 'a call-expr =
  TypeOperation typeop (type : 'a type)
| Attribute attr
| AssociationEnd (from-role : role option) role
| AssociationClass (from-role : role option) 'a
| AssociationClassEnd role
| Operation op (args : 'a expr list)
| TupleElement telem
| Iterate (iterators : vname list) (iterators-type : 'a type option)
  (var : vname) (var-type : 'a type option) (init-expr : 'a expr)
  (body-expr : 'a expr)
| Iterator iterator (iterators : vname list) (iterators-type : 'a type option)
  (body-expr : 'a expr)

```

**definition**  $\text{tuple-element-name} \equiv \text{fst}$

**definition**  $\text{tuple-element-type} \equiv \text{fst} \circ \text{snd}$

**definition**  $\text{tuple-element-expr} \equiv \text{snd} \circ \text{snd}$

**declare**  $[[\text{coercion } \text{Literal} :: 'a \text{ literal-expr} \Rightarrow 'a \text{ expr} ]]$

**abbreviation**  $\text{TypeOperationCall } \text{src } k \text{ op } ty \equiv$

$\text{Call } \text{src } k \text{ (TypeOperation } \text{op } ty)$

**abbreviation**  $\text{AttributeCall } \text{src } k \text{ attr} \equiv$

$\text{Call } \text{src } k \text{ (Attribute } \text{attr)}$



**abbreviation** *AssociationEndCall* *src k from role*  $\equiv$   
*Call src k (AssociationEnd from role)*

**abbreviation** *AssociationClassCall* *src k from cls*  $\equiv$   
*Call src k (AssociationClass from cls)*

**abbreviation** *AssociationClassEndCall* *src k role*  $\equiv$   
*Call src k (AssociationClassEnd role)*

**abbreviation** *OperationCall* *src k op as*  $\equiv$   
*Call src k (Operation op as)*

**abbreviation** *TupleElementCall* *src k elem*  $\equiv$   
*Call src k (TupleElement elem)*

**abbreviation** *IterateCall* *src k its its-ty v ty init body*  $\equiv$   
*Call src k (Iterate its its-ty v ty init body)*

**abbreviation** *AnyIteratorCall* *src k its its-ty body*  $\equiv$   
*Call src k (Iterator AnyIter its its-ty body)*

**abbreviation** *ClosureIteratorCall* *src k its its-ty body*  $\equiv$   
*Call src k (Iterator ClosureIter its its-ty body)*

**abbreviation** *CollectIteratorCall* *src k its its-ty body*  $\equiv$   
*Call src k (Iterator CollectIter its its-ty body)*

**abbreviation** *CollectNestedIteratorCall* *src k its its-ty body*  $\equiv$   
*Call src k (Iterator CollectNestedIter its its-ty body)*

**abbreviation** *ExistsIteratorCall* *src k its its-ty body*  $\equiv$   
*Call src k (Iterator ExistsIter its its-ty body)*

**abbreviation** *ForAllIteratorCall* *src k its its-ty body*  $\equiv$   
*Call src k (Iterator ForAllIter its its-ty body)*

**abbreviation** *OneIteratorCall* *src k its its-ty body*  $\equiv$   
*Call src k (Iterator OneIter its its-ty body)*

**abbreviation** *IsUniqueIteratorCall* *src k its its-ty body*  $\equiv$   
*Call src k (Iterator IsUniqueIter its its-ty body)*

**abbreviation** *SelectIteratorCall* *src k its its-ty body*  $\equiv$   
*Call src k (Iterator SelectIter its its-ty body)*

**abbreviation** *RejectIteratorCall* *src k its its-ty body*  $\equiv$   
*Call src k (Iterator RejectIter its its-ty body)*

**abbreviation** *SortedByIteratorCall* *src k its its-ty body*  $\equiv$   
*Call src k (Iterator SortedByIter its its-ty body)*

**end**



## Chapter 5

# Object Model

```
theory OCL-Object-Model  
  imports OCL-Syntax  
begin
```

I see no reason why objects should refer nulls using multi-valued associations. Therefore, multi-valued associations have collection types with non-nullable element types.

**definition**

```
assoc-end-type end  $\equiv$   
  let  $\mathcal{C} = \text{assoc-end-class end}$  in  
  if  $\text{assoc-end-max end} \leq (1 :: \text{nat})$  then  
    if  $\text{assoc-end-min end} = (0 :: \text{nat})$   
      then  $\langle \mathcal{C} \rangle_{\mathcal{T}}[?]$   
      else  $\langle \mathcal{C} \rangle_{\mathcal{T}}[1]$   
    else  
      if  $\text{assoc-end-unique end}$  then  
        if  $\text{assoc-end-ordered end}$   
          then  $\text{OrderedSet } \langle \mathcal{C} \rangle_{\mathcal{T}}[1]$   
          else  $\text{Set } \langle \mathcal{C} \rangle_{\mathcal{T}}[1]$   
        else  
          if  $\text{assoc-end-ordered end}$   
            then  $\text{Sequence } \langle \mathcal{C} \rangle_{\mathcal{T}}[1]$   
            else  $\text{Bag } \langle \mathcal{C} \rangle_{\mathcal{T}}[1]$ 
```

**definition** *class-assoc-type*  $\mathcal{A} \equiv \text{Set } \langle \mathcal{A} \rangle_{\mathcal{T}}[1]$

**definition** *class-assoc-end-type end*  $\equiv \langle \text{assoc-end-class end} \rangle_{\mathcal{T}}[1]$

**definition** *oper-type op*  $\equiv$

```
let  $\text{params} = \text{oper-out-params op}$  in  
if  $\text{length params} = 0$   
then  $\text{oper-result op}$   
else  $\text{Tuple } (\text{fmap-of-list } (\text{map } (\lambda p. (\text{param-name } p, \text{param-type } p))$   
   $(\text{params } @ [(STR "result", \text{oper-result op}, Out)]))$ 
```

```

class ocl-object-model =
  fixes classes :: 'a :: semilattice-sup fset
  and attributes :: 'a  $\rightarrow_f$  attr  $\rightarrow_f$  'a type
  and associations :: assoc  $\rightarrow_f$  role  $\rightarrow_f$  'a assoc-end
  and association-classes :: 'a  $\rightarrow_f$  assoc
  and operations :: ('a type, 'a expr) oper-spec list
  and literals :: 'a enum  $\rightarrow_f$  elit fset
  assumes assoc-end-min-less-eq-max:
    assoc | $\in$ | fmdom associations  $\implies$ 
    fmlookup associations assoc = Some ends  $\implies$ 
    role | $\in$ | fmdom ends  $\implies$ 
    fmlookup ends role = Some end  $\implies$ 
    assoc-end-min end  $\leq$  assoc-end-max end
  assumes association-ends-unique:
    association-ends' classes associations C from role end1  $\implies$ 
    association-ends' classes associations C from role end2  $\implies$  end1 = end2
begin

interpretation base: object-model
  <proof>

abbreviation owned-attribute  $\equiv$  base.owned-attribute
abbreviation attribute  $\equiv$  base.attribute
abbreviation association-ends  $\equiv$  base.association-ends
abbreviation owned-association-end  $\equiv$  base.owned-association-end
abbreviation association-end  $\equiv$  base.association-end
abbreviation referred-by-association-class  $\equiv$  base.referred-by-association-class
abbreviation association-class-end  $\equiv$  base.association-class-end
abbreviation operation  $\equiv$  base.operation
abbreviation operation-defined  $\equiv$  base.operation-defined
abbreviation static-operation  $\equiv$  base.static-operation
abbreviation static-operation-defined  $\equiv$  base.static-operation-defined
abbreviation has-literal  $\equiv$  base.has-literal

lemmas attribute-det = base.attribute-det
lemmas attribute-self-or-inherited = base.attribute-self-or-inherited
lemmas attribute-closest = base.attribute-closest
lemmas association-end-det = base.association-end-det
lemmas association-end-self-or-inherited = base.association-end-self-or-inherited
lemmas association-end-closest = base.association-end-closest
lemmas association-class-end-det = base.association-class-end-det
lemmas operation-det = base.operation-det
lemmas static-operation-det = base.static-operation-det

end

end

```

# Chapter 6

## Typing

```
theory OCL-Typing  
  imports OCL-Object-Model HOL-Library.Transitive-Closure-Table  
begin
```

The following rules are more restrictive than rules given in the OCL specification. This allows one to identify more errors in expressions. However, these restrictions may be revised if necessary. Perhaps some of them could be separated and should cause warnings instead of errors.

### 6.1 Operations Typing

#### 6.1.1 Metaclass Operations

All basic types in the theory are either nullable or non-nullable. For example, instead of *Boolean* type we have two types: *Boolean[1]* and *Boolean[?]*. The *allInstances()* operation is extended accordingly:

```
Boolean[1].allInstances() = Set{true, false}  
Boolean[?].allInstances() = Set{true, false, null}
```

```
inductive mataop-type where  
  mataop-type  $\tau$  AllInstancesOp (Set  $\tau$ )
```

#### 6.1.2 Type Operations

At first we decided to allow casting only to subtypes. However sometimes it is necessary to cast expressions to supertypes, for example, to access overridden attributes of a supertype. So we allow casting to subtypes and supertypes. Casting to other types is meaningless.

According to the Section 7.4.7 of the OCL specification *oclAsType()* can be applied to collections as well as to single values. I guess we can allow *oclIsTypeOf()* and *oclIsKindOf()* for collections too.

Please take a note that the following expressions are prohibited, because they always return true or false:

```
1.oclIsKindOf(OclAny[?])
1.oclIsKindOf(String[1])
```

Please take a note that:

```
Set{1,2,null,'abc'}->selectByKind(Integer[1]) = Set{1,2}
Set{1,2,null,'abc'}->selectByKind(Integer[?]) = Set{1,2,null}
```

The following expressions are prohibited, because they always returns either the same or empty collections:

```
Set{1,2,null,'abc'}->selectByKind(OclAny[?])
Set{1,2,null,'abc'}->selectByKind(Collection(Boolean[1]))
```

**inductive** *typeop-type* **where**

```
 $\sigma < \tau \vee \tau < \sigma \implies$ 
typeop-type DotCall OclAsTypeOp  $\tau \sigma \sigma$ 
```

```
|  $\sigma < \tau \implies$ 
typeop-type DotCall OclIsTypeOfOp  $\tau \sigma$  Boolean[1]
|  $\sigma < \tau \implies$ 
typeop-type DotCall OclIsKindOfOp  $\tau \sigma$  Boolean[1]
```

```
| element-type  $\tau \rho \implies \sigma < \rho \implies$ 
update-element-type  $\tau \sigma v \implies$ 
typeop-type ArrowCall SelectByKindOp  $\tau \sigma v$ 
```

```
| element-type  $\tau \rho \implies \sigma < \rho \implies$ 
update-element-type  $\tau \sigma v \implies$ 
typeop-type ArrowCall SelectByTypeOp  $\tau \sigma v$ 
```

### 6.1.3 OclSuper Operations

It makes sense to compare values only with compatible types.

**inductive** *super-binop-type*

```
:: super-binop  $\implies ('a :: \text{order}) \text{ type} \implies 'a \text{ type} \implies 'a \text{ type} \implies \text{bool}$  where
 $\tau \leq \sigma \vee \sigma < \tau \implies$ 
super-binop-type EqualOp  $\tau \sigma$  Boolean[1]
|  $\tau \leq \sigma \vee \sigma < \tau \implies$ 
super-binop-type NotEqualOp  $\tau \sigma$  Boolean[1]
```

### 6.1.4 OclAny Operations

The OCL specification defines *toString()* operation only for boolean and numeric types. However, I guess it is a good idea to define it once for all basic types. Maybe it should be defined for collections as well.

**inductive** *any-unop-type* **where**

```

     $\tau \leq \text{OclAny}[\?] \implies$ 
    any-unop-type OclAsSetOp  $\tau$  (Set (to-required-type  $\tau$ ))
|  $\tau \leq \text{OclAny}[\?] \implies$ 
    any-unop-type OclIsNewOp  $\tau$  Boolean[1]
|  $\tau \leq \text{OclAny}[\?] \implies$ 
    any-unop-type OclIsUndefinedOp  $\tau$  Boolean[1]
|  $\tau \leq \text{OclAny}[\?] \implies$ 
    any-unop-type OclIsInvalidOp  $\tau$  Boolean[1]
|  $\tau \leq \text{OclAny}[\?] \implies$ 
    any-unop-type OclLocaleOp  $\tau$  String[1]
|  $\tau \leq \text{OclAny}[\?] \implies$ 
    any-unop-type ToStringOp  $\tau$  String[1]

```

### 6.1.5 Boolean Operations

Please take a note that:

