Safe OCL

Denis Nikiforov

March 17, 2025

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 distincts 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 ($\tau[1]$), 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

Chapter 1

Preliminaries

1.1 Errorable

theory *Errorable*
imports *Main*
begin

notation *bot* (‹⊥›)

typedef *'a errorable* (‹-⊥› [*21*] *21*) = *UNIV* :: *'a option set* ⟨*proof*⟩

definition *errorable* :: *'a* ⇒ *'a errorable* (‹-⊥› [*1000*] *1000*) **where**
errorable x = *Abs-errorable* (*Some x*)

instantiation *errorable* :: (*type*) *bot*
begin
definition ⊥ ≡ *Abs-errorable None*
instance ⟨*proof*⟩
end

free-constructors *case-errorable* **for**
errorable
| ⊥ :: *'a errorable*
⟨*proof*⟩

copy-bnf *'a errorable*

end

1.2 Transitive Closures

theory *Transitive-Closure-Ext*
imports *HOL−Library.FuncSet*
begin

1.2.1 Basic Properties

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

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

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

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

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

1.2.2 Helper Lemmas

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

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

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

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

1.2.3 Transitive Closure Preservation

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

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

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

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

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

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′*:
$(\bigwedge x\ y.\ R\ x\ y \Longrightarrow S\ (f\ x)\ (f\ y)) \Longrightarrow$
$(\bigwedge y.\ S\ (f\ y)\ (g\ y)) \Longrightarrow$
$R^{**}\ x\ y \Longrightarrow S^{**}\ (f\ x)\ (g\ y)$
⟨*proof*⟩

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

1.2.4 Transitive Closure Reflection

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

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

lemma *reflect-tranclp*:
assumes *refl-f*: $\bigwedge x\ y.\ S\ (f\ x)\ (f\ y) \Longrightarrow R\ x\ y$
assumes *bij-f*: *bij-on-trancl S f*
assumes *prem*: $S^{++}\ (f\ x)\ (f\ y)$
shows $R^{++}\ x\ y$
⟨*proof*⟩

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

lemma *reflect-rtranclp*:
$(\bigwedge x\ y.\ S\ (f\ x)\ (f\ y) \Longrightarrow R\ x\ y) \Longrightarrow$
bij-on-trancl S f $\Longrightarrow$
$S^{**}\ (f\ x)\ (f\ y) \Longrightarrow R^{**}\ x\ y$
⟨*proof*⟩

end

1.3 Finite Maps

theory *Finite-Map-Ext*
imports *HOL−Library.Finite-Map*
begin

type-notation *fmap* (‹(- $\rightharpoonup_f$ /-)› [*22*, *21*] *21*)

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

syntax
-fmaplet :: ['*a*, '*a*] ⇒ *fmaplet* (‹- /$\mapsto_f$/ -›)
-fmaplets :: ['*a*, '*a*] ⇒ *fmaplet* (‹- /[$\mapsto_f$]/ -›)
:: *fmaplet* ⇒ *fmaplets* (‹-›)
-FMaplets :: [*fmaplet*, *fmaplets*] ⇒ *fmaplets* (‹-,/ -›)
-FMapUpd :: ['*a* ⇀ '*b*, *fmaplets*] ⇒ '*a* ⇀ '*b* (‹-/'(-')› [*900*, *0*] *900*)
-FMap :: *fmaplets* ⇒ '*a* ⇀ '*b* (‹(*1*[-])›)

syntax (*ASCII*)
-fmaplet :: ['*a*, '*a*] ⇒ *fmaplet* (‹- /|->f/ -›)
-fmaplets :: ['*a*, '*a*] ⇒ *fmaplet* (‹- /[|->f]/ -›)

syntax-consts
-fmaplet *-fmaplets* *-FMaplets* *-FMapUpd* *-FMap* ⇌ *fmupd*

translations
-FMapUpd m (*-FMaplets xy ms*) ⇌ *-FMapUpd* (*-FMapUpd m xy*) *ms*
-FMapUpd m (*-fmaplet x y*) ⇌ *CONST fmupd x y m*
-FMap ms ⇌ *-FMapUpd* (*CONST fmempty*) *ms*
-FMap (*-FMaplets ms1 ms2*) ↽ *-FMapUpd* (*-FMap ms1*) *ms2*
-FMaplets ms1 (*-FMaplets ms2 ms3*) ↽ *-FMaplets* (*-FMaplets ms1 ms2*) *ms3*

1.3.1 Helper Lemmas

lemma *fmrel-on-fset-fmdom*:
fmrel-on-fset (*fmdom ym*) *f xm ym* ⟹
k |∈| *fmdom ym* ⟹
k |∈| *fmdom xm*
⟨*proof*⟩

1.3.2 Merge Operation

definition *fmmerge f xm ym* ≡
fmap-of-list (*map*
(λ*k*. (*k*, *f* (*the* (*fmlookup xm k*)) (*the* (*fmlookup ym k*))))
(*sorted-list-of-fset* (*fmdom xm* |∩| *fmdom ym*)))

lemma *fmdom-fmmerge* [*simp*]:
fmdom (*fmmerge g xm ym*) = *fmdom xm* |∩| *fmdom ym*
⟨*proof*⟩

lemma *fmmerge-commut*:
assumes ⋀*x y*. *x* ∈ *fmran'* *xm* ⟹ *f x y* = *f y x*
shows *fmmerge f xm ym* = *fmmerge f ym xm*
⟨*proof*⟩

lemma *fmrel-on-fset-fmmerge1* [*intro*]:
assumes $\bigwedge x\ y\ z.\ z \in fmran'\ zm \Longrightarrow f\ x\ z \Longrightarrow f\ y\ z \Longrightarrow f\ (g\ x\ y)\ z$
assumes *fmrel-on-fset* (*fmdom zm*) *f xm zm*
assumes *fmrel-on-fset* (*fmdom zm*) *f ym zm*
shows *fmrel-on-fset* (*fmdom zm*) *f* (*fmmerge g xm ym*) *zm*
⟨*proof*⟩

lemma *fmrel-on-fset-fmmerge2* [*intro*]:
assumes $\bigwedge x\ y.\ x \in fmran'\ xm \Longrightarrow f\ x\ (g\ x\ y)$
shows *fmrel-on-fset* (*fmdom ym*) *f xm* (*fmmerge g xm ym*)
⟨*proof*⟩

1.3.3 Acyclicity

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

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

lemma *fmrel-acyclic*:
acyclicP-on (*fmran' xm*) *R* ⟹
fmrel R^{++} *xm ym* ⟹
fmrel R ym xm ⟹
xm = *ym*
⟨*proof*⟩

lemma *fmrel-acyclic'*:
assumes *acyclicP-on* (*fmran' ym*) *R*
assumes *fmrel* R^{++} *xm ym*
assumes *fmrel R ym xm*
shows *xm* = *ym*
⟨*proof*⟩

lemma *fmrel-on-fset-acyclic*:
acyclicP-on (*fmran' xm*) *R* ⟹
fmrel-on-fset (*fmdom ym*) R^{++} *xm ym* ⟹
fmrel-on-fset (*fmdom xm*) *R ym xm* ⟹
xm = *ym*
⟨*proof*⟩

lemma *fmrel-on-fset-acyclic'*:
acyclicP-on (*fmran' ym*) *R* ⟹
fmrel-on-fset (*fmdom ym*) R^{++} *xm ym* ⟹
fmrel-on-fset (*fmdom xm*) *R ym xm* ⟹
xm = *ym*
⟨*proof*⟩

1.3.4 Transitive Closures

lemma *fmrel-trans*:

$(\bigwedge x\ y\ z.\ x \in fmran'\ xm \Longrightarrow P\ x\ y \Longrightarrow Q\ y\ z \Longrightarrow R\ x\ z) \Longrightarrow$
$fmrel\ P\ xm\ ym \Longrightarrow fmrel\ Q\ ym\ zm \Longrightarrow fmrel\ R\ xm\ zm$
⟨*proof*⟩

lemma *fmrel-on-fset-trans*:
$(\bigwedge x\ y\ z.\ x \in fmran'\ xm \Longrightarrow P\ x\ y \Longrightarrow Q\ y\ z \Longrightarrow R\ x\ z) \Longrightarrow$
$fmrel\text{-}on\text{-}fset\ (fmdom\ ym)\ P\ xm\ ym \Longrightarrow$
$fmrel\text{-}on\text{-}fset\ (fmdom\ zm)\ Q\ ym\ zm \Longrightarrow$
$fmrel\text{-}on\text{-}fset\ (fmdom\ zm)\ R\ xm\ zm$
⟨*proof*⟩

lemma *trancl-to-fmrel*:
$(fmrel\ f)^{++}\ xm\ ym \Longrightarrow fmrel\ f^{++}\ xm\ ym$
⟨*proof*⟩

lemma *fmrel-trancl-fmdom-eq*:
$(fmrel\ f)^{++}\ xm\ ym \Longrightarrow fmdom\ xm = fmdom\ ym$
⟨*proof*⟩

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

lemma *fmupd-fmdrop*:
$fmlookup\ xm\ k = Some\ x \Longrightarrow$
$xm = fmupd\ k\ x\ (fmdrop\ k\ xm)$
⟨*proof*⟩

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*⟩

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*: $P\ (fmap\text{-}of\text{-}list\ [])\ (fmap\text{-}of\text{-}list\ [])$
and *step*:
$\bigwedge x\ xm\ y\ ym\ i.$
$[\![R\ x\ y;\ fmrel\ R\ xm\ ym;\ fmdom\ xm = fmdom\ ym;\ P\ xm\ ym]\!] \Longrightarrow$
$P\ (fmupd\ i\ x\ xm)\ (fmupd\ i\ y\ ym)$
shows $P\ xm\ ym$
⟨*proof*⟩

The proof was derived from the accepted answer on the website Stack

Overflow that is available at https://stackoverflow.com/a/53585232/632199 and provided with the permission of the author of the answer.

lemma *fmrel-to-rtrancl*:
assumes *as-r*: *reflp r*
and *rel-rpp-xm-ym*: *fmrel* r^{**} *xm ym*
shows (*fmrel r*)** *xm ym*
⟨*proof*⟩

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

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

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

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

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

1.3.5 Size Calculation

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

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

interpretation *tcf*: *comp-fun-commute tcf*

⟨*proof*⟩

lemma *ffold-rec-exp*:
assumes k |∈| *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*⟩

lemma *elem-le-ffold* [*intro*]:
k |∈| *fmdom* $x \Longrightarrow$
f (*the* (*fmlookup* x k)) < *Suc* (*ffold tcf 0* (*fset-of-fmap* (*fmmap* f x)))
⟨*proof*⟩

lemma *elem-le-ffold′* [*intro*]:
$z \in$ *fmran′* $x \Longrightarrow$
f z < *Suc* (*ffold tcf 0* (*fset-of-fmap* (*fmmap* f x)))
⟨*proof*⟩

1.3.6 Code Setup

abbreviation *fmmerge-fun* f xm $ym \equiv$
fmap-of-list (*map*
(λk. *if* k |∈| *fmdom* $xm \wedge$ k |∈| *fmdom* ym
then (k, f (*the* (*fmlookup* xm k)) (*the* (*fmlookup* ym k)))
else (k, *undefined*))
(*sorted-list-of-fset* (*fmdom* xm |∩| *fmdom* ym)))

lemma *fmmerge-fun-simp* [*code-abbrev*, *simp*]:
fmmerge-fun f xm ym = *fmmerge* f xm ym
⟨*proof*⟩

end

1.4 Tuples

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

1.4.1 Definitions

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

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

1.4.2 Helper Lemmas

lemma *fmrel-to-subtuple*:

fmrel r xm ym $\Longrightarrow$ *subtuple r xm ym*
⟨*proof*⟩

lemma *subtuple-eq-fmrel-fmrestrict-fset*:
subtuple r xm ym = *fmrel r* (*fmrestrict-fset* (*fmdom ym*) *xm*) *ym*
⟨*proof*⟩

lemma *subtuple-fmdom*:
subtuple f xm ym $\Longrightarrow$
subtuple g ym xm $\Longrightarrow$
fmdom xm = *fmdom ym*
⟨*proof*⟩

1.4.3 Basic Properties

lemma *subtuple-refl*:
reflp R $\Longrightarrow$ *subtuple R xm xm*
⟨*proof*⟩

lemma *subtuple-mono* [*mono*]:
($\bigwedge$*x y. x* ∈ *fmran' xm* $\Longrightarrow$ *y* ∈ *fmran' ym* $\Longrightarrow$ *f x y* $\longrightarrow$ *g x y*) $\Longrightarrow$
subtuple f xm ym $\longrightarrow$ *subtuple g xm ym*
⟨*proof*⟩

lemma *strict-subtuple-mono* [*mono*]:
($\bigwedge$*x y. x* ∈ *fmran' xm* $\Longrightarrow$ *y* ∈ *fmran' ym* $\Longrightarrow$ *f x y* $\longrightarrow$ *g x y*) $\Longrightarrow$
strict-subtuple f xm ym $\longrightarrow$ *strict-subtuple g xm ym*
⟨*proof*⟩

lemma *subtuple-antisym*:
assumes *subtuple* (λ*x y. f x y* ∨ *x* = *y*) *xm ym*
assumes *subtuple* (λ*x y. f x y* ∧ ¬ *f y x* ∨ *x* = *y*) *ym xm*
shows *xm* = *ym*
⟨*proof*⟩

lemma *strict-subtuple-antisym*:
strict-subtuple (λ*x y. f x y* ∨ *x* = *y*) *xm ym* $\Longrightarrow$
strict-subtuple (λ*x y. f x y* ∧ ¬ *f y x* ∨ *x* = *y*) *ym xm* $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *subtuple-acyclic*:
assumes *acyclicP-on* (*fmran' xm*) *P*
assumes *subtuple* (λ*x y. P x y* ∨ *x* = *y*)$^{++}$ *xm ym*
assumes *subtuple* (λ*x y. P x y* ∨ *x* = *y*) *ym xm*
shows *xm* = *ym*
⟨*proof*⟩

lemma *subtuple-acyclic'*:
assumes *acyclicP-on* (*fmran' ym*) *P*

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

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

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

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

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

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

lemma *subtuple-fmmerge2* [*intro*]:
$(\bigwedge x\ y.\ x \in fmran'\ xm \Longrightarrow f\ x\ (g\ x\ y)) \Longrightarrow$
$subtuple\ f\ xm\ (fmmerge\ g\ xm\ ym)$
⟨*proof*⟩

1.4.4 Transitive Closures

lemma *trancl-to-subtuple*:
$(subtuple\ r)^{++}\ xm\ ym \Longrightarrow$

subtuple r^{++} *xm ym*
⟨*proof*⟩

lemma *rtrancl-to-subtuple*:
*(subtuple r)** xm ym* ⟹
*subtuple r** xm ym*
⟨*proof*⟩

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

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

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

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

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

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

1.4.5 Code Setup

abbreviation *subtuple-fun f xm ym* ≡
fBall (*fmdom ym*) (λ*x*. *rel-option f* (*fmlookup xm x*) (*fmlookup ym x*))

abbreviation *strict-subtuple-fun f xm ym* ≡
subtuple-fun f xm ym ∧ *xm* ≠ *ym*

lemma *subtuple-fun-simp* [*code-abbrev*, *simp*]:
subtuple-fun f xm ym = *subtuple f xm ym*
⟨*proof*⟩

lemma *strict-subtuple-fun-simp* [*code-abbrev*, *simp*]:
strict-subtuple-fun f xm ym = *strict-subtuple f xm ym*
⟨*proof*⟩

end

1.5 Object Model

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

The section defines a generic object model.

1.5.1 Type Synonyms

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

datatype *param-dir* = *In* | *Out* | *InOut*

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

definition *assoc-end-class* :: *'c assoc-end* ⇒ *'c* ≡ *fst*
definition *assoc-end-min* :: *'c assoc-end* ⇒ *nat* ≡ *fst* ∘ *snd*
definition *assoc-end-max* :: *'c assoc-end* ⇒ *enat* ≡ *fst* ∘ *snd* ∘ *snd*
definition *assoc-end-ordered* :: *'c assoc-end* ⇒ *bool* ≡ *fst* ∘ *snd* ∘ *snd* ∘ *snd*
definition *assoc-end-unique* :: *'c assoc-end* ⇒ *bool* ≡ *snd* ∘ *snd* ∘ *snd* ∘ *snd*

definition *oper-name* :: (*'t*, *'e*) *oper-spec* ⇒ *oper* ≡ *fst*

definition *oper-context* :: ('t, 'e) *oper-spec* ⇒ 't ≡ *fst* ∘ *snd*
definition *oper-params* :: ('t, 'e) *oper-spec* ⇒ 't *param-spec list* ≡ *fst* ∘ *snd* ∘ *snd*

definition *oper-result* :: ('t, 'e) *oper-spec* ⇒ 't ≡ *fst* ∘ *snd* ∘ *snd* ∘ *snd*
definition *oper-static* :: ('t, 'e) *oper-spec* ⇒ *bool* ≡ *fst* ∘ *snd* ∘ *snd* ∘ *snd* ∘ *snd*
definition *oper-body* :: ('t, 'e) *oper-spec* ⇒ 'e *option* ≡ *snd* ∘ *snd* ∘ *snd* ∘ *snd* ∘ *snd*

definition *param-name* ::'t *param-spec* ⇒ *param* ≡ *fst*
definition *param-type* ::'t *param-spec* ⇒ 't ≡ *fst* ∘ *snd*
definition *param-dir* ::'t *param-spec* ⇒ *param-dir* ≡ *snd* ∘ *snd*

definition *oper-in-params op* ≡
filter (λp. *param-dir p* = *In* ∨ *param-dir p* = *InOut*) (*oper-params op*)

definition *oper-out-params op* ≡
filter (λp. *param-dir p* = *Out* ∨ *param-dir p* = *InOut*) (*oper-params op*)

1.5.2 Attributes

inductive *owned-attribute'* **where**
$\mathcal{C}$ |∈| *fmdom attributes* ⟹
fmlookup attributes $\mathcal{C}$ = *Some* $attrs_{\mathcal{C}}$ ⟹
fmlookup $attrs_{\mathcal{C}}$ *attr* = *Some* τ ⟹
owned-attribute' attributes $\mathcal{C}$ *attr* τ

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

inductive *closest-attribute* **where**
owned-attribute' attributes $\mathcal{D}$ *attr* τ ⟹
$\mathcal{C} \leq \mathcal{D}$ ⟹
¬ *attribute-not-closest attributes* $\mathcal{C}$ *attr* $\mathcal{D}$ τ ⟹
closest-attribute attributes $\mathcal{C}$ *attr* $\mathcal{D}$ τ

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

inductive *unique-closest-attribute* **where**
closest-attribute attributes $\mathcal{C}$ *attr* $\mathcal{D}$ τ ⟹
¬ *closest-attribute-not-unique attributes* $\mathcal{C}$ *attr* $\mathcal{D}$ τ ⟹
unique-closest-attribute attributes $\mathcal{C}$ *attr* $\mathcal{D}$ τ

1.5.3 Association Ends

inductive *role-refer-class* **where**
role $|\in|$ *fmdom ends* $\Longrightarrow$
fmlookup ends role $=$ *Some end* $\Longrightarrow$
assoc-end-class end $= \mathcal{C} \Longrightarrow$
role-refer-class ends $\mathcal{C}$ *role*

inductive *association-ends'* **where**
$\mathcal{C}$ $|\in|$ *classes* $\Longrightarrow$
assoc $|\in|$ *fmdom associations* $\Longrightarrow$
fmlookup associations assoc $=$ *Some ends* $\Longrightarrow$
role-refer-class ends $\mathcal{C}$ *from* $\Longrightarrow$
role $|\in|$ *fmdom ends* $\Longrightarrow$
fmlookup ends role $=$ *Some end* $\Longrightarrow$
role $\neq$ *from* $\Longrightarrow$
association-ends' classes associations $\mathcal{C}$ *from role end*

inductive *association-ends-not-unique'* **where**
association-ends' classes associations $\mathcal{C}$ *from role* $end_1 \Longrightarrow$
association-ends' classes associations $\mathcal{C}$ *from role* $end_2 \Longrightarrow$
$end_1 \neq end_2 \Longrightarrow$
association-ends-not-unique' classes associations

inductive *owned-association-end'* **where**
association-ends' classes associations $\mathcal{C}$ *from role end* $\Longrightarrow$
owned-association-end' classes associations $\mathcal{C}$ *None role end*
| *association-ends' classes associations* $\mathcal{C}$ *from role end* $\Longrightarrow$
owned-association-end' classes associations $\mathcal{C}$ (*Some from*) *role end*

inductive *association-end-not-closest* **where**
owned-association-end' classes associations $\mathcal{D}'$ *from role end'* $\Longrightarrow$
$\mathcal{C} \leq \mathcal{D}' \Longrightarrow$
$\mathcal{D}' < \mathcal{D} \Longrightarrow$
association-end-not-closest classes associations $\mathcal{C}$ *from role* $\mathcal{D}$ *end*

inductive *closest-association-end* **where**
owned-association-end' classes associations $\mathcal{D}$ *from role end* $\Longrightarrow$
$\mathcal{C} \leq \mathcal{D} \Longrightarrow$
$\neg$ *association-end-not-closest classes associations* $\mathcal{C}$ *from role* $\mathcal{D}$ *end* $\Longrightarrow$
closest-association-end classes associations $\mathcal{C}$ *from role* $\mathcal{D}$ *end*

inductive *closest-association-end-not-unique* **where**
closest-association-end classes associations $\mathcal{C}$ *from role* $\mathcal{D}'$ *end'* $\Longrightarrow$
$\mathcal{D} \neq \mathcal{D}' \vee$ *end* $\neq$ *end'* $\Longrightarrow$
closest-association-end-not-unique classes associations $\mathcal{C}$ *from role* $\mathcal{D}$ *end*

inductive *unique-closest-association-end* **where**
closest-association-end classes associations $\mathcal{C}$ *from role* $\mathcal{D}$ *end* $\Longrightarrow$
$\neg$ *closest-association-end-not-unique classes associations* $\mathcal{C}$ *from role* $\mathcal{D}$ *end* $\Longrightarrow$

unique-closest-association-end classes associations $\mathcal{C}$ *from role* $\mathcal{D}$ *end*

