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* **..**

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

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

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

copy-bnf *'a errorable*

end

1.2 Transitive Closures

theory *Transitive-Closure-Ext*

imports *HOL−Library.FuncSet*
begin

1.2.1 Basic Properties

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

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

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

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

abbreviation *bij-on-trancl R f* ≡ *inj f* ∧ *surj-on-trancl R f*

1.2.2 Helper Lemmas

lemma *rtranclp-eq-rtranclp* [*iff*]:
  $(\lambda x\ y.\ P\ x\ y \lor x = y)^{**} = P^{**}$
proof (*intro ext iffI*)
  fix *x y*
  have $(\lambda x\ y.\ P\ x\ y \lor x = y)^{**}\ x\ y \longrightarrow P^{==**}\ x\ y$
    by (*rule mono-rtranclp*) *simp*
  thus $(\lambda x\ y.\ P\ x\ y \lor x = y)^{**}\ x\ y \Longrightarrow P^{**}\ x\ y$
    by *simp*
  show $P^{**}\ x\ y \Longrightarrow (\lambda x\ y.\ P\ x\ y \lor x = y)^{**}\ x\ y$
    by (*metis* (*no-types*, *lifting*) *mono-rtranclp*)
qed

lemma *tranclp-eq-rtranclp* [*iff*]:
  $(\lambda x\ y.\ P\ x\ y \lor x = y)^{++} = P^{**}$
proof (*intro ext iffI*)
  fix *x y*
  have $(\lambda x\ y.\ P\ x\ y \lor x = y)^{**}\ x\ y \longrightarrow P^{==**}\ x\ y$
    by (*rule mono-rtranclp*) *simp*
  thus $(\lambda x\ y.\ P\ x\ y \lor x = y)^{++}\ x\ y \Longrightarrow P^{**}\ x\ y$
    using *tranclp-into-rtranclp* **by** *force*
  show $P^{**}\ x\ y \Longrightarrow (\lambda x\ y.\ P\ x\ y \lor x = y)^{++}\ x\ y$
    by (*metis* (*mono-tags*, *lifting*) *mono-rtranclp rtranclpD tranclp.r-into-trancl*)
qed

lemma *rtranclp-eq-rtranclp′* [*iff*]:
  $(\lambda x\ y.\ P\ x\ y \land x \neq y)^{**} = P^{**}$
proof (*intro ext iffI*)
  fix *x y*
  show $(\lambda x\ y.\ P\ x\ y \land x \neq y)^{**}\ x\ y \Longrightarrow P^{**}\ x\ y$
    by (*metis* (*no-types*, *lifting*) *mono-rtranclp*)

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

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

1.2.3 Transitive Closure Preservation

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

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

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

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

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

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

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

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

1.2.4 Transitive Closure Reflection

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

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

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

qed

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

lemma *reflect-rtranclp*:
  $(\bigwedge x\ y.\ S\ (f\ x)\ (f\ y) \Longrightarrow R\ x\ y) \Longrightarrow$
  bij-on-trancl S f $\Longrightarrow$
  $S^{**}\ (f\ x)\ (f\ y) \Longrightarrow R^{**}\ x\ y$
 unfolding *Nitpick.rtranclp-unfold*
 by (*metis* (*full-types*) *injD reflect-tranclp*)

end

1.3 Finite Maps

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

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

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

syntax
  -fmaplet :: [′*a*, ′*a*] ⇒ *fmaplet* (‹- /↦$_f$/ -›)
  -fmaplets :: [′*a*, ′*a*] ⇒ *fmaplet* (‹- /[↦$_f$]/ -›)
  :: *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* $\Longrightarrow$
  k |∈| *fmdom ym* $\Longrightarrow$

```
  k |∈| fmdom xm
  by (metis fmdom-notD fmdom-notI fmrel-on-fsetD option.rel-sel)
```

1.3.2 Merge Operation

```
definition fmmerge f xm ym ≡
  fmap-of-list (map
    (λk. (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
  by (auto simp add: fmmerge-def fmdom-of-list)

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 −
  obtain zm where zm: zm = sorted-list-of-fset (fmdom xm |∩| fmdom ym)
    by auto
  with assms have
      map (λk. (k, f (the (fmlookup xm k)) (the (fmlookup ym k)))) zm =
      map (λk. (k, f (the (fmlookup ym k)) (the (fmlookup xm k)))) zm
    by (auto) (metis fmdom-notI fmran'I option.collapse)
  thus ?thesis
    unfolding fmmerge-def zm
    by (metis (no-types, lifting) inf-commute)
qed

lemma fmrel-on-fset-fmmerge1 [intro]:
  assumes ⋀x y z. z ∈ fmran' zm ⟹ f x z ⟹ f y z ⟹ 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 −
  {
    fix x a b c
    assume x |∈| fmdom zm
    moreover hence x |∈| fmdom xm |∩| fmdom ym
      by (meson assms(2) assms(3) finterI fmrel-on-fset-fmdom)
    moreover assume fmlookup xm x = Some a
               and fmlookup ym x = Some b
               and fmlookup zm x = Some c
    moreover from assms calculation have f (g a b) c
      by (metis fmran'I fmrel-on-fsetD option.rel-inject(2))
    ultimately have
        rel-option f (fmlookup (fmmerge g xm ym) x) (fmlookup zm x)
      unfolding fmmerge-def fmlookup-of-list apply auto
      unfolding option-rel-Some2 apply (rule-tac ?x= g a b in exI)
```

unfolding *map-of-map-restrict restrict-map-def*
by *auto*
}
with *assms*(*2*) *assms*(*3*) **show** *?thesis*
by (*meson fmdomE fmrel-on-fsetI fmrel-on-fset-fmdom*)
qed

lemma *fmrel-on-fset-fmmerge2* [*intro*]:
assumes $\bigwedge x\ y.\ x \in fmran'\ xm \Longrightarrow f\ x\ (g\ x\ y)$
shows *fmrel-on-fset* (*fmdom ym*) *f xm* (*fmmerge g xm ym*)
proof −
{
fix *x a b*
assume *x* |∈| *fmdom xm* |∩| *fmdom ym*
and *fmlookup xm x* = *Some a*
and *fmlookup ym x* = *Some b*
hence *rel-option f* (*fmlookup xm x*) (*fmlookup* (*fmmerge g xm ym*) *x*)
unfolding *fmmerge-def fmlookup-of-list* **apply** *auto*
unfolding *option-rel-Some1* **apply** (*rule-tac ?x= g a b* **in** *exI*)
by (*auto simp add*: *map-of-map-restrict assms fmran'I*)
}
with *assms* **show** *?thesis*
apply *auto*
apply (*rule fmrel-on-fsetI*)
by (*metis* (*full-types*) *finterD1 fmdomE fmdom-fmmerge fmdom-notD rel-option-None2*)
qed

1.3.3 Acyclicity

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

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

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

from *assms*(*1*) **have**
rel-option R^{++} (*fmlookup xm x*) (*fmlookup ym x*) $\Longrightarrow$
rel-option R (*fmlookup ym x*) (*fmlookup xm x*) $\Longrightarrow$
rel-option R (*fmlookup xm x*) (*fmlookup ym x*)
by (*metis* (*full-types*) *fmdom′-notD fmlookup-dom′-iff*
fmran′I option.rel-sel option.sel
tranclp-into-tranclp2 tranclp-unfold)
}
with *assms* **show** *?thesis*
unfolding *fmrel-iff*
by (*metis fmap.rel-mono-strong fmrelI fmrel-acyclic tranclp.simps*)
qed

lemma *fmrel-on-fset-acyclic*:
acyclicP-on (*fmran′ xm*) *R* $\Longrightarrow$
fmrel-on-fset (*fmdom ym*) R^{++} *xm ym* $\Longrightarrow$
fmrel-on-fset (*fmdom xm*) *R ym xm* $\Longrightarrow$
xm = *ym*
unfolding *fmrel-on-fset-fmrel-restrict*
by (*metis* (*no-types*, *lifting*) *fmdom-filter fmfilter-alt-defs*(*5*)
fmfilter-cong fmlookup-filter fmrel-acyclic fmrel-fmdom-eq
fmrestrict-fset-dom option.simps(*3*))

lemma *fmrel-on-fset-acyclic′*:
acyclicP-on (*fmran′ ym*) *R* $\Longrightarrow$
fmrel-on-fset (*fmdom ym*) R^{++} *xm ym* $\Longrightarrow$
fmrel-on-fset (*fmdom xm*) *R ym xm* $\Longrightarrow$
xm = *ym*
unfolding *fmrel-on-fset-fmrel-restrict*
by (*metis* (*no-types*, *lifting*) *ffmember-filter fmdom-filter*
fmfilter-alt-defs(*5*) *fmfilter-cong fmrel-acyclic′*
fmrel-fmdom-eq fmrestrict-fset-dom)

1.3.4 Transitive Closures

lemma *fmrel-trans*:
($\bigwedge x\ y\ z.\ x \in$ *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*
unfolding *fmrel-iff*
by (*metis fmdomE fmdom-notD fmran′I option.rel-inject*(*2*) *option.rel-sel*)

lemma *fmrel-on-fset-trans*:
($\bigwedge x\ y\ z.\ x \in$ *fmran′ xm* $\Longrightarrow P\ x\ y \Longrightarrow Q\ y\ z \Longrightarrow R\ x\ z$) $\Longrightarrow$
fmrel-on-fset (*fmdom ym*) *P xm ym* $\Longrightarrow$
fmrel-on-fset (*fmdom zm*) *Q ym zm* $\Longrightarrow$
fmrel-on-fset (*fmdom zm*) *R xm zm*
apply (*rule fmrel-on-fsetI*)
unfolding *option.rel-sel* **apply** *auto*
apply (*meson fmdom-notI fmrel-on-fset-fmdom*)

by (*metis fmdom-notI fmran'I fmrel-on-fsetD fmrel-on-fset-fmdom*
option.rel-sel option.sel)

lemma *trancl-to-fmrel*:
$(fmrel\ f)^{++}\ xm\ ym \Longrightarrow fmrel\ f^{++}\ xm\ ym$
apply (*induct rule*: *tranclp-induct*)
apply (*simp add*: *fmap.rel-mono-strong*)
by (*rule fmrel-trans*; *auto*)

lemma *fmrel-trancl-fmdom-eq*:
$(fmrel\ f)^{++}\ xm\ ym \Longrightarrow fmdom\ xm = fmdom\ ym$
by (*induct rule*: *tranclp-induct*; *simp add*: *fmrel-fmdom-eq*)

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

lemma *fmupd-fmdrop*:
fmlookup xm k = *Some x* ⟹
xm = *fmupd k x* (*fmdrop k xm*)
apply (*rule fmap-ext*)
unfolding *fmlookup-drop fmupd-lookup*
by *auto*

lemma *fmap-eqdom-Cons1*:
assumes *fmlookup xm i* = *None*
and *fmdom* (*fmupd i x xm*) = *fmdom ym*
and *fmrel R* (*fmupd i x xm*) *ym*
shows (∃ *z zm*. *fmlookup zm i* = *None* ∧ *ym* = (*fmupd i z zm*) ∧
R x z ∧ *fmrel R xm zm*)
proof −
from *assms*(*2*) **obtain** *y* **where** *fmlookup ym i* = *Some y* **by** *force*
then obtain *z zm* **where** *z-zm*: *ym* = *fmupd i z zm* ∧ *fmlookup zm i* = *None*
using *fmupd-fmdrop* **by** *force*
{
assume ¬ *R x z*
with *z-zm* **have** ¬ *fmrel R* (*fmupd i x xm*) *ym*
by (*metis fmrel-iff fmupd-lookup option.simps*(*11*))
}
with *assms*(*3*) **moreover have** *R x z* **by** *auto*
{
assume ¬ *fmrel R xm zm*
with *assms*(*1*) **have** ¬ *fmrel R* (*fmupd i x xm*) *ym*
by (*metis fmrel-iff fmupd-lookup option.rel-sel z-zm*)
}
with *assms*(*3*) **moreover have** *fmrel R xm zm* **by** *auto*
ultimately show *?thesis* **using** *z-zm* **by** *blast*
qed

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

```
lemma fmap-eqdom-induct [consumes 2, case-names nil step]:
  assumes R: fmrel R xm ym
    and dom-eq: fmdom xm = fmdom ym
    and nil: P (fmap-of-list []) (fmap-of-list [])
    and step:
     ⋀x xm y ym i.
     ⟦R x y; fmrel R xm ym; fmdom xm = fmdom ym; P xm ym⟧ ⟹
     P (fmupd i x xm) (fmupd i y ym)
  shows P xm ym
  using R dom-eq
proof (induct xm arbitrary: ym)
  case fmempty thus ?case
    by (metis fempty-iff fmdom-empty fmempty-of-list fmfilter-alt-defs(5)
          fmfilter-false fmrestrict-fset-dom local.nil)
next
  case (fmupd i x xm) show ?case
  proof -
    obtain y where fmlookup ym i = Some y
      by (metis fmupd.prems(1) fmrel-cases fmupd-lookup option.discI)
    from fmupd.hyps(2) fmupd.prems(1) fmupd.prems(2) obtain z zm where
       fmlookup zm i = None and
      ym-eq-z-zm: ym = (fmupd i z zm) and
      R-x-z: R x z and
      R-xm-zm: fmrel R xm zm
      using fmap-eqdom-Cons1 by metis
    hence dom-xm-eq-dom-zm: fmdom xm = fmdom zm
      using fmrel-fmdom-eq by blast
    with R-xm-zm fmupd.hyps(1) have P xm zm by blast
    with R-x-z R-xm-zm dom-xm-eq-dom-zm have
       P (fmupd i x xm) (fmupd i z zm)
      by (rule step)
    thus ?thesis by (simp add: ym-eq-z-zm)
  qed
qed
```

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

```
lemma fmrel-to-rtrancl:
  assumes as-r: reflp r
      and rel-rpp-xm-ym: fmrel r** xm ym
    shows (fmrel r)** xm ym
proof -
  from rel-rpp-xm-ym have fmdom xm = fmdom ym
    using fmrel-fmdom-eq by blast
  with rel-rpp-xm-ym show (fmrel r)** xm ym
  proof (induct rule: fmap-eqdom-induct)
    case nil show ?case by auto
```

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

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

```
lemma fmrel-to-trancl:
  assumes reflp r
      and fmrel r++ xm ym
    shows (fmrel r)++ xm ym
proof -
  from assms(2) have fmrel r** xm ym
    by (drule-tac ?Ra= r** in fmap.rel-mono-strong; auto)
  with assms(1) have (fmrel r)** xm ym
    by (simp add: fmrel-to-rtrancl)
  with assms(1) show ?thesis
    by (metis fmap.rel-reflp reflpD rtranclpD tranclp.r-into-trancl)
qed
```

```
lemma fmrel-tranclp-induct:
```

