Zermelo Fraenkel Set Theory in Higher-Order Logic

Lawrence C. Paulson
Computer Laboratory
University of Cambridge

December 14, 2021

Abstract

This entry is a new formalisation of ZFC set theory in Isabelle/HOL. It is logically equivalent to Obua's HOLZF [2]; the point is to have the closest possible integration with the rest of Isabelle/HOL, minimising the amount of new notations and exploiting type classes.

There is a type V of sets and a function *elts* :: $V \Rightarrow V$ *set* mapping a set to its elements. Classes simply have type V *set*, and the predicate *small* identifies those classes that correspond to actual sets. Type classes connected with orders and lattices are used to minimise the amount of new notation for concepts such as the subset relation, union and intersection. Basic concepts are formalised: Cartesian products, disjoint sums, natural numbers, functions, etc.

More advanced set-theoretic concepts, such as transfinite induction, ordinals, cardinals and the transitive closure of a set, are also provided. The definition of addition and multiplication for general sets (not just ordinals) follows Kirby [1]. The development includes essential results about cardinal arithmetic. It also develops ordinal exponentiation, Cantor normal form and the concept of indecomposable ordinals. There are numerous results about order types.

The theory provides two type classes with the aim of facilitating developments that combine V with other Isabelle/HOL types: *embeddable*, the class of types that can be injected into V (including V itself as well as $V*V$, V *list*, etc.), and *small*, the class of types that correspond to some ZF set.

Contents

theory *ZFC-Library*
imports *HOL−Library.Countable-Set HOL−Library.Equipollence HOL−Cardinals.Cardinals*

begin

Equipollence and Lists.

lemma *countable-iff-lepoll*: *countable* $A \longleftrightarrow A \lesssim (UNIV :: nat\ set)$
by (*auto simp*: *countable-def lepoll-def*)

lemma *infinite-times-eqpoll-self*:
assumes *infinite* A **shows** $A \times A \approx A$
by (*simp add*: *Times-same-infinite-bij-betw assms eqpoll-def*)

lemma *infinite-finite-times-lepoll-self*:
assumes *infinite* A *finite* B **shows** $A \times B \lesssim A$
proof −
have $B \lesssim A$
by (*simp add*: *assms finite-lepoll-infinite*)
then have $A \times B \lesssim A \times A$
by (*simp add*: *subset-imp-lepoll times-lepoll-mono*)
also have $\ldots \approx A$
by (*simp add*: ‹*infinite A*› *infinite-times-eqpoll-self*)
finally show *?thesis* **.**
qed

lemma *lists-n-lepoll-self*:
assumes *infinite* A **shows** $\{l \in lists\ A.\ length\ l = n\} \lesssim A$
proof (*induction n*)
case *0*
have $\{l \in lists\ A.\ length\ l = 0\} = \{[]\}$
by *auto*
then show *?case*
by (*metis Set.set-insert assms ex-in-conv finite.emptyI singleton-lepoll*)
next
case (*Suc n*)
have $\{l \in lists\ A.\ length\ l = Suc\ n\} = (\bigcup x{\in}A.\ \bigcup l \in \{l \in lists\ A.\ length\ l = n\}.\ \{x\#l\})$
by (*auto simp*: *length-Suc-conv*)
also have $\ldots \lesssim A \times \{l \in lists\ A.\ length\ l = n\}$
unfolding *lepoll-iff*
by (*rule-tac* $x{=}\lambda(x,l).\ Cons\ x\ l$ **in** *exI*) *auto*
also have $\ldots \lesssim A$
proof (*cases finite* $\{l \in lists\ A.\ length\ l = n\}$)
case *True*
then show *?thesis*
using *assms infinite-finite-times-lepoll-self* **by** *blast*
next
case *False*
have $A \times \{l \in lists\ A.\ length\ l = n\} \lesssim A \times A$

by (*simp add*: *Suc.IH subset-imp-lepoll times-lepoll-mono*)
also have $\ldots \approx A$
by (*simp add*: *assms infinite-times-eqpoll-self*)
finally show *?thesis* **.**
qed
finally show *?case* **.**
qed

lemma *infinite-eqpoll-lists*:
assumes *infinite A* **shows** *lists A* $\approx A$
proof −
have *lists A* $\lesssim$ *Sigma UNIV* $(\lambda n.\ \{l \in lists\ A.\ length\ l = n\})$
unfolding *lepoll-iff*
by (*rule-tac x*=*snd* **in** *exI*) (*auto simp*: *in-listsI snd-image-Sigma*)
also have $\ldots \lesssim$ (*UNIV*::*nat set*) $\times A$
by (*rule Sigma-lepoll-mono*) (*auto simp*: *lists-n-lepoll-self assms*)
also have $\ldots \lesssim A \times A$
by (*metis assms infinite-le-lepoll order-refl subset-imp-lepoll times-lepoll-mono*)
also have $\ldots \approx A$
by (*simp add*: *assms infinite-times-eqpoll-self*)
finally show *?thesis*
by (*simp add*: *lepoll-antisym lepoll-lists*)
qed

end

1 The ZF Axioms, Ordinals and Transfinite Recursion

theory *ZFC-in-HOL*
imports *ZFC-Library*

begin

1.1 Syntax and axioms

hide-const (**open**) *list.set Sum subset*

notation
inf (**infixl** $\sqcap$ *70*) **and**
sup (**infixl** $\sqcup$ *65*) **and**
Inf ($\bigsqcap$) **and**
Sup ($\bigsqcup$)

typedecl *V*

Presentation refined by Dmitriy Traytel

axiomatization *elts* :: $V \Rightarrow V\ set$

where *ext* [*intro?*]: $elts\ x = elts\ y \Longrightarrow x{=}y$
and *down-raw*: $Y \subseteq elts\ x \Longrightarrow Y \in range\ elts$
and *Union-raw*: $X \in range\ elts \Longrightarrow Union\ (elts\ `\ X) \in range\ elts$
and *Pow-raw*: $X \in range\ elts \Longrightarrow inv\ elts\ `\ Pow\ X \in range\ elts$
and *replacement-raw*: $X \in range\ elts \Longrightarrow f\ `\ X \in range\ elts$
and *inf-raw*: $range\ (g :: nat \Rightarrow V) \in range\ elts$
and *foundation*: $wf\ \{(x,y).\ x \in elts\ y\}$

lemma *mem-not-refl* [*simp*]: $i \notin elts\ i$
using *wf-not-refl* [*OF foundation*] **by** *force*

lemma *mem-not-sym*: $\neg\ (x \in elts\ y \wedge y \in elts\ x)$
using *wf-not-sym* [*OF foundation*] **by** *force*

A set is small if it can be injected into the extension of a V-set.

definition *small* :: *'a set* ⇒ *bool*
where $small\ X \equiv \exists\ V\text{-}of :: 'a \Rightarrow V.\ inj\text{-}on\ V\text{-}of\ X \wedge V\text{-}of\ `\ X \in range\ elts$

lemma *small-empty* [*iff*]: *small* {}
by (*simp add*: *small-def down-raw*)

lemma *small-iff-range*: $small\ X \longleftrightarrow X \in range\ elts$
apply (*simp add*: *small-def*)
by (*metis inj-on-id2 replacement-raw the-inv-into-onto*)

Small classes can be mapped to sets.

definition $set\ X \equiv (if\ small\ X\ then\ inv\ elts\ X\ else\ inv\ elts\ \{\})$

lemma *set-of-elts* [*simp*]: $set\ (elts\ x) = x$
by (*force simp add*: *ext set-def f-inv-into-f small-def*)

lemma *elts-of-set* [*simp*]: $elts\ (set\ X) = (if\ small\ X\ then\ X\ else\ \{\})$
by (*simp add*: *ZFC-in-HOL.set-def down-raw f-inv-into-f small-iff-range*)

lemma *down*: $Y \subseteq elts\ x \Longrightarrow small\ Y$
by (*simp add*: *down-raw small-iff-range*)

lemma *Union* [*intro*]: $small\ X \Longrightarrow small\ (Union\ (elts\ `\ X))$
by (*simp add*: *Union-raw small-iff-range*)

lemma *Pow*: $small\ X \Longrightarrow small\ (set\ `\ Pow\ X)$
unfolding *small-iff-range* **using** *Pow-raw set-def down* **by** *force*

declare *replacement-raw* [*intro*,*simp*]

lemma *replacement* [*intro*,*simp*]:
assumes *small X*
shows *small* (*f* ` *X*)
proof −

```
  let ?A = inv-into X f ‘ (f ‘ X)
  have AX: ?A ⊆ X
    by (simp add: image-subsetI inv-into-into)
  have inj: inj-on f ?A
    by (simp add: f-inv-into-f inj-on-def)
  have injo: inj-on (inv-into X f) (f ‘ X)
    using inj-on-inv-into by blast
  have ∃ V-of. inj-on V-of (f ‘ X) ∧ V-of ‘ f ‘ X ∈ range elts
    if inj-on V-of X and V-of ‘ X = elts x
    for V-of :: 'a ⇒ V and x
  proof (intro exI conjI)
    show inj-on (V-of ∘ inv-into X f) (f ‘ X)
      by (meson ‹inv-into X f ‘ f ‘ X ⊆ X› comp-inj-on inj-on-subset injo that)
    have (λx. V-of (inv-into X f (f x))) ‘ X = elts (set (V-of ‘ ?A))
      by (metis AX down elts-of-set image-image image-mono that(2))
    then show (V-of ∘ inv-into X f) ‘ f ‘ X ∈ range elts
      by (metis image-comp image-image rangeI)
  qed
  then show ?thesis
    using assms by (auto simp: small-def)
qed

lemma small-image-iff [simp]: inj-on f A ⟹ small (f ‘ A) ⟷ small A
  by (metis replacement the-inv-into-onto)
```

A little bootstrapping is needed to characterise *small* for sets of arbitrary type.

```
lemma inf: small (range (g :: nat ⇒ V))
  by (simp add: inf-raw small-iff-range)

lemma small-image-nat-V [simp]: small (g ‘ N) for g :: nat ⇒ V
 by (metis (mono-tags, opaque-lifting) down elts-of-set image-iff inf rangeI subsetI)

lemma Finite-V:
  fixes X :: V set
  assumes finite X shows small X
 using ex-bij-betw-nat-finite [OF assms] unfolding bij-betw-def by (metis small-image-nat-V)

lemma small-insert-V:
  fixes X :: V set
  assumes small X
  shows small (insert a X)
proof (cases finite X)
  case True
  then show ?thesis
    by (simp add: Finite-V)
next
  case False
  show ?thesis
```

```
    using infinite-imp-bij-betw2 [OF False]
   by (metis replacement Un-insert-right assms bij-betw-imp-surj-on sup-bot.right-neutral)
qed

lemma small-UN-V [simp,intro]:
  fixes B :: 'a ⇒ V set
  assumes X: small X and B: ⋀x. x ∈ X ⟹ small (B x)
  shows small (⋃x∈X. B x)
proof −
  have (⋃ (elts ‘ (λx. ZFC-in-HOL.set (B x)) ‘ X)) = (⋃ (B ‘ X))
    using B by force
  then show ?thesis
    using Union [OF replacement [OF X, of λx. ZFC-in-HOL.set (B x)]] by simp
qed

definition vinsert where vinsert x y ≡ set (insert x (elts y))

lemma elts-vinsert [simp]: elts (vinsert x y) = insert x (elts y)
  using down small-insert-V vinsert-def by auto

definition succ where succ x ≡ vinsert x x

lemma elts-succ [simp]: elts (succ x) = insert x (elts x)
  by (simp add: succ-def)

lemma finite-imp-small:
  assumes finite X shows small X
  using assms
proof induction
  case empty
  then show ?case
    by simp
next
  case (insert a X)
  then obtain V-of u where u: inj-on V-of X V-of ‘ X = elts u
    by (meson small-def image-iff)
  show ?case
    unfolding small-def
  proof (intro exI conjI)
    show inj-on (V-of(a:=u)) (insert a X)
      using u
      apply (clarsimp simp add: inj-on-def)
      by (metis image-eqI mem-not-refl)
    have (V-of(a:=u)) ‘ insert a X = elts (vinsert u u)
      using insert.hyps(2) u(2) by auto
    then show (V-of(a:=u)) ‘ insert a X ∈ range elts
      by (blast intro:  elim: )
  qed
qed
```

lemma *small-insert*:
assumes *small X*
shows *small* (*insert a X*)
proof (*cases finite X*)
case *True*
then show *?thesis*
by (*simp add*: *finite-imp-small*)
next
case *False*
show *?thesis*
using *infinite-imp-bij-betw2* [*OF False*]
by (*metis replacement Un-insert-right assms bij-betw-imp-surj-on sup-bot.right-neutral*)
qed

lemma *smaller-than-small*:
assumes *small A B* ⊆ *A* **shows** *small B*
using *assms*
by (*metis down elts-of-set image-mono small-def small-iff-range subset-inj-on*)

lemma *small-insert-iff* [*iff*]: *small* (*insert a X*) ⟷ *small X*
by (*meson small-insert smaller-than-small subset-insertI*)

lemma *small-iff*: *small X* ⟷ (∃ *x*. *X* = *elts x*)
by (*metis down elts-of-set subset-refl*)

lemma *small-elts* [*iff*]: *small* (*elts x*)
by (*auto simp*: *small-iff*)

lemma *small-diff* [*iff*]: *small* (*elts a* − *X*)
by (*meson Diff-subset down*)

lemma *small-set* [*simp*]: *small* (*list.set xs*)
by (*simp add*: *ZFC-in-HOL.finite-imp-small*)

lemma *small-upair*: *small* {*x*,*y*}
by *simp*

lemma *small-Un-elts*: *small* (*elts x* ∪ *elts y*)
using *Union* [*OF small-upair*] **by** *auto*

lemma *small-eqcong*: ⟦*small X*; *X* ≈ *Y*⟧ ⟹ *small Y*
by (*metis bij-betw-imp-surj-on eqpoll-def replacement*)

lemma *big-UNIV* [*simp*]: ¬ *small* (*UNIV*::*V set*) (**is** ¬ *small ?U*)
proof
assume *small ?U*
then have *small A* **for** *A* :: *V set*
by (*metis* (*full-types*) *UNIV-I down small-iff subsetI*)

then have *range elts = UNIV*
by (*meson small-iff surj-def*)
then show *False*
by (*metis Cantors-paradox Pow-UNIV*)
qed

lemma *inj-on-set*: *inj-on set* (*Collect small*)
by (*metis elts-of-set inj-onI mem-Collect-eq*)

lemma *set-injective* [*simp*]: ⟦*small X*; *small Y*⟧ ⟹ *set X = set Y* ⟷ *X=Y*
by (*metis elts-of-set*)

1.2 Type classes and other basic setup

instantiation *V* :: *zero*
begin
definition *zero-V* **where** *0* ≡ *set* {}
instance ..
end

lemma *elts-0* [*simp*]: *elts 0 = {}*
by (*simp add*: *zero-V-def*)

lemma *set-empty* [*simp*]: *set {} = 0*
by (*simp add*: *zero-V-def*)

instantiation *V* :: *one*
begin
definition *one-V* **where** *1* ≡ *succ 0*
instance ..
end

lemma *elts-1* [*simp*]: *elts 1 = {0}*
by (*simp add*: *one-V-def*)

lemma *insert-neq-0* [*simp*]: *set* (*insert a X*) *= 0* ⟷ ¬ *small X*
unfolding *zero-V-def*
by (*metis elts-of-set empty-not-insert set-of-elts small-insert-iff*)

lemma *elts-eq-empty-iff* [*simp*]: *elts x = {}* ⟷ *x=0*
by (*auto simp*: *ZFC-in-HOL.ext*)

instantiation *V* :: *distrib-lattice*
begin

definition *inf-V* **where** *inf-V x y* ≡ *set* (*elts x* ∩ *elts y*)

definition *sup-V* **where** *sup-V x y* ≡ *set* (*elts x* ∪ *elts y*)

definition *less-eq-V* **where** *less-eq-V x y* ≡ *elts x* ⊆ *elts y*

definition *less-V* **where** *less-V x y* ≡ *less-eq x y* ∧ *x* ≠ (*y*::*V*)

instance
proof
  show (*x* < *y*) = (*x* ≤ *y* ∧ ¬ *y* ≤ *x*) **for** *x* :: *V* **and** *y* :: *V*
    using *ext less-V-def less-eq-V-def* **by** *auto*
  show *x* ≤ *x* **for** *x* :: *V*
    by (*simp add*: *less-eq-V-def*)
  show *x* ≤ *z* **if** *x* ≤ *y y* ≤ *z* **for** *x y z* :: *V*
    using *that* **by** (*auto simp*: *less-eq-V-def*)
  show *x* = *y* **if** *x* ≤ *y y* ≤ *x* **for** *x y* :: *V*
    using *that* **by** (*simp add*: *ext less-eq-V-def*)
  show *inf x y* ≤ *x* **for** *x y* :: *V*
    by (*metis down elts-of-set inf-V-def inf-sup-ord*(*1*) *less-eq-V-def*)
  show *inf x y* ≤ *y* **for** *x y* :: *V*
    by (*metis Int-lower2 down elts-of-set inf-V-def less-eq-V-def*)
  show *x* ≤ *inf y z* **if** *x* ≤ *y x* ≤ *z* **for** *x y z* :: *V*
  proof −
    have *small* (*elts y* ∩ *elts z*)
      by (*meson down inf.cobounded1*)
    then show *?thesis*
      using *elts-of-set inf-V-def less-eq-V-def that* **by** *auto*
  qed
  show *x* ≤ *x* ⊔ *y y* ≤ *x* ⊔ *y* **for** *x y* :: *V*
    by (*simp-all add*: *less-eq-V-def small-Un-elts sup-V-def*)
  show *sup y z* ≤ *x* **if** *y* ≤ *x z* ≤ *x* **for** *x y z* :: *V*
    using *less-eq-V-def sup-V-def that* **by** *auto*
  show *sup x* (*inf y z*) = *inf* (*x* ⊔ *y*) (*sup x z*) **for** *x y z* :: *V*
  proof −
    have *small* (*elts y* ∩ *elts z*)
      by (*meson down inf.cobounded2*)
    then show *?thesis*
      by (*simp add*: *Un-Int-distrib inf-V-def small-Un-elts sup-V-def*)
  qed
qed
end

lemma *V-equalityI* [*intro*]: (⋀*x*. *x* ∈ *elts a* ⟷ *x* ∈ *elts b*) ⟹ *a* = *b*
  by (*meson dual-order.antisym less-eq-V-def subsetI*)

lemma *vsubsetI* [*intro*!]: (⋀*x*. *x* ∈ *elts a* ⟹ *x* ∈ *elts b*) ⟹ *a* ≤ *b*
  by (*simp add*: *less-eq-V-def subsetI*)

lemma *vsubsetD* [*elim*, *intro*?]: *a* ≤ *b* ⟹ *c* ∈ *elts a* ⟹ *c* ∈ *elts b*
  using *less-eq-V-def* **by** *auto*

lemma *rev-vsubsetD*: *c* ∈ *elts a* ⟹ *a* ≤ *b* ⟹ *c* ∈ *elts b*

— The same, with reversed premises for use with *erule* – cf. $[\![?P;\ ?P \longrightarrow ?Q]\!] \Longrightarrow ?Q$.
by (*rule vsubsetD*)

lemma *vsubsetCE* [*elim*,*no-atp*]: $a \leq b \Longrightarrow (c \notin elts\ a \Longrightarrow P) \Longrightarrow (c \in elts\ b \Longrightarrow P) \Longrightarrow P$
— Classical elimination rule.
using *vsubsetD* **by** *blast*

lemma *set-image-le-iff*: $small\ A \Longrightarrow set\ (f\ `\ A) \leq B \longleftrightarrow (\forall x{\in}A.\ f\ x \in elts\ B)$
by *auto*

lemma *eq0-iff*: $x = 0 \longleftrightarrow (\forall y.\ y \notin elts\ x)$
by *auto*

lemma *less-eq-V-0-iff* [*simp*]: $x \leq 0 \longleftrightarrow x = 0$ **for** $x{::}V$
by *auto*

lemma *subset-iff-less-eq-V*:
assumes *small B* **shows** $A \subseteq B \longleftrightarrow set\ A \leq set\ B \wedge small\ A$
using *assms down small-iff* **by** *auto*

lemma *small-Collect* [*simp*]: $small\ A \Longrightarrow small\ \{x \in A.\ P\ x\}$
by (*simp add*: *smaller-than-small*)

lemma *small-Union-iff*: $small\ (\bigcup(elts\ `\ X)) \longleftrightarrow small\ X$
proof
show *small X*
if $small\ (\bigcup\ (elts\ `\ X))$
proof −
have $X \subseteq set\ `\ Pow\ (\bigcup\ (elts\ `\ X))$
by *fastforce*
then show *?thesis*
using *Pow subset-iff-less-eq-V that* **by** *auto*
qed
qed *auto*

lemma *not-less-0* [*iff*]:
fixes $x{::}V$ **shows** $\neg\ x < 0$
by (*simp add*: *less-eq-V-def less-le-not-le*)

lemma *le-0* [*iff*]:
fixes $x{::}V$ **shows** $0 \leq x$
by *auto*

lemma *min-0L* [*simp*]: $min\ 0\ n = 0$ **for** $n :: V$
by (*simp add*: *min-absorb1*)

lemma *min-0R* [*simp*]: $min\ n\ 0 = 0$ **for** $n :: V$

by (*simp add*: *min-absorb2*)

lemma *neq0-conv*: $\bigwedge n::V.\ n \neq 0 \longleftrightarrow 0 < n$
by (*simp add*: *less-V-def*)

definition *VPow* :: $V \Rightarrow V$
where $VPow\ x \equiv set\ (set\ `\ Pow\ (elts\ x))$

lemma *VPow-iff* [*iff*]: $y \in elts\ (VPow\ x) \longleftrightarrow y \leq x$
using *down Pow*
apply (*auto simp*: *VPow-def less-eq-V-def*)
using *less-eq-V-def* **apply** *fastforce*
done

lemma *VPow-le-VPow-iff* [*simp*]: $VPow\ a \leq VPow\ b \longleftrightarrow a \leq b$
by *auto*

lemma *elts-VPow*: $elts\ (VPow\ x) = set\ `\ Pow\ (elts\ x)$
by (*auto simp*: *VPow-def Pow*)

lemma *small-sup-iff* [*simp*]: $small\ (X \cup Y) \longleftrightarrow small\ X \wedge small\ Y$ **for** $X::V\ set$
by (*metis down elts-of-set small-Un-elts sup-ge1 sup-ge2*)

lemma *elts-sup-iff* [*simp*]: $elts\ (x \sqcup y) = elts\ x \cup elts\ y$
by (*simp add*: *sup-V-def*)

lemma *trad-foundation*:
assumes *z*: $z \neq 0$ **shows** $\exists\, w.\ w \in elts\ z \wedge w \sqcap z = 0$
using *foundation assms*
by (*simp add*: *wf-eq-minimal*) (*metis Int-emptyI equals0I inf-V-def set-of-elts zero-V-def*)

instantiation *V* :: *Sup*
begin
definition *Sup-V* **where** $Sup\text{-}V\ X \equiv if\ small\ X\ then\ set\ (Union\ (elts\ `\ X))\ else\ 0$
instance ..
end

instantiation *V* :: *Inf*
begin
definition *Inf-V* **where** $Inf\text{-}V\ X \equiv if\ X = \{\}\ then\ 0\ else\ set\ (Inter\ (elts\ `\ X))$
instance ..
end

lemma *V-disjoint-iff*: $x \sqcap y = 0 \longleftrightarrow elts\ x \cap elts\ y = \{\}$
by (*metis down elts-of-set inf-V-def inf-le1 zero-V-def*)