1.5.4 Association Classes

inductive *referred-by-association-class''* **where**
fmlookup association-classes $\mathcal{A}$ = *Some assoc* $\Longrightarrow$
fmlookup associations assoc = *Some ends* $\Longrightarrow$
role-refer-class ends $\mathcal{C}$ *from* $\Longrightarrow$
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* $\mathcal{A}$ $\Longrightarrow$
referred-by-association-class' *association-classes associations* $\mathcal{C}$ *None* $\mathcal{A}$
| *referred-by-association-class''* *association-classes associations* $\mathcal{C}$ *from* $\mathcal{A}$ $\Longrightarrow$
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* $\mathcal{A}$ $\Longrightarrow$
$\mathcal{C} \leq \mathcal{D}'$ $\Longrightarrow$
$\mathcal{D}' < \mathcal{D}$ $\Longrightarrow$
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* $\mathcal{A}$ $\Longrightarrow$
$\mathcal{C} \leq \mathcal{D}$ $\Longrightarrow$
$\neg$ *association-class-not-closest association-classes associations* $\mathcal{C}$ *from* $\mathcal{A}$ $\mathcal{D}$ $\Longrightarrow$
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* $\mathcal{A}$ $\mathcal{D}'$ $\Longrightarrow$
$\mathcal{D} \neq \mathcal{D}'$ $\Longrightarrow$
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* $\mathcal{A}$ $\mathcal{D}$ $\Longrightarrow$
$\neg$ *closest-association-class-not-unique*
association-classes associations $\mathcal{C}$ *from* $\mathcal{A}$ $\mathcal{D}$ $\Longrightarrow$
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* $\Longrightarrow$
fmlookup associations assoc = *Some ends* $\Longrightarrow$
fmlookup ends role = *Some end* $\Longrightarrow$
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'* $\Longrightarrow$

end $\neq$ *end'* $\Longrightarrow$
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* $\Longrightarrow$
$\neg$ *association-class-end-not-unique*
association-classes associations $\mathcal{A}$ *role end* $\Longrightarrow$
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* $\Longrightarrow$
oper-name op = *name* $\Longrightarrow$
$\tau \leq$ *oper-context op* $\Longrightarrow$
list-all2 $(\lambda\sigma$ *param.* $\sigma \leq$ *param-type param*$)$ π $($*oper-in-params op*$)$ $\Longrightarrow$
any-operation' operations τ *name* π *op*

inductive *operation'* **where**
any-operation' operations τ *name* π *op* $\Longrightarrow$
$\neg$ *oper-static op* $\Longrightarrow$
operation' operations τ *name* π *op*

inductive *operation-not-unique* **where**
operation' operations τ *name* π *oper'* $\Longrightarrow$
oper $\neq$ *oper'* $\Longrightarrow$
operation-not-unique operations τ *name* π *oper*

inductive *unique-operation* **where**
operation' operations τ *name* π *oper* $\Longrightarrow$
$\neg$ *operation-not-unique operations* τ *name* π *oper* $\Longrightarrow$
unique-operation operations τ *name* π *oper*

inductive *operation-defined'* **where**
unique-operation operations τ *name* π *oper* $\Longrightarrow$
operation-defined' operations τ *name* π

inductive *static-operation'* **where**
any-operation' operations τ *name* π *op* $\Longrightarrow$
oper-static op $\Longrightarrow$
static-operation' operations τ *name* π *op*

inductive *static-operation-not-unique* **where**
static-operation' operations τ *name* π *oper'* $\Longrightarrow$
oper $\neq$ *oper'* $\Longrightarrow$
static-operation-not-unique operations τ *name* π *oper*

inductive *unique-static-operation* **where**
static-operation' operations τ *name* π *oper* $\Longrightarrow$

¬ *static-operation-not-unique operations* τ *name* π *oper* $\Longrightarrow$
unique-static-operation operations τ *name* π *oper*

inductive *static-operation-defined′* **where**
unique-static-operation operations τ *name* π *oper* $\Longrightarrow$
static-operation-defined′ operations τ *name* π

1.5.7 Literals

inductive *has-literal′* **where**
fmlookup literals e = *Some lits* $\Longrightarrow$
lit $|\in|$ *lits* $\Longrightarrow$
has-literal′ literals e lit

1.5.8 Definition

locale *object-model* =
fixes *classes* :: $'a$ :: *semilattice-sup fset*
and *attributes* :: $'a \rightharpoonup_f$ *attr* $\rightharpoonup_f$ $'t$ :: *order*
and *associations* :: *assoc* $\rightharpoonup_f$ *role* $\rightharpoonup_f$ $'a$ *assoc-end*
and *association-classes* :: $'a \rightharpoonup_f$ *assoc*
and *operations* :: ($'t$, $'e$) *oper-spec list*
and *literals* :: $'n \rightharpoonup_f$ *elit fset*
assumes *assoc-end-min-less-eq-max*:
assoc $|\in|$ *fmdom associations* $\Longrightarrow$
fmlookup associations assoc = *Some ends* $\Longrightarrow$
role $|\in|$ *fmdom ends* $\Longrightarrow$
fmlookup ends role = *Some end* $\Longrightarrow$
assoc-end-min end $\leq$ *assoc-end-max end*
assumes *association-ends-unique*:
association-ends′ classes associations $\mathcal{C}$ *from role* end_1 $\Longrightarrow$
association-ends′ classes associations $\mathcal{C}$ *from role* end_2 $\Longrightarrow$ $end_1 = end_2$
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* ≡
unique-association-class-end association-classes associations

abbreviation *operation* ≡
unique-operation operations

abbreviation *operation-defined* ≡
operation-defined′ operations

abbreviation *static-operation* ≡
unique-static-operation operations

abbreviation *static-operation-defined* ≡
static-operation-defined′ operations

abbreviation *has-literal* ≡
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* $\mathcal{D}_1$ τ_1 ⟹
attribute $\mathcal{C}$ *attr* $\mathcal{D}_2$ τ_2 ⟹ $\mathcal{D}_1 = \mathcal{D}_2 \wedge \tau_1 = \tau_2$
⟨*proof*⟩

lemma (**in** *object-model*) *attribute-self-or-inherited*:
attribute $\mathcal{C}$ *attr* $\mathcal{D}$ τ ⟹ $\mathcal{C} \leq \mathcal{D}$
⟨*proof*⟩

lemma (**in** *object-model*) *attribute-closest*:
attribute $\mathcal{C}$ *attr* $\mathcal{D}$ τ ⟹
owned-attribute $\mathcal{D}'$ *attr* τ ⟹
$\mathcal{C} \leq \mathcal{D}'$ ⟹ ¬ $\mathcal{D}' < \mathcal{D}$
⟨*proof*⟩

lemma (**in** *object-model*) *association-end-det*:
association-end $\mathcal{C}$ *from role* $\mathcal{D}_1$ end_1 ⟹
association-end $\mathcal{C}$ *from role* $\mathcal{D}_2$ end_2 ⟹ $\mathcal{D}_1 = \mathcal{D}_2 \wedge end_1 = end_2$
⟨*proof*⟩

lemma (**in** *object-model*) *association-end-self-or-inherited*:
association-end $\mathcal{C}$ *from role* $\mathcal{D}$ *end* $\Longrightarrow \mathcal{C} \leq \mathcal{D}$
⟨*proof*⟩

lemma (**in** *object-model*) *association-end-closest*:
association-end $\mathcal{C}$ *from role* $\mathcal{D}$ *end* $\Longrightarrow$
owned-association-end $\mathcal{D}'$ *from role end* $\Longrightarrow$
$\mathcal{C} \leq \mathcal{D}' \Longrightarrow \neg\ \mathcal{D}' < \mathcal{D}$
⟨*proof*⟩

lemma (**in** *object-model*) *association-class-end-det*:
association-class-end $\mathcal{A}$ *role* $end_1 \Longrightarrow$
association-class-end $\mathcal{A}$ *role* $end_2 \Longrightarrow end_1 = end_2$
⟨*proof*⟩

lemma (**in** *object-model*) *operation-det*:
operation τ *name* π $oper_1 \Longrightarrow$
operation τ *name* π $oper_2 \Longrightarrow oper_1 = oper_2$
⟨*proof*⟩

lemma (**in** *object-model*) *static-operation-det*:
static-operation τ *name* π $oper_1 \Longrightarrow$
static-operation τ *name* π $oper_2 \Longrightarrow oper_1 = oper_2$
⟨*proof*⟩

1.5.10 Code Setup

declare *owned-attribute′.intros*[*folded Predicate-Compile.contains-def*, *code-pred-intro*]
code-pred (*modes*:
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow bool$) *owned-attribute′*
⟨*proof*⟩

code-pred *unique-closest-attribute* ⟨*proof*⟩

declare *role-refer-class.intros*[*folded Predicate-Compile.contains-def*, *code-pred-intro*]
code-pred (*modes*:
$i \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow o \Rightarrow o \Rightarrow bool$) *role-refer-class*
⟨*proof*⟩

declare *association-ends′.intros*[*folded Predicate-Compile.contains-def*, *code-pred-intro*]

code-pred (*modes*:
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$) *association-ends'*
⟨*proof*⟩

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

code-pred (*modes*:
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$) *owned-association-end'* ⟨*proof*⟩

code-pred (*modes*:
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$,

$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$) *closest-association-end* ⟨*proof*⟩

code-pred (*modes*:
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow bool$) *closest-association-end-not-unique*
⟨*proof*⟩

code-pred (*modes*:
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool$) *unique-closest-association-end* ⟨*proof*⟩

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

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

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

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

declare *any-operation′.intros*[*folded Predicate-Compile.contains-def*, *code-pred-intro*]
code-pred [*show-modes*] *any-operation′*
⟨*proof*⟩

code-pred (*modes*:
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$) *operation′* ⟨*proof*⟩

code-pred (*modes*:
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$) *operation-not-unique* ⟨*proof*⟩

code-pred (*modes*:
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$) *unique-operation* ⟨*proof*⟩

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

code-pred (*modes*:
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$) *static-operation′* ⟨*proof*⟩

code-pred (*modes*:
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$) *static-operation-not-unique* ⟨*proof*⟩

code-pred (*modes*:
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$) *unique-static-operation* ⟨*proof*⟩

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

declare *has-literal'.intros*[*folded Predicate-Compile.contains-def*, *code-pred-intro*]
code-pred (*modes*:
$i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow bool$) *has-literal'*
⟨*proof*⟩

end

Chapter 2

Basic Types

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

2.1 Definition

Basic types are parameterized over classes.

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

datatype (*'a* :: *order*) *basic-type* =
  OclAny
| *OclVoid*
| *Boolean*
| *Real*
| *Integer*
| *UnlimitedNatural*
| *String*
| *ObjectType 'a* (‹⟨-⟩$_\mathcal{T}$› [*0*] *1000*)
| *Enum 'a enum*

inductive *basic-subtype* (**infix** ‹$\sqsubset_B$› *65*) **where**
  OclVoid $\sqsubset_B$ *Boolean*
| *OclVoid* $\sqsubset_B$ *UnlimitedNatural*
| *OclVoid* $\sqsubset_B$ *String*
| *OclVoid* $\sqsubset_B$ $\langle\mathcal{C}\rangle_\mathcal{T}$
| *OclVoid* $\sqsubset_B$ *Enum* $\mathcal{E}$

| *UnlimitedNatural* $\sqsubset_B$ *Integer*
| *Integer* $\sqsubset_B$ *Real*
| $\mathcal{C} < \mathcal{D} \Longrightarrow \langle\mathcal{C}\rangle_\mathcal{T} \sqsubset_B \langle\mathcal{D}\rangle_\mathcal{T}$

| *Boolean* $\sqsubset_B$ *OclAny*
| *Real* $\sqsubset_B$ *OclAny*

| *String* $\sqsubset_B$ *OclAny*
| $\langle \mathcal{C} \rangle_\mathcal{T} \sqsubset_B$ *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*!]: $\tau \sqsubset_B \langle \mathcal{C} \rangle_\mathcal{T}$
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*!]: $\langle \mathcal{C} \rangle_\mathcal{T} \sqsubset_B \sigma$

lemma *basic-subtype-asym*:
$\tau \sqsubset_B \sigma \Longrightarrow \sigma \sqsubset_B \tau \Longrightarrow$ *False*
⟨*proof*⟩

2.2 Partial Order of Basic Types

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

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

2.2.1 Strict Introduction Rules

lemma *type-less-x-OclAny-intro* [*intro*]:
$\tau \neq$ *OclAny* $\Longrightarrow \tau <$ *OclAny*
⟨*proof*⟩

lemma *type-less-OclVoid-x-intro* [*intro*]:
$\tau \neq$ *OclVoid* $\Longrightarrow$ *OclVoid* $< \tau$
⟨*proof*⟩

lemma *type-less-x-Real-intro* [*intro*]:
$\tau =$ *UnlimitedNatural* $\Longrightarrow \tau <$ *Real*
$\tau =$ *Integer* $\Longrightarrow \tau <$ *Real*
⟨*proof*⟩

lemma *type-less-x-Integer-intro* [*intro*]:
$\tau =$ *UnlimitedNatural* $\Longrightarrow \tau <$ *Integer*

⟨*proof*⟩

lemma *type-less-x-ObjectType-intro* [*intro*]:
$\tau = \langle \mathcal{C} \rangle_{\mathcal{T}} \Longrightarrow \mathcal{C} < \mathcal{D} \Longrightarrow \tau < \langle \mathcal{D} \rangle_{\mathcal{T}}$
⟨*proof*⟩

2.2.2 Strict Elimination Rules

lemma *type-less-x-OclAny* [*elim*!]:
$\tau < \mathit{OclAny} \Longrightarrow$
$(\tau = \mathit{OclVoid} \Longrightarrow P) \Longrightarrow$
$(\tau = \mathit{Boolean} \Longrightarrow P) \Longrightarrow$
$(\tau = \mathit{Integer} \Longrightarrow P) \Longrightarrow$
$(\tau = \mathit{UnlimitedNatural} \Longrightarrow P) \Longrightarrow$
$(\tau = \mathit{Real} \Longrightarrow P) \Longrightarrow$
$(\tau = \mathit{String} \Longrightarrow P) \Longrightarrow$
$(\bigwedge \mathcal{E}.\ \tau = \mathit{Enum}\ \mathcal{E} \Longrightarrow P) \Longrightarrow$
$(\bigwedge \mathcal{C}.\ \tau = \langle \mathcal{C} \rangle_{\mathcal{T}} \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-x-OclVoid* [*elim*!]:
$\tau < \mathit{OclVoid} \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-x-Boolean* [*elim*!]:
$\tau < \mathit{Boolean} \Longrightarrow$
$(\tau = \mathit{OclVoid} \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-x-Real* [*elim*!]:
$\tau < \mathit{Real} \Longrightarrow$
$(\tau = \mathit{OclVoid} \Longrightarrow P) \Longrightarrow$
$(\tau = \mathit{UnlimitedNatural} \Longrightarrow P) \Longrightarrow$
$(\tau = \mathit{Integer} \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-x-Integer* [*elim*!]:
$\tau < \mathit{Integer} \Longrightarrow$
$(\tau = \mathit{OclVoid} \Longrightarrow P) \Longrightarrow$
$(\tau = \mathit{UnlimitedNatural} \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-x-UnlimitedNatural* [*elim*!]:
$\tau < \mathit{UnlimitedNatural} \Longrightarrow$
$(\tau = \mathit{OclVoid} \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-x-String* [*elim*!]:
$\tau < \mathit{String} \Longrightarrow$

$(\tau = OclVoid \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-x-ObjectType* [*elim*!]:
$\tau < \langle\mathcal{D}\rangle_{\mathcal{T}} \Longrightarrow$
$(\tau = OclVoid \Longrightarrow P) \Longrightarrow$
$(\bigwedge\mathcal{C}.\ \tau = \langle\mathcal{C}\rangle_{\mathcal{T}} \Longrightarrow \mathcal{C} < \mathcal{D} \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-x-Enum* [*elim*!]:
$\tau < Enum\ \mathcal{E} \Longrightarrow$
$(\tau = OclVoid \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

2.2.3 Properties

lemma *basic-subtype-irrefl*:
$\tau < \tau \Longrightarrow False$
for τ :: *'a basic-type*
⟨*proof*⟩

lemma *tranclp-less-basic-type*:
$(\tau, \sigma) \in \{(\tau, \sigma).\ \tau \sqsubset_B \sigma\}^+ \longleftrightarrow \tau < \sigma$
⟨*proof*⟩

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

lemma *less-le-not-le-basic-type*:
$\tau < \sigma \longleftrightarrow \tau \leq \sigma \wedge \neg\ \sigma \leq \tau$
for τ σ :: *'a basic-type*
⟨*proof*⟩

lemma *antisym-basic-type*:
$\tau \leq \sigma \Longrightarrow \sigma \leq \tau \Longrightarrow \tau = \sigma$
for τ σ :: *'a basic-type*
⟨*proof*⟩

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

instance
⟨*proof*⟩

end

2.2.4 Non-Strict Introduction Rules

lemma *type-less-eq-x-OclAny-intro* [*intro*]:
$\tau \leq OclAny$
⟨*proof*⟩

lemma *type-less-eq-OclVoid-x-intro* [*intro*]:
$OclVoid \leq \tau$
⟨*proof*⟩

lemma *type-less-eq-x-Real-intro* [*intro*]:
$\tau = UnlimitedNatural \Longrightarrow \tau \leq Real$
$\tau = Integer \Longrightarrow \tau \leq Real$
⟨*proof*⟩

lemma *type-less-eq-x-Integer-intro* [*intro*]:
$\tau = UnlimitedNatural \Longrightarrow \tau \leq Integer$
⟨*proof*⟩

lemma *type-less-eq-x-ObjectType-intro* [*intro*]:
$\tau = \langle \mathcal{C} \rangle_{\mathcal{T}} \Longrightarrow \mathcal{C} \leq \mathcal{D} \Longrightarrow \tau \leq \langle \mathcal{D} \rangle_{\mathcal{T}}$
⟨*proof*⟩

2.2.5 Non-Strict Elimination Rules

lemma *type-less-eq-x-OclAny* [*elim*!]:
$\tau \leq OclAny \Longrightarrow$
$(\tau = OclVoid \Longrightarrow P) \Longrightarrow$
$(\tau = OclAny \Longrightarrow P) \Longrightarrow$
$(\tau = Boolean \Longrightarrow P) \Longrightarrow$
$(\tau = Integer \Longrightarrow P) \Longrightarrow$
$(\tau = UnlimitedNatural \Longrightarrow P) \Longrightarrow$
$(\tau = Real \Longrightarrow P) \Longrightarrow$
$(\tau = String \Longrightarrow P) \Longrightarrow$
$(\bigwedge \mathcal{E}.\ \tau = Enum\ \mathcal{E} \Longrightarrow P) \Longrightarrow$
$(\bigwedge \mathcal{C}.\ \tau = \langle \mathcal{C} \rangle_{\mathcal{T}} \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-eq-x-OclVoid* [*elim*!]:
$\tau \leq OclVoid \Longrightarrow$
$(\tau = OclVoid \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-eq-x-Boolean* [*elim*!]:
$\tau \leq Boolean \Longrightarrow$
$(\tau = OclVoid \Longrightarrow P) \Longrightarrow$
$(\tau = Boolean \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

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

$\tau \leq \mathit{Real} \Longrightarrow$
$(\tau = \mathit{OclVoid} \Longrightarrow P) \Longrightarrow$
$(\tau = \mathit{UnlimitedNatural} \Longrightarrow P) \Longrightarrow$
$(\tau = \mathit{Integer} \Longrightarrow P) \Longrightarrow$
$(\tau = \mathit{Real} \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-eq-x-Integer* [*elim*!]:
$\tau \leq \mathit{Integer} \Longrightarrow$
$(\tau = \mathit{OclVoid} \Longrightarrow P) \Longrightarrow$
$(\tau = \mathit{UnlimitedNatural} \Longrightarrow P) \Longrightarrow$
$(\tau = \mathit{Integer} \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-eq-x-UnlimitedNatural* [*elim*!]:
$\tau \leq \mathit{UnlimitedNatural} \Longrightarrow$
$(\tau = \mathit{OclVoid} \Longrightarrow P) \Longrightarrow$
$(\tau = \mathit{UnlimitedNatural} \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-eq-x-String* [*elim*!]:
$\tau \leq \mathit{String} \Longrightarrow$
$(\tau = \mathit{OclVoid} \Longrightarrow P) \Longrightarrow$
$(\tau = \mathit{String} \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-eq-x-ObjectType* [*elim*!]:
$\tau \leq \langle \mathcal{D} \rangle_{\mathcal{T}} \Longrightarrow$
$(\tau = \mathit{OclVoid} \Longrightarrow P) \Longrightarrow$
$(\bigwedge \mathcal{C}.\ \tau = \langle \mathcal{C} \rangle_{\mathcal{T}} \Longrightarrow \mathcal{C} \leq \mathcal{D} \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-eq-x-Enum* [*elim*!]:
$\tau \leq \mathit{Enum}\ \mathcal{E} \Longrightarrow$
$(\tau = \mathit{OclVoid} \Longrightarrow P) \Longrightarrow$
$(\tau = \mathit{Enum}\ \mathcal{E} \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

