

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 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	9
1.2.4 Transitive Closure Reflection	10
1.3 Finite Maps	11
1.3.1 Helper Lemmas	11
1.3.2 Merge Operation	12
1.3.3 Acyclicity	13
1.3.4 Transitive Closures	14
1.3.5 Size Calculation	18
1.3.6 Code Setup	19
1.4 Tuples	19
1.4.1 Definitions	19
1.4.2 Helper Lemmas	20
1.4.3 Basic Properties	20
1.4.4 Transitive Closures	23
1.4.5 Code Setup	24
1.5 Object Model	25
1.5.1 Type Synonyms	25
1.5.2 Attributes	26
1.5.3 Association Ends	26
1.5.4 Association Classes	27
1.5.5 Association Class Ends	28
1.5.6 Operations	29
1.5.7 Literals	30
1.5.8 Definition	30
1.5.9 Properties	31
1.5.10 Code Setup	32

<b>2</b>	<b>Basic Types</b>	<b>37</b>
2.1	Definition . . . . .	37
2.2	Partial Order of Basic Types . . . . .	38
2.2.1	Strict Introduction Rules . . . . .	38
2.2.2	Strict Elimination Rules . . . . .	39
2.2.3	Properties . . . . .	41
2.2.4	Non-Strict Introduction Rules . . . . .	41
2.2.5	Non-Strict Elimination Rules . . . . .	42
2.2.6	Simplification Rules . . . . .	43
2.3	Upper Semilattice of Basic Types . . . . .	44
2.4	Code Setup . . . . .	45
<b>3</b>	<b>Types</b>	<b>47</b>
3.1	Definition . . . . .	47
3.2	Constructors Bijectivity on Transitive Closures . . . . .	49
3.3	Partial Order of Types . . . . .	50
3.3.1	Strict Introduction Rules . . . . .	50
3.3.2	Strict Elimination Rules . . . . .	53
3.3.3	Properties . . . . .	55
3.3.4	Non-Strict Introduction Rules . . . . .	56
3.3.5	Non-Strict Elimination Rules . . . . .	57
3.3.6	Simplification Rules . . . . .	58
3.4	Upper Semilattice of Types . . . . .	59
3.5	Helper Relations . . . . .	62
3.6	Determinism . . . . .	64
3.7	Code Setup . . . . .	64
<b>4</b>	<b>Abstract Syntax</b>	<b>69</b>
4.1	Preliminaries . . . . .	69
4.2	Standard Library Operations . . . . .	69
4.3	Expressions . . . . .	71
<b>5</b>	<b>Object Model</b>	<b>75</b>
<b>6</b>	<b>Typing</b>	<b>77</b>
6.1	Operations Typing . . . . .	77
6.1.1	Metaclass Operations . . . . .	77
6.1.2	Type Operations . . . . .	77
6.1.3	OclSuper Operations . . . . .	78
6.1.4	OclAny Operations . . . . .	78
6.1.5	Boolean Operations . . . . .	79
6.1.6	Numeric Operations . . . . .	79
6.1.7	String Operations . . . . .	81
6.1.8	Collection Operations . . . . .	81

6.1.9	Coercions . . . . .	84
6.1.10	Simplification Rules . . . . .	85
6.1.11	Determinism . . . . .	85
6.2	Expressions Typing . . . . .	88
6.3	Elimination Rules . . . . .	92
6.4	Simplification Rules . . . . .	93
6.5	Determinism . . . . .	94
6.6	Code Setup . . . . .	98
<b>7</b>	<b>Normalization</b> . . . . .	<b>99</b>
7.1	Normalization Rules . . . . .	99
7.2	Elimination Rules . . . . .	103
7.3	Simplification Rules . . . . .	104
7.4	Determinism . . . . .	104
7.5	Normalized Expressions Typing . . . . .	107
7.6	Code Setup . . . . .	107
<b>8</b>	<b>Examples</b> . . . . .	<b>109</b>
8.1	Classes . . . . .	109
8.2	Object Model . . . . .	111
8.3	Simplification Rules . . . . .	113
8.4	Basic Types . . . . .	115
8.4.1	Positive Cases . . . . .	115
8.4.2	Negative Cases . . . . .	115
8.5	Types . . . . .	116
8.5.1	Positive Cases . . . . .	116
8.5.2	Negative Cases . . . . .	116
8.6	Typing . . . . .	116
8.6.1	Positive Cases . . . . .	116
8.6.2	Negative Cases . . . . .	119
8.7	Code . . . . .	119
8.7.1	Positive Cases . . . . .	119
8.7.2	Negative Cases . . . . .	120



# Chapter 1

## Preliminaries

### 1.1 Errorable

```
theory Errorable
  imports Main
begin

notation bot (  $\perp$  )

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

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 ..
end

free-constructors case-errorable for
  errorable
|  $\perp$  :: 'a errorable
  unfolding errorable-def bot-errorable-def
  apply (metis Abs-errorable-cases not-None-eq)
  apply (metis Abs-errorable-inverse UNIV-I option.inject)
  by (simp add: Abs-errorable-inject)

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** (*intro ext iffI*)

**fix**  $x y$

**have**  $(\lambda x y. P x y \vee x = y)^{**} x y \longrightarrow P^{**} x y$

**by** (*rule mono-rtranclp*) *simp*

**thus**  $(\lambda x y. P x y \vee x = y)^{**} x y \Longrightarrow P^{**} x y$

**by** *simp*

**show**  $P^{**} x y \Longrightarrow (\lambda x y. P x y \vee x = y)^{**} x y$

**by** (*metis (no-types, lifting) mono-rtranclp*)

**qed**

**lemma** *tranclp-eq-rtranclp* [iff]:

$$(\lambda x y. P x y \vee x = y)^{++} = P^{**}$$

**proof** (*intro ext iffI*)

**fix**  $x y$

**have**  $(\lambda x y. P x y \vee x = y)^{**} x y \longrightarrow P^{**} x y$

**by** (*rule mono-rtranclp*) *simp*

**thus**  $(\lambda x y. P x y \vee x = y)^{++} x y \Longrightarrow P^{**} x y$

**using** *tranclp-into-rtranclp* **by** *force*

**show**  $P^{**} x y \Longrightarrow (\lambda x y. P x y \vee x = y)^{++} x y$

**by** (*metis (mono-tags, lifting) mono-rtranclp rtranclpD tranclp.r-into-trancl*)

**qed**

**lemma** *rtranclp-eq-rtranclp'* [iff]:

$$(\lambda x y. P x y \wedge x \neq y)^{**} = P^{**}$$

**proof** (*intro ext iffI*)

**fix**  $x y$

**show**  $(\lambda x y. P x y \wedge x \neq y)^{**} x y \Longrightarrow P^{**} x y$

**by** (*metis (no-types, lifting) mono-rtranclp*)



**assume**  $P^{**} x y$   
**hence**  $(\inf P (\neq))^{**} x y$   
**by**  $(\text{simp add: rtranclp-r-diff-Id})$   
**also have**  $(\inf P (\neq))^{**} x y \longrightarrow (\lambda x y. P x y \wedge x \neq y)^{**} x y$   
**by**  $(\text{rule mono-rtranclp}) \text{ simp}$   
**finally show**  $P^{**} x y \Longrightarrow (\lambda x y. P x y \wedge x \neq y)^{**} x y$  **by**  $\text{simp}$   
**qed**

**lemma**  $\text{tranclp-tranclp-to-tranclp-r}$ :  
**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** –  
**have**  $(\bigwedge x y z. R^{++} x y \Longrightarrow R y z \Longrightarrow P x \Longrightarrow P z \Longrightarrow P y) \Longrightarrow$   
 $R^{++} y z \Longrightarrow R^{++} x y \Longrightarrow P x \longrightarrow P z \longrightarrow P y$   
**by**  $(\text{erule tranclp-induct, auto}) (\text{meson tranclp-trans})$   
**thus**  $?thesis$  **using**  $\text{assms}$  **by**  $\text{auto}$   
**qed**

### 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**  $\text{preserve-tranclp}$ :  
**assumes**  $\bigwedge x y. R x y \Longrightarrow S (f x) (f y)$   
**assumes**  $R^{++} x y$   
**shows**  $S^{++} (f x) (f y)$   
**proof** –  
**define**  $P$  **where**  $P: P = (\lambda x y. S^{++} (f x) (f y))$   
**define**  $r$  **where**  $r: r = (\lambda x y. S (f x) (f y))$   
**have**  $r^{++} x y$  **by**  $(\text{insert assms } r; \text{erule tranclp-trans-induct; auto})$   
**moreover have**  $\bigwedge x y. r x y \Longrightarrow P x y$  **unfolding**  $P r$  **by**  $\text{simp}$   
**moreover have**  $\bigwedge x y z. r^{++} x y \Longrightarrow P x y \Longrightarrow r^{++} y z \Longrightarrow P y z \Longrightarrow P x z$   
**unfolding**  $P$  **by**  $\text{auto}$   
**ultimately have**  $P x y$  **by**  $(\text{rule tranclp-trans-induct})$   
**with**  $P$  **show**  $?thesis$  **by**  $\text{simp}$   
**qed**

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

**lemma**  $\text{preserve-rtranclp}$ :  
 $(\bigwedge x y. R x y \Longrightarrow S (f x) (f y)) \Longrightarrow$   
 $R^{**} x y \Longrightarrow S^{**} (f x) (f y)$   
**unfolding**  $\text{Nitpick.rtranclp-unfold}$   
**by**  $(\text{metis preserve-tranclp})$

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'*:  
 $(\bigwedge x y. R x y \implies S (f x) (f y)) \implies$   
 $(\bigwedge y. S (f y) (g y)) \implies$   
 $R^{**} x y \implies S^{**} (f x) (g y)$   
**by** (*metis preserve-rtranclp rtranclp.rtrancl-into-rtrancl*)

**lemma** *preserve-rtranclp''*:  
 $(\bigwedge x y. R x y \implies S (f x) (f y)) \implies$   
 $(\bigwedge y. S (f y) (g y)) \implies$   
 $R^{**} x y \implies S^{++} (f x) (g y)$   
**apply** (*rule-tac ?b= f y in rtranclp-into-tranclp1, auto*)  
**by** (*rule preserve-rtranclp, auto*)

#### 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** *reft-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**  $R^{++} x y$   
**proof** –  
**define**  $B$  **where**  $B$ :  $B = \text{range } f$   
**define**  $g$  **where**  $g$ :  $g = \text{the-inv-into UNIV } f$   
**define**  $gr$  **where**  $gr$ :  $gr = \text{restrict } g B$   
**define**  $P$  **where**  $P$ :  $P = (\lambda x y. x \in B \longrightarrow y \in B \longrightarrow R^{++} (gr x) (gr y))$   
**from** *prem* **have** *major*:  $S^{++} (f x) (f y)$  **by** *blast*  
**from** *reft-f* *bij-f* **have** *cases-1*:  $\bigwedge x y. S x y \implies P x y$   
**unfolding**  $B P g gr$   
**by** (*simp add: f-the-inv-into-f tranclp.r-into-trancl*)  
**from** *reft-f* *bij-f*  
**have**  $(\bigwedge x y z. S^{++} x y \implies S^{++} y z \implies x \in B \implies z \in B \implies y \in B)$   
**unfolding**  $B$   
**by** (*rule-tac ?z= z in tranclp-tranclp-to-tranclp-r, auto, blast*)  
**with**  $P$  **have** *cases-2*:  
 $\bigwedge x y z. S^{++} x y \implies P x y \implies S^{++} y z \implies P y z \implies P x z$   
**unfolding**  $B$   
**by** *auto*  
**from** *major* *cases-1* *cases-2* **have**  $P (f x) (f y)$   
**by** (*rule tranclp-trans-induct*)  
**with** *bij-f* **show** *?thesis* **unfolding**  $P B g gr$  **by** (*simp add: the-inv-f-f*)

qed

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

**lemma** *reflect-rtranclp*:

$(\bigwedge x y. S (f x) (f y) \implies R x y) \implies$

*bij-on-trancl*  $S f \implies$

$S^{**} (f x) (f y) \implies R^{**} x y$

**unfolding** *Nitpick.rtranclp-unfold*

**by** (*metis (full-types) injD reflect-tranclp*)

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**

*-fmaplet*  $:: [ 'a, 'a ] \Rightarrow \text{fmaplet} \quad ( - / \mapsto_f / - )$   
*-fmaplets*  $:: [ 'a, 'a ] \Rightarrow \text{fmaplet} \quad ( - / [\mapsto_f] / - )$   
 $:: \text{fmaplet} \Rightarrow \text{fmaplets} \quad ( - )$   
*-FMaplets*  $:: [\text{fmaplet}, \text{fmaplets}] \Rightarrow \text{fmaplets} \quad ( -, / - )$   
*-FMapUpd*  $:: [ 'a \rightarrow 'b, \text{fmaplets} ] \Rightarrow 'a \rightarrow 'b \quad ( - / (-) )$  [900, 0] 900)  
*-FMap*  $:: \text{fmaplets} \Rightarrow 'a \rightarrow 'b \quad ( (1[-]) )$

**syntax** (*ASCII*)

*-fmaplet*  $:: [ 'a, 'a ] \Rightarrow \text{fmaplet} \quad ( - / | \rightarrow_f / - )$   
*-fmaplets*  $:: [ 'a, 'a ] \Rightarrow \text{fmaplet} \quad ( - / [ | \rightarrow_f ] / - )$

**translations**

*-FMapUpd*  $m (-FMaplets \ xy \ ms) \quad \rightleftharpoons \text{-FMapUpd} (-FMapUpd \ m \ xy) \ ms$   
*-FMapUpd*  $m (-fmaplet \ x \ y) \quad \rightleftharpoons \text{CONST} \ \text{fmupd} \ x \ y \ m$   
*-FMap*  $ms \quad \rightleftharpoons \text{-FMapUpd} (\text{CONST} \ \text{fmempty}) \ ms$   
*-FMap*  $(-FMaplets \ ms1 \ ms2) \quad \leftarrow \text{-FMapUpd} (-FMap \ ms1) \ ms2$   
*-FMaplets*  $ms1 (-FMaplets \ ms2 \ ms3) \quad \leftarrow \text{-FMaplets} (-FMaplets \ ms1 \ ms2) \ ms3$

### 1.3.1 Helper Lemmas

**lemma** *fmrel-on-fset-fmdom*:

*fmrel-on-fset*  $(\text{fmdom} \ ym) \ f \ x \ m \ ym \implies$

$k \in | \text{fmdom} \ ym \implies$

$k \in | \text{fmdom} \ x \ m$

**by** (*metis fmdom-notD fmdom-notI fmrel-on-fsetD option.rel-sel*)

### 1.3.2 Merge Operation

**definition**  $fmmerge\ f\ xm\ ym \equiv$   
 $fmap\text{-of-list}\ (map$   
 $(\lambda k. (k, f\ (the\ (fmlookup\ xm\ k))\ (the\ (fmlookup\ ym\ k))))$   
 $(sorted\text{-list-of-fset}\ (fmdom\ xm\ |\cap|\ fmdom\ ym)))$

**lemma**  $fmdom\text{-}fmmerge\ [simp]:$   
 $fmdom\ (fmmerge\ g\ xm\ ym) = fmdom\ xm\ |\cap|\ fmdom\ ym$   
**by**  $(auto\ simp\ add: fmmerge\text{-}def\ fmdom\text{-of-list})$

**lemma**  $fmmerge\text{-}commut:$   
**assumes**  $\bigwedge x\ y. x \in fmran'\ xm \implies f\ x\ y = f\ y\ x$   
**shows**  $fmmerge\ f\ xm\ ym = fmmerge\ f\ ym\ xm$

**proof** –  
**obtain**  $zm$  **where**  $zm: zm = sorted\text{-list-of-fset}\ (fmdom\ xm\ |\cap|\ fmdom\ ym)$   
**by**  $auto$   
**with**  $assms$  **have**  
 $map\ (\lambda k. (k, f\ (the\ (fmlookup\ xm\ k))\ (the\ (fmlookup\ ym\ k))))\ zm =$   
 $map\ (\lambda k. (k, f\ (the\ (fmlookup\ ym\ k))\ (the\ (fmlookup\ xm\ k))))\ zm$   
**by**  $(auto)\ (metis\ fmdom\text{-}notI\ fmran'\ I\ option.\ collapse)$   
**thus**  $?thesis$   
**unfolding**  $fmmerge\text{-}def\ zm$   
**by**  $(metis\ (no\text{-}types,\ lifting)\ inf\text{-}commute)$   
**qed**

**lemma**  $fmrel\text{-}on\text{-}fset\text{-}fmmerge1\ [intro]:$   
**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** –  
 $\{$   
**fix**  $x\ a\ b\ c$   
**assume**  $x \in fmdom\ zm$   
**moreover** **hence**  $x \in fmdom\ xm\ |\cap|\ fmdom\ ym$   
**by**  $(meson\ assms(2)\ assms(3)\ finterI\ fmrel\text{-}on\text{-}fset\text{-}fmdom)$   
**moreover** **assume**  $fmlookup\ xm\ x = Some\ a$   
**and**  $fmlookup\ ym\ x = Some\ b$   
**and**  $fmlookup\ zm\ x = Some\ c$   
**moreover** **from**  $assms$  **calculation** **have**  $f\ (g\ a\ b)\ c$   
**by**  $(metis\ fmran'\ I\ fmrel\text{-}on\text{-}fsetD\ option.\ rel\text{-}inject(2))$   
**ultimately** **have**  
 $rel\text{-}option\ f\ (fmlookup\ (fmmerge\ g\ xm\ ym)\ x)\ (fmlookup\ zm\ x)$   
**unfolding**  $fmmerge\text{-}def\ fmlookup\text{-of-list}$  **apply**  $auto$   
**unfolding**  $option\text{-}rel\text{-}Some2$  **apply**  $(rule\text{-}tac\ ?x = g\ a\ b\ \text{in}\ exI)$   
**unfolding**  $map\text{-of-map-restrict}\ restrict\text{-map-def}$   
**by**  $auto$   
 $\}$   
**with**  $assms(2)\ assms(3)$  **show**  $?thesis$

by (*meson fmdomE fmrel-on-fsetI fmrel-on-fset-fmdom*)  
qed

**lemma** *fmrel-on-fset-fmmerge2* [*intro*]:

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

**shows** *fmrel-on-fset* (*fmdom ym*) *f xm* (*fmmerge g xm ym*)

**proof** –

{

**fix** *x a b*

**assume**  $x \in | \text{fmdom}\ xm \cap | \text{fmdom}\ ym$

**and** *fmlookup xm x = Some a*

**and** *fmlookup ym x = Some b*

**hence** *rel-option f* (*fmlookup xm x*) (*fmlookup (fmmerge g xm ym) x*)

**unfolding** *fmmerge-def fmlookup-of-list* **apply** *auto*

**unfolding** *option-rel-Some1* **apply** (*rule-tac ?x= g a b in exI*)

**by** (*auto simp add: map-of-map-restrict assms fmran'I*)

}

**with** *assms* **show** *?thesis*

**apply** *auto*

**apply** (*rule fmrel-on-fsetI*)

**by** (*metis (full-types) finterD1 fmdomE fmdom-fmmerge fmdom-notD rel-option-None2*)

qed

### 1.3.3 Acyclicity

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

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

**lemma** *fmrel-acyclic*:

*acyclicP-on (fmran' xm) R*  $\implies$

*fmrel R<sup>++</sup> xm ym*  $\implies$

*fmrel R ym xm*  $\implies$

*xm = ym*

**by** (*metis (full-types) fmap-ext fmran'I fmrel-cases option.sel tranclp.trancl-into-trancl tranclp-unfold*)

**lemma** *fmrel-acyclic'*:

**assumes** *acyclicP-on (fmran' ym) R*

**assumes** *fmrel R<sup>++</sup> xm ym*

**assumes** *fmrel R ym xm*

**shows** *xm = ym*

**proof** –

{

**fix** *x*

**from** *assms(1)* **have**

*rel-option R<sup>++</sup> (fmlookup xm x) (fmlookup ym x)*  $\implies$

*rel-option R (fmlookup ym x) (fmlookup xm x)*  $\implies$

*rel-option R (fmlookup xm x) (fmlookup ym x)*

```

    by (metis (full-types) fmdom'-notD fmlookup-dom'-iff
        fmran'I option.rel-sel option.sel
        tranclp-into-tranclp2 tranclp-unfold)
  }
  with assms show ?thesis
  unfolding fmrel-iff
  by (metis fmap.rel-mono-strong fmrelI fmrel-acyclic tranclp.simps)
qed

```

**lemma** *fmrel-on-fset-acyclic*:

```

  acyclicP-on (fmran' xm) R  $\implies$ 
  fmrel-on-fset (fmdom ym) R++ xm ym  $\implies$ 
  fmrel-on-fset (fmdom xm) R ym xm  $\implies$ 
  xm = ym
  unfolding fmrel-on-fset-fmrel-restrict
  by (metis (no-types, lifting) fmdom-filter fmfilter-alt-defs(5)
      fmfilter-cong fmlookup-filter fmrel-acyclic fmrel-fmdom-eq
      fmrestrict-fset-dom option.simps(3))

```

**lemma** *fmrel-on-fset-acyclic'*:

```

  acyclicP-on (fmran' ym) R  $\implies$ 
  fmrel-on-fset (fmdom ym) R++ xm ym  $\implies$ 
  fmrel-on-fset (fmdom xm) R ym xm  $\implies$ 
  xm = ym
  unfolding fmrel-on-fset-fmrel-restrict
  by (metis (no-types, lifting) fmember-filter fmdom-filter
      fmfilter-alt-defs(5) fmfilter-cong fmrel-acyclic'
      fmrel-fmdom-eq fmrestrict-fset-dom)

```

### 1.3.4 Transitive Closures

**lemma** *fmrel-trans*:

```

  ( $\bigwedge x y z. x \in \text{fmran}' xm \implies P x y \implies Q y z \implies R x z$ )  $\implies$ 
  fmrel P xm ym  $\implies$  fmrel Q ym zm  $\implies$  fmrel R xm zm
  unfolding fmrel-iff
  by (metis fmdomE fmdom-notD fmran'I option.rel-inject(2) option.rel-sel)

```

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

```

  ( $\bigwedge x y z. x \in \text{fmran}' xm \implies P x y \implies Q y z \implies R x z$ )  $\implies$ 
  fmrel-on-fset (fmdom ym) P xm ym  $\implies$ 
  fmrel-on-fset (fmdom zm) Q ym zm  $\implies$ 
  fmrel-on-fset (fmdom zm) R xm zm
  apply (rule fmrel-on-fsetI)
  unfolding option.rel-sel apply auto
  apply (meson fmdom-notI fmrel-on-fset-fmdom)
  by (metis fmdom-notI fmran'I fmrel-on-fsetD fmrel-on-fset-fmdom
      option.rel-sel option.sel)

```

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