```

    true or false : Boolean[1]
    true and null : Boolean[?]
    null and null : OclVoid[?]

```

**inductive** *boolean-unop-type* **where**  
 $\tau \leq \text{Boolean}[\?] \implies$   
*boolean-unop-type* *NotOp*  $\tau$   $\tau$

**inductive** *boolean-binop-type* **where**  
 $\tau \sqcup \sigma = \varrho \implies \varrho \leq \text{Boolean}[\?] \implies$   
*boolean-binop-type* *AndOp*  $\tau$   $\sigma$   $\varrho$   
|  $\tau \sqcup \sigma = \varrho \implies \varrho \leq \text{Boolean}[\?] \implies$   
*boolean-binop-type* *OrOp*  $\tau$   $\sigma$   $\varrho$   
|  $\tau \sqcup \sigma = \varrho \implies \varrho \leq \text{Boolean}[\?] \implies$   
*boolean-binop-type* *XorOp*  $\tau$   $\sigma$   $\varrho$   
|  $\tau \sqcup \sigma = \varrho \implies \varrho \leq \text{Boolean}[\?] \implies$   
*boolean-binop-type* *ImpliesOp*  $\tau$   $\sigma$   $\varrho$

### 6.1.6 Numeric Operations

The expression  $1 + \text{null}$  is not well-typed. Nullable numeric values should be converted to non-nullable ones. This is a significant difference from the OCL specification.

Please take a note that many operations automatically casts unlimited naturals to integers.

The difference between *oclAsType(Integer)* and *toInteger()* for unlimited naturals is unclear.

**inductive** *numeric-unop-type* **where**  
 $\tau = \text{Real}[1] \implies$   
*numeric-unop-type* *UMinusOp*  $\tau$  *Real*[1]  
|  $\tau = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies$

*numeric-unop-type UMinusOp*  $\tau$  *Integer*[1]

|  $\tau = \text{Real}[1] \implies$   
*numeric-unop-type AbsOp*  $\tau$  *Real*[1]  
 |  $\tau = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies$   
*numeric-unop-type AbsOp*  $\tau$  *Integer*[1]

|  $\tau = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies$   
*numeric-unop-type FloorOp*  $\tau$  *Integer*[1]  
 |  $\tau = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies$   
*numeric-unop-type RoundOp*  $\tau$  *Integer*[1]

|  $\tau = \text{UnlimitedNatural}[1] \implies$   
*numeric-unop-type numeric-unop.ToIntegerOp*  $\tau$  *Integer*[1]

**inductive** *numeric-binop-type* **where**

$\tau \sqcup \sigma = \varrho \implies \varrho = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies$   
*numeric-binop-type PlusOp*  $\tau$   $\sigma$   $\varrho$

|  $\tau \sqcup \sigma = \text{Real}[1] \implies$   
*numeric-binop-type MinusOp*  $\tau$   $\sigma$  *Real*[1]  
 |  $\tau \sqcup \sigma = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies$   
*numeric-binop-type MinusOp*  $\tau$   $\sigma$  *Integer*[1]

|  $\tau \sqcup \sigma = \varrho \implies \varrho = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies$   
*numeric-binop-type MultOp*  $\tau$   $\sigma$   $\varrho$

|  $\tau = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies \sigma = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies$   
*numeric-binop-type DivideOp*  $\tau$   $\sigma$  *Real*[1]

|  $\tau \sqcup \sigma = \varrho \implies \varrho = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies$   
*numeric-binop-type DivOp*  $\tau$   $\sigma$   $\varrho$   
 |  $\tau \sqcup \sigma = \varrho \implies \varrho = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies$   
*numeric-binop-type ModOp*  $\tau$   $\sigma$   $\varrho$

|  $\tau \sqcup \sigma = \varrho \implies \varrho = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies$   
*numeric-binop-type MaxOp*  $\tau$   $\sigma$   $\varrho$   
 |  $\tau \sqcup \sigma = \varrho \implies \varrho = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies$   
*numeric-binop-type MinOp*  $\tau$   $\sigma$   $\varrho$

|  $\tau = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies \sigma = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies$   
*numeric-binop-type numeric-binop.LessOp*  $\tau$   $\sigma$  *Boolean*[1]  
 |  $\tau = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies \sigma = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies$   
*numeric-binop-type numeric-binop.LessEqOp*  $\tau$   $\sigma$  *Boolean*[1]  
 |  $\tau = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies \sigma = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies$   
*numeric-binop-type numeric-binop.GreaterOp*  $\tau$   $\sigma$  *Boolean*[1]  
 |  $\tau = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies \sigma = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies$   
*numeric-binop-type numeric-binop.GreaterEqOp*  $\tau$   $\sigma$  *Boolean*[1]



### 6.1.7 String Operations

**inductive** *string-unop-type* **where**

*string-unop-type* *SizeOp* *String*[1] *Integer*[1]  
| *string-unop-type* *CharactersOp* *String*[1] (*Sequence* *String*[1])  
| *string-unop-type* *ToUpperCaseOp* *String*[1] *String*[1]  
| *string-unop-type* *ToLowerCaseOp* *String*[1] *String*[1]  
| *string-unop-type* *ToBooleanOp* *String*[1] *Boolean*[1]  
| *string-unop-type* *ToIntegerOp* *String*[1] *Integer*[1]  
| *string-unop-type* *ToRealOp* *String*[1] *Real*[1]

**inductive** *string-binop-type* **where**

*string-binop-type* *ConcatOp* *String*[1] *String*[1] *String*[1]  
| *string-binop-type* *EqualsIgnoreCaseOp* *String*[1] *String*[1] *Boolean*[1]  
| *string-binop-type* *LessOp* *String*[1] *String*[1] *Boolean*[1]  
| *string-binop-type* *LessEqOp* *String*[1] *String*[1] *Boolean*[1]  
| *string-binop-type* *GreaterOp* *String*[1] *String*[1] *Boolean*[1]  
| *string-binop-type* *GreaterEqOp* *String*[1] *String*[1] *Boolean*[1]  
| *string-binop-type* *IndexOfOp* *String*[1] *String*[1] *Integer*[1]  
|  $\tau = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies$   
*string-binop-type* *AtOp* *String*[1]  $\tau$  *String*[1]

**inductive** *string-ternop-type* **where**

$\sigma = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies$   
 $\varrho = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies$   
*string-ternop-type* *SubstringOp* *String*[1]  $\sigma$   $\varrho$  *String*[1]

### 6.1.8 Collection Operations

Please take a note, that *flatten()* preserves a collection kind.

**inductive** *collection-unop-type* **where**

*element-type*  $\tau - \implies$   
*collection-unop-type* *CollectionSizeOp*  $\tau$  *Integer*[1]  
| *element-type*  $\tau - \implies$   
*collection-unop-type* *IsEmptyOp*  $\tau$  *Boolean*[1]  
| *element-type*  $\tau - \implies$   
*collection-unop-type* *NotEmptyOp*  $\tau$  *Boolean*[1]  
  
| *element-type*  $\tau \sigma \implies \sigma = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies$   
*collection-unop-type* *CollectionMaxOp*  $\tau \sigma$   
| *element-type*  $\tau \sigma \implies \text{operation } \sigma \text{ STR "max" } [\sigma] \text{ oper} \implies$   
*collection-unop-type* *CollectionMaxOp*  $\tau \sigma$   
  
| *element-type*  $\tau \sigma \implies \sigma = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies$   
*collection-unop-type* *CollectionMinOp*  $\tau \sigma$   
| *element-type*  $\tau \sigma \implies \text{operation } \sigma \text{ STR "min" } [\sigma] \text{ oper} \implies$   
*collection-unop-type* *CollectionMinOp*  $\tau \sigma$   
  
| *element-type*  $\tau \sigma \implies \sigma = \text{UnlimitedNatural}[1] - \text{Real}[1] \implies$

```

    collection-unop-type SumOp  $\tau$   $\sigma$ 
| element-type  $\tau$   $\sigma \implies$  operation  $\sigma$  STR "+" [ $\sigma$ ] oper  $\implies$ 
    collection-unop-type SumOp  $\tau$   $\sigma$ 

| element-type  $\tau$   $\sigma \implies$ 
    collection-unop-type AsSetOp  $\tau$  (Set  $\sigma$ )
| element-type  $\tau$   $\sigma \implies$ 
    collection-unop-type AsOrderedSetOp  $\tau$  (OrderedSet  $\sigma$ )
| element-type  $\tau$   $\sigma \implies$ 
    collection-unop-type AsBagOp  $\tau$  (Bag  $\sigma$ )
| element-type  $\tau$   $\sigma \implies$ 
    collection-unop-type AsSequenceOp  $\tau$  (Sequence  $\sigma$ )

| update-element-type  $\tau$  (to-single-type  $\tau$ )  $\sigma \implies$ 
    collection-unop-type FlattenOp  $\tau$   $\sigma$ 

| collection-unop-type FirstOp (OrderedSet  $\tau$ )  $\tau$ 
| collection-unop-type FirstOp (Sequence  $\tau$ )  $\tau$ 
| collection-unop-type LastOp (OrderedSet  $\tau$ )  $\tau$ 
| collection-unop-type LastOp (Sequence  $\tau$ )  $\tau$ 
| collection-unop-type ReverseOp (OrderedSet  $\tau$ ) (OrderedSet  $\tau$ )
| collection-unop-type ReverseOp (Sequence  $\tau$ ) (Sequence  $\tau$ )

```

Please take a note that if both arguments are collections, then an element type of the resulting collection is a super type of element types of original collections. However for single-valued operations (*append()*, *insertAt()*, ...) this behavior looks undesirable. So we restrict such arguments to have a subtype of the collection element type.

Please take a note that we allow the following expressions:

```

let nullable_value : Integer[?] = null in
Sequence{1..3}->includes(nullable_value) and
Sequence{1..3}->includes(null) and
Sequence{1..3}->includesAll(Set{1,null})

```

The OCL specification defines *including()* and *excluding()* operations for the *Sequence* type but does not define them for the *OrderedSet* type. We define them for all collection types.

It is a good idea to prohibit including of values that do not conform to a collection element type. However we do not restrict it.

At first we defined the following typing rules for the *excluding()* operation:

```

| element-type  $\tau$   $\varrho \implies \sigma \leq \varrho \implies \sigma \neq \text{OclVoid}[?] \implies$ 
    collection-binop-type ExcludingOp  $\tau$   $\sigma$   $\tau$ 
| element-type  $\tau$   $\varrho \implies \sigma \leq \varrho \implies \sigma = \text{OclVoid}[?] \implies$ 
    update-element-type  $\tau$  (to-required-type  $\varrho$ )  $v \implies$ 
    collection-binop-type ExcludingOp  $\tau$   $\sigma$   $v$ 

```

This operation could play a special role in a definition of safe navigation operations:

`Sequence{1,2,null}->exculding(null) : Integer[1]`

However it is more natural to use a `selectByKind(T[1])` operation instead.

**inductive** *collection-binop-type* **where**

*element-type*  $\tau \varrho \implies \sigma \leq \text{to-optional-type-nested } \varrho \implies$   
*collection-binop-type* `IncludesOp`  $\tau \sigma$  `Boolean[1]`

| *element-type*  $\tau \varrho \implies \sigma \leq \text{to-optional-type-nested } \varrho \implies$   
*collection-binop-type* `ExcludesOp`  $\tau \sigma$  `Boolean[1]`

| *element-type*  $\tau \varrho \implies \sigma \leq \text{to-optional-type-nested } \varrho \implies$   
*collection-binop-type* `CountOp`  $\tau \sigma$  `Integer[1]`

| *element-type*  $\tau \varrho \implies \text{element-type } \sigma v \implies v \leq \text{to-optional-type-nested } \varrho \implies$   
*collection-binop-type* `IncludesAllOp`  $\tau \sigma$  `Boolean[1]`

| *element-type*  $\tau \varrho \implies \text{element-type } \sigma v \implies v \leq \text{to-optional-type-nested } \varrho \implies$   
*collection-binop-type* `ExcludesAllOp`  $\tau \sigma$  `Boolean[1]`

| *element-type*  $\tau \varrho \implies \text{element-type } \sigma v \implies$   
*collection-binop-type* `ProductOp`  $\tau \sigma$   
`(Set (Tuple (fmap-of-list [(STR "first",  $\varrho$ ), (STR "second",  $v$ )])))`

| *collection-binop-type* `UnionOp` `(Set  $\tau$ ) (Set  $\sigma$ ) (Set ( $\tau \sqcup \sigma$ ))`

| *collection-binop-type* `UnionOp` `(Set  $\tau$ ) (Bag  $\sigma$ ) (Bag ( $\tau \sqcup \sigma$ ))`

| *collection-binop-type* `UnionOp` `(Bag  $\tau$ ) (Set  $\sigma$ ) (Bag ( $\tau \sqcup \sigma$ ))`

| *collection-binop-type* `UnionOp` `(Bag  $\tau$ ) (Bag  $\sigma$ ) (Bag ( $\tau \sqcup \sigma$ ))`

| *collection-binop-type* `IntersectionOp` `(Set  $\tau$ ) (Set  $\sigma$ ) (Set ( $\tau \sqcap \sigma$ ))`

| *collection-binop-type* `IntersectionOp` `(Set  $\tau$ ) (Bag  $\sigma$ ) (Set ( $\tau \sqcap \sigma$ ))`

| *collection-binop-type* `IntersectionOp` `(Bag  $\tau$ ) (Set  $\sigma$ ) (Set ( $\tau \sqcap \sigma$ ))`

| *collection-binop-type* `IntersectionOp` `(Bag  $\tau$ ) (Bag  $\sigma$ ) (Bag ( $\tau \sqcap \sigma$ ))`

| *collection-binop-type* `SetMinusOp` `(Set  $\tau$ ) (Set  $\sigma$ ) (Set  $\tau$ )`

| *collection-binop-type* `SymmetricDifferenceOp` `(Set  $\tau$ ) (Set  $\sigma$ ) (Set ( $\tau \sqcup \sigma$ ))`

| *element-type*  $\tau \varrho \implies \text{update-element-type } \tau (\varrho \sqcup \sigma) v \implies$   
*collection-binop-type* `IncludingOp`  $\tau \sigma v$

| *element-type*  $\tau \varrho \implies \sigma \leq \varrho \implies$   
*collection-binop-type* `ExcludingOp`  $\tau \sigma \tau$

|  $\sigma \leq \tau \implies$   
*collection-binop-type* `AppendOp` `(OrderedSet  $\tau$ )  $\sigma$  (OrderedSet  $\tau$ )`

|  $\sigma \leq \tau \implies$   
*collection-binop-type* `AppendOp` `(Sequence  $\tau$ )  $\sigma$  (Sequence  $\tau$ )`

|  $\sigma \leq \tau \implies$   
*collection-binop-type* `PrependOp` `(OrderedSet  $\tau$ )  $\sigma$  (OrderedSet  $\tau$ )`

|  $\sigma \leq \tau \implies$   
*collection-binop-type* `PrependOp` `(Sequence  $\tau$ )  $\sigma$  (Sequence  $\tau$ )`

|  $\sigma = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies$   
*collection-binop-type* *CollectionAtOp* (*OrderedSet*  $\tau$ )  $\sigma$   $\tau$

|  $\sigma = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies$   
*collection-binop-type* *CollectionAtOp* (*Sequence*  $\tau$ )  $\sigma$   $\tau$

|  $\sigma \leq \tau \implies$   
*collection-binop-type* *CollectionIndexOfOp* (*OrderedSet*  $\tau$ )  $\sigma$  *Integer*[1]

|  $\sigma \leq \tau \implies$   
*collection-binop-type* *CollectionIndexOfOp* (*Sequence*  $\tau$ )  $\sigma$  *Integer*[1]

**inductive** *collection-ternop-type* **where**

$\sigma = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies \varrho \leq \tau \implies$   
*collection-ternop-type* *InsertAtOp* (*OrderedSet*  $\tau$ )  $\sigma$   $\varrho$  (*OrderedSet*  $\tau$ )

|  $\sigma = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies \varrho \leq \tau \implies$   
*collection-ternop-type* *InsertAtOp* (*Sequence*  $\tau$ )  $\sigma$   $\varrho$  (*Sequence*  $\tau$ )

|  $\sigma = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies$   
 $\varrho = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies$   
*collection-ternop-type* *SubOrderedSetOp* (*OrderedSet*  $\tau$ )  $\sigma$   $\varrho$  (*OrderedSet*  $\tau$ )

|  $\sigma = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies$   
 $\varrho = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies$   
*collection-ternop-type* *SubSequenceOp* (*Sequence*  $\tau$ )  $\sigma$   $\varrho$  (*Sequence*  $\tau$ )

### 6.1.9 Coercions

**inductive** *unop-type* **where**

*any-unop-type* *op*  $\tau$   $\sigma \implies$   
*unop-type* (*Inl op*) *DotCall*  $\tau$   $\sigma$

| *boolean-unop-type* *op*  $\tau$   $\sigma \implies$   
*unop-type* (*Inr (Inl op)*) *DotCall*  $\tau$   $\sigma$

| *numeric-unop-type* *op*  $\tau$   $\sigma \implies$   
*unop-type* (*Inr (Inr (Inl op))*) *DotCall*  $\tau$   $\sigma$

| *string-unop-type* *op*  $\tau$   $\sigma \implies$   
*unop-type* (*Inr (Inr (Inr (Inl op)))*) *DotCall*  $\tau$   $\sigma$

| *collection-unop-type* *op*  $\tau$   $\sigma \implies$   
*unop-type* (*Inr (Inr (Inr (Inr op)))*) *ArrowCall*  $\tau$   $\sigma$

**inductive** *binop-type* **where**

*super-binop-type* *op*  $\tau$   $\sigma$   $\varrho \implies$   
*binop-type* (*Inl op*) *DotCall*  $\tau$   $\sigma$   $\varrho$

| *boolean-binop-type* *op*  $\tau$   $\sigma$   $\varrho \implies$   
*binop-type* (*Inr (Inl op)*) *DotCall*  $\tau$   $\sigma$   $\varrho$

| *numeric-binop-type* *op*  $\tau$   $\sigma$   $\varrho \implies$   
*binop-type* (*Inr (Inr (Inl op))*) *DotCall*  $\tau$   $\sigma$   $\varrho$

| *string-binop-type* *op*  $\tau$   $\sigma$   $\varrho \implies$   
*binop-type* (*Inr (Inr (Inr (Inl op)))*) *DotCall*  $\tau$   $\sigma$   $\varrho$

| *collection-binop-type* *op*  $\tau$   $\sigma$   $\varrho \implies$   
*binop-type* (*Inr (Inr (Inr (Inr op)))*) *ArrowCall*  $\tau$   $\sigma$   $\varrho$

**inductive ternop-type where**

*string-ternop-type*  $op \tau \sigma \varrho v \implies$   
*ternop-type* (*Inl op*) *DotCall*  $\tau \sigma \varrho v$   
| *collection-ternop-type*  $op \tau \sigma \varrho v \implies$   
*ternop-type* (*Inr op*) *ArrowCall*  $\tau \sigma \varrho v$

**inductive op-type where**

*unop-type*  $op k \tau v \implies$   
*op-type* (*Inl op*)  $k \tau [] v$   
| *binop-type*  $op k \tau \sigma v \implies$   
*op-type* (*Inr (Inl op)*)  $k \tau [\sigma] v$   
| *ternop-type*  $op k \tau \sigma \varrho v \implies$   
*op-type* (*Inr (Inr (Inl op))*)  $k \tau [\sigma, \varrho] v$   
| *operation*  $\tau op \pi oper \implies$   
*op-type* (*Inr (Inr (Inr op))*) *DotCall*  $\tau \pi (oper\text{-}type\ oper)$

### 6.1.10 Simplification Rules

**inductive-simps** *op-type-alt-simps*:

*mataop-type*  $\tau op \sigma$   
*typeop-type*  $k op \tau \sigma \varrho$

*op-type*  $op k \tau \pi \sigma$   
*unop-type*  $op k \tau \sigma$   
*binop-type*  $op k \tau \sigma \varrho$   
*ternop-type*  $op k \tau \sigma \varrho v$

*any-unop-type*  $op \tau \sigma$   
*boolean-unop-type*  $op \tau \sigma$   
*numeric-unop-type*  $op \tau \sigma$   
*string-unop-type*  $op \tau \sigma$   
*collection-unop-type*  $op \tau \sigma$

*super-binop-type*  $op \tau \sigma \varrho$   
*boolean-binop-type*  $op \tau \sigma \varrho$   
*numeric-binop-type*  $op \tau \sigma \varrho$   
*string-binop-type*  $op \tau \sigma \varrho$   
*collection-binop-type*  $op \tau \sigma \varrho$

*string-ternop-type*  $op \tau \sigma \varrho v$   
*collection-ternop-type*  $op \tau \sigma \varrho v$

### 6.1.11 Determinism

**lemma** *typeop-type-det*:

*typeop-type*  $op k \tau \sigma \varrho_1 \implies$   
*typeop-type*  $op k \tau \sigma \varrho_2 \implies \varrho_1 = \varrho_2$   
⟨*proof*⟩

**lemma** *any-unop-type-det*:

$any-unop-type\ op\ \tau\ \sigma_1 \implies$   
 $any-unop-type\ op\ \tau\ \sigma_2 \implies \sigma_1 = \sigma_2$   
 ⟨proof⟩

**lemma** *boolean-unop-type-det:*

$boolean-unop-type\ op\ \tau\ \sigma_1 \implies$   
 $boolean-unop-type\ op\ \tau\ \sigma_2 \implies \sigma_1 = \sigma_2$   
 ⟨proof⟩

**lemma** *numeric-unop-type-det:*

$numeric-unop-type\ op\ \tau\ \sigma_1 \implies$   
 $numeric-unop-type\ op\ \tau\ \sigma_2 \implies \sigma_1 = \sigma_2$   
 ⟨proof⟩

**lemma** *string-unop-type-det:*

$string-unop-type\ op\ \tau\ \sigma_1 \implies$   
 $string-unop-type\ op\ \tau\ \sigma_2 \implies \sigma_1 = \sigma_2$   
 ⟨proof⟩

**lemma** *collection-unop-type-det:*

$collection-unop-type\ op\ \tau\ \sigma_1 \implies$   
 $collection-unop-type\ op\ \tau\ \sigma_2 \implies \sigma_1 = \sigma_2$   
 ⟨proof⟩

**lemma** *unop-type-det:*

$unop-type\ op\ k\ \tau\ \sigma_1 \implies$   
 $unop-type\ op\ k\ \tau\ \sigma_2 \implies \sigma_1 = \sigma_2$   
 ⟨proof⟩

**lemma** *super-binop-type-det:*

$super-binop-type\ op\ \tau\ \sigma\ \varrho_1 \implies$   
 $super-binop-type\ op\ \tau\ \sigma\ \varrho_2 \implies \varrho_1 = \varrho_2$   
 ⟨proof⟩

**lemma** *boolean-binop-type-det:*

$boolean-binop-type\ op\ \tau\ \sigma\ \varrho_1 \implies$   
 $boolean-binop-type\ op\ \tau\ \sigma\ \varrho_2 \implies \varrho_1 = \varrho_2$   
 ⟨proof⟩

**lemma** *numeric-binop-type-det:*

$numeric-binop-type\ op\ \tau\ \sigma\ \varrho_1 \implies$   
 $numeric-binop-type\ op\ \tau\ \sigma\ \varrho_2 \implies \varrho_1 = \varrho_2$   
 ⟨proof⟩

**lemma** *string-binop-type-det:*

$string-binop-type\ op\ \tau\ \sigma\ \varrho_1 \implies$   
 $string-binop-type\ op\ \tau\ \sigma\ \varrho_2 \implies \varrho_1 = \varrho_2$   
 ⟨proof⟩

**lemma** *collection-binop-type-det*:

*collection-binop-type op*  $\tau$   $\sigma$   $\varrho_1 \implies$   
*collection-binop-type op*  $\tau$   $\sigma$   $\varrho_2 \implies \varrho_1 = \varrho_2$   