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

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

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

1.3.5 Size Calculation

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

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

interpretation *tcf*: *comp-fun-commute tcf*
proof
fix $x\ y$ :: $'a \times \mathit{nat}$
show $\mathit{tcf}\ y \circ \mathit{tcf}\ x = \mathit{tcf}\ x \circ \mathit{tcf}\ y$
proof −
fix z
have $(\mathit{tcf}\ y \circ \mathit{tcf}\ x)\ z = \mathit{snd}\ y + \mathit{snd}\ x + z$ **by** *auto*
also have $(\mathit{tcf}\ x \circ \mathit{tcf}\ y)\ z = \mathit{snd}\ y + \mathit{snd}\ x + z$ **by** *auto*
finally have $(\mathit{tcf}\ y \circ \mathit{tcf}\ x)\ z = (\mathit{tcf}\ x \circ \mathit{tcf}\ y)\ z$ **by** *auto*
thus $(\mathit{tcf}\ y \circ \mathit{tcf}\ x) = (\mathit{tcf}\ x \circ \mathit{tcf}\ y)$ **by** *auto*
qed
qed

lemma *ffold-rec-exp*:
assumes $k\ |{\in}|\ \mathit{fmdom}\ x$
and $\mathit{ky} = (k,\ \mathit{the}\ (\mathit{fmlookup}\ (\mathit{fmmap}\ f\ x)\ k))$

shows *ffold tcf 0* (*fset-of-fmap* (*fmmap f x*)) =
tcf ky (*ffold tcf 0* ((*fset-of-fmap* (*fmmap f x*)) |−| {|*ky*|}))
proof −
have *ky* |∈| (*fset-of-fmap* (*fmmap f x*))
using *assms* **by** *auto*
thus *?thesis*
by (*simp add*: *tcf.ffold-rec*)
qed

lemma *elem-le-ffold* [*intro*]:
k |∈| *fmdom x* ⟹
f (*the* (*fmlookup x k*)) < *Suc* (*ffold tcf 0* (*fset-of-fmap* (*fmmap f x*)))
by (*subst ffold-rec-exp*, *auto*)

lemma *elem-le-ffold′* [*intro*]:
z ∈ *fmran′ x* ⟹
f z < *Suc* (*ffold tcf 0* (*fset-of-fmap* (*fmmap f x*)))
apply (*erule fmran′E*)
apply (*frule fmdomI*)
by (*subst ffold-rec-exp*, *auto*)

1.3.6 Code Setup

abbreviation *fmmerge-fun f xm ym* ≡
fmap-of-list (*map*
(λ*k*. *if k* |∈| *fmdom xm* ∧ *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*
unfolding *fmmerge-def*
apply (*rule-tac ?f= fmap-of-list* **in** *HOL.arg-cong*)
by *simp*

end

1.4 Tuples

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

1.4.1 Definitions

abbreviation *subtuple f xm ym* ≡ *fmrel-on-fset* (*fmdom ym*) *f xm ym*

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

1.4.2 Helper Lemmas

lemma *fmrel-to-subtuple*:
fmrel r xm ym $\Longrightarrow$ *subtuple r xm ym*
unfolding *fmrel-on-fset-fmrel-restrict* **by** *blast*

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

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

1.4.3 Basic Properties

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

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

lemma *strict-subtuple-mono* [*mono*]:
($\bigwedge$*x y*. *x* $\in$ *fmran′ xm* $\Longrightarrow$ *y* $\in$ *fmran′ ym* $\Longrightarrow$ *f x y* $\longrightarrow$ *g x y*) $\Longrightarrow$
strict-subtuple f xm ym $\longrightarrow$ *strict-subtuple g xm ym*
using *subtuple-mono* **by** *blast*

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

qed

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

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

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

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

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

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

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

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

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

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

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*)
by (*rule-tac ?S= fmdom ym* **in** *fmrel-on-fsubset*; *auto*)

1.4.4 Transitive Closures

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

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

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

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

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

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

lemma *trancl-to-strict-subtuple'*:
acyclicP-on (*fmran' ym*) $R \Longrightarrow$
(*strict-subtuple* ($\lambda x\ y.\ R\ x\ y \lor x = y$))$^{++}$ *xm ym* $\Longrightarrow$
strict-subtuple ($\lambda x\ y.\ R\ x\ y \lor x = y$)** *xm ym*
apply (*erule converse-tranclp-induct*)
apply (*metis* (*no-types*, *lifting*) *r-into-rtranclp strict-subtuple-mono*)
using *strict-subtuple-trans'''* **by** *blast*

lemma *subtuple-rtranclp-intro*:
assumes $\bigwedge$*xm ym*. *R* (*f xm*) (*f ym*) $\Longrightarrow$ *subtuple R xm ym*
and *bij-on-trancl R f*
and R^{**} (*f xm*) (*f ym*)
shows *subtuple* R^{**} *xm ym*
proof −
have (λ*xm ym*. *R* (*f xm*) (*f ym*))** *xm ym*
apply (*insert assms*(*2*) *assms*(*3*))
by (*rule reflect-rtranclp*; *auto*)
with *assms*(*1*) **have** (*subtuple R*)** *xm ym*
by (*metis* (*mono-tags*, *lifting*) *mono-rtranclp*)
hence *subtuple* R^{**} *xm ym*
by (*rule rtrancl-to-subtuple*)
thus *?thesis* **by** *simp*
qed

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

1.4.5 Code Setup

abbreviation *subtuple-fun f xm ym* $\equiv$
fBall (*fmdom ym*) (λx. *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*
by (*simp add*: *fmrel-on-fset-alt-def*)

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

end

1.5 Object Model

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

The section defines a generic object model.

1.5.1 Type Synonyms

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

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

type-synonym *'c assoc-end* = *'c* × *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 $oper\text{-}static :: ('t,\ 'e)\ oper\text{-}spec \Rightarrow bool \equiv fst \circ snd \circ snd \circ snd \circ snd$
definition $oper\text{-}body :: ('t,\ 'e)\ oper\text{-}spec \Rightarrow 'e\ option \equiv snd \circ snd \circ snd \circ snd \circ snd$

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

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

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

1.5.2 Attributes

inductive $owned\text{-}attribute'$ **where**
$\mathcal{C}\ |\in|\ fmdom\ attributes \Longrightarrow$
$fmlookup\ attributes\ \mathcal{C} = Some\ attrs_{\mathcal{C}} \Longrightarrow$
$fmlookup\ attrs_{\mathcal{C}}\ attr = Some\ \tau \Longrightarrow$
$owned\text{-}attribute'\ attributes\ \mathcal{C}\ attr\ \tau$

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

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

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

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

1.5.3 Association Ends

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

role-refer-class ends $\mathcal{C}$ *role*

inductive *association-ends′* **where**
$\mathcal{C}$ |∈| *classes* ⟹
assoc |∈| *fmdom associations* ⟹
fmlookup associations assoc = *Some ends* ⟹
role-refer-class ends $\mathcal{C}$ *from* ⟹
role |∈| *fmdom ends* ⟹
fmlookup ends role = *Some end* ⟹
role ≠ *from* ⟹
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 ⟹
association-ends′ classes associations $\mathcal{C}$ *from role* end_2 ⟹
$end_1 \neq end_2$ ⟹
association-ends-not-unique′ classes associations

inductive *owned-association-end′* **where**
association-ends′ classes associations $\mathcal{C}$ *from role end* ⟹
owned-association-end′ classes associations $\mathcal{C}$ *None role end*
| *association-ends′ classes associations* $\mathcal{C}$ *from role end* ⟹
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′* ⟹
$\mathcal{C} \leq \mathcal{D}'$ ⟹
$\mathcal{D}' < \mathcal{D}$ ⟹
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* ⟹
$\mathcal{C} \leq \mathcal{D}$ ⟹
¬ *association-end-not-closest classes associations* $\mathcal{C}$ *from role* $\mathcal{D}$ *end* ⟹
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′* ⟹
$\mathcal{D} \neq \mathcal{D}' \vee end \neq end'$ ⟹
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* ⟹
¬ *closest-association-end-not-unique classes associations* $\mathcal{C}$ *from role* $\mathcal{D}$ *end* ⟹
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* $\Longrightarrow$
$\neg$ *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* $\equiv$
unique-association-class-end association-classes associations

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

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

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

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

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

end

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

declare *has-literal′.simps* [*simp*]

1.5.9 Properties

lemma (**in** *object-model*) *attribute-det*:
attribute $\mathcal{C}$ *attr* $\mathcal{D}_1$ τ_1 $\Longrightarrow$
attribute $\mathcal{C}$ *attr* $\mathcal{D}_2$ τ_2 $\Longrightarrow \mathcal{D}_1 = \mathcal{D}_2 \wedge \tau_1 = \tau_2$
by (*meson closest-attribute-not-unique.intros unique-closest-attribute.cases*)

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

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

lemma (**in** *object-model*) *association-end-det*:
association-end $\mathcal{C}$ *from role* $\mathcal{D}_1$ end_1 $\Longrightarrow$
association-end $\mathcal{C}$ *from role* $\mathcal{D}_2$ end_2 $\Longrightarrow \mathcal{D}_1 = \mathcal{D}_2 \wedge end_1 = end_2$
by (*meson closest-association-end-not-unique.intros unique-closest-association-end.cases*)

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

association-end $\mathcal{C}$ *from role* $\mathcal{D}$ *end* $\Longrightarrow$ $\mathcal{C} \leq \mathcal{D}$
by (*meson closest-association-end.cases unique-closest-association-end.cases*)

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* $\Longrightarrow$
$\mathcal{C} \leq \mathcal{D}' \Longrightarrow \neg\, \mathcal{D}' < \mathcal{D}$
by (*meson association-end-not-closest.intros closest-association-end.cases unique-closest-association-end.cases*)

lemma (**in** *object-model*) *association-class-end-det*:
association-class-end $\mathcal{A}$ *role* end_1 $\Longrightarrow$
association-class-end $\mathcal{A}$ *role* end_2 $\Longrightarrow$ $end_1 = end_2$
by (*meson association-class-end-not-unique.intros unique-association-class-end.cases*)

lemma (**in** *object-model*) *operation-det*:
operation τ *name* π $oper_1$ $\Longrightarrow$
operation τ *name* π $oper_2$ $\Longrightarrow$ $oper_1 = oper_2$
by (*meson operation-not-unique.intros unique-operation.cases*)

lemma (**in** *object-model*) *static-operation-det*:
static-operation τ *name* π $oper_1$ $\Longrightarrow$
static-operation τ *name* π $oper_2$ $\Longrightarrow$ $oper_1 = oper_2$
by (*meson static-operation-not-unique.intros unique-static-operation.cases*)

1.5.10 Code Setup

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

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

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

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

$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow o \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow bool$,
$i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$) *association-ends'*
by (*auto simp*: *Predicate-Compile.contains-def elim*: *association-ends'.cases*)

code-pred *association-ends-not-unique'* **.**

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'* **.**

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* .

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* .

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* .

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

code-pred *association-class-end′* .

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

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

declare *any-operation′.intros*[*folded Predicate-Compile.contains-def*, *code-pred-intro*]
code-pred [*show-modes*] *any-operation′*
by (*elim any-operation′.cases*) (*simp add*: *Predicate-Compile.contains-def*)

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

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

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* .

code-pred *operation-defined′* .

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′* .

code-pred (*modes*:

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

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* .

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

declare *has-literal′.intros*[*folded Predicate-Compile.contains-def*, *code-pred-intro*]
code-pred (*modes*:
$i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow bool$) *has-literal′*
by (*elim has-literal′.cases*) (*simp add*: *Predicate-Compile.contains-def*)

end

Chapter 2

Basic Types

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

2.1 Definition

Basic types are parameterized over classes.

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

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

inductive *basic-subtype* (**infix** ‹$\sqsubset_B$› *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*
  by (*induct rule*: *basic-subtype.induct*, *auto*)

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

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

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

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

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

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

2.2.2 Strict Elimination Rules

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

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

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

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

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

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

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

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

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

2.2.3 Properties

lemma *basic-subtype-irrefl*:
$\tau < \tau \Longrightarrow$ *False*
for τ :: *'a basic-type*
by (*cases* τ; *auto*)

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

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

lemma *less-le-not-le-basic-type*:
$\tau < \sigma \longleftrightarrow \tau \leq \sigma \wedge \neg\, \sigma \leq \tau$
for τ σ :: *'a basic-type*
unfolding *less-basic-type-def less-eq-basic-type-def*
apply (*rule iffI*; *auto*)
apply (*metis* (*mono-tags*) *basic-subtype-irrefl*
less-basic-type-def tranclp-rtranclp-tranclp)
by (*drule rtranclpD*; *auto*)

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

lemma *order-refl-basic-type* [*iff*]:
$\tau \leq \tau$
for τ :: *'a basic-type*
by (*simp add*: *less-eq-basic-type-def*)

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

end

2.2.4 Non-Strict Introduction Rules

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

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

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

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

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

2.2.5 Non-Strict Elimination Rules

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

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

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

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

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

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

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

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

lemma *type-less-eq-x-ObjectType* [*elim*!]:
$\tau \le \langle\mathcal{D}\rangle_{\mathcal{T}} \Longrightarrow$
$(\tau = OclVoid \Longrightarrow P) \Longrightarrow$
$(\bigwedge\mathcal{C}.\ \tau = \langle\mathcal{C}\rangle_{\mathcal{T}} \Longrightarrow \mathcal{C} \le \mathcal{D} \Longrightarrow P) \Longrightarrow P$
by (*drule le-imp-less-or-eq*; *auto*)

lemma *type-less-eq-x-Enum* [*elim*!]:
$\tau \le Enum\ \mathcal{E} \Longrightarrow$
$(\tau = OclVoid \Longrightarrow P) \Longrightarrow$
$(\tau = Enum\ \mathcal{E} \Longrightarrow P) \Longrightarrow P$
by (*drule le-imp-less-or-eq*; *auto*)

2.2.6 Simplification Rules

lemma *basic-type-less-left-simps* [*simp*]:
$OclAny < \sigma = False$
$OclVoid < \sigma = (\sigma \ne OclVoid)$
$Boolean < \sigma = (\sigma = OclAny)$
$Real < \sigma = (\sigma = OclAny)$
$Integer < \sigma = (\sigma = OclAny \lor \sigma = Real)$
$UnlimitedNatural < \sigma = (\sigma = OclAny \lor \sigma = Real \lor \sigma = Integer)$
$String < \sigma = (\sigma = OclAny)$
$ObjectType\ \mathcal{C} < \sigma = (\exists\mathcal{D}.\ \sigma = OclAny \lor \sigma = ObjectType\ \mathcal{D} \land \mathcal{C} < \mathcal{D})$
$Enum\ \mathcal{E} < \sigma = (\sigma = OclAny)$
by (*induct* σ, *auto*)

lemma *basic-type-less-right-simps* [*simp*]:
$\tau < OclAny = (\tau \ne OclAny)$

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

2.3 Upper Semilattice of Basic Types

notation *sup* (**infixl** ‹⊔› *65*)

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

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

lemma *sup-ge1-ObjectType*:
$\langle\mathcal{C}\rangle_{\mathcal{T}} \le \langle\mathcal{C}\rangle_{\mathcal{T}} \sqcup \sigma$
apply (*induct* σ; *simp add*: *basic-subtype.simps*
less-eq-basic-type-def r-into-rtranclp)
by (*metis Nitpick.rtranclp-unfold basic-subtype.intros*(*8*)
le-imp-less-or-eq r-into-rtranclp sup-ge1)

lemma *sup-ge1-basic-type*:
$\tau \le \tau \sqcup \sigma$
for τ σ :: *'a basic-type*
apply (*induct* τ, *auto*)
using *sup-ge1-ObjectType* **by** *auto*

lemma *sup-commut-basic-type*:
$\tau \sqcup \sigma = \sigma \sqcup \tau$
for τ σ :: *'a basic-type*
by (*induct* τ; *induct* σ; *auto simp add*: *sup.commute*)

lemma *sup-least-basic-type*:
$\tau \le \varrho \Longrightarrow \sigma \le \varrho \Longrightarrow \tau \sqcup \sigma \le \varrho$
for τ σ ϱ :: *'a basic-type*
by (*induct* ϱ; *auto*)

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

```
end
```

2.4 Code Setup

```
code-pred basic-subtype .

fun basic-subtype-fun :: 'a::order basic-type ⇒ '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⟩_T σ = (case σ
    of ⟨D⟩_T ⇒ C < D
     | OclAny ⇒ True
     | - ⇒ False)
| basic-subtype-fun (Enum -) σ = (σ = OclAny)

lemma less-basic-type-code [code]:
   (<) = basic-subtype-fun
proof (intro ext iffI)
  fix τ σ :: 'a basic-type
  show τ < σ ⟹ basic-subtype-fun τ σ
    apply (cases σ; auto)
    using basic-subtype-fun.elims(3) by fastforce
  show basic-subtype-fun τ σ ⟹ τ < σ
    apply (erule basic-subtype-fun.elims, auto)
    by (cases σ, auto)
qed

lemma less-eq-basic-type-code [code]:
   (≤) = (λx y. basic-subtype-fun x y ∨ x = y)
  unfolding dual-order.order-iff-strict less-basic-type-code
  by auto

end
```

Chapter 3

Types

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

3.1 Definition

Types are parameterized over classes.