I've no idea why *bdd-above* is treated differently from *bdd-below*, but anyway

lemma *bdd-above-iff-small* [*simp*]: *bdd-above X* = *small X* **for** *X*:: *V set*
proof
  show *small X* **if** *bdd-above X*
  proof −
    obtain *a* **where** ∀ *x*∈*X*. *x* ≤ *a*
      using *that* ‹*bdd-above X*› *bdd-above-def* **by** *blast*
    then show *small X*
      by (*meson VPow-iff* ‹∀ *x*∈*X*. *x* ≤ *a*› *down subsetI*)
  qed
  show *bdd-above X*
    if *small X*
  proof −
    have ∀ *x*∈*X*. *x* ≤ ⋃ *X*
      by (*simp add*: *SUP-upper Sup-V-def Union less-eq-V-def that*)
    then show *?thesis*
      by (*meson bdd-above-def*)
  qed
qed

instantiation *V* :: *conditionally-complete-lattice*
begin

definition *bdd-below-V* **where** *bdd-below-V X* ≡ *X* ≠ {}

instance
  proof
  show ⋂ *X* ≤ *x* **if** *x* ∈ *X bdd-below X*
    for *x* :: *V* **and** *X* :: *V set*
    using *that* **by** (*auto simp*: *bdd-below-V-def Inf-V-def split*: *if-split-asm*)
  show *z* ≤ ⋂ *X*
    if *X* ≠ {} ⋀*x*. *x* ∈ *X* ⟹ *z* ≤ *x*
    for *X* :: *V set* **and** *z* :: *V*
    using *that*
   apply (*clarsimp simp add*: *bdd-below-V-def Inf-V-def less-eq-V-def split*: *if-split-asm*)
    by (*meson INT-subset-iff down eq-refl equals0I*)
  show *x* ≤ ⋃ *X* **if** *x* ∈ *X* **and** *bdd-above X* **for** *x* :: *V* **and** *X* :: *V set*
    using *that Sup-V-def* **by** *auto*
  show ⋃ *X* ≤ (*z*::*V*) **if** *X* ≠ {} ⋀*x*. *x* ∈ *X* ⟹ *x* ≤ *z* **for** *X* :: *V set* **and** *z* :: *V*
    using *that* **by** (*simp add*: *SUP-least Sup-V-def less-eq-V-def*)
qed
end

lemma *Sup-upper*: ⟦*x* ∈ *A*; *small A*⟧ ⟹ *x* ≤ ⋃ *A* **for** *A*:: *V set*
  by (*auto simp*: *Sup-V-def SUP-upper Union less-eq-V-def*)

lemma *Sup-least*:
  fixes *z*::*V* **shows** (⋀*x*. *x* ∈ *A* ⟹ *x* ≤ *z*) ⟹ ⋃ *A* ≤ *z*
  by (*auto simp*: *Sup-V-def SUP-least less-eq-V-def*)

lemma *Sup-empty* [*simp*]: $\bigsqcup\{\} = (0::V)$
using *Sup-V-def* **by** *auto*

lemma *elts-Sup* [*simp*]: *small* $X \Longrightarrow$ *elts* $(\bigsqcup\ X) = \bigcup(elts\ `\ X)$
by (*auto simp*: *Sup-V-def*)

lemma *sup-V-0-left* [*simp*]: $0 \sqcup a = a$ **and** *sup-V-0-right* [*simp*]: $a \sqcup 0 = a$ **for** $a::V$
by *auto*

lemma *Sup-V-insert*:
fixes $x::V$ **assumes** *small A* **shows** $\bigsqcup(insert\ x\ A) = x \sqcup \bigsqcup A$
by (*simp add*: *assms cSup-insert-If*)

lemma *Sup-Un-distrib*: $[\![small\ A; small\ B]\!] \Longrightarrow \bigsqcup(A \cup B) = \bigsqcup A \sqcup \bigsqcup B$ **for** $A::V$ *set*
by *auto*

lemma *SUP-sup-distrib*:
fixes $f :: V \Rightarrow V$
shows *small* $A \Longrightarrow (SUP\ x{\in}A.\ f\ x \sqcup g\ x) = \bigsqcup\ (f\ `\ A) \sqcup \bigsqcup\ (g\ `\ A)$
by (*force simp*:)

lemma *SUP-const* [*simp*]: $(SUP\ y \in A.\ a) = (if\ A = \{\}\ then\ (0::V)\ else\ a)$
by *simp*

lemma *cSUP-subset-mono*:
fixes $f :: 'a \Rightarrow V\ set$ **and** $g :: 'a \Rightarrow V\ set$
shows $[\![A \subseteq B; \bigwedge x.\ x \in A \Longrightarrow f\ x \leq g\ x]\!] \Longrightarrow \bigsqcup\ (f\ `\ A) \leq \bigsqcup\ (g\ `\ B)$
by (*simp add*: *SUP-subset-mono*)

lemma *mem-Sup-iff* [*iff*]: $x \in elts\ (\bigsqcup X) \longleftrightarrow x \in \bigcup\ (elts\ `\ X) \wedge small\ X$
using *Sup-V-def* **by** *auto*

lemma *cSUP-UNION*:
fixes $B :: V \Rightarrow V\ set$ **and** $f :: V \Rightarrow V$
assumes *ne*: *small A* **and** *bdd-UN*: *small* $(\bigcup x{\in}A.\ f\ `\ B\ x)$
shows $\bigsqcup(f\ `\ (\bigcup x{\in}A.\ B\ x)) = \bigsqcup((\lambda x.\ \bigsqcup(f\ `\ B\ x))\ `\ A)$
proof −
have *bdd*: $\bigwedge x.\ x \in A \Longrightarrow small\ (f\ `\ B\ x)$
using *bdd-UN subset-iff-less-eq-V*
by (*meson SUP-upper smaller-than-small*)
then have *bdd2*: *small* $((\lambda x.\ \bigsqcup(f\ `\ B\ x))\ `\ A)$
using *ne*(*1*) **by** *blast*
have $\bigsqcup(f\ `\ (\bigcup x{\in}A.\ B\ x)) \leq \bigsqcup((\lambda x.\ \bigsqcup(f\ `\ B\ x))\ `\ A)$
using *assms* **by** (*fastforce simp add*: *intro*!: *cSUP-least intro*: *cSUP-upper2 simp*: *bdd2 bdd*)
moreover have $\bigsqcup((\lambda x.\ \bigsqcup(f\ `\ B\ x))\ `\ A) \leq \bigsqcup(f\ `\ (\bigcup x{\in}A.\ B\ x))$

using *assms* **by** (*fastforce simp add*: *intro*!: *cSUP-least intro*: *cSUP-upper simp*: *image-UN bdd-UN*)
ultimately show *?thesis*
by (*rule order-antisym*)
qed

lemma *Sup-subset-mono*: *small* $B \Longrightarrow A \subseteq B \Longrightarrow Sup\ A \leq Sup\ B$ **for** $A :: V\ set$
by *auto*

lemma *Sup-le-iff*: *small* $A \Longrightarrow Sup\ A \leq a \longleftrightarrow (\forall\, x{\in}A.\ x \leq a)$ **for** $A :: V\ set$
by *auto*

lemma *SUP-le-iff*: *small* $(f\ `\ A) \Longrightarrow \bigsqcup(f\ `\ A) \leq u \longleftrightarrow (\forall\, x{\in}A.\ f\ x \leq u)$ **for** $f :: V \Rightarrow V$
by *blast*

lemma *Sup-eq-0-iff* [*simp*]: $\bigsqcup A = 0 \longleftrightarrow A \subseteq \{0\} \vee \neg\ small\ A$ **for** $A :: V\ set$
using *Sup-upper* **by** *fastforce*

lemma *Sup-Union-commute*:
fixes $f :: V \Rightarrow V\ set$
assumes *small* A $\bigwedge x.\ x{\in}A \Longrightarrow small\ (f\ x)$
shows $\bigsqcup\ (\bigcup x{\in}A.\ f\ x) = (SUP\ x{\in}A.\ \bigsqcup\ (f\ x))$
using *assms*
by (*force simp*: *subset-iff-less-eq-V intro*!: *antisym*)

lemma *Sup-eq-Sup*:
fixes $B :: V\ set$
assumes $B \subseteq A$ *small* A **and** $*$: $\bigwedge x.\ x \in A \Longrightarrow \exists\, y \in B.\ x \leq y$
shows $Sup\ A = Sup\ B$
proof −
have *small* B
using *assms subset-iff-less-eq-V* **by** *auto*
moreover have $\exists\, y{\in}B.\ u \in elts\ y$
if $x \in A$ $u \in elts\ x$ **for** $u\ x$
using *that* $*$ **by** *blast*
moreover have $\exists\, x{\in}A.\ v \in elts\ x$
if $y \in B$ $v \in elts\ y$ **for** $v\ y$
using *that* ‹$B \subseteq A$› **by** *blast*
ultimately show *?thesis*
using *assms* **by** *auto*
qed

1.3 Successor function

lemma *vinsert-not-empty* [*simp*]: *vinsert a A* $\neq 0$
and *empty-not-vinsert* [*simp*]: $0 \neq$ *vinsert a A*
by (*auto simp*: *vinsert-def*)

lemma *succ-not-0* [*simp*]: *succ n* ≠ *0* **and** *zero-not-succ* [*simp*]: *0* ≠ *succ n*
by (*auto simp*: *succ-def*)

instantiation *V* :: *zero-neq-one*
begin
instance
by *intro-classes* (*metis elts-0 elts-succ empty-iff insert-iff one-V-def set-of-elts*)
end

instantiation *V* :: *zero-less-one*
begin
instance
by *intro-classes* (*simp add*: *less-V-def*)
end

lemma *succ-ne-self* [*simp*]: *i* ≠ *succ i*
by (*metis elts-succ insertI1 mem-not-refl*)

lemma *succ-notin-self*: *succ i* ∉ *elts i*
using *elts-succ mem-not-refl* **by** *blast*

lemma *le-succE*: *succ i* ≤ *succ j* ⟹ *i* ≤ *j*
using *less-eq-V-def mem-not-sym* **by** *auto*

lemma *succ-inject-iff* [*iff*]: *succ i* = *succ j* ⟷ *i* = *j*
by (*simp add*: *dual-order.antisym le-succE*)

lemma *inj-succ*: *inj succ*
by (*simp add*: *inj-def*)

lemma *succ-neq-zero*: *succ x* ≠ *0*
by (*metis elts-0 elts-succ insert-not-empty*)

definition *pred* **where** *pred i* ≡ *THE j*. *i* = *succ j*

lemma *pred-succ* [*simp*]: *pred* (*succ i*) = *i*
by (*simp add*: *pred-def*)

1.4 Ordinals

definition *Transset* **where** *Transset x* ≡ ∀ *y* ∈ *elts x*. *y* ≤ *x*

definition *Ord* **where** *Ord x* ≡ *Transset x* ∧ (∀ *y* ∈ *elts x*. *Transset y*)

abbreviation *ON* **where** *ON* ≡ *Collect Ord*

1.4.1 Transitive sets

lemma *Transset-0* [*iff*]: *Transset 0*
by (*auto simp*: *Transset-def*)

lemma *Transset-succ* [*intro*]:
assumes *Transset x* **shows** *Transset* (*succ x*)
using *assms*
by (*auto simp*: *Transset-def succ-def less-eq-V-def*)

lemma *Transset-Sup*:
assumes $\bigwedge x.\ x \in X \Longrightarrow$ *Transset x* **shows** *Transset* ($\bigsqcup X$)
proof (*cases small X*)
case *True*
with *assms* **show** *?thesis*
by (*simp add*: *Transset-def*) (*meson Sup-upper assms dual-order.trans*)
qed (*simp add*: *Sup-V-def*)

lemma *Transset-sup*:
assumes *Transset x Transset y* **shows** *Transset* ($x \sqcup y$)
using *Transset-def assms* **by** *fastforce*

lemma *Transset-inf*: $[\![$*Transset i*; *Transset j*$]\!] \Longrightarrow$ *Transset* ($i \sqcap j$)
by (*simp add*: *Transset-def rev-vsubsetD*)

lemma *Transset-VPow*: *Transset*(i) $\Longrightarrow$ *Transset*(*VPow*(i))
by (*auto simp*: *Transset-def*)

lemma *Transset-Inf*: ($\bigwedge i.\ i \in A \Longrightarrow$ *Transset i*) $\Longrightarrow$ *Transset* ($\bigsqcap\ A$)
by (*force simp*: *Transset-def Inf-V-def*)

lemma *Transset-SUP*: ($\bigwedge x.\ x \in A \Longrightarrow$ *Transset* (*B x*)) $\Longrightarrow$ *Transset* ($\bigsqcup\ (B\ `\ A)$)
by (*metis Transset-Sup imageE*)

lemma *Transset-INT*: ($\bigwedge x.\ x \in A \Longrightarrow$ *Transset* (*B x*)) $\Longrightarrow$ *Transset* ($\bigsqcap\ (B\ `\ A)$)
by (*metis Transset-Inf imageE*)

1.4.2 Zero, successor, sups

lemma *Ord-0* [*iff*]: *Ord 0*
by (*auto simp*: *Ord-def*)

lemma *Ord-succ* [*intro*]:
assumes *Ord x* **shows** *Ord* (*succ x*)
using *assms* **by** (*auto simp*: *Ord-def*)

lemma *Ord-Sup*:
assumes $\bigwedge x.\ x \in X \Longrightarrow$ *Ord x* **shows** *Ord* ($\bigsqcup X$)
proof (*cases small X*)
case *True*
with *assms* **show** *?thesis*
by (*auto simp*: *Ord-def Transset-Sup*)
qed (*simp add*: *Sup-V-def*)

lemma *Ord-Union*:
assumes $\bigwedge x.\ x \in X \Longrightarrow Ord\ x\ small\ X$ **shows** $Ord\ (set\ (\bigcup\ (elts\ `\ X)))$
by (*metis Ord-Sup Sup-V-def assms*)

lemma *Ord-sup*:
assumes $Ord\ x\ Ord\ y$ **shows** $Ord\ (x \sqcup y)$
using *assms*
proof (*clarsimp simp*: *Ord-def*)
show $Transset\ (x \sqcup y) \wedge (\forall\, y{\in}elts\ x \cup elts\ y.\ Transset\ y)$
if $Transset\ x\ Transset\ y\ \forall\, y{\in}elts\ x.\ Transset\ y\ \forall\, y{\in}elts\ y.\ Transset\ y$
using *Ord-def Transset-sup assms* **by** *auto*
qed

lemma *big-ON* [*simp*]: $\neg\ small\ ON$
proof
assume *small ON*
then have $set\ ON \in ON$
by (*metis Ord-Union Ord-succ Sup-upper elts-Sup elts-succ insertI1 mem-Collect-eq mem-not-refl set-of-elts vsubsetD*)
then show *False*
by (*metis* ‹*small ON*› *elts-of-set mem-not-refl*)
qed

lemma *Ord-1* [*iff*]: *Ord 1*
using *Ord-succ one-V-def succ-def vinsert-def* **by** *fastforce*

lemma *OrdmemD*: $Ord\ k \Longrightarrow j \in elts\ k \Longrightarrow j < k$
using *Ord-def Transset-def less-V-def* **by** *auto*

lemma *Ord-trans*: $[\![\ i \in elts\ j;\ \ j \in elts\ k;\ \ Ord\ k\ ]\!] \Longrightarrow i \in elts\ k$
using *Ord-def Transset-def* **by** *blast*

lemma *mem-0-Ord*:
assumes *k*: *Ord k* **and** *knz*: $k \neq 0$ **shows** $0 \in elts\ k$
by (*metis Ord-def Transset-def inf.orderE k knz trad-foundation*)

lemma *Ord-in-Ord*: $[\![\ Ord\ k;\ \ m \in elts\ k\ ]\!] \Longrightarrow Ord\ m$
using *Ord-def Ord-trans* **by** *blast*

lemma *OrdI*: $[\![Transset\ i;\ \bigwedge x.\ x \in elts\ i \Longrightarrow Transset\ x]\!] \Longrightarrow Ord\ i$
by (*simp add*: *Ord-def*)

lemma *Ord-is-Transset*: $Ord\ i \Longrightarrow Transset\ i$
by (*simp add*: *Ord-def*)

lemma *Ord-contains-Transset*: $[\![Ord\ i;\ j \in elts\ i]\!] \Longrightarrow Transset\ j$
using *Ord-def* **by** *blast*

lemma *ON-imp-Ord*:
assumes $H \subseteq ON\ x \in H$
shows *Ord x*
using *assms* **by** *blast*

lemma *elts-subset-ON*: *Ord* $\alpha \Longrightarrow$ *elts* $\alpha \subseteq ON$
using *Ord-in-Ord* **by** *blast*

lemma *Transset-pred* [*simp*]: *Transset* $x \Longrightarrow \bigsqcup(elts\ (succ\ x)) = x$
by (*fastforce simp*: *Transset-def*)

lemma *Ord-pred* [*simp*]: *Ord* $\beta \Longrightarrow \bigsqcup\ (insert\ \beta\ (elts\ \beta)) = \beta$
using *Ord-def Transset-pred* **by** *auto*

1.4.3 Induction, Linearity, etc.

lemma *Ord-induct* [*consumes 1*, *case-names step*]:
assumes *k*: *Ord k*
and *step*: $\bigwedge x.[\![\ Ord\ x;\ \bigwedge y.\ y \in elts\ x \Longrightarrow P\ y\ ]\!] \Longrightarrow P\ x$
shows *P k*
using *foundation k*
proof (*induction k rule*: *wf-induct-rule*)
case (*less x*)
then show *?case*
using *Ord-in-Ord local.step* **by** *auto*
qed

Comparability of ordinals

lemma *Ord-linear*: *Ord k* $\Longrightarrow$ *Ord l* $\Longrightarrow$ $k \in elts\ l \vee k{=}l \vee l \in elts\ k$
proof (*induct k arbitrary*: *l rule*: *Ord-induct*)
case (*step k*)
note *step-k* = *step*
show *?case* **using** ‹*Ord l*›
proof (*induct l rule*: *Ord-induct*)
case (*step l*)
thus *?case* **using** *step-k*
by (*metis Ord-trans V-equalityI*)
qed
qed