 ⟨*proof*⟩

**lemma** *binop-type-det*:

*binop-type op*  $k$   $\tau$   $\sigma$   $\varrho_1 \implies$   
*binop-type op*  $k$   $\tau$   $\sigma$   $\varrho_2 \implies \varrho_1 = \varrho_2$   
 ⟨*proof*⟩

**lemma** *string-ternop-type-det*:

*string-ternop-type op*  $\tau$   $\sigma$   $\varrho$   $v_1 \implies$   
*string-ternop-type op*  $\tau$   $\sigma$   $\varrho$   $v_2 \implies v_1 = v_2$   
 ⟨*proof*⟩

**lemma** *collection-ternop-type-det*:

*collection-ternop-type op*  $\tau$   $\sigma$   $\varrho$   $v_1 \implies$   
*collection-ternop-type op*  $\tau$   $\sigma$   $\varrho$   $v_2 \implies v_1 = v_2$   
 ⟨*proof*⟩

**lemma** *ternop-type-det*:

*ternop-type op*  $k$   $\tau$   $\sigma$   $\varrho$   $v_1 \implies$   
*ternop-type op*  $k$   $\tau$   $\sigma$   $\varrho$   $v_2 \implies v_1 = v_2$   
 ⟨*proof*⟩

**lemma** *op-type-det*:

*op-type op*  $k$   $\tau$   $\pi$   $\sigma \implies$   
*op-type op*  $k$   $\tau$   $\pi$   $\varrho \implies \sigma = \varrho$   
 ⟨*proof*⟩

## 6.2 Expressions Typing

The following typing rules are preliminary. The final rules are given at the end of the next chapter.

**inductive** *typing* :: ('a :: ocl-object-model) type env  $\Rightarrow$  'a expr  $\Rightarrow$  'a type  $\Rightarrow$  bool  
 ( (1-/  $\vdash_E$ / (- :/ -)) [51,51,51] 50)  
**and** *collection-parts-typing* ( (1-/  $\vdash_C$ / (- :/ -)) [51,51,51] 50)  
**and** *collection-part-typing* ( (1-/  $\vdash_P$ / (- :/ -)) [51,51,51] 50)  
**and** *iterator-typing* ( (1-/  $\vdash_I$ / (- :/ -)) [51,51,51] 50)  
**and** *expr-list-typing* ( (1-/  $\vdash_L$ / (- :/ -)) [51,51,51] 50) **where**

— Primitive Literals

*NullLiteralT*:

$\Gamma \vdash_E$  *NullLiteral* : *OclVoid*[?]

|*BooleanLiteralT*:

$\Gamma \vdash_E$  *BooleanLiteral*  $c$  : *Boolean*[1]

|*RealLiteralT*:

$$\Gamma \vdash_E \text{RealLiteral } c : \text{Real}[1]$$

|*IntegerLiteralT*:

$$\Gamma \vdash_E \text{IntegerLiteral } c : \text{Integer}[1]$$

|*UnlimitedNaturalLiteralT*:

$$\Gamma \vdash_E \text{UnlimitedNaturalLiteral } c : \text{UnlimitedNatural}[1]$$

|*StringLiteralT*:

$$\Gamma \vdash_E \text{StringLiteral } c : \text{String}[1]$$

|*EnumLiteralT*:

$$\text{has-literal enum lit} \implies$$

$$\Gamma \vdash_E \text{EnumLiteral } \text{enum lit} : (\text{Enum } \text{enum})[1]$$

— Collection Literals

|*SetLiteralT*:

$$\Gamma \vdash_C \text{prts} : \tau \implies$$

$$\Gamma \vdash_E \text{CollectionLiteral } \text{SetKind } \text{prts} : \text{Set } \tau$$

|*OrderedSetLiteralT*:

$$\Gamma \vdash_C \text{prts} : \tau \implies$$

$$\Gamma \vdash_E \text{CollectionLiteral } \text{OrderedSetKind } \text{prts} : \text{OrderedSet } \tau$$

|*BagLiteralT*:

$$\Gamma \vdash_C \text{prts} : \tau \implies$$

$$\Gamma \vdash_E \text{CollectionLiteral } \text{BagKind } \text{prts} : \text{Bag } \tau$$

|*SequenceLiteralT*:

$$\Gamma \vdash_C \text{prts} : \tau \implies$$

$$\Gamma \vdash_E \text{CollectionLiteral } \text{SequenceKind } \text{prts} : \text{Sequence } \tau$$

— We prohibit empty collection literals, because their type is unclear. We could use *OclVoid*[1] element type for empty collections, but the typing rules will give wrong types for nested collections, because, for example,  $\text{OclVoid}[1] \sqcup \text{Set}(\text{Integer}[1]) = \text{OclSuper}$

|*CollectionPartsSingletonT*:

$$\Gamma \vdash_P x : \tau \implies$$

$$\Gamma \vdash_C [x] : \tau$$

|*CollectionPartsListT*:

$$\Gamma \vdash_P x : \tau \implies$$

$$\Gamma \vdash_C y \# xs : \sigma \implies$$

$$\Gamma \vdash_C x \# y \# xs : \tau \sqcup \sigma$$

|*CollectionPartItemT*:

$$\Gamma \vdash_E a : \tau \implies$$

$$\Gamma \vdash_P \text{CollectionItem } a : \tau$$

|*CollectionPartRangeT*:

$$\Gamma \vdash_E a : \tau \implies$$

$$\Gamma \vdash_E b : \sigma \implies$$

$$\tau = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies$$

$$\sigma = \text{UnlimitedNatural}[1] - \text{Integer}[1] \implies$$

$$\Gamma \vdash_P \text{CollectionRange } a \ b : \text{Integer}[1]$$



— Tuple Literals

— We do not prohibit empty tuples, because it could be useful.  $Tuple()$  is a supertype of all other tuple types.

|*TupleLiteralNilT*:

$$\Gamma \vdash_E \text{TupleLiteral } [] : \text{Tuple } \text{fmempty}$$

|*TupleLiteralConsT*:

$$\Gamma \vdash_E \text{TupleLiteral } \text{elems} : \text{Tuple } \xi \implies$$

$$\Gamma \vdash_E \text{tuple-element-expr } \text{el} : \tau \implies$$

$$\text{tuple-element-type } \text{el} = \text{Some } \sigma \implies$$

$$\tau \leq \sigma \implies$$

$$\Gamma \vdash_E \text{TupleLiteral } (\text{el } \# \text{elems}) : \text{Tuple } (\xi(\text{tuple-element-name } \text{el} \mapsto_f \sigma))$$

— Misc Expressions

|*LetT*:

$$\Gamma \vdash_E \text{init} : \sigma \implies$$

$$\sigma \leq \tau \implies$$

$$\Gamma(v \mapsto_f \tau) \vdash_E \text{body} : \varrho \implies$$

$$\Gamma \vdash_E \text{Let } v \text{ (Some } \tau) \text{ init body} : \varrho$$

|*VarT*:

$$\text{fmlookup } \Gamma \text{ } v = \text{Some } \tau \implies$$

$$\Gamma \vdash_E \text{Var } v : \tau$$

|*IfT*:

$$\Gamma \vdash_E a : \text{Boolean}[1] \implies$$

$$\Gamma \vdash_E b : \sigma \implies$$

$$\Gamma \vdash_E c : \varrho \implies$$

$$\Gamma \vdash_E \text{If } a \text{ } b \text{ } c : \sigma \sqcup \varrho$$

— Call Expressions

|*MetaOperationCallT*:

$$\text{metaop-type } \tau \text{ op } \sigma \implies$$

$$\Gamma \vdash_E \text{MetaOperationCall } \tau \text{ op} : \sigma$$

|*StaticOperationCallT*:

$$\Gamma \vdash_L \text{params} : \pi \implies$$

$$\text{static-operation } \tau \text{ op } \pi \text{ oper} \implies$$

$$\Gamma \vdash_E \text{StaticOperationCall } \tau \text{ op } \text{params} : \text{oper-type } \text{oper}$$

|*TypeOperationCallT*:

$$\Gamma \vdash_E a : \tau \implies$$

$$\text{typeop-type } k \text{ op } \tau \text{ } \sigma \text{ } \varrho \implies$$

$$\Gamma \vdash_E \text{TypeOperationCall } a \text{ } k \text{ op } \sigma : \varrho$$

|*AttributeCallT*:

$$\Gamma \vdash_E \text{src} : \langle C \rangle_{\mathcal{T}}[1] \implies$$

$$\text{attribute } C \text{ prop } \mathcal{D} \text{ } \tau \implies$$

$$\Gamma \vdash_E \text{AttributeCall } \text{src } \text{DotCall } \text{prop} : \tau$$

|*AssociationEndCallT*:

$$\begin{array}{l}
\Gamma \vdash_E \text{src} : \langle C \rangle_{\mathcal{T}}[1] \Longrightarrow \\
\text{association-end } C \text{ from role } \mathcal{D} \text{ end} \Longrightarrow \\
\Gamma \vdash_E \text{AssociationEndCall src DotCall from role} : \text{assoc-end-type end} \\
| \text{AssociationClassCallT:} \\
\Gamma \vdash_E \text{src} : \langle C \rangle_{\mathcal{T}}[1] \Longrightarrow \\
\text{referred-by-association-class } C \text{ from } \mathcal{A} \mathcal{D} \Longrightarrow \\
\Gamma \vdash_E \text{AssociationClassCall src DotCall from } \mathcal{A} : \text{class-assoc-type } \mathcal{A} \\
| \text{AssociationClassEndCallT:} \\
\Gamma \vdash_E \text{src} : \langle \mathcal{A} \rangle_{\mathcal{T}}[1] \Longrightarrow \\
\text{association-class-end } \mathcal{A} \text{ role end} \Longrightarrow \\
\Gamma \vdash_E \text{AssociationClassEndCall src DotCall role} : \text{class-assoc-end-type end} \\
| \text{OperationCallT:} \\
\Gamma \vdash_E \text{src} : \tau \Longrightarrow \\
\Gamma \vdash_L \text{params} : \pi \Longrightarrow \\
\text{op-type } op \ k \ \tau \ \pi \ \sigma \Longrightarrow \\
\Gamma \vdash_E \text{OperationCall src k op params} : \sigma \\
| \text{TupleElementCallT:} \\
\Gamma \vdash_E \text{src} : \text{Tuple } \pi \Longrightarrow \\
\text{fmlookup } \pi \ \text{elem} = \text{Some } \tau \Longrightarrow \\
\Gamma \vdash_E \text{TupleElementCall src DotCall elem} : \tau
\end{array}$$

— Iterator Expressions

$$\begin{array}{l}
| \text{IteratorT:} \\
\Gamma \vdash_E \text{src} : \tau \Longrightarrow \\
\text{element-type } \tau \ \sigma \Longrightarrow \\
\sigma \leq \text{its-ty} \Longrightarrow \\
\Gamma \text{ ++}_f \text{ fmap-of-list (map } (\lambda it. (it, \text{its-ty})) \text{ its)} \vdash_E \text{body} : \varrho \Longrightarrow \\
\Gamma \vdash_I (\text{src}, \text{its}, (\text{Some } \text{its-ty}), \text{body}) : (\tau, \sigma, \varrho) \\
| \text{IterateT:} \\
\Gamma \vdash_I (\text{src}, \text{its}, \text{its-ty}, \text{Let res (Some res-t) res-init body}) : (\tau, \sigma, \varrho) \Longrightarrow \\
\varrho \leq \text{res-t} \Longrightarrow \\
\Gamma \vdash_E \text{IterateCall src ArrowCall its its-ty res (Some res-t) res-init body} : \varrho \\
| \text{AnyIteratorT:} \\
\Gamma \vdash_I (\text{src}, \text{its}, \text{its-ty}, \text{body}) : (\tau, \sigma, \varrho) \Longrightarrow \\
\text{length its} \leq 1 \Longrightarrow \\
\varrho \leq \text{Boolean}[?] \Longrightarrow \\
\Gamma \vdash_E \text{AnyIteratorCall src ArrowCall its its-ty body} : \sigma \\
| \text{ClosureIteratorT:} \\
\Gamma \vdash_I (\text{src}, \text{its}, \text{its-ty}, \text{body}) : (\tau, \sigma, \varrho) \Longrightarrow \\
\text{length its} \leq 1 \Longrightarrow \\
\text{to-single-type } \varrho \leq \sigma \Longrightarrow \\
\text{to-unique-collection } \tau \ v \Longrightarrow \\
\Gamma \vdash_E \text{ClosureIteratorCall src ArrowCall its its-ty body} : v \\
| \text{CollectIteratorT:} \\
\Gamma \vdash_I (\text{src}, \text{its}, \text{its-ty}, \text{body}) : (\tau, \sigma, \varrho) \Longrightarrow
\end{array}$$

$$\begin{aligned}
& \text{length } its \leq 1 \implies \\
& \text{to-nonunique-collection } \tau \ v \implies \\
& \text{update-element-type } v \ (\text{to-single-type } \varrho) \ \varphi \implies \\
& \Gamma \vdash_E \text{CollectIteratorCall } src \ \text{ArrowCall } its \ its\text{-ty } body : \varphi \\
| \text{CollectNestedIteratorT:} \\
& \Gamma \vdash_I (src, its, its\text{-ty}, body) : (\tau, \sigma, \varrho) \implies \\
& \text{length } its \leq 1 \implies \\
& \text{to-nonunique-collection } \tau \ v \implies \\
& \text{update-element-type } v \ \varrho \ \varphi \implies \\
& \Gamma \vdash_E \text{CollectNestedIteratorCall } src \ \text{ArrowCall } its \ its\text{-ty } body : \varphi \\
| \text{ExistsIteratorT:} \\
& \Gamma \vdash_I (src, its, its\text{-ty}, body) : (\tau, \sigma, \varrho) \implies \\
& \varrho \leq \text{Boolean}[\varphi] \implies \\
& \Gamma \vdash_E \text{ExistsIteratorCall } src \ \text{ArrowCall } its \ its\text{-ty } body : \varrho \\
| \text{ForAllIteratorT:} \\
& \Gamma \vdash_I (src, its, its\text{-ty}, body) : (\tau, \sigma, \varrho) \implies \\
& \varrho \leq \text{Boolean}[\varphi] \implies \\