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

datatype (*plugins del*: *size*) *'a type* =
  OclSuper
| *Required 'a basic-type* (‹-[1]› [1000] 1000)
| *Optional 'a basic-type* (‹-[?]› [1000] 1000)
| *Collection 'a type*
| *Set 'a type*
| *OrderedSet 'a type*
| *Bag 'a type*
| *Sequence 'a type*
| *Tuple telem* $\rightharpoonup_f$ *'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* :: $'a\ type_\bot \equiv \bot$

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 ..

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*[?] ⊏ *OclSuper*

| $\tau \sqsubset \sigma \Longrightarrow$ *Collection* τ ⊏ *Collection* σ
| $\tau \sqsubset \sigma \Longrightarrow$ *Set* τ ⊏ *Set* σ
| $\tau \sqsubset \sigma \Longrightarrow$ *OrderedSet* τ ⊏ *OrderedSet* σ
| $\tau \sqsubset \sigma \Longrightarrow$ *Bag* τ ⊏ *Bag* σ
| $\tau \sqsubset \sigma \Longrightarrow$ *Sequence* τ ⊏ *Sequence* σ
| *Set* τ ⊏ *Collection* τ
| *OrderedSet* τ ⊏ *Collection* τ
| *Bag* τ ⊏ *Collection* τ
| *Sequence* τ ⊏ *Collection* τ
| *Collection OclSuper* ⊏ *OclSuper*

| *strict-subtuple* ($\lambda\tau\ \sigma.\ \tau \sqsubset \sigma \lor \tau = \sigma$) π ξ $\Longrightarrow$
Tuple π ⊏ *Tuple* ξ
| *Tuple* π ⊏ *OclSuper*

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

inductive-cases *subtype-x-OclSuper* [*elim*!]: τ ⊏ *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*!]: τ ⊏ *Collection* σ
inductive-cases *subtype-x-Set* [*elim*!]: τ ⊏ *Set* σ
inductive-cases *subtype-x-OrderedSet* [*elim*!]: τ ⊏ *OrderedSet* σ
inductive-cases *subtype-x-Bag* [*elim*!]: τ ⊏ *Bag* σ
inductive-cases *subtype-x-Sequence* [*elim*!]: τ ⊏ *Sequence* σ
inductive-cases *subtype-x-Tuple* [*elim*!]: τ ⊏ *Tuple* π

inductive-cases *subtype-OclSuper-x* [*elim*!]: *OclSuper* ⊏ σ
inductive-cases *subtype-Collection-x* [*elim*!]: *Collection* τ ⊏ σ

lemma *subtype-asym*:

$\tau \sqsubset \sigma \Longrightarrow \sigma \sqsubset \tau \Longrightarrow$ *False*
apply (*induct rule*: *subtype.induct*)
using *basic-subtype-asym* **apply** *auto*
using *subtuple-antisym* **by** *fastforce*

3.2 Constructors Bijectivity on Transitive Closures

lemma *Required-bij-on-trancl* [*simp*]:
bij-on-trancl subtype Required
by (*auto simp add*: *inj-def*)

lemma *not-subtype-Optional-Required*:
$subtype^{++}\ \tau[?]\ \sigma \Longrightarrow \sigma = \varrho[1] \Longrightarrow P$
by (*induct arbitrary*: ϱ *rule*: *tranclp-induct*; *auto*)

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

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

lemma *Collection-bij-on-trancl* [*simp*]:
bij-on-trancl subtype Collection
apply (*auto simp add*: *inj-def*)
using *subtype-tranclp-Collection-x* **by** *auto*

lemma *Set-bij-on-trancl* [*simp*]:
bij-on-trancl subtype Set
by (*auto simp add*: *inj-def*)

lemma *OrderedSet-bij-on-trancl* [*simp*]:
bij-on-trancl subtype OrderedSet
by (*auto simp add*: *inj-def*)

lemma *Bag-bij-on-trancl* [*simp*]:
bij-on-trancl subtype Bag
by (*auto simp add*: *inj-def*)

lemma *Sequence-bij-on-trancl* [*simp*]:
bij-on-trancl subtype Sequence
by (*auto simp add*: *inj-def*)

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

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

3.3 Partial Order of Types

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

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

3.3.1 Strict Introduction Rules

lemma *type-less-x-Required-intro* [*intro*]:
$\tau = \varrho[1] \Longrightarrow \varrho < \sigma \Longrightarrow \tau < \sigma[1]$
unfolding *less-type-def less-basic-type-def*
by *simp* (*rule preserve-tranclp*; *auto*)

lemma *type-less-x-Optional-intro* [*intro*]:
$\tau = \varrho[1] \Longrightarrow \varrho \leq \sigma \Longrightarrow \tau < \sigma[?]$
$\tau = \varrho[?] \Longrightarrow \varrho < \sigma \Longrightarrow \tau < \sigma[?]$
unfolding *less-type-def less-basic-type-def less-eq-basic-type-def*
apply *simp-all*
apply (*rule preserve-rtranclp''*; *auto*)
by (*rule preserve-tranclp*; *auto*)

lemma *type-less-x-Collection-intro* [*intro*]:
$\tau = Collection\ \varrho \Longrightarrow \varrho < \sigma \Longrightarrow \tau < Collection\ \sigma$
$\tau = Set\ \varrho \Longrightarrow \varrho \leq \sigma \Longrightarrow \tau < Collection\ \sigma$
$\tau = OrderedSet\ \varrho \Longrightarrow \varrho \leq \sigma \Longrightarrow \tau < Collection\ \sigma$
$\tau = Bag\ \varrho \Longrightarrow \varrho \leq \sigma \Longrightarrow \tau < Collection\ \sigma$
$\tau = Sequence\ \varrho \Longrightarrow \varrho \leq \sigma \Longrightarrow \tau < Collection\ \sigma$
unfolding *less-type-def less-eq-type-def*
apply *simp-all*
apply (*rule-tac ?f= Collection* **in** *preserve-tranclp*; *auto*)
apply (*rule preserve-rtranclp''*; *auto*)
apply (*rule preserve-rtranclp''*; *auto*)
apply (*rule preserve-rtranclp''*; *auto*)
by (*rule preserve-rtranclp''*; *auto*)

lemma *type-less-x-Set-intro* [*intro*]:
$\tau = Set\ \varrho \Longrightarrow \varrho < \sigma \Longrightarrow \tau < Set\ \sigma$
unfolding *less-type-def*
by *simp* (*rule preserve-tranclp*; *auto*)

lemma *type-less-x-OrderedSet-intro* [*intro*]:
$\tau = OrderedSet\ \varrho \Longrightarrow \varrho < \sigma \Longrightarrow \tau < OrderedSet\ \sigma$
unfolding *less-type-def*
by *simp* (*rule preserve-tranclp*; *auto*)

lemma *type-less-x-Bag-intro* [*intro*]:
  $\tau = Bag\ \varrho \Longrightarrow \varrho < \sigma \Longrightarrow \tau < Bag\ \sigma$
  unfolding *less-type-def*
  by *simp* (*rule preserve-tranclp*; *auto*)

lemma *type-less-x-Sequence-intro* [*intro*]:
  $\tau = Sequence\ \varrho \Longrightarrow \varrho < \sigma \Longrightarrow \tau < Sequence\ \sigma$
  unfolding *less-type-def*
  by *simp* (*rule preserve-tranclp*; *auto*)

lemma *fun-or-eq-refl* [*intro*]:
  $reflp\ (\lambda x\ y.\ f\ x\ y \vee x = y)$
  by (*simp add*: *reflpI*)

lemma *type-less-x-Tuple-intro* [*intro*]:
  assumes $\tau = Tuple\ \pi$
    and $strict\text{-}subtuple\ (\le)\ \pi\ \xi$
   shows $\tau < Tuple\ \xi$
proof −
  have $subtuple\ (\lambda\tau\ \sigma.\ \tau \sqsubset \sigma \vee \tau = \sigma)^{**}\ \pi\ \xi$
   using *assms*(*2*) *less-eq-type-def* **by** *auto*
  hence $(subtuple\ (\lambda\tau\ \sigma.\ \tau \sqsubset \sigma \vee \tau = \sigma))^{++}\ \pi\ \xi$
   by *simp* (*rule subtuple-to-trancl*; *auto*)
  hence $(strict\text{-}subtuple\ (\lambda\tau\ \sigma.\ \tau \sqsubset \sigma \vee \tau = \sigma))^{**}\ \pi\ \xi$
   by (*simp add*: *tranclp-into-rtranclp*)
  hence $(strict\text{-}subtuple\ (\lambda\tau\ \sigma.\ \tau \sqsubset \sigma \vee \tau = \sigma))^{++}\ \pi\ \xi$
   by (*meson assms*(*2*) *rtranclpD*)
  thus *?thesis*
   unfolding *less-type-def*
   using *assms*(*1*) **apply** *simp*
   by (*rule preserve-tranclp*; *auto*)
qed

lemma *type-less-x-OclSuper-intro* [*intro*]:
  $\tau \neq OclSuper \Longrightarrow \tau < OclSuper$
  unfolding *less-type-def*
proof (*induct* τ)
  case *OclSuper* **thus** *?case* **by** *auto*
next
  case (*Required* τ)
  have $subtype^{**}\ \tau[1]\ OclAny[1]$
   apply (*rule-tac ?f= Required* **in** *preserve-rtranclp*[*of basic-subtype*], *auto*)
   by (*metis less-eq-basic-type-def type-less-eq-x-OclAny-intro*)
  also have $subtype^{++}\ OclAny[1]\ OclAny[?]$ **by** *auto*
  also have $subtype^{++}\ OclAny[?]\ OclSuper$ **by** *auto*
  finally show *?case* **by** *auto*
next
  case (*Optional* τ)

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

3.3.2 Strict Elimination Rules

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

lemma *type-less-x-Optional* [*elim*!]:
  $\tau < \sigma[?] \Longrightarrow$
  $(\bigwedge\varrho.\ \tau = \varrho[1] \Longrightarrow \varrho \le \sigma \Longrightarrow P) \Longrightarrow$
  $(\bigwedge\varrho.\ \tau = \varrho[?] \Longrightarrow \varrho < \sigma \Longrightarrow P) \Longrightarrow P$
  unfolding *less-type-def*
  apply (*induct rule*: *converse-tranclp-induct*)
  apply (*metis subtype-x-Optional eq-refl less-basic-type-def tranclp.r-into-trancl*)
  apply (*erule subtype.cases*; *auto*)
  apply (*simp add*: *converse-rtranclp-into-rtranclp less-eq-basic-type-def*)
  by (*simp add*: *less-basic-type-def tranclp-into-tranclp2*)

lemma *type-less-x-Collection* [*elim*!]:
  $\tau <$ *Collection* $\sigma \Longrightarrow$
  $(\bigwedge\varrho.\ \tau =$ *Collection* $\varrho \Longrightarrow \varrho < \sigma \Longrightarrow P) \Longrightarrow$
  $(\bigwedge\varrho.\ \tau =$ *Set* $\varrho \Longrightarrow \varrho \le \sigma \Longrightarrow P) \Longrightarrow$
  $(\bigwedge\varrho.\ \tau =$ *OrderedSet* $\varrho \Longrightarrow \varrho \le \sigma \Longrightarrow P) \Longrightarrow$
  $(\bigwedge\varrho.\ \tau =$ *Bag* $\varrho \Longrightarrow \varrho \le \sigma \Longrightarrow P) \Longrightarrow$
  $(\bigwedge\varrho.\ \tau =$ *Sequence* $\varrho \Longrightarrow \varrho \le \sigma \Longrightarrow P) \Longrightarrow P$
  unfolding *less-type-def*
  apply (*induct rule*: *converse-tranclp-induct*)
  apply (*metis* (*mono-tags*) *Nitpick.rtranclp-unfold*
      subtype-x-Collection less-eq-type-def tranclp.r-into-trancl)
  by (*erule subtype.cases*;
    auto simp add: *converse-rtranclp-into-rtranclp less-eq-type-def*
        tranclp-into-tranclp2 tranclp-into-rtranclp)

lemma *type-less-x-Set* [*elim*!]:
  assumes $\tau <$ *Set* σ
    and $\bigwedge\varrho.\ \tau =$ *Set* $\varrho \Longrightarrow \varrho < \sigma \Longrightarrow P$
  shows P
proof –
  from *assms*(*1*) **obtain** ϱ **where** $\tau =$ *Set* ϱ
    unfolding *less-type-def*

```
    by (induct rule: converse-tranclp-induct; auto)
  moreover have ⋀τ σ. Set τ < Set σ ⟹ τ < σ
    unfolding less-type-def
    by (rule reflect-tranclp; auto)
  ultimately show ?thesis
    using assms by auto
qed

lemma type-less-x-OrderedSet [elim!]:
  assumes τ < OrderedSet σ
      and ⋀ϱ. τ = OrderedSet ϱ ⟹ ϱ < σ ⟹ P
    shows P
proof −
  from assms(1) obtain ϱ where τ = OrderedSet ϱ
    unfolding less-type-def
    by (induct rule: converse-tranclp-induct; auto)
  moreover have ⋀τ σ. OrderedSet τ < OrderedSet σ ⟹ τ < σ
    unfolding less-type-def
    by (rule reflect-tranclp; auto)
  ultimately show ?thesis
    using assms by auto
qed

lemma type-less-x-Bag [elim!]:
  assumes τ < Bag σ
      and ⋀ϱ. τ = Bag ϱ ⟹ ϱ < σ ⟹ P
    shows P
proof −
  from assms(1) obtain ϱ where τ = Bag ϱ
    unfolding less-type-def
    by (induct rule: converse-tranclp-induct; auto)
  moreover have ⋀τ σ. Bag τ < Bag σ ⟹ τ < σ
    unfolding less-type-def
    by (rule reflect-tranclp; auto)
  ultimately show ?thesis
    using assms by auto
qed

lemma type-less-x-Sequence [elim!]:
  assumes τ < Sequence σ
      and ⋀ϱ. τ = Sequence ϱ ⟹ ϱ < σ ⟹ P
    shows P
proof −
  from assms(1) obtain ϱ where τ = Sequence ϱ
    unfolding less-type-def
    by (induct rule: converse-tranclp-induct; auto)
  moreover have ⋀τ σ. Sequence τ < Sequence σ ⟹ τ < σ
    unfolding less-type-def
    by (rule reflect-tranclp; auto)
```

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

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

lemma *type-less-x-Tuple′*:
assumes $\tau <$ *Tuple* ξ
and *acyclicP-on* (*fmran′* ξ) *subtype*
and $\bigwedge\pi.\ \tau = Tuple\ \pi \Longrightarrow strict\text{-}subtuple\ (\le)\ \pi\ \xi \Longrightarrow P$
shows P
proof −
from *assms*(*1*) **obtain** π **where** $\tau = Tuple\ \pi$
unfolding *less-type-def*
by (*induct rule*: *converse-tranclp-induct*; *auto*)
moreover from *assms*(*2*) **have**
$\bigwedge\pi.\ Tuple\ \pi < Tuple\ \xi \Longrightarrow strict\text{-}subtuple\ (\le)\ \pi\ \xi$
unfolding *less-type-def less-eq-type-def*
by (*rule-tac ?f= Tuple* **in** *strict-subtuple-rtranclp-intro*; *auto*)
ultimately show *?thesis*
using *assms* **by** *auto*
qed

lemma *type-less-x-OclSuper* [*elim*!]:
$\tau < OclSuper \Longrightarrow (\tau \ne OclSuper \Longrightarrow P) \Longrightarrow P$
unfolding *less-type-def*
by (*drule tranclpD*, *auto*)