The trichotomy law for ordinals

lemma *Ord-linear-lt*:
assumes *Ord k Ord l*
obtains (*lt*) $k < l$ | (*eq*) $k{=}l$ | (*gt*) $l < k$
using *Ord-linear OrdmemD assms* **by** *blast*

lemma *Ord-linear2*:
assumes *Ord k Ord l*
obtains (*lt*) $k < l$ | (*ge*) $l \leq k$

by (*metis Ord-linear-lt eq-refl assms order.strict-implies-order*)

lemma *Ord-linear-le*:
assumes *Ord k Ord l*
obtains (*le*) $k \leq l$ | (*ge*) $l \leq k$
by (*meson Ord-linear2 le-less assms*)

lemma *union-less-iff* [*simp*]: $[\![Ord\ i;\ Ord\ j]\!] \Longrightarrow i \sqcup j < k \longleftrightarrow i<k \wedge j<k$
by (*metis Ord-linear-le le-iff-sup sup.order-iff sup.strict-boundedE*)

lemma *Ord-mem-iff-lt*: *Ord k* $\Longrightarrow$ *Ord l* $\Longrightarrow$ $k \in elts\ l \longleftrightarrow k < l$
by (*metis Ord-linear OrdmemD less-le-not-le*)

lemma *Ord-Collect-lt*: *Ord* $\alpha \Longrightarrow \{\xi.\ Ord\ \xi \wedge \xi < \alpha\} = elts\ \alpha$
by (*auto simp flip*: *Ord-mem-iff-lt elim*: *Ord-in-Ord OrdmemD*)

lemma *Ord-not-less*: $[\![Ord\ x;\ Ord\ y]\!] \Longrightarrow \neg\ x < y \longleftrightarrow y \leq x$
by (*metis* (*no-types*) *Ord-linear2 leD*)

lemma *Ord-not-le*: $[\![Ord\ x;\ Ord\ y]\!] \Longrightarrow \neg\ x \leq y \longleftrightarrow y < x$
by (*metis* (*no-types*) *Ord-linear2 leD*)

lemma *le-succ-iff*: *Ord i* $\Longrightarrow$ *Ord j* $\Longrightarrow$ $succ\ i \leq succ\ j \longleftrightarrow i \leq j$
by (*metis Ord-linear-le Ord-succ le-succE order-antisym*)

lemma *succ-le-iff*: *Ord i* $\Longrightarrow$ *Ord j* $\Longrightarrow$ $succ\ i \leq j \longleftrightarrow i < j$
using *Ord-mem-iff-lt dual-order.strict-implies-order less-eq-V-def* **by** *fastforce*

lemma *succ-in-Sup-Ord*:
assumes *eq*: $succ\ \beta = \bigsqcup A$ **and** *small A* $A \subseteq ON\ Ord\ \beta$
shows $succ\ \beta \in A$
proof −
have $\neg \bigsqcup A \leq \beta$
using *eq* ‹*Ord* β› *succ-le-iff* **by** *fastforce*
then show *?thesis*
using *assms* **by** (*metis Ord-linear2 Sup-least Sup-upper eq-iff mem-Collect-eq subsetD succ-le-iff*)
qed

lemma *in-succ-iff*: *Ord i* $\Longrightarrow$ $j \in elts\ (ZFC\text{-}in\text{-}HOL.succ\ i) \longleftrightarrow Ord\ j \wedge j \leq i$
by (*metis Ord-in-Ord Ord-mem-iff-lt Ord-not-le Ord-succ succ-le-iff*)

lemma *zero-in-succ* [*simp*,*intro*]: *Ord i* $\Longrightarrow$ $0 \in elts\ (succ\ i)$
using *mem-0-Ord* **by** *auto*

lemma *Ord-finite-Sup*: $[\![finite\ A;\ A \subseteq ON;\ A \neq \{\}]\!] \Longrightarrow \bigsqcup A \in A$
proof (*induction A rule*: *finite-induct*)
case (*insert x A*)
then have ∗: *small A* $A \subseteq ON\ Ord\ x$

by (*auto simp add*: *ZFC-in-HOL.finite-imp-small insert.hyps*)
show *?case*
proof (*cases* $A = \{\}$)
case *False*
then have $\bigsqcup A \in A$
using *insert* **by** *blast*
then have $\bigsqcup A \leq x$ **if** $x \sqcup \bigsqcup A \notin A$
using $*$ **by** (*metis ON-imp-Ord Ord-linear-le sup.absorb2 that*)
then show *?thesis*
by (*fastforce simp*: ‹*small A*› *Sup-V-insert*)
qed *auto*
qed *auto*

1.4.4 The natural numbers

primrec *ord-of-nat* :: *nat* $\Rightarrow$ *V* **where**
ord-of-nat 0 = *0*
| *ord-of-nat* (*Suc n*) = *succ* (*ord-of-nat n*)

lemma *ord-of-nat-eq-initial*: *ord-of-nat n* = *set* (*ord-of-nat* ‘ {..<*n*})
by (*induction n*) (*auto simp*: *lessThan-Suc succ-def*)

lemma *mem-ord-of-nat-iff* [*simp*]: $x \in$ *elts* (*ord-of-nat n*) $\longleftrightarrow$ ($\exists\, m{<}n.\ x$ = *ord-of-nat m*)
by (*subst ord-of-nat-eq-initial*) *auto*

lemma *elts-ord-of-nat*: *elts* (*ord-of-nat k*) = *ord-of-nat* ‘ {..<*k*}
by *auto*

lemma *Ord-equality*: *Ord i* $\Longrightarrow$ *i* = $\bigsqcup$ (*succ* ‘ *elts i*)
by (*force intro*: *Ord-trans*)

lemma *Ord-ord-of-nat* [*simp*]: *Ord* (*ord-of-nat k*)
by (*induct k*, *auto*)

lemma *ord-of-nat-equality*: *ord-of-nat n* = $\bigsqcup$ ((*succ* $\circ$ *ord-of-nat*) ‘ {..<*n*})
by (*metis Ord-equality Ord-ord-of-nat elts-of-set image-comp small-image-nat-V ord-of-nat-eq-initial*)

definition ω :: *V* **where** $\omega \equiv$ *set* (*range ord-of-nat*)

lemma *elts-ω*: *elts* ω = {α. $\exists\, n.\ \alpha$ = *ord-of-nat n*}
by (*auto simp*: ω*-def image-iff*)

lemma *nat-into-Ord* [*simp*]: *n* $\in$ *elts* ω $\Longrightarrow$ *Ord n*
by (*metis Ord-ord-of-nat ω-def elts-of-set image-iff inf*)

lemma *Sup-ω*: $\bigsqcup$(*elts* ω) = ω
unfolding ω*-def* **by** *force*

lemma *Ord-ω* [*iff*]: *Ord ω*
by (*metis Ord-Sup Sup-ω nat-into-Ord*)

lemma *zero-in-omega* [*iff*]: *0 ∈ elts ω*
by (*metis ω-def elts-of-set inf ord-of-nat.simps*(*1*) *rangeI*)

lemma *succ-in-omega* [*simp*]: *n ∈ elts ω ⟹ succ n ∈ elts ω*
by (*metis ω-def elts-of-set image-iff small-image-nat-V ord-of-nat.simps*(*2*) *rangeI*)

lemma *ord-of-eq-0*: *ord-of-nat j = 0 ⟹ j = 0*
by (*induct j*) (*auto simp*: *succ-neq-zero*)

lemma *ord-of-nat-le-omega*: *ord-of-nat n ≤ ω*
by (*metis Sup-ω ZFC-in-HOL.Sup-upper ω-def elts-of-set inf rangeI*)

lemma *ord-of-eq-0-iff* [*simp*]: *ord-of-nat n = 0 ⟷ n=0*
by (*auto simp*: *ord-of-eq-0*)

lemma *ord-of-nat-inject* [*iff*]: *ord-of-nat i = ord-of-nat j ⟷ i=j*
proof (*induct i arbitrary*: *j*)
case *0* **show** *?case*
using *ord-of-eq-0* **by** *auto*
next
case (*Suc i*) **then show** *?case*
by *auto* (*metis elts-0 elts-succ insert-not-empty not0-implies-Suc ord-of-nat.simps succ-inject-iff*)
qed

corollary *inj-ord-of-nat*: *inj ord-of-nat*
by (*simp add*: *linorder-injI*)

corollary *countable*:
assumes *countable X* **shows** *small X*
proof −
have *X ⊆ range* (*from-nat-into X*)
by (*simp add*: *assms subset-range-from-nat-into*)
then show *?thesis*
by (*meson inf-raw inj-ord-of-nat replacement small-def smaller-than-small*)
qed

corollary *infinite-ω*: *infinite* (*elts ω*)
using *range-inj-infinite* [*of ord-of-nat*]
by (*simp add*: *ω-def inj-ord-of-nat*)

corollary *ord-of-nat-mono-iff* [*iff*]: *ord-of-nat i ≤ ord-of-nat j ⟷ i ≤ j*
by (*metis Ord-def Ord-ord-of-nat Transset-def eq-iff mem-ord-of-nat-iff not-less ord-of-nat-inject*)

corollary *ord-of-nat-strict-mono-iff* [*iff*]: *ord-of-nat i* < *ord-of-nat j* ⟷ *i* < *j*
by (*simp add*: *less-le-not-le*)

lemma *small-image-nat* [*simp*]:
fixes *N* :: *nat set* **shows** *small* (*g* ‘ *N*)
by (*simp add*: *countable*)

lemma *finite-Ord-omega*: *α* ∈ *elts ω* ⟹ *finite* (*elts α*)
proof (*clarsimp simp add*: *ω-def*)
show *finite* (*elts* (*ord-of-nat n*)) **if** *α* = *ord-of-nat n* **for** *n*
using *that* **by** (*simp add*: *ord-of-nat-eq-initial* [*of n*])
qed

lemma *infinite-Ord-omega*: *Ord α* ⟹ *infinite* (*elts α*) ⟹ *ω* ≤ *α*
by (*meson Ord-ω Ord-linear2 Ord-mem-iff-lt finite-Ord-omega*)

lemma *ord-of-minus-1*: *n* > *0* ⟹ *ord-of-nat n* = *succ* (*ord-of-nat* (*n* − *1*))
by (*metis Suc-diff-1 ord-of-nat.simps*(*2*))

lemma *card-ord-of-nat* [*simp*]: *card* (*elts* (*ord-of-nat m*)) = *m*
by (*induction m*) (*auto simp*: *ω-def finite-Ord-omega*)

lemma *ord-of-nat-ω* [*iff*]:*ord-of-nat n* ∈ *elts ω*
by (*simp add*: *ω-def*)

lemma *succ-ω-iff* [*iff*]: *succ n* ∈ *elts ω* ⟷ *n* ∈ *elts ω*
by (*metis Ord-ω OrdmemD elts-vinsert insert-iff less-V-def succ-def succ-in-omega vsubsetD*)

lemma *ω-gt0* [*simp*]: *ω* > *0*
by (*simp add*: *OrdmemD*)

lemma *ω-gt1* [*simp*]: *ω* > *1*
by (*simp add*: *OrdmemD one-V-def*)

1.4.5 Limit ordinals

definition *Limit* :: *V*⇒*bool*
where *Limit i* ≡ *Ord i* ∧ *0* ∈ *elts i* ∧ (∀ *y*. *y* ∈ *elts i* ⟶ *succ y* ∈ *elts i*)

lemma *zero-not-Limit* [*iff*]: ¬ *Limit 0*
by (*simp add*: *Limit-def*)

lemma *not-succ-Limit* [*simp*]: ¬ *Limit*(*succ i*)
by (*metis Limit-def Ord-mem-iff-lt elts-succ insertI1 less-irrefl*)

lemma *Limit-is-Ord*: *Limit ξ* ⟹ *Ord ξ*
by (*simp add*: *Limit-def*)

lemma *succ-in-Limit-iff*: *Limit* $\xi \Longrightarrow$ *succ* $\alpha \in$ *elts* $\xi \longleftrightarrow \alpha \in$ *elts* ξ
by (*metis Limit-def OrdmemD elts-succ insertI1 less-V-def vsubsetD*)

lemma *Limit-eq-Sup-self* [*simp*]: *Limit* $i \Longrightarrow$ *Sup* (*elts* i) $= i$
apply (*rule order-antisym*)
apply (*simp add*: *Limit-def Ord-def Transset-def Sup-least*)
by (*metis Limit-def Ord-equality Sup-V-def SUP-le-iff Sup-upper small-elts*)

lemma *zero-less-Limit*: *Limit* $\beta \Longrightarrow 0 < \beta$
by (*simp add*: *Limit-def OrdmemD*)

lemma *non-Limit-ord-of-nat* [*iff*]: $\neg$ *Limit* (*ord-of-nat* m)
by (*metis Limit-def mem-ord-of-nat-iff not-succ-Limit ord-of-eq-0-iff ord-of-minus-1*)

lemma *Limit-omega* [*iff*]: *Limit* ω
by (*simp add*: *Limit-def*)

lemma *omega-nonzero* [*simp*]: $\omega \neq 0$
using *Limit-omega* **by** *fastforce*

lemma *Ord-cases-lemma*:
assumes *Ord* k **shows** $k = 0 \vee (\exists j.\ k =$ *succ* $j) \vee$ *Limit* k
proof (*cases Limit* k)
case *False*
have *succ* $j \in$ *elts* k **if** $\forall j.\ k \neq$ *succ* $j\ j \in$ *elts* k **for** j
by (*metis Ord-in-Ord Ord-linear Ord-succ assms elts-succ insertE mem-not-sym that*)
with *assms* **show** *?thesis*
by (*auto simp*: *Limit-def mem-0-Ord*)
qed *auto*

lemma *Ord-cases* [*cases type*: V, *case-names 0 succ limit*]:
assumes *Ord* k
obtains $k = 0 \mid l$ **where** *Ord* l *succ* $l = k \mid$ *Limit* k
by (*metis assms Ord-cases-lemma Ord-in-Ord elts-succ insertI1*)

lemma *non-succ-LimitI*:
assumes $i{\neq}0$ *Ord*(i) $\bigwedge y.$ *succ*(y) $\neq i$
shows *Limit*(i)
using *Ord-cases-lemma assms* **by** *blast*

lemma *Ord-induct3* [*consumes 1*, *case-names 0 succ Limit*, *induct type*: V]:
assumes α: *Ord* α
and P: $P\ 0$ $\bigwedge\alpha.$ $[\![$*Ord* α; $P\ \alpha]\!] \Longrightarrow P$ (*succ* α)
$\bigwedge\alpha.$ $[\![$*Limit* α; $\bigwedge\xi.\ \xi \in$ *elts* $\alpha \Longrightarrow P\ \xi]\!] \Longrightarrow P$ (*SUP* $\xi \in$ *elts* $\alpha.\ \xi$)
shows $P\ \alpha$
using α
proof (*induction* α *rule*: *Ord-induct*)
case (*step* α)

then show *?case*
by (*metis Limit-eq-Sup-self Ord-cases P elts-succ image-ident insertI1*)
qed

1.4.6 Properties of LEAST for ordinals

lemma
assumes *Ord k P k*
shows *Ord-LeastI*: *P* (*LEAST i. Ord i* ∧ *P i*) **and** *Ord-Least-le*: (*LEAST i. Ord i* ∧ *P i*) ≤ *k*
proof −
have *P* (*LEAST i. Ord i* ∧ *P i*) ∧ (*LEAST i. Ord i* ∧ *P i*) ≤ *k*
using *assms*
proof (*induct k rule*: *Ord-induct*)
case (*step x*) **then have** *P x* **by** *simp*
show *?case* **proof** (*rule classical*)
assume *assm*: ¬ (*P* (*LEAST a. Ord a* ∧ *P a*) ∧ (*LEAST a. Ord a* ∧ *P a*) ≤ *x*)
have ⋀*y*. *Ord y* ∧ *P y* ⟹ *x* ≤ *y*
proof (*rule classical*)
fix *y*
assume *y*: *Ord y* ∧ *P y* ¬ *x* ≤ *y*
with *step* **obtain** *P* (*LEAST a. Ord a* ∧ *P a*) **and** *le*: (*LEAST a. Ord a* ∧ *P a*) ≤ *y*
by (*meson Ord-linear2 Ord-mem-iff-lt*)
with *assm* **have** *x* < (*LEAST a. Ord a* ∧ *P a*)
by (*meson Ord-linear-le y order.trans* ‹*Ord x*›)
then show *x* ≤ *y*
using *le* **by** *auto*
qed
then have *Least*: (*LEAST a. Ord a* ∧ *P a*) = *x*
by (*simp add*: *Least-equality* ‹*Ord x*› *step.prems*)
with ‹*P x*› **show** *?thesis* **by** *simp*
qed
qed
then show *P* (*LEAST i. Ord i* ∧ *P i*) **and** (*LEAST i. Ord i* ∧ *P i*) ≤ *k* **by** *auto*
qed

lemma *Ord-Least*:
assumes *Ord k P k*
shows *Ord* (*LEAST i. Ord i* ∧ *P i*)
proof −
have *Ord* (*LEAST i. Ord i* ∧ (*Ord i* ∧ *P i*))
using *Ord-LeastI* [**where** *P* = λ*i*. *Ord i* ∧ *P i*] *assms* **by** *blast*
then show *?thesis*
by *simp*
qed

— The following 3 lemmas are due to Brian Huffman

lemma *Ord-LeastI-ex*: $\exists\, i.\ Ord\ i \wedge P\ i \Longrightarrow P\ (LEAST\ i.\ Ord\ i \wedge P\ i)$
using *Ord-LeastI* **by** *blast*

lemma *Ord-LeastI2*:
$[\![Ord\ a;\ P\ a;\ \bigwedge x.\ [\![Ord\ x;\ P\ x]\!] \Longrightarrow Q\ x]\!] \Longrightarrow Q\ (LEAST\ i.\ Ord\ i \wedge P\ i)$
by (*blast intro*: *Ord-LeastI Ord-Least*)

lemma *Ord-LeastI2-ex*:
$\exists\, a.\ Ord\ a \wedge P\ a \Longrightarrow (\bigwedge x.\ [\![Ord\ x;\ P\ x]\!] \Longrightarrow Q\ x) \Longrightarrow Q\ (LEAST\ i.\ Ord\ i \wedge P\ i)$
by (*blast intro*: *Ord-LeastI-ex Ord-Least*)

lemma *Ord-LeastI2-wellorder*:
assumes *Ord a P a*
and $\bigwedge a.\ [\![\ P\ a;\ \forall\, b.\ Ord\ b \wedge P\ b \longrightarrow a \leq b\ ]\!] \Longrightarrow Q\ a$
shows $Q\ (LEAST\ i.\ Ord\ i \wedge P\ i)$
proof (*rule LeastI2-order*)
show $Ord\ (LEAST\ i.\ Ord\ i \wedge P\ i) \wedge P\ (LEAST\ i.\ Ord\ i \wedge P\ i)$
using *Ord-Least Ord-LeastI assms* **by** *auto*
next
fix *y* **assume** $Ord\ y \wedge P\ y$ **thus** $(LEAST\ i.\ Ord\ i \wedge P\ i) \leq y$
by (*simp add*: *Ord-Least-le*)
next
fix *x* **assume** $Ord\ x \wedge P\ x\ \forall\, y.\ Ord\ y \wedge P\ y \longrightarrow x \leq y$ **thus** $Q\ x$
by (*simp add*: *assms*(*3*))
qed

lemma *Ord-LeastI2-wellorder-ex*:
assumes $\exists\, x.\ Ord\ x \wedge P\ x$
and $\bigwedge a.\ [\![\ P\ a;\ \forall\, b.\ Ord\ b \wedge P\ b \longrightarrow a \leq b\ ]\!] \Longrightarrow Q\ a$
shows $Q\ (LEAST\ i.\ Ord\ i \wedge P\ i)$
using *assms* **by** *clarify* (*blast intro*!: *Ord-LeastI2-wellorder*)

lemma *not-less-Ord-Least*: $[\![k < (LEAST\ x.\ Ord\ x \wedge P\ x);\ Ord\ k]\!] \Longrightarrow \neg\ P\ k$
using *Ord-Least-le less-le-not-le* **by** *auto*

lemma *exists-Ord-Least-iff*: $(\exists\, \alpha.\ Ord\ \alpha \wedge P\ \alpha) \longleftrightarrow (\exists\, \alpha.\ Ord\ \alpha \wedge P\ \alpha \wedge (\forall\, \beta < \alpha.\ Ord\ \beta \longrightarrow \neg\ P\ \beta))$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?rhs* **thus** *?lhs* **by** *blast*
next
assume *H*: *?lhs* **then obtain** α **where** α: $Ord\ \alpha\ P\ \alpha$ **by** *blast*
let $?x = LEAST\ \alpha.\ Ord\ \alpha \wedge P\ \alpha$
have *Ord ?x*
by (*metis Ord-Least* α)
moreover
{ fix β **assume** *m*: $\beta < ?x\ Ord\ \beta$
from *not-less-Ord-Least*[*OF m*] **have** $\neg\ P\ \beta$ **. }**
ultimately show *?rhs*

using *Ord-LeastI-ex*[*OF H*] **by** *blast*
qed

lemma *Ord-mono-imp-increasing*:
assumes *fun-hD*: $h \in D \to D$
and *mono-h*: *strict-mono-on h D*
and $D \subseteq ON$ **and** ν: $\nu \in D$
shows $\nu \leq h\ \nu$
proof (*rule ccontr*)
assume *non*: $\neg\ \nu \leq h\ \nu$
define μ **where** $\mu \equiv LEAST\ \mu.\ Ord\ \mu \wedge \neg\ \mu \leq h\ \mu \wedge \mu \in D$
have *Ord* ν
using ν ‹$D \subseteq ON$› **by** *blast*
then have μ: $\neg\ \mu \leq h\ \mu \wedge \mu \in D$
unfolding μ-*def* **by** (*rule Ord-LeastI*) (*simp add*: ν *non*)
have *Ord* ($h\ \nu$)
using *assms* **by** *auto*
then have *Ord* (h ($h\ \nu$))
by (*meson ON-imp-Ord* ν *assms funcset-mem*)
have *Ord* μ
using μ ‹$D \subseteq ON$› **by** *blast*
then have $h\ \mu < \mu$
by (*metis ON-imp-Ord Ord-linear2 PiE* μ ‹$D \subseteq ON$› *fun-hD*)
then have $\neg\ h\ \mu \leq h\ (h\ \mu)$
using μ *fun-hD mono-h* **by** (*force simp*: *strict-mono-on-def*)
moreover have $*$: $h\ \mu \in D$
using μ *fun-hD* **by** *auto*
moreover have *Ord* ($h\ \mu$)
using ‹$D \subseteq ON$› $*$ **by** *blast*
ultimately have $\mu \leq h\ \mu$
by (*simp add*: μ-*def Ord-Least-le*)
then show *False*
using μ **by** *blast*
qed

lemma *le-Sup-iff*:
assumes $A \subseteq ON$ *Ord x small A* **shows** $x \leq \bigsqcup A \longleftrightarrow (\forall\, y \in ON.\ y{<}x \longrightarrow (\exists\, a{\in}A.\ y < a))$
proof (*intro iffI ballI impI*)
show $\exists\, a{\in}A.\ y < a$
if $x \leq \bigsqcup\ A$ $y \in ON$ $y < x$
for y
proof −
have $\neg \bigsqcup\ A \leq y$ *Ord y*
using *that* **by** *auto*
then show *?thesis*
by (*metis Ord-linear2 Sup-least* ‹$A \subseteq ON$› *mem-Collect-eq subset-eq*)
qed
show $x \leq \bigsqcup\ A$

if $\forall\, y{\in}ON.\ y < x \longrightarrow (\exists\, a{\in}A.\ y < a)$
using *that assms*
by (*metis Ord-Sup Ord-linear-le Sup-upper less-le-not-le mem-Collect-eq subsetD*)
qed

lemma *le-SUP-iff*: $[\![f\ `\ A \subseteq ON;\ Ord\ x;\ small\ A]\!] \Longrightarrow x \leq \bigsqcup(f\ `\ A) \longleftrightarrow (\forall\, y \in ON.\ y{<}x \longrightarrow (\exists\, i{\in}A.\ y < f\ i))$
by (*simp add*: *le-Sup-iff*)