2.2.6 Simplification Rules

lemma *basic-type-less-left-simps* [*simp*]:
$\mathit{OclAny} < \sigma = \mathit{False}$
$\mathit{OclVoid} < \sigma = (\sigma \neq \mathit{OclVoid})$
$\mathit{Boolean} < \sigma = (\sigma = \mathit{OclAny})$
$\mathit{Real} < \sigma = (\sigma = \mathit{OclAny})$
$\mathit{Integer} < \sigma = (\sigma = \mathit{OclAny} \vee \sigma = \mathit{Real})$
$\mathit{UnlimitedNatural} < \sigma = (\sigma = \mathit{OclAny} \vee \sigma = \mathit{Real} \vee \sigma = \mathit{Integer})$
$\mathit{String} < \sigma = (\sigma = \mathit{OclAny})$
$\mathit{ObjectType}\ \mathcal{C} < \sigma = (\exists \mathcal{D}.\ \sigma = \mathit{OclAny} \vee \sigma = \mathit{ObjectType}\ \mathcal{D} \wedge \mathcal{C} < \mathcal{D})$

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

lemma *basic-type-less-right-simps* [*simp*]:
$\tau < \mathit{OclAny} = (\tau \neq \mathit{OclAny})$
$\tau < \mathit{OclVoid} = \mathit{False}$
$\tau < \mathit{Boolean} = (\tau = \mathit{OclVoid})$
$\tau < \mathit{Real} = (\tau = \mathit{Integer} \vee \tau = \mathit{UnlimitedNatural} \vee \tau = \mathit{OclVoid})$
$\tau < \mathit{Integer} = (\tau = \mathit{UnlimitedNatural} \vee \tau = \mathit{OclVoid})$
$\tau < \mathit{UnlimitedNatural} = (\tau = \mathit{OclVoid})$
$\tau < \mathit{String} = (\tau = \mathit{OclVoid})$
$\tau < \mathit{ObjectType}\ \mathcal{D} = (\exists \mathcal{C}.\ \tau = \mathit{ObjectType}\ \mathcal{C} \wedge \mathcal{C} < \mathcal{D} \vee \tau = \mathit{OclVoid})$
$\tau < \mathit{Enum}\ \mathcal{E} = (\tau = \mathit{OclVoid})$
⟨*proof*⟩

2.3 Upper Semilattice of Basic Types

notation *sup* (**infixl** ‹⊔› *65*)

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

fun *sup-basic-type* **where**
$\langle \mathcal{C} \rangle_{\mathcal{T}} \sqcup \sigma = (\mathit{case}\ \sigma\ \mathit{of}\ \mathit{OclVoid} \Rightarrow \langle \mathcal{C} \rangle_{\mathcal{T}} \mid \langle \mathcal{D} \rangle_{\mathcal{T}} \Rightarrow \langle \mathcal{C} \sqcup \mathcal{D} \rangle_{\mathcal{T}} \mid \text{-} \Rightarrow \mathit{OclAny})$
| $\tau \sqcup \sigma = (\mathit{if}\ \tau \leq \sigma\ \mathit{then}\ \sigma\ \mathit{else}\ (\mathit{if}\ \sigma \leq \tau\ \mathit{then}\ \tau\ \mathit{else}\ \mathit{OclAny}))$

lemma *sup-ge1-ObjectType*:
$\langle \mathcal{C} \rangle_{\mathcal{T}} \leq \langle \mathcal{C} \rangle_{\mathcal{T}} \sqcup \sigma$
⟨*proof*⟩

lemma *sup-ge1-basic-type*:
$\tau \leq \tau \sqcup \sigma$
for $\tau\ \sigma$ :: *'a basic-type*
⟨*proof*⟩

lemma *sup-commut-basic-type*:
$\tau \sqcup \sigma = \sigma \sqcup \tau$
for $\tau\ \sigma$ :: *'a basic-type*
⟨*proof*⟩

lemma *sup-least-basic-type*:
$\tau \leq \varrho \Longrightarrow \sigma \leq \varrho \Longrightarrow \tau \sqcup \sigma \leq \varrho$
for $\tau\ \sigma\ \varrho$ :: *'a basic-type*
⟨*proof*⟩

instance
⟨*proof*⟩

end

2.4 Code Setup

code-pred *basic-subtype* ⟨*proof*⟩

fun *basic-subtype-fun* :: *'a*::*order basic-type* ⇒ *'a basic-type* ⇒ *bool* **where**
  basic-subtype-fun OclAny σ = *False*
| *basic-subtype-fun OclVoid σ* = (*σ* ≠ *OclVoid*)
| *basic-subtype-fun Boolean σ* = (*σ* = *OclAny*)
| *basic-subtype-fun Real σ* = (*σ* = *OclAny*)
| *basic-subtype-fun Integer σ* = (*σ* = *Real* ∨ *σ* = *OclAny*)
| *basic-subtype-fun UnlimitedNatural σ* = (*σ* = *Integer* ∨ *σ* = *Real* ∨ *σ* = *OclAny*)

| *basic-subtype-fun String σ* = (*σ* = *OclAny*)
| *basic-subtype-fun* ⟨*C*⟩$_\mathcal{T}$ *σ* = (*case σ*
   of ⟨*D*⟩$_\mathcal{T}$ ⇒ *C* < *D*
    | *OclAny* ⇒ *True*
    | - ⇒ *False*)
| *basic-subtype-fun* (*Enum* -) *σ* = (*σ* = *OclAny*)

lemma *less-basic-type-code* [*code*]:
  (<) = *basic-subtype-fun*
⟨*proof*⟩

lemma *less-eq-basic-type-code* [*code*]:
  (≤) = (λ*x y*. *basic-subtype-fun x y* ∨ *x* = *y*)
 ⟨*proof*⟩

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* $\rightharpoonup_f$ *'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*$_\perp \equiv \perp$

instantiation *type* :: (*type*) *size*
begin

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

| *size-type* (*Optional* τ) = *0*
| *size-type* (*Collection* τ) = *Suc* (*size-type* τ)
| *size-type* (*Set* τ) = *Suc* (*size-type* τ)
| *size-type* (*OrderedSet* τ) = *Suc* (*size-type* τ)
| *size-type* (*Bag* τ) = *Suc* (*size-type* τ)
| *size-type* (*Sequence* τ) = *Suc* (*size-type* τ)
| *size-type* (*Tuple* π) = *Suc* (*ffold tcf 0* (*fset-of-fmap* (*fmmap size-type* π)))

instance ⟨*proof*⟩

end

inductive *subtype* :: *'a*::*order type* ⇒ *'a type* ⇒ *bool* (**infix** ‹⊏› *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* σ
| $\tau \sqsubset \sigma \Longrightarrow$ *Set* $\tau \sqsubset$ *Set* σ
| $\tau \sqsubset \sigma \Longrightarrow$ *OrderedSet* $\tau \sqsubset$ *OrderedSet* σ
| $\tau \sqsubset \sigma \Longrightarrow$ *Bag* $\tau \sqsubset$ *Bag* σ
| $\tau \sqsubset \sigma \Longrightarrow$ *Sequence* $\tau \sqsubset$ *Sequence* σ
| *Set* $\tau \sqsubset$ *Collection* τ
| *OrderedSet* $\tau \sqsubset$ *Collection* τ
| *Bag* $\tau \sqsubset$ *Collection* τ
| *Sequence* $\tau \sqsubset$ *Collection* τ
| *Collection OclSuper* $\sqsubset$ *OclSuper*

| *strict-subtuple* $(\lambda\tau\ \sigma.\ \tau \sqsubset \sigma \vee \tau = \sigma)\ \pi\ \xi \Longrightarrow$
  Tuple $\pi \sqsubset$ *Tuple* ξ
| *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* σ
inductive-cases *subtype-x-Set* [*elim*!]: $\tau \sqsubset$ *Set* σ
inductive-cases *subtype-x-OrderedSet* [*elim*!]: $\tau \sqsubset$ *OrderedSet* σ
inductive-cases *subtype-x-Bag* [*elim*!]: $\tau \sqsubset$ *Bag* σ
inductive-cases *subtype-x-Sequence* [*elim*!]: $\tau \sqsubset$ *Sequence* σ
inductive-cases *subtype-x-Tuple* [*elim*!]: $\tau \sqsubset$ *Tuple* π

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

lemma *subtype-asym*:

$\tau \sqsubset \sigma \Longrightarrow \sigma \sqsubset \tau \Longrightarrow \mathit{False}$
⟨*proof*⟩

3.2 Constructors Bijectivity on Transitive Closures

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

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

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

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

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

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

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

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

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

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

3.3 Partial Order of Types

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

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

3.3.1 Strict Introduction Rules

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

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

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

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

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

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

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

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

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

shows $\tau < \mathit{Tuple}\ \xi$
⟨*proof*⟩

lemma *type-less-x-OclSuper-intro* [*intro*]:
$\tau \neq \mathit{OclSuper} \Longrightarrow \tau < \mathit{OclSuper}$
⟨*proof*⟩

3.3.2 Strict Elimination Rules

lemma *type-less-x-Required* [*elim*!]:
assumes $\tau < \sigma[1]$
and $\bigwedge \varrho.\ \tau = \varrho[1] \Longrightarrow \varrho < \sigma \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *type-less-x-Optional* [*elim*!]:
$\tau < \sigma[?] \Longrightarrow$
$(\bigwedge \varrho.\ \tau = \varrho[1] \Longrightarrow \varrho \leq \sigma \Longrightarrow P) \Longrightarrow$
$(\bigwedge \varrho.\ \tau = \varrho[?] \Longrightarrow \varrho < \sigma \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

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

lemma *type-less-x-Set* [*elim*!]:
assumes $\tau < \mathit{Set}\ \sigma$
and $\bigwedge \varrho.\ \tau = \mathit{Set}\ \varrho \Longrightarrow \varrho < \sigma \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *type-less-x-OrderedSet* [*elim*!]:
assumes $\tau < \mathit{OrderedSet}\ \sigma$
and $\bigwedge \varrho.\ \tau = \mathit{OrderedSet}\ \varrho \Longrightarrow \varrho < \sigma \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *type-less-x-Bag* [*elim*!]:
assumes $\tau < \mathit{Bag}\ \sigma$
and $\bigwedge \varrho.\ \tau = \mathit{Bag}\ \varrho \Longrightarrow \varrho < \sigma \Longrightarrow P$
shows P
⟨*proof*⟩

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

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

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

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

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

3.3.3 Properties

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

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

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

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

lemma *order-trans-type*:
$\tau \leq \sigma \Longrightarrow \sigma \leq \varrho \Longrightarrow \tau \leq \varrho$
for $\tau\ \sigma\ \varrho$ :: *'a type*
⟨*proof*⟩

lemma *antisym-type*:
$\tau \leq \sigma \Longrightarrow \sigma \leq \tau \Longrightarrow \tau = \sigma$
for $\tau\ \sigma$ :: *'a type*
⟨*proof*⟩

instance
⟨*proof*⟩

end

3.3.4 Non-Strict Introduction Rules

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

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

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

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

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

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

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

lemma *type-less-eq-x-Tuple-intro* [*intro*]:
$\tau = Tuple\ \pi \Longrightarrow subtuple\ (\leq)\ \pi\ \xi \Longrightarrow \tau \leq Tuple\ \xi$
⟨*proof*⟩

lemma *type-less-eq-x-OclSuper-intro* [*intro*]:
$\tau \leq OclSuper$
⟨*proof*⟩

3.3.5 Non-Strict Elimination Rules

lemma *type-less-eq-x-Required* [*elim*!]:
$\tau \leq \sigma[1] \Longrightarrow$
$(\bigwedge \varrho.\ \tau = \varrho[1] \Longrightarrow \varrho \leq \sigma \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-eq-x-Optional* [*elim*!]:
$\tau \leq \sigma[?] \Longrightarrow$
$(\bigwedge \varrho.\ \tau = \varrho[1] \Longrightarrow \varrho \leq \sigma \Longrightarrow P) \Longrightarrow$
$(\bigwedge \varrho.\ \tau = \varrho[?] \Longrightarrow \varrho \leq \sigma \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-eq-x-Collection* [*elim*!]:
$\tau \leq Collection\ \sigma \Longrightarrow$
$(\bigwedge \varrho.\ \tau = Set\ \varrho \Longrightarrow \varrho \leq \sigma \Longrightarrow P) \Longrightarrow$
$(\bigwedge \varrho.\ \tau = OrderedSet\ \varrho \Longrightarrow \varrho \leq \sigma \Longrightarrow P) \Longrightarrow$
$(\bigwedge \varrho.\ \tau = Bag\ \varrho \Longrightarrow \varrho \leq \sigma \Longrightarrow P) \Longrightarrow$
$(\bigwedge \varrho.\ \tau = Sequence\ \varrho \Longrightarrow \varrho \leq \sigma \Longrightarrow P) \Longrightarrow$
$(\bigwedge \varrho.\ \tau = Collection\ \varrho \Longrightarrow \varrho \leq \sigma \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-eq-x-Set* [*elim*!]:
$\tau \leq Set\ \sigma \Longrightarrow$
$(\bigwedge \varrho.\ \tau = Set\ \varrho \Longrightarrow \varrho \leq \sigma \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-eq-x-OrderedSet* [*elim*!]:
$\tau \leq OrderedSet\ \sigma \Longrightarrow$
$(\bigwedge \varrho.\ \tau = OrderedSet\ \varrho \Longrightarrow \varrho \leq \sigma \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-eq-x-Bag* [*elim*!]:
$\tau \leq Bag\ \sigma \Longrightarrow$
$(\bigwedge \varrho.\ \tau = Bag\ \varrho \Longrightarrow \varrho \leq \sigma \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-eq-x-Sequence* [*elim*!]:
$\tau \leq Sequence\ \sigma \Longrightarrow$
$(\bigwedge \varrho.\ \tau = Sequence\ \varrho \Longrightarrow \varrho \leq \sigma \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-x-Tuple* [*elim*!]:
$\tau < Tuple\ \xi \Longrightarrow$
$(\bigwedge \pi.\ \tau = Tuple\ \pi \Longrightarrow strict\text{-}subtuple\ (\leq)\ \pi\ \xi \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *type-less-eq-x-Tuple* [*elim*!]:
$\tau \leq Tuple\ \xi \Longrightarrow$
$(\bigwedge \pi.\ \tau = Tuple\ \pi \Longrightarrow subtuple\ (\leq)\ \pi\ \xi \Longrightarrow P) \Longrightarrow P$

⟨*proof*⟩

3.3.6 Simplification Rules

lemma *type-less-left-simps* [*simp*]:
$OclSuper < \sigma = False$
$\varrho[1] < \sigma = (\exists \upsilon.$
$\sigma = OclSuper \vee$
$\sigma = \upsilon[1] \wedge \varrho < \upsilon \vee$
$\sigma = \upsilon[?] \wedge \varrho \leq \upsilon)$
$\varrho[?] < \sigma = (\exists \upsilon.$
$\sigma = OclSuper \vee$
$\sigma = \upsilon[?] \wedge \varrho < \upsilon)$
$Collection\ \tau < \sigma = (\exists \varphi.$
$\sigma = OclSuper \vee$
$\sigma = Collection\ \varphi \wedge \tau < \varphi)$
$Set\ \tau < \sigma = (\exists \varphi.$
$\sigma = OclSuper \vee$
$\sigma = Collection\ \varphi \wedge \tau \leq \varphi \vee$
$\sigma = Set\ \varphi \wedge \tau < \varphi)$
$OrderedSet\ \tau < \sigma = (\exists \varphi.$
$\sigma = OclSuper \vee$
$\sigma = Collection\ \varphi \wedge \tau \leq \varphi \vee$
$\sigma = OrderedSet\ \varphi \wedge \tau < \varphi)$
$Bag\ \tau < \sigma = (\exists \varphi.$
$\sigma = OclSuper \vee$
$\sigma = Collection\ \varphi \wedge \tau \leq \varphi \vee$
$\sigma = Bag\ \varphi \wedge \tau < \varphi)$
$Sequence\ \tau < \sigma = (\exists \varphi.$
$\sigma = OclSuper \vee$
$\sigma = Collection\ \varphi \wedge \tau \leq \varphi \vee$
$\sigma = Sequence\ \varphi \wedge \tau < \varphi)$
$Tuple\ \pi < \sigma = (\exists \xi.$
$\sigma = OclSuper \vee$
$\sigma = Tuple\ \xi \wedge strict\text{-}subtuple\ (\leq)\ \pi\ \xi)$
⟨*proof*⟩

lemma *type-less-right-simps* [*simp*]:
$\tau < OclSuper = (\tau \neq OclSuper)$
$\tau < \upsilon[1] = (\exists \varrho.\ \tau = \varrho[1] \wedge \varrho < \upsilon)$
$\tau < \upsilon[?] = (\exists \varrho.\ \tau = \varrho[1] \wedge \varrho \leq \upsilon \vee \tau = \varrho[?] \wedge \varrho < \upsilon)$
$\tau < Collection\ \sigma = (\exists \varphi.$
$\tau = Collection\ \varphi \wedge \varphi < \sigma \vee$
$\tau = Set\ \varphi \wedge \varphi \leq \sigma \vee$
$\tau = OrderedSet\ \varphi \wedge \varphi \leq \sigma \vee$
$\tau = Bag\ \varphi \wedge \varphi \leq \sigma \vee$
$\tau = Sequence\ \varphi \wedge \varphi \leq \sigma)$
$\tau < Set\ \sigma = (\exists \varphi.\ \tau = Set\ \varphi \wedge \varphi < \sigma)$
$\tau < OrderedSet\ \sigma = (\exists \varphi.\ \tau = OrderedSet\ \varphi \wedge \varphi < \sigma)$

$\tau < \mathit{Bag}\ \sigma = (\exists \varphi.\ \tau = \mathit{Bag}\ \varphi \wedge \varphi < \sigma)$
$\tau < \mathit{Sequence}\ \sigma = (\exists \varphi.\ \tau = \mathit{Sequence}\ \varphi \wedge \varphi < \sigma)$
$\tau < \mathit{Tuple}\ \xi = (\exists \pi.\ \tau = \mathit{Tuple}\ \pi \wedge \mathit{strict\text{-}subtuple}\ (\leq)\ \pi\ \xi)$
⟨*proof*⟩

3.4 Upper Semilattice of Types

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

fun *sup-type* **where**
$\mathit{OclSuper} \sqcup \sigma = \mathit{OclSuper}$
| $\mathit{Required}\ \tau \sqcup \sigma = (\mathit{case}\ \sigma$
$\mathit{of}\ \varrho[1] \Rightarrow (\tau \sqcup \varrho)[1]$
$\mid \varrho[?] \Rightarrow (\tau \sqcup \varrho)[?]$
$\mid \text{-} \Rightarrow \mathit{OclSuper})$
| $\mathit{Optional}\ \tau \sqcup \sigma = (\mathit{case}\ \sigma$
$\mathit{of}\ \varrho[1] \Rightarrow (\tau \sqcup \varrho)[?]$
$\mid \varrho[?] \Rightarrow (\tau \sqcup \varrho)[?]$
$\mid \text{-} \Rightarrow \mathit{OclSuper})$
| $\mathit{Collection}\ \tau \sqcup \sigma = (\mathit{case}\ \sigma$
$\mathit{of}\ \mathit{Collection}\ \varrho \Rightarrow \mathit{Collection}\ (\tau \sqcup \varrho)$
$\mid \mathit{Set}\ \varrho \Rightarrow \mathit{Collection}\ (\tau \sqcup \varrho)$
$\mid \mathit{OrderedSet}\ \varrho \Rightarrow \mathit{Collection}\ (\tau \sqcup \varrho)$
$\mid \mathit{Bag}\ \varrho \Rightarrow \mathit{Collection}\ (\tau \sqcup \varrho)$
$\mid \mathit{Sequence}\ \varrho \Rightarrow \mathit{Collection}\ (\tau \sqcup \varrho)$
$\mid \text{-} \Rightarrow \mathit{OclSuper})$
| $\mathit{Set}\ \tau \sqcup \sigma = (\mathit{case}\ \sigma$
$\mathit{of}\ \mathit{Collection}\ \varrho \Rightarrow \mathit{Collection}\ (\tau \sqcup \varrho)$
$\mid \mathit{Set}\ \varrho \Rightarrow \mathit{Set}\ (\tau \sqcup \varrho)$
$\mid \mathit{OrderedSet}\ \varrho \Rightarrow \mathit{Collection}\ (\tau \sqcup \varrho)$
$\mid \mathit{Bag}\ \varrho \Rightarrow \mathit{Collection}\ (\tau \sqcup \varrho)$
$\mid \mathit{Sequence}\ \varrho \Rightarrow \mathit{Collection}\ (\tau \sqcup \varrho)$
$\mid \text{-} \Rightarrow \mathit{OclSuper})$
| $\mathit{OrderedSet}\ \tau \sqcup \sigma = (\mathit{case}\ \sigma$
$\mathit{of}\ \mathit{Collection}\ \varrho \Rightarrow \mathit{Collection}\ (\tau \sqcup \varrho)$
$\mid \mathit{Set}\ \varrho \Rightarrow \mathit{Collection}\ (\tau \sqcup \varrho)$
$\mid \mathit{OrderedSet}\ \varrho \Rightarrow \mathit{OrderedSet}\ (\tau \sqcup \varrho)$
$\mid \mathit{Bag}\ \varrho \Rightarrow \mathit{Collection}\ (\tau \sqcup \varrho)$
$\mid \mathit{Sequence}\ \varrho \Rightarrow \mathit{Collection}\ (\tau \sqcup \varrho)$
$\mid \text{-} \Rightarrow \mathit{OclSuper})$
| $\mathit{Bag}\ \tau \sqcup \sigma = (\mathit{case}\ \sigma$
$\mathit{of}\ \mathit{Collection}\ \varrho \Rightarrow \mathit{Collection}\ (\tau \sqcup \varrho)$
$\mid \mathit{Set}\ \varrho \Rightarrow \mathit{Collection}\ (\tau \sqcup \varrho)$
$\mid \mathit{OrderedSet}\ \varrho \Rightarrow \mathit{Collection}\ (\tau \sqcup \varrho)$
$\mid \mathit{Bag}\ \varrho \Rightarrow \mathit{Bag}\ (\tau \sqcup \varrho)$
$\mid \mathit{Sequence}\ \varrho \Rightarrow \mathit{Collection}\ (\tau \sqcup \varrho)$
$\mid \text{-} \Rightarrow \mathit{OclSuper})$
| $\mathit{Sequence}\ \tau \sqcup \sigma = (\mathit{case}\ \sigma$

```
    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 (fmmerge-fun (⊔) π ξ)
     | - ⇒ OclSuper)

lemma sup-ge1-type:
  τ ≤ τ ⊔ σ
 for τ σ :: 'a type
⟨proof⟩

lemma sup-commut-type:
  τ ⊔ σ = σ ⊔ τ
 for τ σ :: 'a type
⟨proof⟩

lemma sup-least-type:
  τ ≤ ϱ ⟹ σ ≤ ϱ ⟹ τ ⊔ σ ≤ ϱ
 for τ σ ϱ :: 'a type
⟨proof⟩

instance
 ⟨proof⟩

end
```