3.3.3 Properties

lemma *subtype-irrefl*:
$\tau < \tau \Longrightarrow False$
for τ :: *′a type*
apply (*induct* τ, *auto*)
apply (*erule type-less-x-Tuple′*, *auto*)
unfolding *less-type-def tranclp-unfold*
by *auto*

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

lemma *less-le-not-le-type*:
$\tau < \sigma \longleftrightarrow \tau \le \sigma \land \neg\ \sigma \le \tau$
for $\tau\ \sigma$ :: *′a type*
proof
show $\tau < \sigma \Longrightarrow \tau \le \sigma \land \neg\ \sigma \le \tau$

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

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

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

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

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

end

3.3.4 Non-Strict Introduction Rules

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

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

lemma *type-less-eq-x-Collection-intro* [*intro*]:
$\tau = \mathit{Collection}\ \varrho \Longrightarrow \varrho \leq \sigma \Longrightarrow \tau \leq \mathit{Collection}\ \sigma$
$\tau = \mathit{Set}\ \varrho \Longrightarrow \varrho \leq \sigma \Longrightarrow \tau \leq \mathit{Collection}\ \sigma$
$\tau = \mathit{OrderedSet}\ \varrho \Longrightarrow \varrho \leq \sigma \Longrightarrow \tau \leq \mathit{Collection}\ \sigma$
$\tau = \mathit{Bag}\ \varrho \Longrightarrow \varrho \leq \sigma \Longrightarrow \tau \leq \mathit{Collection}\ \sigma$

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

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

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

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

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

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

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

3.3.5 Non-Strict Elimination Rules

lemma *type-less-eq-x-Required* [*elim!*]:
$\tau \leq \sigma[1] \Longrightarrow$
$(\bigwedge\varrho.\ \tau = \varrho[1] \Longrightarrow \varrho \leq \sigma \Longrightarrow P) \Longrightarrow P$
by (*drule le-imp-less-or-eq*; *auto*)

lemma *type-less-eq-x-Optional* [*elim!*]:
$\tau \leq \sigma[?] \Longrightarrow$
$(\bigwedge\varrho.\ \tau = \varrho[1] \Longrightarrow \varrho \leq \sigma \Longrightarrow P) \Longrightarrow$
$(\bigwedge\varrho.\ \tau = \varrho[?] \Longrightarrow \varrho \leq \sigma \Longrightarrow P) \Longrightarrow P$
by (*drule le-imp-less-or-eq*, *auto*)

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

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

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

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

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

lemma *type-less-x-Tuple* [*elim*!]:
$\tau <$ *Tuple* $\xi \Longrightarrow$
$(\bigwedge\pi.\ \tau = \mathit{Tuple}\ \pi \Longrightarrow \mathit{strict\text{-}subtuple}\ (\leq)\ \pi\ \xi \Longrightarrow P) \Longrightarrow P$
apply (*erule type-less-x-Tuple′*)
by (*meson acyclic-def subtype-acyclic*)

lemma *type-less-eq-x-Tuple* [*elim*!]:
$\tau \leq$ *Tuple* $\xi \Longrightarrow$
$(\bigwedge\pi.\ \tau = \mathit{Tuple}\ \pi \Longrightarrow \mathit{subtuple}\ (\leq)\ \pi\ \xi \Longrightarrow P) \Longrightarrow P$
apply (*drule le-imp-less-or-eq*, *auto*)
by (*simp add*: *fmap.rel-refl fmrel-to-subtuple*)

3.3.6 Simplification Rules

lemma *type-less-left-simps* [*simp*]:
OclSuper $< \sigma =$ *False*
$\varrho[1] < \sigma = (\exists\upsilon.$
$\sigma = \mathit{OclSuper}\ \vee$
$\sigma = \upsilon[1] \wedge \varrho < \upsilon\ \vee$
$\sigma = \upsilon[?] \wedge \varrho \leq \upsilon)$
$\varrho[?] < \sigma = (\exists\upsilon.$
$\sigma = \mathit{OclSuper}\ \vee$
$\sigma = \upsilon[?] \wedge \varrho < \upsilon)$
Collection $\tau < \sigma = (\exists\varphi.$
$\sigma = \mathit{OclSuper}\ \vee$
$\sigma = \mathit{Collection}\ \varphi \wedge \tau < \varphi)$
Set $\tau < \sigma = (\exists\varphi.$
$\sigma = \mathit{OclSuper}\ \vee$
$\sigma = \mathit{Collection}\ \varphi \wedge \tau \leq \varphi\ \vee$

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

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

3.4 Upper Semilattice of Types

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

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

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

lemma sup-ge1-type:
  τ ≤ τ ⊔ σ
  for τ σ :: 'a type
proof (induct τ arbitrary: σ)
  case OclSuper show ?case by simp
  case (Required τ) show ?case by (induct σ; auto)
  case (Optional τ) show ?case by (induct σ; auto)
  case (Collection τ) thus ?case by (induct σ; auto)
  case (Set τ) thus ?case by (induct σ; auto)
```

```
  case (OrderedSet τ) thus ?case by (induct σ; auto)
  case (Bag τ) thus ?case by (induct σ; auto)
  case (Sequence τ) thus ?case by (induct σ; auto)
next
  case (Tuple π)
  moreover have Tuple-less-eq-sup:
    (⋀τ σ. τ ∈ fmran' π ⟹ τ ≤ τ ⊔ σ) ⟹
    Tuple π ≤ Tuple π ⊔ σ
   by (cases σ, auto)
  ultimately show ?case by (cases σ, auto)
qed

lemma sup-commut-type:
  τ ⊔ σ = σ ⊔ τ
  for τ σ :: 'a type
proof (induct τ arbitrary: σ)
  case OclSuper show ?case by (cases σ; simp add: less-eq-type-def)
  case (Required τ) show ?case by (cases σ; simp add: sup-commute)
  case (Optional τ) show ?case by (cases σ; simp add: sup-commute)
  case (Collection τ) thus ?case by (cases σ; simp)
  case (Set τ) thus ?case by (cases σ; simp)
  case (OrderedSet τ) thus ?case by (cases σ; simp)
  case (Bag τ) thus ?case by (cases σ; simp)
  case (Sequence τ) thus ?case by (cases σ; simp)
next
  case (Tuple π) thus ?case
    apply (cases σ; simp add: less-eq-type-def)
    using fmmerge-commut by blast
qed

lemma sup-least-type:
  τ ≤ ϱ ⟹ σ ≤ ϱ ⟹ τ ⊔ σ ≤ ϱ
  for τ σ ϱ :: 'a type
proof (induct ϱ arbitrary: τ σ)
  case OclSuper show ?case using eq-refl by auto
next
  case (Required x) show ?case
    apply (insert Required)
    by (erule type-less-eq-x-Required; erule type-less-eq-x-Required; auto)
next
  case (Optional x) show ?case
    apply (insert Optional)
    by (erule type-less-eq-x-Optional; erule type-less-eq-x-Optional; auto)
next
  case (Collection ϱ) show ?case
    apply (insert Collection)
    by (erule type-less-eq-x-Collection; erule type-less-eq-x-Collection; auto)
next
  case (Set ϱ) show ?case
```

apply (*insert Set*)
by (*erule type-less-eq-x-Set*; *erule type-less-eq-x-Set*; *auto*)
next
case (*OrderedSet* ϱ) **show** *?case*
apply (*insert OrderedSet*)
by (*erule type-less-eq-x-OrderedSet*; *erule type-less-eq-x-OrderedSet*; *auto*)
next
case (*Bag* ϱ) **show** *?case*
apply (*insert Bag*)
by (*erule type-less-eq-x-Bag*; *erule type-less-eq-x-Bag*; *auto*)
next
case (*Sequence* ϱ) **thus** *?case*
apply (*insert Sequence*)
by (*erule type-less-eq-x-Sequence*; *erule type-less-eq-x-Sequence*; *auto*)
next
case (*Tuple* π) **show** *?case*
apply (*insert Tuple*)
apply (*erule type-less-eq-x-Tuple*; *erule type-less-eq-x-Tuple*; *auto*)
by (*rule-tac ?π*= (*fmmerge* (⊔) π′ π′′) **in** *type-less-eq-x-Tuple-intro*;
simp add: *fmrel-on-fset-fmmerge1*)
qed

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

end

3.5 Helper Relations

abbreviation *between* (‹-/ = -—-› [*51*, *51*, *51*] *50*) **where**
$x = y{-}z \equiv y \le x \land x \le z$

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)

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

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* τ = τ

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 σ_1 $\Longrightarrow$
element-type τ σ_2 $\Longrightarrow$ $\sigma_1 = \sigma_2$
by (*induct rule*: *element-type.induct*; *simp add*: *element-type.simps*)

lemma *update-element-type-det*:
update-element-type τ σ ϱ_1 $\Longrightarrow$
update-element-type τ σ ϱ_2 $\Longrightarrow$ $\varrho_1 = \varrho_2$
by (*induct rule*: *update-element-type.induct*; *simp add*: *update-element-type.simps*)

lemma *to-unique-collection-det*:
to-unique-collection τ σ_1 $\Longrightarrow$
to-unique-collection τ σ_2 $\Longrightarrow$ $\sigma_1 = \sigma_2$
by (*induct rule*: *to-unique-collection.induct*; *simp add*: *to-unique-collection.simps*)

lemma *to-nonunique-collection-det*:
to-nonunique-collection τ σ_1 $\Longrightarrow$
to-nonunique-collection τ σ_2 $\Longrightarrow$ $\sigma_1 = \sigma_2$
by (*induct rule*: *to-nonunique-collection.induct*; *simp add*: *to-nonunique-collection.simps*)

lemma *to-ordered-collection-det*:
to-ordered-collection τ σ_1 $\Longrightarrow$
to-ordered-collection τ σ_2 $\Longrightarrow$ $\sigma_1 = \sigma_2$
by (*induct rule*: *to-ordered-collection.induct*; *simp add*: *to-ordered-collection.simps*)

3.7 Code Setup

code-pred *subtype* **.**

function *subtype-fun* :: *'a::order type* ⇒ *'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 ϱ ∨ $\tau = \varrho$
| - ⇒ *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)
  by pat-completeness auto
termination
  by (relation measure (λ(xs, ys). size ys) ;
      auto simp add: elem-le-ffold' fmran'I)

lemma less-type-code [code]:
  (<) = subtype-fun
proof (intro ext iffI)
  fix τ σ :: 'a type
  show τ < σ ⟹ subtype-fun τ σ
  proof (induct τ arbitrary: σ)
    case OclSuper thus ?case by (cases σ; auto)
  next
    case (Required τ) thus ?case
      by (cases σ; auto simp: less-basic-type-code less-eq-basic-type-code)
  next
    case (Optional τ) thus ?case
      by (cases σ; auto simp: less-basic-type-code less-eq-basic-type-code)
```

```
next
  case (Collection τ) thus ?case by (cases σ; auto)
next
  case (Set τ) thus ?case by (cases σ; auto)
next
  case (OrderedSet τ) thus ?case by (cases σ; auto)
next
  case (Bag τ) thus ?case by (cases σ; auto)
next
  case (Sequence τ) thus ?case by (cases σ; auto)
next
  case (Tuple π)
  have
     ⋀ξ. subtuple (≤) π ξ ⟶
     subtuple (λτ σ. subtype-fun τ σ ∨ τ = σ) π ξ
    by (rule subtuple-mono; auto simp add: Tuple.hyps)
  with Tuple.prems show ?case by (cases σ; auto)
qed
show  subtype-fun τ σ ⟹ τ < σ
proof (induct σ arbitrary: τ)
  case OclSuper thus ?case by (cases σ; auto)
next
  case (Required σ) show ?case
    by (insert Required) (erule subtype-fun.elims;
        auto simp: less-basic-type-code less-eq-basic-type-code)
next
  case (Optional σ) show ?case
    by (insert Optional) (erule subtype-fun.elims;
        auto simp: less-basic-type-code less-eq-basic-type-code)
next
  case (Collection σ) show ?case
    apply (insert Collection)
    apply (erule subtype-fun.elims; auto)
    using order.strict-implies-order by auto
next
  case (Set σ) show ?case
    by (insert Set) (erule subtype-fun.elims; auto)
next
  case (OrderedSet σ) show ?case
    by (insert OrderedSet) (erule subtype-fun.elims; auto)
next
  case (Bag σ) show ?case
    by (insert Bag) (erule subtype-fun.elims; auto)
next
  case (Sequence σ) show ?case
    by (insert Sequence) (erule subtype-fun.elims; auto)
next
  case (Tuple ξ)
  have subtuple-imp-simp:
```

$\bigwedge\pi$. *subtuple* ($\lambda\tau$ σ. *subtype-fun* τ σ $\vee$ $\tau = \sigma$) π ξ $\longrightarrow$
subtuple ($\leq$) π ξ
by (*rule subtuple-mono*; *auto simp add*: *Tuple.hyps less-imp-le*)
show *?case*
apply (*insert Tuple*)
by (*erule subtype-fun.elims*; *auto simp add*: *subtuple-imp-simp*)
qed
qed

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

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

end

Chapter 4

Abstract Syntax

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

4.1 Preliminaries

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

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

typedef *unat* = *UNIV* :: *nat option set* **..**

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

instantiation *unat* :: *infinity*
begin
definition ∞ ≡ *Abs-unat None*
instance ..
end

free-constructors *cases-unat* **for**
  unat
| ∞ :: *unat*
  unfolding *unat-def infinity-unat-def*
  apply (*metis Rep-unat-inverse option.collapse*)
  apply (*metis Abs-unat-inverse UNIV-I option.sel*)
  by (*simp add*: *Abs-unat-inject*)

4.2 Standard Library Operations

datatype *metaop* = *AllInstancesOp*

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

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

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

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

datatype *numeric-unop* = *UMinusOp* | *AbsOp* | *FloorOp* | *RoundOp* | *ToIntegerOp*
datatype *numeric-binop* = *PlusOp* | *MinusOp* | *MultOp* | *DivideOp*
| *DivOp* | *ModOp* | *MaxOp* | *MinOp*
| *LessOp* | *LessEqOp* | *GreaterOp* | *GreaterEqOp*

datatype *string-unop* = *SizeOp* | *ToUpperCaseOp* | *ToLowerCaseOp* | *Character-sOp*
| *ToBooleanOp* | *ToIntegerOp* | *ToRealOp*
datatype *string-binop* = *ConcatOp* | *IndexOfOp* | *EqualsIgnoreCaseOp* | *AtOp*
| *LessOp* | *LessEqOp* | *GreaterOp* | *GreaterEqOp*
datatype *string-ternop* = *SubstringOp*

datatype *collection-unop* = *CollectionSizeOp* | *IsEmptyOp* | *NotEmptyOp*
| *CollectionMaxOp* | *CollectionMinOp* | *SumOp*
| *AsSetOp* | *AsOrderedSetOp* | *AsSequenceOp* | *AsBagOp* | *FlattenOp*
| *FirstOp* | *LastOp* | *ReverseOp*
datatype *collection-binop* = *IncludesOp* | *ExcludesOp*
| *CountOp*| *IncludesAllOp* | *ExcludesAllOp* | *ProductOp*
| *UnionOp* | *IntersectionOp* | *SetMinusOp* | *SymmetricDifferenceOp*
| *IncludingOp* | *ExcludingOp*
| *AppendOp* | *PrependOp* | *CollectionAtOp* | *CollectionIndexOfOp*
datatype *collection-ternop* = *InsertAtOp* | *SubOrderedSetOp* | *SubSequenceOp*

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

declare [[*coercion* *Inl* :: *any-unop* ⇒ *unop*]]
declare [[*coercion* *Inr* ∘ *Inl* :: *boolean-unop* ⇒ *unop*]]
declare [[*coercion* *Inr* ∘ *Inr* ∘ *Inl* :: *numeric-unop* ⇒ *unop*]]
declare [[*coercion* *Inr* ∘ *Inr* ∘ *Inr* ∘ *Inl* :: *string-unop* ⇒ *unop*]]
declare [[*coercion* *Inr* ∘ *Inr* ∘ *Inr* ∘ *Inr* :: *collection-unop* ⇒ *unop*]]

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

declare [[*coercion* *Inl* :: *super-binop* ⇒ *binop*]]
declare [[*coercion* *Inr* ∘ *Inl* :: *boolean-binop* ⇒ *binop*]]
declare [[*coercion* *Inr* ∘ *Inr* ∘ *Inl* :: *numeric-binop* ⇒ *binop*]]

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

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

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

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

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

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

declare [[*coercion Inr* ∘ *Inr* ∘ *Inl* ∘ *Inl* :: *string-ternop* ⇒ *op*]]
declare [[*coercion Inr* ∘ *Inr* ∘ *Inl* ∘ *Inr* :: *collection-ternop* ⇒ *op*]]

declare [[*coercion Inr* ∘ *Inr* ∘ *Inr* :: *oper* ⇒ *op*]]

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

4.3 Expressions

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

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

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

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

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

Please take a note that for *AssociationEnd* and *AssociationClass* call expressions one can specify an optional role of a source class (*from-role*). It differs from the OCL specification, which allows one to specify a role of a destination class. However, the latter one does not allow one to determine uniquely a set of linked objects, for example, in a ternary self relation.