1.5 Transfinite Recursion and the V-levels

definition *transrec* :: $[[V{\Rightarrow}V, V]{\Rightarrow}V,\ V] \Rightarrow V$
where *transrec H a* $\equiv$ *wfrec* $\{(x,y).\ x \in elts\ y\}$ *H a*

lemma *transrec*: *transrec H a* $=$ *H* $(\lambda x \in elts\ a.\ transrec\ H\ x)$ *a*
proof −
have (*cut* (*wfrec* $\{(x,\ y).\ x \in elts\ y\}$ *H*) $\{(x,\ y).\ x \in elts\ y\}$ *a*)
$=$ $(\lambda x{\in}elts\ a.\ wfrec\ \{(x,\ y).\ x \in elts\ y\}\ H\ x)$
by (*force simp*: *cut-def*)
then show *?thesis*
unfolding *transrec-def*
by (*simp add*: *foundation wfrec*)
qed

Avoids explosions in proofs; resolve it with a meta-level definition

lemma *def-transrec*:
$[\![\bigwedge x.\ f\ x \equiv transrec\ H\ x]\!] \Longrightarrow f\ a = H(\lambda x \in elts\ a.\ f\ x)\ a$
by (*metis restrict-ext transrec*)

lemma *eps-induct* [*case-names step*]:
assumes $\bigwedge x.\ (\bigwedge y.\ y \in elts\ x \Longrightarrow P\ y) \Longrightarrow P\ x$
shows *P a*
using *wf-induct* [*OF foundation*] *assms* **by** *auto*

definition *Vfrom* :: $[V, V] \Rightarrow V$
where *Vfrom a* $\equiv$ *transrec* $(\lambda f\ x.\ a \sqcup \bigsqcup((\lambda y.\ VPow(f\ y))\ `\ elts\ x))$

abbreviation *Vset* :: $V \Rightarrow V$ **where** *Vset* $\equiv$ *Vfrom 0*

lemma *Vfrom*: *Vfrom a i* $=$ $a \sqcup \bigsqcup((\lambda j.\ VPow(Vfrom\ a\ j))\ `\ elts\ i)$
apply (*subst Vfrom-def*)
apply (*subst transrec*)
using *Vfrom-def* **by** *auto*

lemma *Vfrom-0* [*simp*]: *Vfrom a 0* $=$ *a*
by (*subst Vfrom*) *auto*

lemma *Vset*: $Vset\ i = \bigsqcup((\lambda j.\ VPow(Vset\ j))\ `\ elts\ i)$
by (*subst Vfrom*) *auto*

lemma *Vfrom-mono1*:
assumes $a \leq b$ **shows** $Vfrom\ a\ i \leq Vfrom\ b\ i$
proof (*induction i rule*: *eps-induct*)
case (*step i*)
then have $a \sqcup (SUP\ j{\in}elts\ i.\ VPow\ (Vfrom\ a\ j)) \leq b \sqcup (SUP\ j{\in}elts\ i.\ VPow\ (Vfrom\ b\ j))$
by (*intro sup-mono cSUP-subset-mono* ‹$a \leq b$›) *auto*
then show *?case*
by (*metis Vfrom*)
qed

lemma *Vfrom-mono2*: $Vfrom\ a\ i \leq Vfrom\ a\ (i \sqcup j)$
proof (*induction arbitrary*: *j rule*: *eps-induct*)
case (*step i*)
then have $a \sqcup (SUP\ j{\in}elts\ i.\ VPow\ (Vfrom\ a\ j))$
$\leq a \sqcup (SUP\ j{\in}elts\ (i \sqcup j).\ VPow\ (Vfrom\ a\ j))$
by (*intro sup-mono cSUP-subset-mono order-refl*) *auto*
then show *?case*
by (*metis Vfrom*)
qed

lemma *Vfrom-mono*: $[\![Ord\ i;\ a{\leq}b;\ i{\leq}j]\!] \Longrightarrow Vfrom\ a\ i \leq Vfrom\ b\ j$
by (*metis* (*no-types*) *Vfrom-mono1 Vfrom-mono2 dual-order.trans sup.absorb-iff2*)

lemma *Transset-Vfrom*: $Transset(A) \Longrightarrow Transset(Vfrom\ A\ i)$
proof (*induction i rule*: *eps-induct*)
case (*step i*)
then show *?case*
by (*metis Transset-SUP Transset-VPow Transset-sup Vfrom*)
qed

lemma *Transset-Vset* [*simp*]: $Transset(Vset\ i)$
by (*simp add*: *Transset-Vfrom*)

lemma *Vfrom-sup*: $Vfrom\ a\ (i \sqcup j) = Vfrom\ a\ i \sqcup Vfrom\ a\ j$
proof (*rule order-antisym*)
show $Vfrom\ a\ (i \sqcup j) \leq Vfrom\ a\ i \sqcup Vfrom\ a\ j$
by (*simp add*: *Vfrom* [*of a i* ⊔ *j*] *Vfrom* [*of a i*] *Vfrom* [*of a j*] *Sup-Un-distrib image-Un sup.assoc sup.left-commute*)
show $Vfrom\ a\ i \sqcup Vfrom\ a\ j \leq Vfrom\ a\ (i \sqcup j)$
by (*metis Vfrom-mono2 le-supI sup-commute*)
qed

lemma *Vfrom-succ-Ord*:
assumes *Ord i* **shows** $Vfrom\ a\ (succ\ i) = a \sqcup VPow(Vfrom\ a\ i)$
proof (*cases i = 0*)

```
  case True
  then show ?thesis
    by (simp add: Vfrom [of - succ 0])
next
  case False
  have *: (SUP x∈elts i. VPow (Vfrom a x)) ≤ VPow (Vfrom a i)
  proof (rule cSup-least)
    show (λx. VPow (Vfrom a x)) ‘ elts i ≠ {}
      using False by auto
    show x ≤ VPow (Vfrom a i) if x ∈ (λx. VPow (Vfrom a x)) ‘ elts i for x
      using that
     by clarsimp (meson Ord-in-Ord Ord-linear-le Vfrom-mono assms mem-not-refl
order-refl vsubsetD)
  qed
  show ?thesis
  proof (rule Vfrom [THEN trans])
    show a ⊔ (SUP j∈elts (succ i). VPow (Vfrom a j)) = a ⊔ VPow (Vfrom a i)
      using assms
      by (intro sup-mono order-antisym) (auto simp: Sup-V-insert *)
  qed
qed

lemma Vset-succ: Ord i ⟹ Vset(succ(i)) = VPow(Vset(i))
  by (simp add: Vfrom-succ-Ord)

lemma Vfrom-Sup:
  assumes X ≠ {} small X
  shows Vfrom a (Sup X) = (SUP y∈X. Vfrom a y)
proof (rule order-antisym)
  have Vfrom a (⋃ X) = a ⊔ (SUP j∈elts (⋃ X). VPow (Vfrom a j))
    by (metis Vfrom)
  also have ... ≤ ⋃ (Vfrom a ‘ X)
  proof −
    have a ≤ ⋃ (Vfrom a ‘ X)
      by (metis Vfrom all-not-in-conv assms bdd-above-iff-small cSUP-upper2 re-
placement sup-ge1)
    moreover have (SUP j∈elts (⋃ X). VPow (Vfrom a j)) ≤ ⋃ (Vfrom a ‘ X)
    proof −
      have VPow (Vfrom a x) ≤ ⋃ (Vfrom a ‘ X)
        if y ∈ X x ∈ elts y for x y
      proof −
        have VPow (Vfrom a x) ≤ Vfrom a y
            by (metis Vfrom bdd-above-iff-small cSUP-upper2 le-supI2 order-refl
replacement small-elts that(2))
        also have ... ≤ ⋃ (Vfrom a ‘ X)
          using assms that by (force intro: cSUP-upper)
        finally show ?thesis .
      qed
      then show ?thesis
```

by (*simp add*: *SUP-le-iff* ‹*small X*›)
qed
ultimately show *?thesis*
by *auto*
qed
finally show *Vfrom a* (⨆ *X*) ≤ ⨆ (*Vfrom a* ‘ *X*) **.**
have ⋀*x*. *x* ∈ *X* ⟹
a ⊔ (*SUP j*∈*elts x*. *VPow* (*Vfrom a j*))
≤ *a* ⊔ (*SUP j*∈*elts* (⨆ *X*). *VPow* (*Vfrom a j*))
using *cSUP-subset-mono* ‹*small X*› **by** *auto*
then show ⨆ (*Vfrom a* ‘ *X*) ≤ *Vfrom a* (⨆ *X*)
by (*metis Vfrom assms*(*1*) *cSUP-least*)
qed

lemma *Limit-Vfrom-eq*:
Limit(*i*) ⟹ *Vfrom a i* = (*SUP y* ∈ *elts i*. *Vfrom a y*)
by (*metis Limit-def Limit-eq-Sup-self Vfrom-Sup ex-in-conv small-elts*)

end

2 Cartesian products, Disjoint Sums, Ranks, Cardinals

theory *ZFC-Cardinals*
imports *ZFC-in-HOL*

begin

declare [[*coercion-enabled*]]
declare [[*coercion ord-of-nat* :: *nat* ⇒ *V*]]

2.1 Ordered Pairs

lemma *singleton-eq-iff* [*iff*]: *set* {*a*} = *set* {*b*} ⟷ *a*=*b*
by *simp*

lemma *doubleton-eq-iff*: *set* {*a*,*b*} = *set* {*c*,*d*} ⟷ (*a*=*c* ∧ *b*=*d*) ∨ (*a*=*d* ∧ *b*=*c*)
by (*simp add*: *Set.doubleton-eq-iff*)

definition *vpair* :: *V* ⇒ *V* ⇒ *V*
where *vpair a b* = *set* {*set* {*a*},*set* {*a*,*b*}}

definition *vfst* :: *V* ⇒ *V*
where *vfst p* ≡ *THE x*. ∃ *y*. *p* = *vpair x y*

definition *vsnd* :: *V* ⇒ *V*
where *vsnd p* ≡ *THE y*. ∃ *x*. *p* = *vpair x y*

definition *vsplit* :: $[[V,\ V] \Rightarrow 'a,\ V] \Rightarrow 'a::\{\}$ — for pattern-matching
  where *vsplit c* $\equiv \lambda p.$ *c* (*vfst p*) (*vsnd p*)

nonterminal *Vs*
syntax (*ASCII*)
  -Tuple :: $[V,\ Vs] \Rightarrow V$ (<(-,/ -)>)
  -hpattern :: [*pttrn*, *patterns*] $\Rightarrow$ *pttrn* (<-,/ ->)
syntax
  :: $V \Rightarrow Vs$ (-)
  -Enum :: $[V,\ Vs] \Rightarrow Vs$ (-,/ -)
  -Tuple :: $[V,\ Vs] \Rightarrow V$ ($\langle$(-,/ -)$\rangle$)
  -hpattern :: [*pttrn*, *patterns*] $\Rightarrow$ *pttrn* ($\langle$-,/ -$\rangle$)
translations
  <*x*, *y*, *z*> $\rightleftharpoons$ <*x*, <*y*, *z*>>
  <*x*, *y*> $\rightleftharpoons$ *CONST vpair x y*
  <*x*, *y*, *z*> $\rightleftharpoons$ <*x*, <*y*, *z*>>
  λ<*x*,*y*,*zs*>. *b* $\rightleftharpoons$ *CONST vsplit*(λx <*y*,*zs*>. *b*)
  λ<*x*,*y*>. *b* $\rightleftharpoons$ *CONST vsplit*($\lambda x\ y.$ *b*)

lemma *vpair-def'*: *vpair a b* = *set* {*set* {*a*,*a*},*set* {*a*,*b*}}
  by (*simp add*: *vpair-def*)

lemma *vpair-iff* [*simp*]: *vpair a b* = *vpair a' b'* $\longleftrightarrow$ *a*=*a'* $\wedge$ *b*=*b'*
  unfolding *vpair-def' doubleton-eq-iff* **by** *auto*

lemmas *vpair-inject* = *vpair-iff* [*THEN iffD1*, *THEN conjE*, *elim*!]

lemma *vfst-conv* [*simp*]: *vfst* $\langle$*a*,*b*$\rangle$ = *a*
  by (*simp add*: *vfst-def*)

lemma *vsnd-conv* [*simp*]: *vsnd* $\langle$*a*,*b*$\rangle$ = *b*
  by (*simp add*: *vsnd-def*)

lemma *vsplit* [*simp*]: *vsplit c* $\langle$*a*,*b*$\rangle$ = *c a b*
  by (*simp add*: *vsplit-def*)

lemma *vpair-neq-fst*: $\langle$*a*,*b*$\rangle \neq a$
  by (*metis elts-of-set insertI1 mem-not-sym small-upair vpair-def'*)

lemma *vpair-neq-snd*: $\langle$*a*,*b*$\rangle \neq b$
  by (*metis elts-of-set insertI1 mem-not-sym small-upair subsetD subset-insertI vpair-def'*)

lemma *vpair-nonzero* [*simp*]: $\langle$*x*,*y*$\rangle \neq 0$
  by (*metis elts-0 elts-of-set empty-not-insert small-upair vpair-def*)

lemma *zero-notin-vpair*: *0* $\notin$ *elts* $\langle$*x*,*y*$\rangle$
  by (*auto simp*: *vpair-def*)

lemma *inj-on-vpair* [*simp*]: *inj-on* (λ(*x*, *y*). ⟨*x*, *y*⟩) *A*
by (*auto simp*: *inj-on-def*)

2.2 Generalized Cartesian product

definition *VSigma* :: *V* ⇒ (*V* ⇒ *V*) ⇒ *V*
where *VSigma A B* ≡ *set*(⋃*x* ∈ *elts A*. ⋃*y* ∈ *elts* (*B x*). {⟨*x*,*y*⟩})

abbreviation *vtimes* **where** *vtimes A B* ≡ *VSigma A* (λ*x*. *B*)

definition *pairs* :: *V* ⇒ (*V* ∗ *V*)*set*
where *pairs r* ≡ {(*x*,*y*). ⟨*x*,*y*⟩ ∈ *elts r*}

lemma *pairs-iff-elts*: (*x*,*y*) ∈ *pairs z* ⟷ ⟨*x*,*y*⟩ ∈ *elts z*
by (*simp add*: *pairs-def*)

lemma *VSigma-iff* [*simp*]: ⟨*a*,*b*⟩ ∈ *elts* (*VSigma A B*) ⟷ *a* ∈ *elts A* ∧ *b* ∈ *elts* (*B a*)
by (*auto simp*: *VSigma-def UNION-singleton-eq-range*)

lemma *VSigmaI* [*intro*!]: ⟦ *a* ∈ *elts A*; *b* ∈ *elts* (*B a*)⟧ ⟹ ⟨*a*,*b*⟩ ∈ *elts* (*VSigma A B*)
by *simp*

lemmas *VSigmaD1* = *VSigma-iff* [*THEN iffD1*, *THEN conjunct1*]
lemmas *VSigmaD2* = *VSigma-iff* [*THEN iffD1*, *THEN conjunct2*]

The general elimination rule

lemma *VSigmaE* [*elim*!]:
assumes *c* ∈ *elts* (*VSigma A B*)
obtains *x y* **where** *x* ∈ *elts A y* ∈ *elts* (*B x*) *c*=⟨*x*,*y*⟩
using *assms* **by** (*auto simp*: *VSigma-def split*: *if-split-asm*)

lemma *VSigmaE2* [*elim*!]:
assumes ⟨*a*,*b*⟩ ∈ *elts* (*VSigma A B*) **obtains** *a* ∈ *elts A* **and** *b* ∈ *elts* (*B a*)
using *assms* **by** *auto*

lemma *VSigma-empty1* [*simp*]: *VSigma 0 B* = *0*
by *auto*

lemma *times-iff* [*simp*]: ⟨*a*,*b*⟩ ∈ *elts* (*vtimes A B*) ⟷ *a* ∈ *elts A* ∧ *b* ∈ *elts B*
by *simp*

lemma *timesI* [*intro*!]: ⟦*a* ∈ *elts A*; *b* ∈ *elts B*⟧ ⟹ ⟨*a*,*b*⟩ ∈ *elts* (*vtimes A B*)
by *simp*

lemma *times-empty2* [*simp*]: *vtimes A 0* = *0*
using *elts-0* **by** *blast*

lemma *times-empty-iff*: *VSigma A B = 0* $\longleftrightarrow$ *A=0* $\vee$ ($\forall$ *x* $\in$ *elts A. B x = 0*)
by (*metis VSigmaE VSigmaI elts-0 empty-iff trad-foundation*)

lemma *elts-VSigma*: *elts* (*VSigma a b*) = (λ(*x*,*y*). *vpair x y*) ‘ *Sigma* (*elts a*) (λ*x*. *elts* (*b x*))
by *auto*

lemma *small-Times* [*simp*]:
assumes *small A small B*
shows *small* (*A* $\times$ *B*)
proof −
obtain *f a g b* **where** *inj-on f A inj-on g B* **and** *f*: *f* ‘ *A* = *elts a* **and** *g*: *g* ‘ *B* = *elts b*
using *assms* **by** (*auto simp*: *small-def*)
define *h* **where** *h* ≡ λ(*x*,*y*). ⟨*f x*, *g y*⟩
show *?thesis*
unfolding *small-def*
proof (*intro exI conjI*)
show *inj-on h* (*A* $\times$ *B*)
using ‹*inj-on f A*› ‹*inj-on g B*› **by** (*simp add*: *h-def inj-on-def*)
have *h* ‘ (*A* $\times$ *B*) = *elts* (*vtimes a b*)
using *f g* **by** (*fastforce simp*: *h-def image-iff split*: *prod.split*)
then show *h* ‘ (*A* $\times$ *B*) $\in$ *range elts*
by *blast*
qed
qed

2.3 Disjoint Sum

definition *vsum* :: *V* $\Rightarrow$ *V* $\Rightarrow$ *V* (**infixl** ⨄ *65*) **where**
A ⨄ *B* ≡ (*VSigma* (*set* {*0*}) (λ*x*. *A*)) $\sqcup$ (*VSigma* (*set* {*1*}) (λ*x*. *B*))

definition *Inl* :: *V*$\Rightarrow$*V* **where**
Inl a ≡ ⟨*0*,*a*⟩

definition *Inr* :: *V*$\Rightarrow$*V* **where**
Inr b ≡ ⟨*1*,*b*⟩

lemmas *sum-defs* = *vsum-def Inl-def Inr-def*

lemma *Inl-nonzero* [*simp*]:*Inl x* $\neq$ *0*
by (*metis Inl-def vpair-nonzero*)

lemma *Inr-nonzero* [*simp*]:*Inr x* $\neq$ *0*
by (*metis Inr-def vpair-nonzero*)

2.3.1 Equivalences for the injections and an elimination rule

lemma *Inl-in-sum-iff* [*iff*]: *Inl a* $\in$ *elts* (*A* ⨄ *B*) $\longleftrightarrow$ *a* $\in$ *elts A*

by (*auto simp*: *sum-defs*)

lemma *Inr-in-sum-iff* [*iff*]: *Inr b* ∈ *elts* (*A* ⨄ *B*) ⟷ *b* ∈ *elts B*
by (*auto simp*: *sum-defs*)

lemma *sumE* [*elim*!]:
assumes *u*: *u* ∈ *elts* (*A* ⨄ *B*)
obtains *x* **where** *x* ∈ *elts A u*=*Inl x* | *y* **where** *y* ∈ *elts B u*=*Inr y* **using** *u*
by (*auto simp*: *sum-defs*)