3.5 Helper Relations

```
abbreviation between (‹-/ = -—-› [51, 51, 51] 50) where
  x = y—z ≡ y ≤ x ∧ x ≤ z

inductive element-type where
  element-type (Collection τ) τ
|  element-type (Set τ) τ
|  element-type (OrderedSet τ) τ
|  element-type (Bag τ) τ
|  element-type (Sequence τ) τ

lemma element-type-alt-simps:
  element-type τ σ =
   (Collection σ = τ ∨
    Set σ = τ ∨
    OrderedSet σ = τ ∨
    Bag σ = τ ∨
    Sequence σ = τ)
```

⟨*proof*⟩

inductive *update-element-type* **where**
update-element-type (*Collection* -) τ (*Collection* τ)
| *update-element-type* (*Set* -) τ (*Set* τ)
| *update-element-type* (*OrderedSet* -) τ (*OrderedSet* τ)
| *update-element-type* (*Bag* -) τ (*Bag* τ)
| *update-element-type* (*Sequence* -) τ (*Sequence* τ)

inductive *to-unique-collection* **where**
to-unique-collection (*Collection* τ) (*Set* τ)
| *to-unique-collection* (*Set* τ) (*Set* τ)
| *to-unique-collection* (*OrderedSet* τ) (*OrderedSet* τ)
| *to-unique-collection* (*Bag* τ) (*Set* τ)
| *to-unique-collection* (*Sequence* τ) (*OrderedSet* τ)

inductive *to-nonunique-collection* **where**
to-nonunique-collection (*Collection* τ) (*Bag* τ)
| *to-nonunique-collection* (*Set* τ) (*Bag* τ)
| *to-nonunique-collection* (*OrderedSet* τ) (*Sequence* τ)
| *to-nonunique-collection* (*Bag* τ) (*Bag* τ)
| *to-nonunique-collection* (*Sequence* τ) (*Sequence* τ)

inductive *to-ordered-collection* **where**
to-ordered-collection (*Collection* τ) (*Sequence* τ)
| *to-ordered-collection* (*Set* τ) (*OrderedSet* τ)
| *to-ordered-collection* (*OrderedSet* τ) (*OrderedSet* τ)
| *to-ordered-collection* (*Bag* τ) (*Sequence* τ)
| *to-ordered-collection* (*Sequence* τ) (*Sequence* τ)

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* τ) = *to-single-type* τ
| *to-single-type* (*Set* τ) = *to-single-type* τ
| *to-single-type* (*OrderedSet* τ) = *to-single-type* τ
| *to-single-type* (*Bag* τ) = *to-single-type* τ
| *to-single-type* (*Sequence* τ) = *to-single-type* τ
| *to-single-type* (*Tuple* π) = *Tuple* π

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* τ) = *Collection* (*to-optional-type-nested* τ)
| *to-optional-type-nested* (*Set* τ) = *Set* (*to-optional-type-nested* τ)
| *to-optional-type-nested* (*OrderedSet* τ) = *OrderedSet* (*to-optional-type-nested* τ)

| *to-optional-type-nested* (*Bag* τ) = *Bag* (*to-optional-type-nested* τ)
| *to-optional-type-nested* (*Sequence* τ) = *Sequence* (*to-optional-type-nested* τ)
| *to-optional-type-nested* (*Tuple* π) = *Tuple* (*fmmap to-optional-type-nested* π)

3.6 Determinism

lemma *element-type-det*:
element-type τ $\sigma_1 \Longrightarrow$
element-type τ $\sigma_2 \Longrightarrow \sigma_1 = \sigma_2$
⟨*proof*⟩

lemma *update-element-type-det*:
update-element-type τ σ $\varrho_1 \Longrightarrow$
update-element-type τ σ $\varrho_2 \Longrightarrow \varrho_1 = \varrho_2$
⟨*proof*⟩

lemma *to-unique-collection-det*:
to-unique-collection τ $\sigma_1 \Longrightarrow$
to-unique-collection τ $\sigma_2 \Longrightarrow \sigma_1 = \sigma_2$
⟨*proof*⟩

lemma *to-nonunique-collection-det*:
to-nonunique-collection τ $\sigma_1 \Longrightarrow$
to-nonunique-collection τ $\sigma_2 \Longrightarrow \sigma_1 = \sigma_2$
⟨*proof*⟩

lemma *to-ordered-collection-det*:
to-ordered-collection τ $\sigma_1 \Longrightarrow$
to-ordered-collection τ $\sigma_2 \Longrightarrow \sigma_1 = \sigma_2$
⟨*proof*⟩

3.7 Code Setup

code-pred *subtype* ⟨*proof*⟩

function *subtype-fun* :: *'a*::*order type* ⇒ *'a type* ⇒ *bool* **where**
subtype-fun OclSuper - = *False*
| *subtype-fun* (*Required* τ) σ = (*case* σ
of OclSuper ⇒ *True*
| $\varrho[1]$ ⇒ *basic-subtype-fun* τ ϱ
| $\varrho[?]$ ⇒ *basic-subtype-fun* τ $\varrho \vee \tau = \varrho$
| - ⇒ *False*)
| *subtype-fun* (*Optional* τ) σ = (*case* σ

```
    of OclSuper ⇒ True
     | ϱ[?] ⇒ basic-subtype-fun τ ϱ
     | - ⇒ False)
|  subtype-fun (Collection τ) σ = (case σ
    of OclSuper ⇒ True
     | Collection ϱ ⇒ subtype-fun τ ϱ
     | - ⇒ False)
|  subtype-fun (Set τ) σ = (case σ
    of OclSuper ⇒ True
     | Collection ϱ ⇒ subtype-fun τ ϱ ∨ τ = ϱ
     | Set ϱ ⇒ subtype-fun τ ϱ
     | - ⇒ False)
|  subtype-fun (OrderedSet τ) σ = (case σ
    of OclSuper ⇒ True
     | Collection ϱ ⇒ subtype-fun τ ϱ ∨ τ = ϱ
     | OrderedSet ϱ ⇒ subtype-fun τ ϱ
     | - ⇒ False)
|  subtype-fun (Bag τ) σ = (case σ
    of OclSuper ⇒ True
     | Collection ϱ ⇒ subtype-fun τ ϱ ∨ τ = ϱ
     | Bag ϱ ⇒ subtype-fun τ ϱ
     | - ⇒ False)
|  subtype-fun (Sequence τ) σ = (case σ
    of OclSuper ⇒ True
     | Collection ϱ ⇒ subtype-fun τ ϱ ∨ τ = ϱ
     | Sequence ϱ ⇒ subtype-fun τ ϱ
     | - ⇒ False)
|  subtype-fun (Tuple π) σ = (case σ
    of OclSuper ⇒ True
     | Tuple ξ ⇒ strict-subtuple-fun (λτ σ. subtype-fun τ σ ∨ τ = σ) π ξ
     | - ⇒ False)
  ⟨proof⟩
termination
  ⟨proof⟩

lemma less-type-code [code]:
   (<) = subtype-fun
⟨proof⟩

lemma less-eq-type-code [code]:
   (≤) = (λx y. subtype-fun x y ∨ x = y)
  ⟨proof⟩

code-pred element-type ⟨proof⟩
code-pred update-element-type ⟨proof⟩
code-pred to-unique-collection ⟨proof⟩
code-pred to-nonunique-collection ⟨proof⟩
code-pred to-ordered-collection ⟨proof⟩
```

end

Chapter 4

Abstract Syntax

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

4.1 Preliminaries

type-synonym *vname* = *String.literal*
type-synonym *'a env* = *vname* $\rightharpoonup_f$ *'a*

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

typedef *unat* = *UNIV* :: *nat option set* ⟨*proof*⟩

definition *unat x* ≡ *Abs-unat* (*Some x*)

instantiation *unat* :: *infinity*
begin
definition ∞ ≡ *Abs-unat None*
instance ⟨*proof*⟩
end

free-constructors *cases-unat* **for**
unat
| ∞ :: *unat*
⟨*proof*⟩

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* | *CharactersOp*
| *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* ≡
  Call src k (*TypeOperation op ty*)
abbreviation *AttributeCall src k attr* ≡
  Call src k (*Attribute attr*)

abbreviation *AssociationEndCall src k from role* ≡
Call src k (*AssociationEnd from role*)
abbreviation *AssociationClassCall src k from cls* ≡
Call src k (*AssociationClass from cls*)
abbreviation *AssociationClassEndCall src k role* ≡
Call src k (*AssociationClassEnd role*)
abbreviation *OperationCall src k op as* ≡
Call src k (*Operation op as*)
abbreviation *TupleElementCall src k elem* ≡
Call src k (*TupleElement elem*)
abbreviation *IterateCall src k its its-ty v ty init body* ≡
Call src k (*Iterate its its-ty v ty init body*)
abbreviation *AnyIteratorCall src k its its-ty body* ≡
Call src k (*Iterator AnyIter its its-ty body*)
abbreviation *ClosureIteratorCall src k its its-ty body* ≡
Call src k (*Iterator ClosureIter its its-ty body*)
abbreviation *CollectIteratorCall src k its its-ty body* ≡
Call src k (*Iterator CollectIter its its-ty body*)
abbreviation *CollectNestedIteratorCall src k its its-ty body* ≡
Call src k (*Iterator CollectNestedIter its its-ty body*)
abbreviation *ExistsIteratorCall src k its its-ty body* ≡
Call src k (*Iterator ExistsIter its its-ty body*)
abbreviation *ForAllIteratorCall src k its its-ty body* ≡
Call src k (*Iterator ForAllIter its its-ty body*)
abbreviation *OneIteratorCall src k its its-ty body* ≡
Call src k (*Iterator OneIter its its-ty body*)
abbreviation *IsUniqueIteratorCall src k its its-ty body* ≡
Call src k (*Iterator IsUniqueIter its its-ty body*)
abbreviation *SelectIteratorCall src k its its-ty body* ≡
Call src k (*Iterator SelectIter its its-ty body*)
abbreviation *RejectIteratorCall src k its its-ty body* ≡
Call src k (*Iterator RejectIter its its-ty body*)
abbreviation *SortedByIteratorCall src k its its-ty body* ≡
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 ≡
let $\mathcal{C}$ = *assoc-end-class end in*
if assoc-end-max end ≤ (*1* :: *nat*) *then*
if assoc-end-min end = (*0* :: *nat*)
then $\langle\mathcal{C}\rangle_{\mathcal{T}}[?]$
else $\langle\mathcal{C}\rangle_{\mathcal{T}}[1]$
else
if assoc-end-unique end then
if assoc-end-ordered end
then OrderedSet $\langle\mathcal{C}\rangle_{\mathcal{T}}[1]$
else Set $\langle\mathcal{C}\rangle_{\mathcal{T}}[1]$
else
if assoc-end-ordered end
then Sequence $\langle\mathcal{C}\rangle_{\mathcal{T}}[1]$
else Bag $\langle\mathcal{C}\rangle_{\mathcal{T}}[1]$

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

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

definition *oper-type op* ≡
let params = *oper-out-params op in*
if length params = *0*
then oper-result op
else Tuple (*fmap-of-list* (*map* (λ*p*. (*param-name p*, *param-type p*))
(*params* @ [(*STR* ''*result*'', *oper-result op*, *Out*)])))

class *ocl-object-model* =
  fixes *classes* :: *'a* :: *semilattice-sup fset*
    and *attributes* :: *'a* $\rightharpoonup_f$ *attr* $\rightharpoonup_f$ *'a type*
    and *associations* :: *assoc* $\rightharpoonup_f$ *role* $\rightharpoonup_f$ *'a assoc-end*
    and *association-classes* :: *'a* $\rightharpoonup_f$ *assoc*
    and *operations* :: (*'a type*, *'a expr*) *oper-spec list*
    and *literals* :: *'a enum* $\rightharpoonup_f$ *elit fset*
  assumes *assoc-end-min-less-eq-max*:
    assoc $|\in|$ *fmdom associations* $\Longrightarrow$
    fmlookup associations assoc = *Some ends* $\Longrightarrow$
    role $|\in|$ *fmdom ends* $\Longrightarrow$
    fmlookup ends role = *Some end* $\Longrightarrow$
    assoc-end-min end $\leq$ *assoc-end-max end*
  assumes *association-ends-unique*:
    association-ends' *classes associations* $\mathcal{C}$ *from role* end_1 $\Longrightarrow$
    association-ends' *classes associations* $\mathcal{C}$ *from role* end_2 $\Longrightarrow$ $end_1 = end_2$
begin

interpretation *base*: *object-model*
  ⟨*proof*⟩

abbreviation *owned-attribute* ≡ *base.owned-attribute*
abbreviation *attribute* ≡ *base.attribute*
abbreviation *association-ends* ≡ *base.association-ends*
abbreviation *owned-association-end* ≡ *base.owned-association-end*
abbreviation *association-end* ≡ *base.association-end*
abbreviation *referred-by-association-class* ≡ *base.referred-by-association-class*
abbreviation *association-class-end* ≡ *base.association-class-end*
abbreviation *operation* ≡ *base.operation*
abbreviation *operation-defined* ≡ *base.operation-defined*
abbreviation *static-operation* ≡ *base.static-operation*
abbreviation *static-operation-defined* ≡ *base.static-operation-defined*
abbreviation *has-literal* ≡ *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 τ *AllInstancesOp* (*Set* τ)

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**
$\sigma < \tau \vee \tau < \sigma \Longrightarrow$
typeop-type DotCall OclAsTypeOp $\tau\ \sigma\ \sigma$

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

| *element-type* $\tau\ \varrho \Longrightarrow \sigma < \varrho \Longrightarrow$
update-element-type $\tau\ \sigma\ \upsilon \Longrightarrow$
typeop-type ArrowCall SelectByKindOp $\tau\ \sigma\ \upsilon$

| *element-type* $\tau\ \varrho \Longrightarrow \sigma < \varrho \Longrightarrow$
update-element-type $\tau\ \sigma\ \upsilon \Longrightarrow$
typeop-type ArrowCall SelectByTypeOp $\tau\ \sigma\ \upsilon$

6.1.3 OclSuper Operations

It makes sense to compare values only with compatible types.

inductive *super-binop-type*
:: *super-binop* $\Rightarrow$ ('*a* :: *order*) *type* $\Rightarrow$ '*a type* $\Rightarrow$ '*a type* $\Rightarrow$ *bool* **where**
$\tau \leq \sigma \vee \sigma < \tau \Longrightarrow$
super-binop-type EqualOp $\tau\ \sigma$ *Boolean*[*1*]
| $\tau \leq \sigma \vee \sigma < \tau \Longrightarrow$
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**

$\tau \leq OclAny[?] \Longrightarrow$
any-unop-type OclAsSetOp τ (*Set* (*to-required-type* τ))
| $\tau \leq OclAny[?] \Longrightarrow$
any-unop-type OclIsNewOp τ *Boolean*[*1*]
| $\tau \leq OclAny[?] \Longrightarrow$
any-unop-type OclIsUndefinedOp τ *Boolean*[*1*]
| $\tau \leq OclAny[?] \Longrightarrow$
any-unop-type OclIsInvalidOp τ *Boolean*[*1*]
| $\tau \leq OclAny[?] \Longrightarrow$
any-unop-type OclLocaleOp τ *String*[*1*]
| $\tau \leq OclAny[?] \Longrightarrow$
any-unop-type ToStringOp τ *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**
$\tau \leq Boolean[?] \Longrightarrow$
boolean-unop-type NotOp τ τ

inductive *boolean-binop-type* **where**
$\tau \sqcup \sigma = \varrho \Longrightarrow \varrho \leq Boolean[?] \Longrightarrow$
boolean-binop-type AndOp τ σ ϱ
| $\tau \sqcup \sigma = \varrho \Longrightarrow \varrho \leq Boolean[?] \Longrightarrow$
boolean-binop-type OrOp τ σ ϱ
| $\tau \sqcup \sigma = \varrho \Longrightarrow \varrho \leq Boolean[?] \Longrightarrow$
boolean-binop-type XorOp τ σ ϱ
| $\tau \sqcup \sigma = \varrho \Longrightarrow \varrho \leq Boolean[?] \Longrightarrow$
boolean-binop-type ImpliesOp τ σ ϱ

6.1.6 Numeric Operations

The expression *1 + 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**
$\tau = Real[1] \Longrightarrow$
numeric-unop-type UMinusOp τ *Real*[*1*]
| $\tau = UnlimitedNatural[1] - Integer[1] \Longrightarrow$

$\mathit{numeric\text{-}unop\text{-}type}\ \mathit{UMinusOp}\ \tau\ \mathit{Integer}[1]$

$\mid\ \tau = \mathit{Real}[1] \Longrightarrow$
$\mathit{numeric\text{-}unop\text{-}type}\ \mathit{AbsOp}\ \tau\ \mathit{Real}[1]$
$\mid\ \tau = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Integer}[1] \Longrightarrow$
$\mathit{numeric\text{-}unop\text{-}type}\ \mathit{AbsOp}\ \tau\ \mathit{Integer}[1]$

$\mid\ \tau = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Real}[1] \Longrightarrow$
$\mathit{numeric\text{-}unop\text{-}type}\ \mathit{FloorOp}\ \tau\ \mathit{Integer}[1]$
$\mid\ \tau = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Real}[1] \Longrightarrow$
$\mathit{numeric\text{-}unop\text{-}type}\ \mathit{RoundOp}\ \tau\ \mathit{Integer}[1]$

$\mid\ \tau = \mathit{UnlimitedNatural}[1] \Longrightarrow$
$\mathit{numeric\text{-}unop\text{-}type}\ \mathit{numeric\text{-}unop.ToIntegerOp}\ \tau\ \mathit{Integer}[1]$

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

$\mid\ \tau \sqcup \sigma = \mathit{Real}[1] \Longrightarrow$
$\mathit{numeric\text{-}binop\text{-}type}\ \mathit{MinusOp}\ \tau\ \sigma\ \mathit{Real}[1]$
$\mid\ \tau \sqcup \sigma = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Integer}[1] \Longrightarrow$
$\mathit{numeric\text{-}binop\text{-}type}\ \mathit{MinusOp}\ \tau\ \sigma\ \mathit{Integer}[1]$

$\mid\ \tau \sqcup \sigma = \varrho \Longrightarrow \varrho = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Real}[1] \Longrightarrow$
$\mathit{numeric\text{-}binop\text{-}type}\ \mathit{MultOp}\ \tau\ \sigma\ \varrho$

$\mid\ \tau = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Real}[1] \Longrightarrow \sigma = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Real}[1] \Longrightarrow$
$\mathit{numeric\text{-}binop\text{-}type}\ \mathit{DivideOp}\ \tau\ \sigma\ \mathit{Real}[1]$

$\mid\ \tau \sqcup \sigma = \varrho \Longrightarrow \varrho = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Integer}[1] \Longrightarrow$
$\mathit{numeric\text{-}binop\text{-}type}\ \mathit{DivOp}\ \tau\ \sigma\ \varrho$
$\mid\ \tau \sqcup \sigma = \varrho \Longrightarrow \varrho = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Integer}[1] \Longrightarrow$
$\mathit{numeric\text{-}binop\text{-}type}\ \mathit{ModOp}\ \tau\ \sigma\ \varrho$

$\mid\ \tau \sqcup \sigma = \varrho \Longrightarrow \varrho = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Real}[1] \Longrightarrow$
$\mathit{numeric\text{-}binop\text{-}type}\ \mathit{MaxOp}\ \tau\ \sigma\ \varrho$
$\mid\ \tau \sqcup \sigma = \varrho \Longrightarrow \varrho = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Real}[1] \Longrightarrow$
$\mathit{numeric\text{-}binop\text{-}type}\ \mathit{MinOp}\ \tau\ \sigma\ \varrho$

$\mid\ \tau = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Real}[1] \Longrightarrow \sigma = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Real}[1] \Longrightarrow$
$\mathit{numeric\text{-}binop\text{-}type}\ \mathit{numeric\text{-}binop.LessOp}\ \tau\ \sigma\ \mathit{Boolean}[1]$
$\mid\ \tau = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Real}[1] \Longrightarrow \sigma = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Real}[1] \Longrightarrow$
$\mathit{numeric\text{-}binop\text{-}type}\ \mathit{numeric\text{-}binop.LessEqOp}\ \tau\ \sigma\ \mathit{Boolean}[1]$
$\mid\ \tau = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Real}[1] \Longrightarrow \sigma = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Real}[1] \Longrightarrow$
$\mathit{numeric\text{-}binop\text{-}type}\ \mathit{numeric\text{-}binop.GreaterOp}\ \tau\ \sigma\ \mathit{Boolean}[1]$
$\mid\ \tau = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Real}[1] \Longrightarrow \sigma = \mathit{UnlimitedNatural}[1]\text{-}\mathit{Real}[1] \Longrightarrow$
$\mathit{numeric\text{-}binop\text{-}type}\ \mathit{numeric\text{-}binop.GreaterEqOp}\ \tau\ \sigma\ \mathit{Boolean}[1]$

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]*
| τ = *UnlimitedNatural[1]*−*Integer[1]* $\Longrightarrow$
string-binop-type AtOp String[1] τ *String[1]*

inductive *string-ternop-type* **where**
σ = *UnlimitedNatural[1]*−*Integer[1]* $\Longrightarrow$
ϱ = *UnlimitedNatural[1]*−*Integer[1]* $\Longrightarrow$
string-ternop-type SubstringOp String[1] σ ϱ *String[1]*