& \Gamma \vdash_E \text{ForAllIteratorCall } src \ \text{ArrowCall } its \ its\text{-ty } body : \varrho \\
| \text{OneIteratorT:} \\
& \Gamma \vdash_I (src, its, its\text{-ty}, body) : (\tau, \sigma, \varrho) \implies \\
& \text{length } its \leq 1 \implies \\
& \varrho \leq \text{Boolean}[\varphi] \implies \\
& \Gamma \vdash_E \text{OneIteratorCall } src \ \text{ArrowCall } its \ its\text{-ty } body : \text{Boolean}[1] \\
| \text{IsUniqueIteratorT:} \\
& \Gamma \vdash_I (src, its, its\text{-ty}, body) : (\tau, \sigma, \varrho) \implies \\
& \text{length } its \leq 1 \implies \\
& \Gamma \vdash_E \text{IsUniqueIteratorCall } src \ \text{ArrowCall } its \ its\text{-ty } body : \text{Boolean}[1] \\
| \text{SelectIteratorT:} \\
& \Gamma \vdash_I (src, its, its\text{-ty}, body) : (\tau, \sigma, \varrho) \implies \\
& \text{length } its \leq 1 \implies \\
& \varrho \leq \text{Boolean}[\varphi] \implies \\
& \Gamma \vdash_E \text{SelectIteratorCall } src \ \text{ArrowCall } its \ its\text{-ty } body : \tau \\
| \text{RejectIteratorT:} \\
& \Gamma \vdash_I (src, its, its\text{-ty}, body) : (\tau, \sigma, \varrho) \implies \\
& \text{length } its \leq 1 \implies \\
& \varrho \leq \text{Boolean}[\varphi] \implies \\
& \Gamma \vdash_E \text{RejectIteratorCall } src \ \text{ArrowCall } its \ its\text{-ty } body : \tau \\
| \text{SortedByIteratorT:} \\
& \Gamma \vdash_I (src, its, its\text{-ty}, body) : (\tau, \sigma, \varrho) \implies \\
& \text{length } its \leq 1 \implies \\
& \text{to-ordered-collection } \tau \ v \implies \\
& \Gamma \vdash_E \text{SortedByIteratorCall } src \ \text{ArrowCall } its \ its\text{-ty } body : v
\end{aligned}$$

— Expression Lists

$$\begin{aligned}
| \text{ExprListNilT:} \\
& \Gamma \vdash_L [] : [] \\
| \text{ExprListConsT:} \\
& \Gamma \vdash_E \text{expr} : \tau \implies
\end{aligned}$$

$$\begin{aligned} \Gamma \vdash_L \text{exprs} : \pi &\implies \\ \Gamma \vdash_L \text{expr} \# \text{exprs} : \tau \# \pi \end{aligned}$$

### 6.3 Elimination Rules

**inductive-cases** *NullLiteralTE* [elim]:  $\Gamma \vdash_E \text{NullLiteral} : \tau$   
**inductive-cases** *BooleanLiteralTE* [elim]:  $\Gamma \vdash_E \text{BooleanLiteral } c : \tau$   
**inductive-cases** *RealLiteralTE* [elim]:  $\Gamma \vdash_E \text{RealLiteral } c : \tau$   
**inductive-cases** *IntegerLiteralTE* [elim]:  $\Gamma \vdash_E \text{IntegerLiteral } c : \tau$   
**inductive-cases** *UnlimitedNaturalLiteralTE* [elim]:  $\Gamma \vdash_E \text{UnlimitedNaturalLiteral } c : \tau$   
**inductive-cases** *StringLiteralTE* [elim]:  $\Gamma \vdash_E \text{StringLiteral } c : \tau$   
**inductive-cases** *EnumLiteralTE* [elim]:  $\Gamma \vdash_E \text{EnumLiteral } \text{enm } \text{lit} : \tau$   
**inductive-cases** *CollectionLiteralTE* [elim]:  $\Gamma \vdash_E \text{CollectionLiteral } k \text{ prts} : \tau$   
**inductive-cases** *TupleLiteralTE* [elim]:  $\Gamma \vdash_E \text{TupleLiteral } \text{elems} : \tau$

**inductive-cases** *LetTE* [elim]:  $\Gamma \vdash_E \text{Let } v \tau \text{ init } \text{body} : \sigma$   
**inductive-cases** *VarTE* [elim]:  $\Gamma \vdash_E \text{Var } v : \tau$   
**inductive-cases** *IfTE* [elim]:  $\Gamma \vdash_E \text{If } a \ b \ c : \tau$

**inductive-cases** *MetaOperationCallTE* [elim]:  $\Gamma \vdash_E \text{MetaOperationCall } \tau \text{ op} : \sigma$

**inductive-cases** *StaticOperationCallTE* [elim]:  $\Gamma \vdash_E \text{StaticOperationCall } \tau \text{ op } \text{as} : \sigma$

**inductive-cases** *TypeOperationCallTE* [elim]:  $\Gamma \vdash_E \text{TypeOperationCall } a \ k \ \text{op } \sigma : \tau$   
**inductive-cases** *AttributeCallTE* [elim]:  $\Gamma \vdash_E \text{AttributeCall } \text{src } k \ \text{prop} : \tau$   
**inductive-cases** *AssociationEndCallTE* [elim]:  $\Gamma \vdash_E \text{AssociationEndCall } \text{src } k \ \text{role } \text{from} : \tau$   
**inductive-cases** *AssociationClassCallTE* [elim]:  $\Gamma \vdash_E \text{AssociationClassCall } \text{src } k \ a \ \text{from} : \tau$   
**inductive-cases** *AssociationClassEndCallTE* [elim]:  $\Gamma \vdash_E \text{AssociationClassEndCall } \text{src } k \ \text{role} : \tau$   
**inductive-cases** *OperationCallTE* [elim]:  $\Gamma \vdash_E \text{OperationCall } \text{src } k \ \text{op } \text{params} : \tau$   
**inductive-cases** *TupleElementCallTE* [elim]:  $\Gamma \vdash_E \text{TupleElementCall } \text{src } k \ \text{elem} : \tau$

**inductive-cases** *IteratorTE* [elim]:  $\Gamma \vdash_I (\text{src}, \text{its}, \text{body}) : \text{ys}$   
**inductive-cases** *IterateTE* [elim]:  $\Gamma \vdash_E \text{IterateCall } \text{src } k \ \text{its } \text{its-ty } \text{res } \text{res-t } \text{res-init } \text{body} : \tau$   
**inductive-cases** *AnyIteratorTE* [elim]:  $\Gamma \vdash_E \text{AnyIteratorCall } \text{src } k \ \text{its } \text{its-ty } \text{body} : \tau$   
**inductive-cases** *ClosureIteratorTE* [elim]:  $\Gamma \vdash_E \text{ClosureIteratorCall } \text{src } k \ \text{its } \text{its-ty } \text{body} : \tau$   
**inductive-cases** *CollectIteratorTE* [elim]:  $\Gamma \vdash_E \text{CollectIteratorCall } \text{src } k \ \text{its } \text{its-ty } \text{body} : \tau$   
**inductive-cases** *CollectNestedIteratorTE* [elim]:  $\Gamma \vdash_E \text{CollectNestedIteratorCall}$

$src\ k\ its\ its\text{-}ty\ body : \tau$

**inductive-cases** *ExistsIteratorTE* [elim]:  $\Gamma \vdash_E \text{ExistsIteratorCall } src\ k\ its\ its\text{-}ty\ body : \tau$

**inductive-cases** *ForAllIteratorTE* [elim]:  $\Gamma \vdash_E \text{ForAllIteratorCall } src\ k\ its\ its\text{-}ty\ body : \tau$

**inductive-cases** *OneIteratorTE* [elim]:  $\Gamma \vdash_E \text{OneIteratorCall } src\ k\ its\ its\text{-}ty\ body : \tau$

**inductive-cases** *IsUniqueIteratorTE* [elim]:  $\Gamma \vdash_E \text{IsUniqueIteratorCall } src\ k\ its\ its\text{-}ty\ body : \tau$

**inductive-cases** *SelectIteratorTE* [elim]:  $\Gamma \vdash_E \text{SelectIteratorCall } src\ k\ its\ its\text{-}ty\ body : \tau$

**inductive-cases** *RejectIteratorTE* [elim]:  $\Gamma \vdash_E \text{RejectIteratorCall } src\ k\ its\ its\text{-}ty\ body : \tau$

**inductive-cases** *SortedByIteratorTE* [elim]:  $\Gamma \vdash_E \text{SortedByIteratorCall } src\ k\ its\ its\text{-}ty\ body : \tau$

**inductive-cases** *CollectionPartsNilTE* [elim]:  $\Gamma \vdash_C [x] : \tau$

**inductive-cases** *CollectionPartsItemTE* [elim]:  $\Gamma \vdash_C x \# y \# xs : \tau$

**inductive-cases** *CollectionItemTE* [elim]:  $\Gamma \vdash_P \text{CollectionItem } a : \tau$

**inductive-cases** *CollectionRangeTE* [elim]:  $\Gamma \vdash_P \text{CollectionRange } a\ b : \tau$

**inductive-cases** *ExprListTE* [elim]:  $\Gamma \vdash_L \text{exprs} : \pi$

## 6.4 Simplification Rules

**inductive-simps** *typing-alt-simps*:

$\Gamma \vdash_E \text{NullLiteral} : \tau$

$\Gamma \vdash_E \text{BooleanLiteral } c : \tau$

$\Gamma \vdash_E \text{RealLiteral } c : \tau$

$\Gamma \vdash_E \text{UnlimitedNaturalLiteral } c : \tau$

$\Gamma \vdash_E \text{IntegerLiteral } c : \tau$

$\Gamma \vdash_E \text{StringLiteral } c : \tau$

$\Gamma \vdash_E \text{EnumLiteral } enm\ lit : \tau$

$\Gamma \vdash_E \text{CollectionLiteral } k\ prts : \tau$

$\Gamma \vdash_E \text{TupleLiteral } [] : \tau$

$\Gamma \vdash_E \text{TupleLiteral } (x \# xs) : \tau$

$\Gamma \vdash_E \text{Let } v\ \tau\ \text{init } body : \sigma$

$\Gamma \vdash_E \text{Var } v : \tau$

$\Gamma \vdash_E \text{If } a\ b\ c : \tau$

$\Gamma \vdash_E \text{MetaOperationCall } \tau\ op : \sigma$

$\Gamma \vdash_E \text{StaticOperationCall } \tau\ op\ as : \sigma$

$\Gamma \vdash_E \text{TypeOperationCall } a\ k\ op\ \sigma : \tau$

$\Gamma \vdash_E \text{AttributeCall } src\ k\ prop : \tau$

$\Gamma \vdash_E \text{AssociationEndCall } src\ k\ role\ from : \tau$

$\Gamma \vdash_E \text{AssociationClassCall } src\ k\ a\ from : \tau$

$$\begin{aligned}
&\Gamma \vdash_E \text{AssociationClassEndCall } src \ k \ role : \tau \\
&\Gamma \vdash_E \text{OperationCall } src \ k \ op \ params : \tau \\
&\Gamma \vdash_E \text{TupleElementCall } src \ k \ elem : \tau \\
\\
&\Gamma \vdash_I (src, its, body) : ys \\
&\Gamma \vdash_E \text{IterateCall } src \ k \ its \ its-ty \ res \ res-t \ res-init \ body : \tau \\
&\Gamma \vdash_E \text{AnyIteratorCall } src \ k \ its \ its-ty \ body : \tau \\
&\Gamma \vdash_E \text{ClosureIteratorCall } src \ k \ its \ its-ty \ body : \tau \\
&\Gamma \vdash_E \text{CollectIteratorCall } src \ k \ its \ its-ty \ body : \tau \\
&\Gamma \vdash_E \text{CollectNestedIteratorCall } src \ k \ its \ its-ty \ body : \tau \\
&\Gamma \vdash_E \text{ExistsIteratorCall } src \ k \ its \ its-ty \ body : \tau \\
&\Gamma \vdash_E \text{ForAllIteratorCall } src \ k \ its \ its-ty \ body : \tau \\
&\Gamma \vdash_E \text{OneIteratorCall } src \ k \ its \ its-ty \ body : \tau \\
&\Gamma \vdash_E \text{IsUniqueIteratorCall } src \ k \ its \ its-ty \ body : \tau \\
&\Gamma \vdash_E \text{SelectIteratorCall } src \ k \ its \ its-ty \ body : \tau \\
&\Gamma \vdash_E \text{RejectIteratorCall } src \ k \ its \ its-ty \ body : \tau \\
&\Gamma \vdash_E \text{SortedByIteratorCall } src \ k \ its \ its-ty \ body : \tau \\
\\
&\Gamma \vdash_C [x] : \tau \\
&\Gamma \vdash_C x \# y \# xs : \tau \\
\\
&\Gamma \vdash_P \text{CollectionItem } a : \tau \\
&\Gamma \vdash_P \text{CollectionRange } a \ b : \tau \\
\\
&\Gamma \vdash_L [] : \pi \\
&\Gamma \vdash_L x \# xs : \pi
\end{aligned}$$

## 6.5 Determinism

**lemma**

*typing-det:*

$$\begin{aligned}
&\Gamma \vdash_E \text{expr} : \tau \implies \\
&\Gamma \vdash_E \text{expr} : \sigma \implies \tau = \sigma \quad \mathbf{and}
\end{aligned}$$

*collection-parts-typing-det:*

$$\begin{aligned}
&\Gamma \vdash_C \text{prts} : \tau \implies \\
&\Gamma \vdash_C \text{prts} : \sigma \implies \tau = \sigma \quad \mathbf{and}
\end{aligned}$$

*collection-part-typing-det:*

$$\begin{aligned}
&\Gamma \vdash_P \text{prt} : \tau \implies \\
&\Gamma \vdash_P \text{prt} : \sigma \implies \tau = \sigma \quad \mathbf{and}
\end{aligned}$$

*iterator-typing-det:*

$$\begin{aligned}
&\Gamma \vdash_I (src, its, body) : xs \implies \\
&\Gamma \vdash_I (src, its, body) : ys \implies xs = ys \quad \mathbf{and}
\end{aligned}$$

*expr-list-typing-det:*

$$\begin{aligned}
&\Gamma \vdash_L \text{exprs} : \pi \implies \\
&\Gamma \vdash_L \text{exprs} : \xi \implies \pi = \xi
\end{aligned}$$

*<proof>*

## 6.6 Code Setup