2.3.2 Injection and freeness equivalences, for rewriting

lemma *Inl-iff* [*iff*]: *Inl a*=*Inl b* ⟷ *a*=*b*
by (*simp add*: *sum-defs*)

lemma *Inr-iff* [*iff*]: *Inr a*=*Inr b* ⟷ *a*=*b*
by (*simp add*: *sum-defs*)

lemma *inj-on-Inl* [*simp*]: *inj-on Inl A*
by (*simp add*: *inj-on-def*)

lemma *inj-on-Inr* [*simp*]: *inj-on Inr A*
by (*simp add*: *inj-on-def*)

lemma *Inl-Inr-iff* [*iff*]: *Inl a*=*Inr b* ⟷ *False*
by (*simp add*: *sum-defs*)

lemma *Inr-Inl-iff* [*iff*]: *Inr b*=*Inl a* ⟷ *False*
by (*simp add*: *sum-defs*)

lemma *sum-empty* [*simp*]: *0* ⨄ *0* = *0*
by *auto*

lemma *elts-vsum*: *elts* (*a* ⨄ *b*) = *Inl* ‘ (*elts a*) ∪ *Inr* ‘ (*elts b*)
by *auto*

lemma *sum-iff*: *u* ∈ *elts* (*A* ⨄ *B*) ⟷ (∃ *x*. *x* ∈ *elts A* ∧ *u*=*Inl x*) ∨ (∃ *y*. *y* ∈ *elts B* ∧ *u*=*Inr y*)
by *blast*

lemma *sum-subset-iff*: *A*⨄ *B* ≤ *C*⨄ *D* ⟷ *A*≤*C* ∧ *B*≤*D*
by (*auto simp*: *less-eq-V-def*)

lemma *sum-equal-iff*:
fixes *A* :: *V* **shows** *A*⨄ *B* = *C*⨄ *D* ⟷ *A*=*C* ∧ *B*=*D*
by (*simp add*: *eq-iff sum-subset-iff*)

definition *is-sum* :: *V* ⇒ *bool*
where *is-sum z* = (∃ *x*. *z* = *Inl x* ∨ *z* = *Inr x*)

definition *sum-case* :: $(V \Rightarrow 'a) \Rightarrow (V \Rightarrow 'a) \Rightarrow V \Rightarrow 'a$
where
sum-case f g a ≡
THE z. $(\forall x.\ a = Inl\ x \longrightarrow z = f\ x) \wedge (\forall y.\ a = Inr\ y \longrightarrow z = g\ y) \wedge (\neg$ *is-sum* $a \longrightarrow z =$ *undefined*)

lemma *sum-case-Inl* [*simp*]: *sum-case f g* (*Inl x*) = *f x*
by (*simp add*: *sum-case-def is-sum-def*)

lemma *sum-case-Inr* [*simp*]: *sum-case f g* (*Inr y*) = *g y*
by (*simp add*: *sum-case-def is-sum-def*)

lemma *sum-case-non* [*simp*]: ¬ *is-sum a* ⟹ *sum-case f g a* = *undefined*
by (*simp add*: *sum-case-def is-sum-def*)

lemma *is-sum-cases*: $(\exists x.\ z = Inl\ x \vee z = Inr\ x) \vee \neg$ *is-sum z*
by (*auto simp*: *is-sum-def*)

lemma *sum-case-split*:
P (*sum-case f g a*) $\longleftrightarrow (\forall x.\ a = Inl\ x \longrightarrow P(f\ x)) \wedge (\forall y.\ a = Inr\ y \longrightarrow P(g\ y)) \wedge (\neg$ *is-sum a* $\longrightarrow P$ *undefined*)
by (*cases is-sum a*) (*auto simp*: *is-sum-def*)

lemma *sum-case-split-asm*:
P (*sum-case f g a*) $\longleftrightarrow \neg ((\exists x.\ a = Inl\ x \wedge \neg\ P(f\ x)) \vee (\exists y.\ a = Inr\ y \wedge \neg\ P(g\ y)) \vee (\neg$ *is-sum a* $\wedge \neg\ P$ *undefined*))
by (*auto simp*: *sum-case-split*)

2.3.3 Applications of disjoint sums and pairs: general union theorems for small sets

lemma *small-Un*:
assumes *X*: *small X* **and** *Y*: *small Y*
shows *small* (*X* ∪ *Y*)
proof −
obtain *f g* :: $'a \Rightarrow V$ **where** *f*: *inj-on f X* **and** *g*: *inj-on g Y*
by (*meson assms small-def*)
define *h* **where** *h* ≡ λ*z. if z* ∈ *X then Inl* (*f z*) *else Inr* (*g z*)
show *?thesis*
unfolding *small-def*
proof (*intro exI conjI*)
show *inj-on h* (*X* ∪ *Y*)
using *f g* **by** (*auto simp add*: *inj-on-def h-def*)
show *h* ‘ (*X* ∪ *Y*) ∈ *range elts*
by (*metis X Y image-Un replacement small-iff-range small-sup-iff*)
qed
qed

lemma *small-UN* [*simp*,*intro*]:
assumes X: *small* X **and** B: $\bigwedge x.\ x \in X \Longrightarrow$ *small* $(B\ x)$
shows *small* $(\bigcup x{\in}X.\ B\ x)$
proof −
obtain $f :: 'a \Rightarrow V$ **where** f: *inj-on f X*
by (*meson assms small-def*)
have $\exists g.$ *inj-on g* $(B\ x) \wedge g\ `\ (B\ x) \in$ *range elts* **if** $x \in X$ **for** x
using B *small-def that* **by** *auto*
then obtain $g{::}'a \Rightarrow 'b \Rightarrow V$ **where** g: $\bigwedge x.\ x \in X \Longrightarrow$ *inj-on* $(g\ x)\ (B\ x)$
by *metis*
define φ **where** $\varphi \equiv \lambda y.\ @x.\ x \in X \wedge y \in B\ x$
have φ: $\varphi\ y \in X \wedge y \in B\ (\varphi\ y)$ **if** $y \in (\bigcup x{\in}X.\ B\ x)$ **for** y
unfolding φ*-def* **by** (*metis* (*mono-tags*, *lifting*) *UN-E someI that*)
define h **where** $h \equiv \lambda y.\ \langle f\ (\varphi\ y),\ g\ (\varphi\ y)\ y\rangle$
show *?thesis*
unfolding *small-def*
proof (*intro exI conjI*)
show *inj-on h* $(\bigcup\ (B\ `\ X))$
using *f g* φ **unfolding** *h-def inj-on-def* **by** (*metis vpair-inject*)
have *small* $(h\ `\bigcup\ (B\ `\ X))$
by (*simp add*: B *X image-UN*)
then show $h\ `\bigcup\ (B\ `\ X) \in$ *range elts*
using *small-iff-range* **by** *blast*
qed
qed

lemma *small-Union* [*simp*,*intro*]:
assumes $\mathcal{A} \subseteq$ *Collect small small* $\mathcal{A}$
shows *small* $(\bigcup\ \mathcal{A})$
using *small-UN* [*of* $\mathcal{A}$ $\lambda x.\ x$] *assms* **by** (*simp add*: *subset-iff*)

2.4 Generalised function space and lambda

definition *VLambda* :: $V \Rightarrow (V \Rightarrow V) \Rightarrow V$
where *VLambda A b* ≡ *set* $((\lambda x.\ \langle x,b\ x\rangle)\ `\ elts\ A)$

definition *app* :: $[V, V] \Rightarrow V$
where *app f x* ≡ *THE y.* $\langle x,y\rangle \in$ *elts f*

lemma *beta* [*simp*]:
assumes $x \in$ *elts A*
shows *app* (*VLambda A b*) $x = b\ x$
using *assms* **by** (*auto simp*: *VLambda-def app-def*)

definition *VPi* :: $V \Rightarrow (V \Rightarrow V) \Rightarrow V$
where *VPi A B* ≡ *set* {$f \in$ *elts* (*VPow*(*VSigma A B*)). *elts A* ≤ *Domain* (*pairs f*) ∧ *single-valued* (*pairs f*)}

lemma *VPi-I*:

assumes $\bigwedge x.\ x \in elts\ A \Longrightarrow b\ x \in elts\ (B\ x)$
shows *VLambda A b* $\in$ *elts* (*VPi A B*)
proof (*clarsimp simp*: *VPi-def*, *intro conjI impI*)
show *VLambda A b* $\leq$ *VSigma A B*
by (*auto simp*: *assms VLambda-def split*: *if-split-asm*)
show *elts A* $\subseteq$ *Domain* (*pairs* (*VLambda A b*))
by (*force simp*: *VLambda-def pairs-iff-elts*)
show *single-valued* (*pairs* (*VLambda A b*))
by (*auto simp*: *VLambda-def single-valued-def pairs-iff-elts*)
show *small* {*f*. *f* $\leq$ *VSigma A B* $\wedge$ *elts A* $\subseteq$ *Domain* (*pairs f*) $\wedge$ *single-valued* (*pairs f*)}
by (*metis* (*mono-tags*, *lifting*) *down VPow-iff mem-Collect-eq subsetI*)
qed

lemma *apply-pair*:
assumes *f*: *f* $\in$ *elts* (*VPi A B*) **and** *x*: *x* $\in$ *elts A*
shows $\langle x,\ app\ f\ x\rangle \in elts\ f$
proof −
have *x* $\in$ *Domain* (*pairs f*)
by (*metis* (*no-types*, *lifting*) *VPi-def assms elts-of-set empty-iff mem-Collect-eq subsetD*)
then obtain *y* **where** *y*: $\langle x,y\rangle \in elts\ f$
using *pairs-iff-elts* **by** *auto*
show *?thesis*
unfolding *app-def*
proof (*rule theI*)
show $\langle x,\ y\rangle \in elts\ f$
by (*rule y*)
show *z* = *y* **if** $\langle x,\ z\rangle \in elts\ f$ **for** *z*
using *f* **unfolding** *VPi-def*
by (*metis* (*mono-tags*, *lifting*) *that elts-of-set empty-iff mem-Collect-eq pairs-iff-elts single-valued-def y*)
qed
qed

lemma *VPi-D*:
assumes *f*: *f* $\in$ *elts* (*VPi A B*) **and** *x*: *x* $\in$ *elts A*
shows *app f x* $\in$ *elts* (*B x*)
proof −
have *f* $\leq$ *VSigma A B*
by (*metis* (*no-types*, *lifting*) *VPi-def elts-of-set empty-iff f VPow-iff mem-Collect-eq*)
then show *?thesis*
using *apply-pair* [*OF assms*] **by** *blast*
qed

lemma *VPi-memberD*:
assumes *f*: *f* $\in$ *elts* (*VPi A B*) **and** *p*: *p* $\in$ *elts f*
obtains *x* **where** *x* $\in$ *elts A* $p = \langle x,\ app\ f\ x\rangle$
proof −

have $f \leq$ *VSigma A B*
by (*metis* (*no-types*, *lifting*) *VPi-def elts-of-set empty-iff f VPow-iff mem-Collect-eq*)
then obtain *x y* **where** $p = \langle x,y\rangle$ $x \in$ *elts A*
using *p* **by** *blast*
then have $y =$ *app f x*
by (*metis* (*no-types*, *lifting*) *VPi-def apply-pair elts-of-set equals0D f mem-Collect-eq p pairs-iff-elts single-valuedD*)
then show *thesis*
using ‹$p = \langle x,\ y\rangle$› ‹$x \in$ *elts A*› *that* **by** *blast*
qed

lemma *fun-ext*:
assumes $f \in$ *elts* (*VPi A B*) $g \in$ *elts* (*VPi A B*) $\bigwedge x.\ x \in$ *elts A* $\Longrightarrow$ *app f x* = *app g x*
shows $f = g$
by (*metis VPi-memberD V-equalityI apply-pair assms*)

lemma *eta*[*simp*]:
assumes $f \in$ *elts* (*VPi A B*)
shows *VLambda A* ((*app*)*f*) = *f*
proof (*rule fun-ext* [*OF - assms*])
show *VLambda A* (*app f*) $\in$ *elts* (*VPi A B*)
using *VPi-D VPi-I assms* **by** *auto*
qed *auto*

lemma *fst-pairs-VLambda*: *fst* ‘ *pairs* (*VLambda A f*) = *elts A*
by (*force simp*: *VLambda-def pairs-def*)

lemma *snd-pairs-VLambda*: *snd* ‘ *pairs* (*VLambda A f*) = *f* ‘ *elts A*
by (*force simp*: *VLambda-def pairs-def*)

lemma *VLambda-eq-D1*: *VLambda A f* = *VLambda B g* $\Longrightarrow$ *A* = *B*
by (*metis ZFC-in-HOL.ext fst-pairs-VLambda*)

lemma *VLambda-eq-D2*: $[\![$*VLambda A f* = *VLambda A g*; $x \in$ *elts A*$]\!] \Longrightarrow$ *f x* = *g x*
by (*metis beta*)

2.5 Transitive closure of a set

definition *TC* :: *V*⇒*V*
where *TC* ≡ *transrec* (λ*f x*. *x* ⊔ $\bigsqcup$ (*f* ‘ *elts x*))

lemma *TC*: *TC a* = *a* ⊔ $\bigsqcup$ (*TC* ‘ *elts a*)
by (*metis* (*no-types*, *lifting*) *SUP-cong TC-def restrict-apply*′ *transrec*)

lemma *TC-0* [*simp*]: *TC 0* = *0*
by (*metis TC ZFC-in-HOL.Sup-empty elts-0 image-is-empty sup-V-0-left*)

lemma *arg-subset-TC*: $a \leq TC\ a$
by (*metis* (*no-types*) *TC sup-ge1*)

lemma *Transset-TC*: *Transset*(*TC a*)
proof (*induction a rule*: *eps-induct*)
case (*step x*)
have *1*: $v \in elts\ (TC\ x)$ **if** $v \in elts\ u\ u \in elts\ x$ **for** *u v*
using *that* **unfolding** *TC* [*of x*]
using *arg-subset-TC* **by** *fastforce*
have *2*: $v \in elts\ (TC\ x)$ **if** $v \in elts\ u\ \exists x{\in}elts\ x.\ u \in elts\ (TC\ x)$ **for** *u v*
using *that step* **unfolding** *TC* [*of x*] *Transset-def* **by** *auto*
show *?case*
unfolding *Transset-def*
by (*subst TC*) (*force intro*: *1 2*)
qed

lemma *TC-least*: $[\![Transset\ x;\ \ a{\leq}x]\!] \Longrightarrow TC\ a \leq x$
proof (*induction a rule*: *eps-induct*)
case (*step y*)
show *?case*
proof (*cases y=0*)
case *True*
then show *?thesis*
by *auto*
next
case *False*
have $\bigsqcup\ (TC\ `\ elts\ y) \leq x$
proof (*rule cSup-least*)
show $TC\ `\ elts\ y \neq \{\}$
using *False* **by** *auto*
show $z \leq x$ **if** $z \in TC\ `\ elts\ y$ **for** *z*
using *that* **by** (*metis Transset-def image-iff step.IH step.prems vsubsetD*)
qed
then show *?thesis*
by (*simp add*: *step TC* [*of y*])
qed
qed

definition *less-TC* (**infix** $\sqsubset$ *50*)
where $x \sqsubset y \equiv x \in elts\ (TC\ y)$

definition *le-TC* (**infix** $\sqsubseteq$ *50*)
where $x \sqsubseteq y \equiv x \sqsubset y \vee x{=}y$

lemma *less-TC-imp-not-le*: $x \sqsubset a \Longrightarrow \neg\ a \leq x$
proof (*induction a arbitrary*: *x rule*: *eps-induct*)
case (*step a*)
then show *?case*

unfolding *TC*[*of a*] *less-TC-def*
using *Transset-TC Transset-def* **by** *force*
qed

lemma *non-TC-less-0* [*iff*]: $\neg\ (x \sqsubset 0)$
using *less-TC-imp-not-le* **by** *blast*

lemma *less-TC-iff*: $x \sqsubset y \longleftrightarrow (\exists\, z \in \mathit{elts}\ y.\ x \sqsubseteq z)$
by (*auto simp*: *less-TC-def le-TC-def TC* [*of y*])

lemma *nonzero-less-TC*: $x \neq 0 \Longrightarrow 0 \sqsubset x$
by (*metis eps-induct le-TC-def less-TC-iff trad-foundation*)

lemma *less-irrefl-TC* [*simp*]: $\neg\ x \sqsubset x$
using *less-TC-imp-not-le* **by** *blast*

lemma *less-asym-TC*: $[\![x \sqsubset y;\ y \sqsubset x]\!] \Longrightarrow \mathit{False}$
by (*metis TC-least Transset-TC Transset-def antisym-conv less-TC-def less-TC-imp-not-le order-refl*)

lemma *le-antisym-TC*: $[\![x \sqsubset y;\ y \sqsubset x]\!] \Longrightarrow x = y$
using *less-asym-TC* **by** *blast*

lemma *less-imp-le-TC* [*iff*]: $x \sqsubset y \Longrightarrow x \sqsubseteq y$
by (*simp add*: *le-TC-def*)

lemma *le-TC-refl* [*iff*]: $x \sqsubseteq x$
by (*simp add*: *le-TC-def*)

lemma *less-TC-trans* [*trans*]: $[\![x \sqsubset y;\ y \sqsubset z]\!] \Longrightarrow x \sqsubset z$
by (*meson TC-least Transset-TC Transset-def less-TC-def less-eq-V-def subsetD*)

lemma *less-le-TC-trans* [*trans*]: $[\![x \sqsubset y;\ y \sqsubseteq z]\!] \Longrightarrow x \sqsubset z$
using *le-TC-def less-TC-trans* **by** *blast*

lemma *le-less-TC-trans* [*trans*]: $[\![x \sqsubseteq y;\ y \sqsubset z]\!] \Longrightarrow x \sqsubset z$
using *le-TC-def less-TC-trans* **by** *blast*

lemma *le-TC-trans* [*trans*]: $[\![x \sqsubseteq y;\ y \sqsubseteq z]\!] \Longrightarrow x \sqsubseteq z$
using *le-TC-def le-less-TC-trans* **by** *blast*

lemma *TC-sup-distrib*: $TC\ (x \sqcup y) = TC\ x \sqcup TC\ y$
by (*simp add*: *Sup-Un-distrib TC* [*of x* ⊔ *y*] *TC* [*of x*] *TC* [*of y*] *image-Un sup.assoc sup-left-commute*)

lemma *TC-Sup-distrib*:
assumes *small X* **shows** $TC\ (\bigsqcup X) = \bigsqcup (TC\ `\ X)$
proof −
have $\bigsqcup\ X \sqcup \bigsqcup\ (\bigcup x{\in}X.\ TC\ `\ \mathit{elts}\ x) \leq \bigsqcup\ (TC\ `\ X)$

using *assms*
apply (*auto simp*: *Sup-le-iff*)
using *arg-subset-TC* **apply** *blast*
by (*metis TC-least Transset-TC Transset-def arg-subset-TC vsubsetD*)
moreover have $\bigsqcup\ (TC\ `\ X) \leq \bigsqcup\ X \sqcup \bigsqcup\ (\bigcup x{\in}X.\ TC\ `\ elts\ x)$
proof (*clarsimp simp add*: *Sup-le-iff assms*)
show $\exists\, x{\in}X.\ y \in elts\ x$
if $x \in X\ y \in elts\ (TC\ x)\ \forall\, x{\in}X.\ \forall\, u{\in}elts\ x.\ y \notin elts\ (TC\ u)$ **for** $x\ y$
using *that* **by** (*auto simp*: *TC* [*of x*])
qed
ultimately show *?thesis*
using *Sup-Un-distrib TC* [*of* $\bigsqcup X$] *image-Union assms*
by (*simp add*: *image-Union inf-sup-aci*(5) *sup.absorb-iff2*)
qed

lemma *TC'*: $TC\ x = x \sqcup TC\ (\bigsqcup\ (elts\ x))$
by (*simp add*: *TC* [*of x*] *TC-Sup-distrib*)

lemma *TC-eq-0-iff* [*simp*]: $TC\ x = 0 \longleftrightarrow x{=}0$
using *arg-subset-TC* **by** *fastforce*

A distinctive induction principle

lemma *TC-induct-down-lemma*:
assumes *ab*: $a \sqsubset b$ **and** *base*: $b \leq d$
and *step*: $\bigwedge y\ z.\ [\![y \sqsubset b;\ y \in elts\ d;\ z \in elts\ y]\!] \Longrightarrow z \in elts\ d$
shows $a \in elts\ d$
proof −
have *Transset* $(TC\ b \sqcap d)$
using *Transset-TC*
unfolding *Transset-def*
by (*metis inf.bounded-iff less-TC-def less-eq-V-def local.step subsetI vsubsetD*)
moreover have $b \leq TC\ b \sqcap d$
by (*simp add*: *arg-subset-TC base*)
ultimately show *?thesis*
using *TC-least* [*THEN vsubsetD*] *ab* **unfolding** *less-TC-def*
by (*meson TC-least le-inf-iff vsubsetD*)
qed

lemma *TC-induct-down* [*consumes 1*, *case-names base step small*]:
assumes $a \sqsubset b$
and $\bigwedge y.\ y \in elts\ b \Longrightarrow P\ y$
and $\bigwedge y\ z.\ [\![y \sqsubset b;\ P\ y;\ z \in elts\ y]\!] \Longrightarrow P\ z$
and *small* (*Collect P*)
shows $P\ a$
using *TC-induct-down-lemma* [*of a b set* (*Collect P*)] *assms*
by (*metis elts-of-set mem-Collect-eq vsubsetI*)