```

(fmrel f)++ xm ym  $\implies$  fmrel f++ xm ym
apply (induct rule: tranclp-induct)
apply (simp add: fmap.rel-mono-strong)
by (rule fmrel-trans; auto)

```

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

```

(fmrel f)++ xm ym  $\implies$  fmdom xm = fmdom ym
by (induct rule: tranclp-induct; simp add: fmrel-fmdom-eq)

```

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*:

```

fmlookup xm k = Some x  $\implies$ 
xm = fmupd k x (fmdrop k xm)
apply (rule fmap-ext)
unfolding fmlookup-drop fmupd-lookup
by auto

```

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

```

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

```

**proof** –

```

from assms(2) obtain y where fmlookup ym i = Some y by force
then obtain z zm where z-zm: ym = fmupd i z zm  $\wedge$  fmlookup zm i = None
using fmupd-fmdrop by force

```

```

{
assume  $\neg$  R x z
with z-zm have  $\neg$  fmrel R (fmupd i x xm) ym
by (metis fmrel-iff fmupd-lookup option.simps(11))
}

```

```

with assms(3) moreover have R x z by auto

```

```

{
assume  $\neg$  fmrel R xm zm
with assms(1) have  $\neg$  fmrel R (fmupd i x xm) ym
by (metis fmrel-iff fmupd-lookup option.rel-sel z-zm)
}

```

```

with assms(3) moreover have fmrel R xm zm by auto
ultimately show ?thesis using z-zm by blast

```

**qed**

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 = fmdom ym
and nil: P (fmap-of-list []) (fmap-of-list [])
and step:
   $\bigwedge x xm y ym i.$ 
   $\llbracket R x y; fmrel R xm ym; fmdom xm = fmdom ym; P xm ym \rrbracket \implies$ 
  P (fmupd i x xm) (fmupd i y ym)
shows P xm ym
using R dom-eq
proof (induct xm arbitrary: ym)
  case fmempty thus ?case
    by (metis fempty-iff fmdom-empty fmempty-of-list fmfilt-alt-defs(5)
      fmfilt-false fmrestrict-fset-dom local.nil)
next
  case (fmupd i x xm) show ?case
  proof -
    obtain y where fmllookup ym i = Some y
    by (metis fmupd.premis(1) fmrel-cases fmupd-lookup option.discI)
    from fmupd.hyps(2) fmupd.premis(1) fmupd.premis(2) obtain z zm where
      fmllookup zm i = None and
      ym-eq-z-zm: ym = (fmupd i z zm) and
      R-x-z: R x z and
      R-xm-zm: fmrel R xm zm
    using fmap-eqdom-Cons1 by metis
    hence dom-xm-eq-dom-zm: fmdom xm = fmdom zm
    using fmrel-fmdom-eq by blast
    with R-xm-zm fmupd.hyps(1) have P xm zm by blast
    with R-x-z R-xm-zm dom-xm-eq-dom-zm have
      P (fmupd i x xm) (fmupd i z zm)
    by (rule step)
    thus ?thesis by (simp add: ym-eq-z-zm)
  qed
qed

```

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 -
  from rel-rpp-xm-ym have fmdom xm = fmdom ym
  using fmrel-fmdom-eq by blast
  with rel-rpp-xm-ym show (fmrel r)** xm ym
proof (induct rule: fmap-eqdom-induct)
  case nil show ?case by auto
next
  case (step x xm y ym i) show ?case
  proof -

```



```

from step.hyps(1) have  $(fmrel\ r)^{**} (fmupd\ i\ x\ xm)\ (fmupd\ i\ y\ xm)$ 
proof (induct rule: rtranclp-induct)
  case base show ?case by simp
next
  case (step y z) show ?case
  proof –
    from as-r have  $fmrel\ r\ xm\ xm$ 
    by (simp add: fmap.rel-reflp reflpD)
    with step.hyps(2) have  $(fmrel\ r)^{**} (fmupd\ i\ y\ xm)\ (fmupd\ i\ z\ xm)$ 
    by (simp add: fmrel-upd r-into-rtranclp)
    with step.hyps(3) show ?thesis by simp
  qed
qed
also from step.hyps(4) have  $(fmrel\ r)^{**} (fmupd\ i\ y\ xm)\ (fmupd\ i\ y\ ym)$ 
proof (induct rule: rtranclp-induct)
  case base show ?case by simp
next
  case (step ya za) show ?case
  proof –
    from step.hyps(2) as-r have  $(fmrel\ r)^{**} (fmupd\ i\ y\ ya)\ (fmupd\ i\ y\ za)$ 
    by (simp add: fmrel-upd r-into-rtranclp reflp-def)
    with step.hyps(3) show ?thesis by simp
  qed
qed
finally show ?thesis by simp
qed
qed
qed

```

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 –
  from assms(2) have  $fmrel\ r^{**}\ xm\ ym$ 
  by (drule-tac ?Ra= r^{**} in fmap.rel-mono-strong; auto)
  with assms(1) have  $(fmrel\ r)^{**}\ xm\ ym$ 
  by (simp add: fmrel-to-rtrancl)
  with assms(1) show ?thesis
  by (metis fmap.rel-reflp reflpD rtranclpD tranclp.r-into-trancl)
qed

```

**lemma** *fmrel-tranclp-induct*:

```

 $fmrel\ r^{++}\ a\ b \implies$ 
 $reflp\ r \implies$ 
 $(\bigwedge y. fmrel\ r\ a\ y \implies P\ y) \implies$ 

```

```

( $\wedge y z. (fmrel\ r)^{++} a\ y \implies fmrel\ r\ y\ z \implies P\ y \implies P\ z$ )  $\implies P\ b$ 
apply (drule fmrel-to-trancl, simp)
by (erule tranclp-induct; simp)

```

**lemma** *fmrel-converse-tranclp-induct*:

```

fmrel r++ a b  $\implies$ 
reflp r  $\implies$ 
( $\wedge y. fmrel\ r\ y\ b \implies P\ y$ )  $\implies$ 
( $\wedge y z. fmrel\ r\ y\ z \implies fmrel\ r^{++}\ z\ b \implies P\ z \implies P\ y$ )  $\implies P\ a$ 
apply (drule fmrel-to-trancl, simp)
by (erule converse-tranclp-induct; simp add: trancl-to-fmrel)

```

**lemma** *fmrel-tranclp-trans-induct*:

```

fmrel r++ a b  $\implies$ 
reflp r  $\implies$ 
( $\wedge x y. fmrel\ r\ x\ y \implies P\ x\ y$ )  $\implies$ 
( $\wedge x y z. fmrel\ r^{++}\ x\ y \implies P\ x\ y \implies fmrel\ r^{++}\ y\ z \implies P\ y\ z \implies P\ x\ z$ )  $\implies$ 
P a b
apply (drule fmrel-to-trancl, simp)
apply (erule tranclp-trans-induct, simp)
using trancl-to-fmrel by blast

```

### 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**

```

fix x y :: 'a  $\times$  nat
show tcf y  $\circ$  tcf x = tcf x  $\circ$  tcf y
proof -
  fix z
  have (tcf y  $\circ$  tcf x) z = snd y + snd x + z by auto
  also have (tcf x  $\circ$  tcf y) z = snd y + snd x + z by auto
  finally have (tcf y  $\circ$  tcf x) z = (tcf x  $\circ$  tcf y) z by auto
  thus (tcf y  $\circ$  tcf x) = (tcf x  $\circ$  tcf y) by auto
qed
qed

```

**lemma** *ffold-rec-exp*:

```

assumes k  $\in$  |fmdom x
  and ky = (k, the (fmlookup (fmmap f x) k))
shows ffold tcf 0 (fset-of-fmap (fmmap f x)) =
  tcf ky (ffold tcf 0 ((fset-of-fmap (fmmap f x)) |-| {ky}))
proof -

```

```

have  $ky \in |$  (fset-of-fmap (fmap f x))
  using assms by auto
thus ?thesis
  by (simp add: tcf.ffold-rec)
qed

```

```

lemma elem-le-ffold [intro]:
   $k \in | \text{fmdom } x \implies$ 
   $f (\text{the } (\text{fmllookup } x \ k)) < \text{Suc } (\text{ffold } \text{tcf } 0 \ (\text{fset-of-fmap } (\text{fmap } f \ x)))$ 
  by (subst ffold-rec-exp, auto)

```

```

lemma elem-le-ffold' [intro]:
   $z \in \text{fmrans } x \implies$ 
   $f \ z < \text{Suc } (\text{ffold } \text{tcf } 0 \ (\text{fset-of-fmap } (\text{fmap } f \ x)))$ 
  apply (erule fmrans'E)
  apply (frule fmdomI)
  by (subst ffold-rec-exp, auto)

```

### 1.3.6 Code Setup

```

abbreviation fmmerge-fun  $f \ x \ y \equiv$ 
  fmap-of-list (map
    ( $\lambda k. \text{if } k \in | \text{fmdom } x \wedge k \in | \text{fmdom } y$ 
      then ( $k, f \ (\text{the } (\text{fmllookup } x \ k)) \ (\text{the } (\text{fmllookup } y \ k))$ )
      else ( $k, \text{undefined}$ ))
    (sorted-list-of-fset ( $\text{fmdom } x \ \cup \ \text{fmdom } y$ )))

```

```

lemma fmmerge-fun-simp [code-abbrev, simp]:
   $\text{fmmerge-fun } f \ x \ y = \text{fmmerge } f \ x \ y$ 
  unfolding fmmerge-def
  apply (rule-tac ?f= fmap-of-list in HOL.arg-cong)
  by simp

```

**end**

## 1.4 Tuples

```

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

```

### 1.4.1 Definitions

```

abbreviation subtuple  $f \ x \ y \equiv \text{fmrel-on-fset } (\text{fmdom } y) \ f \ x \ y$ 

```

```

abbreviation strict-subtuple  $f \ x \ y \equiv \text{subtuple } f \ x \ y \wedge x \neq y$ 

```

### 1.4.2 Helper Lemmas

**lemma** *fmrel-to-subtuple*:

$fmrel\ r\ xm\ ym \implies subtuple\ r\ xm\ ym$   
**unfolding** *fmrel-on-fset-fmrel-restrict* **by** *blast*

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

$subtuple\ r\ xm\ ym = fmrel\ r\ (fmrestrict\ fset\ (fmdom\ ym)\ xm)\ ym$   
**by** (*simp add: fmrel-on-fset-fmrel-restrict*)

**lemma** *subtuple-fmdom*:

$subtuple\ f\ xm\ ym \implies$   
 $subtuple\ g\ ym\ xm \implies$   
 $fmdom\ xm = fmdom\ ym$   
**by** (*meson fmrel-on-fset-fmdom fset-eqI*)

### 1.4.3 Basic Properties

**lemma** *subtuple-refl*:

$reflp\ R \implies subtuple\ R\ xm\ xm$   
**by** (*simp add: fmrel-on-fsetI option.rel-reflp reflpD*)

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

$(\bigwedge x\ y. x \in fmran'\ xm \implies y \in fmran'\ ym \implies f\ x\ y \longrightarrow g\ x\ y) \implies$   
 $subtuple\ f\ xm\ ym \longrightarrow subtuple\ g\ xm\ ym$   
**apply** (*auto*)  
**apply** (*rule fmrel-on-fsetI*)  
**apply** (*drule-tac ?P=f and ?m=xm and ?n=ym in fmrel-on-fsetD, simp*)  
**apply** (*erule option.rel-cases, simp*)  
**by** (*auto simp add: option.rel-sel fmran'I*)

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

$(\bigwedge x\ y. x \in fmran'\ xm \implies y \in fmran'\ ym \implies f\ x\ y \longrightarrow g\ x\ y) \implies$   
 $strict\ subtuple\ f\ xm\ ym \longrightarrow strict\ subtuple\ g\ xm\ ym$   
**using** *subtuple-mono* **by** *blast*

**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** (*rule fmap-ext*)

**fix**  $x$   
**from** *assms* **have**  $fmdom\ xm = fmdom\ ym$   
**using** *subtuple-fmdom* **by** *blast*  
**with** *assms* **have**  $fmrel\ (\lambda x\ y. f\ x\ y \vee x = y)\ xm\ ym$   
**and**  $fmrel\ (\lambda x\ y. f\ x\ y \wedge \neg f\ y\ x \vee x = y)\ ym\ xm$   
**by** (*metis (mono-tags, lifting) fmrel-code fmrel-on-fset-alt-def*)  
**thus**  $fmlookup\ xm\ x = fmlookup\ ym\ x$   
**apply** (*erule-tac ?x=x in fmrel-cases*)  
**by** (*erule-tac ?x=x in fmrel-cases, auto*)**+**

qed

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

*strict-subtuple*  $(\lambda x y. f x y \vee x = y) xm ym \implies$   
*strict-subtuple*  $(\lambda x y. f x y \wedge \neg f y x \vee x = y) ym xm \implies \text{False}$   
**by** (*auto simp add: subtuple-antisym*)

**lemma** *subtuple-acyclic*:

**assumes** *acyclicP-on*  $(fmrans' xm) P$   
**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** (*rule fmap-ext*)  
**fix**  $x$   
**from** *assms* **have** *fmdom-eq*:  $fmdom xm = fmdom ym$   
**using** *subtuple-fmdom* **by** *blast*  
**have**  $\bigwedge x a b. \text{acyclicP-on } (fmrans' xm) P \implies$   
 $fmllookup xm x = \text{Some } a \implies$   
 $fmllookup ym x = \text{Some } b \implies$   
 $P^{**} a b \implies P b a \vee a = b \implies a = b$   
**by** (*meson Nitpick.tranclp-unfold fmrans'I rtranclp-into-tranclp1*)  
**moreover from** *fmdom-eq assms(2)* **have** *fmrel*  $P^{**} xm ym$   
**unfolding** *fmrel-on-fset-fmrel-restrict* **apply** *auto*  
**by** (*metis fmrestrict-fset-dom*)  
**moreover from** *fmdom-eq assms(3)* **have** *fmrel*  $(\lambda x y. P x y \vee x = y) ym xm$   
**unfolding** *fmrel-on-fset-fmrel-restrict* **apply** *auto*  
**by** (*metis fmrestrict-fset-dom*)  
**ultimately show**  $fmllookup xm x = fmllookup ym x$   
**apply** (*erule-tac ?x=x in fmrel-cases*)  
**apply** (*erule-tac ?x=x in fmrel-cases, simp-all*)  
**using** *assms(1)* **by** *blast*

qed

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

**assumes** *acyclicP-on*  $(fmrans' ym) P$   
**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** (*rule fmap-ext*)  
**fix**  $x$   
**from** *assms* **have** *fmdom-eq*:  $fmdom xm = fmdom ym$   
**using** *subtuple-fmdom* **by** *blast*  
**have**  $\bigwedge x a b. \text{acyclicP-on } (fmrans' ym) P \implies$   
 $fmllookup xm x = \text{Some } a \implies$   
 $fmllookup ym x = \text{Some } b \implies$   
 $P^{**} a b \implies P b a \vee a = b \implies a = b$   
**by** (*meson Nitpick.tranclp-unfold fmrans'I rtranclp-into-tranclp2*)  
**moreover from** *fmdom-eq assms(2)* **have** *fmrel*  $P^{**} xm ym$   
**unfolding** *fmrel-on-fset-fmrel-restrict* **apply** *auto*

by (metis fmrestrict-fset-dom)  
**moreover from** *fmdom-eq assms(3)* **have**  $\text{fmrel } (\lambda x y. P x y \vee x = y) \text{ } ym \text{ } xm$   
**unfolding** *fmrel-on-fset-fmrel-restrict* **apply** *auto*  
 by (metis fmrestrict-fset-dom)  
**ultimately show**  $\text{fmlookup } xm \text{ } x = \text{fmlookup } ym \text{ } x$   
**apply** (*erule-tac ?x= x in fmrel-cases*)  
**apply** (*erule-tac ?x= x in fmrel-cases, simp-all*)  
**using** *assms(1)* **by** *blast*  
**qed**

**lemma** *subtuple-acyclic''*:  
 $\text{acyclicP-on } (\text{fmran}' ym) R \implies$   
 $\text{subtuple } R^{**} \text{ } xm \text{ } ym \implies$   
 $\text{subtuple } R \text{ } ym \text{ } xm \implies$   
 $xm = ym$   
**by** (*metis (no-types, lifting) subtuple-acyclic' subtuple-mono tranclp-eq-rtranclp*)

**lemma** *strict-subtuple-trans*:  
 $\text{acyclicP-on } (\text{fmran}' xm) P \implies$   
 $\text{strict-subtuple } (\lambda x y. P x y \vee x = y)^{++} \text{ } xm \text{ } ym \implies$   
 $\text{strict-subtuple } (\lambda x y. P x y \vee x = y) \text{ } ym \text{ } zm \implies$   
 $\text{strict-subtuple } (\lambda x y. P x y \vee x = y)^{++} \text{ } xm \text{ } zm$   
**apply** *auto*  
**apply** (*rule fmrel-on-fset-trans, auto*)  
**by** (*drule-tac ?ym= ym in subtuple-acyclic; auto*)

**lemma** *strict-subtuple-trans'*:  
 $\text{acyclicP-on } (\text{fmran}' zm) P \implies$   
 $\text{strict-subtuple } (\lambda x y. P x y \vee x = y) \text{ } xm \text{ } ym \implies$   
 $\text{strict-subtuple } (\lambda x y. P x y \vee x = y)^{++} \text{ } ym \text{ } zm \implies$   
 $\text{strict-subtuple } (\lambda x y. P x y \vee x = y)^{++} \text{ } xm \text{ } zm$   
**apply** *auto*  
**apply** (*rule fmrel-on-fset-trans, auto*)  
**by** (*drule-tac ?xm= ym in subtuple-acyclic'; auto*)

**lemma** *strict-subtuple-trans''*:  
 $\text{acyclicP-on } (\text{fmran}' zm) R \implies$   
 $\text{strict-subtuple } R \text{ } xm \text{ } ym \implies$   
 $\text{strict-subtuple } R^{**} \text{ } ym \text{ } zm \implies$   
 $\text{strict-subtuple } R^{**} \text{ } xm \text{ } zm$   
**apply** *auto*  
**apply** (*rule fmrel-on-fset-trans, auto*)  
**by** (*drule-tac ?xm= ym in subtuple-acyclic''; auto*)

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

**apply** *auto*  
**apply** (*rule fmrel-on-fset-trans, auto*)  
**by** (*drule-tac ?xm= ym in subtuple-acyclic'; auto*)

**lemma** *subtuple-fmmerge2 [intro]:*  
 $(\wedge x y. x \in \text{fmran}'\ xm \implies f\ x\ (g\ x\ y)) \implies$   
 $\text{subtuple}\ f\ xm\ (\text{fmmerge}\ g\ xm\ ym)$   
**by** (*rule-tac ?S= fmdom ym in fmrel-on-fsubset; auto*)

#### 1.4.4 Transitive Closures

**lemma** *trancl-to-subtuple:*  
 $(\text{subtuple}\ r)^{++}\ xm\ ym \implies$   
 $\text{subtuple}\ r^{++}\ xm\ ym$   
**apply** (*induct rule: tranclp-induct*)  
**apply** (*metis subtuple-mono tranclp.r-into-trancl*)  
**by** (*rule fmrel-on-fset-trans; auto*)

**lemma** *rtrancl-to-subtuple:*  
 $(\text{subtuple}\ r)^{**}\ xm\ ym \implies$   
 $\text{subtuple}\ r^{**}\ xm\ ym$   
**apply** (*induct rule: rtranclp-induct*)  
**apply** (*simp add: fmap.rel-refl-strong fmrel-to-subtuple*)  
**by** (*rule fmrel-on-fset-trans; auto*)

**lemma** *fmrel-to-subtuple-trancl:*  
 $\text{reflp}\ r \implies$   
 $(\text{fmrel}\ r)^{++}\ (\text{fmrestrict-fset}\ (\text{fmdom}\ ym)\ xm)\ ym \implies$   
 $(\text{subtuple}\ r)^{++}\ xm\ ym$   
**apply** (*frule trancl-to-fmrel*)  
**apply** (*rule-tac ?r= r in fmrel-tranclp-induct, auto*)  
**apply** (*metis (no-types, lifting) fmrel-fmdom-eq*  
 $\text{subtuple-eq-fmrel-fmrestrict-fset}\ \text{tranclp.r-into-trancl}$ )  
**by** (*meson fmrel-to-subtuple tranclp.simps*)

**lemma** *subtuple-to-trancl:*  
 $\text{reflp}\ r \implies$   
 $\text{subtuple}\ r^{++}\ xm\ ym \implies$   
 $(\text{subtuple}\ r)^{++}\ xm\ ym$   
**apply** (*rule fmrel-to-subtuple-trancl*)  
**unfolding** *fmrel-on-fset-fmrel-restrict*  
**by** (*simp-all add: fmrel-to-trancl*)

**lemma** *trancl-to-strict-subtuple:*  
 $\text{acyclicP-on}\ (\text{fmran}'\ ym)\ R \implies$   
 $(\text{strict-subtuple}\ R)^{++}\ xm\ ym \implies$   
 $\text{strict-subtuple}\ R^{**}\ xm\ ym$   
**apply** (*erule converse-tranclp-induct*)  
**apply** (*metis r-into-rtranclp strict-subtuple-mono*)

using *strict-subtuple-trans''* by *blast*

**lemma** *trancl-to-strict-subtuple'*:

*acyclicP-on* (*f**mran'* *ym*) *R*  $\implies$   
 $(\text{strict-subtuple } (\lambda x y. R x y \vee x = y))^{++} xm ym \implies$   
 $\text{strict-subtuple } (\lambda x y. R x y \vee x = y)^{**} xm ym$   
**apply** (*erule converse-tranclp-induct*)  
**apply** (*metis* (*no-types*, *lifting*) *r-into-rtranclp strict-subtuple-mono*)  
using *strict-subtuple-trans'''* by *blast*

**lemma** *subtuple-rtranclp-intro*:

**assumes**  $\bigwedge xm ym. R (f xm) (f ym) \implies \text{subtuple } R xm ym$   
**and** *bij-on-trancl* *R f*  
**and**  $R^{**} (f xm) (f ym)$   
**shows**  $\text{subtuple } R^{**} xm ym$

**proof** –

**have**  $(\lambda xm ym. R (f xm) (f ym))^{**} xm ym$   
**apply** (*insert assms*(2) *assms*(3))  
**by** (*rule reflect-rtranclp*; *auto*)  
**with** *assms*(1) **have**  $(\text{subtuple } R)^{**} xm ym$   
**by** (*metis* (*mono-tags*, *lifting*) *mono-rtranclp*)  
**hence**  $\text{subtuple } R^{**} xm ym$   
**by** (*rule trancl-to-subtuple*)  
**thus** *?thesis* by *simp*