6.1.8 Collection Operations

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

inductive *collection-unop-type* **where**
element-type τ - $\Longrightarrow$
collection-unop-type CollectionSizeOp τ *Integer[1]*
| *element-type* τ - $\Longrightarrow$
collection-unop-type IsEmptyOp τ *Boolean[1]*
| *element-type* τ - $\Longrightarrow$
collection-unop-type NotEmptyOp τ *Boolean[1]*

| *element-type* τ σ $\Longrightarrow$ σ = *UnlimitedNatural[1]*−*Real[1]* $\Longrightarrow$
collection-unop-type CollectionMaxOp τ σ
| *element-type* τ σ $\Longrightarrow$ *operation* σ *STR ''max''* [σ] *oper* $\Longrightarrow$
collection-unop-type CollectionMaxOp τ σ

| *element-type* τ σ $\Longrightarrow$ σ = *UnlimitedNatural[1]*−*Real[1]* $\Longrightarrow$
collection-unop-type CollectionMinOp τ σ
| *element-type* τ σ $\Longrightarrow$ *operation* σ *STR ''min''* [σ] *oper* $\Longrightarrow$
collection-unop-type CollectionMinOp τ σ

| *element-type* τ σ $\Longrightarrow$ σ = *UnlimitedNatural[1]*−*Real[1]* $\Longrightarrow$

$$
\begin{array}{l}
\quad \mathit{collection\text{-}unop\text{-}type}\ \mathit{SumOp}\ \tau\ \sigma \\
\mid\ \mathit{element\text{-}type}\ \tau\ \sigma \Longrightarrow \mathit{operation}\ \sigma\ \mathit{STR}\ ''{+}''\ [\sigma]\ \mathit{oper} \Longrightarrow \\
\quad \mathit{collection\text{-}unop\text{-}type}\ \mathit{SumOp}\ \tau\ \sigma \\
\\
\mid\ \mathit{element\text{-}type}\ \tau\ \sigma \Longrightarrow \\
\quad \mathit{collection\text{-}unop\text{-}type}\ \mathit{AsSetOp}\ \tau\ (\mathit{Set}\ \sigma) \\
\mid\ \mathit{element\text{-}type}\ \tau\ \sigma \Longrightarrow \\
\quad \mathit{collection\text{-}unop\text{-}type}\ \mathit{AsOrderedSetOp}\ \tau\ (\mathit{OrderedSet}\ \sigma) \\
\mid\ \mathit{element\text{-}type}\ \tau\ \sigma \Longrightarrow \\
\quad \mathit{collection\text{-}unop\text{-}type}\ \mathit{AsBagOp}\ \tau\ (\mathit{Bag}\ \sigma) \\
\mid\ \mathit{element\text{-}type}\ \tau\ \sigma \Longrightarrow \\
\quad \mathit{collection\text{-}unop\text{-}type}\ \mathit{AsSequenceOp}\ \tau\ (\mathit{Sequence}\ \sigma) \\
\\
\mid\ \mathit{update\text{-}element\text{-}type}\ \tau\ (\mathit{to\text{-}single\text{-}type}\ \tau)\ \sigma \Longrightarrow \\
\quad \mathit{collection\text{-}unop\text{-}type}\ \mathit{FlattenOp}\ \tau\ \sigma \\
\\
\mid\ \mathit{collection\text{-}unop\text{-}type}\ \mathit{FirstOp}\ (\mathit{OrderedSet}\ \tau)\ \tau \\
\mid\ \mathit{collection\text{-}unop\text{-}type}\ \mathit{FirstOp}\ (\mathit{Sequence}\ \tau)\ \tau \\
\mid\ \mathit{collection\text{-}unop\text{-}type}\ \mathit{LastOp}\ (\mathit{OrderedSet}\ \tau)\ \tau \\
\mid\ \mathit{collection\text{-}unop\text{-}type}\ \mathit{LastOp}\ (\mathit{Sequence}\ \tau)\ \tau \\
\mid\ \mathit{collection\text{-}unop\text{-}type}\ \mathit{ReverseOp}\ (\mathit{OrderedSet}\ \tau)\ (\mathit{OrderedSet}\ \tau) \\
\mid\ \mathit{collection\text{-}unop\text{-}type}\ \mathit{ReverseOp}\ (\mathit{Sequence}\ \tau)\ (\mathit{Sequence}\ \tau)
\end{array}
$$

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 orginal 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}->inculdes(nullable_value) and
Sequence{1..3}->inculdes(null) and
Sequence{1..3}->inculdesAll(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:

$$
\begin{array}{l}
\mid\ \text{element-type}\ \tau\ \varrho \Longrightarrow \sigma \leq \varrho \Longrightarrow \sigma \neq \text{OclVoid}[?] \Longrightarrow \\
\quad \text{collection-binop-type ExcludingOp}\ \tau\ \sigma\ \tau \\
\mid\ \text{element-type}\ \tau\ \varrho \Longrightarrow \sigma \leq \varrho \Longrightarrow \sigma = \text{OclVoid}[?] \Longrightarrow \\
\quad \text{update-element-type}\ \tau\ (\text{to-required-type}\ \varrho)\ \upsilon \Longrightarrow \\
\quad \text{collection-binop-type ExcludingOp}\ \tau\ \sigma\ \upsilon
\end{array}
$$

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 τ $\varrho \Longrightarrow$ $\sigma \leq$ *to-optional-type-nested* $\varrho \Longrightarrow$
collection-binop-type IncludesOp τ σ *Boolean*[*1*]
| *element-type* τ $\varrho \Longrightarrow$ $\sigma \leq$ *to-optional-type-nested* $\varrho \Longrightarrow$
collection-binop-type ExcludesOp τ σ *Boolean*[*1*]
| *element-type* τ $\varrho \Longrightarrow$ $\sigma \leq$ *to-optional-type-nested* $\varrho \Longrightarrow$
collection-binop-type CountOp τ σ *Integer*[*1*]

| *element-type* τ $\varrho \Longrightarrow$ *element-type* σ $\upsilon \Longrightarrow$ $\upsilon \leq$ *to-optional-type-nested* $\varrho \Longrightarrow$
collection-binop-type IncludesAllOp τ σ *Boolean*[*1*]
| *element-type* τ $\varrho \Longrightarrow$ *element-type* σ $\upsilon \Longrightarrow$ $\upsilon \leq$ *to-optional-type-nested* $\varrho \Longrightarrow$
collection-binop-type ExcludesAllOp τ σ *Boolean*[*1*]

| *element-type* τ $\varrho \Longrightarrow$ *element-type* σ $\upsilon \Longrightarrow$
collection-binop-type ProductOp τ σ
(*Set* (*Tuple* (*fmap-of-list* [(*STR* ''*first*'', ϱ), (*STR* ''*second*'', υ)])))

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

| *collection-binop-type IntersectionOp* (*Set* τ) (*Set* σ) (*Set* ($\tau \sqcup \sigma$))
| *collection-binop-type IntersectionOp* (*Set* τ) (*Bag* σ) (*Set* ($\tau \sqcup \sigma$))
| *collection-binop-type IntersectionOp* (*Bag* τ) (*Set* σ) (*Set* ($\tau \sqcup \sigma$))
| *collection-binop-type IntersectionOp* (*Bag* τ) (*Bag* σ) (*Bag* ($\tau \sqcup \sigma$))

| *collection-binop-type SetMinusOp* (*Set* τ) (*Set* σ) (*Set* τ)
| *collection-binop-type SymmetricDifferenceOp* (*Set* τ) (*Set* σ) (*Set* ($\tau \sqcup \sigma$))

| *element-type* τ $\varrho \Longrightarrow$ *update-element-type* τ ($\varrho \sqcup \sigma$) $\upsilon \Longrightarrow$
collection-binop-type IncludingOp τ σ υ
| *element-type* τ $\varrho \Longrightarrow$ $\sigma \leq \varrho \Longrightarrow$
collection-binop-type ExcludingOp τ σ τ

| $\sigma \leq \tau \Longrightarrow$
collection-binop-type AppendOp (*OrderedSet* τ) σ (*OrderedSet* τ)
| $\sigma \leq \tau \Longrightarrow$
collection-binop-type AppendOp (*Sequence* τ) σ (*Sequence* τ)
| $\sigma \leq \tau \Longrightarrow$
collection-binop-type PrependOp (*OrderedSet* τ) σ (*OrderedSet* τ)
| $\sigma \leq \tau \Longrightarrow$
collection-binop-type PrependOp (*Sequence* τ) σ (*Sequence* τ)

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

| $\sigma \leq \tau \Longrightarrow$
$\mathit{collection\text{-}binop\text{-}type}\ \mathit{CollectionIndexOfOp}\ (\mathit{OrderedSet}\ \tau)\ \sigma\ \mathit{Integer}[1]$
| $\sigma \leq \tau \Longrightarrow$
$\mathit{collection\text{-}binop\text{-}type}\ \mathit{CollectionIndexOfOp}\ (\mathit{Sequence}\ \tau)\ \sigma\ \mathit{Integer}[1]$

inductive *collection-ternop-type* **where**
$\sigma = \mathit{UnlimitedNatural}[1] - \mathit{Integer}[1] \Longrightarrow \varrho \leq \tau \Longrightarrow$
$\mathit{collection\text{-}ternop\text{-}type}\ \mathit{InsertAtOp}\ (\mathit{OrderedSet}\ \tau)\ \sigma\ \varrho\ (\mathit{OrderedSet}\ \tau)$
| $\sigma = \mathit{UnlimitedNatural}[1] - \mathit{Integer}[1] \Longrightarrow \varrho \leq \tau \Longrightarrow$
$\mathit{collection\text{-}ternop\text{-}type}\ \mathit{InsertAtOp}\ (\mathit{Sequence}\ \tau)\ \sigma\ \varrho\ (\mathit{Sequence}\ \tau)$
| $\sigma = \mathit{UnlimitedNatural}[1] - \mathit{Integer}[1] \Longrightarrow$
$\varrho = \mathit{UnlimitedNatural}[1] - \mathit{Integer}[1] \Longrightarrow$
$\mathit{collection\text{-}ternop\text{-}type}\ \mathit{SubOrderedSetOp}\ (\mathit{OrderedSet}\ \tau)\ \sigma\ \varrho\ (\mathit{OrderedSet}\ \tau)$
| $\sigma = \mathit{UnlimitedNatural}[1] - \mathit{Integer}[1] \Longrightarrow$
$\varrho = \mathit{UnlimitedNatural}[1] - \mathit{Integer}[1] \Longrightarrow$
$\mathit{collection\text{-}ternop\text{-}type}\ \mathit{SubSequenceOp}\ (\mathit{Sequence}\ \tau)\ \sigma\ \varrho\ (\mathit{Sequence}\ \tau)$

6.1.9 Coercions

inductive *unop-type* **where**
$\mathit{any\text{-}unop\text{-}type}\ \mathit{op}\ \tau\ \sigma \Longrightarrow$
$\mathit{unop\text{-}type}\ (\mathit{Inl}\ \mathit{op})\ \mathit{DotCall}\ \tau\ \sigma$
| $\mathit{boolean\text{-}unop\text{-}type}\ \mathit{op}\ \tau\ \sigma \Longrightarrow$
$\mathit{unop\text{-}type}\ (\mathit{Inr}\ (\mathit{Inl}\ \mathit{op}))\ \mathit{DotCall}\ \tau\ \sigma$
| $\mathit{numeric\text{-}unop\text{-}type}\ \mathit{op}\ \tau\ \sigma \Longrightarrow$
$\mathit{unop\text{-}type}\ (\mathit{Inr}\ (\mathit{Inr}\ (\mathit{Inl}\ \mathit{op})))\ \mathit{DotCall}\ \tau\ \sigma$
| $\mathit{string\text{-}unop\text{-}type}\ \mathit{op}\ \tau\ \sigma \Longrightarrow$
$\mathit{unop\text{-}type}\ (\mathit{Inr}\ (\mathit{Inr}\ (\mathit{Inr}\ (\mathit{Inl}\ \mathit{op}))))\ \mathit{DotCall}\ \tau\ \sigma$
| $\mathit{collection\text{-}unop\text{-}type}\ \mathit{op}\ \tau\ \sigma \Longrightarrow$
$\mathit{unop\text{-}type}\ (\mathit{Inr}\ (\mathit{Inr}\ (\mathit{Inr}\ (\mathit{Inr}\ \mathit{op}))))\ \mathit{ArrowCall}\ \tau\ \sigma$

inductive *binop-type* **where**
$\mathit{super\text{-}binop\text{-}type}\ \mathit{op}\ \tau\ \sigma\ \varrho \Longrightarrow$
$\mathit{binop\text{-}type}\ (\mathit{Inl}\ \mathit{op})\ \mathit{DotCall}\ \tau\ \sigma\ \varrho$
| $\mathit{boolean\text{-}binop\text{-}type}\ \mathit{op}\ \tau\ \sigma\ \varrho \Longrightarrow$
$\mathit{binop\text{-}type}\ (\mathit{Inr}\ (\mathit{Inl}\ \mathit{op}))\ \mathit{DotCall}\ \tau\ \sigma\ \varrho$
| $\mathit{numeric\text{-}binop\text{-}type}\ \mathit{op}\ \tau\ \sigma\ \varrho \Longrightarrow$
$\mathit{binop\text{-}type}\ (\mathit{Inr}\ (\mathit{Inr}\ (\mathit{Inl}\ \mathit{op})))\ \mathit{DotCall}\ \tau\ \sigma\ \varrho$
| $\mathit{string\text{-}binop\text{-}type}\ \mathit{op}\ \tau\ \sigma\ \varrho \Longrightarrow$
$\mathit{binop\text{-}type}\ (\mathit{Inr}\ (\mathit{Inr}\ (\mathit{Inr}\ (\mathit{Inl}\ \mathit{op}))))\ \mathit{DotCall}\ \tau\ \sigma\ \varrho$
| $\mathit{collection\text{-}binop\text{-}type}\ \mathit{op}\ \tau\ \sigma\ \varrho \Longrightarrow$
$\mathit{binop\text{-}type}\ (\mathit{Inr}\ (\mathit{Inr}\ (\mathit{Inr}\ (\mathit{Inr}\ \mathit{op}))))\ \mathit{ArrowCall}\ \tau\ \sigma\ \varrho$

inductive *ternop-type* **where**
string-ternop-type op τ σ ϱ υ $\Longrightarrow$
ternop-type (*Inl op*) *DotCall* τ σ ϱ υ
| *collection-ternop-type op* τ σ ϱ υ $\Longrightarrow$
ternop-type (*Inr op*) *ArrowCall* τ σ ϱ υ

inductive *op-type* **where**
unop-type op k τ υ $\Longrightarrow$
op-type (*Inl op*) *k* τ [] υ
| *binop-type op k* τ σ υ $\Longrightarrow$
op-type (*Inr* (*Inl op*)) *k* τ [σ] υ
| *ternop-type op k* τ σ ϱ υ $\Longrightarrow$
op-type (*Inr* (*Inr* (*Inl op*))) *k* τ [σ, ϱ] υ
| *operation* τ *op* π *oper* $\Longrightarrow$
op-type (*Inr* (*Inr* (*Inr op*))) *DotCall* τ π (*oper-type oper*)

6.1.10 Simplification Rules

inductive-simps *op-type-alt-simps*:
mataop-type τ *op* σ
typeop-type k op τ σ ϱ

op-type op k τ π σ
unop-type op k τ σ
binop-type op k τ σ ϱ
ternop-type op k τ σ ϱ υ

any-unop-type op τ σ
boolean-unop-type op τ σ
numeric-unop-type op τ σ
string-unop-type op τ σ
collection-unop-type op τ σ

super-binop-type op τ σ ϱ
boolean-binop-type op τ σ ϱ
numeric-binop-type op τ σ ϱ
string-binop-type op τ σ ϱ
collection-binop-type op τ σ ϱ

string-ternop-type op τ σ ϱ υ
collection-ternop-type op τ σ ϱ υ

6.1.11 Determinism

lemma *typeop-type-det*:
typeop-type op k τ σ ϱ_1 $\Longrightarrow$
typeop-type op k τ σ ϱ_2 $\Longrightarrow$ $\varrho_1 = \varrho_2$
⟨*proof*⟩

lemma *any-unop-type-det*:

any-unop-type op τ σ_1 $\Longrightarrow$
any-unop-type op τ σ_2 $\Longrightarrow$ $\sigma_1 = \sigma_2$
⟨*proof*⟩

lemma *boolean-unop-type-det*:
boolean-unop-type op τ σ_1 $\Longrightarrow$
boolean-unop-type op τ σ_2 $\Longrightarrow$ $\sigma_1 = \sigma_2$
⟨*proof*⟩

lemma *numeric-unop-type-det*:
numeric-unop-type op τ σ_1 $\Longrightarrow$
numeric-unop-type op τ σ_2 $\Longrightarrow$ $\sigma_1 = \sigma_2$
⟨*proof*⟩

lemma *string-unop-type-det*:
string-unop-type op τ σ_1 $\Longrightarrow$
string-unop-type op τ σ_2 $\Longrightarrow$ $\sigma_1 = \sigma_2$
⟨*proof*⟩

lemma *collection-unop-type-det*:
collection-unop-type op τ σ_1 $\Longrightarrow$
collection-unop-type op τ σ_2 $\Longrightarrow$ $\sigma_1 = \sigma_2$
⟨*proof*⟩

lemma *unop-type-det*:
unop-type op k τ σ_1 $\Longrightarrow$
unop-type op k τ σ_2 $\Longrightarrow$ $\sigma_1 = \sigma_2$
⟨*proof*⟩

lemma *super-binop-type-det*:
super-binop-type op τ σ ϱ_1 $\Longrightarrow$
super-binop-type op τ σ ϱ_2 $\Longrightarrow$ $\varrho_1 = \varrho_2$
⟨*proof*⟩

lemma *boolean-binop-type-det*:
boolean-binop-type op τ σ ϱ_1 $\Longrightarrow$
boolean-binop-type op τ σ ϱ_2 $\Longrightarrow$ $\varrho_1 = \varrho_2$
⟨*proof*⟩

lemma *numeric-binop-type-det*:
numeric-binop-type op τ σ ϱ_1 $\Longrightarrow$
numeric-binop-type op τ σ ϱ_2 $\Longrightarrow$ $\varrho_1 = \varrho_2$
⟨*proof*⟩

lemma *string-binop-type-det*:
string-binop-type op τ σ ϱ_1 $\Longrightarrow$
string-binop-type op τ σ ϱ_2 $\Longrightarrow$ $\varrho_1 = \varrho_2$
⟨*proof*⟩

lemma *collection-binop-type-det*:
collection-binop-type op τ σ ϱ$_1$ ⟹
collection-binop-type op τ σ ϱ$_2$ ⟹ *ϱ*$_1$ = *ϱ*$_2$
⟨*proof*⟩

lemma *binop-type-det*:
binop-type op k τ σ ϱ$_1$ ⟹
binop-type op k τ σ ϱ$_2$ ⟹ *ϱ*$_1$ = *ϱ*$_2$
⟨*proof*⟩

lemma *string-ternop-type-det*:
string-ternop-type op τ σ ϱ υ$_1$ ⟹
string-ternop-type op τ σ ϱ υ$_2$ ⟹ *υ*$_1$ = *υ*$_2$
⟨*proof*⟩

lemma *collection-ternop-type-det*:
collection-ternop-type op τ σ ϱ υ$_1$ ⟹
collection-ternop-type op τ σ ϱ υ$_2$ ⟹ *υ*$_1$ = *υ*$_2$
⟨*proof*⟩

lemma *ternop-type-det*:
ternop-type op k τ σ ϱ υ$_1$ ⟹
ternop-type op k τ σ ϱ υ$_2$ ⟹ *υ*$_1$ = *υ*$_2$
⟨*proof*⟩

lemma *op-type-det*:
op-type op k τ π σ ⟹
op-type op k τ π ϱ ⟹ *σ* = *ϱ*
⟨*proof*⟩

6.2 Expressions Typing

The following typing rules are preliminary. The final rules are given at the end of the next chapter.

inductive *typing* :: ('*a* :: *ocl-object-model*) *type env* ⇒ '*a expr* ⇒ '*a type* ⇒ *bool*
(‹(*1-*/ ⊢$_E$/ (- :/ -))› [*51*,*51*,*51*] *50*)
and *collection-parts-typing* (‹(*1-*/ ⊢$_C$/ (- :/ -))› [*51*,*51*,*51*] *50*)
and *collection-part-typing* (‹(*1-*/ ⊢$_P$/ (- :/ -))› [*51*,*51*,*51*] *50*)
and *iterator-typing* (‹(*1-*/ ⊢$_I$/ (- :/ -))› [*51*,*51*,*51*] *50*)
and *expr-list-typing* (‹(*1-*/ ⊢$_L$/ (- :/ -))› [*51*,*51*,*51*] *50*) **where**

— Primitive Literals

NullLiteralT:
Γ ⊢$_E$ *NullLiteral* : *OclVoid*[*?*]
|*BooleanLiteralT*:
Γ ⊢$_E$ *BooleanLiteral c* : *Boolean*[*1*]
|*RealLiteralT*:

$\Gamma \vdash_E \mathit{RealLiteral}\ c : \mathit{Real}[1]$
|*IntegerLiteralT*:
$\Gamma \vdash_E \mathit{IntegerLiteral}\ c : \mathit{Integer}[1]$
|*UnlimitedNaturalLiteralT*:
$\Gamma \vdash_E \mathit{UnlimitedNaturalLiteral}\ c : \mathit{UnlimitedNatural}[1]$
|*StringLiteralT*:
$\Gamma \vdash_E \mathit{StringLiteral}\ c : \mathit{String}[1]$
|*EnumLiteralT*:
has-literal enum lit $\Longrightarrow$
$\Gamma \vdash_E \mathit{EnumLiteral}\ \mathit{enum}\ \mathit{lit} : (\mathit{Enum}\ \mathit{enum})[1]$