2.6 Rank of a set

definition *rank* :: $V \Rightarrow V$
  where *rank a* $\equiv$ *transrec* ($\lambda f\ x.$ *set* ($\bigcup y \in elts\ x.$ *elts* (*succ*(*f y*)))) *a*

lemma *rank*: *rank a* = *set*($\bigcup y \in$ *elts a.* *elts* (*succ*(*rank y*)))
  by (*subst rank-def* [*THEN def-transrec*], *simp*)

lemma *rank-Sup*: *rank a* = $\bigsqcup$((λ*y.* *succ*(*rank y*)) ‘ *elts a*)
  by (*metis elts-Sup image-image rank replacement set-of-elts small-elts*)

lemma *Ord-rank* [*simp*]: *Ord*(*rank a*)
proof (*induction a rule*: *eps-induct*)
  case (*step x*)
  then show *?case*
    unfolding *rank-Sup* [*of x*]
    by (*metis* (*mono-tags*, *lifting*) *Ord-Sup Ord-succ imageE*)
qed

lemma *rank-of-Ord*: *Ord i* $\Longrightarrow$ *rank i* = *i*
  apply (*induction rule*: *Ord-induct*)
  by (*metis* (*no-types*, *lifting*) *Ord-equality SUP-cong rank-Sup*)

lemma *Ord-iff-rank*: *Ord x* $\longleftrightarrow$ *rank x* = *x*
  using *Ord-rank* [*of x*] *rank-of-Ord* **by** *fastforce*

lemma *rank-lt*: *a* $\in$ *elts b* $\Longrightarrow$ *rank a* < *rank b*
  apply (*subst rank* [*of b*])
  by (*metis* (*no-types*, *lifting*) *Ord-mem-iff-lt Ord-rank small-UN UN-iff elts-of-set elts-succ insert-iff rank small-elts*)

lemma *rank-0* [*simp*]: *rank 0* = *0*
  unfolding *rank-def*
  using *transrec* **by** *fastforce*

lemma *rank-succ* [*simp*]: *rank*(*succ x*) = *succ*(*rank x*)
proof (*rule order-antisym*)
  show *rank* (*succ x*) $\leq$ *succ* (*rank x*)
    apply (*subst rank* [*of succ x*])
     apply (*metis* (*no-types*, *lifting*) *Sup-insert elts-of-set elts-succ equals0D image-insert rank small-sup-iff subset-insertI sup.orderE vsubsetI*)
    done
  show *succ* (*rank x*) $\leq$ *rank* (*succ x*)
     by (*metis* (*mono-tags*, *lifting*) *ZFC-in-HOL.Sup-upper elts-succ image-insert insertI1 rank-Sup replacement small-elts*)
qed

lemma *rank-mono*: *a* $\leq$ *b* $\Longrightarrow$ *rank a* $\leq$ *rank b*
  apply (*rule vsubsetI*)
  using *rank* [*of a*] *rank* [*of b*] *small-UN* **by** *auto*

lemma *VsetI*: *rank b* $\sqsubset$ *i* $\Longrightarrow$ *b* $\in$ *elts* (*Vset i*)
proof (*induction i arbitrary*: *b rule*: *eps-induct*)
case (*step x*)
then consider *rank b* $\in$ *elts x* | (∃ *y*∈*elts x*. *rank b* $\in$ *elts* (*TC y*))
using *le-TC-def less-TC-def less-TC-iff* **by** *fastforce*
then have ∃ *y*∈*elts x*. *b* $\leq$ *Vset y*
proof *cases*
case *1*
then have *b* $\leq$ *Vset* (*rank b*)
unfolding *less-eq-V-def subset-iff*
by (*meson Ord-mem-iff-lt Ord-rank le-TC-refl less-TC-iff rank-lt step.IH*)
then show *?thesis*
using *1* **by** *blast*
next
case *2*
then show *?thesis*
using *step.IH*
unfolding *less-eq-V-def subset-iff less-TC-def*
by (*meson Ord-mem-iff-lt Ord-rank Transset-TC Transset-def rank-lt vsubsetD*)
qed
then show *?case*
by (*simp add*: *Vset* [*of x*])
qed

lemma *Ord-VsetI*: ⟦*Ord i*; *rank b* < *i*⟧ $\Longrightarrow$ *b* $\in$ *elts* (*Vset i*)
by (*meson Ord-mem-iff-lt Ord-rank VsetI arg-subset-TC less-TC-def vsubsetD*)

lemma *arg-le-Vset-rank*: *a* $\leq$ *Vset*(*rank a*)
by (*simp add*: *Ord-VsetI rank-lt vsubsetI*)

lemma *two-in-Vset*:
obtains α **where** *x* $\in$ *elts* (*Vset* α) *y* $\in$ *elts* (*Vset* α)
by (*metis Ord-rank Ord-VsetI elts-of-set insert-iff rank-lt small-elts small-insert-iff*)

lemma *rank-eq-0-iff* [*simp*]: *rank x* = *0* $\longleftrightarrow$ *x*=*0*
using *arg-le-Vset-rank* **by** *fastforce*

lemma *small-ranks-imp-small*:
assumes *small* (*rank* ‘ *A*) **shows** *small A*
proof −
define *i* **where** *i* ≡ *set* (⋃(*elts* ‘ (*rank* ‘ *A*)))
have *Ord i*
unfolding *i-def* **using** *Ord-Union Ord-rank assms imageE* **by** *blast*
have ∗: *Vset* (*rank x*) $\leq$ (*Vset i*) **if** *x* $\in$ *A* **for** *x*
unfolding *i-def* **by** (*metis Ord-rank Sup-V-def ZFC-in-HOL.Sup-upper Vfrom-mono assms imageI le-less that*)
have *A* $\subseteq$ *elts* (*VPow* (*Vset i*))
by (*meson* ∗ *VPow-iff arg-le-Vset-rank order.trans subsetI*)

then show *?thesis*
using *down* **by** *blast*
qed

lemma *rank-Union*: *rank*($\bigsqcup$ *A*) = $\bigsqcup$ (*rank* ' *A*)
proof (*rule order-antisym*)
have *elts* (*SUP y*∈*elts* ($\bigsqcup$ *A*). *succ* (*rank y*)) ⊆ *elts* ($\bigsqcup$ (*rank* ' *A*))
apply *auto*
using *Ord-mem-iff-lt Ord-rank rank-lt* **apply** *blast*
by (*meson less-le-not-le rank-lt vsubsetD*)
then show *rank* ($\bigsqcup$ *A*) ≤ $\bigsqcup$ (*rank* ' *A*)
by (*metis less-eq-V-def rank-Sup*)
show $\bigsqcup$ (*rank* ' *A*) ≤ *rank* ($\bigsqcup$ *A*)
proof (*cases small A*)
case *True*
then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *ZFC-in-HOL.Sup-least ZFC-in-HOL.Sup-upper image-iff rank-mono*)
next
case *False*
then have ¬ *small* (*rank* ' *A*)
using *small-ranks-imp-small* **by** *blast*
then show *?thesis*
by *blast*
qed
qed

lemma *small-bounded-rank*: *small* {*x*. *rank x* ∈ *elts a*}
proof −
have {*x*. *rank x* ∈ *elts a*} ⊆ {*x*. *rank x* ⊏ *a*}
using *less-TC-iff* **by** *auto*
also have ... ⊆ *elts* (*Vset a*)
using *VsetI* **by** *blast*
finally show *?thesis*
using *down* **by** *simp*
qed

lemma *small-bounded-rank-le*: *small* {*x*. *rank x* ≤ *a*}
using *small-bounded-rank* [*of VPow a*] *VPow-iff* [*of - a*] **by** *simp*

lemma *TC-rank-lt*: *a* ⊏ *b* ⟹ *rank a* < *rank b*
proof (*induction rule*: *TC-induct-down*)
case (*base y*)
then show *?case*
by (*simp add*: *rank-lt*)
next
case (*step y z*)
then show *?case*
using *less-trans rank-lt* **by** *blast*

next
case *small*
show *?case*
apply (*rule smaller-than-small* [*OF small-bounded-rank-le* [*of rank b*]])
by (*simp add*: *Collect-mono less-V-def*)
qed

lemma *TC-rank-mem*: $x \sqsubset y \Longrightarrow rank\ x \in elts\ (rank\ y)$
by (*simp add*: *Ord-mem-iff-lt TC-rank-lt*)

lemma *wf-TC-less*: *wf* $\{(x,y).\ x \sqsubset y\}$
proof (*rule wf-subset* [*OF wf-inv-image* [*OF foundation*, *of rank*]])
show $\{(x,\ y).\ x \sqsubset y\} \subseteq$ *inv-image* $\{(x,\ y).\ x \in elts\ y\}$ *rank*
by (*auto simp*: *TC-rank-mem inv-image-def*)
qed

lemma *less-TC-minimal*:
assumes *P a*
obtains *x* **where** *P x* $x \sqsubseteq a$ $\bigwedge y.\ y \sqsubset x \Longrightarrow \neg\ P\ y$
using *wfE-min'* [*OF wf-TC-less*, *of* $\{x.\ P\ x \wedge x \sqsubseteq a\}$]
by *simp* (*metis le-TC-def less-le-TC-trans assms*)

lemma *Vfrom-rank-eq*: *Vfrom A* (*rank*(*x*)) = *Vfrom A x*
proof (*rule order-antisym*)
show *Vfrom A* (*rank x*) ≤ *Vfrom A x*
proof (*induction x rule*: *eps-induct*)
case (*step x*)
have (*SUP j*∈*elts* (*rank x*). *VPow* (*Vfrom A j*)) ≤ (*SUP j*∈*elts x*. *VPow* (*Vfrom A j*))
apply (*rule Sup-least*, *clarify*)
apply (*simp add*: *rank* [*of x*])
using *step.IH*
by (*metis Ord-rank OrdmemD Vfrom-mono2 dual-order.trans inf-sup-aci*(*5*) *less-V-def sup.orderE*)
then show *?case*
by (*simp add*: *Vfrom* [*of - x*] *Vfrom* [*of - rank*(*x*)] *sup.coboundedI2*)
qed
show *Vfrom A x* ≤ *Vfrom A* (*rank x*)
proof (*induction x rule*: *eps-induct*)
case (*step x*)
have (*SUP j*∈*elts x*. *VPow* (*Vfrom A j*)) ≤ (*SUP j*∈*elts* (*rank x*). *VPow* (*Vfrom A j*))
using *step.IH TC-rank-mem less-TC-iff* **by** *force*
then show *?case*
by (*simp add*: *Vfrom* [*of - x*] *Vfrom* [*of - rank*(*x*)] *sup.coboundedI2*)
qed
qed

lemma *Vfrom-succ*: *Vfrom A* (*succ*(*i*)) = *A* ⊔ *VPow*(*Vfrom A i*)

by (*metis Ord-rank Vfrom-rank-eq Vfrom-succ-Ord rank-succ*)

lemma *Vset-succ-TC*:
assumes $x \in elts\ (Vset\ (ZFC\text{-}in\text{-}HOL.succ\ k))\ u \sqsubset x$
shows $u \in elts\ (Vset\ k)$
using *assms*
apply (*simp add*: *Vfrom-succ*)
using *TC-least Transset-Vfrom less-TC-def* **by** *auto*

2.7 Cardinal Numbers

We extend the membership relation to a wellordering

definition $VWO :: (V \times V)\ set$
where $VWO \equiv @r.\ \{(x,y).\ x \in elts\ y\} \subseteq r \wedge Well\text{-}order\ r \wedge Field\ r = UNIV$

lemma *VWO*: $\{(x,y).\ x \in elts\ y\} \subseteq VWO \wedge Well\text{-}order\ VWO \wedge Field\ VWO = UNIV$
unfolding *VWO-def*
by (*metis* (*mono-tags*, *lifting*) *VWO-def foundation someI-ex total-well-order-extension*)

lemma *wf-VWO*: $wf(VWO - Id)$
using *VWO well-order-on-def* **by** *blast*

lemma *wf-Ord-less*: $wf\ \{(x,\ y).\ Ord\ y \wedge x < y\}$
by (*metis* (*no-types*, *lifting*) *Ord-mem-iff-lt eps-induct wfPUNIVI wfP-def*)

lemma *refl-VWO*: *refl VWO*
using *VWO order-on-defs* **by** *fastforce*

lemma *trans-VWO*: *trans VWO*
using *VWO* **by** (*simp add*: *VWO wo-rel.TRANS wo-rel-def*)

lemma *antisym-VWO*: *antisym VWO*
using *VWO* **by** (*simp add*: *VWO wo-rel.ANTISYM wo-rel-def*)

lemma *total-VWO*: *total VWO*
using *VWO* **by** (*metis wo-rel.TOTAL wo-rel.intro*)

lemma *total-VWOId*: *total* $(VWO - Id)$
by (*simp add*: *total-VWO*)

lemma *Linear-order-VWO*: *Linear-order VWO*
using *VWO well-order-on-def* **by** *blast*

lemma *wo-rel-VWO*: *wo-rel VWO*
using *VWO wo-rel-def* **by** *blast*

2.7.1 Transitive Closure and VWO

lemma *mem-imp-VWO*: $x \in elts\ y \Longrightarrow (x,y) \in VWO$
using *VWO* **by** *blast*

lemma *less-TC-imp-VWO*: $x \sqsubset y \Longrightarrow (x,y) \in VWO$
unfolding *less-TC-def*
proof (*induction y arbitrary*: *x rule*: *eps-induct*)
case (*step* y' *u*)
then consider $u \in elts\ y' \mid v$ **where** $v \in elts\ y'\ u \in elts\ (TC\ v)$
by (*auto simp*: *TC* [*of* y'])
then show *?case*
proof *cases*
case *2*
then show *?thesis*
by (*meson mem-imp-VWO step.IH transD trans-VWO*)
qed (*use mem-imp-VWO* **in** *blast*)
qed

lemma *le-TC-imp-VWO*: $x \sqsubseteq y \Longrightarrow (x,y) \in VWO$
apply (*auto simp*: *le-TC-def less-TC-imp-VWO*)
by (*metis Diff-iff Linear-order-VWO Linear-order-in-diff-Id UNIV-I VWO*)

lemma *le-TC-0-iff* [*simp*]: $x \sqsubseteq 0 \longleftrightarrow x = 0$
by (*simp add*: *le-TC-def*)

lemma *less-TC-succ*: $x \sqsubset succ\ \beta \longleftrightarrow x \sqsubset \beta \vee x = \beta$
by (*metis elts-succ insert-iff le-TC-def less-TC-iff*)

lemma *le-TC-succ*: $x \sqsubseteq succ\ \beta \longleftrightarrow x \sqsubseteq \beta \vee x = succ\ \beta$
by (*simp add*: *le-TC-def less-TC-succ*)

lemma *Transset-TC-eq* [*simp*]: $Transset\ x \Longrightarrow TC\ x = x$
by (*simp add*: *TC-least arg-subset-TC eq-iff*)

lemma *Ord-TC-less-iff*: $[\![Ord\ \alpha;\ Ord\ \beta]\!] \Longrightarrow \beta \sqsubset \alpha \longleftrightarrow \beta < \alpha$
by (*metis Ord-def Ord-mem-iff-lt Transset-TC-eq less-TC-def*)

lemma *Ord-mem-iff-less-TC*: $Ord\ l \Longrightarrow k \in elts\ l \longleftrightarrow k \sqsubset l$
by (*simp add*: *Ord-def less-TC-def*)

lemma *le-TC-Ord*: $[\![\beta \sqsubseteq \alpha;\ Ord\ \alpha]\!] \Longrightarrow Ord\ \beta$
by (*metis Ord-def Ord-in-Ord Transset-TC-eq le-TC-def less-TC-def*)

lemma *Ord-less-TC-mem*:
assumes $Ord\ \alpha\ \beta \sqsubset \alpha$ **shows** $\beta \in elts\ \alpha$
using *Ord-def assms less-TC-def* **by** *auto*

lemma *VWO-TC-le*: $[\![Ord\ \alpha;\ Ord\ \beta;\ (\beta,\ \alpha) \in VWO]\!] \Longrightarrow \beta \sqsubseteq \alpha$
proof (*induct* α *arbitrary*: β *rule*: *Ord-induct*)

case (*step* α)
then show *?case*
by (*metis Diff-iff Linear-order-VWO Linear-order-in-diff-Id Ord-TC-less-iff Ord-linear2 UNIV-I VWO le-TC-def le-less less-TC-imp-VWO pair-in-Id-conv*)
qed

lemma *VWO-iff-Ord-le* [*simp*]: $[\![Ord\ \alpha;\ Ord\ \beta]\!] \Longrightarrow (\beta,\ \alpha) \in VWO \longleftrightarrow \beta \leq \alpha$
by (*metis VWO-TC-le Ord-TC-less-iff le-TC-def le-TC-imp-VWO le-less*)

lemma *zero-TC-le* [*iff*]: $0 \sqsubseteq y$
using *le-TC-def nonzero-less-TC* **by** *auto*

lemma *succ-le-TC-iff*: $Ord\ j \Longrightarrow succ\ i \sqsubseteq j \longleftrightarrow i \sqsubset j$
by (*metis Ord-in-Ord Ord-linear Ord-mem-iff-less-TC Ord-succ le-TC-def less-TC-succ less-asym-TC*)

lemma *VWO-0-iff* [*simp*]: $(x,0) \in VWO \longleftrightarrow x=0$
proof
show $x = 0$ **if** $(x,\ 0) \in VWO$
using *zero-TC-le* [*of x*] *le-TC-imp-VWO that*
by (*metis DiffI Linear-order-VWO Linear-order-in-diff-Id UNIV-I VWO pair-in-Id-conv*)
qed *auto*

lemma *VWO-antisym*:
assumes $(x,y) \in VWO$ $(y,x) \in VWO$ **shows** $x=y$
by (*metis Diff-iff IdD Linear-order-VWO Linear-order-in-diff-Id UNIV-I VWO assms*)

2.7.2 Relation VWF

definition *VWF* **where** $VWF \equiv VWO - Id$

lemma *wf-VWF* [*iff*]: *wf VWF*
by (*simp add*: *VWF-def wf-VWO*)

lemma *trans-VWF* [*iff*]: *trans VWF*
by (*simp add*: *VWF-def antisym-VWO trans-VWO trans-diff-Id*)

lemma *asym-VWF* [*iff*]: *asym VWF*
by (*metis VWF-def asym.intros irrefl-diff-Id wf-VWF wf-not-sym*)

lemma *total-VWF* [*iff*]: *total VWF*
using *VWF-def total-VWOId* **by** *auto*

lemma *total-on-VWF* [*iff*]: *total-on A VWF*
by (*meson UNIV-I total-VWF total-on-def*)

lemma *VWF-asym*:
assumes $(x,y) \in VWF$ $(y,x) \in VWF$ **shows** *False*

using *VWF-def assms wf-VWO wf-not-sym* **by** *fastforce*

lemma *VWF-non-refl* [*iff*]: $(x,x) \notin VWF$
by *simp*

lemma *VWF-iff-Ord-less* [*simp*]: $[\![Ord\ \alpha;\ Ord\ \beta]\!] \Longrightarrow (\alpha,\beta) \in VWF \longleftrightarrow \alpha < \beta$
by (*simp add*: *VWF-def less-V-def*)

lemma *mem-imp-VWF*: $x \in elts\ y \Longrightarrow (x,y) \in VWF$
using *VWF-def mem-imp-VWO* **by** *fastforce*

2.8 Order types

definition *ordermap* :: $'a\ set \Rightarrow ('a \times 'a)\ set \Rightarrow 'a \Rightarrow V$
where $ordermap\ A\ r \equiv wfrec\ r\ (\lambda f\ x.\ set\ (f\ `\ \{y \in A.\ (y,x) \in r\}))$

definition *ordertype* :: $'a\ set \Rightarrow ('a \times 'a)\ set \Rightarrow V$
where $ordertype\ A\ r \equiv set\ (ordermap\ A\ r\ `\ A)$

lemma *ordermap-type*:
$small\ A \Longrightarrow ordermap\ A\ r \in A \to elts\ (ordertype\ A\ r)$
by (*simp add*: *ordertype-def*)

lemma *ordermap-in-ordertype* [*intro*]: $[\![a \in A;\ small\ A]\!] \Longrightarrow ordermap\ A\ r\ a \in elts\ (ordertype\ A\ r)$
by (*simp add*: *ordertype-def*)

lemma *ordermap*: $wf\ r \Longrightarrow ordermap\ A\ r\ a = set\ (ordermap\ A\ r\ `\ \{y \in A.\ (y,a) \in r\})$
unfolding *ordermap-def*
by (*auto simp*: *wfrec-fixpoint adm-wf-def*)

lemma *wf-Ord-ordermap* [*iff*]: **assumes** *wf r trans r* **shows** $Ord\ (ordermap\ A\ r\ x)$
using ‹*wf r*›
proof (*induction x rule*: *wf-induct-rule*)
case (*less u*)
have *Transset* (*set* (*ordermap A r* ‘ {*y* ∈ *A*. (*y*, *u*) ∈ *r*}))
proof (*clarsimp simp add*: *Transset-def*)
show *x* ∈ *ordermap A r* ‘ {*y* ∈ *A*. (*y*, *u*) ∈ *r*}
if *small* (*ordermap A r* ‘ {*y* ∈ *A*. (*y*, *u*) ∈ *r*})
and *x*: *x* ∈ *elts* (*ordermap A r y*) **and** *y* ∈ *A* (*y*, *u*) ∈ *r* **for** *x y*
proof −
have *ordermap A r y* = *ZFC-in-HOL.set* (*ordermap A r* ‘ {*a* ∈ *A*. (*a*, *y*) ∈ *r*})
using *ordermap assms*(*1*) **by** *force*
then have *x* ∈ *ordermap A r* ‘ {*z* ∈ *A*. (*z*, *y*) ∈ *r*}
by (*metis* (*no-types*, *lifting*) *elts-of-set empty-iff x*)
then have ∃ *v*. *v* ∈ *A* ∧ (*v*, *u*) ∈ *r* ∧ *x* = *ordermap A r v*

```
      using that transD [OF ‹trans r›] by blast
    then show ?thesis
      by blast
  qed
 qed
 moreover have Ord x
  if x ∈ elts (set (ordermap A r ‘ {y ∈ A. (y, u) ∈ r})) for x
  using that less by (auto simp: split: if-split-asm)
 ultimately show ?case
  by (metis (full-types) Ord-def ordermap assms(1))
qed

lemma wf-Ord-ordertype: assumes wf r trans r shows Ord(ordertype A r)
proof −
 have y ≤ set (ordermap A r ‘ A)
  if y = ordermap A r x x ∈ A small (ordermap A r ‘ A) for x y
  using that by (auto simp: less-eq-V-def ordermap [OF ‹wf r›, of A x])
 moreover have z ≤ y if y ∈ ordermap A r ‘ A z ∈ elts y for y z
  by (metis wf-Ord-ordermap OrdmemD assms imageE order.strict-implies-order
that)
 ultimately show ?thesis
  unfolding ordertype-def Ord-def Transset-def by simp
qed

lemma Ord-ordertype [simp]: Ord(ordertype A VWF)
 using wf-Ord-ordertype by blast

lemma Ord-ordermap [simp]: Ord (ordermap A VWF x)
 by blast

lemma ordertype-singleton [simp]:
 assumes wf r
 shows ordertype {x} r = 1
proof −
 have †: {y. y = x ∧ (y, x) ∈ r} = {}
  using assms by auto
 show ?thesis
  by (auto simp add: ordertype-def assms † ordermap [where a=x])
qed
```