**qed**

**lemma** *strict-subtuple-rtranclp-intro*:

**assumes**  $\bigwedge xm ym. R (f xm) (f ym) \implies$   
 $\text{strict-subtuple } (\lambda x y. R x y \vee x = y) xm ym$   
**and** *bij-on-trancl* *R f*  
**and** *acyclicP-on* (*f**mran'* *ym*) *R*  
**and**  $R^{++} (f xm) (f ym)$   
**shows**  $\text{strict-subtuple } R^{**} xm ym$

**proof** –

**have**  $(\lambda xm ym. R (f xm) (f ym))^{++} xm ym$   
**apply** (*insert assms*(1) *assms*(2) *assms*(4))  
**by** (*rule reflect-tranclp*; *auto*)  
**hence**  $(\text{strict-subtuple } (\lambda x y. R x y \vee x = y))^{++} xm ym$   
**by** (*rule tranclp-trans-induct*;  
*auto simp add: assms*(1) *tranclp.r-into-trancl*)  
**with** *assms*(3) **have**  $\text{strict-subtuple } (\lambda x y. R x y \vee x = y)^{**} xm ym$   
**by** (*rule trancl-to-strict-subtuple'*)  
**thus** *?thesis* by *simp*

**qed**

### 1.4.5 Code Setup

**abbreviation** *subtuple-fun* *f xm ym*  $\equiv$

*fBall* (*f**mdom* *ym*) ( $\lambda x. \text{rel-option } f (f\text{mlookup } xm x) (f\text{mlookup } ym x)$ )



**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*  $f\ xm\ ym = \text{subtuple } f\ xm\ ym$   
**by** (*simp add: fmrel-on-fset-alt-def*)

**lemma** *strict-subtuple-fun-simp* [*code-abbrev*, *simp*]:  
*strict-subtuple-fun*  $f\ xm\ ym = \text{strict-subtuple } f\ xm\ ym$   
**by** *simp*

**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* | *Out* | *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-result* :: (*'t*, *'e*) *oper-spec*  $\Rightarrow$  *'t*  $\equiv$  *fst*  $\circ$  *snd*  $\circ$  *snd*  $\circ$  *snd*

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

**definition**  $param-name :: 't param-spec \Rightarrow param \equiv fst$   
**definition**  $param-type :: 't param-spec \Rightarrow 't \equiv fst \circ snd$   
**definition**  $param-dir :: 't param-spec \Rightarrow param-dir \equiv snd \circ snd$

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

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

## 1.5.2 Attributes

**inductive**  $owned-attribute'$  **where**  
 $C \in | fdom\ attributes \Longrightarrow$   
 $fmlookup\ attributes\ C = Some\ attrsc \Longrightarrow$   
 $fmlookup\ attrsc\ attr = Some\ \tau \Longrightarrow$   
 $owned-attribute'\ attributes\ C\ attr\ \tau$

**inductive**  $attribute-not-closest$  **where**  
 $owned-attribute'\ attributes\ \mathcal{D}'\ attr\ \tau' \Longrightarrow$   
 $C \leq \mathcal{D}' \Longrightarrow$   
 $\mathcal{D}' < \mathcal{D} \Longrightarrow$   
 $attribute-not-closest\ attributes\ C\ attr\ \mathcal{D}\ \tau$

**inductive**  $closest-attribute$  **where**  
 $owned-attribute'\ attributes\ \mathcal{D}\ attr\ \tau \Longrightarrow$   
 $C \leq \mathcal{D} \Longrightarrow$   
 $\neg\ attribute-not-closest\ attributes\ C\ attr\ \mathcal{D}\ \tau \Longrightarrow$   
 $closest-attribute\ attributes\ C\ attr\ \mathcal{D}\ \tau$

**inductive**  $closest-attribute-not-unique$  **where**  
 $closest-attribute\ attributes\ C\ attr\ \mathcal{D}'\ \tau' \Longrightarrow$   
 $\mathcal{D} \neq \mathcal{D}' \vee \tau \neq \tau' \Longrightarrow$   
 $closest-attribute-not-unique\ attributes\ C\ attr\ \mathcal{D}\ \tau$

**inductive**  $unique-closest-attribute$  **where**  
 $closest-attribute\ attributes\ C\ attr\ \mathcal{D}\ \tau \Longrightarrow$   
 $\neg\ closest-attribute-not-unique\ attributes\ C\ attr\ \mathcal{D}\ \tau \Longrightarrow$   
 $unique-closest-attribute\ attributes\ C\ attr\ \mathcal{D}\ \tau$

## 1.5.3 Association Ends

**inductive**  $role-refer-class$  **where**  
 $role \in | fdom\ ends \Longrightarrow$   
 $fmlookup\ ends\ role = Some\ end \Longrightarrow$   
 $assoc-end-class\ end = C \Longrightarrow$

*role-refer-class ends C role*

**inductive association-ends' where**

*C |∈| classes  $\implies$   
 assoc |∈| fmdom associations  $\implies$   
 fmllookup associations assoc = Some ends  $\implies$   
 role-refer-class ends C from  $\implies$   
 role |∈| fmdom ends  $\implies$   
 fmllookup 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**

$fmlookup\ association\ classes\ \mathcal{A} = Some\ assoc \implies$   
 $fmlookup\ associations\ assoc = Some\ ends \implies$   
 $role\ refer\ class\ ends\ \mathcal{C}\ from \implies$   
 $referred\ by\ association\ class''\ association\ classes\ associations\ \mathcal{C}\ from\ \mathcal{A}$

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

$referred\ by\ association\ class''\ association\ classes\ associations\ \mathcal{C}\ from\ \mathcal{A} \implies$   
 $referred\ by\ association\ class''\ association\ classes\ associations\ \mathcal{C}\ None\ \mathcal{A}$   
 $|$   $referred\ by\ association\ class''\ association\ classes\ associations\ \mathcal{C}\ from\ \mathcal{A} \implies$   
 $referred\ by\ association\ class''\ association\ classes\ associations\ \mathcal{C}\ (Some\ from)\ \mathcal{A}$

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

$referred\ by\ association\ class''\ association\ classes\ associations\ \mathcal{D}'\ from\ \mathcal{A} \implies$   
 $\mathcal{C} \leq \mathcal{D}' \implies$   
 $\mathcal{D}' < \mathcal{D} \implies$   
 $association\ class\ not\ closest\ association\ classes\ associations\ \mathcal{C}\ from\ \mathcal{A}\ \mathcal{D}$

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

$referred\ by\ association\ class''\ association\ classes\ associations\ \mathcal{D}\ from\ \mathcal{A} \implies$   
 $\mathcal{C} \leq \mathcal{D} \implies$   
 $\neg association\ class\ not\ closest\ association\ classes\ associations\ \mathcal{C}\ from\ \mathcal{A}\ \mathcal{D} \implies$   
 $closest\ association\ class\ association\ classes\ associations\ \mathcal{C}\ from\ \mathcal{A}\ \mathcal{D}$

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

$closest\ association\ class\ association\ classes\ associations\ \mathcal{C}\ from\ \mathcal{A}\ \mathcal{D}' \implies$   
 $\mathcal{D} \neq \mathcal{D}' \implies$   
 $closest\ association\ class\ not\ unique$   
 $association\ classes\ associations\ \mathcal{C}\ from\ \mathcal{A}\ \mathcal{D}$

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

$closest\ association\ class\ association\ classes\ associations\ \mathcal{C}\ from\ \mathcal{A}\ \mathcal{D} \implies$   
 $\neg closest\ association\ class\ not\ unique$   
 $association\ classes\ associations\ \mathcal{C}\ from\ \mathcal{A}\ \mathcal{D} \implies$   
 $unique\ closest\ association\ class\ association\ classes\ associations\ \mathcal{C}\ from\ \mathcal{A}\ \mathcal{D}$

### 1.5.5 Association Class Ends

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

$fmlookup\ association\ classes\ \mathcal{A} = Some\ assoc \implies$   
 $fmlookup\ associations\ assoc = Some\ ends \implies$   
 $fmlookup\ ends\ role = Some\ end \implies$   
 $association\ class\ end''\ association\ classes\ associations\ \mathcal{A}\ role\ end$

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

$association\ class\ end''\ association\ classes\ associations\ \mathcal{A}\ role\ end' \implies$   
 $end \neq end' \implies$   
 $association\ class\ end\ not\ unique\ association\ classes\ associations\ \mathcal{A}\ role\ end$

**inductive** *unique-association-class-end* **where**

*association-class-end'* *association-classes* *associations*  $\mathcal{A}$  *role end*  $\implies$   
 $\neg$  *association-class-end-not-unique*  
*association-classes* *associations*  $\mathcal{A}$  *role end*  $\implies$   
*unique-association-class-end* *association-classes* *associations*  $\mathcal{A}$  *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>  $\implies \text{end}_1 = \text{end}_2$

**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*  $\mathcal{D}_2$   $\tau_2 \implies \mathcal{D}_1 = \mathcal{D}_2 \wedge \tau_1 = \tau_2$   
**by** (*meson closest-attribute-not-unique.intros unique-closest-attribute.cases*)

**lemma** (**in** *object-model*) *attribute-self-or-inherited*:  
*attribute*  $\mathcal{C}$  *attr*  $\mathcal{D}$   $\tau \implies \mathcal{C} \leq \mathcal{D}$   
**by** (*meson closest-attribute.cases unique-closest-attribute.cases*)

**lemma** (**in** *object-model*) *attribute-closest*:  
*attribute*  $\mathcal{C}$  *attr*  $\mathcal{D}$   $\tau \implies$   
*owned-attribute*  $\mathcal{D}'$  *attr*  $\tau \implies$   
 $\mathcal{C} \leq \mathcal{D}' \implies \neg \mathcal{D}' < \mathcal{D}$   
**by** (*meson attribute-not-closest.intros closest-attribute.cases*  
*unique-closest-attribute.cases*)

**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* $_2 \implies \mathcal{D}_1 = \mathcal{D}_2 \wedge \text{end}_1 = \text{end}_2$   
**by** (*meson closest-association-end-not-unique.intros*  
*unique-closest-association-end.cases*)

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

*association-end C from role D end*  $\implies C \leq D$   
**by** (*meson closest-association-end.cases unique-closest-association-end.cases*)

**lemma** (*in object-model*) *association-end-closest*:  
*association-end C from role D end*  $\implies$   
*owned-association-end D' from role end*  $\implies$   
 $C \leq D' \implies \neg D' < D$   
**by** (*meson association-end-not-closest.intros closest-association-end.cases*  
*unique-closest-association-end.cases*)

**lemma** (*in object-model*) *association-class-end-det*:  
*association-class-end A role end<sub>1</sub>*  $\implies$   
*association-class-end A role end<sub>2</sub>*  $\implies end_1 = end_2$   
**by** (*meson association-class-end-not-unique.intros unique-association-class-end.cases*)

**lemma** (*in object-model*) *operation-det*:  
*operation  $\tau$  name  $\pi$  oper<sub>1</sub>*  $\implies$   
*operation  $\tau$  name  $\pi$  oper<sub>2</sub>*  $\implies oper_1 = oper_2$   
**by** (*meson operation-not-unique.intros unique-operation.cases*)

**lemma** (*in object-model*) *static-operation-det*:  
*static-operation  $\tau$  name  $\pi$  oper<sub>1</sub>*  $\implies$   
*static-operation  $\tau$  name  $\pi$  oper<sub>2</sub>*  $\implies oper_1 = oper_2$   
**by** (*meson static-operation-not-unique.intros unique-static-operation.cases*)

### 1.5.10 Code Setup

**declare** *owned-attribute'.intros*[*folded Predicate-Compile.contains-def, code-pred-intro*]  
**code-pred** (*modes*:  
*i*  $\implies$  *i*  $\implies$  *i*  $\implies$  *i*  $\implies$  *bool*,  
*i*  $\implies$  *i*  $\implies$  *i*  $\implies$  *o*  $\implies$  *bool*,  
*i*  $\implies$  *o*  $\implies$  *i*  $\implies$  *i*  $\implies$  *bool*,  
*i*  $\implies$  *o*  $\implies$  *i*  $\implies$  *o*  $\implies$  *bool*) *owned-attribute'*  
**by** (*elim owned-attribute'.cases*) (*simp add: Predicate-Compile.contains-def*)

**code-pred** *unique-closest-attribute* .

**declare** *role-refer-class.intros*[*folded Predicate-Compile.contains-def, code-pred-intro*]  
**code-pred** (*modes*:  
*i*  $\implies$  *i*  $\implies$  *i*  $\implies$  *bool*,  
*i*  $\implies$  *i*  $\implies$  *o*  $\implies$  *bool*,  
*i*  $\implies$  *o*  $\implies$  *i*  $\implies$  *bool*,  
*i*  $\implies$  *o*  $\implies$  *o*  $\implies$  *bool*) *role-refer-class*  
**by** (*elim role-refer-class.cases*) (*simp add: Predicate-Compile.contains-def*)

**declare** *association-ends'.intros*[*folded Predicate-Compile.contains-def, code-pred-intro*]  
**code-pred** (*modes*:





```

i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ o ⇒ i ⇒ 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 .

```

**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 .

```

**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 .

```

**code-pred** *unique-closest-association-class* .

**code-pred** *association-class-end'* .

**code-pred** *association-class-end-not-unique* .

**code-pred** *unique-association-class-end* .

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

**code-pred** [*show-modes*] *any-operation'*

by (*elim any-operation'.cases*) (*simp add: Predicate-Compile.contains-def*)

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

```

i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool) operation' .

```

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

```

i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool) operation-not-unique .

```

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

```

i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool) unique-operation .

```

**code-pred** *operation-defined'* .

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

```

i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool) static-operation' .

```

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

$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}$ ) *static-operation-not-unique* .

**code-pred** (*modes*:  
 $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}$ ,  
 $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$ ) *unique-static-operation* .

**code-pred** *static-operation-defined'* .

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

**code-pred** (*modes*:  
 $i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}$ ,  $i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$ ) *has-literal'*  
**by** (*elim has-literal'.cases*) (*simp add: Predicate-Compile.contains-def*)

**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*:

```
 $\tau \sqsubset_B \sigma \implies \sigma \sqsubset_B \tau \implies \text{False}$ 
by (induct rule: basic-subtype.induct, auto)
```

## 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** –

```
have basic-subtype++ OclVoid OclAny
by (rule-tac ?b= Boolean in tranclp.trancl-into-trancl; auto)
moreover have basic-subtype++ Integer OclAny
by (rule-tac ?b= Real in tranclp.trancl-into-trancl; auto)
moreover hence basic-subtype++ UnlimitedNatural OclAny
by (rule-tac ?b= Integer in tranclp-into-tranclp2; auto)
ultimately show  $\tau \neq \text{OclAny} \implies \tau < \text{OclAny}$ 
unfolding less-basic-type-def
by (induct  $\tau$ , auto)
```

**qed**

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

$\tau \neq \text{OclVoid} \implies \text{OclVoid} < \tau$   
**proof** –  
**have** *basic-subtype<sup>++</sup> OclVoid OclAny*  
**by** (*rule-tac ?b= Boolean in tranclp.trancl-into-trancl; auto*)  
**moreover have** *basic-subtype<sup>++</sup> OclVoid Integer*  
**by** (*rule-tac ?b= UnlimitedNatural in tranclp.trancl-into-trancl; auto*)  
**moreover hence** *basic-subtype<sup>++</sup> OclVoid Real*  
**by** (*rule-tac ?b= Integer in tranclp.trancl-into-trancl; auto*)  
**ultimately show**  $\tau \neq \text{OclVoid} \implies \text{OclVoid} < \tau$   
**unfolding** *less-basic-type-def*  
**by** (*induct  $\tau$ ; auto*)  
**qed**

**lemma** *type-less-x-Real-intro [intro]:*  
 $\tau = \text{UnlimitedNatural} \implies \tau < \text{Real}$   
 $\tau = \text{Integer} \implies \tau < \text{Real}$   
**unfolding** *less-basic-type-def*  
**by** (*rule rtranclp-into-tranclp2, auto*)

**lemma** *type-less-x-Integer-intro [intro]:*  
 $\tau = \text{UnlimitedNatural} \implies \tau < \text{Integer}$   
**unfolding** *less-basic-type-def*  
**by** (*rule rtranclp-into-tranclp2, auto*)

**lemma** *type-less-x-ObjectType-intro [intro]:*  
 $\tau = \langle \mathcal{C} \rangle_{\mathcal{T}} \implies \mathcal{C} < \mathcal{D} \implies \tau < \langle \mathcal{D} \rangle_{\mathcal{T}}$   
**unfolding** *less-basic-type-def*  
**using** *dual-order.order-iff-strict* **by** *blast*

### 2.2.2 Strict Elimination Rules

**lemma** *type-less-x-OclAny [elim!]:*  
 $\tau < \text{OclAny} \implies$   
 $(\tau = \text{OclVoid} \implies P) \implies$   
 $(\tau = \text{Boolean} \implies P) \implies$   
 $(\tau = \text{Integer} \implies P) \implies$   
 $(\tau = \text{UnlimitedNatural} \implies P) \implies$   
 $(\tau = \text{Real} \implies P) \implies$   
 $(\tau = \text{String} \implies P) \implies$   
 $(\bigwedge \mathcal{E}. \tau = \text{Enum } \mathcal{E} \implies P) \implies$   
 $(\bigwedge \mathcal{C}. \tau = \langle \mathcal{C} \rangle_{\mathcal{T}} \implies P) \implies P$   
**unfolding** *less-basic-type-def*  
**by** (*induct rule: converse-tranclp-induct; auto*)

**lemma** *type-less-x-OclVoid [elim!]:*  
 $\tau < \text{OclVoid} \implies P$   
**unfolding** *less-basic-type-def*  
**by** (*induct rule: converse-tranclp-induct; auto*)

**lemma** *type-less-x-Boolean* [elim!]:  
 $\tau < \text{Boolean} \implies$   
 $(\tau = \text{OclVoid} \implies P) \implies P$   
**unfolding** *less-basic-type-def*  
**by** (*induct rule: converse-tranclp-induct; auto*)

**lemma** *type-less-x-Real* [elim!]:  
 $\tau < \text{Real} \implies$   
 $(\tau = \text{OclVoid} \implies P) \implies$   
 $(\tau = \text{UnlimitedNatural} \implies P) \implies$   
 $(\tau = \text{Integer} \implies P) \implies P$   
**unfolding** *less-basic-type-def*  
**by** (*induct rule: converse-tranclp-induct; auto*)

**lemma** *type-less-x-Integer* [elim!]:  
 $\tau < \text{Integer} \implies$   
 $(\tau = \text{OclVoid} \implies P) \implies$   
 $(\tau = \text{UnlimitedNatural} \implies P) \implies P$   
**unfolding** *less-basic-type-def*  
**by** (*induct rule: converse-tranclp-induct; auto*)

**lemma** *type-less-x-UnlimitedNatural* [elim!]:  
 $\tau < \text{UnlimitedNatural} \implies$   
 $(\tau = \text{OclVoid} \implies P) \implies P$   
**unfolding** *less-basic-type-def*  
**by** (*induct rule: converse-tranclp-induct; auto*)

**lemma** *type-less-x-String* [elim!]:  
 $\tau < \text{String} \implies$   
 $(\tau = \text{OclVoid} \implies P) \implies P$   
**unfolding** *less-basic-type-def*  
**by** (*induct rule: converse-tranclp-induct; auto*)

**lemma** *type-less-x-ObjectType* [elim!]:  
 $\tau < \langle \mathcal{D} \rangle_{\tau} \implies$   
 $(\tau = \text{OclVoid} \implies P) \implies$   
 $(\bigwedge \mathcal{C}. \tau = \langle \mathcal{C} \rangle_{\tau} \implies \mathcal{C} < \mathcal{D} \implies P) \implies P$   
**unfolding** *less-basic-type-def*  
**apply** (*induct rule: converse-tranclp-induct*)  
**apply** *auto[1]*  
**using** *less-trans* **by** *auto*

**lemma** *type-less-x-Enum* [elim!]:  
 $\tau < \text{Enum } \mathcal{E} \implies$   
 $(\tau = \text{OclVoid} \implies P) \implies P$   
**unfolding** *less-basic-type-def*  
**by** (*induct rule: converse-tranclp-induct; auto*)



### 2.2.3 Properties

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

$\tau < \tau \implies \text{False}$   
**for**  $\tau :: 'a \text{ basic-type}$   
**by** (*cases*  $\tau$ ; *auto*)

**lemma** *tranclp-less-basic-type*:

$(\tau, \sigma) \in \{(\tau, \sigma). \tau \sqsubseteq_B \sigma\}^+ \iff \tau < \sigma$   
**by** (*simp add: tranclp-unfold less-basic-type-def*)

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

*acyclicP basic-subtype*  
**apply** (*rule acyclicI*)  
**using** *OCL-Basic-Types.basic-subtype-irrefl*  
*OCL-Basic-Types.tranclp-less-basic-type* **by** *auto*

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

$\tau < \sigma \iff \tau \leq \sigma \wedge \neg \sigma \leq \tau$   
**for**  $\tau \sigma :: 'a \text{ basic-type}$   
**unfolding** *less-basic-type-def less-eq-basic-type-def*  
**apply** (*rule iffI; auto*)  
**apply** (*metis (mono-tags) basic-subtype-irrefl*  
*less-basic-type-def tranclp-rtranclp-tranclp*)  
**by** (*drule rtranclpD; auto*)

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

$\tau \leq \sigma \implies \sigma \leq \tau \implies \tau = \sigma$   
**for**  $\tau \sigma :: 'a \text{ basic-type}$   
**unfolding** *less-eq-basic-type-def less-basic-type-def*  
**by** (*metis (mono-tags, lifting) less-eq-basic-type-def*  
*less-le-not-le-basic-type less-basic-type-def rtranclpD*)

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

$\tau \leq \tau$   
**for**  $\tau :: 'a \text{ basic-type}$   
**by** (*simp add: less-eq-basic-type-def*)

**instance**

**by** *standard (auto simp add: less-eq-basic-type-def*  
*less-le-not-le-basic-type antisym-basic-type)*

**end**

### 2.2.4 Non-Strict Introduction Rules

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

$\tau \leq \text{OclAny}$   
**using** *order.order-iff-strict* **by** *auto*

**lemma** *type-less-eq-OclVoid-x-intro* [intro]:  
 $OclVoid \leq \tau$   
**using** *order.order-iff-strict* **by** *auto*

**lemma** *type-less-eq-x-Real-intro* [intro]:  
 $\tau = UnlimitedNatural \implies \tau \leq Real$   
 $\tau = Integer \implies \tau \leq Real$   
**using** *order.order-iff-strict* **by** *auto*