```
code-pred op-type ⟨proof⟩
```

```
code-pred (modes:  
  i ⇒ i ⇒ i ⇒ bool,  
  i ⇒ i ⇒ o ⇒ bool) iterator-typing ⟨proof⟩
```

```
end
```





# Chapter 7

## Normalization

```
theory OCL-Normalization
  imports OCL-Typing
begin
```

### 7.1 Normalization Rules

The following expression normalization rules includes two kinds of an abstract syntax tree transformations:

- determination of implicit types of variables, iterators, and tuple elements,
- unfolding of navigation shorthands and safe navigation operators, described in [Table 7.1](#).

The following variables are used in the table:

- **x** is a non-nullable value,
- **n** is a nullable value,
- **xs** is a collection of non-nullable values,
- **ns** is a collection of nullable values.

Please take a note that name resolution of variables, types, attributes, and associations is out of scope of this section. It should be done on a previous phase during transformation of a concrete syntax tree to an abstract syntax tree.

```
fun string-of-nat :: nat ⇒ string where
  string-of-nat n = (if n < 10 then [char-of (48 + n)]
    else string-of-nat (n div 10) @ [char-of (48 + (n mod 10))])
```

**definition** new-vname ≡ String.implode ◦ string-of-nat ◦ fcard ◦ fmdom

Table 7.1: Expression Normalization Rules

Orig. expr.	Normalized expression
<code>x.op()</code>	<code>x.op()</code>
<code>n.op()</code>	<code>n.op()</code> *
<code>x?.op()</code>	—
<code>n?.op()</code>	<code>if n &lt;&gt; null then n.oclAsType(T[1]).op() else null endif</code> **
<code>x-&gt;op()</code>	<code>x.oclAsSet()-&gt;op()</code>
<code>n-&gt;op()</code>	<code>n.oclAsSet()-&gt;op()</code>
<code>x?-&gt;op()</code>	—
<code>n?-&gt;op()</code>	—
<code>xs.op()</code>	<code>xs-&gt;collect(x   x.op())</code>
<code>ns.op()</code>	<code>ns-&gt;collect(n   n.op())</code> *
<code>xs?.op()</code>	—
<code>ns?.op()</code>	<code>ns-&gt;selectByKind(T[1])-&gt;collect(x   x.op())</code>
<code>xs-&gt;op()</code>	<code>xs-&gt;op()</code>
<code>ns-&gt;op()</code>	<code>ns-&gt;op()</code>
<code>xs?-&gt;op()</code>	—
<code>ns?-&gt;op()</code>	<code>ns-&gt;selectByKind(T[1])-&gt;op()</code>

\* The resulting expression will be ill-typed if the operation is unsafe. An unsafe operation is an operation which is well-typed for a non-nullable source only.

\*\* It would be a good idea to prohibit such a transformation for safe operations. A safe operation is an operation which is well-typed for a nullable source. However, it is hard to define safe operations formally considering operations overloading, complex relations between operation parameters types (please see the typing rules for the equality operator), and user-defined operations.

#### inductive *normalize*

$:: ('a :: \text{ocl-object-model}) \text{ type env} \Rightarrow 'a \text{ expr} \Rightarrow 'a \text{ expr} \Rightarrow \text{bool}$

$(- \vdash - \Rightarrow / - [51,51,51] 50)$  **and**

*normalize-call*  $(- \vdash_C - \Rightarrow / - [51,51,51] 50)$  **and**

*normalize-expr-list*  $(- \vdash_L - \Rightarrow / - [51,51,51] 50)$

**where**

*LiteralN*:

$\Gamma \vdash \text{Literal } a \Rightarrow \text{Literal } a$

|*ExplicitlyTypedLetN*:

$\Gamma \vdash \text{init}_1 \Rightarrow \text{init}_2 \Longrightarrow$

$\Gamma(v \mapsto_f \tau) \vdash \text{body}_1 \Rightarrow \text{body}_2 \Longrightarrow$

$\Gamma \vdash \text{Let } v \text{ (Some } \tau) \text{ init}_1 \text{ body}_1 \Rightarrow \text{Let } v \text{ (Some } \tau) \text{ init}_2 \text{ body}_2$

|*ImplicitlyTypedLetN*:

$\Gamma \vdash \text{init}_1 \Rightarrow \text{init}_2 \Longrightarrow$

$\Gamma \vdash_E \text{init}_2 : \tau \Longrightarrow$

$\Gamma(v \mapsto_f \tau) \vdash \text{body}_1 \Rightarrow \text{body}_2 \Longrightarrow$

$\Gamma \vdash \text{Let } v \text{ None } \text{init}_1 \text{ body}_1 \Rightarrow \text{Let } v \text{ (Some } \tau) \text{ init}_2 \text{ body}_2$   
|VarN:  
 $\Gamma \vdash \text{Var } v \Rightarrow \text{Var } v$   
|IfN:  
 $\Gamma \vdash a_1 \Rightarrow a_2 \Rightarrow$   
 $\Gamma \vdash b_1 \Rightarrow b_2 \Rightarrow$   
 $\Gamma \vdash c_1 \Rightarrow c_2 \Rightarrow$   
 $\Gamma \vdash \text{If } a_1 \text{ } b_1 \text{ } c_1 \Rightarrow \text{If } a_2 \text{ } b_2 \text{ } c_2$   
|MetaOperationCallN:  
 $\Gamma \vdash \text{MetaOperationCall } \tau \text{ op} \Rightarrow \text{MetaOperationCall } \tau \text{ op}$   
|StaticOperationCallN:  
 $\Gamma \vdash_L \text{params}_1 \Rightarrow \text{params}_2 \Rightarrow$   
 $\Gamma \vdash \text{StaticOperationCall } \tau \text{ op } \text{params}_1 \Rightarrow \text{StaticOperationCall } \tau \text{ op } \text{params}_2$   
|OclAnyDotCallN:  
 $\Gamma \vdash \text{src}_1 \Rightarrow \text{src}_2 \Rightarrow$   
 $\Gamma \vdash_E \text{src}_2 : \tau \Rightarrow$   
 $\tau \leq \text{OclAny}[\?] \vee \tau \leq \text{Tuple } \text{fmempty} \Rightarrow$   
 $(\Gamma, \tau) \vdash_C \text{call}_1 \Rightarrow \text{call}_2 \Rightarrow$   
 $\Gamma \vdash \text{Call } \text{src}_1 \text{ DotCall } \text{call}_1 \Rightarrow \text{Call } \text{src}_2 \text{ DotCall } \text{call}_2$   
|OclAnySafeDotCallN:  
 $\Gamma \vdash \text{src}_1 \Rightarrow \text{src}_2 \Rightarrow$   
 $\Gamma \vdash_E \text{src}_2 : \tau \Rightarrow$   
 $\text{OclVoid}[\?] \leq \tau \Rightarrow$   
 $(\Gamma, \text{to-required-type } \tau) \vdash_C \text{call}_1 \Rightarrow \text{call}_2 \Rightarrow$   
 $\text{src}_3 = \text{TypeOperationCall } \text{src}_2 \text{ DotCall } \text{OclAsTypeOp } (\text{to-required-type } \tau) \Rightarrow$   
 $\Gamma \vdash \text{Call } \text{src}_1 \text{ SafeDotCall } \text{call}_1 \Rightarrow$   
 $\text{If } (\text{OperationCall } \text{src}_2 \text{ DotCall } \text{NotEqualOp } [\text{NullLiteral}])$   
 $\text{(Call } \text{src}_3 \text{ DotCall } \text{call}_2)$   
 $\text{NullLiteral}$   
|OclAnyArrowCallN:  
 $\Gamma \vdash \text{src}_1 \Rightarrow \text{src}_2 \Rightarrow$   
 $\Gamma \vdash_E \text{src}_2 : \tau \Rightarrow$   
 $\tau \leq \text{OclAny}[\?] \vee \tau \leq \text{Tuple } \text{fmempty} \Rightarrow$   
 $\text{src}_3 = \text{OperationCall } \text{src}_2 \text{ DotCall } \text{OclAsSetOp } [] \Rightarrow$   
 $\Gamma \vdash_E \text{src}_3 : \sigma \Rightarrow$   
 $(\Gamma, \sigma) \vdash_C \text{call}_1 \Rightarrow \text{call}_2 \Rightarrow$   
 $\Gamma \vdash \text{Call } \text{src}_1 \text{ ArrowCall } \text{call}_1 \Rightarrow \text{Call } \text{src}_3 \text{ ArrowCall } \text{call}_2$   
|CollectionArrowCallN:  
 $\Gamma \vdash \text{src}_1 \Rightarrow \text{src}_2 \Rightarrow$   
 $\Gamma \vdash_E \text{src}_2 : \tau \Rightarrow$   
 $\text{element-type } \tau - \Rightarrow$   
 $(\Gamma, \tau) \vdash_C \text{call}_1 \Rightarrow \text{call}_2 \Rightarrow$   
 $\Gamma \vdash \text{Call } \text{src}_1 \text{ ArrowCall } \text{call}_1 \Rightarrow \text{Call } \text{src}_2 \text{ ArrowCall } \text{call}_2$   
|CollectionSafeArrowCallN:  
 $\Gamma \vdash \text{src}_1 \Rightarrow \text{src}_2 \Rightarrow$   
 $\Gamma \vdash_E \text{src}_2 : \tau \Rightarrow$

$element\text{-}type\ \tau\ \sigma \implies$   
 $OclVoid[?] \leq \sigma \implies$   
 $src_3 = TypeOperationCall\ src_2\ ArrowCall\ SelectByKindOp$   
 $(to\text{-}required\text{-}type\ \sigma) \implies$   
 $\Gamma \vdash_E\ src_3 : \varrho \implies$   
 $(\Gamma, \varrho) \vdash_C\ call_1 \ni call_2 \implies$   
 $\Gamma \vdash\ Call\ src_1\ SafeArrowCall\ call_1 \ni Call\ src_3\ ArrowCall\ call_2$

| *CollectionDotCallN*:

$\Gamma \vdash\ src_1 \ni src_2 \implies$   
 $\Gamma \vdash_E\ src_2 : \tau \implies$   
 $element\text{-}type\ \tau\ \sigma \implies$   
 $(\Gamma, \sigma) \vdash_C\ call_1 \ni call_2 \implies$   
 $it = new\text{-}vname\ \Gamma \implies$   
 $\Gamma \vdash\ Call\ src_1\ DotCall\ call_1 \ni$   
 $CollectIteratorCall\ src_2\ ArrowCall\ [it]\ (Some\ \sigma)\ (Call\ (Var\ it)\ DotCall\ call_2)$

| *CollectionSafeDotCallN*:

$\Gamma \vdash\ src_1 \ni src_2 \implies$   
 $\Gamma \vdash_E\ src_2 : \tau \implies$   
 $element\text{-}type\ \tau\ \sigma \implies$   
 $OclVoid[?] \leq \sigma \implies$   
 $\varrho = to\text{-}required\text{-}type\ \sigma \implies$   
 $src_3 = TypeOperationCall\ src_2\ ArrowCall\ SelectByKindOp\ \varrho \implies$   
 $(\Gamma, \varrho) \vdash_C\ call_1 \ni call_2 \implies$   
 $it = new\text{-}vname\ \Gamma \implies$   
 $\Gamma \vdash\ Call\ src_1\ SafeDotCall\ call_1 \ni$   
 $CollectIteratorCall\ src_3\ ArrowCall\ [it]\ (Some\ \varrho)\ (Call\ (Var\ it)\ DotCall\ call_2)$

| *TypeOperationN*:

$(\Gamma, \tau) \vdash_C\ TypeOperation\ op\ ty \ni TypeOperation\ op\ ty$

| *AttributeN*:

$(\Gamma, \tau) \vdash_C\ Attribute\ attr \ni Attribute\ attr$

| *AssociationEndN*:

$(\Gamma, \tau) \vdash_C\ AssociationEnd\ role\ from \ni AssociationEnd\ role\ from$

| *AssociationClassN*:

$(\Gamma, \tau) \vdash_C\ AssociationClass\ A\ from \ni AssociationClass\ A\ from$

| *AssociationClassEndN*:

$(\Gamma, \tau) \vdash_C\ AssociationClassEnd\ role \ni AssociationClassEnd\ role$

| *OperationN*:

$\Gamma \vdash_L\ params_1 \ni params_2 \implies$   
 $(\Gamma, \tau) \vdash_C\ Operation\ op\ params_1 \ni Operation\ op\ params_2$

| *TupleElementN*:

$(\Gamma, \tau) \vdash_C\ TupleElement\ elem \ni TupleElement\ elem$

| *ExplicitlyTypedIterateN*:

$\Gamma \vdash\ res\text{-}init_1 \ni res\text{-}init_2 \implies$   
 $\Gamma\ ++_f\ fmap\text{-}of\text{-}list\ (map\ (\lambda it.\ (it,\ \sigma))\ its) \vdash$   
 $Let\ res\ res\text{-}t_1\ res\text{-}init_1\ body_1 \ni Let\ res\ res\text{-}t_2\ res\text{-}init_2\ body_2 \implies$   
 $(\Gamma, \tau) \vdash_C\ Iterate\ its\ (Some\ \sigma)\ res\ res\text{-}t_1\ res\text{-}init_1\ body_1 \ni$   
 $Iterate\ its\ (Some\ \sigma)\ res\ res\text{-}t_2\ res\text{-}init_2\ body_2$

|*ImplicitlyTypedIterateN*:

*element-type*  $\tau \sigma \implies$   
 $\Gamma \vdash \text{res-init}_1 \ni \text{res-init}_2 \implies$   
 $\Gamma \text{ ++}_f \text{ fmap-of-list } (\text{map } (\lambda it. (it, \sigma)) \text{ its}) \vdash$   
 $\text{Let res res-t}_1 \text{ res-init}_1 \text{ body}_1 \ni \text{Let res res-t}_2 \text{ res-init}_2 \text{ body}_2 \implies$   
 $(\Gamma, \tau) \vdash_C \text{Iterate its None res res-t}_1 \text{ res-init}_1 \text{ body}_1 \ni$   
 $\text{Iterate its (Some } \sigma) \text{ res res-t}_2 \text{ res-init}_2 \text{ body}_2$

|*ExplicitlyTypedIteratorN*:

$\Gamma \text{ ++}_f \text{ fmap-of-list } (\text{map } (\lambda it. (it, \sigma)) \text{ its}) \vdash \text{body}_1 \ni \text{body}_2 \implies$   
 $(\Gamma, \tau) \vdash_C \text{Iterator iter its (Some } \sigma) \text{ body}_1 \ni \text{Iterator iter its (Some } \sigma) \text{ body}_2$

|*ImplicitlyTypedIteratorN*:

*element-type*  $\tau \sigma \implies$   
 $\Gamma \text{ ++}_f \text{ fmap-of-list } (\text{map } (\lambda it. (it, \sigma)) \text{ its}) \vdash \text{body}_1 \ni \text{body}_2 \implies$   
 $(\Gamma, \tau) \vdash_C \text{Iterator iter its None body}_1 \ni \text{Iterator iter its (Some } \sigma) \text{ body}_2$

|*ExprListNilN*:

$\Gamma \vdash_L [] \ni []$

|*ExprListConsN*:

$\Gamma \vdash x \ni y \implies$

$\Gamma \vdash_L xs \ni ys \implies$

$\Gamma \vdash_L x \# xs \ni y \# ys$

## 7.2 Elimination Rules

**inductive-cases** *LiteralNE* [elim]:  $\Gamma \vdash \text{Literal } a \ni b$

**inductive-cases** *LetNE* [elim]:  $\Gamma \vdash \text{Let } v \text{ t init body} \ni b$

**inductive-cases** *VarNE* [elim]:  $\Gamma \vdash \text{Var } v \ni b$

**inductive-cases** *IfNE* [elim]:  $\Gamma \vdash \text{If } a \text{ b c} \ni d$

**inductive-cases** *MetaOperationCallNE* [elim]:  $\Gamma \vdash \text{MetaOperationCall } \tau \text{ op} \ni b$

**inductive-cases** *StaticOperationCallNE* [elim]:  $\Gamma \vdash \text{StaticOperationCall } \tau \text{ op as} \ni b$

**inductive-cases** *DotCallNE* [elim]:  $\Gamma \vdash \text{Call src DotCall call} \ni b$

**inductive-cases** *SafeDotCallNE* [elim]:  $\Gamma \vdash \text{Call src SafeDotCall call} \ni b$

**inductive-cases** *ArrowCallNE* [elim]:  $\Gamma \vdash \text{Call src ArrowCall call} \ni b$

**inductive-cases** *SafeArrowCallNE* [elim]:  $\Gamma \vdash \text{Call src SafeArrowCall call} \ni b$

**inductive-cases** *CallNE* [elim]:  $(\Gamma, \tau) \vdash_C \text{call} \ni b$

**inductive-cases** *OperationCallNE* [elim]:  $(\Gamma, \tau) \vdash_C \text{Operation op as} \ni \text{call}$

**inductive-cases** *IterateCallNE* [elim]:  $(\Gamma, \tau) \vdash_C \text{Iterate its its-ty res res-t res-init body} \ni \text{call}$

**inductive-cases** *IteratorCallNE* [elim]:  $(\Gamma, \tau) \vdash_C \text{Iterator iter its its-ty body} \ni \text{call}$

**inductive-cases** *ExprListNE* [elim]:  $\Gamma \vdash_L xs \ni ys$

### 7.3 Simplification Rules

**inductive-simps** *normalize-alt-simps*:

$$\Gamma \vdash \text{Literal } a \Rightarrow b$$

$$\Gamma \vdash \text{Let } v \ t \ \text{init } \text{body} \Rightarrow b$$

$$\Gamma \vdash \text{Var } v \Rightarrow b$$

$$\Gamma \vdash \text{If } a \ b \ c \Rightarrow d$$

$$\Gamma \vdash \text{MetaOperationCall } \tau \ \text{op} \Rightarrow b$$

$$\Gamma \vdash \text{StaticOperationCall } \tau \ \text{op } \text{as} \Rightarrow b$$

$$\Gamma \vdash \text{Call } \text{src } \text{DotCall } \text{call} \Rightarrow b$$

$$\Gamma \vdash \text{Call } \text{src } \text{SafeDotCall } \text{call} \Rightarrow b$$

$$\Gamma \vdash \text{Call } \text{src } \text{ArrowCall } \text{call} \Rightarrow b$$

$$\Gamma \vdash \text{Call } \text{src } \text{SafeArrowCall } \text{call} \Rightarrow b$$

$$(\Gamma, \tau) \vdash_C \text{call} \Rightarrow b$$

$$(\Gamma, \tau) \vdash_C \text{Operation } \text{op } \text{as} \Rightarrow \text{call}$$

$$(\Gamma, \tau) \vdash_C \text{Iterate } \text{its } \text{its-ty } \text{res } \text{res-t } \text{res-init } \text{body} \Rightarrow \text{call}$$

$$(\Gamma, \tau) \vdash_C \text{Iterator } \text{iter } \text{its } \text{its-ty } \text{body} \Rightarrow \text{call}$$

$$\Gamma \vdash_L [] \Rightarrow \text{ys}$$

$$\Gamma \vdash_L x \ # \ \text{xs} \Rightarrow \text{ys}$$

### 7.4 Determinism

**lemma** *any-has-not-element-type*:

$$\text{element-type } \tau \ \sigma \Longrightarrow \tau \leq \text{OclAny}[?] \vee \tau \leq \text{Tuple } \text{fmempty} \Longrightarrow \text{False}$$

*<proof>*

**lemma** *any-has-not-element-type'*:

$$\text{element-type } \tau \ \sigma \Longrightarrow \text{OclVoid}[?] \leq \tau \Longrightarrow \text{False}$$

*<proof>*

**lemma**

*normalize-det*:

$$\Gamma \vdash \text{expr} \Rightarrow \text{expr}_1 \Longrightarrow$$

$$\Gamma \vdash \text{expr} \Rightarrow \text{expr}_2 \Longrightarrow \text{expr}_1 = \text{expr}_2 \ \text{and}$$

*normalize-call-det*:

$$\Gamma\text{-}\tau \vdash_C \text{call} \Rightarrow \text{call}_1 \Longrightarrow$$

$$\Gamma\text{-}\tau \vdash_C \text{call} \Rightarrow \text{call}_2 \Longrightarrow \text{call}_1 = \text{call}_2 \ \text{and}$$

*normalize-expr-list-det*:

$$\Gamma \vdash_L \text{xs} \Rightarrow \text{ys} \Longrightarrow$$

$$\Gamma \vdash_L \text{xs} \Rightarrow \text{zs} \Longrightarrow \text{ys} = \text{zs}$$

**for**  $\Gamma :: ('a :: \text{ocl-object-model}) \text{type env}$

**and**  $\Gamma\text{-}\tau :: ('a :: \text{ocl-object-model}) \text{type env} \times 'a \text{type}$

*<proof>*

## 7.5 Normalized Expressions Typing

Here is the final typing rules.

**inductive** *nf-typing* ( (1-/  $\vdash$ / (- :/ -)) [51,51,51] 50) **where**  
 $\Gamma \vdash expr \Rightarrow expr_N \Rightarrow$   
 $\Gamma \vdash_E expr_N : \tau \Rightarrow$   
 $\Gamma \vdash expr : \tau$

**lemma** *nf-typing-det*:  
 $\Gamma \vdash expr : \tau \Rightarrow$   
 $\Gamma \vdash expr : \sigma \Rightarrow \tau = \sigma$   
*<proof>*

## 7.6 Code Setup

**code-pred** *normalize* *<proof>*

**code-pred** *nf-typing* *<proof>*

**definition** *check-type*  $\Gamma expr \tau \equiv$   
*Predicate.eval* (*nf-typing-i-i-i*  $\Gamma expr \tau$ ) ()

**definition** *synthesize-type*  $\Gamma expr \equiv$   
*Predicate.singleton* ( $\lambda-. OclInvalid$ )  
(*Predicate.map errorable* (*nf-typing-i-i-o*  $\Gamma expr$ ))

It is the only usage of the *OclInvalid* type. This type is not required to define typing rules. It is only required to make the typing function total.

**end**





# Chapter 8

## Examples