2.8.1 *ordermap* preserves the orderings in both directions

```
lemma ordermap-mono:
 assumes wx: (w, x) ∈ r and wf r w ∈ A small A
  shows ordermap A r w ∈ elts (ordermap A r x)
proof −
 have small {a ∈ A. (a, x) ∈ r} ∧ w ∈ A ∧ (w, x) ∈ r
  by (simp add: assms)
 then show ?thesis
```

using *assms ordermap* [*of r A*]
by (*metis* (*no-types*, *lifting*) *elts-of-set image-eqI mem-Collect-eq replacement*)
qed

lemma *converse-ordermap-mono*:
assumes *ordermap A r y* ∈ *elts* (*ordermap A r x*) *wf r total-on A r x* ∈ *A y* ∈ *A small A*
shows (*y*, *x*) ∈ *r*
proof (*cases x* = *y*)
case *True*
then show *?thesis*
using *assms*(*1*) *mem-not-refl* **by** *blast*
next
case *False*
then consider (*x*,*y*) ∈ *r* | (*y*,*x*) ∈ *r*
using ‹*total-on A r*› *assms* **by** (*meson UNIV-I total-on-def*)
then show *?thesis*
by (*meson ordermap-mono assms mem-not-sym*)
qed

lemma *converse-ordermap-mono-iff*:
assumes *wf r total-on A r x* ∈ *A y* ∈ *A small A*
shows *ordermap A r y* ∈ *elts* (*ordermap A r x*) ⟷ (*y*, *x*) ∈ *r*
by (*metis assms converse-ordermap-mono ordermap-mono*)

lemma *ordermap-surj*: *elts* (*ordertype A r*) ⊆ *ordermap A r* ‘ *A*
unfolding *ordertype-def* **by** *simp*

lemma *ordermap-bij*:
assumes *wf r total-on A r small A*
shows *bij-betw* (*ordermap A r*) *A* (*elts* (*ordertype A r*))
unfolding *bij-betw-def*
proof (*intro conjI*)
show *inj-on* (*ordermap A r*) *A*
unfolding *inj-on-def* **by** (*metis assms mem-not-refl ordermap-mono total-on-def*)
show *ordermap A r* ‘ *A* = *elts* (*ordertype A r*)
by (*metis ordertype-def* ‹*small A*› *elts-of-set replacement*)
qed

lemma *ordermap-eq-iff* [*simp*]:
⟦*x* ∈ *A*; *y* ∈ *A*; *wf r*; *total-on A r*; *small A*⟧ ⟹ *ordermap A r x* = *ordermap A r y* ⟷ *x* = *y*
by (*metis bij-betw-iff-bijections ordermap-bij*)

lemma *inv-into-ordermap*: *α* ∈ *elts* (*ordertype A r*) ⟹ *inv-into A* (*ordermap A r*) *α* ∈ *A*
by (*meson in-mono inv-into-into ordermap-surj*)

lemma *ordertype-nat-imp-finite*:

assumes *ordertype A r = ord-of-nat m small A wf r total-on A r*
shows *finite A*
proof −
have *A ≈ elts m*
using *eqpoll-def assms ordermap-bij* **by** *fastforce*
then show *?thesis*
using *eqpoll-finite-iff finite-Ord-omega* **by** *blast*
qed

lemma *wf-ordertype-eqpoll*:
assumes *wf r total-on A r small A*
shows *elts* (*ordertype A r*) *≈ A*
using *assms eqpoll-def eqpoll-sym ordermap-bij* **by** *blast*

lemma *ordertype-eqpoll*:
assumes *small A*
shows *elts* (*ordertype A VWF*) *≈ A*
using *assms wf-ordertype-eqpoll total-VWF wf-VWF*
by (*simp add*: *wf-ordertype-eqpoll total-on-def*)

2.9 More advanced *ordertype* and *ordermap* results

lemma *ordermap-VWF-0* [*simp*]: *ordermap A VWF 0 = 0*
by (*simp add*: *ordermap wf-VWO VWF-def*)

lemma *ordertype-empty* [*simp*]: *ordertype* {} *r = 0*
by (*simp add*: *ordertype-def*)

lemma *ordertype-eq-0-iff* [*simp*]: ⟦*small X*; *wf r*⟧ ⟹ *ordertype X r = 0* ⟷ *X = {}*
by (*metis ordertype-def elts-of-set replacement image-is-empty zero-V-def*)

lemma *ordermap-mono-less*:
assumes (*w*, *x*) ∈ *r*
and *wf r trans r*
and *w ∈ A x ∈ A*
and *small A*
shows *ordermap A r w < ordermap A r x*
by (*simp add*: *OrdmemD assms ordermap-mono*)

lemma *ordermap-mono-le*:
assumes (*w*, *x*) ∈ *r* ∨ *w=x*
and *wf r trans r*
and *w ∈ A x ∈ A*
and *small A*
shows *ordermap A r w ≤ ordermap A r x*
by (*metis assms dual-order.strict-implies-order eq-refl ordermap-mono-less*)

lemma *converse-ordermap-le-mono*:

assumes *ordermap A r y* ≤ *ordermap A r x wf r total r* $x \in A$ *small A*
shows $(y, x) \in r \vee y{=}x$
by (*meson UNIV-I assms mem-not-refl ordermap-mono total-on-def vsubsetD*)

lemma *ordertype-mono*:
assumes $X \subseteq Y$ **and** *r*: *wf r trans r* **and** *small Y*
shows *ordertype X r* ≤ *ordertype Y r*
proof −
have *small X*
using *assms smaller-than-small* **by** *fastforce*
have ∗: *ordermap X r x* ≤ *ordermap Y r x* **for** *x*
using ‹*wf r*›
proof (*induction x rule*: *wf-induct-rule*)
case (*less x*)
have *ordermap X r z* < *ordermap Y r x* **if** $z \in X$ **and** *zx*: $(z,x) \in r$ **for** *z*
using *less* [*OF zx*] *assms*
by (*meson Ord-linear2 OrdmemD wf-Ord-ordermap ordermap-mono in-mono leD that*(*1*) *vsubsetD zx*)
then show *?case*
by (*auto simp add*: *ordermap* [*of - X x*] ‹*small X*› *Ord-mem-iff-lt set-image-le-iff less-eq-V-def r*)
qed
show *?thesis*
proof −
have *ordermap Y r* ‘ *Y* = *elts* (*ordertype Y r*)
by (*metis ordertype-def* ‹*small Y*› *elts-of-set replacement*)
then have *ordertype Y r* ∉ *ordermap X r* ‘ *X*
using ∗ ‹*X* ⊆ *Y*› **by** *fastforce*
then show *?thesis*
by (*metis Ord-linear2 Ord-mem-iff-lt ordertype-def wf-Ord-ordertype* ‹*small X*› *elts-of-set replacement r*)
qed
qed

corollary *ordertype-VWF-mono*:
assumes $X \subseteq Y$ *small Y*
shows *ordertype X VWF* ≤ *ordertype Y VWF*
using *assms* **by** (*simp add*: *ordertype-mono*)

lemma *ordertype-UNION-ge*:
assumes $A \in \mathcal{A}$ *wf r trans r* $\mathcal{A} \subseteq$ *Collect small small* $\mathcal{A}$
shows *ordertype A r* ≤ *ordertype* $(\bigcup \mathcal{A})$ *r*
by (*rule ordertype-mono*) (*use assms* **in** *auto*)

lemma *inv-ordermap-mono-less*:
assumes (*inv-into M* (*ordermap M r*) α, *inv-into M* (*ordermap M r*) β) ∈ *r*
and *small M* **and** α: $\alpha \in$ *elts* (*ordertype M r*) **and** β: $\beta \in$ *elts* (*ordertype M r*)
and *wf r trans r*
shows $\alpha < \beta$

proof −
have α = *ordermap M r* (*inv-into M* (*ordermap M r*) α)
by (*metis* α *f-inv-into-f ordermap-surj subset-eq*)
also have ... < *ordermap M r* (*inv-into M* (*ordermap M r*) β)
by (*meson* α β *assms in-mono inv-into-into ordermap-mono-less ordermap-surj*)
also have ... = β
by (*meson* β *f-inv-into-f in-mono ordermap-surj*)
finally show *?thesis* **.**
qed

lemma *inv-ordermap-mono-eq*:
assumes *inv-into M* (*ordermap M r*) α = *inv-into M* (*ordermap M r*) β
and $\alpha \in$ *elts* (*ordertype M r*) $\beta \in$ *elts* (*ordertype M r*)
shows $\alpha = \beta$
by (*metis assms f-inv-into-f ordermap-surj subsetD*)

lemma *inv-ordermap-VWF-mono-le*:
assumes *inv-into M* (*ordermap M VWF*) $\alpha \leq$ *inv-into M* (*ordermap M VWF*) β
and *M* $\subseteq$ *ON small M* **and** α: $\alpha \in$ *elts* (*ordertype M VWF*) **and** β: $\beta \in$ *elts* (*ordertype M VWF*)
shows $\alpha \leq \beta$
proof −
have α = *ordermap M VWF* (*inv-into M* (*ordermap M VWF*) α)
by (*metis* α *f-inv-into-f ordermap-surj subset-eq*)
also have ... ≤ *ordermap M VWF* (*inv-into M* (*ordermap M VWF*) β)
by (*metis ON-imp-Ord VWF-iff-Ord-less assms dual-order.strict-implies-order elts-of-set eq-refl inv-into-into order.not-eq-order-implies-strict ordermap-mono-less ordertype-def replacement trans-VWF wf-VWF*)
also have ... = β
by (*meson* β *f-inv-into-f in-mono ordermap-surj*)
finally show *?thesis* **.**
qed

lemma *inv-ordermap-VWF-mono-iff*:
assumes *M* $\subseteq$ *ON small M* **and** $\alpha \in$ *elts* (*ordertype M VWF*) **and** $\beta \in$ *elts* (*ordertype M VWF*)
shows *inv-into M* (*ordermap M VWF*) $\alpha \leq$ *inv-into M* (*ordermap M VWF*) β $\longleftrightarrow$ $\alpha \leq \beta$
by (*metis ON-imp-Ord Ord-linear-le assms dual-order.eq-iff inv-into-ordermap inv-ordermap-VWF-mono-le*)

lemma *inv-ordermap-VWF-strict-mono-iff*:
assumes *M* $\subseteq$ *ON small M* **and** $\alpha \in$ *elts* (*ordertype M VWF*) **and** $\beta \in$ *elts* (*ordertype M VWF*)
shows *inv-into M* (*ordermap M VWF*) $\alpha <$ *inv-into M* (*ordermap M VWF*) β $\longleftrightarrow$ $\alpha < \beta$
by (*simp add*: *assms inv-ordermap-VWF-mono-iff less-le-not-le*)

lemma *strict-mono-on-ordertype*:
assumes $M \subseteq ON$ *small* M
obtains f **where** $f \in$ *elts* (*ordertype* M *VWF*) $\to$ M *strict-mono-on* f (*elts* (*ordertype* M *VWF*))
proof
show *inv-into* M (*ordermap* M *VWF*) $\in$ *elts* (*ordertype* M *VWF*) $\to$ M
by (*meson Pi-I' in-mono inv-into-into ordermap-surj*)
show *strict-mono-on* (*inv-into* M (*ordermap* M *VWF*)) (*elts* (*ordertype* M *VWF*))
proof (*clarsimp simp*: *strict-mono-on-def*)
fix x y
assume $x \in$ *elts* (*ordertype* M *VWF*) $y \in$ *elts* (*ordertype* M *VWF*) $x < y$
then show *inv-into* M (*ordermap* M *VWF*) $x <$ *inv-into* M (*ordermap* M *VWF*) y
using *assms* **by** (*meson ON-imp-Ord Ord-linear2 inv-into-into inv-ordermap-VWF-mono-le leD ordermap-surj subsetD*)
qed
qed

lemma *ordermap-inc-eq*:
assumes $x \in A$ *small* A
and π: $\bigwedge x\ y.\ [\![x{\in}A;\ y{\in}A;\ (x,y) \in r]\!] \Longrightarrow (\pi\ x,\ \pi\ y) \in s$
and r: *wf* r *total-on* A r **and** *wf* s
shows *ordermap* (π ‘ A) s (π x) = *ordermap* A r x
using ‹*wf* r› ‹$x \in A$›
proof (*induction* x *rule*: *wf-induct-rule*)
case (*less* x)
then have *1*: $\{y \in A.\ (y,\ x) \in r\} = A \cap \{y.\ (y,\ x) \in r\}$
using r **by** *auto*
have *2*: $\{y \in \pi$ ‘ $A.\ (y,\ \pi\ x) \in s\} = \pi$ ‘ $A \cap \{y.\ (y,\ \pi\ x) \in s\}$
by *auto*
have *invπ*: $\bigwedge x\ y.\ [\![x{\in}A;\ y{\in}A;\ (\pi\ x,\ \pi\ y) \in s]\!] \Longrightarrow (x,\ y) \in r$
by (*metis* π ‹*wf* s› ‹*total-on* A r› *total-on-def wf-not-sym*)
have *eq*: f ‘ (π ‘ $A \cap \{y.\ (y,\ \pi\ x) \in s\}$) = ($f \circ \pi$) ‘ ($A \cap \{y.\ (y,\ x) \in r\}$) **for** f :: $'b \Rightarrow V$
using *less* **by** (*auto simp*: *image-subset-iff invπ* π)
show *?case*
using *less*
by (*simp add*: *ordermap* [*OF* ‹*wf* r›, *of* - x] *ordermap* [*OF* ‹*wf* s›, *of* - π x] *1 2 eq*)
qed

lemma *ordertype-inc-eq*:
assumes *small* A
and π: $\bigwedge x\ y.\ [\![x{\in}A;\ y{\in}A;\ (x,y) \in r]\!] \Longrightarrow (\pi\ x,\ \pi\ y) \in s$
and r: *wf* r *total-on* A r **and** *wf* s
shows *ordertype* (π ‘ A) s = *ordertype* A r
proof −
have *ordermap* (π ‘ A) s (π x) = *ordermap* A r x **if** $x \in A$ **for** x
using *assms that* **by** (*auto simp*: *ordermap-inc-eq*)

then show *?thesis*
unfolding *ordertype-def*
by (*metis* (*no-types*, *lifting*) *image-cong image-image*)
qed

lemma *ordertype-inc-le*:
assumes *small A small B*
and π: $\bigwedge x\ y.\ [\![x{\in}A;\ y{\in}A;\ (x,y) \in r]\!] \Longrightarrow (\pi\ x,\ \pi\ y) \in s$
and *r*: *wf r total-on A r* **and** *wf s trans s*
and π ' $A \subseteq B$
shows *ordertype A r* $\leq$ *ordertype B s*
by (*metis assms ordertype-inc-eq ordertype-mono*)

corollary *ordertype-VWF-inc-eq*:
assumes $A \subseteq ON\ \pi$ ' $A \subseteq ON$ *small A* **and** $\bigwedge x\ y.\ [\![x{\in}A;\ y{\in}A;\ x{<}y]\!] \Longrightarrow \pi\ x < \pi\ y$
shows *ordertype* (π ' *A*) *VWF* = *ordertype A VWF*
proof (*rule ordertype-inc-eq*)
show (π *x*, π *y*) $\in$ *VWF*
if $x \in A\ y \in A\ (x,\ y) \in VWF$ **for** *x y*
using *that ON-imp-Ord assms* **by** *auto*
show *total-on A VWF*
by (*meson UNIV-I total-VWF total-on-def*)
qed (*use assms* **in** *auto*)

lemma *ordertype-image-ordermap*:
assumes *small A X* $\subseteq$ *A wf r trans r total-on X r*
shows *ordertype* (*ordermap A r* ' *X*) *VWF* = *ordertype X r*
proof (*rule ordertype-inc-eq*)
show *small X*
by (*meson assms smaller-than-small*)
show (*ordermap A r x*, *ordermap A r y*) $\in$ *VWF*
if $x \in X\ y \in X\ (x,\ y) \in r$ **for** *x y*
by (*meson that wf-Ord-ordermap VWF-iff-Ord-less assms ordermap-mono-less subsetD*)
qed (*use assms* **in** *auto*)

lemma *ordertype-map-image*:
assumes *B* $\subseteq$ *A small A*
shows *ordertype* (*ordermap A VWF* ' *A* $-$ *ordermap A VWF* ' *B*) *VWF* = *ordertype* (*A* $-$ *B*) *VWF*
proof $-$
have *ordermap A VWF* ' *A* $-$ *ordermap A VWF* ' *B* = *ordermap A VWF* ' (*A* $-$ *B*)
using *assms* **by** *auto*
then have *ordertype* (*ordermap A VWF* ' *A* $-$ *ordermap A VWF* ' *B*) *VWF* = *ordertype* (*ordermap A VWF* ' (*A* $-$ *B*)) *VWF*
by *simp*
also have ... = *ordertype* (*A* $-$ *B*) *VWF*

using ‹*small A*› *ordertype-image-ordermap* **by** *fastforce*
finally show *?thesis* **.**
qed

proposition *ordertype-le-ordertype*:
assumes *r*: *wf r total-on A r* **and** *small A*
assumes *s*: *wf s total-on B s trans s* **and** *small B*
shows *ordertype A r* ≤ *ordertype B s* ⟷
(∃ *f* ∈ *A* → *B*. *inj-on f A* ∧ (∀ *x* ∈ *A*. ∀ *y* ∈ *A*. ((*x*,*y*) ∈ *r* ⟶ (*f x*, *f y*) ∈ *s*)))
(**is** *?lhs* = *?rhs*)
proof
assume *L*: *?lhs*
define *f* **where** *f* ≡ *inv-into B* (*ordermap B s*) ∘ *ordermap A r*
show *?rhs*
proof (*intro bexI conjI ballI impI*)
have *AB*: *elts* (*ordertype A r*) ⊆ *ordermap B s* ‘ *B*
by (*metis L assms*(7) *ordertype-def replacement set-of-elts small-elts subset-iff-less-eq-V*)
have *bijA*: *bij-betw* (*ordermap A r*) *A* (*elts* (*ordertype A r*))
using *ordermap-bij* ‹*small A*› *r* **by** *blast*
have *inv-into B* (*ordermap B s*) (*ordermap A r i*) ∈ *B* **if** *i* ∈ *A* **for** *i*
by (*meson L* ‹*small A*› *inv-into-into ordermap-in-ordertype ordermap-surj subsetD that vsubsetD*)
then show *f* ∈ *A* → *B*
by (*auto simp*: *Pi-iff f-def*)
show *inj-on f A*
proof (*clarsimp simp add*: *f-def inj-on-def*)
fix *x y*
assume *x* ∈ *A y* ∈ *A*
and *inv-into B* (*ordermap B s*) (*ordermap A r x*) = *inv-into B* (*ordermap B s*) (*ordermap A r y*)
then have *ordermap A r x* = *ordermap A r y*
by (*meson AB* ‹*small A*› *inv-into-injective ordermap-in-ordertype subsetD*)
then show *x* = *y*
by (*metis* ‹*x* ∈ *A*› ‹*y* ∈ *A*› *bijA bij-betw-inv-into-left*)
qed
next
fix *x y*
assume *x* ∈ *A y* ∈ *A* **and** (*x*, *y*) ∈ *r*
have ‡: *ordermap A r y* ∈ *ordermap B s* ‘ *B*
by (*meson L* ‹*y* ∈ *A*› ‹*small A*› *in-mono ordermap-in-ordertype ordermap-surj vsubsetD*)
moreover have †: ⋀*x*. *inv-into B* (*ordermap B s*) (*ordermap A r x*) = *f x*
by (*simp add*: *f-def*)
then have ∗: *ordermap B s* (*f y*) = *ordermap A r y*
using ‡ **by** (*metis f-inv-into-f*)
moreover have *ordermap A r x* ∈ *ordermap B s* ‘ *B*
by (*meson L* ‹*x* ∈ *A*› ‹*small A*› *in-mono ordermap-in-ordertype ordermap-surj*