**lemma** *type-less-eq-x-Integer-intro* [intro]:  
 $\tau = UnlimitedNatural \implies \tau \leq Integer$   
**using** *order.order-iff-strict* **by** *auto*

**lemma** *type-less-eq-x-ObjectType-intro* [intro]:  
 $\tau = \langle C \rangle_{\tau} \implies C \leq D \implies \tau \leq \langle D \rangle_{\tau}$   
**using** *order.order-iff-strict* **by** *fastforce*

## 2.2.5 Non-Strict Elimination Rules

**lemma** *type-less-eq-x-OclAny* [elim!]:  
 $\tau \leq OclAny \implies$   
 $(\tau = OclVoid \implies P) \implies$   
 $(\tau = OclAny \implies P) \implies$   
 $(\tau = Boolean \implies P) \implies$   
 $(\tau = Integer \implies P) \implies$   
 $(\tau = UnlimitedNatural \implies P) \implies$   
 $(\tau = Real \implies P) \implies$   
 $(\tau = String \implies P) \implies$   
 $(\bigwedge \mathcal{E}. \tau = Enum \ \mathcal{E} \implies P) \implies$   
 $(\bigwedge \mathcal{C}. \tau = \langle C \rangle_{\tau} \implies P) \implies P$   
**by** (*drule le-imp-less-or-eq; auto*)

**lemma** *type-less-eq-x-OclVoid* [elim!]:  
 $\tau \leq OclVoid \implies$   
 $(\tau = OclVoid \implies P) \implies P$   
**by** (*drule le-imp-less-or-eq; auto*)

**lemma** *type-less-eq-x-Boolean* [elim!]:  
 $\tau \leq Boolean \implies$   
 $(\tau = OclVoid \implies P) \implies$   
 $(\tau = Boolean \implies P) \implies P$   
**by** (*drule le-imp-less-or-eq; auto*)

**lemma** *type-less-eq-x-Real* [elim!]:  
 $\tau \leq Real \implies$   
 $(\tau = OclVoid \implies P) \implies$   
 $(\tau = UnlimitedNatural \implies P) \implies$   
 $(\tau = Integer \implies P) \implies$   
 $(\tau = Real \implies P) \implies P$

**by** (*drule le-imp-less-or-eq; auto*)

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

$\tau \leq \text{Integer} \implies$   
 $(\tau = \text{OclVoid} \implies P) \implies$   
 $(\tau = \text{UnlimitedNatural} \implies P) \implies$   
 $(\tau = \text{Integer} \implies P) \implies P$   
**by** (*drule le-imp-less-or-eq; auto*)

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

$\tau \leq \text{UnlimitedNatural} \implies$   
 $(\tau = \text{OclVoid} \implies P) \implies$   
 $(\tau = \text{UnlimitedNatural} \implies P) \implies P$   
**by** (*drule le-imp-less-or-eq; auto*)

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

$\tau \leq \text{String} \implies$   
 $(\tau = \text{OclVoid} \implies P) \implies$   
 $(\tau = \text{String} \implies P) \implies P$   
**by** (*drule le-imp-less-or-eq; auto*)

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

$\tau \leq \langle \mathcal{D} \rangle_{\tau} \implies$   
 $(\tau = \text{OclVoid} \implies P) \implies$   
 $(\bigwedge \mathcal{C}. \tau = \langle \mathcal{C} \rangle_{\tau} \implies \mathcal{C} \leq \mathcal{D} \implies P) \implies P$   
**by** (*drule le-imp-less-or-eq; auto*)

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

$\tau \leq \text{Enum } \mathcal{E} \implies$   
 $(\tau = \text{OclVoid} \implies P) \implies$   
 $(\tau = \text{Enum } \mathcal{E} \implies P) \implies P$   
**by** (*drule le-imp-less-or-eq; auto*)

### 2.2.6 Simplification Rules

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

$\text{OclAny} < \sigma = \text{False}$   
 $\text{OclVoid} < \sigma = (\sigma \neq \text{OclVoid})$   
 $\text{Boolean} < \sigma = (\sigma = \text{OclAny})$   
 $\text{Real} < \sigma = (\sigma = \text{OclAny})$   
 $\text{Integer} < \sigma = (\sigma = \text{OclAny} \vee \sigma = \text{Real})$   
 $\text{UnlimitedNatural} < \sigma = (\sigma = \text{OclAny} \vee \sigma = \text{Real} \vee \sigma = \text{Integer})$   
 $\text{String} < \sigma = (\sigma = \text{OclAny})$   
 $\text{ObjectType } \mathcal{C} < \sigma = (\exists \mathcal{D}. \sigma = \text{OclAny} \vee \sigma = \text{ObjectType } \mathcal{D} \wedge \mathcal{C} < \mathcal{D})$   
 $\text{Enum } \mathcal{E} < \sigma = (\sigma = \text{OclAny})$   
**by** (*induct*  $\sigma$ , *auto*)

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

$\tau < \text{OclAny} = (\tau \neq \text{OclAny})$

```

 $\tau < \text{OclVoid} = \text{False}$ 
 $\tau < \text{Boolean} = (\tau = \text{OclVoid})$ 
 $\tau < \text{Real} = (\tau = \text{Integer} \vee \tau = \text{UnlimitedNatural} \vee \tau = \text{OclVoid})$ 
 $\tau < \text{Integer} = (\tau = \text{UnlimitedNatural} \vee \tau = \text{OclVoid})$ 
 $\tau < \text{UnlimitedNatural} = (\tau = \text{OclVoid})$ 
 $\tau < \text{String} = (\tau = \text{OclVoid})$ 
 $\tau < \text{ObjectType } \mathcal{D} = (\exists \mathcal{C}. \tau = \text{ObjectType } \mathcal{C} \wedge \mathcal{C} < \mathcal{D} \vee \tau = \text{OclVoid})$ 
 $\tau < \text{Enum } \mathcal{E} = (\tau = \text{OclVoid})$ 
by auto

```

## 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*:

```

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

```

**apply** (induct  $\sigma$ ; simp add: basic-subtype.simps  
less-eq-basic-type-def r-into-rtranclp)

**by** (metis Nitpick.rtranclp-unfold basic-subtype.intros(8)  
le-imp-less-or-eq r-into-rtranclp sup-ge1)

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

```

 $\tau \leq \tau \sqcup \sigma$ 

```

**for**  $\tau \sigma :: 'a$  basic-type

**apply** (induct  $\tau$ , auto)

**using** *sup-ge1-ObjectType* **by** *auto*

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

```

 $\tau \sqcup \sigma = \sigma \sqcup \tau$ 

```

**for**  $\tau \sigma :: 'a$  basic-type

**by** (induct  $\tau$ ; induct  $\sigma$ ; auto simp add: sup-commute)

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

```

 $\tau \leq \varrho \Longrightarrow \sigma \leq \varrho \Longrightarrow \tau \sqcup \sigma \leq \varrho$ 

```

**for**  $\tau \sigma \varrho :: 'a$  basic-type

**by** (induct  $\varrho$ ; auto)

**instance**

**by** *standard* (auto simp add: sup-ge1-basic-type  
sup-commut-basic-type sup-least-basic-type)

end

## 2.4 Code Setup

**code-pred** *basic-subtype* .

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

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

$(<) =$  *basic-subtype-fun*

**proof** (intro ext iffI)

**fix**  $\tau \sigma$  :: 'a *basic-type*

**show**  $\tau < \sigma \Longrightarrow$  *basic-subtype-fun*  $\tau \sigma$

**apply** (cases  $\sigma$ ; auto)

**using** *basic-subtype-fun.elims*(3) **by** fastforce

**show** *basic-subtype-fun*  $\tau \sigma \Longrightarrow$   $\tau < \sigma$

**apply** (erule *basic-subtype-fun.elims*, auto)

**by** (cases  $\sigma$ , auto)

qed

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

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

**unfolding** *dual-order.order-iff-strict less-basic-type-code*

**by** auto

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  $\Rightarrow$  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 (fmap size-type  $\pi$ )))

```

**instance ..**

**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:



### 3.2. CONSTRUCTORS BIJECTIVITY ON TRANSITIVE CLOSURES 49

```
 $\tau \sqsubset \sigma \implies \sigma \sqsubset \tau \implies \text{False}$   
apply (induct rule: subtype.induct)  
using basic-subtype-asym apply auto  
using subtuple-antisym by fastforce
```

## 3.2 Constructors Bijectivity on Transitive Closures

**lemma** *Required-bij-on-trancl* [simp]:

```
bij-on-trancl subtype Required  
by (auto simp add: inj-def)
```

**lemma** *not-subtype-Optional-Required*:

```
 $\text{subtype}^{++} \tau[\varrho] \sigma \implies \sigma = \varrho[1] \implies P$   
by (induct arbitrary:  $\varrho$  rule: tranclp-induct; auto)
```

**lemma** *Optional-bij-on-trancl* [simp]:

```
bij-on-trancl subtype Optional  
apply (auto simp add: inj-def)  
using not-subtype-Optional-Required by blast
```

**lemma** *subtype-tranclp-Collection-x*:

```
 $\text{subtype}^{++} (\text{Collection } \tau) \sigma \implies$   
 $(\bigwedge \varrho. \sigma = \text{Collection } \varrho \implies \text{subtype}^{++} \tau \varrho \implies P) \implies$   
 $(\sigma = \text{OclSuper} \implies P) \implies P$   
apply (induct rule: tranclp-induct, auto)  
by (metis subtype-Collection-x subtype-OclSuper-x tranclp.trancl-into-trancl)
```

**lemma** *Collection-bij-on-trancl* [simp]:

```
bij-on-trancl subtype Collection  
apply (auto simp add: inj-def)  
using subtype-tranclp-Collection-x by auto
```

**lemma** *Set-bij-on-trancl* [simp]:

```
bij-on-trancl subtype Set  
by (auto simp add: inj-def)
```

**lemma** *OrderedSet-bij-on-trancl* [simp]:

```
bij-on-trancl subtype OrderedSet  
by (auto simp add: inj-def)
```

**lemma** *Bag-bij-on-trancl* [simp]:

```
bij-on-trancl subtype Bag  
by (auto simp add: inj-def)
```

**lemma** *Sequence-bij-on-trancl* [simp]:

```
bij-on-trancl subtype Sequence  
by (auto simp add: inj-def)
```

**lemma** *Tuple-bij-on-trancl* [simp]:

*bij-on-trancl subtype Tuple*  
**by** (*auto simp add: inj-def*)

### 3.3 Partial Order of Types

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

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

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

#### 3.3.1 Strict Introduction Rules

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

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

**unfolding** *less-type-def less-basic-type-def*

**by** *simp (rule preserve-tranclp; auto)*

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

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

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

**unfolding** *less-type-def less-basic-type-def less-eq-basic-type-def*

**apply** *simp-all*

**apply** (*rule preserve-rtranclp''; auto*)

**by** (*rule preserve-tranclp; auto*)

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

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

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

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

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

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

**unfolding** *less-type-def less-eq-type-def*

**apply** *simp-all*

**apply** (*rule-tac ?f= Collection in preserve-tranclp; auto*)

**apply** (*rule preserve-rtranclp''; auto*)

**apply** (*rule preserve-rtranclp''; auto*)

**apply** (*rule preserve-rtranclp''; auto*)

**by** (*rule preserve-rtranclp''; auto*)

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

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

**unfolding** *less-type-def*

**by** *simp (rule preserve-tranclp; auto)*

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

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

**unfolding** *less-type-def*

**by** *simp (rule preserve-tranclp; auto)*

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

$\tau = \text{Bag } \varrho \implies \varrho < \sigma \implies \tau < \text{Bag } \sigma$   
**unfolding** *less-type-def*  
**by** *simp* (*rule preserve-tranclp; auto*)

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

$\tau = \text{Sequence } \varrho \implies \varrho < \sigma \implies \tau < \text{Sequence } \sigma$   
**unfolding** *less-type-def*  
**by** *simp* (*rule preserve-tranclp; auto*)

**lemma** *fun-or-eq-refl* [*intro*]:

*reflp* ( $\lambda x y. f x y \vee x = y$ )  
**by** (*simp add: reflpI*)

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

**assumes**  $\tau = \text{Tuple } \pi$   
**and** *strict-subtuple* ( $\leq$ )  $\pi \xi$   
**shows**  $\tau < \text{Tuple } \xi$

**proof** –

**have** *subtuple* ( $\lambda \tau \sigma. \tau \sqsubset \sigma \vee \tau = \sigma$ )\*\*  $\pi \xi$   
**using** *assms(2) less-eq-type-def* **by** *auto*  
**hence** (*subtuple* ( $\lambda \tau \sigma. \tau \sqsubset \sigma \vee \tau = \sigma$ ))<sup>++</sup>  $\pi \xi$   
**by** *simp* (*rule subtuple-to-trancl; auto*)  
**hence** (*strict-subtuple* ( $\lambda \tau \sigma. \tau \sqsubset \sigma \vee \tau = \sigma$ ))\*\*  $\pi \xi$   
**by** (*simp add: tranclp-into-rtranclp*)  
**hence** (*strict-subtuple* ( $\lambda \tau \sigma. \tau \sqsubset \sigma \vee \tau = \sigma$ ))<sup>++</sup>  $\pi \xi$   
**by** (*meson assms(2) rtranclpD*)  
**thus** *?thesis*  
**unfolding** *less-type-def*  
**using** *assms(1) apply simp*  
**by** (*rule preserve-tranclp; auto*)

**qed**

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

$\tau \neq \text{OclSuper} \implies \tau < \text{OclSuper}$   
**unfolding** *less-type-def*

**proof** (*induct*  $\tau$ )

**case** *OclSuper* **thus** *?case* **by** *auto*

**next**

**case** (*Required*  $\tau$ )

**have** *subtype*\*\*  $\tau[1] \text{OclAny}[1]$

**apply** (*rule-tac ?f= Required in preserve-rtranclp[of basic-subtype], auto*)

**by** (*metis less-eq-basic-type-def type-less-eq-x-OclAny-intro*)

**also have** *subtype*<sup>++</sup>  $\text{OclAny}[1] \text{OclAny}[?]$  **by** *auto*

**also have** *subtype*<sup>++</sup>  $\text{OclAny}[?] \text{OclSuper}$  **by** *auto*

**finally show** *?case* **by** *auto*

**next**

**case** (*Optional*  $\tau$ )

```

have subtype**  $\tau$ [?] OclAny[?]
  apply (rule-tac ?f= Optional in preserve-rtranclp[of basic-subtype], auto)
  by (metis less-eq-basic-type-def type-less-eq-x-OclAny-intro)
also have subtype++ OclAny[?] OclSuper by auto
finally show ?case by auto
next
case (Collection  $\tau$ )
have subtype** (Collection  $\tau$ ) (Collection OclSuper)
  apply (rule-tac ?f= Collection in preserve-rtranclp[of subtype], auto)
  using Collection.hyps by force
also have subtype++ (Collection OclSuper) OclSuper by auto
finally show ?case by auto
next
case (Set  $\tau$ )
have subtype++ (Set  $\tau$ ) (Collection  $\tau$ ) by auto
also have subtype** (Collection  $\tau$ ) (Collection OclSuper)
  apply (rule-tac ?f= Collection in preserve-rtranclp[of subtype], auto)
  using Set.hyps by force
also have subtype** (Collection OclSuper) OclSuper by auto
finally show ?case by auto
next
case (OrderedSet  $\tau$ )
have subtype++ (OrderedSet  $\tau$ ) (Collection  $\tau$ ) by auto
also have subtype** (Collection  $\tau$ ) (Collection OclSuper)
  apply (rule-tac ?f= Collection in preserve-rtranclp[of subtype], auto)
  using OrderedSet.hyps by force
also have subtype** (Collection OclSuper) OclSuper by auto
finally show ?case by auto
next
case (Bag  $\tau$ )
have subtype++ (Bag  $\tau$ ) (Collection  $\tau$ ) by auto
also have subtype** (Collection  $\tau$ ) (Collection OclSuper)
  apply (rule-tac ?f= Collection in preserve-rtranclp[of subtype], auto)
  using Bag.hyps by force
also have subtype** (Collection OclSuper) OclSuper by auto
finally show ?case by auto
next
case (Sequence  $\tau$ )
have subtype++ (Sequence  $\tau$ ) (Collection  $\tau$ ) by auto
also have subtype** (Collection  $\tau$ ) (Collection OclSuper)
  apply (rule-tac ?f= Collection in preserve-rtranclp[of subtype], auto)
  using Sequence.hyps by force
also have subtype** (Collection OclSuper) OclSuper by auto
finally show ?case by auto
next
case (Tuple  $x$ ) thus ?case by auto
qed

```

### 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**  $P$

**proof** –

**from** *assms(1)* **obtain**  $\varrho$  **where**  $\tau = \varrho[1]$   
**unfolding** *less-type-def*  
**by** (*induct rule: converse-tranclp-induct; auto*)  
**moreover have**  $\bigwedge \tau \sigma. \tau[1] < \sigma[1] \implies \tau < \sigma$   
**unfolding** *less-type-def less-basic-type-def*  
**by** (*rule reflect-tranclp; auto*)  
**ultimately show** *?thesis*  
**using** *assms* **by** *auto*

**qed**

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

$\tau < \sigma[?] \implies$   
 $(\bigwedge \varrho. \tau = \varrho[1] \implies \varrho \leq \sigma \implies P) \implies$   
 $(\bigwedge \varrho. \tau = \varrho[?] \implies \varrho < \sigma \implies P) \implies P$

**unfolding** *less-type-def*

**apply** (*induct rule: converse-tranclp-induct*)

**apply** (*metis subtype-x-Optional eq-refl less-basic-type-def tranclp.r-into-trancl*)

**apply** (*erule subtype.cases; auto*)

**apply** (*simp add: converse-rtranclp-into-rtranclp less-eq-basic-type-def*)

**by** (*simp add: less-basic-type-def tranclp-into-tranclp2*)

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

$\tau < \text{Collection } \sigma \implies$   
 $(\bigwedge \varrho. \tau = \text{Collection } \varrho \implies \varrho < \sigma \implies P) \implies$   
 $(\bigwedge \varrho. \tau = \text{Set } \varrho \implies \varrho \leq \sigma \implies P) \implies$   
 $(\bigwedge \varrho. \tau = \text{OrderedSet } \varrho \implies \varrho \leq \sigma \implies P) \implies$   
 $(\bigwedge \varrho. \tau = \text{Bag } \varrho \implies \varrho \leq \sigma \implies P) \implies$   
 $(\bigwedge \varrho. \tau = \text{Sequence } \varrho \implies \varrho \leq \sigma \implies P) \implies P$

**unfolding** *less-type-def*

**apply** (*induct rule: converse-tranclp-induct*)

**apply** (*metis (mono-tags) Nitpick.rtranclp-unfold*  
*subtype-x-Collection less-eq-type-def tranclp.r-into-trancl*)

**by** (*erule subtype.cases;*

*auto simp add: converse-rtranclp-into-rtranclp less-eq-type-def*  
*tranclp-into-tranclp2 tranclp-into-rtranclp*)

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

**assumes**  $\tau < \text{Set } \sigma$

**and**  $\bigwedge \varrho. \tau = \text{Set } \varrho \implies \varrho < \sigma \implies P$

**shows**  $P$

**proof** –

**from** *assms(1)* **obtain**  $\varrho$  **where**  $\tau = \text{Set } \varrho$

**unfolding** *less-type-def*

by (*induct rule: converse-tranclp-induct; auto*)  
**moreover have**  $\bigwedge \tau \sigma. \text{Set } \tau < \text{Set } \sigma \implies \tau < \sigma$   
 unfolding *less-type-def*  
 by (*rule reflect-tranclp; auto*)  
**ultimately show** *?thesis*  
 using *assms* by *auto*  
**qed**

**lemma** *type-less-x-OrderedSet* [*elim!*]:  
**assumes**  $\tau < \text{OrderedSet } \sigma$   
 and  $\bigwedge \varrho. \tau = \text{OrderedSet } \varrho \implies \varrho < \sigma \implies P$   
**shows**  $P$   
**proof** –  
**from** *assms*(1) **obtain**  $\varrho$  **where**  $\tau = \text{OrderedSet } \varrho$   
 unfolding *less-type-def*  
 by (*induct rule: converse-tranclp-induct; auto*)  
**moreover have**  $\bigwedge \tau \sigma. \text{OrderedSet } \tau < \text{OrderedSet } \sigma \implies \tau < \sigma$   
 unfolding *less-type-def*  
 by (*rule reflect-tranclp; auto*)  
**ultimately show** *?thesis*  
 using *assms* by *auto*  
**qed**

**lemma** *type-less-x-Bag* [*elim!*]:  
**assumes**  $\tau < \text{Bag } \sigma$   
 and  $\bigwedge \varrho. \tau = \text{Bag } \varrho \implies \varrho < \sigma \implies P$   
**shows**  $P$   
**proof** –  
**from** *assms*(1) **obtain**  $\varrho$  **where**  $\tau = \text{Bag } \varrho$   
 unfolding *less-type-def*  
 by (*induct rule: converse-tranclp-induct; auto*)  
**moreover have**  $\bigwedge \tau \sigma. \text{Bag } \tau < \text{Bag } \sigma \implies \tau < \sigma$   
 unfolding *less-type-def*  
 by (*rule reflect-tranclp; auto*)  
**ultimately show** *?thesis*  
 using *assms* by *auto*  
**qed**

**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** –  
**from** *assms*(1) **obtain**  $\varrho$  **where**  $\tau = \text{Sequence } \varrho$   
 unfolding *less-type-def*  
 by (*induct rule: converse-tranclp-induct; auto*)  
**moreover have**  $\bigwedge \tau \sigma. \text{Sequence } \tau < \text{Sequence } \sigma \implies \tau < \sigma$   
 unfolding *less-type-def*  
 by (*rule reflect-tranclp; auto*)

**ultimately show** *?thesis*  
**using** *assms* **by** *auto*  
**qed**

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** –

**from** *assms(1)* **obtain**  $\pi$  **where**  $\tau = \text{Tuple } \pi$   
**unfolding** *less-type-def*  
**by** (*induct rule: converse-tranclp-induct; auto*)  
**moreover from** *assms(2)* **have**  
 $\bigwedge \pi. \text{Tuple } \pi < \text{Tuple } \xi \implies \text{strict-subtuple } (\leq) \pi \xi$   
**unfolding** *less-type-def less-eq-type-def*  
**by** (*rule-tac ?f= Tuple in strict-subtuple-rtranclp-intro; auto*)  
**ultimately show** *?thesis*  
**using** *assms* **by** *auto*  
**qed**

**lemma** *type-less-x-OclSuper [elim!]*:  
 $\tau < \text{OclSuper} \implies (\tau \neq \text{OclSuper} \implies P) \implies P$   
**unfolding** *less-type-def*  
**by** (*drule tranclpD, auto*)

### 3.3.3 Properties

**lemma** *subtype-irrefl*:  
 $\tau < \tau \implies \text{False}$   
**for**  $\tau :: 'a \text{ type}$   
**apply** (*induct  $\tau$ , auto*)  
**apply** (*erule type-less-x-Tuple', auto*)  
**unfolding** *less-type-def tranclp-unfold*  
**by** *auto*

**lemma** *subtype-acyclic*:  
*acyclicP subtype*  
**apply** (*rule acyclicI*)  
**apply** (*simp add: trancl-def*)  
**by** (*metis (mono-tags) OCL-Types.less-type-def OCL-Types.subtype-irrefl*)

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