```
theory OCL-Examples
  imports OCL-Normalization
begin
```

### 8.1 Classes

```
datatype classes1 =
  Object | Person | Employee | Customer | Project | Task | Sprint
```

```
inductive subclass1 where
  c ≠ Object ⇒
  subclass1 c Object
| subclass1 Employee Person
| subclass1 Customer Person
```

```
instantiation classes1 :: semilattice-sup
begin
```

```
definition (<) ≡ subclass1
```

```
definition (≤) ≡ subclass1==
```

```
fun sup-classes1 where
  Object ⊔ - = Object
| Person ⊔ c = (if c = Person ∨ c = Employee ∨ c = Customer
  then Person else Object)
| Employee ⊔ c = (if c = Employee then Employee else
  if c = Person ∨ c = Customer then Person else Object)
| Customer ⊔ c = (if c = Customer then Customer else
  if c = Person ∨ c = Employee then Person else Object)
| Project ⊔ c = (if c = Project then Project else Object)
| Task ⊔ c = (if c = Task then Task else Object)
| Sprint ⊔ c = (if c = Sprint then Sprint else Object)
```

```
lemma less-le-not-le-classes1:
```

```

   $c < d \iff c \leq d \wedge \neg d \leq c$ 
for  $c\ d :: \text{classes1}$ 
   $\langle \text{proof} \rangle$ 

```

**lemma** *order-refl-classes1*:

```

   $c \leq c$ 
for  $c :: \text{classes1}$ 
   $\langle \text{proof} \rangle$ 

```

**lemma** *order-trans-classes1*:

```

   $c \leq d \implies d \leq e \implies c \leq e$ 
for  $c\ d\ e :: \text{classes1}$ 
   $\langle \text{proof} \rangle$ 

```

**lemma** *antisym-classes1*:

```

   $c \leq d \implies d \leq c \implies c = d$ 
for  $c\ d :: \text{classes1}$ 
   $\langle \text{proof} \rangle$ 

```

**lemma** *sup-ge1-classes1*:

```

   $c \leq c \sqcup d$ 
for  $c\ d :: \text{classes1}$ 
   $\langle \text{proof} \rangle$ 

```

**lemma** *sup-ge2-classes1*:

```

   $d \leq c \sqcup d$ 
for  $c\ d :: \text{classes1}$ 
   $\langle \text{proof} \rangle$ 

```

**lemma** *sup-least-classes1*:

```

   $c \leq e \implies d \leq e \implies c \sqcup d \leq e$ 
for  $c\ d\ e :: \text{classes1}$ 
   $\langle \text{proof} \rangle$ 

```

**instance**

```

   $\langle \text{proof} \rangle$ 

```

**end**

**code-pred** *subclass1*  $\langle \text{proof} \rangle$

**fun** *subclass1-fun* **where**

```

  subclass1-fun Object  $\mathcal{C} = \text{False}$ 
| subclass1-fun Person  $\mathcal{C} = (\mathcal{C} = \text{Object})$ 
| subclass1-fun Employee  $\mathcal{C} = (\mathcal{C} = \text{Object} \vee \mathcal{C} = \text{Person})$ 
| subclass1-fun Customer  $\mathcal{C} = (\mathcal{C} = \text{Object} \vee \mathcal{C} = \text{Person})$ 
| subclass1-fun Project  $\mathcal{C} = (\mathcal{C} = \text{Object})$ 
| subclass1-fun Task  $\mathcal{C} = (\mathcal{C} = \text{Object})$ 
| subclass1-fun Sprint  $\mathcal{C} = (\mathcal{C} = \text{Object})$ 

```

**lemma** *less-classes1-code* [*code*]:

(*<*) = *subclass1-fun*  
 ⟨*proof*⟩

**lemma** *less-eq-classes1-code* [*code*]:

(*≤*) = ( $\lambda x y. \text{subclass1-fun } x y \vee x = y$ )  
 ⟨*proof*⟩

## 8.2 Object Model

**abbreviation**  $\Gamma_0 \equiv \text{fmempty} :: \text{classes1 type env}$

**declare** [[*coercion* *ObjectType* :: *classes1*  $\Rightarrow$  *classes1 basic-type* ]]

**declare** [[*coercion* *phantom* :: *String.literal*  $\Rightarrow$  *classes1 enum* ]]

**instantiation** *classes1* :: *ocl-object-model*

**begin**

**definition** *classes-classes1*  $\equiv$

{|*Object*, *Person*, *Employee*, *Customer*, *Project*, *Task*, *Sprint*|}

**definition** *attributes-classes1*  $\equiv \text{fmap-of-list}$  [

(*Person*, *fmap-of-list* [  
 (*STR* "*name*", *String*[1] :: *classes1 type*)]),  
 (*Employee*, *fmap-of-list* [  
 (*STR* "*name*", *String*[1]),  
 (*STR* "*position*", *String*[1])]),  
 (*Customer*, *fmap-of-list* [  
 (*STR* "*vip*", *Boolean*[1])]),  
 (*Project*, *fmap-of-list* [  
 (*STR* "*name*", *String*[1]),  
 (*STR* "*cost*", *Real*[?])]),  
 (*Task*, *fmap-of-list* [  
 (*STR* "*description*", *String*[1])])]

**abbreviation** *assocs*  $\equiv$  [

*STR* "*ProjectManager*"  $\mapsto_f$  [  
*STR* "*projects*"  $\mapsto_f$  (*Project*, 0::*nat*,  $\infty$ ::*enat*, *False*, *True*),  
*STR* "*manager*"  $\mapsto_f$  (*Employee*, 1, 1, *False*, *False*),  
*STR* "*ProjectMember*"  $\mapsto_f$  [  
*STR* "*member-of*"  $\mapsto_f$  (*Project*, 0,  $\infty$ , *False*, *False*),  
*STR* "*members*"  $\mapsto_f$  (*Employee*, 1, 20, *True*, *True*),  
*STR* "*ManagerEmployee*"  $\mapsto_f$  [  
*STR* "*line-manager*"  $\mapsto_f$  (*Employee*, 0, 1, *False*, *False*),  
*STR* "*project-manager*"  $\mapsto_f$  (*Employee*, 0,  $\infty$ , *False*, *False*),  
*STR* "*employees*"  $\mapsto_f$  (*Employee*, 3, 7, *False*, *False*),  
*STR* "*ProjectCustomer*"  $\mapsto_f$  [  
*STR* "*projects*"  $\mapsto_f$  (*Project*, 0,  $\infty$ , *False*, *True*),  
*STR* "*customer*"  $\mapsto_f$  (*Customer*, 1, 1, *False*, *False*),

```

STR "ProjectTask" ↦f [
  STR "project" ↦f (Project, 1, 1, False, False),
  STR "tasks" ↦f (Task, 0, ∞, True, True)],
STR "SprintTaskAssignee" ↦f [
  STR "sprint" ↦f (Sprint, 0, 10, False, True),
  STR "tasks" ↦f (Task, 0, 5, False, True),
  STR "assignee" ↦f (Employee, 0, 1, False, False)]

```

**definition** *associations-classes1* ≡ *assocs*

**definition** *association-classes-classes1* ≡ *fmempty* :: *classes1* →<sub>f</sub> *assoc*

```

context Project
def: membersCount() : Integer[1] = members->size()
def: membersByName(mn : String[1]) : Set(Employee[1]) =
  members->select(member | member.name = mn)
static def: allProjects() : Set(Project[1]) =
  Project[1].allInstances()

```

**definition** *operations-classes1* ≡ [

```

(STR "membersCount", Project[1], [], Integer[1], False,
  Some (OperationCall
    (AssociationEndCall (Var STR "self") DotCall None STR "members")
    ArrowCall CollectionSizeOp [])),
(STR "membersByName", Project[1], [(STR "mn", String[1], In)],
  Set Employee[1], False,
  Some (SelectIteratorCall
    (AssociationEndCall (Var STR "self") DotCall None STR "members")
    ArrowCall [STR "member"] None
    (OperationCall
      (AttributeCall (Var STR "member") DotCall STR "name")
      DotCall EqualOp [Var STR "mn"])),
(STR "allProjects", Project[1], [], Set Project[1], True,
  Some (MetaOperationCall Project[1] AllInstancesOp))
] :: (classes1 type, classes1 expr) oper-spec list

```

**definition** *literals-classes1* ≡ *fmap-of-list* [

```

(STR "E1" :: classes1 enum, {|STR "A", STR "B"|}),
(STR "E2", {|STR "C", STR "D", STR "E"|})

```

**lemma** *assoc-end-min-less-eq-max*:

```

assoc |∈| fmdom assocs ⇒
fmlookup assocs assoc = Some ends ⇒
role |∈| fmdom ends ⇒
fmlookup ends role = Some end ⇒
assoc-end-min end ≤ assoc-end-max end
<proof>

```

```

lemma association-ends-unique:
  assumes association-ends' classes assoc $s$  C from role end $_1$ 
    and association-ends' classes assoc $s$  C from role end $_2$ 
  shows  $end_1 = end_2$ 
<proof>

instance
  <proof>

end

```

### 8.3 Simplification Rules

```

lemma ex-alt-simps [simp]:
   $\exists a. a$ 
   $\exists a. \neg a$ 
   $(\exists a. (a \longrightarrow P) \wedge a) = P$ 
   $(\exists a. \neg a \wedge (\neg a \longrightarrow P)) = P$ 
<proof>

declare numeral-eq-enat [simp]

lemmas basic-type-le-less [simp] = Orderings.order-class.le-less
  for  $x\ y :: 'a$  basic-type

declare element-type-alt-simps [simp]
declare update-element-type.simps [simp]
declare to-unique-collection.simps [simp]
declare to-nonunique-collection.simps [simp]
declare to-ordered-collection.simps [simp]

declare assoc-end-class-def [simp]
declare assoc-end-min-def [simp]
declare assoc-end-max-def [simp]
declare assoc-end-ordered-def [simp]
declare assoc-end-unique-def [simp]

declare oper-name-def [simp]
declare oper-context-def [simp]
declare oper-params-def [simp]
declare oper-result-def [simp]
declare oper-static-def [simp]
declare oper-body-def [simp]

declare oper-in-params-def [simp]
declare oper-out-params-def [simp]

declare assoc-end-type-def [simp]

```

```

declare oper-type-def [simp]

declare op-type-alt-simps [simp]
declare typing-alt-simps [simp]
declare normalize-alt-simps [simp]
declare nf-typing.simps [simp]

declare subclass1.intros [intro]
declare less-classes1-def [simp]

declare literals-classes1-def [simp]

lemma attribute-Employee-name [simp]:
  attribute Employee STR "name" D τ =
    (D = Employee  $\wedge$  τ = String[1])
  ⟨proof⟩

lemma association-end-Project-members [simp]:
  association-end Project None STR "members" D τ =
    (D = Project  $\wedge$  τ = (Employee, 1, 20, True, True))
  ⟨proof⟩

lemma association-end-Employee-projects-simp [simp]:
  association-end Employee None STR "projects" D τ =
    (D = Employee  $\wedge$  τ = (Project, 0, ∞, False, True))
  ⟨proof⟩

lemma static-operation-Project-allProjects [simp]:
  static-operation ⟨Project⟩τ[1] STR "allProjects" [] oper =
    (oper = (STR "allProjects", ⟨Project⟩τ[1], [], Set ⟨Project⟩τ[1], True,
      Some (MetaOperationCall ⟨Project⟩τ[1] AllInstancesOp))))
  ⟨proof⟩

```

## 8.4 Basic Types

### 8.4.1 Positive Cases