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

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

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

abbreviation *TypeOperationCall src k op ty* ≡
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* (*λp.* (*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* :: $assoc \rightharpoonup_f role \rightharpoonup_f 'a\ assoc\text{-}end$
    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*:
    $assoc\ |\in|\ fmdom\ associations \Longrightarrow$
    $fmlookup\ associations\ assoc = Some\ ends \Longrightarrow$
    $role\ |\in|\ fmdom\ ends \Longrightarrow$
    $fmlookup\ ends\ role = Some\ end \Longrightarrow$
    $assoc\text{-}end\text{-}min\ end \leq assoc\text{-}end\text{-}max\ end$
  assumes *association-ends-unique*:
    $association\text{-}ends'\ classes\ associations\ \mathcal{C}\ from\ role\ end_1 \Longrightarrow$
    $association\text{-}ends'\ classes\ associations\ \mathcal{C}\ from\ role\ end_2 \Longrightarrow end_1 = end_2$
begin

interpretation *base*: *object-model*
 by *standard* (*simp-all add*: *local.assoc-end-min-less-eq-max local.association-ends-unique*)

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**
$\sigma < \tau \lor \tau < \sigma \Longrightarrow$
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* ⇒ (*'a* :: *order*) *type* ⇒ *'a type* ⇒ *'a type* ⇒ *bool* **where**
$\tau \le \sigma \lor \sigma < \tau \Longrightarrow$
super-binop-type EqualOp $\tau\ \sigma$ *Boolean*[*1*]
| $\tau \le \sigma \lor \sigma < \tau \Longrightarrow$
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**

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

inductive *boolean-binop-type* **where**
$\tau \sqcup \sigma = \varrho \Longrightarrow \varrho \leq \mathit{Boolean}[?] \Longrightarrow$
boolean-binop-type AndOp τ σ ϱ
| $\tau \sqcup \sigma = \varrho \Longrightarrow \varrho \leq \mathit{Boolean}[?] \Longrightarrow$
boolean-binop-type OrOp τ σ ϱ
| $\tau \sqcup \sigma = \varrho \Longrightarrow \varrho \leq \mathit{Boolean}[?] \Longrightarrow$
boolean-binop-type XorOp τ σ ϱ
| $\tau \sqcup \sigma = \varrho \Longrightarrow \varrho \leq \mathit{Boolean}[?] \Longrightarrow$
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**
$\tau = \mathit{Real}[1] \Longrightarrow$
numeric-unop-type UMinusOp τ *Real*[*1*]
| $\tau = \mathit{UnlimitedNatural}[1]\text{--}\mathit{Integer}[1] \Longrightarrow$

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

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

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

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

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

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

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

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

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

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

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

6.1.7 String Operations

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

inductive *string-binop-type* **where**
string-binop-type ConcatOp String[*1*] *String*[*1*] *String*[*1*]
| *string-binop-type EqualsIgnoreCaseOp String*[*1*] *String*[*1*] *Boolean*[*1*]
| *string-binop-type LessOp String*[*1*] *String*[*1*] *Boolean*[*1*]
| *string-binop-type LessEqOp String*[*1*] *String*[*1*] *Boolean*[*1*]
| *string-binop-type GreaterOp String*[*1*] *String*[*1*] *Boolean*[*1*]
| *string-binop-type GreaterEqOp String*[*1*] *String*[*1*] *Boolean*[*1*]
| *string-binop-type IndexOfOp String*[*1*] *String*[*1*] *Integer*[*1*]
| 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*] $\Longrightarrow$

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

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:

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

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**
$element\text{-}type\ \tau\ \varrho \Longrightarrow \sigma \le to\text{-}optional\text{-}type\text{-}nested\ \varrho \Longrightarrow$
$collection\text{-}binop\text{-}type\ IncludesOp\ \tau\ \sigma\ Boolean[1]$
$|\ element\text{-}type\ \tau\ \varrho \Longrightarrow \sigma \le to\text{-}optional\text{-}type\text{-}nested\ \varrho \Longrightarrow$
$collection\text{-}binop\text{-}type\ ExcludesOp\ \tau\ \sigma\ Boolean[1]$
$|\ element\text{-}type\ \tau\ \varrho \Longrightarrow \sigma \le to\text{-}optional\text{-}type\text{-}nested\ \varrho \Longrightarrow$
$collection\text{-}binop\text{-}type\ CountOp\ \tau\ \sigma\ Integer[1]$

$|\ element\text{-}type\ \tau\ \varrho \Longrightarrow element\text{-}type\ \sigma\ \upsilon \Longrightarrow \upsilon \le to\text{-}optional\text{-}type\text{-}nested\ \varrho \Longrightarrow$
$collection\text{-}binop\text{-}type\ IncludesAllOp\ \tau\ \sigma\ Boolean[1]$
$|\ element\text{-}type\ \tau\ \varrho \Longrightarrow element\text{-}type\ \sigma\ \upsilon \Longrightarrow \upsilon \le to\text{-}optional\text{-}type\text{-}nested\ \varrho \Longrightarrow$
$collection\text{-}binop\text{-}type\ ExcludesAllOp\ \tau\ \sigma\ Boolean[1]$

$|\ element\text{-}type\ \tau\ \varrho \Longrightarrow element\text{-}type\ \sigma\ \upsilon \Longrightarrow$
$collection\text{-}binop\text{-}type\ ProductOp\ \tau\ \sigma$
$(Set\ (Tuple\ (fmap\text{-}of\text{-}list\ [(STR\ ''first'',\ \varrho),\ (STR\ ''second'',\ \upsilon)])))$

$|\ collection\text{-}binop\text{-}type\ UnionOp\ (Set\ \tau)\ (Set\ \sigma)\ (Set\ (\tau \sqcup \sigma))$
$|\ collection\text{-}binop\text{-}type\ UnionOp\ (Set\ \tau)\ (Bag\ \sigma)\ (Bag\ (\tau \sqcup \sigma))$
$|\ collection\text{-}binop\text{-}type\ UnionOp\ (Bag\ \tau)\ (Set\ \sigma)\ (Bag\ (\tau \sqcup \sigma))$
$|\ collection\text{-}binop\text{-}type\ UnionOp\ (Bag\ \tau)\ (Bag\ \sigma)\ (Bag\ (\tau \sqcup \sigma))$

$|\ collection\text{-}binop\text{-}type\ IntersectionOp\ (Set\ \tau)\ (Set\ \sigma)\ (Set\ (\tau \sqcup \sigma))$
$|\ collection\text{-}binop\text{-}type\ IntersectionOp\ (Set\ \tau)\ (Bag\ \sigma)\ (Set\ (\tau \sqcup \sigma))$
$|\ collection\text{-}binop\text{-}type\ IntersectionOp\ (Bag\ \tau)\ (Set\ \sigma)\ (Set\ (\tau \sqcup \sigma))$
$|\ collection\text{-}binop\text{-}type\ IntersectionOp\ (Bag\ \tau)\ (Bag\ \sigma)\ (Bag\ (\tau \sqcup \sigma))$

$|\ collection\text{-}binop\text{-}type\ SetMinusOp\ (Set\ \tau)\ (Set\ \sigma)\ (Set\ \tau)$
$|\ collection\text{-}binop\text{-}type\ SymmetricDifferenceOp\ (Set\ \tau)\ (Set\ \sigma)\ (Set\ (\tau \sqcup \sigma))$

$|\ element\text{-}type\ \tau\ \varrho \Longrightarrow update\text{-}element\text{-}type\ \tau\ (\varrho \sqcup \sigma)\ \upsilon \Longrightarrow$
$collection\text{-}binop\text{-}type\ IncludingOp\ \tau\ \sigma\ \upsilon$
$|\ element\text{-}type\ \tau\ \varrho \Longrightarrow \sigma \le \varrho \Longrightarrow$
$collection\text{-}binop\text{-}type\ ExcludingOp\ \tau\ \sigma\ \tau$

$|\ \sigma \le \tau \Longrightarrow$
$collection\text{-}binop\text{-}type\ AppendOp\ (OrderedSet\ \tau)\ \sigma\ (OrderedSet\ \tau)$
$|\ \sigma \le \tau \Longrightarrow$
$collection\text{-}binop\text{-}type\ AppendOp\ (Sequence\ \tau)\ \sigma\ (Sequence\ \tau)$
$|\ \sigma \le \tau \Longrightarrow$
$collection\text{-}binop\text{-}type\ PrependOp\ (OrderedSet\ \tau)\ \sigma\ (OrderedSet\ \tau)$
$|\ \sigma \le \tau \Longrightarrow$
$collection\text{-}binop\text{-}type\ PrependOp\ (Sequence\ \tau)\ \sigma\ (Sequence\ \tau)$

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

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

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

6.1.9 Coercions

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

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

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 $\Longrightarrow$ $\varrho_1 = \varrho_2$
 by (*induct rule*: *typeop-type.induct*;
   auto simp add: *typeop-type.simps update-element-type-det*)

lemma *any-unop-type-det*:
  any-unop-type op τ $\sigma_1 \Longrightarrow$
  any-unop-type op τ $\sigma_2 \Longrightarrow \sigma_1 = \sigma_2$
 by (*induct rule*: *any-unop-type.induct*; *simp add*: *any-unop-type.simps*)

lemma *boolean-unop-type-det*:
  boolean-unop-type op τ $\sigma_1 \Longrightarrow$
  boolean-unop-type op τ $\sigma_2 \Longrightarrow \sigma_1 = \sigma_2$
 by (*induct rule*: *boolean-unop-type.induct*; *simp add*: *boolean-unop-type.simps*)

lemma *numeric-unop-type-det*:
  numeric-unop-type op τ $\sigma_1 \Longrightarrow$
  numeric-unop-type op τ $\sigma_2 \Longrightarrow \sigma_1 = \sigma_2$
 by (*induct rule*: *numeric-unop-type.induct*; *auto simp add*: *numeric-unop-type.simps*)

lemma *string-unop-type-det*:
  string-unop-type op τ $\sigma_1 \Longrightarrow$
  string-unop-type op τ $\sigma_2 \Longrightarrow \sigma_1 = \sigma_2$
 by (*induct rule*: *string-unop-type.induct*; *simp add*: *string-unop-type.simps*)

lemma *collection-unop-type-det*:
  collection-unop-type op τ $\sigma_1 \Longrightarrow$
  collection-unop-type op τ $\sigma_2 \Longrightarrow \sigma_1 = \sigma_2$
 apply (*induct rule*: *collection-unop-type.induct*)
 by (*erule collection-unop-type.cases*;
   auto simp add: *element-type-det update-element-type-det*)+

lemma *unop-type-det*:
  unop-type op k τ $\sigma_1 \Longrightarrow$
  unop-type op k τ $\sigma_2 \Longrightarrow \sigma_1 = \sigma_2$
 by (*induct rule*: *unop-type.induct*;
   simp add: *unop-type.simps any-unop-type-det*
     boolean-unop-type-det numeric-unop-type-det
     string-unop-type-det collection-unop-type-det)

lemma *super-binop-type-det*:
  super-binop-type op τ σ $\varrho_1 \Longrightarrow$
  super-binop-type op τ σ $\varrho_2 \Longrightarrow \varrho_1 = \varrho_2$
 by (*induct rule*: *super-binop-type.induct*; *auto simp add*: *super-binop-type.simps*)

lemma *boolean-binop-type-det*:
  boolean-binop-type op τ σ $\varrho_1 \Longrightarrow$
  boolean-binop-type op τ σ $\varrho_2 \Longrightarrow \varrho_1 = \varrho_2$
 by (*induct rule*: *boolean-binop-type.induct*; *simp add*: *boolean-binop-type.simps*)

lemma *numeric-binop-type-det*:
  numeric-binop-type op τ σ $\varrho_1 \Longrightarrow$
  numeric-binop-type op τ σ $\varrho_2 \Longrightarrow \varrho_1 = \varrho_2$
 by (*induct rule*: *numeric-binop-type.induct*;