```

apply (auto simp add: less-type-def less-eq-type-def)
by (metis (mono-tags) subtype-irrefl less-type-def tranclp-rtranclp-tranclp)
show  $\tau \leq \sigma \wedge \neg \sigma \leq \tau \implies \tau < \sigma$ 
apply (auto simp add: less-type-def less-eq-type-def)
by (metis rtranclpD)
qed

```

```

lemma order-refl-type [iff]:
   $\tau \leq \tau$ 
for  $\tau :: 'a$  type
unfolding less-eq-type-def by simp

```

```

lemma order-trans-type:
   $\tau \leq \sigma \implies \sigma \leq \rho \implies \tau \leq \rho$ 
for  $\tau \sigma \rho :: 'a$  type
unfolding less-eq-type-def by simp

```

```

lemma antisym-type:
   $\tau \leq \sigma \implies \sigma \leq \tau \implies \tau = \sigma$ 
for  $\tau \sigma :: 'a$  type
unfolding less-eq-type-def less-type-def
by (metis (mono-tags, lifting) less-eq-type-def
      less-le-not-le-type less-type-def rtranclpD)

```

```

instance
apply intro-classes
apply (simp add: less-le-not-le-type)
apply (simp)
using order-trans-type apply blast
by (simp add: antisym-type)

```

**end**

### 3.3.4 Non-Strict Introduction Rules

```

lemma type-less-eq-x-Required-intro [intro]:
   $\tau = \rho[1] \implies \rho \leq \sigma \implies \tau \leq \sigma[1]$ 
unfolding dual-order.order-iff-strict by auto

```

```

lemma type-less-eq-x-Optional-intro [intro]:
   $\tau = \rho[1] \implies \rho \leq \sigma \implies \tau \leq \sigma[?]$ 
   $\tau = \rho[?] \implies \rho \leq \sigma \implies \tau \leq \sigma[?]$ 
unfolding dual-order.order-iff-strict by auto

```

```

lemma type-less-eq-x-Collection-intro [intro]:
   $\tau = \text{Collection } \rho \implies \rho \leq \sigma \implies \tau \leq \text{Collection } \sigma$ 
   $\tau = \text{Set } \rho \implies \rho \leq \sigma \implies \tau \leq \text{Collection } \sigma$ 
   $\tau = \text{OrderedSet } \rho \implies \rho \leq \sigma \implies \tau \leq \text{Collection } \sigma$ 
   $\tau = \text{Bag } \rho \implies \rho \leq \sigma \implies \tau \leq \text{Collection } \sigma$ 

```



$\tau = \text{Sequence } \varrho \implies \varrho \leq \sigma \implies \tau \leq \text{Collection } \sigma$   
**unfolding** *dual-order.order-iff-strict* **by** *auto*

**lemma** *type-less-eq-x-Set-intro* [*intro*]:  
 $\tau = \text{Set } \varrho \implies \varrho \leq \sigma \implies \tau \leq \text{Set } \sigma$   
**unfolding** *dual-order.order-iff-strict* **by** *auto*

**lemma** *type-less-eq-x-OrderedSet-intro* [*intro*]:  
 $\tau = \text{OrderedSet } \varrho \implies \varrho \leq \sigma \implies \tau \leq \text{OrderedSet } \sigma$   
**unfolding** *dual-order.order-iff-strict* **by** *auto*

**lemma** *type-less-eq-x-Bag-intro* [*intro*]:  
 $\tau = \text{Bag } \varrho \implies \varrho \leq \sigma \implies \tau \leq \text{Bag } \sigma$   
**unfolding** *dual-order.order-iff-strict* **by** *auto*

**lemma** *type-less-eq-x-Sequence-intro* [*intro*]:  
 $\tau = \text{Sequence } \varrho \implies \varrho \leq \sigma \implies \tau \leq \text{Sequence } \sigma$   
**unfolding** *dual-order.order-iff-strict* **by** *auto*

**lemma** *type-less-eq-x-Tuple-intro* [*intro*]:  
 $\tau = \text{Tuple } \pi \implies \text{subtuple } (\leq) \pi \xi \implies \tau \leq \text{Tuple } \xi$   
**using** *dual-order.strict-iff-order* **by** *blast*

**lemma** *type-less-eq-x-OclSuper-intro* [*intro*]:  
 $\tau \leq \text{OclSuper}$   
**unfolding** *dual-order.order-iff-strict* **by** *auto*

### 3.3.5 Non-Strict Elimination Rules

**lemma** *type-less-eq-x-Required* [*elim!*]:  
 $\tau \leq \sigma[I] \implies$   
 $(\bigwedge \varrho. \tau = \varrho[I] \implies \varrho \leq \sigma \implies P) \implies P$   
**by** (*drule le-imp-less-or-eq*; *auto*)

**lemma** *type-less-eq-x-Optional* [*elim!*]:  
 $\tau \leq \sigma[?] \implies$   
 $(\bigwedge \varrho. \tau = \varrho[I] \implies \varrho \leq \sigma \implies P) \implies$   
 $(\bigwedge \varrho. \tau = \varrho[?] \implies \varrho \leq \sigma \implies P) \implies P$   
**by** (*drule le-imp-less-or-eq*, *auto*)

**lemma** *type-less-eq-x-Collection* [*elim!*]:  
 $\tau \leq \text{Collection } \sigma \implies$   
 $(\bigwedge \varrho. \tau = \text{Set } \varrho \implies \varrho \leq \sigma \implies P) \implies$   
 $(\bigwedge \varrho. \tau = \text{OrderedSet } \varrho \implies \varrho \leq \sigma \implies P) \implies$   
 $(\bigwedge \varrho. \tau = \text{Bag } \varrho \implies \varrho \leq \sigma \implies P) \implies$   
 $(\bigwedge \varrho. \tau = \text{Sequence } \varrho \implies \varrho \leq \sigma \implies P) \implies$   
 $(\bigwedge \varrho. \tau = \text{Collection } \varrho \implies \varrho \leq \sigma \implies P) \implies P$   
**by** (*drule le-imp-less-or-eq*; *auto*)

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

$$\begin{aligned} & \tau \leq \text{Set } \sigma \implies \\ & (\bigwedge \varrho. \tau = \text{Set } \varrho \implies \varrho \leq \sigma \implies P) \implies P \\ & \text{by (drule le-imp-less-or-eq; auto)} \end{aligned}$$

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

$$\begin{aligned} & \tau \leq \text{OrderedSet } \sigma \implies \\ & (\bigwedge \varrho. \tau = \text{OrderedSet } \varrho \implies \varrho \leq \sigma \implies P) \implies P \\ & \text{by (drule le-imp-less-or-eq; auto)} \end{aligned}$$

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

$$\begin{aligned} & \tau \leq \text{Bag } \sigma \implies \\ & (\bigwedge \varrho. \tau = \text{Bag } \varrho \implies \varrho \leq \sigma \implies P) \implies P \\ & \text{by (drule le-imp-less-or-eq; auto)} \end{aligned}$$

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

$$\begin{aligned} & \tau \leq \text{Sequence } \sigma \implies \\ & (\bigwedge \varrho. \tau = \text{Sequence } \varrho \implies \varrho \leq \sigma \implies P) \implies P \\ & \text{by (drule le-imp-less-or-eq; auto)} \end{aligned}$$

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

$$\begin{aligned} & \tau < \text{Tuple } \xi \implies \\ & (\bigwedge \pi. \tau = \text{Tuple } \pi \implies \text{strict-subtuple } (\leq) \pi \xi \implies P) \implies P \\ & \text{apply (erule type-less-x-Tuple')} \\ & \text{by (meson acyclic-def subtype-acyclic)} \end{aligned}$$

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

$$\begin{aligned} & \tau \leq \text{Tuple } \xi \implies \\ & (\bigwedge \pi. \tau = \text{Tuple } \pi \implies \text{subtuple } (\leq) \pi \xi \implies P) \implies P \\ & \text{apply (drule le-imp-less-or-eq, auto)} \\ & \text{by (simp add: fmap.rel-refl fmrel-to-subtuple)} \end{aligned}$$

### 3.3.6 Simplification Rules

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

$$\begin{aligned} & \text{OclSuper } < \sigma = \text{False} \\ & \varrho[1] < \sigma = (\exists v. \\ & \quad \sigma = \text{OclSuper } \vee \\ & \quad \sigma = v[1] \wedge \varrho < v \vee \\ & \quad \sigma = v[?] \wedge \varrho \leq v) \\ & \varrho[?] < \sigma = (\exists v. \\ & \quad \sigma = \text{OclSuper } \vee \\ & \quad \sigma = v[?] \wedge \varrho < v) \\ & \text{Collection } \tau < \sigma = (\exists \varphi. \\ & \quad \sigma = \text{OclSuper } \vee \\ & \quad \sigma = \text{Collection } \varphi \wedge \tau < \varphi) \\ & \text{Set } \tau < \sigma = (\exists \varphi. \\ & \quad \sigma = \text{OclSuper } \vee \\ & \quad \sigma = \text{Collection } \varphi \wedge \tau \leq \varphi \vee \end{aligned}$$

```

     $\sigma = \text{Set } \varphi \wedge \tau < \varphi$ 
  OrderedSet  $\tau < \sigma = (\exists \varphi.$ 
     $\sigma = \text{OclSuper } \vee$ 
     $\sigma = \text{Collection } \varphi \wedge \tau \leq \varphi \vee$ 
     $\sigma = \text{OrderedSet } \varphi \wedge \tau < \varphi)$ 
  Bag  $\tau < \sigma = (\exists \varphi.$ 
     $\sigma = \text{OclSuper } \vee$ 
     $\sigma = \text{Collection } \varphi \wedge \tau \leq \varphi \vee$ 
     $\sigma = \text{Bag } \varphi \wedge \tau < \varphi)$ 
  Sequence  $\tau < \sigma = (\exists \varphi.$ 
     $\sigma = \text{OclSuper } \vee$ 
     $\sigma = \text{Collection } \varphi \wedge \tau \leq \varphi \vee$ 
     $\sigma = \text{Sequence } \varphi \wedge \tau < \varphi)$ 
  Tuple  $\pi < \sigma = (\exists \xi.$ 
     $\sigma = \text{OclSuper } \vee$ 
     $\sigma = \text{Tuple } \xi \wedge \text{strict-subtuple } (\leq) \pi \xi)$ 
  by (induct  $\sigma$ ; auto)+

```

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

```

 $\tau < \text{OclSuper} = (\tau \neq \text{OclSuper})$ 
 $\tau < v[1] = (\exists \varrho. \tau = \varrho[1] \wedge \varrho < v)$ 
 $\tau < v[?] = (\exists \varrho. \tau = \varrho[1] \wedge \varrho \leq v \vee \tau = \varrho[?] \wedge \varrho < v)$ 
 $\tau < \text{Collection } \sigma = (\exists \varphi.$ 
   $\tau = \text{Collection } \varphi \wedge \varphi < \sigma \vee$ 
   $\tau = \text{Set } \varphi \wedge \varphi \leq \sigma \vee$ 
   $\tau = \text{OrderedSet } \varphi \wedge \varphi \leq \sigma \vee$ 
   $\tau = \text{Bag } \varphi \wedge \varphi \leq \sigma \vee$ 
   $\tau = \text{Sequence } \varphi \wedge \varphi \leq \sigma)$ 
 $\tau < \text{Set } \sigma = (\exists \varphi. \tau = \text{Set } \varphi \wedge \varphi < \sigma)$ 
 $\tau < \text{OrderedSet } \sigma = (\exists \varphi. \tau = \text{OrderedSet } \varphi \wedge \varphi < \sigma)$ 
 $\tau < \text{Bag } \sigma = (\exists \varphi. \tau = \text{Bag } \varphi \wedge \varphi < \sigma)$ 
 $\tau < \text{Sequence } \sigma = (\exists \varphi. \tau = \text{Sequence } \varphi \wedge \varphi < \sigma)$ 
 $\tau < \text{Tuple } \xi = (\exists \pi. \tau = \text{Tuple } \pi \wedge \text{strict-subtuple } (\leq) \pi \xi)$ 
  by auto

```

### 3.4 Upper Semilattice of Types

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

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

```

  OclSuper  $\sqcup$   $\sigma = \text{OclSuper}$ 
| Required  $\tau \sqcup \sigma = (\text{case } \sigma$ 
  of  $\varrho[1] \Rightarrow (\tau \sqcup \varrho)[1]$ 
  |  $\varrho[?] \Rightarrow (\tau \sqcup \varrho)[?]$ 
  | -  $\Rightarrow \text{OclSuper}$ )
| Optional  $\tau \sqcup \sigma = (\text{case } \sigma$ 
  of  $\varrho[1] \Rightarrow (\tau \sqcup \varrho)[1]$ 
  |  $\varrho[?] \Rightarrow (\tau \sqcup \varrho)[?]$ 

```

```

| - ⇒ OclSuper)
| Collection τ ⊔ σ = (case σ
  of Collection ρ ⇒ Collection (τ ⊔ ρ)
  | Set ρ ⇒ Collection (τ ⊔ ρ)
  | OrderedSet ρ ⇒ Collection (τ ⊔ ρ)
  | Bag ρ ⇒ Collection (τ ⊔ ρ)
  | Sequence ρ ⇒ Collection (τ ⊔ ρ)
  | - ⇒ OclSuper)
| Set τ ⊔ σ = (case σ
  of Collection ρ ⇒ Collection (τ ⊔ ρ)
  | Set ρ ⇒ Set (τ ⊔ ρ)
  | OrderedSet ρ ⇒ Collection (τ ⊔ ρ)
  | Bag ρ ⇒ Collection (τ ⊔ ρ)
  | Sequence ρ ⇒ Collection (τ ⊔ ρ)
  | - ⇒ OclSuper)
| OrderedSet τ ⊔ σ = (case σ
  of Collection ρ ⇒ Collection (τ ⊔ ρ)
  | Set ρ ⇒ Collection (τ ⊔ ρ)
  | OrderedSet ρ ⇒ OrderedSet (τ ⊔ ρ)
  | Bag ρ ⇒ Collection (τ ⊔ ρ)
  | Sequence ρ ⇒ Collection (τ ⊔ ρ)
  | - ⇒ OclSuper)
| Bag τ ⊔ σ = (case σ
  of Collection ρ ⇒ Collection (τ ⊔ ρ)
  | Set ρ ⇒ Collection (τ ⊔ ρ)
  | OrderedSet ρ ⇒ Collection (τ ⊔ ρ)
  | Bag ρ ⇒ Bag (τ ⊔ ρ)
  | Sequence ρ ⇒ Collection (τ ⊔ ρ)
  | - ⇒ OclSuper)
| Sequence τ ⊔ σ = (case σ
  of Collection ρ ⇒ Collection (τ ⊔ ρ)
  | Set ρ ⇒ Collection (τ ⊔ ρ)
  | OrderedSet ρ ⇒ Collection (τ ⊔ ρ)
  | Bag ρ ⇒ Collection (τ ⊔ ρ)
  | Sequence ρ ⇒ Sequence (τ ⊔ ρ)
  | - ⇒ OclSuper)
| Tuple π ⊔ σ = (case σ
  of Tuple ξ ⇒ Tuple (fmerge-fun (⊔) π ξ)
  | - ⇒ OclSuper)

```

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

$\tau \leq \tau \sqcup \sigma$

**for**  $\tau \sigma :: 'a$  type

**proof** (*induct*  $\tau$  arbitrary:  $\sigma$ )

**case** *OclSuper* **show** ?case **by** *simp*

**case** (*Required*  $\tau$ ) **show** ?case **by** (*induct*  $\sigma$ ; *auto*)

**case** (*Optional*  $\tau$ ) **show** ?case **by** (*induct*  $\sigma$ ; *auto*)

**case** (*Collection*  $\tau$ ) **thus** ?case **by** (*induct*  $\sigma$ ; *auto*)

**case** (*Set*  $\tau$ ) **thus** ?case **by** (*induct*  $\sigma$ ; *auto*)

```

case (OrderedSet  $\tau$ ) thus ?case by (induct  $\sigma$ ; auto)
case (Bag  $\tau$ ) thus ?case by (induct  $\sigma$ ; auto)
case (Sequence  $\tau$ ) thus ?case by (induct  $\sigma$ ; auto)
next
case (Tuple  $\pi$ )
  moreover have Tuple-less-eq-sup:
    ( $\bigwedge \tau \sigma. \tau \in \text{fmran}' \pi \implies \tau \leq \tau \sqcup \sigma \implies$ 
      Tuple  $\pi \leq$  Tuple  $\pi \sqcup \sigma$ )
    by (cases  $\sigma$ , auto)
  ultimately show ?case by (cases  $\sigma$ , auto)
qed

```

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

```

 $\tau \sqcup \sigma = \sigma \sqcup \tau$ 
for  $\tau \sigma :: 'a$  type
proof (induct  $\tau$  arbitrary:  $\sigma$ )
  case OclSuper show ?case by (cases  $\sigma$ ; simp add: less-eq-type-def)
  case (Required  $\tau$ ) show ?case by (cases  $\sigma$ ; simp add: sup-commute)
  case (Optional  $\tau$ ) show ?case by (cases  $\sigma$ ; simp add: sup-commute)
  case (Collection  $\tau$ ) thus ?case by (cases  $\sigma$ ; simp)
  case (Set  $\tau$ ) thus ?case by (cases  $\sigma$ ; simp)
  case (OrderedSet  $\tau$ ) thus ?case by (cases  $\sigma$ ; simp)
  case (Bag  $\tau$ ) thus ?case by (cases  $\sigma$ ; simp)
  case (Sequence  $\tau$ ) thus ?case by (cases  $\sigma$ ; simp)
next
case (Tuple  $\pi$ ) thus ?case
  apply (cases  $\sigma$ ; simp add: less-eq-type-def)
  using fmmerge-commut by blast
qed

```

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

```

 $\tau \leq \varrho \implies \sigma \leq \varrho \implies \tau \sqcup \sigma \leq \varrho$ 
for  $\tau \sigma \varrho :: 'a$  type
proof (induct  $\varrho$  arbitrary:  $\tau \sigma$ )
  case OclSuper show ?case using eq-refl by auto
next
case (Required  $x$ ) show ?case
  apply (insert Required)
  by (erule type-less-eq-x-Required; erule type-less-eq-x-Required; auto)
next
case (Optional  $x$ ) show ?case
  apply (insert Optional)
  by (erule type-less-eq-x-Optional; erule type-less-eq-x-Optional; auto)
next
case (Collection  $\varrho$ ) show ?case
  apply (insert Collection)
  by (erule type-less-eq-x-Collection; erule type-less-eq-x-Collection; auto)
next
case (Set  $\varrho$ ) show ?case

```

```

    apply (insert Set)
    by (erule type-less-eq-x-Set; erule type-less-eq-x-Set; auto)
next
  case (OrderedSet  $\rho$ ) show ?case
    apply (insert OrderedSet)
    by (erule type-less-eq-x-OrderedSet; erule type-less-eq-x-OrderedSet; auto)
next
  case (Bag  $\rho$ ) show ?case
    apply (insert Bag)
    by (erule type-less-eq-x-Bag; erule type-less-eq-x-Bag; auto)
next
  case (Sequence  $\rho$ ) thus ?case
    apply (insert Sequence)
    by (erule type-less-eq-x-Sequence; erule type-less-eq-x-Sequence; auto)
next
  case (Tuple  $\pi$ ) show ?case
    apply (insert Tuple)
    apply (erule type-less-eq-x-Tuple; erule type-less-eq-x-Tuple; auto)
    by (rule-tac ? $\pi$  = (fmmerge ( $\sqcup$ )  $\pi'$   $\pi''$ ) in type-less-eq-x-Tuple-intro;
        simp add: fmrel-on-fset-fmmerge1)
qed

instance
  apply intro-classes
  apply (simp add: sup-ge1-type)
  apply (simp add: sup-commut-type sup-ge1-type)
  by (simp add: sup-least-type)

end

```

### 3.5 Helper Relations

**abbreviation** *between* ( $- / = \dashv$  [ $51, 51, 51$ ]  $50$ ) **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  $\sigma = \tau$ )

by (auto simp add: element-type.simps)

**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  $\tau \sigma_2 \implies \sigma_1 = \sigma_2$ 
by (induct rule: element-type.induct; simp add: element-type.simps)

```

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

```

update-element-type  $\tau \sigma \varrho_1 \implies$ 
update-element-type  $\tau \sigma \varrho_2 \implies \varrho_1 = \varrho_2$ 
by (induct rule: update-element-type.induct; simp add: update-element-type.simps)

```

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

```

to-unique-collection  $\tau \sigma_1 \implies$ 
to-unique-collection  $\tau \sigma_2 \implies \sigma_1 = \sigma_2$ 
by (induct rule: to-unique-collection.induct; simp add: to-unique-collection.simps)

```

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

```

to-nonunique-collection  $\tau \sigma_1 \implies$ 
to-nonunique-collection  $\tau \sigma_2 \implies \sigma_1 = \sigma_2$ 
by (induct rule: to-nonunique-collection.induct; simp add: to-nonunique-collection.simps)

```

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

```

to-ordered-collection  $\tau \sigma_1 \implies$ 
to-ordered-collection  $\tau \sigma_2 \implies \sigma_1 = \sigma_2$ 
by (induct rule: to-ordered-collection.induct; simp add: to-ordered-collection.simps)

```

### 3.7 Code Setup

**code-pred** *subtype* .

**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)
by pat-completeness auto
termination
by (relation measure ( $\lambda(xs, ys)$ . size ys) ;
auto simp add: elem-le-ffold' fmran'I)

```

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

( $<$ ) = subtype-fun

**proof** (intro ext iffI)

**fix**  $\tau$   $\sigma$  :: 'a type

**show**  $\tau < \sigma \Longrightarrow$  subtype-fun  $\tau$   $\sigma$

**proof** (induct  $\tau$  arbitrary:  $\sigma$ )

**case** OclSuper **thus** ?case **by** (cases  $\sigma$ ; auto)

**next**

**case** (Required  $\tau$ ) **thus** ?case

**by** (cases  $\sigma$ ; auto simp: less-basic-type-code less-eq-basic-type-code)

**next**

**case** (Optional  $\tau$ ) **thus** ?case

**by** (cases  $\sigma$ ; auto simp: less-basic-type-code less-eq-basic-type-code)