— Collection Literals

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

— We prohibit empty collection literals, because their type is unclear. We could use $\mathit{OclVoid}[1]$ element type for empty collections, but the typing rules will give wrong types for nested collections, because, for example, $\mathit{OclVoid}[1] \sqcup \mathit{Set}(\mathit{Integer}[1]) = \mathit{OclSuper}$

|*CollectionPartsSingletonT*:
$\Gamma \vdash_P x : \tau \Longrightarrow$
$\Gamma \vdash_C [x] : \tau$
|*CollectionPartsListT*:
$\Gamma \vdash_P x : \tau \Longrightarrow$
$\Gamma \vdash_C y \# xs : \sigma \Longrightarrow$
$\Gamma \vdash_C x \# y \# xs : \tau \sqcup \sigma$

|*CollectionPartItemT*:
$\Gamma \vdash_E a : \tau \Longrightarrow$
$\Gamma \vdash_P \mathit{CollectionItem}\ a : \tau$
|*CollectionPartRangeT*:
$\Gamma \vdash_E a : \tau \Longrightarrow$
$\Gamma \vdash_E b : \sigma \Longrightarrow$
$\tau = \mathit{UnlimitedNatural}[1] - \mathit{Integer}[1] \Longrightarrow$
$\sigma = \mathit{UnlimitedNatural}[1] - \mathit{Integer}[1] \Longrightarrow$
$\Gamma \vdash_P \mathit{CollectionRange}\ a\ b : \mathit{Integer}[1]$

— Tuple Literals
— We do not prohibit empty tuples, because it could be useful. *Tuple*() is a supertype of all other tuple types.

|*TupleLiteralNilT*:
  $\Gamma \vdash_E$ *TupleLiteral* [] : *Tuple fmempty*
|*TupleLiteralConsT*:
  $\Gamma \vdash_E$ *TupleLiteral elems* : *Tuple* $\xi \Longrightarrow$
  $\Gamma \vdash_E$ *tuple-element-expr el* : $\tau \Longrightarrow$
  tuple-element-type el = *Some* $\sigma \Longrightarrow$
  $\tau \leq \sigma \Longrightarrow$
  $\Gamma \vdash_E$ *TupleLiteral* (*el* # *elems*) : *Tuple* (ξ(*tuple-element-name el* $\mapsto_f$ σ))

— Misc Expressions

|*LetT*:
  $\Gamma \vdash_E$ *init* : $\sigma \Longrightarrow$
  $\sigma \leq \tau \Longrightarrow$
  $\Gamma(v \mapsto_f \tau) \vdash_E$ *body* : $\varrho \Longrightarrow$
  $\Gamma \vdash_E$ *Let v* (*Some* τ) *init body* : ϱ
|*VarT*:
  fmlookup Γ *v* = *Some* $\tau \Longrightarrow$
  $\Gamma \vdash_E$ *Var v* : τ
|*IfT*:
  $\Gamma \vdash_E$ *a* : *Boolean*[*1*] $\Longrightarrow$
  $\Gamma \vdash_E$ *b* : $\sigma \Longrightarrow$
  $\Gamma \vdash_E$ *c* : $\varrho \Longrightarrow$
  $\Gamma \vdash_E$ *If a b c* : $\sigma \sqcup \varrho$

— Call Expressions

|*MetaOperationCallT*:
  mataop-type τ *op* $\sigma \Longrightarrow$
  $\Gamma \vdash_E$ *MetaOperationCall* τ *op* : σ
|*StaticOperationCallT*:
  $\Gamma \vdash_L$ *params* : $\pi \Longrightarrow$
  static-operation τ *op* π *oper* $\Longrightarrow$
  $\Gamma \vdash_E$ *StaticOperationCall* τ *op params* : *oper-type oper*

|*TypeOperationCallT*:
  $\Gamma \vdash_E$ *a* : $\tau \Longrightarrow$
  typeop-type k op τ σ $\varrho \Longrightarrow$
  $\Gamma \vdash_E$ *TypeOperationCall a k op* σ : ϱ

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

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

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

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

|*OperationCallT*:
$\Gamma \vdash_E src : \tau \Longrightarrow$
$\Gamma \vdash_L params : \pi \Longrightarrow$
op-type op k τ π $\sigma \Longrightarrow$
$\Gamma \vdash_E$ *OperationCall src k op params* : σ

|*TupleElementCallT*:
$\Gamma \vdash_E src$: *Tuple* $\pi \Longrightarrow$
fmlookup π *elem* = *Some* $\tau \Longrightarrow$
$\Gamma \vdash_E$ *TupleElementCall src DotCall elem* : τ

— Iterator Expressions

|*IteratorT*:
$\Gamma \vdash_E src : \tau \Longrightarrow$
element-type τ $\sigma \Longrightarrow$
$\sigma \leq$ *its-ty* $\Longrightarrow$
$\Gamma +\!\!+_f$ *fmap-of-list* (*map* (λ*it*. (*it*, *its-ty*)) *its*) $\vdash_E$ *body* : $\varrho \Longrightarrow$
$\Gamma \vdash_I$ (*src*, *its*, (*Some its-ty*), *body*) : (τ, σ, ϱ)

|*IterateT*:
$\Gamma \vdash_I$ (*src*, *its*, *its-ty*, *Let res* (*Some res-t*) *res-init body*) : (τ, σ, ϱ) $\Longrightarrow$
$\varrho \leq$ *res-t* $\Longrightarrow$
$\Gamma \vdash_E$ *IterateCall src ArrowCall its its-ty res* (*Some res-t*) *res-init body* : ϱ

|*AnyIteratorT*:
$\Gamma \vdash_I$ (*src*, *its*, *its-ty*, *body*) : (τ, σ, ϱ) $\Longrightarrow$
length its $\leq$ 1 $\Longrightarrow$
$\varrho \leq$ *Boolean*[*?*] $\Longrightarrow$
$\Gamma \vdash_E$ *AnyIteratorCall src ArrowCall its its-ty body* : σ

|*ClosureIteratorT*:
$\Gamma \vdash_I$ (*src*, *its*, *its-ty*, *body*) : (τ, σ, ϱ) $\Longrightarrow$
length its $\leq$ 1 $\Longrightarrow$
to-single-type $\varrho \leq \sigma \Longrightarrow$
to-unique-collection τ $\upsilon \Longrightarrow$
$\Gamma \vdash_E$ *ClosureIteratorCall src ArrowCall its its-ty body* : υ

|*CollectIteratorT*:
$\Gamma \vdash_I$ (*src*, *its*, *its-ty*, *body*) : (τ, σ, ϱ) $\Longrightarrow$

$\mathit{length\ its} \leq 1 \Longrightarrow$
$\mathit{to\text{-}nonunique\text{-}collection}\ \tau\ \upsilon \Longrightarrow$
$\mathit{update\text{-}element\text{-}type}\ \upsilon\ (\mathit{to\text{-}single\text{-}type}\ \varrho)\ \varphi \Longrightarrow$
$\Gamma \vdash_E \mathit{CollectIteratorCall\ src\ ArrowCall\ its\ its\text{-}ty\ body} : \varphi$
|*CollectNestedIteratorT*:
$\Gamma \vdash_I (\mathit{src}, \mathit{its}, \mathit{its\text{-}ty}, \mathit{body}) : (\tau, \sigma, \varrho) \Longrightarrow$
$\mathit{length\ its} \leq 1 \Longrightarrow$
$\mathit{to\text{-}nonunique\text{-}collection}\ \tau\ \upsilon \Longrightarrow$
$\mathit{update\text{-}element\text{-}type}\ \upsilon\ \varrho\ \varphi \Longrightarrow$
$\Gamma \vdash_E \mathit{CollectNestedIteratorCall\ src\ ArrowCall\ its\ its\text{-}ty\ body} : \varphi$
|*ExistsIteratorT*:
$\Gamma \vdash_I (\mathit{src}, \mathit{its}, \mathit{its\text{-}ty}, \mathit{body}) : (\tau, \sigma, \varrho) \Longrightarrow$
$\varrho \leq \mathit{Boolean}[?] \Longrightarrow$
$\Gamma \vdash_E \mathit{ExistsIteratorCall\ src\ ArrowCall\ its\ its\text{-}ty\ body} : \varrho$
|*ForAllIteratorT*:
$\Gamma \vdash_I (\mathit{src}, \mathit{its}, \mathit{its\text{-}ty}, \mathit{body}) : (\tau, \sigma, \varrho) \Longrightarrow$
$\varrho \leq \mathit{Boolean}[?] \Longrightarrow$
$\Gamma \vdash_E \mathit{ForAllIteratorCall\ src\ ArrowCall\ its\ its\text{-}ty\ body} : \varrho$
|*OneIteratorT*:
$\Gamma \vdash_I (\mathit{src}, \mathit{its}, \mathit{its\text{-}ty}, \mathit{body}) : (\tau, \sigma, \varrho) \Longrightarrow$
$\mathit{length\ its} \leq 1 \Longrightarrow$
$\varrho \leq \mathit{Boolean}[?] \Longrightarrow$
$\Gamma \vdash_E \mathit{OneIteratorCall\ src\ ArrowCall\ its\ its\text{-}ty\ body} : \mathit{Boolean}[1]$
|*IsUniqueIteratorT*:
$\Gamma \vdash_I (\mathit{src}, \mathit{its}, \mathit{its\text{-}ty}, \mathit{body}) : (\tau, \sigma, \varrho) \Longrightarrow$
$\mathit{length\ its} \leq 1 \Longrightarrow$
$\Gamma \vdash_E \mathit{IsUniqueIteratorCall\ src\ ArrowCall\ its\ its\text{-}ty\ body} : \mathit{Boolean}[1]$
|*SelectIteratorT*:
$\Gamma \vdash_I (\mathit{src}, \mathit{its}, \mathit{its\text{-}ty}, \mathit{body}) : (\tau, \sigma, \varrho) \Longrightarrow$
$\mathit{length\ its} \leq 1 \Longrightarrow$
$\varrho \leq \mathit{Boolean}[?] \Longrightarrow$
$\Gamma \vdash_E \mathit{SelectIteratorCall\ src\ ArrowCall\ its\ its\text{-}ty\ body} : \tau$
|*RejectIteratorT*:
$\Gamma \vdash_I (\mathit{src}, \mathit{its}, \mathit{its\text{-}ty}, \mathit{body}) : (\tau, \sigma, \varrho) \Longrightarrow$
$\mathit{length\ its} \leq 1 \Longrightarrow$
$\varrho \leq \mathit{Boolean}[?] \Longrightarrow$
$\Gamma \vdash_E \mathit{RejectIteratorCall\ src\ ArrowCall\ its\ its\text{-}ty\ body} : \tau$
|*SortedByIteratorT*:
$\Gamma \vdash_I (\mathit{src}, \mathit{its}, \mathit{its\text{-}ty}, \mathit{body}) : (\tau, \sigma, \varrho) \Longrightarrow$
$\mathit{length\ its} \leq 1 \Longrightarrow$
$\mathit{to\text{-}ordered\text{-}collection}\ \tau\ \upsilon \Longrightarrow$
$\Gamma \vdash_E \mathit{SortedByIteratorCall\ src\ ArrowCall\ its\ its\text{-}ty\ body} : \upsilon$

— Expression Lists

|*ExprListNilT*:
$\Gamma \vdash_L [] : []$
|*ExprListConsT*:
$\Gamma \vdash_E \mathit{expr} : \tau \Longrightarrow$

$\Gamma \vdash_L exprs : \pi \Longrightarrow$
$\Gamma \vdash_L expr \# exprs : \tau \# \pi$

6.3 Elimination Rules

inductive-cases *NullLiteralTE* [*elim*]: $\Gamma \vdash_E NullLiteral : \tau$
inductive-cases *BooleanLiteralTE* [*elim*]: $\Gamma \vdash_E BooleanLiteral\ c : \tau$
inductive-cases *RealLiteralTE* [*elim*]: $\Gamma \vdash_E RealLiteral\ c : \tau$
inductive-cases *IntegerLiteralTE* [*elim*]: $\Gamma \vdash_E IntegerLiteral\ c : \tau$
inductive-cases *UnlimitedNaturalLiteralTE* [*elim*]: $\Gamma \vdash_E UnlimitedNaturalLiteral\ c : \tau$
inductive-cases *StringLiteralTE* [*elim*]: $\Gamma \vdash_E StringLiteral\ c : \tau$
inductive-cases *EnumLiteralTE* [*elim*]: $\Gamma \vdash_E EnumLiteral\ enm\ lit : \tau$
inductive-cases *CollectionLiteralTE* [*elim*]: $\Gamma \vdash_E CollectionLiteral\ k\ prts : \tau$
inductive-cases *TupleLiteralTE* [*elim*]: $\Gamma \vdash_E TupleLiteral\ elems : \tau$

inductive-cases *LetTE* [*elim*]: $\Gamma \vdash_E Let\ v\ \tau\ init\ body : \sigma$
inductive-cases *VarTE* [*elim*]: $\Gamma \vdash_E Var\ v : \tau$
inductive-cases *IfTE* [*elim*]: $\Gamma \vdash_E If\ a\ b\ c : \tau$

inductive-cases *MetaOperationCallTE* [*elim*]: $\Gamma \vdash_E MetaOperationCall\ \tau\ op : \sigma$

inductive-cases *StaticOperationCallTE* [*elim*]: $\Gamma \vdash_E StaticOperationCall\ \tau\ op\ as : \sigma$

inductive-cases *TypeOperationCallTE* [*elim*]: $\Gamma \vdash_E TypeOperationCall\ a\ k\ op\ \sigma : \tau$
inductive-cases *AttributeCallTE* [*elim*]: $\Gamma \vdash_E AttributeCall\ src\ k\ prop : \tau$
inductive-cases *AssociationEndCallTE* [*elim*]: $\Gamma \vdash_E AssociationEndCall\ src\ k\ role\ from : \tau$
inductive-cases *AssociationClassCallTE* [*elim*]: $\Gamma \vdash_E AssociationClassCall\ src\ k\ a\ from : \tau$
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*]: $\Gamma \vdash_E IterateCall\ src\ k\ its\ its\text{-}ty\ res\ res\text{-}t\ res\text{-}init\ body : \tau$
inductive-cases *AnyIteratorTE* [*elim*]: $\Gamma \vdash_E AnyIteratorCall\ src\ k\ its\ its\text{-}ty\ body : \tau$
inductive-cases *ClosureIteratorTE* [*elim*]: $\Gamma \vdash_E ClosureIteratorCall\ src\ k\ its\ its\text{-}ty\ body : \tau$
inductive-cases *CollectIteratorTE* [*elim*]: $\Gamma \vdash_E CollectIteratorCall\ src\ k\ its\ its\text{-}ty\ body : \tau$
inductive-cases *CollectNestedIteratorTE* [*elim*]: $\Gamma \vdash_E CollectNestedIteratorCall$

src k its its-ty body : τ
inductive-cases *ExistsIteratorTE* [*elim*]: $\Gamma \vdash_E$ *ExistsIteratorCall src k its its-ty body* : τ
inductive-cases *ForAllIteratorTE* [*elim*]: $\Gamma \vdash_E$ *ForAllIteratorCall src k its its-ty body* : τ
inductive-cases *OneIteratorTE* [*elim*]: $\Gamma \vdash_E$ *OneIteratorCall src k its its-ty body* : τ
inductive-cases *IsUniqueIteratorTE* [*elim*]: $\Gamma \vdash_E$ *IsUniqueIteratorCall src k its its-ty body* : τ
inductive-cases *SelectIteratorTE* [*elim*]: $\Gamma \vdash_E$ *SelectIteratorCall src k its its-ty body* : τ
inductive-cases *RejectIteratorTE* [*elim*]: $\Gamma \vdash_E$ *RejectIteratorCall src k its its-ty body* : τ
inductive-cases *SortedByIteratorTE* [*elim*]: $\Gamma \vdash_E$ *SortedByIteratorCall src k its its-ty body* : τ

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

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

6.4 Simplification Rules

inductive-simps *typing-alt-simps*:
$\Gamma \vdash_E$ *NullLiteral* : τ
$\Gamma \vdash_E$ *BooleanLiteral c* : τ
$\Gamma \vdash_E$ *RealLiteral c* : τ
$\Gamma \vdash_E$ *UnlimitedNaturalLiteral c* : τ
$\Gamma \vdash_E$ *IntegerLiteral c* : τ
$\Gamma \vdash_E$ *StringLiteral c* : τ
$\Gamma \vdash_E$ *EnumLiteral enm lit* : τ
$\Gamma \vdash_E$ *CollectionLiteral k prts* : τ
$\Gamma \vdash_E$ *TupleLiteral* [] : τ
$\Gamma \vdash_E$ *TupleLiteral* $(x \# xs)$: τ

$\Gamma \vdash_E$ *Let v* τ *init body* : σ
$\Gamma \vdash_E$ *Var v* : τ
$\Gamma \vdash_E$ *If a b c* : τ

$\Gamma \vdash_E$ *MetaOperationCall* τ *op* : σ
$\Gamma \vdash_E$ *StaticOperationCall* τ *op as* : σ

$\Gamma \vdash_E$ *TypeOperationCall a k op* σ : τ
$\Gamma \vdash_E$ *AttributeCall src k prop* : τ
$\Gamma \vdash_E$ *AssociationEndCall src k role from* : τ
$\Gamma \vdash_E$ *AssociationClassCall src k a from* : τ

$\Gamma \vdash_E \mathit{AssociationClassEndCall\ src\ k\ role} : \tau$
$\Gamma \vdash_E \mathit{OperationCall\ src\ k\ op\ params} : \tau$
$\Gamma \vdash_E \mathit{TupleElementCall\ src\ k\ elem} : \tau$

$\Gamma \vdash_I (\mathit{src}, \mathit{its}, \mathit{body}) : \mathit{ys}$
$\Gamma \vdash_E \mathit{IterateCall\ src\ k\ its\ its\text{-}ty\ res\ res\text{-}t\ res\text{-}init\ body} : \tau$
$\Gamma \vdash_E \mathit{AnyIteratorCall\ src\ k\ its\ its\text{-}ty\ body} : \tau$
$\Gamma \vdash_E \mathit{ClosureIteratorCall\ src\ k\ its\ its\text{-}ty\ body} : \tau$
$\Gamma \vdash_E \mathit{CollectIteratorCall\ src\ k\ its\ its\text{-}ty\ body} : \tau$
$\Gamma \vdash_E \mathit{CollectNestedIteratorCall\ src\ k\ its\ its\text{-}ty\ body} : \tau$
$\Gamma \vdash_E \mathit{ExistsIteratorCall\ src\ k\ its\ its\text{-}ty\ body} : \tau$
$\Gamma \vdash_E \mathit{ForAllIteratorCall\ src\ k\ its\ its\text{-}ty\ body} : \tau$
$\Gamma \vdash_E \mathit{OneIteratorCall\ src\ k\ its\ its\text{-}ty\ body} : \tau$
$\Gamma \vdash_E \mathit{IsUniqueIteratorCall\ src\ k\ its\ its\text{-}ty\ body} : \tau$
$\Gamma \vdash_E \mathit{SelectIteratorCall\ src\ k\ its\ its\text{-}ty\ body} : \tau$
$\Gamma \vdash_E \mathit{RejectIteratorCall\ src\ k\ its\ its\text{-}ty\ body} : \tau$
$\Gamma \vdash_E \mathit{SortedByIteratorCall\ src\ k\ its\ its\text{-}ty\ body} : \tau$

$\Gamma \vdash_C [x] : \tau$
$\Gamma \vdash_C x \# y \# xs : \tau$

$\Gamma \vdash_P \mathit{CollectionItem\ a} : \tau$
$\Gamma \vdash_P \mathit{CollectionRange\ a\ b} : \tau$

$\Gamma \vdash_L [] : \pi$
$\Gamma \vdash_L x \# xs : \pi$

6.5 Determinism

lemma
typing-det:
$\Gamma \vdash_E \mathit{expr} : \tau \Longrightarrow$
$\Gamma \vdash_E \mathit{expr} : \sigma \Longrightarrow \tau = \sigma$ **and**
collection-parts-typing-det:
$\Gamma \vdash_C \mathit{prts} : \tau \Longrightarrow$
$\Gamma \vdash_C \mathit{prts} : \sigma \Longrightarrow \tau = \sigma$ **and**
collection-part-typing-det:
$\Gamma \vdash_P \mathit{prt} : \tau \Longrightarrow$
$\Gamma \vdash_P \mathit{prt} : \sigma \Longrightarrow \tau = \sigma$ **and**
iterator-typing-det:
$\Gamma \vdash_I (\mathit{src}, \mathit{its}, \mathit{body}) : \mathit{xs} \Longrightarrow$
$\Gamma \vdash_I (\mathit{src}, \mathit{its}, \mathit{body}) : \mathit{ys} \Longrightarrow \mathit{xs} = \mathit{ys}$ **and**
expr-list-typing-det:
$\Gamma \vdash_L \mathit{exprs} : \pi \Longrightarrow$
$\Gamma \vdash_L \mathit{exprs} : \xi \Longrightarrow \pi = \xi$
⟨*proof*⟩

6.6 Code Setup

code-pred *op-type* ⟨*proof*⟩

code-pred (*modes*:
$i \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow bool$) *iterator-typing* ⟨*proof*⟩

end

Chapter 7

Normalization

theory *OCL-Normalization*
imports *OCL-Typing*
begin

7.1 Normalization Rules

The following expression normalization rules includes two kinds of an abstract syntax tree transformations:

- determination of implicit types of variables, iterators, and tuple elements,
- unfolding of navigation shorthands and safe navigation operators, described in Table 7.1.

The following variables are used in the table:

- `x` is a non-nullable value,
- `n` is a nullable value,
- `xs` is a collection of non-nullable values,
- `ns` is a collection of nullable values.