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

lemma *string-binop-type-det*:
$\mathit{string\text{-}binop\text{-}type}\ op\ \tau\ \sigma\ \varrho_1 \Longrightarrow$
$\mathit{string\text{-}binop\text{-}type}\ op\ \tau\ \sigma\ \varrho_2 \Longrightarrow \varrho_1 = \varrho_2$
by (*induct rule*: *string-binop-type.induct*; *simp add*: *string-binop-type.simps*)

lemma *collection-binop-type-det*:
$\mathit{collection\text{-}binop\text{-}type}\ op\ \tau\ \sigma\ \varrho_1 \Longrightarrow$
$\mathit{collection\text{-}binop\text{-}type}\ op\ \tau\ \sigma\ \varrho_2 \Longrightarrow \varrho_1 = \varrho_2$
apply (*induct rule*: *collection-binop-type.induct*; *simp add*: *collection-binop-type.simps*)
using *element-type-det update-element-type-det* **by** *blast+*

lemma *binop-type-det*:
$\mathit{binop\text{-}type}\ op\ k\ \tau\ \sigma\ \varrho_1 \Longrightarrow$
$\mathit{binop\text{-}type}\ op\ k\ \tau\ \sigma\ \varrho_2 \Longrightarrow \varrho_1 = \varrho_2$
by (*induct rule*: *binop-type.induct*;
simp add: *binop-type.simps super-binop-type-det*
boolean-binop-type-det numeric-binop-type-det
string-binop-type-det collection-binop-type-det)

lemma *string-ternop-type-det*:
$\mathit{string\text{-}ternop\text{-}type}\ op\ \tau\ \sigma\ \varrho\ \upsilon_1 \Longrightarrow$
$\mathit{string\text{-}ternop\text{-}type}\ op\ \tau\ \sigma\ \varrho\ \upsilon_2 \Longrightarrow \upsilon_1 = \upsilon_2$
by (*induct rule*: *string-ternop-type.induct*; *simp add*: *string-ternop-type.simps*)

lemma *collection-ternop-type-det*:
$\mathit{collection\text{-}ternop\text{-}type}\ op\ \tau\ \sigma\ \varrho\ \upsilon_1 \Longrightarrow$
$\mathit{collection\text{-}ternop\text{-}type}\ op\ \tau\ \sigma\ \varrho\ \upsilon_2 \Longrightarrow \upsilon_1 = \upsilon_2$
by (*induct rule*: *collection-ternop-type.induct*; *simp add*: *collection-ternop-type.simps*)

lemma *ternop-type-det*:
$\mathit{ternop\text{-}type}\ op\ k\ \tau\ \sigma\ \varrho\ \upsilon_1 \Longrightarrow$
$\mathit{ternop\text{-}type}\ op\ k\ \tau\ \sigma\ \varrho\ \upsilon_2 \Longrightarrow \upsilon_1 = \upsilon_2$
by (*induct rule*: *ternop-type.induct*;
simp add: *ternop-type.simps string-ternop-type-det collection-ternop-type-det*)

lemma *op-type-det*:
$\mathit{op\text{-}type}\ op\ k\ \tau\ \pi\ \sigma \Longrightarrow$
$\mathit{op\text{-}type}\ op\ k\ \tau\ \pi\ \varrho \Longrightarrow \sigma = \varrho$
apply (*induct rule*: *op-type.induct*)
apply (*erule op-type.cases*; *simp add*: *unop-type-det*)
apply (*erule op-type.cases*; *simp add*: *binop-type-det*)
apply (*erule op-type.cases*; *simp add*: *ternop-type-det*)
by (*erule op-type.cases*; *simp*; *metis operation-det*)

6.2 Expressions Typing

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

inductive *typing* :: ($'a$ :: *ocl-object-model*) *type env* ⇒ $'a$ *expr* ⇒ $'a$ *type* ⇒ *bool*
(‹(*1*-/ $\vdash_E$/ (- :/ -))› [*51*,*51*,*51*] *50*)
and *collection-parts-typing* (‹(*1*-/ $\vdash_C$/ (- :/ -))› [*51*,*51*,*51*] *50*)
and *collection-part-typing* (‹(*1*-/ $\vdash_P$/ (- :/ -))› [*51*,*51*,*51*] *50*)
and *iterator-typing* (‹(*1*-/ $\vdash_I$/ (- :/ -))› [*51*,*51*,*51*] *50*)
and *expr-list-typing* (‹(*1*-/ $\vdash_L$/ (- :/ -))› [*51*,*51*,*51*] *50*) **where**

— Primitive Literals

NullLiteralT:
$\Gamma \vdash_E$ *NullLiteral* : *OclVoid*[*?*]
|*BooleanLiteralT*:
$\Gamma \vdash_E$ *BooleanLiteral c* : *Boolean*[*1*]
|*RealLiteralT*:
$\Gamma \vdash_E$ *RealLiteral c* : *Real*[*1*]
|*IntegerLiteralT*:
$\Gamma \vdash_E$ *IntegerLiteral c* : *Integer*[*1*]
|*UnlimitedNaturalLiteralT*:
$\Gamma \vdash_E$ *UnlimitedNaturalLiteral c* : *UnlimitedNatural*[*1*]
|*StringLiteralT*:
$\Gamma \vdash_E$ *StringLiteral c* : *String*[*1*]
|*EnumLiteralT*:
has-literal enum lit ⟹
$\Gamma \vdash_E$ *EnumLiteral enum lit* : (*Enum enum*)[*1*]

— Collection Literals

|*SetLiteralT*:
$\Gamma \vdash_C$ *prts* : τ ⟹
$\Gamma \vdash_E$ *CollectionLiteral SetKind prts* : *Set* τ
|*OrderedSetLiteralT*:
$\Gamma \vdash_C$ *prts* : τ ⟹
$\Gamma \vdash_E$ *CollectionLiteral OrderedSetKind prts* : *OrderedSet* τ
|*BagLiteralT*:
$\Gamma \vdash_C$ *prts* : τ ⟹
$\Gamma \vdash_E$ *CollectionLiteral BagKind prts* : *Bag* τ
|*SequenceLiteralT*:
$\Gamma \vdash_C$ *prts* : τ ⟹
$\Gamma \vdash_E$ *CollectionLiteral SequenceKind prts* : *Sequence* τ

— We prohibit empty collection literals, because their type is unclear. We could use *OclVoid*[*1*] element type for empty collections, but the typing rules will give wrong types for nested collections, because, for example, *OclVoid*[*1*] ⊔ *Set*(*Integer*[*1*]) = *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*:
$\Gamma \vdash_E \mathit{TupleLiteral}\ [] : \mathit{Tuple\ fmempty}$
|*TupleLiteralConsT*:
$\Gamma \vdash_E \mathit{TupleLiteral\ elems} : \mathit{Tuple}\ \xi \Longrightarrow$
$\Gamma \vdash_E \mathit{tuple\text{-}element\text{-}expr\ el} : \tau \Longrightarrow$
$\mathit{tuple\text{-}element\text{-}type\ el} = \mathit{Some}\ \sigma \Longrightarrow$
$\tau \leq \sigma \Longrightarrow$
$\Gamma \vdash_E \mathit{TupleLiteral}\ (el \# elems) : \mathit{Tuple}\ (\xi(\mathit{tuple\text{-}element\text{-}name\ el} \mapsto_f \sigma))$

— Misc Expressions

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

— Call Expressions

$|MetaOperationCallT$:
$\quad$ $mataop\text{-}type\ \tau\ op\ \sigma \Longrightarrow$
$\quad$ $\Gamma \vdash_E MetaOperationCall\ \tau\ op : \sigma$
$|StaticOperationCallT$:
$\quad$ $\Gamma \vdash_L params : \pi \Longrightarrow$
$\quad$ $static\text{-}operation\ \tau\ op\ \pi\ oper \Longrightarrow$
$\quad$ $\Gamma \vdash_E StaticOperationCall\ \tau\ op\ params : oper\text{-}type\ oper$

$|TypeOperationCallT$:
$\quad$ $\Gamma \vdash_E a : \tau \Longrightarrow$
$\quad$ $typeop\text{-}type\ k\ op\ \tau\ \sigma\ \varrho \Longrightarrow$
$\quad$ $\Gamma \vdash_E TypeOperationCall\ a\ k\ op\ \sigma : \varrho$

$|AttributeCallT$:
$\quad$ $\Gamma \vdash_E src : \langle\mathcal{C}\rangle_{\mathcal{T}}[1] \Longrightarrow$
$\quad$ $attribute\ \mathcal{C}\ prop\ \mathcal{D}\ \tau \Longrightarrow$
$\quad$ $\Gamma \vdash_E AttributeCall\ src\ DotCall\ prop : \tau$
$|AssociationEndCallT$:
$\quad$ $\Gamma \vdash_E src : \langle\mathcal{C}\rangle_{\mathcal{T}}[1] \Longrightarrow$
$\quad$ $association\text{-}end\ \mathcal{C}\ from\ role\ \mathcal{D}\ end \Longrightarrow$
$\quad$ $\Gamma \vdash_E AssociationEndCall\ src\ DotCall\ from\ role : assoc\text{-}end\text{-}type\ end$
$|AssociationClassCallT$:
$\quad$ $\Gamma \vdash_E src : \langle\mathcal{C}\rangle_{\mathcal{T}}[1] \Longrightarrow$
$\quad$ $referred\text{-}by\text{-}association\text{-}class\ \mathcal{C}\ from\ \mathcal{A}\ \mathcal{D} \Longrightarrow$
$\quad$ $\Gamma \vdash_E AssociationClassCall\ src\ DotCall\ from\ \mathcal{A} : class\text{-}assoc\text{-}type\ \mathcal{A}$
$|AssociationClassEndCallT$:
$\quad$ $\Gamma \vdash_E src : \langle\mathcal{A}\rangle_{\mathcal{T}}[1] \Longrightarrow$
$\quad$ $association\text{-}class\text{-}end\ \mathcal{A}\ role\ end \Longrightarrow$
$\quad$ $\Gamma \vdash_E AssociationClassEndCall\ src\ DotCall\ role : class\text{-}assoc\text{-}end\text{-}type\ end$
$|OperationCallT$:
$\quad$ $\Gamma \vdash_E src : \tau \Longrightarrow$
$\quad$ $\Gamma \vdash_L params : \pi \Longrightarrow$
$\quad$ $op\text{-}type\ op\ k\ \tau\ \pi\ \sigma \Longrightarrow$
$\quad$ $\Gamma \vdash_E OperationCall\ src\ k\ op\ params : \sigma$

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

— Iterator Expressions

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

$|$*IterateT*:
$\Gamma \vdash_I$ (*src*, *its*, *its-ty*, *Let res* (*Some res-t*) *res-init body*) : $(\tau, \sigma, \varrho) \Longrightarrow$
$\varrho \leq$ *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*) : $(\tau, \sigma, \varrho) \Longrightarrow$
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*) : $(\tau, \sigma, \varrho) \Longrightarrow$
length its $\leq$ *1* $\Longrightarrow$
to-single-type $\varrho \leq \sigma \Longrightarrow$
to-unique-collection $\tau\ \upsilon \Longrightarrow$
$\Gamma \vdash_E$ *ClosureIteratorCall src ArrowCall its its-ty body* : υ
$|$*CollectIteratorT*:
$\Gamma \vdash_I$ (*src*, *its*, *its-ty*, *body*) : $(\tau, \sigma, \varrho) \Longrightarrow$
length its $\leq$ *1* $\Longrightarrow$
to-nonunique-collection $\tau\ \upsilon \Longrightarrow$
update-element-type υ (*to-single-type* ϱ) $\varphi \Longrightarrow$
$\Gamma \vdash_E$ *CollectIteratorCall src ArrowCall its its-ty body* : φ
$|$*CollectNestedIteratorT*:
$\Gamma \vdash_I$ (*src*, *its*, *its-ty*, *body*) : $(\tau, \sigma, \varrho) \Longrightarrow$
length its $\leq$ *1* $\Longrightarrow$
to-nonunique-collection $\tau\ \upsilon \Longrightarrow$
update-element-type $\upsilon\ \varrho\ \varphi \Longrightarrow$
$\Gamma \vdash_E$ *CollectNestedIteratorCall src ArrowCall its its-ty body* : φ
$|$*ExistsIteratorT*:
$\Gamma \vdash_I$ (*src*, *its*, *its-ty*, *body*) : $(\tau, \sigma, \varrho) \Longrightarrow$
$\varrho \leq$ *Boolean*[*?*] $\Longrightarrow$
$\Gamma \vdash_E$ *ExistsIteratorCall src ArrowCall its its-ty body* : ϱ
$|$*ForAllIteratorT*:
$\Gamma \vdash_I$ (*src*, *its*, *its-ty*, *body*) : $(\tau, \sigma, \varrho) \Longrightarrow$
$\varrho \leq$ *Boolean*[*?*] $\Longrightarrow$
$\Gamma \vdash_E$ *ForAllIteratorCall src ArrowCall its its-ty body* : ϱ
$|$*OneIteratorT*:
$\Gamma \vdash_I$ (*src*, *its*, *its-ty*, *body*) : $(\tau, \sigma, \varrho) \Longrightarrow$
length its $\leq$ *1* $\Longrightarrow$
$\varrho \leq$ *Boolean*[*?*] $\Longrightarrow$
$\Gamma \vdash_E$ *OneIteratorCall src ArrowCall its its-ty body* : *Boolean*[*1*]
$|$*IsUniqueIteratorT*:
$\Gamma \vdash_I$ (*src*, *its*, *its-ty*, *body*) : $(\tau, \sigma, \varrho) \Longrightarrow$
length its $\leq$ *1* $\Longrightarrow$
$\Gamma \vdash_E$ *IsUniqueIteratorCall src ArrowCall its its-ty body* : *Boolean*[*1*]
$|$*SelectIteratorT*:
$\Gamma \vdash_I$ (*src*, *its*, *its-ty*, *body*) : $(\tau, \sigma, \varrho) \Longrightarrow$
length its $\leq$ *1* $\Longrightarrow$
$\varrho \leq$ *Boolean*[*?*] $\Longrightarrow$

$\Gamma \vdash_E$ *SelectIteratorCall src ArrowCall its its-ty body* : τ
|*RejectIteratorT*:
$\Gamma \vdash_I$ (*src*, *its*, *its-ty*, *body*) : $(\tau, \sigma, \varrho) \Longrightarrow$
length its $\leq$ *1* $\Longrightarrow$
$\varrho \leq$ *Boolean*[*?*] $\Longrightarrow$
$\Gamma \vdash_E$ *RejectIteratorCall src ArrowCall its its-ty body* : τ
|*SortedByIteratorT*:
$\Gamma \vdash_I$ (*src*, *its*, *its-ty*, *body*) : $(\tau, \sigma, \varrho) \Longrightarrow$
length its $\leq$ *1* $\Longrightarrow$
to-ordered-collection τ υ $\Longrightarrow$
$\Gamma \vdash_E$ *SortedByIteratorCall src ArrowCall its its-ty body* : υ

— Expression Lists

|*ExprListNilT*:
$\Gamma \vdash_L$ [] : []
|*ExprListConsT*:
$\Gamma \vdash_E$ *expr* : $\tau \Longrightarrow$
$\Gamma \vdash_L$ *exprs* : $\pi \Longrightarrow$
$\Gamma \vdash_L$ *expr* # *exprs* : τ # π

6.3 Elimination Rules

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

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

inductive-cases *MetaOperationCallTE* [*elim*]: $\Gamma \vdash_E$ *MetaOperationCall* τ *op* : σ

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

inductive-cases *TypeOperationCallTE* [*elim*]: $\Gamma \vdash_E$ *TypeOperationCall a k op* σ : τ
inductive-cases *AttributeCallTE* [*elim*]: $\Gamma \vdash_E$ *AttributeCall src k prop* : τ
inductive-cases *AssociationEndCallTE* [*elim*]: $\Gamma \vdash_E$ *AssociationEndCall src k role from* : τ
inductive-cases *AssociationClassCallTE* [*elim*]: $\Gamma \vdash_E$ *AssociationClassCall src k*

a from : τ
inductive-cases *AssociationClassEndCallTE* [*elim*]: $\Gamma \vdash_E$ *AssociationClassEnd-Call src k role* : τ
inductive-cases *OperationCallTE* [*elim*]: $\Gamma \vdash_E$ *OperationCall src k op params* : τ
inductive-cases *TupleElementCallTE* [*elim*]: $\Gamma \vdash_E$ *TupleElementCall src k elem* : τ

inductive-cases *IteratorTE* [*elim*]: $\Gamma \vdash_I$ (*src*, *its*, *body*) : *ys*
inductive-cases *IterateTE* [*elim*]: $\Gamma \vdash_E$ *IterateCall src k its its-ty res res-t res-init body* : τ
inductive-cases *AnyIteratorTE* [*elim*]: $\Gamma \vdash_E$ *AnyIteratorCall src k its its-ty body* : τ
inductive-cases *ClosureIteratorTE* [*elim*]: $\Gamma \vdash_E$ *ClosureIteratorCall src k its its-ty body* : τ
inductive-cases *CollectIteratorTE* [*elim*]: $\Gamma \vdash_E$ *CollectIteratorCall src k its its-ty body* : τ
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
$\Gamma \vdash_E$ *AssociationClassEndCall src k role* : τ
$\Gamma \vdash_E$ *OperationCall src k op params* : τ
$\Gamma \vdash_E$ *TupleElementCall src k elem* : τ