```

next
  case (Collection  $\tau$ ) thus ?case by (cases  $\sigma$ ; auto)
next
  case (Set  $\tau$ ) thus ?case by (cases  $\sigma$ ; auto)
next
  case (OrderedSet  $\tau$ ) thus ?case by (cases  $\sigma$ ; auto)
next
  case (Bag  $\tau$ ) thus ?case by (cases  $\sigma$ ; auto)
next
  case (Sequence  $\tau$ ) thus ?case by (cases  $\sigma$ ; auto)
next
  case (Tuple  $\pi$ )
  have
     $\bigwedge \xi. \text{subtuple } (\leq) \pi \xi \longrightarrow$ 
     $\text{subtuple } (\lambda \tau \sigma. \text{subtype-fun } \tau \sigma \vee \tau = \sigma) \pi \xi$ 
    by (rule subtuple-mono; auto simp add: Tuple.hyps)
  with Tuple.premis show ?case by (cases  $\sigma$ ; auto)
qed
show subtype-fun  $\tau \sigma \implies \tau < \sigma$ 
proof (induct  $\sigma$  arbitrary:  $\tau$ )
  case OclSuper thus ?case by (cases  $\sigma$ ; auto)
next
  case (Required  $\sigma$ ) show ?case
  by (insert Required) (erule subtype-fun.elims;
    auto simp: less-basic-type-code less-eq-basic-type-code)
next
  case (Optional  $\sigma$ ) show ?case
  by (insert Optional) (erule subtype-fun.elims;
    auto simp: less-basic-type-code less-eq-basic-type-code)
next
  case (Collection  $\sigma$ ) show ?case
  apply (insert Collection)
  apply (erule subtype-fun.elims; auto)
  using order.strict-implies-order by auto
next
  case (Set  $\sigma$ ) show ?case
  by (insert Set) (erule subtype-fun.elims; auto)
next
  case (OrderedSet  $\sigma$ ) show ?case
  by (insert OrderedSet) (erule subtype-fun.elims; auto)
next
  case (Bag  $\sigma$ ) show ?case
  by (insert Bag) (erule subtype-fun.elims; auto)
next
  case (Sequence  $\sigma$ ) show ?case
  by (insert Sequence) (erule subtype-fun.elims; auto)
next
  case (Tuple  $\xi$ )
  have subtuple-imp-simp:

```

```

     $\wedge \pi. \text{subtuple } (\lambda \tau \sigma. \text{subtype-fun } \tau \sigma \vee \tau = \sigma) \pi \xi \longrightarrow$ 
     $\text{subtuple } (\leq) \pi \xi$ 
    by (rule subtuple-mono; auto simp add: Tuple.hyps less-imp-le)
  show ?case
    apply (insert Tuple)
    by (erule subtype-fun.elims; auto simp add: subtuple-imp-simp)
qed
qed

```

```

lemma less-eq-type-code [code]:
   $(\leq) = (\lambda x y. \text{subtype-fun } x y \vee x = y)$ 
  unfolding dual-order.order-iff-strict less-type-code
  by auto

```

```

code-pred element-type .
code-pred update-element-type .
code-pred to-unique-collection .
code-pred to-nonunique-collection .
code-pred to-ordered-collection .

```

```

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 ..
```

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

```
instantiation unat :: infinity
```

```
begin
```

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

```
instance ..
```

```
end
```

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

```
  unat
```

```
|  $\infty$  :: unat
```

```
  unfolding unat-def infinity-unat-def
```

```
  apply (metis Rep-unat-inverse option.collapse)
```

```
  apply (metis Abs-unat-inverse UNIV-I option.sel)
```

```
  by (simp add: Abs-unat-inject)
```

### 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 tuple-element-name ≡ fst
definition tuple-element-type ≡ fst ∘ snd
definition tuple-element-expr ≡ snd ∘ snd

declare [[coercion Literal :: 'a literal-expr ⇒ 'a expr]]

```



**abbreviation** *TypeOperationCall* *src k op ty*  $\equiv$   
*Call src k (TypeOperation op ty)*

**abbreviation** *AttributeCall* *src k attr*  $\equiv$   
*Call src k (Attribute 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
  by standard (simp-all add: local.assoc-end-min-less-eq-max local.association-ends-unique)

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

```
| element-type  $\tau \varrho \implies \sigma < \varrho \implies$ 
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*  $op k \tau \sigma \varrho_2 \implies \varrho_1 = \varrho_2$

**by** (*induct rule*: *typeop-type.induct*;  
*auto simp add*: *typeop-type.simps update-element-type-det*)

**lemma** *any-unop-type-det*:

*any-unop-type op  $\tau$   $\sigma_1 \implies$*

*any-unop-type op  $\tau$   $\sigma_2 \implies \sigma_1 = \sigma_2$*

**by** (*induct rule: any-unop-type.induct; simp add: any-unop-type.simps*)

**lemma** *boolean-unop-type-det*:

*boolean-unop-type op  $\tau$   $\sigma_1 \implies$*

*boolean-unop-type op  $\tau$   $\sigma_2 \implies \sigma_1 = \sigma_2$*

**by** (*induct rule: boolean-unop-type.induct; simp add: boolean-unop-type.simps*)

**lemma** *numeric-unop-type-det*:

*numeric-unop-type op  $\tau$   $\sigma_1 \implies$*

*numeric-unop-type op  $\tau$   $\sigma_2 \implies \sigma_1 = \sigma_2$*

**by** (*induct rule: numeric-unop-type.induct; auto simp add: numeric-unop-type.simps*)

**lemma** *string-unop-type-det*:

*string-unop-type op  $\tau$   $\sigma_1 \implies$*

*string-unop-type op  $\tau$   $\sigma_2 \implies \sigma_1 = \sigma_2$*

**by** (*induct rule: string-unop-type.induct; simp add: string-unop-type.simps*)

**lemma** *collection-unop-type-det*:

*collection-unop-type op  $\tau$   $\sigma_1 \implies$*

*collection-unop-type op  $\tau$   $\sigma_2 \implies \sigma_1 = \sigma_2$*

**apply** (*induct rule: collection-unop-type.induct*)

**by** (*erule collection-unop-type.cases;*

*auto simp add: element-type-det update-element-type-det*)**+**

**lemma** *unop-type-det*:

*unop-type op  $k$   $\tau$   $\sigma_1 \implies$*

*unop-type op  $k$   $\tau$   $\sigma_2 \implies \sigma_1 = \sigma_2$*

**by** (*induct rule: unop-type.induct;*

*simp add: unop-type.simps any-unop-type-det*

*boolean-unop-type-det numeric-unop-type-det*

*string-unop-type-det collection-unop-type-det*)

**lemma** *super-binop-type-det*:

*super-binop-type op  $\tau$   $\sigma$   $\varrho_1 \implies$*

*super-binop-type op  $\tau$   $\sigma$   $\varrho_2 \implies \varrho_1 = \varrho_2$*

**by** (*induct rule: super-binop-type.induct; auto simp add: super-binop-type.simps*)

**lemma** *boolean-binop-type-det*:

*boolean-binop-type op  $\tau$   $\sigma$   $\varrho_1 \implies$*

*boolean-binop-type op  $\tau$   $\sigma$   $\varrho_2 \implies \varrho_1 = \varrho_2$*

**by** (*induct rule: boolean-binop-type.induct; simp add: boolean-binop-type.simps*)

**lemma** *numeric-binop-type-det*:

*numeric-binop-type op  $\tau$   $\sigma$   $\varrho_1 \implies$*

*numeric-binop-type op  $\tau$   $\sigma$   $\varrho_2 \implies \varrho_1 = \varrho_2$*

**by** (*induct rule: numeric-binop-type.induct;*

*auto simp add: numeric-binop-type.simps split: if-splits)*

**lemma** *string-binop-type-det:*

*string-binop-type op  $\tau$   $\sigma$   $\varrho_1 \implies$*

*string-binop-type op  $\tau$   $\sigma$   $\varrho_2 \implies \varrho_1 = \varrho_2$*

**by** (*induct rule: string-binop-type.induct; simp add: string-binop-type.simps*)

**lemma** *collection-binop-type-det:*

*collection-binop-type op  $\tau$   $\sigma$   $\varrho_1 \implies$*

*collection-binop-type op  $\tau$   $\sigma$   $\varrho_2 \implies \varrho_1 = \varrho_2$*

**apply** (*induct rule: collection-binop-type.induct; simp add: collection-binop-type.simps*)

**using** *element-type-det update-element-type-det* **by** *blast+*

**lemma** *binop-type-det:*

*binop-type op  $k$   $\tau$   $\sigma$   $\varrho_1 \implies$*

*binop-type op  $k$   $\tau$   $\sigma$   $\varrho_2 \implies \varrho_1 = \varrho_2$*

**by** (*induct rule: binop-type.induct;*

*simp add: binop-type.simps super-binop-type-det*

*boolean-binop-type-det numeric-binop-type-det*

*string-binop-type-det collection-binop-type-det*)

**lemma** *string-ternop-type-det:*

*string-ternop-type op  $\tau$   $\sigma$   $\varrho$   $v_1 \implies$*

*string-ternop-type op  $\tau$   $\sigma$   $\varrho$   $v_2 \implies v_1 = v_2$*

**by** (*induct rule: string-ternop-type.induct; simp add: string-ternop-type.simps*)

**lemma** *collection-ternop-type-det:*

*collection-ternop-type op  $\tau$   $\sigma$   $\varrho$   $v_1 \implies$*

*collection-ternop-type op  $\tau$   $\sigma$   $\varrho$   $v_2 \implies v_1 = v_2$*

**by** (*induct rule: collection-ternop-type.induct; simp add: collection-ternop-type.simps*)

**lemma** *ternop-type-det:*

*ternop-type op  $k$   $\tau$   $\sigma$   $\varrho$   $v_1 \implies$*

*ternop-type op  $k$   $\tau$   $\sigma$   $\varrho$   $v_2 \implies v_1 = v_2$*

**by** (*induct rule: ternop-type.induct;*

*simp add: ternop-type.simps string-ternop-type-det collection-ternop-type-det*)

**lemma** *op-type-det:*

*op-type op  $k$   $\tau$   $\pi$   $\sigma \implies$*

*op-type op  $k$   $\tau$   $\pi$   $\varrho \implies \sigma = \varrho$*

**apply** (*induct rule: op-type.induct*)

**apply** (*erule op-type.cases; simp add: unop-type-det*)

**apply** (*erule op-type.cases; simp add: binop-type-det*)

**apply** (*erule op-type.cases; simp add: ternop-type-det*)

**by** (*erule op-type.cases; simp; metis operation-det*)

## 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*:  
 $\Gamma \vdash_E \text{NullLiteral} : \text{OclVoid}[?]$   
*BooleanLiteralT*:  
 $\Gamma \vdash_E \text{BooleanLiteral } c : \text{Boolean}[1]$   
*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*:  
*has-literal enum lit*  $\Longrightarrow$   
 $\Gamma \vdash_E \text{EnumLiteral } \text{enum lit} : (\text{Enum } \text{enum})[1]$

— Collection Literals

*SetLiteralT*:  
 $\Gamma \vdash_C \text{prts} : \tau \Longrightarrow$   
 $\Gamma \vdash_E \text{CollectionLiteral } \text{SetKind } \text{prts} : \text{Set } \tau$   
*OrderedSetLiteralT*:  
 $\Gamma \vdash_C \text{prts} : \tau \Longrightarrow$   
 $\Gamma \vdash_E \text{CollectionLiteral } \text{OrderedSetKind } \text{prts} : \text{OrderedSet } \tau$   
*BagLiteralT*:  
 $\Gamma \vdash_C \text{prts} : \tau \Longrightarrow$   
 $\Gamma \vdash_E \text{CollectionLiteral } \text{BagKind } \text{prts} : \text{Bag } \tau$   
*SequenceLiteralT*:  
 $\Gamma \vdash_C \text{prts} : \tau \Longrightarrow$   
 $\Gamma \vdash_E \text{CollectionLiteral } \text{SequenceKind } \text{prts} : \text{Sequence } \tau$

— 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, *OclVoid*[1]  $\sqcup$  *Set*(*Integer*[1]) = *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*:

$$\Gamma \vdash_E \text{TupleLiteral } elems : \text{Tuple } \xi \implies$$

$$\Gamma \vdash_E \text{tuple-element-expr } el : \tau \implies$$

$$\text{tuple-element-type } el = \text{Some } \sigma \implies$$

$$\tau \leq \sigma \implies$$

$$\Gamma \vdash_E \text{TupleLiteral } (el \# elems) : \text{Tuple } (\xi(\text{tuple-element-name } el \mapsto_f \sigma))$$

— Misc Expressions

|*LetT*:

$$\Gamma \vdash_E \text{init} : \sigma \implies$$

$$\sigma \leq \tau \implies$$

$$\Gamma(v \mapsto_f \tau) \vdash_E \text{body} : \varrho \implies$$

$$\Gamma \vdash_E \text{Let } v \ (\text{Some } \tau) \ \text{init } \text{body} : \varrho$$

|*VarT*:

$$\text{fmlookup } \Gamma \ v = \text{Some } \tau \implies$$

$$\Gamma \vdash_E \text{Var } v : \tau$$

|*IfT*:

$$\Gamma \vdash_E a : \text{Boolean}[1] \implies$$

$$\Gamma \vdash_E b : \sigma \implies$$

$$\Gamma \vdash_E c : \varrho \implies$$

$$\Gamma \vdash_E \text{If } a \ b \ c : \sigma \sqcup \varrho$$

— Call Expressions

|*MetaOperationCallT*:  
 $\text{metaop-type } \tau \text{ op } \sigma \implies$   
 $\Gamma \vdash_E \text{MetaOperationCall } \tau \text{ op} : \sigma$

|*StaticOperationCallT*:  
 $\Gamma \vdash_L \text{params} : \pi \implies$   
 $\text{static-operation } \tau \text{ op } \pi \text{ oper} \implies$   
 $\Gamma \vdash_E \text{StaticOperationCall } \tau \text{ op } \text{params} : \text{oper-type oper}$

|*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*:  
 $\Gamma \vdash_E \text{src} : \langle \mathcal{C} \rangle_{\mathcal{T}}[1] \implies$   
 $\text{attribute } \mathcal{C} \text{ prop } \mathcal{D} \text{ } \tau \implies$   
 $\Gamma \vdash_E \text{AttributeCall } \text{src } \text{DotCall } \text{prop} : \tau$

|*AssociationEndCallT*:  
 $\Gamma \vdash_E \text{src} : \langle \mathcal{C} \rangle_{\mathcal{T}}[1] \implies$   
 $\text{association-end } \mathcal{C} \text{ from role } \mathcal{D} \text{ end} \implies$   
 $\Gamma \vdash_E \text{AssociationEndCall } \text{src } \text{DotCall } \text{from role} : \text{assoc-end-type end}$

|*AssociationClassCallT*:  
 $\Gamma \vdash_E \text{src} : \langle \mathcal{C} \rangle_{\mathcal{T}}[1] \implies$   
 $\text{referred-by-association-class } \mathcal{C} \text{ from } \mathcal{A} \text{ } \mathcal{D} \implies$   
 $\Gamma \vdash_E \text{AssociationClassCall } \text{src } \text{DotCall } \text{from } \mathcal{A} : \text{class-assoc-type } \mathcal{A}$

|*AssociationClassEndCallT*:  
 $\Gamma \vdash_E \text{src} : \langle \mathcal{A} \rangle_{\mathcal{T}}[1] \implies$   
 $\text{association-class-end } \mathcal{A} \text{ role end} \implies$   
 $\Gamma \vdash_E \text{AssociationClassEndCall } \text{src } \text{DotCall } \text{role} : \text{class-assoc-end-type end}$

|*OperationCallT*:  
 $\Gamma \vdash_E \text{src} : \tau \implies$   
 $\Gamma \vdash_L \text{params} : \pi \implies$   
 $\text{op-type } \text{op } k \text{ } \tau \text{ } \pi \text{ } \sigma \implies$   
 $\Gamma \vdash_E \text{OperationCall } \text{src } k \text{ op } \text{params} : \sigma$

|*TupleElementCallT*:  
 $\Gamma \vdash_E \text{src} : \text{Tuple } \pi \implies$   
 $\text{fmlookup } \pi \text{ elem} = \text{Some } \tau \implies$   
 $\Gamma \vdash_E \text{TupleElementCall } \text{src } \text{DotCall } \text{elem} : \tau$

— Iterator Expressions

|*IteratorT*:  
 $\Gamma \vdash_E \text{src} : \tau \implies$   
 $\text{element-type } \tau \text{ } \sigma \implies$   
 $\sigma \leq \text{its-ty} \implies$   
 $\Gamma \text{ ++}_f \text{fmap-of-list } (\text{map } (\lambda it. (it, \text{its-ty})) \text{its}) \vdash_E \text{body} : \varrho \implies$   
 $\Gamma \vdash_I (\text{src}, \text{its}, (\text{Some } \text{its-ty}), \text{body}) : (\tau, \sigma, \varrho)$

|*IterateT*:

$$\begin{aligned} & \Gamma \vdash_I (\text{src}, \text{its}, \text{its-ty}, \text{Let res (Some res-t) res-init body}) : (\tau, \sigma, \varrho) \Longrightarrow \\ & \varrho \leq \text{res-t} \Longrightarrow \\ & \Gamma \vdash_E \text{IterateCall src ArrowCall its its-ty res (Some res-t) res-init body} : \varrho \end{aligned}$$

|*AnyIteratorT*:

$$\begin{aligned} & \Gamma \vdash_I (\text{src}, \text{its}, \text{its-ty}, \text{body}) : (\tau, \sigma, \varrho) \Longrightarrow \\ & \text{length its} \leq 1 \Longrightarrow \\ & \varrho \leq \text{Boolean}[\?] \Longrightarrow \\ & \Gamma \vdash_E \text{AnyIteratorCall src ArrowCall its its-ty body} : \sigma \end{aligned}$$

|*ClosureIteratorT*:

$$\begin{aligned} & \Gamma \vdash_I (\text{src}, \text{its}, \text{its-ty}, \text{body}) : (\tau, \sigma, \varrho) \Longrightarrow \\ & \text{length its} \leq 1 \Longrightarrow \\ & \text{to-single-type } \varrho \leq \sigma \Longrightarrow \\ & \text{to-unique-collection } \tau \ v \Longrightarrow \\ & \Gamma \vdash_E \text{ClosureIteratorCall src ArrowCall its its-ty body} : v \end{aligned}$$

|*CollectIteratorT*:

$$\begin{aligned} & \Gamma \vdash_I (\text{src}, \text{its}, \text{its-ty}, \text{body}) : (\tau, \sigma, \varrho) \Longrightarrow \\ & \text{length its} \leq 1 \Longrightarrow \\ & \text{to-nonunique-collection } \tau \ v \Longrightarrow \\ & \text{update-element-type } v \ (\text{to-single-type } \varrho) \ \varphi \Longrightarrow \\ & \Gamma \vdash_E \text{CollectIteratorCall src ArrowCall its its-ty body} : \varphi \end{aligned}$$

|*CollectNestedIteratorT*:

$$\begin{aligned} & \Gamma \vdash_I (\text{src}, \text{its}, \text{its-ty}, \text{body}) : (\tau, \sigma, \varrho) \Longrightarrow \\ & \text{length its} \leq 1 \Longrightarrow \\ & \text{to-nonunique-collection } \tau \ v \Longrightarrow \\ & \text{update-element-type } v \ \varrho \ \varphi \Longrightarrow \\ & \Gamma \vdash_E \text{CollectNestedIteratorCall src ArrowCall its its-ty body} : \varphi \end{aligned}$$

|*ExistsIteratorT*:

$$\begin{aligned} & \Gamma \vdash_I (\text{src}, \text{its}, \text{its-ty}, \text{body}) : (\tau, \sigma, \varrho) \Longrightarrow \\ & \varrho \leq \text{Boolean}[\?] \Longrightarrow \\ & \Gamma \vdash_E \text{ExistsIteratorCall src ArrowCall its its-ty body} : \varrho \end{aligned}$$

|*ForAllIteratorT*:

$$\begin{aligned} & \Gamma \vdash_I (\text{src}, \text{its}, \text{its-ty}, \text{body}) : (\tau, \sigma, \varrho) \Longrightarrow \\ & \varrho \leq \text{Boolean}[\?] \Longrightarrow \\ & \Gamma \vdash_E \text{ForAllIteratorCall src ArrowCall its its-ty body} : \varrho \end{aligned}$$

|*OneIteratorT*:

$$\begin{aligned} & \Gamma \vdash_I (\text{src}, \text{its}, \text{its-ty}, \text{body}) : (\tau, \sigma, \varrho) \Longrightarrow \\ & \text{length its} \leq 1 \Longrightarrow \\ & \varrho \leq \text{Boolean}[\?] \Longrightarrow \\ & \Gamma \vdash_E \text{OneIteratorCall src ArrowCall its its-ty body} : \text{Boolean}[1] \end{aligned}$$

|*IsUniqueIteratorT*:

$$\begin{aligned} & \Gamma \vdash_I (\text{src}, \text{its}, \text{its-ty}, \text{body}) : (\tau, \sigma, \varrho) \Longrightarrow \\ & \text{length its} \leq 1 \Longrightarrow \\ & \Gamma \vdash_E \text{IsUniqueIteratorCall src ArrowCall its its-ty body} : \text{Boolean}[1] \end{aligned}$$

|*SelectIteratorT*:

$$\begin{aligned} & \Gamma \vdash_I (\text{src}, \text{its}, \text{its-ty}, \text{body}) : (\tau, \sigma, \varrho) \Longrightarrow \\ & \text{length its} \leq 1 \Longrightarrow \\ & \varrho \leq \text{Boolean}[\?] \Longrightarrow \end{aligned}$$

$\Gamma \vdash_E \text{SelectIteratorCall } src \text{ ArrowCall } its \text{ its-ty } body : \tau$   
 $| \text{RejectIteratorT:}$   
 $\Gamma \vdash_I (src, its, its\text{-ty}, body) : (\tau, \sigma, \varrho) \implies$   
 $length\ its \leq 1 \implies$   
 $\varrho \leq \text{Boolean}[?] \implies$   
 $\Gamma \vdash_E \text{RejectIteratorCall } src \text{ ArrowCall } its \text{ its-ty } body : \tau$   
 $| \text{SortedByIteratorT:}$   
 $\Gamma \vdash_I (src, its, its\text{-ty}, body) : (\tau, \sigma, \varrho) \implies$   
 $length\ its \leq 1 \implies$   
 $to\text{-ordered}\text{-collection } \tau \ v \implies$   
 $\Gamma \vdash_E \text{SortedByIteratorCall } src \text{ ArrowCall } its \text{ its-ty } body : \nu$

— Expression Lists

$| \text{ExprListNilT:}$   
 $\Gamma \vdash_L [] : []$   
 $| \text{ExprListConsT:}$   
 $\Gamma \vdash_E expr : \tau \implies$   
 $\Gamma \vdash_L exprs : \pi \implies$   
 $\Gamma \vdash_L expr \# exprs : \tau \# \pi$

### 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]:  $\Gamma \vdash_E \text{EnumLiteral } enm \ lit : \tau$   
**inductive-cases** *CollectionLiteralTE* [elim]:  $\Gamma \vdash_E \text{CollectionLiteral } k \ prts : \tau$   
**inductive-cases** *TupleLiteralTE* [elim]:  $\Gamma \vdash_E \text{TupleLiteral } elems : \tau$   
  
**inductive-cases** *LetTE* [elim]:  $\Gamma \vdash_E \text{Let } v \ \tau \ \text{init } body : \sigma$   
**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]:  $\Gamma \vdash_E \text{MetaOperationCall } \tau \ op : \sigma$   
  
**inductive-cases** *StaticOperationCallTE* [elim]:  $\Gamma \vdash_E \text{StaticOperationCall } \tau \ op \ as : \sigma$   
  
**inductive-cases** *TypeOperationCallTE* [elim]:  $\Gamma \vdash_E \text{TypeOperationCall } a \ k \ op \ \sigma : \tau$   
**inductive-cases** *AttributeCallTE* [elim]:  $\Gamma \vdash_E \text{AttributeCall } src \ k \ prop : \tau$   
**inductive-cases** *AssociationEndCallTE* [elim]:  $\Gamma \vdash_E \text{AssociationEndCall } src \ k \ role \ from : \tau$   
**inductive-cases** *AssociationClassCallTE* [elim]:  $\Gamma \vdash_E \text{AssociationClassCall } src \ k$

$a$  from :  $\tau$

**inductive-cases** *AssociationClassEndCallTE* [elim]:  $\Gamma \vdash_E$  *AssociationClassEndCall*  $src$   $k$   $role$  :  $\tau$

**inductive-cases** *OperationCallTE* [elim]:  $\Gamma \vdash_E$  *OperationCall*  $src$   $k$   $op$   $params$  :  $\tau$

**inductive-cases** *TupleElementCallTE* [elim]:  $\Gamma \vdash_E$  *TupleElementCall*  $src$   $k$   $elem$  :  $\tau$

**inductive-cases** *IteratorTE* [elim]:  $\Gamma \vdash_I$  ( $src$ ,  $its$ ,  $body$ ) :  $ys$

**inductive-cases** *IterateTE* [elim]:  $\Gamma \vdash_E$  *IterateCall*  $src$   $k$   $its$   $its-ty$   $res$   $res-t$   $res-init$   $body$  :  $\tau$

**inductive-cases** *AnyIteratorTE* [elim]:  $\Gamma \vdash_E$  *AnyIteratorCall*  $src$   $k$   $its$   $its-ty$   $body$  :  $\tau$

**inductive-cases** *ClosureIteratorTE* [elim]:  $\Gamma \vdash_E$  *ClosureIteratorCall*  $src$   $k$   $its$   $its-ty$   $body$  :  $\tau$

**inductive-cases** *CollectIteratorTE* [elim]:  $\Gamma \vdash_E$  *CollectIteratorCall*  $src$   $k$   $its$   $its-ty$   $body$  :  $\tau$

**inductive-cases** *CollectNestedIteratorTE* [elim]:  $\Gamma \vdash_E$  *CollectNestedIteratorCall*  $src$   $k$   $its$   $its-ty$   $body$  :  $\tau$

**inductive-cases** *ExistsIteratorTE* [elim]:  $\Gamma \vdash_E$  *ExistsIteratorCall*  $src$   $k$   $its$   $its-ty$   $body$  :  $\tau$

**inductive-cases** *ForAllIteratorTE* [elim]:  $\Gamma \vdash_E$  *ForAllIteratorCall*  $src$   $k$   $its$   $its-ty$   $body$  :  $\tau$

**inductive-cases** *OneIteratorTE* [elim]:  $\Gamma \vdash_E$  *OneIteratorCall*  $src$   $k$   $its$   $its-ty$   $body$  :  $\tau$

**inductive-cases** *IsUniqueIteratorTE* [elim]:  $\Gamma \vdash_E$  *IsUniqueIteratorCall*  $src$   $k$   $its$   $its-ty$   $body$  :  $\tau$

**inductive-cases** *SelectIteratorTE* [elim]:  $\Gamma \vdash_E$  *SelectIteratorCall*  $src$   $k$   $its$   $its-ty$   $body$  :  $\tau$

**inductive-cases** *RejectIteratorTE* [elim]:  $\Gamma \vdash_E$  *RejectIteratorCall*  $src$   $k$   $its$   $its-ty$   $body$  :  $\tau$

**inductive-cases** *SortedByIteratorTE* [elim]:  $\Gamma \vdash_E$  *SortedByIteratorCall*  $src$   $k$   $its$   $its-ty$   $body$  :  $\tau$

**inductive-cases** *CollectionPartsNilTE* [elim]:  $\Gamma \vdash_C$  [ $x$ ] :  $\tau$

**inductive-cases** *CollectionPartsItemTE* [elim]:  $\Gamma \vdash_C$   $x$  #  $y$  #  $xs$  :  $\tau$

**inductive-cases** *CollectionItemTE* [elim]:  $\Gamma \vdash_P$  *CollectionItem*  $a$  :  $\tau$

**inductive-cases** *CollectionRangeTE* [elim]:  $\Gamma \vdash_P$  *CollectionRange*  $a$   $b$  :  $\tau$

**inductive-cases** *ExprListTE* [elim]:  $\Gamma \vdash_L$   $exprs$  :  $\pi$

## 6.4 Simplification Rules

**inductive-simps** *typing-alt-simps*:

$\Gamma \vdash_E$  *NullLiteral* :  $\tau$

$\Gamma \vdash_E$  *BooleanLiteral*  $c$  :  $\tau$

$\Gamma \vdash_E$  *RealLiteral*  $c$  :  $\tau$

$\Gamma \vdash_E$  *UnlimitedNaturalLiteral*  $c$  :  $\tau$

$$\begin{aligned}
&\Gamma \vdash_E \text{IntegerLiteral } c : \tau \\
&\Gamma \vdash_E \text{StringLiteral } c : \tau \\
&\Gamma \vdash_E \text{EnumLiteral } \text{enm } \text{lit} : \tau \\
&\Gamma \vdash_E \text{CollectionLiteral } k \text{ 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 } \text{body} : \sigma \\
&\Gamma \vdash_E \text{Var } v : \tau \\
&\Gamma \vdash_E \text{If } a \ b \ c : \tau \\
\\
&\Gamma \vdash_E \text{MetaOperationCall } \tau \ \text{op} : \sigma \\
&\Gamma \vdash_E \text{StaticOperationCall } \tau \ \text{op } \text{as} : \sigma \\
\\
&\Gamma \vdash_E \text{TypeOperationCall } a \ k \ \text{op } \sigma : \tau \\
&\Gamma \vdash_E \text{AttributeCall } \text{src } k \ \text{prop} : \tau \\
&\Gamma \vdash_E \text{AssociationEndCall } \text{src } k \ \text{role } \text{from} : \tau \\
&\Gamma \vdash_E \text{AssociationClassCall } \text{src } k \ a \ \text{from} : \tau \\
&\Gamma \vdash_E \text{AssociationClassEndCall } \text{src } k \ \text{role} : \tau \\
&\Gamma \vdash_E \text{OperationCall } \text{src } k \ \text{op } \text{params} : \tau \\
&\Gamma \vdash_E \text{TupleElementCall } \text{src } k \ \text{elem} : \tau \\
\\
&\Gamma \vdash_I (\text{src}, \text{its}, \text{body}) : \text{ys} \\
&\Gamma \vdash_E \text{IterateCall } \text{src } k \ \text{its } \text{its-ty } \text{res } \text{res-t } \text{res-init } \text{body} : \tau \\
&\Gamma \vdash_E \text{AnyIteratorCall } \text{src } k \ \text{its } \text{its-ty } \text{body} : \tau \\
&\Gamma \vdash_E \text{ClosureIteratorCall } \text{src } k \ \text{its } \text{its-ty } \text{body} : \tau \\
&\Gamma \vdash_E \text{CollectIteratorCall } \text{src } k \ \text{its } \text{its-ty } \text{body} : \tau \\
&\Gamma \vdash_E \text{CollectNestedIteratorCall } \text{src } k \ \text{its } \text{its-ty } \text{body} : \tau \\
&\Gamma \vdash_E \text{ExistsIteratorCall } \text{src } k \ \text{its } \text{its-ty } \text{body} : \tau \\
&\Gamma \vdash_E \text{ForAllIteratorCall } \text{src } k \ \text{its } \text{its-ty } \text{body} : \tau \\
&\Gamma \vdash_E \text{OneIteratorCall } \text{src } k \ \text{its } \text{its-ty } \text{body} : \tau \\
&\Gamma \vdash_E \text{IsUniqueIteratorCall } \text{src } k \ \text{its } \text{its-ty } \text{body} : \tau \\
&\Gamma \vdash_E \text{SelectIteratorCall } \text{src } k \ \text{its } \text{its-ty } \text{body} : \tau \\
&\Gamma \vdash_E \text{RejectIteratorCall } \text{src } k \ \text{its } \text{its-ty } \text{body} : \tau \\
&\Gamma \vdash_E \text{SortedByIteratorCall } \text{src } k \ \text{its } \text{its-ty } \text{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:*