Please take a note that name resolution of variables, types, attributes, and associations is out of scope of this section. It should be done on a previous phase during transformation of a concrete syntax tree to an abstract syntax tree.

fun *string-of-nat* :: *nat* ⇒ *string* **where**
string-of-nat n = (*if n* < *10 then* [*char-of* (*48* + *n*)]
else string-of-nat (*n div 10*) @ [*char-of* (*48* + (*n mod 10*))])

definition *new-vname* ≡ *String.implode* ∘ *string-of-nat* ∘ *fcard* ∘ *fmdom*

Table 7.1: Expression Normalization Rules

Orig. expr.	Normalized expression
`x.op()`	`x.op()`
`n.op()`	`n.op()`*
`x?.op()`	—
`n?.op()`	`if n <> null then n.oclAsType(T[1]).op() else null endif`**
`x->op()`	`x.oclAsSet()->op()`
`n->op()`	`n.oclAsSet()->op()`
`x?->op()`	—
`n?->op()`	—
`xs.op()`	`xs->collect(x | x.op())`
`ns.op()`	`ns->collect(n | n.op())`*
`xs?.op()`	—
`ns?.op()`	`ns->selectByKind(T[1])->collect(x | x.op())`
`xs->op()`	`xs->op()`
`ns->op()`	`ns->op()`
`xs?->op()`	—
`ns?->op()`	`ns->selectByKind(T[1])->op()`

* 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* :: *ocl-object-model*) *type env* ⇒ '*a expr* ⇒ '*a expr* ⇒ *bool*
(‹- ⊢ - ⇛/ -› [*51*,*51*,*51*] *50*) **and**
normalize-call (‹- ⊢$_C$ - ⇛/ -› [*51*,*51*,*51*] *50*) **and**
normalize-expr-list (‹- ⊢$_L$ - ⇛/ -› [*51*,*51*,*51*] *50*)
where
LiteralN:
Γ ⊢ *Literal a* ⇛ *Literal a*
|*ExplicitlyTypedLetN*:
Γ ⊢ $init_1$ ⇛ $init_2$ ⟹
Γ(v ↦$_f$ τ) ⊢ $body_1$ ⇛ $body_2$ ⟹
Γ ⊢ *Let* v (*Some* τ) $init_1$ $body_1$ ⇛ *Let* v (*Some* τ) $init_2$ $body_2$
|*ImplicitlyTypedLetN*:
Γ ⊢ $init_1$ ⇛ $init_2$ ⟹
Γ ⊢$_E$ $init_2$: τ ⟹
Γ(v ↦$_f$ τ) ⊢ $body_1$ ⇛ $body_2$ ⟹

$\Gamma \vdash \mathit{Let}\ v\ \mathit{None}\ \mathit{init}_1\ \mathit{body}_1 \Rrightarrow \mathit{Let}\ v\ (\mathit{Some}\ \tau)\ \mathit{init}_2\ \mathit{body}_2$
|*VarN*:
$\Gamma \vdash \mathit{Var}\ v \Rrightarrow \mathit{Var}\ v$
|*IfN*:
$\Gamma \vdash a_1 \Rrightarrow a_2 \Longrightarrow$
$\Gamma \vdash b_1 \Rrightarrow b_2 \Longrightarrow$
$\Gamma \vdash c_1 \Rrightarrow c_2 \Longrightarrow$
$\Gamma \vdash \mathit{If}\ a_1\ b_1\ c_1 \Rrightarrow \mathit{If}\ a_2\ b_2\ c_2$

|*MetaOperationCallN*:
$\Gamma \vdash \mathit{MetaOperationCall}\ \tau\ \mathit{op} \Rrightarrow \mathit{MetaOperationCall}\ \tau\ \mathit{op}$
|*StaticOperationCallN*:
$\Gamma \vdash_L \mathit{params}_1 \Rrightarrow \mathit{params}_2 \Longrightarrow$
$\Gamma \vdash \mathit{StaticOperationCall}\ \tau\ \mathit{op}\ \mathit{params}_1 \Rrightarrow \mathit{StaticOperationCall}\ \tau\ \mathit{op}\ \mathit{params}_2$

|*OclAnyDotCallN*:
$\Gamma \vdash \mathit{src}_1 \Rrightarrow \mathit{src}_2 \Longrightarrow$
$\Gamma \vdash_E \mathit{src}_2 : \tau \Longrightarrow$
$\tau \leq \mathit{OclAny}[?] \vee \tau \leq \mathit{Tuple}\ \mathit{fmempty} \Longrightarrow$
$(\Gamma, \tau) \vdash_C \mathit{call}_1 \Rrightarrow \mathit{call}_2 \Longrightarrow$
$\Gamma \vdash \mathit{Call}\ \mathit{src}_1\ \mathit{DotCall}\ \mathit{call}_1 \Rrightarrow \mathit{Call}\ \mathit{src}_2\ \mathit{DotCall}\ \mathit{call}_2$
|*OclAnySafeDotCallN*:
$\Gamma \vdash \mathit{src}_1 \Rrightarrow \mathit{src}_2 \Longrightarrow$
$\Gamma \vdash_E \mathit{src}_2 : \tau \Longrightarrow$
$\mathit{OclVoid}[?] \leq \tau \Longrightarrow$
$(\Gamma, \mathit{to\text{-}required\text{-}type}\ \tau) \vdash_C \mathit{call}_1 \Rrightarrow \mathit{call}_2 \Longrightarrow$
$\mathit{src}_3 = \mathit{TypeOperationCall}\ \mathit{src}_2\ \mathit{DotCall}\ \mathit{OclAsTypeOp}\ (\mathit{to\text{-}required\text{-}type}\ \tau) \Longrightarrow$
$\Gamma \vdash \mathit{Call}\ \mathit{src}_1\ \mathit{SafeDotCall}\ \mathit{call}_1 \Rrightarrow$
$\quad \mathit{If}\ (\mathit{OperationCall}\ \mathit{src}_2\ \mathit{DotCall}\ \mathit{NotEqualOp}\ [\mathit{NullLiteral}])$
$\quad\quad (\mathit{Call}\ \mathit{src}_3\ \mathit{DotCall}\ \mathit{call}_2)$
$\quad\quad \mathit{NullLiteral}$
|*OclAnyArrowCallN*:
$\Gamma \vdash \mathit{src}_1 \Rrightarrow \mathit{src}_2 \Longrightarrow$
$\Gamma \vdash_E \mathit{src}_2 : \tau \Longrightarrow$
$\tau \leq \mathit{OclAny}[?] \vee \tau \leq \mathit{Tuple}\ \mathit{fmempty} \Longrightarrow$
$\mathit{src}_3 = \mathit{OperationCall}\ \mathit{src}_2\ \mathit{DotCall}\ \mathit{OclAsSetOp}\ [] \Longrightarrow$
$\Gamma \vdash_E \mathit{src}_3 : \sigma \Longrightarrow$
$(\Gamma, \sigma) \vdash_C \mathit{call}_1 \Rrightarrow \mathit{call}_2 \Longrightarrow$
$\Gamma \vdash \mathit{Call}\ \mathit{src}_1\ \mathit{ArrowCall}\ \mathit{call}_1 \Rrightarrow \mathit{Call}\ \mathit{src}_3\ \mathit{ArrowCall}\ \mathit{call}_2$

|*CollectionArrowCallN*:
$\Gamma \vdash \mathit{src}_1 \Rrightarrow \mathit{src}_2 \Longrightarrow$
$\Gamma \vdash_E \mathit{src}_2 : \tau \Longrightarrow$
$\mathit{element\text{-}type}\ \tau\ \text{-} \Longrightarrow$
$(\Gamma, \tau) \vdash_C \mathit{call}_1 \Rrightarrow \mathit{call}_2 \Longrightarrow$
$\Gamma \vdash \mathit{Call}\ \mathit{src}_1\ \mathit{ArrowCall}\ \mathit{call}_1 \Rrightarrow \mathit{Call}\ \mathit{src}_2\ \mathit{ArrowCall}\ \mathit{call}_2$
|*CollectionSafeArrowCallN*:
$\Gamma \vdash \mathit{src}_1 \Rrightarrow \mathit{src}_2 \Longrightarrow$
$\Gamma \vdash_E \mathit{src}_2 : \tau \Longrightarrow$

$$
\begin{array}{l}
\quad \mathit{element\text{-}type}\ \tau\ \sigma \Longrightarrow \\
\quad \mathit{OclVoid}[?] \leq \sigma \Longrightarrow \\
\quad \mathit{src}_3 = \mathit{TypeOperationCall}\ \mathit{src}_2\ \mathit{ArrowCall}\ \mathit{SelectByKindOp} \\
\qquad\quad (\mathit{to\text{-}required\text{-}type}\ \sigma) \Longrightarrow \\
\quad \Gamma \vdash_E \mathit{src}_3 : \varrho \Longrightarrow \\
\quad (\Gamma,\ \varrho) \vdash_C \mathit{call}_1 \Rrightarrow \mathit{call}_2 \Longrightarrow \\
\quad \Gamma \vdash \mathit{Call}\ \mathit{src}_1\ \mathit{SafeArrowCall}\ \mathit{call}_1 \Rrightarrow \mathit{Call}\ \mathit{src}_3\ \mathit{ArrowCall}\ \mathit{call}_2 \\
|\mathit{CollectionDotCallN}: \\
\quad \Gamma \vdash \mathit{src}_1 \Rrightarrow \mathit{src}_2 \Longrightarrow \\
\quad \Gamma \vdash_E \mathit{src}_2 : \tau \Longrightarrow \\
\quad \mathit{element\text{-}type}\ \tau\ \sigma \Longrightarrow \\
\quad (\Gamma,\ \sigma) \vdash_C \mathit{call}_1 \Rrightarrow \mathit{call}_2 \Longrightarrow \\
\quad \mathit{it} = \mathit{new\text{-}vname}\ \Gamma \Longrightarrow \\
\quad \Gamma \vdash \mathit{Call}\ \mathit{src}_1\ \mathit{DotCall}\ \mathit{call}_1 \Rrightarrow \\
\quad\ \mathit{CollectIteratorCall}\ \mathit{src}_2\ \mathit{ArrowCall}\ [\mathit{it}]\ (\mathit{Some}\ \sigma)\ (\mathit{Call}\ (\mathit{Var}\ \mathit{it})\ \mathit{DotCall}\ \mathit{call}_2) \\
|\mathit{CollectionSafeDotCallN}: \\
\quad \Gamma \vdash \mathit{src}_1 \Rrightarrow \mathit{src}_2 \Longrightarrow \\
\quad \Gamma \vdash_E \mathit{src}_2 : \tau \Longrightarrow \\
\quad \mathit{element\text{-}type}\ \tau\ \sigma \Longrightarrow \\
\quad \mathit{OclVoid}[?] \leq \sigma \Longrightarrow \\
\quad \varrho = \mathit{to\text{-}required\text{-}type}\ \sigma \Longrightarrow \\
\quad \mathit{src}_3 = \mathit{TypeOperationCall}\ \mathit{src}_2\ \mathit{ArrowCall}\ \mathit{SelectByKindOp}\ \varrho \Longrightarrow \\
\quad (\Gamma,\ \varrho) \vdash_C \mathit{call}_1 \Rrightarrow \mathit{call}_2 \Longrightarrow \\
\quad \mathit{it} = \mathit{new\text{-}vname}\ \Gamma \Longrightarrow \\
\quad \Gamma \vdash \mathit{Call}\ \mathit{src}_1\ \mathit{SafeDotCall}\ \mathit{call}_1 \Rrightarrow \\
\quad\ \mathit{CollectIteratorCall}\ \mathit{src}_3\ \mathit{ArrowCall}\ [\mathit{it}]\ (\mathit{Some}\ \varrho)\ (\mathit{Call}\ (\mathit{Var}\ \mathit{it})\ \mathit{DotCall}\ \mathit{call}_2) \\
\\
|\mathit{TypeOperationN}: \\
\quad (\Gamma,\ \tau) \vdash_C \mathit{TypeOperation}\ \mathit{op}\ \mathit{ty} \Rrightarrow \mathit{TypeOperation}\ \mathit{op}\ \mathit{ty} \\
|\mathit{AttributeN}: \\
\quad (\Gamma,\ \tau) \vdash_C \mathit{Attribute}\ \mathit{attr} \Rrightarrow \mathit{Attribute}\ \mathit{attr} \\
|\mathit{AssociationEndN}: \\
\quad (\Gamma,\ \tau) \vdash_C \mathit{AssociationEnd}\ \mathit{role}\ \mathit{from} \Rrightarrow \mathit{AssociationEnd}\ \mathit{role}\ \mathit{from} \\
|\mathit{AssociationClassN}: \\
\quad (\Gamma,\ \tau) \vdash_C \mathit{AssociationClass}\ \mathcal{A}\ \mathit{from} \Rrightarrow \mathit{AssociationClass}\ \mathcal{A}\ \mathit{from} \\
|\mathit{AssociationClassEndN}: \\
\quad (\Gamma,\ \tau) \vdash_C \mathit{AssociationClassEnd}\ \mathit{role} \Rrightarrow \mathit{AssociationClassEnd}\ \mathit{role} \\
|\mathit{OperationN}: \\
\quad \Gamma \vdash_L \mathit{params}_1 \Rrightarrow \mathit{params}_2 \Longrightarrow \\
\quad (\Gamma,\ \tau) \vdash_C \mathit{Operation}\ \mathit{op}\ \mathit{params}_1 \Rrightarrow \mathit{Operation}\ \mathit{op}\ \mathit{params}_2 \\
|\mathit{TupleElementN}: \\
\quad (\Gamma,\ \tau) \vdash_C \mathit{TupleElement}\ \mathit{elem} \Rrightarrow \mathit{TupleElement}\ \mathit{elem} \\
\\
|\mathit{ExplicitlyTypedIterateN}: \\
\quad \Gamma \vdash \mathit{res\text{-}init}_1 \Rrightarrow \mathit{res\text{-}init}_2 \Longrightarrow \\
\quad \Gamma +\!\!+_f \mathit{fmap\text{-}of\text{-}list}\ (\mathit{map}\ (\lambda \mathit{it}.\ (\mathit{it},\ \sigma))\ \mathit{its}) \vdash \\
\qquad \mathit{Let}\ \mathit{res}\ \mathit{res\text{-}t}_1\ \mathit{res\text{-}init}_1\ \mathit{body}_1 \Rrightarrow \mathit{Let}\ \mathit{res}\ \mathit{res\text{-}t}_2\ \mathit{res\text{-}init}_2\ \mathit{body}_2 \Longrightarrow \\
\quad (\Gamma,\ \tau) \vdash_C \mathit{Iterate}\ \mathit{its}\ (\mathit{Some}\ \sigma)\ \mathit{res}\ \mathit{res\text{-}t}_1\ \mathit{res\text{-}init}_1\ \mathit{body}_1 \Rrightarrow \\
\qquad \mathit{Iterate}\ \mathit{its}\ (\mathit{Some}\ \sigma)\ \mathit{res}\ \mathit{res\text{-}t}_2\ \mathit{res\text{-}init}_2\ \mathit{body}_2
\end{array}
$$

|*ImplicitlyTypedIterateN*:
element-type τ σ $\Longrightarrow$
$\Gamma \vdash$ *res-init*$_1$ $\Rrightarrow$ *res-init*$_2$ $\Longrightarrow$
Γ $+\!\!+_f$ *fmap-of-list* (*map* (λ*it*. (*it*, σ)) *its*) $\vdash$
Let res res-t$_1$ *res-init*$_1$ *body*$_1$ $\Rrightarrow$ *Let res res-t*$_2$ *res-init*$_2$ *body*$_2$ $\Longrightarrow$
(Γ, τ) $\vdash_C$ *Iterate its None res res-t*$_1$ *res-init*$_1$ *body*$_1$ $\Rrightarrow$
Iterate its (*Some* σ) *res res-t*$_2$ *res-init*$_2$ *body*$_2$

|*ExplicitlyTypedIteratorN*:
Γ $+\!\!+_f$ *fmap-of-list* (*map* (λ*it*. (*it*, σ)) *its*) $\vdash$ *body*$_1$ $\Rrightarrow$ *body*$_2$ $\Longrightarrow$
(Γ, τ) $\vdash_C$ *Iterator iter its* (*Some* σ) *body*$_1$ $\Rrightarrow$ *Iterator iter its* (*Some* σ) *body*$_2$
|*ImplicitlyTypedIteratorN*:
element-type τ σ $\Longrightarrow$
Γ $+\!\!+_f$ *fmap-of-list* (*map* (λ*it*. (*it*, σ)) *its*) $\vdash$ *body*$_1$ $\Rrightarrow$ *body*$_2$ $\Longrightarrow$
(Γ, τ) $\vdash_C$ *Iterator iter its None body*$_1$ $\Rrightarrow$ *Iterator iter its* (*Some* σ) *body*$_2$

|*ExprListNilN*:
$\Gamma \vdash_L [] \Rrightarrow []$
|*ExprListConsN*:
$\Gamma \vdash x \Rrightarrow y \Longrightarrow$
$\Gamma \vdash_L xs \Rrightarrow ys \Longrightarrow$
$\Gamma \vdash_L x \# xs \Rrightarrow y \# ys$

7.2 Elimination Rules

inductive-cases *LiteralNE* [*elim*]: $\Gamma \vdash$ *Literal a* $\Rrightarrow$ *b*
inductive-cases *LetNE* [*elim*]: $\Gamma \vdash$ *Let v t init body* $\Rrightarrow$ *b*
inductive-cases *VarNE* [*elim*]: $\Gamma \vdash$ *Var v* $\Rrightarrow$ *b*
inductive-cases *IfNE* [*elim*]: $\Gamma \vdash$ *If a b c* $\Rrightarrow$ *d*

inductive-cases *MetaOperationCallNE* [*elim*]: $\Gamma \vdash$ *MetaOperationCall* τ *op* $\Rrightarrow$ *b*

inductive-cases *StaticOperationCallNE* [*elim*]: $\Gamma \vdash$ *StaticOperationCall* τ *op as* $\Rrightarrow$ *b*
inductive-cases *DotCallNE* [*elim*]: $\Gamma \vdash$ *Call src DotCall call* $\Rrightarrow$ *b*
inductive-cases *SafeDotCallNE* [*elim*]: $\Gamma \vdash$ *Call src SafeDotCall call* $\Rrightarrow$ *b*
inductive-cases *ArrowCallNE* [*elim*]: $\Gamma \vdash$ *Call src ArrowCall call* $\Rrightarrow$ *b*
inductive-cases *SafeArrowCallNE* [*elim*]: $\Gamma \vdash$ *Call src SafeArrowCall call* $\Rrightarrow$ *b*

inductive-cases *CallNE* [*elim*]: (Γ, τ) $\vdash_C$ *call* $\Rrightarrow$ *b*
inductive-cases *OperationCallNE* [*elim*]: (Γ, τ) $\vdash_C$ *Operation op as* $\Rrightarrow$ *call*
inductive-cases *IterateCallNE* [*elim*]: (Γ, τ) $\vdash_C$ *Iterate its its-ty res res-t res-init body* $\Rrightarrow$ *call*
inductive-cases *IteratorCallNE* [*elim*]: (Γ, τ) $\vdash_C$ *Iterator iter its its-ty body* $\Rrightarrow$ *call*

inductive-cases *ExprListNE* [*elim*]: $\Gamma \vdash_L$ *xs* $\Rrightarrow$ *ys*

7.3 Simplification Rules

inductive-simps *normalize-alt-simps*:
$\Gamma \vdash \mathit{Literal}\ a \Rrightarrow b$
$\Gamma \vdash \mathit{Let}\ v\ t\ \mathit{init}\ \mathit{body} \Rrightarrow b$
$\Gamma \vdash \mathit{Var}\ v \Rrightarrow b$
$\Gamma \vdash \mathit{If}\ a\ b\ c \Rrightarrow d$

$\Gamma \vdash \mathit{MetaOperationCall}\ \tau\ \mathit{op} \Rrightarrow b$
$\Gamma \vdash \mathit{StaticOperationCall}\ \tau\ \mathit{op}\ \mathit{as} \Rrightarrow b$
$\Gamma \vdash \mathit{Call}\ \mathit{src}\ \mathit{DotCall}\ \mathit{call} \Rrightarrow b$
$\Gamma \vdash \mathit{Call}\ \mathit{src}\ \mathit{SafeDotCall}\ \mathit{call} \Rrightarrow b$
$\Gamma \vdash \mathit{Call}\ \mathit{src}\ \mathit{ArrowCall}\ \mathit{call} \Rrightarrow b$
$\Gamma \vdash \mathit{Call}\ \mathit{src}\ \mathit{SafeArrowCall}\ \mathit{call} \Rrightarrow b$

$(\Gamma, \tau) \vdash_C \mathit{call} \Rrightarrow b$
$(\Gamma, \tau) \vdash_C \mathit{Operation}\ \mathit{op}\ \mathit{as} \Rrightarrow \mathit{call}$
$(\Gamma, \tau) \vdash_C \mathit{Iterate}\ \mathit{its}\ \mathit{its\text{-}ty}\ \mathit{res}\ \mathit{res\text{-}t}\ \mathit{res\text{-}init}\ \mathit{body} \Rrightarrow \mathit{call}$
$(\Gamma, \tau) \vdash_C \mathit{Iterator}\ \mathit{iter}\ \mathit{its}\ \mathit{its\text{-}ty}\ \mathit{body} \Rrightarrow \mathit{call}$

$\Gamma \vdash_L [] \Rrightarrow \mathit{ys}$
$\Gamma \vdash_L x \# \mathit{xs} \Rrightarrow \mathit{ys}$