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

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

$\Gamma \vdash_P$ *CollectionItem a* : τ
$\Gamma \vdash_P$ *CollectionRange a b* : τ

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

6.5 Determinism

lemma
typing-det:

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

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

```
    using TupleElementCallT.hyps by fastforce
next
  case (IteratorT Γ src τ σ its-ty its body ϱ) show ?case
    apply (insert IteratorT.prems)
    apply (erule IteratorTE)
    using IteratorT.hyps element-type-det by blast
next
  case (IterateT Γ src its its-ty res res-t res-init body τ σ ϱ) show ?case
    apply (insert IterateT.prems)
    using IterateT.hyps by blast
next
  case (AnyIteratorT Γ src its its-ty body τ σ ϱ) thus ?case
    by (meson AnyIteratorTE Pair-inject)
next
  case (ClosureIteratorT Γ src its its-ty body τ σ ϱ υ) show ?case
    apply (insert ClosureIteratorT.prems)
    apply (erule ClosureIteratorTE)
    using ClosureIteratorT.hyps to-unique-collection-det by blast
next
  case (CollectIteratorT Γ src its its-ty body τ σ ϱ υ) show ?case
    apply (insert CollectIteratorT.prems)
    apply (erule CollectIteratorTE)
    using CollectIteratorT.hyps to-nonunique-collection-det
      update-element-type-det Pair-inject by metis
next
  case (CollectNestedIteratorT Γ src its its-ty body τ σ ϱ υ) show ?case
    apply (insert CollectNestedIteratorT.prems)
    apply (erule CollectNestedIteratorTE)
    using CollectNestedIteratorT.hyps to-nonunique-collection-det
      update-element-type-det Pair-inject by metis
next
  case (ExistsIteratorT Γ src its its-ty body τ σ ϱ) show ?case
    apply (insert ExistsIteratorT.prems)
    apply (erule ExistsIteratorTE)
    using ExistsIteratorT.hyps Pair-inject by metis
next
  case (ForAllIteratorT Γ 𝓜 src its its-ty body τ σ ϱ) show ?case
    apply (insert ForAllIteratorT.prems)
    apply (erule ForAllIteratorTE)
    using ForAllIteratorT.hyps Pair-inject by metis
next
  case (OneIteratorT Γ 𝓜 src its its-ty body τ σ ϱ) show ?case
    apply (insert OneIteratorT.prems)
    apply (erule OneIteratorTE)
    by simp
next
  case (IsUniqueIteratorT Γ 𝓜 src its its-ty body τ σ ϱ) show ?case
    apply (insert IsUniqueIteratorT.prems)
    apply (erule IsUniqueIteratorTE)
```

```
    by simp
next
  case (SelectIteratorT Γ ℳ src its its-ty body τ σ ϱ) show ?case
    apply (insert SelectIteratorT.prems)
    apply (erule SelectIteratorTE)
    using SelectIteratorT.hyps by blast
next
  case (RejectIteratorT Γ ℳ src its its-ty body τ σ ϱ) show ?case
    apply (insert RejectIteratorT.prems)
    apply (erule RejectIteratorTE)
    using RejectIteratorT.hyps by blast
next
  case (SortedByIteratorT Γ ℳ src its its-ty body τ σ ϱ υ) show ?case
    apply (insert SortedByIteratorT.prems)
    apply (erule SortedByIteratorTE)
    using SortedByIteratorT.hyps to-ordered-collection-det by blast
next
  case (ExprListNilT Γ) thus ?case
    using expr-list-typing.cases by auto
next
  case (ExprListConsT Γ expr τ exprs π) show ?case
    apply (insert ExprListConsT.prems)
    apply (erule ExprListTE)
    by (simp-all add: ExprListConsT.hyps)
qed
```

6.6 Code Setup

```
code-pred op-type .

code-pred (modes:
    i ⇒ i ⇒ i ⇒ bool,
    i ⇒ i ⇒ o ⇒ bool) iterator-typing .

end
```

Chapter 7

Normalization

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

7.1 Normalization Rules

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

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

The following variables are used in the table:

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

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

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

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

Table 7.1: Expression Normalization Rules

Orig. expr.	Normalized expression
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 (‹- $\vdash_C$ - ⇛/ -› [*51*,*51*,*51*] *50*) **and**
normalize-expr-list (‹- $\vdash_L$ - ⇛/ -› [*51*,*51*,*51*] *50*)
where
LiteralN:
Γ ⊢ *Literal a* ⇛ *Literal a*
|*ExplicitlyTypedLetN*:
$\Gamma \vdash init_1 \Rrightarrow init_2 \Longrightarrow$
$\Gamma(v \mapsto_f \tau) \vdash body_1 \Rrightarrow body_2 \Longrightarrow$
$\Gamma \vdash Let\ v\ (Some\ \tau)\ init_1\ body_1 \Rrightarrow Let\ v\ (Some\ \tau)\ init_2\ body_2$
|*ImplicitlyTypedLetN*:
$\Gamma \vdash init_1 \Rrightarrow init_2 \Longrightarrow$
$\Gamma \vdash_E init_2 : \tau \Longrightarrow$
$\Gamma(v \mapsto_f \tau) \vdash body_1 \Rrightarrow body_2 \Longrightarrow$

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

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

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

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

$\quad$ *element-type* $\tau\ \sigma \Longrightarrow$
$\quad$ *OclVoid*[?] $\leq \sigma \Longrightarrow$
$\quad$ $src_3 = \mathit{TypeOperationCall}\ src_2\ \mathit{ArrowCall}\ \mathit{SelectByKindOp}$
$\qquad\qquad (\textit{to-required-type}\ \sigma) \Longrightarrow$
$\quad$ $\Gamma \vdash_E src_3 : \varrho \Longrightarrow$
$\quad$ $(\Gamma,\ \varrho) \vdash_C call_1 \Rrightarrow call_2 \Longrightarrow$
$\quad$ $\Gamma \vdash \mathit{Call}\ src_1\ \mathit{SafeArrowCall}\ call_1 \Rrightarrow \mathit{Call}\ src_3\ \mathit{ArrowCall}\ call_2$
| *CollectionDotCallN*:
$\quad$ $\Gamma \vdash src_1 \Rrightarrow src_2 \Longrightarrow$
$\quad$ $\Gamma \vdash_E src_2 : \tau \Longrightarrow$
$\quad$ *element-type* $\tau\ \sigma \Longrightarrow$
$\quad$ $(\Gamma,\ \sigma) \vdash_C call_1 \Rrightarrow call_2 \Longrightarrow$
$\quad$ $it = \textit{new-vname}\ \Gamma \Longrightarrow$
$\quad$ $\Gamma \vdash \mathit{Call}\ src_1\ \mathit{DotCall}\ call_1 \Rrightarrow$
$\quad$ $\mathit{CollectIteratorCall}\ src_2\ \mathit{ArrowCall}\ [it]\ (\mathit{Some}\ \sigma)\ (\mathit{Call}\ (\mathit{Var}\ it)\ \mathit{DotCall}\ call_2)$
| *CollectionSafeDotCallN*:
$\quad$ $\Gamma \vdash src_1 \Rrightarrow src_2 \Longrightarrow$
$\quad$ $\Gamma \vdash_E src_2 : \tau \Longrightarrow$
$\quad$ *element-type* $\tau\ \sigma \Longrightarrow$
$\quad$ *OclVoid*[?] $\leq \sigma \Longrightarrow$
$\quad$ $\varrho = \textit{to-required-type}\ \sigma \Longrightarrow$
$\quad$ $src_3 = \mathit{TypeOperationCall}\ src_2\ \mathit{ArrowCall}\ \mathit{SelectByKindOp}\ \varrho \Longrightarrow$
$\quad$ $(\Gamma,\ \varrho) \vdash_C call_1 \Rrightarrow call_2 \Longrightarrow$
$\quad$ $it = \textit{new-vname}\ \Gamma \Longrightarrow$
$\quad$ $\Gamma \vdash \mathit{Call}\ src_1\ \mathit{SafeDotCall}\ call_1 \Rrightarrow$
$\quad$ $\mathit{CollectIteratorCall}\ src_3\ \mathit{ArrowCall}\ [it]\ (\mathit{Some}\ \varrho)\ (\mathit{Call}\ (\mathit{Var}\ it)\ \mathit{DotCall}\ call_2)$

| *TypeOperationN*:
$\quad$ $(\Gamma,\ \tau) \vdash_C \mathit{TypeOperation}\ op\ ty \Rrightarrow \mathit{TypeOperation}\ op\ ty$
| *AttributeN*:
$\quad$ $(\Gamma,\ \tau) \vdash_C \mathit{Attribute}\ attr \Rrightarrow \mathit{Attribute}\ attr$
| *AssociationEndN*:
$\quad$ $(\Gamma,\ \tau) \vdash_C \mathit{AssociationEnd}\ role\ from \Rrightarrow \mathit{AssociationEnd}\ role\ from$
| *AssociationClassN*:
$\quad$ $(\Gamma,\ \tau) \vdash_C \mathit{AssociationClass}\ \mathcal{A}\ from \Rrightarrow \mathit{AssociationClass}\ \mathcal{A}\ from$
| *AssociationClassEndN*:
$\quad$ $(\Gamma,\ \tau) \vdash_C \mathit{AssociationClassEnd}\ role \Rrightarrow \mathit{AssociationClassEnd}\ role$
| *OperationN*:
$\quad$ $\Gamma \vdash_L params_1 \Rrightarrow params_2 \Longrightarrow$
$\quad$ $(\Gamma,\ \tau) \vdash_C \mathit{Operation}\ op\ params_1 \Rrightarrow \mathit{Operation}\ op\ params_2$
| *TupleElementN*:
$\quad$ $(\Gamma,\ \tau) \vdash_C \mathit{TupleElement}\ elem \Rrightarrow \mathit{TupleElement}\ elem$

| *ExplicitlyTypedIterateN*:
$\quad$ $\Gamma \vdash \textit{res-init}_1 \Rrightarrow \textit{res-init}_2 \Longrightarrow$
$\quad$ $\Gamma \mathbin{++_f} \textit{fmap-of-list}\ (\mathit{map}\ (\lambda it.\ (it,\ \sigma))\ its) \vdash$
$\qquad$ $\mathit{Let}\ res\ \textit{res-t}_1\ \textit{res-init}_1\ body_1 \Rrightarrow \mathit{Let}\ res\ \textit{res-t}_2\ \textit{res-init}_2\ body_2 \Longrightarrow$
$\quad$ $(\Gamma,\ \tau) \vdash_C \mathit{Iterate}\ its\ (\mathit{Some}\ \sigma)\ res\ \textit{res-t}_1\ \textit{res-init}_1\ body_1 \Rrightarrow$
$\qquad$ $\mathit{Iterate}\ its\ (\mathit{Some}\ \sigma)\ res\ \textit{res-t}_2\ \textit{res-init}_2\ body_2$

$|ImplicitlyTypedIterateN$:
$\quad element\text{-}type\ \tau\ \sigma \Longrightarrow$
$\quad \Gamma \vdash res\text{-}init_1 \Rrightarrow res\text{-}init_2 \Longrightarrow$
$\quad \Gamma ++_f fmap\text{-}of\text{-}list\ (map\ (\lambda it.\ (it, \sigma))\ its) \vdash$
$\quad\quad Let\ res\ res\text{-}t_1\ res\text{-}init_1\ body_1 \Rrightarrow Let\ res\ res\text{-}t_2\ res\text{-}init_2\ body_2 \Longrightarrow$
$\quad (\Gamma, \tau) \vdash_C Iterate\ its\ None\ res\ res\text{-}t_1\ res\text{-}init_1\ body_1 \Rrightarrow$
$\quad\quad Iterate\ its\ (Some\ \sigma)\ res\ res\text{-}t_2\ res\text{-}init_2\ body_2$

$|ExplicitlyTypedIteratorN$:
$\quad \Gamma ++_f fmap\text{-}of\text{-}list\ (map\ (\lambda it.\ (it, \sigma))\ its) \vdash body_1 \Rrightarrow body_2 \Longrightarrow$
$\quad (\Gamma, \tau) \vdash_C Iterator\ iter\ its\ (Some\ \sigma)\ body_1 \Rrightarrow Iterator\ iter\ its\ (Some\ \sigma)\ body_2$
$|ImplicitlyTypedIteratorN$:
$\quad element\text{-}type\ \tau\ \sigma \Longrightarrow$
$\quad \Gamma ++_f fmap\text{-}of\text{-}list\ (map\ (\lambda it.\ (it, \sigma))\ its) \vdash body_1 \Rrightarrow body_2 \Longrightarrow$
$\quad (\Gamma, \tau) \vdash_C Iterator\ iter\ its\ None\ body_1 \Rrightarrow Iterator\ iter\ its\ (Some\ \sigma)\ body_2$

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

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*]: $\Gamma \vdash MetaOperationCall\ \tau\ op \Rrightarrow b$

inductive-cases *StaticOperationCallNE* [*elim*]: $\Gamma \vdash StaticOperationCall\ \tau\ op\ as \Rrightarrow b$
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*]: $(\Gamma, \tau) \vdash_C call \Rrightarrow b$
inductive-cases *OperationCallNE* [*elim*]: $(\Gamma, \tau) \vdash_C Operation\ op\ as \Rrightarrow call$
inductive-cases *IterateCallNE* [*elim*]: $(\Gamma, \tau) \vdash_C Iterate\ its\ its\text{-}ty\ res\ res\text{-}t\ res\text{-}init\ body \Rrightarrow call$
inductive-cases *IteratorCallNE* [*elim*]: $(\Gamma, \tau) \vdash_C Iterator\ iter\ its\ its\text{-}ty\ body \Rrightarrow call$

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

7.3 Simplification Rules

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

$\Gamma \vdash$ *MetaOperationCall* τ *op* $\Rrightarrow$ *b*
$\Gamma \vdash$ *StaticOperationCall* τ *op as* $\Rrightarrow$ *b*
$\Gamma \vdash$ *Call src DotCall call* $\Rrightarrow$ *b*
$\Gamma \vdash$ *Call src SafeDotCall call* $\Rrightarrow$ *b*
$\Gamma \vdash$ *Call src ArrowCall call* $\Rrightarrow$ *b*
$\Gamma \vdash$ *Call src SafeArrowCall call* $\Rrightarrow$ *b*

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

$\Gamma \vdash_L$ [] $\Rrightarrow$ *ys*
$\Gamma \vdash_L$ *x* # *xs* $\Rrightarrow$ *ys*