```

     $\Gamma \vdash_E \text{expr} : \tau \implies$ 
     $\Gamma \vdash_E \text{expr} : \sigma \implies \tau = \sigma$  and
collection-parts-typing-det:
     $\Gamma \vdash_C \text{prts} : \tau \implies$ 
     $\Gamma \vdash_C \text{prts} : \sigma \implies \tau = \sigma$  and
collection-part-typing-det:
     $\Gamma \vdash_P \text{prt} : \tau \implies$ 
     $\Gamma \vdash_P \text{prt} : \sigma \implies \tau = \sigma$  and
iterator-typing-det:
     $\Gamma \vdash_I (\text{src}, \text{its}, \text{body}) : xs \implies$ 
     $\Gamma \vdash_I (\text{src}, \text{its}, \text{body}) : ys \implies xs = ys$  and
expr-list-typing-det:
     $\Gamma \vdash_L \text{exprs} : \pi \implies$ 
     $\Gamma \vdash_L \text{exprs} : \xi \implies \pi = \xi$ 
proof (induct arbitrary:  $\sigma$  and  $\sigma$  and  $\sigma$  and  $ys$  and  $\xi$ 
    rule: typing-collection-parts-typing-collection-part-typing-iterator-typing-expr-list-typing.inducts)
  case (NullLiteralT  $\Gamma$ ) thus ?case by auto
next
  case (BooleanLiteralT  $\Gamma$   $c$ ) thus ?case by auto
next
  case (RealLiteralT  $\Gamma$   $c$ ) thus ?case by auto
next
  case (IntegerLiteralT  $\Gamma$   $c$ ) thus ?case by auto
next
  case (UnlimitedNaturalLiteralT  $\Gamma$   $c$ ) thus ?case by auto
next
  case (StringLiteralT  $\Gamma$   $c$ ) thus ?case by auto
next
  case (EnumLiteralT  $\Gamma$   $\tau$  lit) thus ?case by auto
next
  case (SetLiteralT  $\Gamma$  prts  $\tau$ ) thus ?case by blast
next
  case (OrderedSetLiteralT  $\Gamma$  prts  $\tau$ ) thus ?case by blast
next
  case (BagLiteralT  $\Gamma$  prts  $\tau$ ) thus ?case by blast
next
  case (SequenceLiteralT  $\Gamma$  prts  $\tau$ ) thus ?case by blast
next
  case (CollectionPartsSingletonT  $\Gamma$   $x$   $\tau$ ) thus ?case by blast
next
  case (CollectionPartsListT  $\Gamma$   $x$   $\tau$   $y$   $xs$   $\sigma$ ) thus ?case by blast
next
  case (CollectionPartItemT  $\Gamma$   $a$   $\tau$ ) thus ?case by blast
next
  case (CollectionPartRangeT  $\Gamma$   $a$   $\tau$   $b$   $\sigma$ ) thus ?case by blast
next
  case (TupleLiteralNilT  $\Gamma$ ) thus ?case by auto
next
  case (TupleLiteralConsT  $\Gamma$  elems  $\xi$  el  $\tau$ ) show ?case

```

```

    apply (insert TupleLiteralConstT.premis)
    apply (erule TupleLiteralTE, simp)
    using TupleLiteralConstT.hyps by auto
next
  case (LetT  $\Gamma$   $\mathcal{M}$  init  $\sigma$   $\tau$  v body  $\varrho$ ) thus ?case by blast
next
  case (VarT  $\Gamma$  v  $\tau$   $\mathcal{M}$ ) thus ?case by auto
next
  case (IfT  $\Gamma$  a  $\tau$  b  $\sigma$  c  $\varrho$ ) thus ?case
    apply (insert IfT.premis)
    apply (erule IfTE)
    by (simp add: IfT.hyps)
next
  case (MetaOperationCallT  $\tau$  op  $\sigma$   $\Gamma$ ) thus ?case
    by (metis MetaOperationCallTE mataop-type.cases)
next
  case (StaticOperationCallT  $\tau$  op  $\pi$  oper  $\Gamma$  as) thus ?case
    apply (insert StaticOperationCallT.premis)
    apply (erule StaticOperationCallTE)
    using StaticOperationCallT.hyps static-operation-det by blast
next
  case (TypeOperationCallT  $\Gamma$  a  $\tau$  op  $\sigma$   $\varrho$ ) thus ?case
    by (metis TypeOperationCallTE typeop-type-det)
next
  case (AttributeCallT  $\Gamma$  src  $\tau$   $\mathcal{C}$  prop  $\mathcal{D}$   $\sigma$ ) show ?case
    apply (insert AttributeCallT.premis)
    apply (erule AttributeCallTE)
    using AttributeCallT.hyps attribute-det by blast
next
  case (AssociationEndCallT  $\Gamma$  src  $\mathcal{C}$  from role  $\mathcal{D}$  end) show ?case
    apply (insert AssociationEndCallT.premis)
    apply (erule AssociationEndCallTE)
    using AssociationEndCallT.hyps association-end-det by blast
next
  case (AssociationClassCallT  $\Gamma$  src  $\mathcal{C}$  from  $\mathcal{A}$ ) thus ?case by blast
next
  case (AssociationClassEndCallT  $\Gamma$  src  $\tau$   $\mathcal{A}$  role end) show ?case
    apply (insert AssociationClassEndCallT.premis)
    apply (erule AssociationClassEndCallTE)
    using AssociationClassEndCallT.hyps association-class-end-det by blast
next
  case (OperationCallT  $\Gamma$  src  $\tau$  params  $\pi$  op k) show ?case
    apply (insert OperationCallT.premis)
    apply (erule OperationCallTE)
    using OperationCallT.hyps op-type-det by blast
next
  case (TupleElementCallT  $\Gamma$  src  $\pi$  elem  $\tau$ ) thus ?case
    apply (insert TupleElementCallT.premis)
    apply (erule TupleElementCallTE)

```



```

    using TupleElementCallT.hyps by fastforce
next
case (IteratorT  $\Gamma$  src  $\tau$   $\sigma$  its-ty its body  $\varrho$ ) show ?case
  apply (insert IteratorT.prem)
  apply (erule IteratorTE)
  using IteratorT.hyps element-type-det by blast
next
case (IterateT  $\Gamma$  src its its-ty res res-t res-init body  $\tau$   $\sigma$   $\varrho$ ) show ?case
  apply (insert IterateT.prem)
  using IterateT.hyps by blast
next
case (AnyIteratorT  $\Gamma$  src its its-ty body  $\tau$   $\sigma$   $\varrho$ ) thus ?case
  by (meson AnyIteratorTE Pair-inject)
next
case (ClosureIteratorT  $\Gamma$  src its its-ty body  $\tau$   $\sigma$   $\varrho$   $v$ ) show ?case
  apply (insert ClosureIteratorT.prem)
  apply (erule ClosureIteratorTE)
  using ClosureIteratorT.hyps to-unique-collection-det by blast
next
case (CollectIteratorT  $\Gamma$  src its its-ty body  $\tau$   $\sigma$   $\varrho$   $v$ ) show ?case
  apply (insert CollectIteratorT.prem)
  apply (erule CollectIteratorTE)
  using CollectIteratorT.hyps to-nonunique-collection-det
  update-element-type-det Pair-inject by metis
next
case (CollectNestedIteratorT  $\Gamma$  src its its-ty body  $\tau$   $\sigma$   $\varrho$   $v$ ) show ?case
  apply (insert CollectNestedIteratorT.prem)
  apply (erule CollectNestedIteratorTE)
  using CollectNestedIteratorT.hyps to-nonunique-collection-det
  update-element-type-det Pair-inject by metis
next
case (ExistsIteratorT  $\Gamma$  src its its-ty body  $\tau$   $\sigma$   $\varrho$ ) show ?case
  apply (insert ExistsIteratorT.prem)
  apply (erule ExistsIteratorTE)
  using ExistsIteratorT.hyps Pair-inject by metis
next
case (ForAllIteratorT  $\Gamma$   $\mathcal{M}$  src its its-ty body  $\tau$   $\sigma$   $\varrho$ ) show ?case
  apply (insert ForAllIteratorT.prem)
  apply (erule ForAllIteratorTE)
  using ForAllIteratorT.hyps Pair-inject by metis
next
case (OneIteratorT  $\Gamma$   $\mathcal{M}$  src its its-ty body  $\tau$   $\sigma$   $\varrho$ ) show ?case
  apply (insert OneIteratorT.prem)
  apply (erule OneIteratorTE)
  by simp
next
case (IsUniqueIteratorT  $\Gamma$   $\mathcal{M}$  src its its-ty body  $\tau$   $\sigma$   $\varrho$ ) show ?case
  apply (insert IsUniqueIteratorT.prem)
  apply (erule IsUniqueIteratorTE)

```

```

    by simp
  next
  case (SelectIteratorT  $\Gamma$   $\mathcal{M}$  src its its-ty body  $\tau$   $\sigma$   $\varrho$ ) show ?case
    apply (insert SelectIteratorT.prem)
    apply (erule SelectIteratorTE)
    using SelectIteratorT.hyps by blast
  next
  case (RejectIteratorT  $\Gamma$   $\mathcal{M}$  src its its-ty body  $\tau$   $\sigma$   $\varrho$ ) show ?case
    apply (insert RejectIteratorT.prem)
    apply (erule RejectIteratorTE)
    using RejectIteratorT.hyps by blast
  next
  case (SortedByIteratorT  $\Gamma$   $\mathcal{M}$  src its its-ty body  $\tau$   $\sigma$   $\varrho$   $v$ ) show ?case
    apply (insert SortedByIteratorT.prem)
    apply (erule SortedByIteratorTE)
    using SortedByIteratorT.hyps to-ordered-collection-det by blast
  next
  case (ExprListNilT  $\Gamma$ ) thus ?case
    using expr-list-typing.cases by auto
  next
  case (ExprListConsT  $\Gamma$  expr  $\tau$  exprs  $\pi$ ) show ?case
    apply (insert ExprListConsT.prem)
    apply (erule ExprListTE)
    by (simp-all add: ExprListConsT.hyps)
qed

```

## 6.6 Code Setup

code-pred *op-type* .