7.4 Determinism

lemma *any-has-not-element-type*:
$\mathit{element\text{-}type}\ \tau\ \sigma \Longrightarrow \tau \leq \mathit{OclAny}[?] \vee \tau \leq \mathit{Tuple}\ \mathit{fmempty} \Longrightarrow \mathit{False}$
⟨*proof*⟩

lemma *any-has-not-element-type′*:
$\mathit{element\text{-}type}\ \tau\ \sigma \Longrightarrow \mathit{OclVoid}[?] \leq \tau \Longrightarrow \mathit{False}$
⟨*proof*⟩

lemma
normalize-det:
$\Gamma \vdash \mathit{expr} \Rrightarrow \mathit{expr}_1 \Longrightarrow$
$\Gamma \vdash \mathit{expr} \Rrightarrow \mathit{expr}_2 \Longrightarrow \mathit{expr}_1 = \mathit{expr}_2$ **and**
normalize-call-det:
$\Gamma\text{-}\tau \vdash_C \mathit{call} \Rrightarrow \mathit{call}_1 \Longrightarrow$
$\Gamma\text{-}\tau \vdash_C \mathit{call} \Rrightarrow \mathit{call}_2 \Longrightarrow \mathit{call}_1 = \mathit{call}_2$ **and**
normalize-expr-list-det:
$\Gamma \vdash_L \mathit{xs} \Rrightarrow \mathit{ys} \Longrightarrow$
$\Gamma \vdash_L \mathit{xs} \Rrightarrow \mathit{zs} \Longrightarrow \mathit{ys} = \mathit{zs}$
for Γ :: $('a$ :: *ocl-object-model*$)$ *type env*
and $\Gamma\text{-}\tau$ :: $('a$ :: *ocl-object-model*$)$ *type env* $\times$ $'a$ *type*
⟨*proof*⟩

7.5 Normalized Expressions Typing

Here is the final typing rules.

inductive *nf-typing* (‹(*1-*/ ⊢/ (- :/ -))› [*51*,*51*,*51*] *50*) **where**
$\Gamma \vdash expr \Rrightarrow expr_N \Longrightarrow$
$\Gamma \vdash_E expr_N : \tau \Longrightarrow$
$\Gamma \vdash expr : \tau$

lemma *nf-typing-det*:
$\Gamma \vdash expr : \tau \Longrightarrow$
$\Gamma \vdash expr : \sigma \Longrightarrow \tau = \sigma$
⟨*proof*⟩

7.6 Code Setup

code-pred *normalize* ⟨*proof*⟩

code-pred *nf-typing* ⟨*proof*⟩

definition *check-type* Γ *expr* τ ≡
Predicate.eval (*nf-typing-i-i-i* Γ *expr* τ) ()

definition *synthesize-type* Γ *expr* ≡
Predicate.singleton (λ-. *OclInvalid*)
(*Predicate.map errorable* (*nf-typing-i-i-o* Γ *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**
$c \neq Object \Longrightarrow$
subclass1 c Object
| *subclass1 Employee Person*
| *subclass1 Customer Person*

instantiation *classes1* :: *semilattice-sup*
begin

definition $(<) \equiv$ *subclass1*
definition $(\leq) \equiv$ *subclass1*$^{==}$

fun *sup-classes1* **where**
Object $\sqcup$ - = *Object*
| *Person* $\sqcup$ *c* = (*if c* = *Person* $\vee$ *c* = *Employee* $\vee$ *c* = *Customer*
then Person else Object)
| *Employee* $\sqcup$ *c* = (*if c* = *Employee then Employee else*
if c = *Person* $\vee$ *c* = *Customer then Person else Object*)
| *Customer* $\sqcup$ *c* = (*if c* = *Customer then Customer else*
if c = *Person* $\vee$ *c* = *Employee then Person else Object*)
| *Project* $\sqcup$ *c* = (*if c* = *Project then Project else Object*)
| *Task* $\sqcup$ *c* = (*if c* = *Task then Task else Object*)
| *Sprint* $\sqcup$ *c* = (*if c* = *Sprint then Sprint else Object*)

lemma *less-le-not-le-classes1*:

$c < d \longleftrightarrow c \leq d \wedge \neg\ d \leq c$
for $c\ d$ :: *classes1*
⟨*proof*⟩

lemma *order-refl-classes1*:
$c \leq c$
for c :: *classes1*
⟨*proof*⟩

lemma *order-trans-classes1*:
$c \leq d \Longrightarrow d \leq e \Longrightarrow c \leq e$
for $c\ d\ e$ :: *classes1*
⟨*proof*⟩

lemma *antisym-classes1*:
$c \leq d \Longrightarrow d \leq c \Longrightarrow c = d$
for $c\ d$ :: *classes1*
⟨*proof*⟩

lemma *sup-ge1-classes1*:
$c \leq c \sqcup d$
for $c\ d$ :: *classes1*
⟨*proof*⟩

lemma *sup-ge2-classes1*:
$d \leq c \sqcup d$
for $c\ d$ :: *classes1*
⟨*proof*⟩

lemma *sup-least-classes1*:
$c \leq e \Longrightarrow d \leq e \Longrightarrow c \sqcup d \leq e$
for $c\ d\ e$ :: *classes1*
⟨*proof*⟩

instance
⟨*proof*⟩

end

code-pred *subclass1* ⟨*proof*⟩

fun *subclass1-fun* **where**
subclass1-fun Object $\mathcal{C}$ = *False*
| *subclass1-fun Person* $\mathcal{C}$ = ($\mathcal{C}$ = *Object*)
| *subclass1-fun Employee* $\mathcal{C}$ = ($\mathcal{C}$ = *Object* ∨ $\mathcal{C}$ = *Person*)
| *subclass1-fun Customer* $\mathcal{C}$ = ($\mathcal{C}$ = *Object* ∨ $\mathcal{C}$ = *Person*)
| *subclass1-fun Project* $\mathcal{C}$ = ($\mathcal{C}$ = *Object*)
| *subclass1-fun Task* $\mathcal{C}$ = ($\mathcal{C}$ = *Object*)
| *subclass1-fun Sprint* $\mathcal{C}$ = ($\mathcal{C}$ = *Object*)

lemma *less-classes1-code* [*code*]:
$(<) = \mathit{subclass1\text{-}fun}$
⟨*proof*⟩

lemma *less-eq-classes1-code* [*code*]:
$(\leq) = (\lambda x\ y.\ \mathit{subclass1\text{-}fun}\ x\ y \vee x = y)$
⟨*proof*⟩

8.2 Object Model

abbreviation $\Gamma_0 \equiv \mathit{fmempty} :: \mathit{classes1\ type\ env}$
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* ≡ [
STR ''*ProjectManager*'' ↦$_f$ [
STR ''*projects*'' ↦$_f$ (*Project*, *0*::*nat*, ∞::*enat*, *False*, *True*),
STR ''*manager*'' ↦$_f$ (*Employee*, *1*, *1*, *False*, *False*)],
STR ''*ProjectMember*'' ↦$_f$ [
STR ''*member-of*'' ↦$_f$ (*Project*, *0*, ∞, *False*, *False*),
STR ''*members*'' ↦$_f$ (*Employee*, *1*, *20*, *True*, *True*)],
STR ''*ManagerEmployee*'' ↦$_f$ [
STR ''*line-manager*'' ↦$_f$ (*Employee*, *0*, *1*, *False*, *False*),
STR ''*project-manager*'' ↦$_f$ (*Employee*, *0*, ∞, *False*, *False*),
STR ''*employees*'' ↦$_f$ (*Employee*, *3*, *7*, *False*, *False*)],
STR ''*ProjectCustomer*'' ↦$_f$ [
STR ''*projects*'' ↦$_f$ (*Project*, *0*, ∞, *False*, *True*),
STR ''*customer*'' ↦$_f$ (*Customer*, *1*, *1*, *False*, *False*)],

STR ''ProjectTask'' ↦$_f$ [
STR ''project'' ↦$_f$ (Project, 1, 1, False, False),
STR ''tasks'' ↦$_f$ (Task, 0, ∞, True, True)],
STR ''SprintTaskAssignee'' ↦$_f$ [
STR ''sprint'' ↦$_f$ (Sprint, 0, 10, False, True),
STR ''tasks'' ↦$_f$ (Task, 0, 5, False, True),
STR ''assignee'' ↦$_f$ (Employee, 0, 1, False, False)]]

definition *associations-classes1 ≡ assocs*

definition *association-classes-classes1 ≡ fmempty :: classes1 ⇀$_f$ 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 ≡ [*
(STR ''membersCount'', Project[1], [], Integer[1], False,
Some (OperationCall
(AssociationEndCall (Var STR ''self'') DotCall None STR ''members'')
ArrowCall CollectionSizeOp [])),
(STR ''membersByName'', Project[1], [(STR ''mn'', String[1], In)],
Set Employee[1], False,
Some (SelectIteratorCall
(AssociationEndCall (Var STR ''self'') DotCall None STR ''members'')
ArrowCall [STR ''member''] None
(OperationCall
(AttributeCall (Var STR ''member'') DotCall STR ''name'')
DotCall EqualOp [Var STR ''mn'']))),
(STR ''allProjects'', Project[1], [], Set Project[1], True,
Some (MetaOperationCall Project[1] AllInstancesOp))
] :: (classes1 type, classes1 expr) oper-spec list

definition *literals-classes1 ≡ fmap-of-list [*
(STR ''E1'' :: classes1 enum, {|STR ''A'', STR ''B''|}),
(STR ''E2'', {|STR ''C'', STR ''D'', STR ''E''|})]

lemma *assoc-end-min-less-eq-max*:
assoc |∈| fmdom assocs ⟹
fmlookup assocs assoc = Some ends ⟹
role |∈| fmdom ends ⟹
fmlookup ends role = Some end ⟹
assoc-end-min end ≤ assoc-end-max end
⟨*proof*⟩

lemma *association-ends-unique*:
assumes *association-ends' classes assocs* $\mathcal{C}$ *from role* end_1
and *association-ends' classes assocs* $\mathcal{C}$ *from role* end_2
shows $end_1 = end_2$
⟨*proof*⟩

instance
⟨*proof*⟩

end

8.3 Simplification Rules

lemma *ex-alt-simps* [*simp*]:
$\exists\, a.\ a$
$\exists\, a.\ \neg\ a$
$(\exists\, a.\ (a \longrightarrow P) \wedge a) = P$
$(\exists\, a.\ \neg\ a \wedge (\neg\ a \longrightarrow P)) = P$
⟨*proof*⟩

declare *numeral-eq-enat* [*simp*]

lemmas *basic-type-le-less* [*simp*] = *Orderings.order-class.le-less*
for *x y* :: *'a basic-type*

declare *element-type-alt-simps* [*simp*]
declare *update-element-type.simps* [*simp*]
declare *to-unique-collection.simps* [*simp*]
declare *to-nonunique-collection.simps* [*simp*]
declare *to-ordered-collection.simps* [*simp*]

declare *assoc-end-class-def* [*simp*]
declare *assoc-end-min-def* [*simp*]
declare *assoc-end-max-def* [*simp*]
declare *assoc-end-ordered-def* [*simp*]
declare *assoc-end-unique-def* [*simp*]

declare *oper-name-def* [*simp*]
declare *oper-context-def* [*simp*]
declare *oper-params-def* [*simp*]
declare *oper-result-def* [*simp*]
declare *oper-static-def* [*simp*]
declare *oper-body-def* [*simp*]

declare *oper-in-params-def* [*simp*]
declare *oper-out-params-def* [*simp*]

declare *assoc-end-type-def* [*simp*]

declare *oper-type-def* [*simp*]

declare *op-type-alt-simps* [*simp*]
declare *typing-alt-simps* [*simp*]
declare *normalize-alt-simps* [*simp*]
declare *nf-typing.simps* [*simp*]

declare *subclass1.intros* [*intro*]
declare *less-classes1-def* [*simp*]

declare *literals-classes1-def* [*simp*]

lemma *attribute-Employee-name* [*simp*]:
attribute Employee STR ''name'' $\mathcal{D}$ τ =
($\mathcal{D}$ = Employee $\wedge$ τ = String[1])
⟨*proof*⟩

lemma *association-end-Project-members* [*simp*]:
association-end Project None STR ''members'' $\mathcal{D}$ τ =
($\mathcal{D}$ = Project $\wedge$ τ = (Employee, 1, 20, True, True))
⟨*proof*⟩

lemma *association-end-Employee-projects-simp* [*simp*]:
association-end Employee None STR ''projects'' $\mathcal{D}$ τ =
($\mathcal{D}$ = Employee $\wedge$ τ = (Project, 0, ∞, False, True))
⟨*proof*⟩

lemma *static-operation-Project-allProjects* [*simp*]:
static-operation ⟨Project⟩$_\mathcal{T}$[1] STR ''allProjects'' [] oper =
(oper = (STR ''allProjects'', ⟨Project⟩$_\mathcal{T}$[1], [], Set ⟨Project⟩$_\mathcal{T}$[1], True,
Some (MetaOperationCall ⟨Project⟩$_\mathcal{T}$[1] AllInstancesOp)))
⟨*proof*⟩

8.4 Basic Types

8.4.1 Positive Cases

lemma *UnlimitedNatural* $<$ (*Real* :: *classes1 basic-type*) ⟨*proof*⟩
lemma ⟨*Employee*⟩$_\mathcal{T}$ $<$ ⟨*Person*⟩$_\mathcal{T}$ ⟨*proof*⟩
lemma ⟨*Person*⟩$_\mathcal{T}$ $\leq$ *OclAny* ⟨*proof*⟩

8.4.2 Negative Cases

lemma $\neg$ *String* $\leq$ (*Boolean* :: *classes1 basic-type*) ⟨*proof*⟩

8.5 Types

8.5.1 Positive Cases

lemma *Integer*[*?*] < (*OclSuper* :: *classes1 type*) ⟨*proof*⟩
lemma *Collection Real*[*?*] < (*OclSuper* :: *classes1 type*) ⟨*proof*⟩
lemma *Set* (*Collection Boolean*[*1*]) < (*OclSuper* :: *classes1 type*) ⟨*proof*⟩
lemma *Set* (*Bag Boolean*[*1*]) < *Set* (*Collection Boolean*[*?*] :: *classes1 type*)
⟨*proof*⟩
lemma *Tuple* (*fmap-of-list* [(*STR* ''*a*'', *Boolean*[*1*]), (*STR* ''*b*'', *Integer*[*1*])]) <
Tuple (*fmap-of-list* [(*STR* ''*a*'', *Boolean*[*?*] :: *classes1 type*)]) ⟨*proof*⟩

lemma *Integer*[*1*] ⊔ (*Real*[*?*] :: *classes1 type*) = *Real*[*?*] ⟨*proof*⟩
lemma *Set Integer*[*1*] ⊔ *Set* (*Real*[*1*] :: *classes1 type*) = *Set Real*[*1*] ⟨*proof*⟩
lemma *Set Integer*[*1*] ⊔ *Bag* (*Boolean*[*?*] :: *classes1 type*) = *Collection OclAny*[*?*]

⟨*proof*⟩
lemma *Set Integer*[*1*] ⊔ (*Real*[*1*] :: *classes1 type*) = *OclSuper* ⟨*proof*⟩

8.5.2 Negative Cases

lemma ¬ *OrderedSet Boolean*[*1*] < *Set* (*Boolean*[*1*] :: *classes1 type*) ⟨*proof*⟩

8.6 Typing

8.6.1 Positive Cases

```
E1::A : E1[1]
```

lemma
Γ$_0$ ⊢ *EnumLiteral STR ''E1'' STR ''A''* : (*Enum STR ''E1''*)[*1*]
⟨*proof*⟩

```
true or false : Boolean[1]
```

lemma
Γ$_0$ ⊢ *OperationCall* (*BooleanLiteral True*) *DotCall OrOp*
[*BooleanLiteral False*] : *Boolean*[*1*]
⟨*proof*⟩

```
null and true : Boolean[?]
```

lemma
Γ$_0$ ⊢ *OperationCall* (*NullLiteral*) *DotCall AndOp*
[*BooleanLiteral True*] : *Boolean*[*?*]
⟨*proof*⟩

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

lemma
Γ$_0$ ⊢ *Let* (*STR ''x''*) (*Some Real*[*1*]) (*IntegerLiteral 5*)
(*OperationCall* (*Var STR ''x''*) *DotCall PlusOp* [*IntegerLiteral 7*]) : *Real*[*1*]
⟨*proof*⟩

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

lemma
$\Gamma_0 \vdash$ *OperationCall* (*NullLiteral*) *DotCall OclIsUndefinedOp* [] : *Boolean*[*1*]
⟨*proof*⟩

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

lemma
$\Gamma_0 \vdash$ *OperationCall* (*CollectionLiteral SequenceKind*
[*CollectionRange* (*IntegerLiteral 1*) (*IntegerLiteral 5*),
CollectionItem NullLiteral])
DotCall OclIsUndefinedOp [] : *Sequence Boolean*[*1*]
⟨*proof*⟩

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

lemma
$\Gamma_0 \vdash$ *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*])]))
⟨*proof*⟩

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

lemma
$\Gamma_0 \vdash$ *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*]
⟨*proof*⟩

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

lemma
$\Gamma_0 \vdash$ *OperationCall* (*CollectionLiteral SequenceKind*
[*CollectionRange* (*IntegerLiteral 1*) (*IntegerLiteral 5*),
CollectionItem NullLiteral])
SafeArrowCall CollectionMaxOp [] : *Integer*[*1*]
⟨*proof*⟩

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

lemma

$\Gamma_0 \vdash$ *Let* (*STR* ''*x*'') (*Some* (*Sequence String*[*?*]))
(*CollectionLiteral SequenceKind*
[*CollectionItem* (*StringLiteral* ''*abc*''),
CollectionItem (*StringLiteral* ''*zxc*'')])
(*AnyIteratorCall* (*Var STR* ''*x*'') *ArrowCall*
[*STR* ''*it*''] *None*
(*OperationCall* (*Var STR* ''*it*'') *DotCall EqualOp*
[*StringLiteral* ''*test*'']))
: *String*[*?*]
⟨*proof*⟩

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

lemma
$\Gamma_0 \vdash$ *Let STR* ''*x*'' (*Some* (*Sequence String*[*?*]))
(*CollectionLiteral SequenceKind*
[*CollectionItem* (*StringLiteral* ''*abc*''),
CollectionItem (*StringLiteral* ''*zxc*'')])
(*ClosureIteratorCall* (*Var STR* ''*x*'') *SafeArrowCall*
[*STR* ''*it*''] *None*
(*Var STR* ''*it*''))
: *OrderedSet String*[*1*]
⟨*proof*⟩

```
context Employee:
name : String[1]
```

lemma
Γ_0(*STR* ''*self*'' $\mapsto_f$ *Employee*[*1*]) $\vdash$
AttributeCall (*Var STR* ''*self*'') *DotCall STR* ''*name*'' : *String*[*1*]
⟨*proof*⟩

```
context Employee:
projects : Set(Project[1])
```

lemma
Γ_0(*STR* ''*self*'' $\mapsto_f$ *Employee*[*1*]) $\vdash$
AssociationEndCall (*Var STR* ''*self*'') *DotCall None*
STR ''*projects*'' : *Set Project*[*1*]
⟨*proof*⟩

```
context Employee:
projects.members : Bag(Employee[1])
```

lemma
Γ_0(*STR* ''*self*'' $\mapsto_f$ *Employee*[*1*]) $\vdash$
AssociationEndCall (*AssociationEndCall* (*Var STR* ''*self*'')
DotCall None STR ''*projects*'')
DotCall None STR ''*members*'' : *Bag Employee*[*1*]
⟨*proof*⟩

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

lemma
$\Gamma_0 \vdash$ *MetaOperationCall Project*[*?*] *AllInstancesOp* : *Set Project*[*?*]

⟨*proof*⟩

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

lemma
$\Gamma_0 \vdash$ *StaticOperationCall Project*[*1*] *STR ''allProjects''* [] : *Set Project*[*1*]
⟨*proof*⟩

8.6.2 Negative Cases

```
true = null
```

lemma
$\nexists \tau.\ \Gamma_0 \vdash$ *OperationCall* (*BooleanLiteral True*) *DotCall EqualOp*
[*NullLiteral*] : τ
⟨*proof*⟩

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

lemma
$\nexists \tau.\ \Gamma_0 \vdash$ *Let STR ''x''* (*Some Boolean*[*1*]) (*IntegerLiteral 5*)
(*OperationCall* (*Var STR ''x''*) *DotCall AndOp* [*BooleanLiteral True*]) : τ
⟨*proof*⟩

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

lemma
$\nexists \tau.\ \Gamma_0 \vdash$ *Let STR ''x''* (*Some* (*Sequence String*[*?*]))
(*CollectionLiteral SequenceKind*
[*CollectionItem* (*StringLiteral ''abc''*),
CollectionItem (*StringLiteral ''zxc''*)])
(*ClosureIteratorCall* (*Var STR ''x''*) *ArrowCall* [*STR ''it''*] *None*
(*IntegerLiteral 1*)) : τ
⟨*proof*⟩

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

lemma
$\nexists \tau.\ \Gamma_0 \vdash$ *OperationCall* (*CollectionLiteral SequenceKind*
[*CollectionRange* (*IntegerLiteral 1*) (*IntegerLiteral 5*),
CollectionItem NullLiteral])
ArrowCall CollectionMaxOp [] : τ
⟨*proof*⟩

8.7 Code

8.7.1 Positive Cases

values {($\mathcal{D}$, τ). *attribute Employee STR ''name''* $\mathcal{D}$ τ}
values {($\mathcal{D}$, *end*). *association-end Employee None STR ''employees''* $\mathcal{D}$ *end*}
values {($\mathcal{D}$, *end*). *association-end Employee* (*Some STR ''project-manager''*) *STR ''employees''* $\mathcal{D}$ *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
{τ. Γ_0(*STR* ''*self*'' $\mapsto_f$ *Employee*[*1*]) $\vdash$
AssociationEndCall (*AssociationEndCall* (*Var STR* ''*self*'')
DotCall None STR ''*projects*'')
DotCall None STR ''*members*'' : τ}

8.7.2 Negative Cases

values {($\mathcal{D}$, τ). *attribute Employee STR* ''*name2*'' $\mathcal{D}$ τ}
value *has-literal STR* ''*E1*'' *STR* ''*C*''

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

values
{τ. Γ_0 $\vdash$ *OperationCall* (*CollectionLiteral SequenceKind*
[*CollectionRange* (*IntegerLiteral 1*) (*IntegerLiteral 5*),
CollectionItem NullLiteral])
ArrowCall CollectionMaxOp [] : τ}

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, 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.