```

lemma UnlimitedNatural < (Real :: classes1 basic-type) ⟨proof⟩
lemma ⟨Employee⟩τ < ⟨Person⟩τ ⟨proof⟩
lemma ⟨Person⟩τ ≤ OclAny ⟨proof⟩

```

### 8.4.2 Negative Cases

```

lemma  $\neg$  String ≤ (Boolean :: classes1 basic-type) ⟨proof⟩

```

## 8.5 Types

### 8.5.1 Positive Cases

**lemma**  $Integer[?] < (OclSuper :: classes1\ type) \langle proof \rangle$   
**lemma**  $Collection\ Real[?] < (OclSuper :: classes1\ type) \langle proof \rangle$   
**lemma**  $Set\ (Collection\ Boolean[1]) < (OclSuper :: classes1\ type) \langle proof \rangle$   
**lemma**  $Set\ (Bag\ Boolean[1]) < Set\ (Collection\ Boolean[?] :: classes1\ type) \langle proof \rangle$   
**lemma**  $Tuple\ (fmap-of-list\ [(STR\ "a",\ Boolean[1]),\ (STR\ "b",\ Integer[1])]) < Tuple\ (fmap-of-list\ [(STR\ "a",\ Boolean[?] :: classes1\ type)]) \langle proof \rangle$   
  
**lemma**  $Integer[1] \sqcup (Real[?] :: classes1\ type) = Real[?] \langle proof \rangle$   
**lemma**  $Set\ Integer[1] \sqcup Set\ (Real[1] :: classes1\ type) = Set\ Real[1] \langle proof \rangle$   
**lemma**  $Set\ Integer[1] \sqcup Bag\ (Boolean[?] :: classes1\ type) = Collection\ OclAny[?] \langle proof \rangle$   
**lemma**  $Set\ Integer[1] \sqcup (Real[1] :: classes1\ type) = OclSuper \langle proof \rangle$

### 8.5.2 Negative Cases

**lemma**  $\neg OrderedSet\ Boolean[1] < Set\ (Boolean[1] :: classes1\ type) \langle proof \rangle$

## 8.6 Typing

### 8.6.1 Positive Cases

$E1 :: A : E1[1]$

**lemma**  
 $\Gamma_0 \vdash EnumLiteral\ STR\ "E1"\ STR\ "A" : (Enum\ STR\ "E1")[1] \langle proof \rangle$   
 $true\ or\ false : Boolean[1]$

**lemma**  
 $\Gamma_0 \vdash OperationCall\ (BooleanLiteral\ True)\ DotCall\ OrOp\ [BooleanLiteral\ False] : Boolean[1] \langle proof \rangle$   
 $null\ and\ true : Boolean[?]$

**lemma**  
 $\Gamma_0 \vdash OperationCall\ (NullLiteral)\ DotCall\ AndOp\ [BooleanLiteral\ True] : Boolean[?] \langle proof \rangle$   
 $let\ x : Real[1] = 5\ in\ x + 7 : Real[1]$

**lemma**  
 $\Gamma_0 \vdash Let\ (STR\ "x")\ (Some\ Real[1])\ (IntegerLiteral\ 5)\ (OperationCall\ (Var\ STR\ "x")\ DotCall\ PlusOp\ [IntegerLiteral\ 7]) : Real[1] \langle proof \rangle$

```
null.oclIsUndefined() : Boolean[1]
```

**lemma**

```
 $\Gamma_0 \vdash \text{OperationCall } (\text{NullLiteral}) \text{ DotCall } \text{OclIsUndefinedOp } [] : \text{Boolean}[1]$ 
⟨proof⟩
```

```
Sequence{1..5, null}.oclIsUndefined() : Sequence(Boolean[1])
```

**lemma**

```
 $\Gamma_0 \vdash \text{OperationCall } (\text{CollectionLiteral } \text{SequenceKind}$ 
   $[\text{CollectionRange } (\text{IntegerLiteral } 1) (\text{IntegerLiteral } 5),$ 
   $\text{CollectionItem } \text{NullLiteral}])$ 
   $\text{DotCall } \text{OclIsUndefinedOp } [] : \text{Sequence } \text{Boolean}[1]$ 
⟨proof⟩
```

```
Sequence{1..5}->product(Set{'a', 'b'})
  : Set(Tuple(first: Integer[1], second: String[1]))
```

**lemma**

```
 $\Gamma_0 \vdash \text{OperationCall } (\text{CollectionLiteral } \text{SequenceKind}$ 
   $[\text{CollectionRange } (\text{IntegerLiteral } 1) (\text{IntegerLiteral } 5)])$ 
   $\text{ArrowCall } \text{ProductOp}$ 
   $[\text{CollectionLiteral } \text{SetKind}$ 
   $[\text{CollectionItem } (\text{StringLiteral } "a"),$ 
   $\text{CollectionItem } (\text{StringLiteral } "b")]) :$ 
   $\text{Set } (\text{Tuple } (\text{fmap-of-list } [$ 
   $(\text{STR } "first", \text{Integer}[1]), (\text{STR } "second", \text{String}[1]))])$ 
⟨proof⟩
```

```
Sequence{1..5, null}?->iterate(x, acc : Real[1] = 0 | acc + x)
  : Real[1]
```

**lemma**

```
 $\Gamma_0 \vdash \text{IterateCall } (\text{CollectionLiteral } \text{SequenceKind}$ 
   $[\text{CollectionRange } (\text{IntegerLiteral } 1) (\text{IntegerLiteral } 5),$ 
   $\text{CollectionItem } \text{NullLiteral}]) \text{ SafeArrowCall}$ 
   $[\text{STR } "x"] \text{ None}$ 
   $(\text{STR } "acc") (\text{Some } \text{Real}[1]) (\text{IntegerLiteral } 0)$ 
   $(\text{OperationCall } (\text{Var } \text{STR } "acc") \text{ DotCall } \text{PlusOp } [\text{Var } \text{STR } "x"])$ 
   $: \text{Real}[1]$ 
⟨proof⟩
```

```
Sequence{1..5, null}?->max() : Integer[1]
```

**lemma**

```
 $\Gamma_0 \vdash \text{OperationCall } (\text{CollectionLiteral } \text{SequenceKind}$ 
   $[\text{CollectionRange } (\text{IntegerLiteral } 1) (\text{IntegerLiteral } 5),$ 
   $\text{CollectionItem } \text{NullLiteral}])$ 
   $\text{SafeArrowCall } \text{CollectionMaxOp } [] : \text{Integer}[1]$ 
⟨proof⟩
```

```
let x : Sequence(String[?]) = Sequence{'abc', 'zxc'} in
  x->any(it | it = 'test') : String[?]
```

**lemma**



```

 $\Gamma_0 \vdash \text{Let } (STR \text{ "x"}) \text{ (Some (Sequence String[?]))}$ 
  (CollectionLiteral SequenceKind
    [CollectionItem (StringLiteral "abc"),
     CollectionItem (StringLiteral "zxc")])
  (AnyIteratorCall (Var STR "x") ArrowCall
    [STR "it"] None
    (OperationCall (Var STR "it") DotCall EqualOp
      [StringLiteral "test"]))) : String[?]
⟨proof⟩

let x : Sequence(String[?]) = Sequence{'abc', 'zxc'} in
x?->closure(it | it) : OrderedSet(String[1])

```

**lemma**

```

 $\Gamma_0 \vdash \text{Let } STR \text{ "x"} \text{ (Some (Sequence String[?]))}$ 
  (CollectionLiteral SequenceKind
    [CollectionItem (StringLiteral "abc"),
     CollectionItem (StringLiteral "zxc")])
  (ClosureIteratorCall (Var STR "x") SafeArrowCall
    [STR "it"] None
    (Var STR "it")) : OrderedSet String[1]
⟨proof⟩

```

```

context Employee:
name : String[1]

```

**lemma**

```

 $\Gamma_0(STR \text{ "self"} \mapsto_f \text{Employee}[1]) \vdash$ 
  AttributeCall (Var STR "self") DotCall STR "name" : String[1]
⟨proof⟩

```

```

context Employee:
projects : Set(Project[1])

```

**lemma**

```

 $\Gamma_0(STR \text{ "self"} \mapsto_f \text{Employee}[1]) \vdash$ 
  AssociationEndCall (Var STR "self") DotCall None
  STR "projects" : Set Project[1]
⟨proof⟩

```

```

context Employee:
projects.members : Bag(Employee[1])

```

**lemma**

```

 $\Gamma_0(STR \text{ "self"} \mapsto_f \text{Employee}[1]) \vdash$ 
  AssociationEndCall (AssociationEndCall (Var STR "self")
    DotCall None STR "projects")
  DotCall None STR "members" : Bag Employee[1]
⟨proof⟩

```

```

Project[?].allInstances() : Set(Project[?])

```

**lemma**

```

 $\Gamma_0 \vdash \text{MetaOperationCall } Project[?] \text{ AllInstancesOp} : \text{Set } Project[?]$ 

```

*<proof>*

Project[1]::allProjects() : Set(Project[1])

**lemma**

$\Gamma_0 \vdash \text{StaticOperationCall Project}[1] \text{ STR "allProjects" [] : Set Project}[1]$   
*<proof>*

## 8.6.2 Negative Cases

true = null

**lemma**

$\nexists \tau. \Gamma_0 \vdash \text{OperationCall (BooleanLiteral True) DotCall EqualOp}$   
 [NullLiteral] :  $\tau$   
*<proof>*

let x : Boolean[1] = 5 in x and true

**lemma**

$\nexists \tau. \Gamma_0 \vdash \text{Let STR "x" (Some Boolean[1]) (IntegerLiteral 5)}$   
 (OperationCall (Var STR "x") DotCall AndOp [BooleanLiteral True]) :  $\tau$   
*<proof>*

let x : Sequence(String[?]) = Sequence{'abc', 'zxc'} in  
 x->closure(it | 1)

**lemma**

$\nexists \tau. \Gamma_0 \vdash \text{Let STR "x" (Some (Sequence String[?]))}$   
 (CollectionLiteral SequenceKind  
 [CollectionItem (StringLiteral "abc"),  
 CollectionItem (StringLiteral "zxc")])  
 (ClosureIteratorCall (Var STR "x") ArrowCall [STR "it"] None  
 (IntegerLiteral 1)) :  $\tau$   
*<proof>*

Sequence{1..5, null}->max()

**lemma**

$\nexists \tau. \Gamma_0 \vdash \text{OperationCall (CollectionLiteral SequenceKind}$   
 [CollectionRange (IntegerLiteral 1) (IntegerLiteral 5),  
 CollectionItem NullLiteral])  
 ArrowCall CollectionMaxOp [] :  $\tau$   
*<proof>*

## 8.7 Code

### 8.7.1 Positive Cases

**values** {(D,  $\tau$ ). attribute Employee STR "name" D  $\tau$ }  
**values** {(D, end). association-end Employee None STR "employees" D end}  
**values** {(D, end). association-end Employee (Some STR "project-manager") STR  
 "employees" D end}

```

values {op. operation Project[1] STR "membersCount" [] op}
values {op. operation Project[1] STR "membersByName" [String[1]] op}
value has-literal STR "E1" STR "A"

  context Employee:
    projects.members : Bag(Employee[1])

values
  { $\tau. \Gamma_0(\text{STR } "self" \mapsto_f \text{Employee}[1]) \vdash$ 
    AssociationEndCall (AssociationEndCall (Var STR "self")
      DotCall None STR "projects")
    DotCall None STR "members" :  $\tau$ }

```

### 8.7.2 Negative Cases

```

values {( $\mathcal{D}, \tau$ ). attribute Employee STR "name2"  $\mathcal{D} \tau$ }
value has-literal STR "E1" STR "C"

  Sequence{1..5, null}->max()

values
  { $\tau. \Gamma_0 \vdash \text{OperationCall} (\text{CollectionLiteral SequenceKind}$ 
    [CollectionRange (IntegerLiteral 1) (IntegerLiteral 5),
      CollectionItem NullLiteral])
    ArrowCall CollectionMaxOp [] :  $\tau$ }

end

```



# Bibliography

- [1] Object Management Group, “Object Constraint Language (OCL). Version 2.4,” Feb. 2014. <http://www.omg.org/spec/OCL/2.4/>.
- [2] A. D. Brucker, F. Tuong, and B. Wolff, “Featherweight OCL: A proposal for a machine-checked formal semantics for OCL 2.5,” *Archive of Formal Proofs*, Jan. 2014. [http://isa-afp.org/entries/Featherweight\\_OCL.html](http://isa-afp.org/entries/Featherweight_OCL.html), Formal proof development.
- [3] E. D. Willink, “Safe navigation in OCL,” in *Proceedings of the 15th International Workshop on OCL and Textual Modeling co-located with 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, Canada, September 28, 2015*. (A. D. Brucker, M. Egea, M. Gogolla, and F. Tuong, eds.), vol. 1512 of *CEUR Workshop Proceedings*, pp. 81–88, CEUR-WS.org, 2015.