7.4 Determinism

lemma *any-has-not-element-type*:
  element-type τ σ $\Longrightarrow$ $\tau \leq$ *OclAny*[?] $\vee$ $\tau \leq$ *Tuple fmempty* $\Longrightarrow$ *False*
 by (*erule element-type.cases*; *auto*)

lemma *any-has-not-element-type'*:
  element-type τ σ $\Longrightarrow$ *OclVoid*[?] $\leq \tau$ $\Longrightarrow$ *False*
 by (*erule element-type.cases*; *auto*)

lemma
 normalize-det:
  $\Gamma \vdash$ *expr* $\Rrightarrow$ $expr_1$ $\Longrightarrow$
  $\Gamma \vdash$ *expr* $\Rrightarrow$ $expr_2$ $\Longrightarrow$ $expr_1 = expr_2$ **and**
 normalize-call-det:
  Γ-τ $\vdash_C$ *call* $\Rrightarrow$ $call_1$ $\Longrightarrow$
  Γ-τ $\vdash_C$ *call* $\Rrightarrow$ $call_2$ $\Longrightarrow$ $call_1 = call_2$ **and**
 normalize-expr-list-det:
  $\Gamma \vdash_L$ *xs* $\Rrightarrow$ *ys* $\Longrightarrow$
  $\Gamma \vdash_L$ *xs* $\Rrightarrow$ *zs* $\Longrightarrow$ *ys* = *zs*
 for Γ :: ('*a* :: *ocl-object-model*) *type env*
 and Γ-τ :: ('*a* :: *ocl-object-model*) *type env* $\times$ '*a type*
proof (*induct arbitrary*: $expr_2$ **and** $call_2$ **and** *zs*
  rule: *normalize-normalize-call-normalize-expr-list.inducts*)
 case (*LiteralN* Γ *a*) **thus** *?case* **by** *auto*
next

case (*ExplicitlyTypedLetN* Γ $init_1$ $init_2$ *v* τ $body_1$ $body_2$) **thus** *?case*
by *blast*
next
case (*ImplicitlyTypedLetN* Γ $init_1$ $init_2$ τ *v* $body_1$ $body_2$) **thus** *?case*
by (*metis* (*mono-tags*, *lifting*) *LetNE option.distinct*(*1*) *typing-det*)
next
case (*VarN* Γ *v*) **thus** *?case* **by** *auto*
next
case (*IfN* Γ a_1 a_2 b_1 b_2 c_1 c_2) **thus** *?case*
apply (*insert IfN.prems*)
apply (*erule IfNE*)
by (*simp add*: *IfN.hyps*)
next
case (*MetaOperationCallN* Γ τ *op*) **thus** *?case* **by** *auto*
next
case (*StaticOperationCallN* Γ $params_1$ $params_2$ τ *op*) **thus** *?case* **by** *blast*
next
case (*OclAnyDotCallN* Γ src_1 src_2 τ $call_1$ $call_2$) **show** *?case*
apply (*insert OclAnyDotCallN.prems*)
apply (*erule DotCallNE*)
using *OclAnyDotCallN.hyps typing-det* **apply** *metis*
using *OclAnyDotCallN.hyps any-has-not-element-type typing-det* **by** *metis*
next
case (*OclAnySafeDotCallN* Γ src_1 src_2 τ $call_1$ $call_2$) **show** *?case*
apply (*insert OclAnySafeDotCallN.prems*)
apply (*erule SafeDotCallNE*)
using *OclAnySafeDotCallN.hyps typing-det comp-apply*
apply (*metis* (*no-types*, *lifting*) *list.simps*(*8*) *list.simps*(*9*))
using *OclAnySafeDotCallN.hyps typing-det any-has-not-element-type′*
by *metis*
next
case (*OclAnyArrowCallN* Γ src_1 src_2 τ src_3 σ $call_1$ $call_2$) **show** *?case*
apply (*insert OclAnyArrowCallN.prems*)
apply (*erule ArrowCallNE*)
using *OclAnyArrowCallN.hyps typing-det comp-apply* **apply** *metis*
using *OclAnyArrowCallN.hyps typing-det any-has-not-element-type*
by *metis*
next
case (*CollectionArrowCallN* Γ src_1 src_2 τ *uu* $call_1$ $call_2$) **show** *?case*
apply (*insert CollectionArrowCallN.prems*)
apply (*erule ArrowCallNE*)
using *CollectionArrowCallN.hyps typing-det any-has-not-element-type*
apply *metis*
using *CollectionArrowCallN.hyps typing-det* **by** *metis*
next
case (*CollectionSafeArrowCallN* Γ src_1 src_2 τ σ src_3 ϱ $call_1$ $call_2$) **show** *?case*
apply (*insert CollectionSafeArrowCallN.prems*)
apply (*erule SafeArrowCallNE*)
using *CollectionSafeArrowCallN.hyps typing-det element-type-det* **by** *metis*

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

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

7.5 Normalized Expressions Typing

Here is the final typing rules.

```
inductive nf-typing (‹(1-/ ⊢/ (- :/ -))› [51,51,51] 50) where
  Γ ⊢ expr ⇛ exprN ⟹
  Γ ⊢E exprN : τ ⟹
  Γ ⊢ expr : τ

lemma nf-typing-det:
  Γ ⊢ expr : τ ⟹
  Γ ⊢ expr : σ ⟹ τ = σ
  by (metis nf-typing.cases normalize-det typing-det)
```

7.6 Code Setup

```
code-pred normalize .

code-pred nf-typing .

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**
$Object \sqcup \text{-} = Object$
| $Person \sqcup c = (if\ c = Person \vee c = Employee \vee c = Customer$
$then\ Person\ else\ Object)$
| $Employee \sqcup c = (if\ c = Employee\ then\ Employee\ else$
$if\ c = Person \vee c = Customer\ then\ Person\ else\ Object)$
| $Customer \sqcup c = (if\ c = Customer\ then\ Customer\ else$
$if\ c = Person \vee c = Employee\ then\ Person\ else\ Object)$
| $Project \sqcup c = (if\ c = Project\ then\ Project\ else\ Object)$
| $Task \sqcup c = (if\ c = Task\ then\ Task\ else\ Object)$
| $Sprint \sqcup c = (if\ c = Sprint\ then\ Sprint\ else\ Object)$

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

```
  c < d ⟷ c ≤ d ∧ ¬ d ≤ c
  for c d :: classes1
  unfolding less-classes1-def less-eq-classes1-def
  using subclass1.simps by auto

lemma order-refl-classes1:
  c ≤ c
  for c :: classes1
  unfolding less-eq-classes1-def by simp

lemma order-trans-classes1:
  c ≤ d ⟹ d ≤ e ⟹ c ≤ e
  for c d e :: classes1
  unfolding less-eq-classes1-def
  using subclass1.simps by auto

lemma antisym-classes1:
  c ≤ d ⟹ d ≤ c ⟹ c = d
  for c d :: classes1
  unfolding less-eq-classes1-def
  using subclass1.simps by auto

lemma sup-ge1-classes1:
  c ≤ c ⊔ d
  for c d :: classes1
  by (induct c; auto simp add: less-eq-classes1-def less-classes1-def subclass1.simps)

lemma sup-ge2-classes1:
  d ≤ c ⊔ d
  for c d :: classes1
  by (induct c; auto simp add: less-eq-classes1-def less-classes1-def subclass1.simps)

lemma sup-least-classes1:
  c ≤ e ⟹ d ≤ e ⟹ c ⊔ d ≤ e
  for c d e :: classes1
  by (induct c; induct d;
      auto simp add: less-eq-classes1-def less-classes1-def subclass1.simps)

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

end
```

code-pred *subclass1* **.**

fun *subclass1-fun* **where**
subclass1-fun Object $\mathcal{C}$ = *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*]:
(<) = *subclass1-fun*
proof (*intro ext iffI*)
fix $\mathcal{C}$ $\mathcal{D}$ :: *classes1*
show $\mathcal{C} < \mathcal{D} \Longrightarrow$ *subclass1-fun* $\mathcal{C}$ $\mathcal{D}$
unfolding *less-classes1-def*
apply (*erule subclass1.cases*, *auto*)
using *subclass1-fun.elims*(*3*) **by** *blast*
show *subclass1-fun* $\mathcal{C}$ $\mathcal{D} \Longrightarrow \mathcal{C} < \mathcal{D}$
by (*erule subclass1-fun.elims*, *auto simp add*: *less-classes1-def subclass1.intros*)
qed

lemma *less-eq-classes1-code* [*code*]:
(≤) = (λ*x y*. *subclass1-fun x y* ∨ *x* = *y*)
unfolding *dual-order.order-iff-strict less-classes1-code*
by *auto*

8.2 Object Model

abbreviation Γ_0 ≡ *fmempty* :: *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*'' $\mapsto_f$ [
STR ''*projects*'' $\mapsto_f$ (*Project*, *0*::*nat*, ∞::*enat*, *False*, *True*),
STR ''*manager*'' $\mapsto_f$ (*Employee*, *1*, *1*, *False*, *False*)],
STR ''*ProjectMember*'' $\mapsto_f$ [
STR ''*member-of*'' $\mapsto_f$ (*Project*, *0*, ∞, *False*, *False*),
STR ''*members*'' $\mapsto_f$ (*Employee*, *1*, *20*, *True*, *True*)],
STR ''*ManagerEmployee*'' $\mapsto_f$ [
STR ''*line-manager*'' $\mapsto_f$ (*Employee*, *0*, *1*, *False*, *False*),
STR ''*project-manager*'' $\mapsto_f$ (*Employee*, *0*, ∞, *False*, *False*),
STR ''*employees*'' $\mapsto_f$ (*Employee*, *3*, *7*, *False*, *False*)],
STR ''*ProjectCustomer*'' $\mapsto_f$ [
STR ''*projects*'' $\mapsto_f$ (*Project*, *0*, ∞, *False*, *True*),
STR ''*customer*'' $\mapsto_f$ (*Customer*, *1*, *1*, *False*, *False*)],
STR ''*ProjectTask*'' $\mapsto_f$ [
STR ''*project*'' $\mapsto_f$ (*Project*, *1*, *1*, *False*, *False*),
STR ''*tasks*'' $\mapsto_f$ (*Task*, *0*, ∞, *True*, *True*)],
STR ''*SprintTaskAssignee*'' $\mapsto_f$ [
STR ''*sprint*'' $\mapsto_f$ (*Sprint*, *0*, *10*, *False*, *True*),
STR ''*tasks*'' $\mapsto_f$ (*Task*, *0*, *5*, *False*, *True*),
STR ''*assignee*'' $\mapsto_f$ (*Employee*, *0*, *1*, *False*, *False*)]]

definition *associations-classes1* ≡ *assocs*

definition *association-classes-classes1* ≡ *fmempty* :: *classes1* $\rightharpoonup_f$ *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*
unfolding *assoc-end-min-def assoc-end-max-def*
using *zero-enat-def one-enat-def numeral-eq-enat* **by** *auto*

lemma *association-ends-unique*:
assumes *association-ends' classes assocs* $\mathcal{C}$ *from role* end_1
and *association-ends' classes assocs* $\mathcal{C}$ *from role* end_2
shows $end_1 = end_2$
proof −
have ¬ *association-ends-not-unique' classes assocs* **by** *eval*
with *assms* **show** *?thesis*
using *association-ends-not-unique'.simps* **by** *blast*
qed

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

end

8.3 Simplification Rules

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

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

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

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

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

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

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

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

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

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

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

lemma *attribute-Employee-name* [*simp*]:
attribute Employee STR ''name'' $\mathcal{D}$ τ =
($\mathcal{D}$ = *Employee* $\wedge$ τ = *String*[*1*])
proof −
have *attribute Employee STR ''name'' Employee String*[*1*]
by *eval*
thus *?thesis*
using *attribute-det* **by** *blast*
qed

lemma *association-end-Project-members* [*simp*]:
association-end Project None STR ''members'' $\mathcal{D}$ τ =
($\mathcal{D}$ = *Project* ∧ τ = (*Employee*, *1*, *20*, *True*, *True*))
proof −
have *association-end Project None STR ''members''*
Project (*Employee*, *1*, *20*, *True*, *True*)
by *eval*
thus *?thesis*
using *association-end-det* **by** *blast*
qed

lemma *association-end-Employee-projects-simp* [*simp*]:
association-end Employee None STR ''projects'' $\mathcal{D}$ τ =
($\mathcal{D}$ = *Employee* ∧ τ = (*Project*, *0*, ∞, *False*, *True*))
proof −
have *association-end Employee None STR ''projects''*
Employee (*Project*, *0*, ∞, *False*, *True*)
by *eval*
thus *?thesis*
using *association-end-det* **by** *blast*
qed

lemma *static-operation-Project-allProjects* [*simp*]:
static-operation ⟨*Project*⟩$_{\mathcal{T}}$[*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 −
have *static-operation* ⟨*Project*⟩$_{\mathcal{T}}$[*1*] *STR ''allProjects''* []
(*STR ''allProjects''*, ⟨*Project*⟩$_{\mathcal{T}}$[*1*], [], *Set* ⟨*Project*⟩$_{\mathcal{T}}$[*1*], *True*,
Some (*MetaOperationCall* ⟨*Project*⟩$_{\mathcal{T}}$[*1*] *AllInstancesOp*))
by *eval*
thus *?thesis*
using *static-operation-det* **by** *blast*
qed

8.4 Basic Types

8.4.1 Positive Cases

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

8.4.2 Negative Cases

lemma ¬ *String* ≤ (*Boolean* :: *classes1 basic-type*) **by** *simp*

8.5 Types

8.5.1 Positive Cases

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

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

by *simp*
lemma *Set Integer[1]* ⊔ *(Real[1] :: classes1 type) = OclSuper* **by** *simp*

8.5.2 Negative Cases

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

8.6 Typing

8.6.1 Positive Cases

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

lemma
Γ_0 ⊢ *EnumLiteral STR ''E1'' STR ''A'' : (Enum STR ''E1'')[1]*
by *simp*

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

lemma
Γ_0 ⊢ *OperationCall (BooleanLiteral True) DotCall OrOp*
[BooleanLiteral False] : Boolean[1]
by *simp*

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

lemma
Γ_0 ⊢ *OperationCall (NullLiteral) DotCall AndOp*
[BooleanLiteral True] : Boolean[?]
by *simp*

```
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]
by *simp*

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

lemma
$\Gamma_0 \vdash$ *OperationCall* (*NullLiteral*) *DotCall OclIsUndefinedOp* [] : *Boolean*[*1*]
by *simp*

```
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*]
by *simp*

```
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*])]))
by *simp*

```
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*]
by *simp*

```
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*]
by *simp*

```
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*[*?*]
by *simp*

```
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*]
by *simp*

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

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

```
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*]
by *simp*

```
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*]
by *simp*

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

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

by *simp*

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

lemma
$\Gamma_0 \vdash$ *StaticOperationCall Project*[*1*] *STR ''allProjects''* [] : *Set Project*[*1*]
by *simp*

8.6.2 Negative Cases

```
true = null
```

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

```
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
by *simp*

```
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
by *simp*

```
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 −
have ¬ *operation-defined* (*Integer*[*?*] :: *classes1 type*) *STR ''max''* [*Integer*[*?*]]
by *eval*
thus *?thesis* **by** *simp*
qed

8.7 Code

8.7.1 Positive Cases

values $\{(\mathcal{D}, \tau).$ *attribute Employee STR ''name''* $\mathcal{D}\ \tau\}$

values $\{(\mathcal{D}, end).\ association\text{-}end\ Employee\ None\ STR\ ''employees''\ \mathcal{D}\ end\}$
values $\{(\mathcal{D}, end).\ association\text{-}end\ Employee\ (Some\ STR\ ''project\text{-}manager'')\ STR\ ''employees''\ \mathcal{D}\ end\}$
values $\{op.\ operation\ Project[1]\ STR\ ''membersCount''\ []\ op\}$
values $\{op.\ operation\ Project[1]\ STR\ ''membersByName''\ [String[1]]\ op\}$
value $has\text{-}literal\ STR\ ''E1''\ STR\ ''A''$

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

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

8.7.2 Negative Cases

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

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

values
$\{\tau.\ \Gamma_0 \vdash OperationCall\ (CollectionLiteral\ SequenceKind$
$[CollectionRange\ (IntegerLiteral\ 1)\ (IntegerLiteral\ 5),$
$CollectionItem\ NullLiteral])$
$ArrowCall\ CollectionMaxOp\ [] : \tau\}$

end

Bibliography

[1] Object Management Group, "Object Constraint Language (OCL). Version 2.4," Feb. 2014. http://www.omg.org/spec/OCL/2.4/.

[2] A. D. Brucker, F. Tuong, and B. Wolff, "Featherweight OCL: A proposal for a machine-checked formal semantics for OCL 2.5," *Archive of Formal Proofs*, Jan. 2014. http://isa-afp.org/entries/Featherweight_OCL.html, 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.