```

code-pred (modes:
   $i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}$ ,
   $i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$ ) iterator-typing .

```

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*:

$$\begin{aligned} &\Gamma \vdash \text{Literal } a \Rightarrow b \\ &\Gamma \vdash \text{Let } v \ t \ \text{init } \text{body} \Rightarrow b \\ &\Gamma \vdash \text{Var } v \Rightarrow b \\ &\Gamma \vdash \text{If } a \ b \ c \Rightarrow d \\ \\ &\Gamma \vdash \text{MetaOperationCall } \tau \ \text{op} \Rightarrow b \\ &\Gamma \vdash \text{StaticOperationCall } \tau \ \text{op } \text{as} \Rightarrow b \\ &\Gamma \vdash \text{Call } \text{src } \text{DotCall } \text{call} \Rightarrow b \\ &\Gamma \vdash \text{Call } \text{src } \text{SafeDotCall } \text{call} \Rightarrow b \\ &\Gamma \vdash \text{Call } \text{src } \text{ArrowCall } \text{call} \Rightarrow b \\ &\Gamma \vdash \text{Call } \text{src } \text{SafeArrowCall } \text{call} \Rightarrow b \\ \\ &(\Gamma, \tau) \vdash_C \text{call} \Rightarrow b \\ &(\Gamma, \tau) \vdash_C \text{Operation } \text{op } \text{as} \Rightarrow \text{call} \\ &(\Gamma, \tau) \vdash_C \text{Iterate } \text{its } \text{its-ty } \text{res } \text{res-t } \text{res-init } \text{body} \Rightarrow \text{call} \\ &(\Gamma, \tau) \vdash_C \text{Iterator } \text{iter } \text{its } \text{its-ty } \text{body} \Rightarrow \text{call} \\ \\ &\Gamma \vdash_L [] \Rightarrow \text{ys} \\ &\Gamma \vdash_L x \# \text{xs} \Rightarrow \text{ys} \end{aligned}$$

### 7.4 Determinism

**lemma** *any-has-not-element-type*:

$$\begin{aligned} &\text{element-type } \tau \ \sigma \Longrightarrow \tau \leq \text{OclAny}[?] \vee \tau \leq \text{Tuple } \text{fmempty} \Longrightarrow \text{False} \\ &\text{by } (\text{erule } \text{element-type.cases}; \text{ auto}) \end{aligned}$$

**lemma** *any-has-not-element-type'*:

$$\begin{aligned} &\text{element-type } \tau \ \sigma \Longrightarrow \text{OclVoid}[?] \leq \tau \Longrightarrow \text{False} \\ &\text{by } (\text{erule } \text{element-type.cases}; \text{ auto}) \end{aligned}$$

**lemma**

*normalize-det*:

$$\begin{aligned} &\Gamma \vdash \text{expr} \Rightarrow \text{expr}_1 \Longrightarrow \\ &\Gamma \vdash \text{expr} \Rightarrow \text{expr}_2 \Longrightarrow \text{expr}_1 = \text{expr}_2 \quad \text{and} \end{aligned}$$

*normalize-call-det*:

$$\begin{aligned} &\Gamma\text{-}\tau \vdash_C \text{call} \Rightarrow \text{call}_1 \Longrightarrow \\ &\Gamma\text{-}\tau \vdash_C \text{call} \Rightarrow \text{call}_2 \Longrightarrow \text{call}_1 = \text{call}_2 \quad \text{and} \end{aligned}$$

*normalize-expr-list-det*:

$$\begin{aligned} &\Gamma \vdash_L \text{xs} \Rightarrow \text{ys} \Longrightarrow \\ &\Gamma \vdash_L \text{xs} \Rightarrow \text{zs} \Longrightarrow \text{ys} = \text{zs} \end{aligned}$$

**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** (*induct arbitrary: expr<sub>2</sub> and call<sub>2</sub> and zs*

*rule: normalize-normalize-call-normalize-expr-list.inducts*)

**case** (*LiteralN*  $\Gamma \ a$ ) **thus** ?*case* **by** *auto*

**next**



```

  case (ExplicitlyTypedLetN  $\Gamma$  init1 init2 v  $\tau$  body1 body2) thus ?case
    by blast
next
  case (ImplicitlyTypedLetN  $\Gamma$  init1 init2  $\tau$  v body1 body2) thus ?case
    by (metis (mono-tags, lifting) LetNE option.distinct(1) typing-det)
next
  case (VarN  $\Gamma$  v) thus ?case by auto
next
  case (IfN  $\Gamma$  a1 a2 b1 b2 c1 c2) thus ?case
    apply (insert IfN.prems)
    apply (erule IfNE)
    by (simp add: IfN.hyps)
next
  case (MetaOperationCallN  $\Gamma$   $\tau$  op) thus ?case by auto
next
  case (StaticOperationCallN  $\Gamma$  params1 params2  $\tau$  op) thus ?case by blast
next
  case (OclAnyDotCallN  $\Gamma$  src1 src2  $\tau$  call1 call2) show ?case
    apply (insert OclAnyDotCallN.prems)
    apply (erule DotCallNE)
    using OclAnyDotCallN.hyps typing-det apply metis
    using OclAnyDotCallN.hyps any-has-not-element-type typing-det by metis
next
  case (OclAnySafeDotCallN  $\Gamma$  src1 src2  $\tau$  call1 call2) show ?case
    apply (insert OclAnySafeDotCallN.prems)
    apply (erule SafeDotCallNE)
    using OclAnySafeDotCallN.hyps typing-det comp-apply
    apply (metis (no-types, lifting) list.simps(8) list.simps(9))
    using OclAnySafeDotCallN.hyps typing-det any-has-not-element-type'
    by metis
next
  case (OclAnyArrowCallN  $\Gamma$  src1 src2  $\tau$  src3  $\sigma$  call1 call2) show ?case
    apply (insert OclAnyArrowCallN.prems)
    apply (erule ArrowCallNE)
    using OclAnyArrowCallN.hyps typing-det comp-apply apply metis
    using OclAnyArrowCallN.hyps typing-det any-has-not-element-type
    by metis
next
  case (CollectionArrowCallN  $\Gamma$  src1 src2  $\tau$  wu call1 call2) show ?case
    apply (insert CollectionArrowCallN.prems)
    apply (erule ArrowCallNE)
    using CollectionArrowCallN.hyps typing-det any-has-not-element-type
    apply metis
    using CollectionArrowCallN.hyps typing-det by metis
next
  case (CollectionSafeArrowCallN  $\Gamma$  src1 src2  $\tau$   $\sigma$  src3  $\rho$  call1 call2) show ?case
    apply (insert CollectionSafeArrowCallN.prems)
    apply (erule SafeArrowCallNE)
    using CollectionSafeArrowCallN.hyps typing-det element-type-det by metis

```

```

next
  case (CollectionDotCallN  $\Gamma$   $src_1$   $src_2$   $\tau$   $\sigma$   $call_1$   $call_2$   $it$ ) show ?case
    apply (insert CollectionDotCallN.prem $s$ )
    apply (erule DotCallNE)
    using CollectionDotCallN.hyps typing-det any-has-not-element-type
    apply metis
    using CollectionDotCallN.hyps typing-det element-type-det by metis
next
  case (CollectionSafeDotCallN  $\Gamma$   $src_1$   $src_2$   $\tau$   $\sigma$   $src_3$   $call_1$   $call_2$   $it$ ) show ?case
    apply (insert CollectionSafeDotCallN.prem $s$ )
    apply (erule SafeDotCallNE)
    using CollectionSafeDotCallN.hyps typing-det any-has-not-element-type'
    apply metis
    using CollectionSafeDotCallN.hyps typing-det element-type-det by metis
next
  case (TypeOperationN  $\Gamma$   $\tau$   $op$   $ty$ ) thus ?case by auto
next
  case (AttributeN  $\Gamma$   $\tau$   $attr$ ) thus ?case by auto
next
  case (AssociationEndN  $\Gamma$   $\tau$   $role$   $from$ ) thus ?case by auto
next
  case (AssociationClassN  $\Gamma$   $\tau$   $\mathcal{A}$   $from$ ) thus ?case by auto
next
  case (AssociationClassEndN  $\Gamma$   $\tau$   $role$ ) thus ?case by auto
next
  case (OperationN  $\Gamma$   $params_1$   $params_2$   $\tau$   $op$ ) thus ?case by blast
next
  case (TupleElementN  $\Gamma$   $\tau$   $elem$ ) thus ?case by auto
next
  case (ExplicitlyTypedIterateN
     $\Gamma$   $res-init_1$   $res-init_2$   $\sigma$   $its$   $res$   $res-t_1$   $body_1$   $res-t_2$   $body_2$   $\tau$ )
  show ?case
    apply (insert ExplicitlyTypedIterateN.prem $s$ )
    apply (erule IterateCallNE)
    using ExplicitlyTypedIterateN.hyps element-type-det by blast+
next
  case (ImplicitlyTypedIterateN
     $\tau$   $\sigma$   $\Gamma$   $res-init_1$   $res-init_2$   $its$   $res$   $res-t_1$   $body_1$   $res-t_2$   $body_2$ )
  show ?case
    apply (insert ImplicitlyTypedIterateN.prem $s$ )
    apply (erule IterateCallNE)
    using ImplicitlyTypedIterateN.hyps element-type-det by blast+
next
  case (ExplicitlyTypedIteratorN  $\Gamma$   $\sigma$   $its$   $body_1$   $body_2$   $\tau$   $iter$ )
  show ?case
    apply (insert ExplicitlyTypedIteratorN.prem $s$ )
    apply (erule IteratorCallNE)
    using ExplicitlyTypedIteratorN.hyps element-type-det by blast+
next

```

```

case (ImplicitlyTypedIteratorN  $\tau$   $\sigma$   $\Gamma$  its body1 body2 iter)
show ?case
  apply (insert ImplicitlyTypedIteratorN.prems)
  apply (erule IteratorCallNE)
  using ImplicitlyTypedIteratorN.hyps element-type-det by blast+
next
  case (ExprListNilN  $\Gamma$ ) thus ?case
    using normalize-expr-list.cases by auto
next
  case (ExprListConsN  $\Gamma$  x y xs ys) thus ?case by blast
qed

```

## 7.5 Normalized Expressions Typing

Here is the final typing rules.

```

inductive nf-typing ( (1-/  $\vdash$ / (-  $:$ / -)) [51,51,51] 50) where
   $\Gamma \vdash \text{expr} \Rightarrow \text{expr}_N \Longrightarrow$ 
   $\Gamma \vdash_E \text{expr}_N : \tau \Longrightarrow$ 
   $\Gamma \vdash \text{expr} : \tau$ 

```

**lemma** *nf-typing-det*:

```

 $\Gamma \vdash \text{expr} : \tau \Longrightarrow$ 
 $\Gamma \vdash \text{expr} : \sigma \Longrightarrow \tau = \sigma$ 
by (metis nf-typing.cases normalize-det typing-det)

```

## 7.6 Code Setup

**code-pred** *normalize* .

**code-pred** *nf-typing* .

**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  $c\ d :: \text{classes1}$ 
unfolding less-classes1-def less-eq-classes1-def
using subclass1.simps by auto

```

```

lemma order-refl-classes1:
     $c \leq c$ 
for  $c :: \text{classes1}$ 
unfolding less-eq-classes1-def by simp

```

```

lemma order-trans-classes1:
     $c \leq d \implies d \leq e \implies c \leq e$ 
for  $c\ d\ e :: \text{classes1}$ 
unfolding less-eq-classes1-def
using subclass1.simps by auto

```

```

lemma antisym-classes1:
     $c \leq d \implies d \leq c \implies c = d$ 
for  $c\ d :: \text{classes1}$ 
unfolding less-eq-classes1-def
using subclass1.simps by auto

```

```

lemma sup-ge1-classes1:
     $c \leq c \sqcup d$ 
for  $c\ d :: \text{classes1}$ 
by (induct c; auto simp add: less-eq-classes1-def less-classes1-def subclass1.simps)

```

```

lemma sup-ge2-classes1:
     $d \leq c \sqcup d$ 
for  $c\ d :: \text{classes1}$ 
by (induct c; auto simp add: less-eq-classes1-def less-classes1-def subclass1.simps)

```

```

lemma sup-least-classes1:
     $c \leq e \implies d \leq e \implies c \sqcup d \leq e$ 
for  $c\ d\ e :: \text{classes1}$ 
by (induct c; induct d;
    auto simp add: less-eq-classes1-def less-classes1-def subclass1.simps)

```

```

instance
apply intro-classes
apply (simp add: less-le-not-le-classes1)
apply (simp add: order-refl-classes1)
apply (rule order-trans-classes1; auto)
apply (simp add: antisym-classes1)
apply (simp add: sup-ge1-classes1)
apply (simp add: sup-ge2-classes1)
by (simp add: sup-least-classes1)

```

```

end

```

**code-pred** *subclass1* .

```
fun subclass1-fun where
  subclass1-fun Object C = False
| subclass1-fun Person C = (C = Object)
| subclass1-fun Employee C = (C = Object ∨ C = Person)
| subclass1-fun Customer C = (C = Object ∨ C = Person)
| subclass1-fun Project C = (C = Object)
| subclass1-fun Task C = (C = Object)
| subclass1-fun Sprint C = (C = Object)
```

**lemma** *less-classes1-code* [*code*]:

(*<*) = *subclass1-fun*

**proof** (*intro ext iffI*)

**fix** C D :: *classes1*

**show** C < D ⇒ *subclass1-fun* C D

**unfolding** *less-classes1-def*

**apply** (*erule subclass1.cases, auto*)

**using** *subclass1-fun.elims(3)* **by** *blast*

**show** *subclass1-fun* C D ⇒ C < D

**by** (*erule subclass1-fun.elims, auto simp add: less-classes1-def subclass1.intros*)

**qed**

**lemma** *less-eq-classes1-code* [*code*]:

(*≤*) = ( $\lambda x y. \textit{subclass1-fun } x y \vee x = y$ )

**unfolding** *dual-order.order-iff-strict less-classes1-code*

**by** *auto*

## 8.2 Object Model

**abbreviation**  $\Gamma_0 \equiv \textit{fmempty} :: \textit{classes1 type env}$

**declare** [[*coercion* *ObjectType* :: *classes1* ⇒ *classes1 basic-type* ]]

**declare** [[*coercion* *phantom* :: *String.literal* ⇒ *classes1 enum* ]]

**instantiation** *classes1* :: *ocl-object-model*

**begin**

**definition** *classes-classes1* ≡

{|*Object, Person, Employee, Customer, Project, Task, Sprint*|}

**definition** *attributes-classes1* ≡ *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"  $\mapsto_f$  [
  STR "project"  $\mapsto_f$  (Project, 1, 1, False, False),
  STR "tasks"  $\mapsto_f$  (Task, 0,  $\infty$ , True, True)],
STR "SprintTaskAssignee"  $\mapsto_f$  [
  STR "sprint"  $\mapsto_f$  (Sprint, 0, 10, False, True),
  STR "tasks"  $\mapsto_f$  (Task, 0, 5, False, True),
  STR "assignee"  $\mapsto_f$  (Employee, 0, 1, False, False)]]
```

**definition** *associations-classes1*  $\equiv$  *assocs*

**definition** *association-classes-classes1*  $\equiv$  *fmempty* :: *classes1*  $\rightarrow_f$  *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*  $\equiv$  [

```
(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*  $\equiv$  *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 assoc  $\implies$ 
fmlookup assoc assoc = Some ends  $\implies$ 
role |∈| fmdom ends  $\implies$ 
fmlookup ends role = Some end  $\implies$ 
assoc-end-min end  $\leq$  assoc-end-max end

```

**unfolding** *assoc-end-min-def* *assoc-end-max-def*

**using** *zero-enat-def* *one-enat-def* *numeral-eq-enat* **by** *auto*

**lemma** *association-ends-unique*:

```

assumes association-ends' classes assoc C from role end1
and association-ends' classes assoc C from role end2
shows end1 = end2

```

**proof** –

```

have  $\neg$  association-ends-not-unique' classes assoc by eval
with assms show ?thesis
using association-ends-not-unique'.simps by blast

```

**qed**

**instance**

```

apply standard
unfolding associations-classes1-def
using assoc-end-min-less-eq-max apply blast
using association-ends-unique by blast

```

**end**

## 8.3 Simplification Rules

**lemma** *ex-alt-simps* [*simp*]:

```

 $\exists a. a$ 
 $\exists a. \neg a$ 
 $(\exists a. (a \longrightarrow P) \wedge a) = P$ 
 $(\exists a. \neg a \wedge (\neg a \longrightarrow P)) = P$ 
by auto

```

```

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"  $\mathcal{D} \tau =$ 
  ( $\mathcal{D} = \text{Employee} \wedge \tau = \text{String}[1]$ )
proof –
  have attribute Employee STR "name" Employee String[1]
  by eval
  thus ?thesis
  using attribute-det by blast
qed

```

```

lemma association-end-Project-members [simp]:
  association-end Project None STR "members" D τ =
    (D = Project ∧ τ = (Employee, 1, 20, True, True))
proof –
  have association-end Project None STR "members"
    Project (Employee, 1, 20, True, True)
  by eval
  thus ?thesis
  using association-end-det by blast
qed

```

```

lemma association-end-Employee-projects-simp [simp]:
  association-end Employee None STR "projects" D τ =
    (D = Employee ∧ τ = (Project, 0, ∞, False, True))
proof –
  have association-end Employee None STR "projects"
    Employee (Project, 0, ∞, False, True)
  by eval
  thus ?thesis
  using association-end-det by blast
qed

```

```

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 –
  have static-operation ⟨Project⟩τ[1] STR "allProjects" []
    (STR "allProjects", ⟨Project⟩τ[1], [], Set ⟨Project⟩τ[1], True,
      Some (MetaOperationCall ⟨Project⟩τ[1] AllInstancesOp))
  by eval
  thus ?thesis
  using static-operation-det by blast
qed

```

## 8.4 Basic Types

### 8.4.1 Positive Cases

```

lemma UnlimitedNatural < (Real :: classes1 basic-type) by simp
lemma ⟨Employee⟩τ < ⟨Person⟩τ by auto
lemma ⟨Person⟩τ ≤ OclAny by simp

```

### 8.4.2 Negative Cases

```

lemma  $\neg$  String ≤ (Boolean :: classes1 basic-type) by simp

```

## 8.5 Types

### 8.5.1 Positive Cases

**lemma**  $Integer[?] < (OclSuper :: classes1\ type)$  **by simp**  
**lemma**  $Collection\ Real[?] < (OclSuper :: classes1\ type)$  **by simp**  
**lemma**  $Set\ (Collection\ Boolean[1]) < (OclSuper :: classes1\ type)$  **by simp**  
**lemma**  $Set\ (Bag\ Boolean[1]) < Set\ (Collection\ Boolean[?] :: classes1\ type)$   
**by simp**  
**lemma**  $Tuple\ (fmap-of-list\ [(STR\ "a",\ Boolean[1]),\ (STR\ "b",\ Integer[1])]) <$   
 $Tuple\ (fmap-of-list\ [(STR\ "a",\ Boolean[?] :: classes1\ type)])$  **by eval**

**lemma**  $Integer[1] \sqcup (Real[?] :: classes1\ type) = Real[?]$  **by simp**  
**lemma**  $Set\ Integer[1] \sqcup Set\ (Real[1] :: classes1\ type) = Set\ Real[1]$  **by simp**  
**lemma**  $Set\ Integer[1] \sqcup Bag\ (Boolean[?] :: classes1\ type) = Collection\ OclAny[?]$   
**by simp**  
**lemma**  $Set\ Integer[1] \sqcup (Real[1] :: classes1\ type) = OclSuper$  **by simp**

### 8.5.2 Negative Cases

**lemma**  $\neg\ OrderedSet\ Boolean[1] < Set\ (Boolean[1] :: classes1\ type)$  **by simp**

## 8.6 Typing

### 8.6.1 Positive Cases

$E1 :: A : E1[1]$

**lemma**  
 $\Gamma_0 \vdash EnumLiteral\ STR\ "E1"\ STR\ "A" : (Enum\ STR\ "E1")[1]$   
**by simp**

$true\ or\ false : Boolean[1]$

**lemma**  
 $\Gamma_0 \vdash OperationCall\ (BooleanLiteral\ True)\ DotCall\ OrOp$   
 $[BooleanLiteral\ False] : Boolean[1]$   
**by simp**

$null\ and\ true : Boolean[?]$

**lemma**  
 $\Gamma_0 \vdash OperationCall\ (NullLiteral)\ DotCall\ AndOp$   
 $[BooleanLiteral\ True] : Boolean[?]$   
**by simp**

$let\ x : Real[1] = 5\ in\ x + 7 : Real[1]$

**lemma**  
 $\Gamma_0 \vdash Let\ (STR\ "x")\ (Some\ Real[1])\ (IntegerLiteral\ 5)$   
 $(OperationCall\ (Var\ STR\ "x")\ DotCall\ PlusOp\ [IntegerLiteral\ 7]) : Real[1]$   
**by simp**

```
    null.oclIsUndefined() : Boolean[1]
```

**lemma**

```
Γ0 ⊢ OperationCall (NullLiteral) DotCall OclIsUndefinedOp [] : Boolean[1]
by simp
```

```
    Sequence{1..5, null}.oclIsUndefined() : Sequence(Boolean[1])
```

**lemma**

```
Γ0 ⊢ OperationCall (CollectionLiteral SequenceKind
  [CollectionRange (IntegerLiteral 1) (IntegerLiteral 5),
   CollectionItem NullLiteral])
  DotCall OclIsUndefinedOp [] : Sequence Boolean[1]
by simp
```

```
    Sequence{1..5}->product(Set{'a', 'b'})
      : Set(Tuple(first: Integer[1], second: String[1]))
```

**lemma**

```
Γ0 ⊢ OperationCall (CollectionLiteral SequenceKind
  [CollectionRange (IntegerLiteral 1) (IntegerLiteral 5)])
  ArrowCall ProductOp
  [CollectionLiteral SetKind
   [CollectionItem (StringLiteral "a"),
    CollectionItem (StringLiteral "b")]) :
  Set (Tuple (fmap-of-list [
    (STR "first", Integer[1]), (STR "second", String[1])]))
by simp
```

```
    Sequence{1..5, null}?->iterate(x, acc : Real[1] = 0 | acc + x)
      : Real[1]
```

**lemma**

```
Γ0 ⊢ IterateCall (CollectionLiteral SequenceKind
  [CollectionRange (IntegerLiteral 1) (IntegerLiteral 5),
   CollectionItem NullLiteral]) SafeArrowCall
  [STR "x" None
   (STR "acc") (Some Real[1]) (IntegerLiteral 0)
   (OperationCall (Var STR "acc") DotCall PlusOp [Var STR "x"])] : Real[1]
by simp
```

```
    Sequence{1..5, null}?->max() : Integer[1]
```

**lemma**

```
Γ0 ⊢ OperationCall (CollectionLiteral SequenceKind
  [CollectionRange (IntegerLiteral 1) (IntegerLiteral 5),
   CollectionItem NullLiteral])
  SafeArrowCall CollectionMaxOp [] : Integer[1]
by simp
```

```
    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[?]
by simp

  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]
by simp

  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]
by simp

  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]
by simp

  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]
by simp

  Project[?].allInstances() : Set(Project[?])

lemma
 $\Gamma_0 \vdash \text{MetaOperationCall Project[?] AllInstancesOp} : \text{Set Project[?]}$ 

```

by *simp*

Project[1]::allProjects() : Set(Project[1])

lemma

$\Gamma_0 \vdash \text{StaticOperationCall Project[1] STR "allProjects" [] : Set Project[1]}$

by *simp*

## 8.6.2 Negative Cases

true = null

lemma

$\nexists \tau. \Gamma_0 \vdash \text{OperationCall (BooleanLiteral True) DotCall EqualOp [NullLiteral] : } \tau$

by *simp*

let x : Boolean[1] = 5 in x and true

lemma

$\nexists \tau. \Gamma_0 \vdash \text{Let STR "x" (Some Boolean[1]) (IntegerLiteral 5) (OperationCall (Var STR "x") DotCall AndOp [BooleanLiteral True]) : } \tau$

by *simp*

let x : Sequence(String[?]) = Sequence{'abc', 'zxc'} in  
x->closure(it | 1)

lemma

$\nexists \tau. \Gamma_0 \vdash \text{Let STR "x" (Some (Sequence String[?])) (CollectionLiteral SequenceKind [CollectionItem (StringLiteral "abc"), CollectionItem (StringLiteral "zxc")]) (ClosureIteratorCall (Var STR "x") ArrowCall [STR "it"] None (IntegerLiteral 1)) : } \tau$

by *simp*

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 –

have  $\neg \text{operation-defined (Integer[?] :: classes1 type) STR "max" [Integer[?]]}$

by *eval*

thus *?thesis* by *simp*

qed

## 8.7 Code

### 8.7.1 Positive Cases

values  $\{(D, \tau). \text{attribute Employee STR "name" } D \tau\}$

```

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
  { $\tau$ .  $\Gamma_0$ (STR "self"  $\mapsto_f$  Employee[1])  $\vdash$ 
  AssociationEndCall (AssociationEndCall (Var STR "self")
  DotCall None STR "projects")
  DotCall None STR "members" :  $\tau$ }

```

### 8.7.2 Negative Cases

```

values {(D,  $\tau$ ). attribute Employee STR "name2" D  $\tau$ }
value has-literal STR "E1" STR "C"

```

```

    Sequence{1..5, null}->max()

```

```

values
  { $\tau$ .  $\Gamma_0$   $\vdash$  OperationCall (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.