vsubsetD)
  moreover have *ordermap A r x < ordermap A r y*
    using ∗ *r s* **by** (*metis* (*no-types*) *wf-Ord-ordermap OrdmemD* ‹(*x*, *y*) ∈ *r*› ‹*x* ∈ *A*› ‹*small A*› *ordermap-mono*)
  ultimately show (*f x*, *f y*) ∈ *s*
  using † *s* **by** (*metis assms*(7) *f-inv-into-f inv-into-into less-asym ordermap-mono-less total-on-def*)
 qed
next
 assume *R*: *?rhs*
 then obtain *f* **where** *f*: *f*∈*A* → *B inj-on f A* ∀ *x*∈*A*. ∀ *y*∈*A*. (*x*, *y*) ∈ *r* ⟶ (*f x*, *f y*) ∈ *s*
  by *blast*
 show *?lhs*
  by (*rule ordertype-inc-le* [**where** *π*=*f*]) (*use f assms* **in** *auto*)
qed

lemma *iso-imp-ordertype-eq-ordertype*:
 assumes *iso*: *iso r r′ f*
  and *wf r*
  and *Total r*
  and *sm*: *small* (*Field r*)
 shows *ordertype* (*Field r*) *r* = *ordertype* (*Field r′*) *r′*
 by (*metis* (*no-types*, *lifting*) *iso-forward iso-wf assms iso-Field ordertype-inc-eq sm*)

lemma *ordertype-infinite-ge-ω*:
 assumes *infinite A small A*
 shows *ordertype A VWF* ≥ *ω*
proof −
 have *inj-on* (*ordermap A VWF*) *A*
  by (*meson ordermap-bij* ‹*small A*› *bij-betw-def total-on-VWF wf-VWF*)
 then have *infinite* (*ordermap A VWF* ‘ *A*)
  using ‹*infinite A*› *finite-image-iff* **by** *blast*
 then show *?thesis*
  using *Ord-ordertype* ‹*small A*› *infinite-Ord-omega* **by** (*auto simp*: *ordertype-def*)
qed

lemma *ordertype-eqI*:
 assumes *wf r total-on A r small A wf s*
      bij-betw f A B (∀ *x* ∈ *A*. ∀ *y* ∈ *A*. (*f x*, *f y*) ∈ *s* ⟷ (*x*,*y*) ∈ *r*)
 shows *ordertype A r* = *ordertype B s*
 by (*metis assms bij-betw-imp-surj-on ordertype-inc-eq*)

lemma *ordermap-eq-self*:
 assumes *Ord α* **and** *x*: *x* ∈ *elts α*
 shows *ordermap* (*elts α*) *VWF x* = *x*
 using *Ord-in-Ord* [*OF assms*] *x*
proof (*induction x rule*: *Ord-induct*)

case (*step x*)
have *1*: {*y* ∈ *elts* α. (*y*, *x*) ∈ *VWF*} = *elts x* (**is** *?A* = -)
proof
show *?A* ⊆ *elts x*
using ‹*Ord* α› **by** *clarify* (*meson Ord-in-Ord Ord-mem-iff-lt VWF-iff-Ord-less step.hyps*)
show *elts x* ⊆ *?A*
using ‹*Ord* α› **by** *clarify* (*meson Ord-in-Ord Ord-trans OrdmemD VWF-iff-Ord-less step.prems*)
qed
show *?case*
using *step*
by (*simp add*: *ordermap* [*OF wf-VWF*, *of - x*] *1 Ord-trans* [*of - - α*] *step.prems* ‹*Ord* α› *cong*: *image-cong*)
qed

lemma *ordertype-eq-Ord* [*simp*]:
assumes *Ord* α
shows *ordertype* (*elts* α) *VWF* = α
using *assms ordermap-eq-self* [*OF assms*] **by** (*simp add*: *ordertype-def*)

proposition *ordertype-eq-iff*:
assumes α: *Ord* α **and** *r*: *wf r* **and** *small A total-on A r trans r*
shows *ordertype A r* = α ⟷
(∃ *f*. *bij-betw f A* (*elts* α) ∧ (∀ *x* ∈ *A*. ∀ *y* ∈ *A*. *f x* < *f y* ⟷ (*x*,*y*) ∈ *r*))
(**is** *?lhs* = *?rhs*)
proof *safe*
assume *eq*: α = *ordertype A r*
show ∃ *f*. *bij-betw f A* (*elts* (*ordertype A r*)) ∧ (∀ *x*∈*A*. ∀ *y*∈*A*. *f x* < *f y* ⟷ ((*x*, *y*) ∈ *r*))
proof (*intro exI conjI ballI*)
show *bij-betw* (*ordermap A r*) *A* (*elts* (*ordertype A r*))
by (*simp add*: *assms ordermap-bij*)
then show *ordermap A r x* < *ordermap A r y* ⟷ (*x*, *y*) ∈ *r*
if *x* ∈ *A y* ∈ *A* **for** *x y*
using *that assms*
by (*metis order.asym ordermap-mono-less total-on-def*)
qed
next
fix *f*
assume *f*: *bij-betw f A* (*elts* α) ∀ *x*∈*A*. ∀ *y*∈*A*. *f x* < *f y* ⟷ (*x*, *y*) ∈ *r*
have *ordertype A r* = *ordertype* (*elts* α) *VWF*
proof (*rule ordertype-eqI*)
show ∀ *x*∈*A*. ∀ *y*∈*A*. ((*f x*, *f y*) ∈ *VWF*) = ((*x*, *y*) ∈ *r*)
by (*meson Ord-in-Ord VWF-iff-Ord-less* α *bij-betwE f*)
qed (*use assms f* **in** *auto*)
then show *?lhs*
by (*simp add*: α)

qed

corollary *ordertype-VWF-eq-iff*:
assumes *Ord* α *small A*
shows *ordertype A VWF* $= \alpha \longleftrightarrow$
$(\exists f.$ *bij-betw f A* (*elts* α) $\wedge$ $(\forall x \in A.\ \forall y \in A.\ f\,x < f\,y \longleftrightarrow (x,y) \in VWF))$
by (*metis UNIV-I assms ordertype-eq-iff total-VWF total-on-def trans-VWF wf-VWF*)

lemma *ordertype-le-Ord*:
assumes *Ord* α $X \subseteq$ *elts* α
shows *ordertype X VWF* $\leq \alpha$
by (*metis assms ordertype-VWF-mono ordertype-eq-Ord small-elts*)

lemma *ordertype-inc-le-Ord*:
assumes *small A Ord* α
and π: $\bigwedge x\ y.\ [\![x{\in}A;\ y{\in}A;\ (x,y) \in r]\!] \Longrightarrow \pi\ x < \pi\ y$
and *wf r total-on A r*
and *sub*: π ‘ $A \subseteq$ *elts* α
shows *ordertype A r* $\leq \alpha$
proof −
have $\bigwedge x\ y.\ [\![x{\in}A;\ y{\in}A;\ (x,y) \in r]\!] \Longrightarrow (\pi\ x,\ \pi\ y) \in VWF$
by (*meson Ord-in-Ord VWF-iff-Ord-less* π ‹*Ord* α› *sub image-subset-iff*)
with *assms* **show** *?thesis*
by (*metis ordertype-inc-eq ordertype-le-Ord wf-VWF*)
qed

lemma *le-ordertype-obtains-subset*:
assumes α: $\beta \leq \alpha$ *ordertype H VWF* $= \alpha$ **and** *small H Ord* β
obtains *G* **where** $G \subseteq H$ *ordertype G VWF* $= \beta$
proof (*intro exI conjI that*)
let *?f* = *ordermap H VWF*
show ‡: *inv-into H ?f* ‘ *elts* $\beta \subseteq H$
unfolding *image-subset-iff*
by (*metis* α *inv-into-into ordermap-surj subsetD vsubsetD*)
have $\exists f.$ *bij-betw f* (*inv-into H ?f* ‘ *elts* β) (*elts* β) $\wedge$ $(\forall x{\in}$*inv-into H ?f* ‘ *elts* $\beta.$
$\forall y{\in}$*inv-into H ?f* ‘ *elts* $\beta.$ $(f\,x < f\,y) = ((x,\ y) \in VWF))$
proof (*intro exI conjI ballI iffI*)
show *bij-betw ?f* (*inv-into H ?f* ‘ *elts* β) (*elts* β)
using *ordermap-bij* [*OF wf-VWF total-on-VWF* ‹*small H*›] α
by (*metis bij-betw-inv-into-RIGHT bij-betw-subset less-eq-V-def* ‡)
next
fix *x y*
assume *x*: $x \in$ *inv-into H ?f* ‘ *elts* β
and *y*: $y \in$ *inv-into H ?f* ‘ *elts* β
show *?f x* < *?f y* **if** $(x,y) \in VWF$
using *that* ‡ ‹*small H*› *in-mono ordermap-mono-less x y* **by** *fastforce*
show $(x,y) \in VWF$ **if** *?f x* < *?f y*
using *that* ‡ ‹*small H*› *in-mono ordermap-mono-less* [*OF - wf-VWF trans-VWF*]

x y
by (*metis UNIV-I less-imp-not-less total-VWF total-on-def*)
qed
then show *ordertype* (*inv-into H ?f ‘ elts β*) *VWF* = *β*
by (*subst ordertype-eq-iff*) (*use assms* **in** *auto*)
qed

lemma *ordertype-infinite-ω*:
assumes *A* ⊆ *elts ω infinite A*
shows *ordertype A VWF* = *ω*
proof (*rule antisym*)
show *ordertype A VWF* ≤ *ω*
by (*simp add*: *assms ordertype-le-Ord*)
show *ω* ≤ *ordertype A VWF*
using *assms down ordertype-infinite-ge-ω* **by** *auto*
qed

For infinite sets of natural numbers

lemma *ordertype-nat-ω*:
assumes *infinite N* **shows** *ordertype N less-than* = *ω*
proof −
have *small N*
by (*meson inj-on-def ord-of-nat-inject small-def small-iff-range small-image-nat-V*)
have *ordertype* (*ord-of-nat ‘ N*) *VWF* = *ω*
by (*force simp*: *assms finite-image-iff inj-on-def intro*: *ordertype-infinite-ω*)
moreover have *ordertype* (*ord-of-nat ‘ N*) *VWF* = *ordertype N less-than*
by (*auto intro*: *ordertype-inc-eq* ‹*small N*›)
ultimately show *?thesis*
by *simp*
qed

proposition *ordertype-eq-ordertype*:
assumes *r*: *wf r total-on A r trans r* **and** *small A*
assumes *s*: *wf s total-on B s trans s* **and** *small B*
shows *ordertype A r* = *ordertype B s* ⟷
(∃*f*. *bij-betw f A B* ∧ (∀ *x* ∈ *A*. ∀ *y* ∈ *A*. (*f x*, *f y*) ∈ *s* ⟷ (*x*,*y*) ∈ *r*))
(**is** *?lhs* = *?rhs*)
proof
assume *L*: *?lhs*
define *γ* **where** *γ* = *ordertype A r*
have *A*: *bij-betw* (*ordermap A r*) *A* (*ordermap A r ‘ A*)
by (*meson ordermap-bij assms*(*4*) *bij-betw-def r*)
have *B*: *bij-betw* (*ordermap B s*) *B* (*ordermap B s ‘ B*)
by (*meson ordermap-bij assms*(*8*) *bij-betw-def s*)
define *f* **where** *f* ≡ *inv-into B* (*ordermap B s*) *o ordermap A r*
show *?rhs*
proof (*intro exI conjI*)
have *bijA*: *bij-betw* (*ordermap A r*) *A* (*elts γ*)
unfolding *γ-def* **using** *ordermap-bij* ‹*small A*› *r* **by** *blast*

moreover have *bijB*: *bij-betw* (*ordermap B s*) *B* (*elts* γ)
by (*simp add*: *L* γ*-def ordermap-bij* ‹*small B*› *s*)
ultimately show *bij*: *bij-betw f A B*
unfolding *f-def* **using** *bij-betw-comp-iff bij-betw-inv-into* **by** *blast*
have *invB*: $\bigwedge\alpha.\ \alpha \in elts\ \gamma \Longrightarrow ordermap\ B\ s\ (inv\text{-}into\ B\ (ordermap\ B\ s)\ \alpha) = \alpha$
by (*meson bijB bij-betw-inv-into-right*)
have *ordermap-A-*γ: $\bigwedge a.\ a \in A \Longrightarrow ordermap\ A\ r\ a \in elts\ \gamma$
using *bijA bij-betwE* **by** *auto*
have *f-in-B*: $\bigwedge a.\ a \in A \Longrightarrow f\ a \in B$
using *bij bij-betwE* **by** *fastforce*
show $\forall\, x{\in}A.\ \forall\, y{\in}A.\ (f\ x, f\ y) \in s \longleftrightarrow (x, y) \in r$
proof (*intro iffI ballI*)
fix *x y*
assume $x \in A\ y \in A$ **and** *ins*: $(f\ x, f\ y) \in s$
then have *ordermap A r x* $<$ *ordermap A r y*
unfolding *o-def*
by (*metis* (*mono-tags*, *lifting*) *f-def* ‹*small B*› *comp-apply f-in-B invB ordermap-A-*γ *ordermap-mono-less s*(*1*) *s*(*3*))
then show $(x, y) \in r$
by (*metis* ‹$x \in A$› ‹$y \in A$› ‹*small A*› *order.asym ordermap-mono-less r total-on-def*)
next
fix *x y*
assume $x \in A\ y \in A$ **and** $(x, y) \in r$
then have *ordermap A r x* $<$ *ordermap A r y*
by (*simp add*: ‹*small A*› *ordermap-mono-less r*)
then have $(f\ y, f\ x) \notin s$
by (*metis* (*mono-tags*, *lifting*) ‹$x \in A$› ‹$y \in A$› ‹*small B*› *comp-apply f-def f-in-B invB order.asym ordermap-A-*γ *ordermap-mono-less s*(*1*) *s*(*3*))
moreover have $f\ y \neq f\ x$
by (*metis* ‹$(x, y) \in r$› ‹$x \in A$› ‹$y \in A$› *bij bij-betw-inv-into-left r*(*1*) *wf-not-sym*)
ultimately show $(f\ x, f\ y) \in s$
by (*meson* ‹$x \in A$› ‹$y \in A$› *f-in-B s*(*2*) *total-on-def*)
qed
qed
next
assume *?rhs*
then show *?lhs*
using *assms ordertype-eqI* **by** *blast*
qed

corollary *ordertype-eq-ordertype-iso*:
assumes *r*: *wf r total-on A r trans r* **and** *small A* **and** *FA*: *Field r* $= A$
assumes *s*: *wf s total-on B s trans s* **and** *small B* **and** *FB*: *Field s* $= B$
shows *ordertype A r* $=$ *ordertype B s* $\longleftrightarrow$ $(\exists f.\ iso\ r\ s\ f)$
(**is** *?lhs* $=$ *?rhs*)
proof

assume *L*: *?lhs*
then obtain *f* **where** *bij-betw f A B* $\forall x \in A.\ \forall y \in A.\ (f\ x, f\ y) \in s \longleftrightarrow (x,y) \in r$
using *assms ordertype-eq-ordertype* **by** *blast*
then show *?rhs*
using *FA FB iso-iff2* **by** *blast*
next
assume *?rhs*
then show *?lhs*
using *FA FB* ‹*small A*› *iso-imp-ordertype-eq-ordertype r* **by** *blast*
qed

lemma *Limit-ordertype-imp-Field-Restr*:
assumes *Lim*: *Limit* (*ordertype A r*) **and** *r*: *wf r total-on A r* **and** *small A*
shows *Field* (*Restr r A*) = *A*
proof −
have $\exists\, y{\in}A.\ (x,y) \in r$ **if** $x \in A$ **for** *x*
proof −
let *?oy* = *succ* (*ordermap A r x*)
have §: *?oy* ∈ *elts* (*ordertype A r*)
by (*simp add*: *Lim* ‹*small A*› *ordermap-in-ordertype succ-in-Limit-iff that*)
then have *A*: *inv-into A* (*ordermap A r*) *?oy* ∈ *A*
by (*simp add*: *inv-into-ordermap*)
moreover have (*x*, *inv-into A* (*ordermap A r*) *?oy*) ∈ *r*
proof −
have *ordermap A r x* ∈ *elts* (*ordermap A r* (*inv-into A* (*ordermap A r*) *?oy*))
by (*metis* § *elts-succ f-inv-into-f insert-iff ordermap-surj subsetD*)
then show *?thesis*
by (*metis* ‹*small A*› *A converse-ordermap-mono r that*)
qed
ultimately show *?thesis* **..**
qed
then have *A* ⊆ *Field* (*Restr r A*)
by (*auto simp*: *Field-def*)
then show *?thesis*
by (*simp add*: *Field-Restr-subset subset-antisym*)
qed

lemma *ordertype-Field-Restr*:
assumes *wf r total-on A r trans r small A Field* (*Restr r A*) = *A*
shows *ordertype* (*Field* (*Restr r A*)) (*Restr r A*) = *ordertype A r*
using *assms* **by** (*force simp*: *ordertype-eq-ordertype wf-Restr total-on-def trans-Restr*)

proposition *ordertype-eq-ordertype-iso-Restr*:
assumes *r*: *wf r total-on A r trans r* **and** *small A* **and** *FA*: *Field* (*Restr r A*) = *A*
assumes *s*: *wf s total-on B s trans s* **and** *small B* **and** *FB*: *Field* (*Restr s B*) = *B*
shows *ordertype A r* = *ordertype B s* $\longleftrightarrow$ ($\exists f.$ *iso* (*Restr r A*) (*Restr s B*) *f*)

(**is** *?lhs = ?rhs*)
proof
assume *L*: *?lhs*
then obtain *f* **where** *bij-betw f A B* ∀ *x* ∈ *A*. ∀ *y* ∈ *A*. (*f x*, *f y*) ∈ *s* ⟷ (*x*,*y*) ∈ *r*
using *assms ordertype-eq-ordertype* **by** *blast*
then show *?rhs*
using *FA FB bij-betwE* **unfolding** *iso-iff2* **by** *fastforce*
next
assume *?rhs*
moreover
have *ordertype* (*Field* (*Restr r A*)) (*Restr r A*) = *ordertype A r*
using *FA* ‹*small A*› *ordertype-Field-Restr r* **by** *blast*
moreover
have *ordertype* (*Field* (*Restr s B*)) (*Restr s B*) = *ordertype B s*
using *FB* ‹*small B*› *ordertype-Field-Restr s* **by** *blast*
ultimately show *?lhs*
using *iso-imp-ordertype-eq-ordertype FA FB* ‹*small A*› *r*
by (*fastforce intro*: *total-on-imp-Total-Restr trans-Restr wf-Int1*)
qed

lemma *ordermap-insert*:
assumes *Ord* α **and** *y*: *Ord y y* ≤ α **and** *U*: *U* ⊆ *elts* α
shows *ordermap* (*insert* α *U*) *VWF y* = *ordermap U VWF y*
using *y*
proof (*induction rule*: *Ord-induct*)
case (*step y*)
then have *1*: {*u* ∈ *U*. (*u*, *y*) ∈ *VWF*} = *elts y* ∩ *U*
apply (*simp add*: *set-eq-iff*)
by (*meson Ord-in-Ord Ord-mem-iff-lt VWF-iff-Ord-less assms subsetD*)
have *2*: {*u* ∈ *insert* α *U*. (*u*, *y*) ∈ *VWF*} = *elts y* ∩ *U*
apply (*simp add*: *set-eq-iff*)
by (*meson Ord-in-Ord Ord-mem-iff-lt VWF-iff-Ord-less assms leD step.hyps step.prems subsetD*)
show *?case*
using *step*
apply (*simp only*: *ordermap* [*OF wf-VWF*, *of - y*] *1 2*)
by (*meson Int-lower1 Ord-is-Transset Sup.SUP-cong Transset-def assms*(*1*) *in-mono vsubsetD*)
qed

lemma *ordertype-insert*:
assumes *Ord* α **and** *U*: *U* ⊆ *elts* α
shows *ordertype* (*insert* α *U*) *VWF* = *succ* (*ordertype U VWF*)
proof −
have †: {*y* ∈ *insert* α *U*. (*y*, α) ∈ *VWF*} = *U* {*y* ∈ *U*. (*y*, α) ∈ *VWF*} = *U*
using *Ord-in-Ord OrdmemD assms* **by** *auto*
have *eq*: ⋀*x*. *x* ∈ *U* ⟹ *ordermap* (*insert* α *U*) *VWF x* = *ordermap U VWF x*
by (*meson Ord-in-Ord Ord-is-Transset Transset-def U assms*(*1*) *in-mono or-*

dermap-insert)
have *ordertype* (*insert* α *U*) *VWF* =
ZFC-in-HOL.set (*insert* (*ordermap U VWF* α) (*ordermap U VWF* ‘ *U*))
by (*simp add*: *ordertype-def ordermap-insert assms eq*)
also have ... = *succ* (*ZFC-in-HOL.set* (*ordermap U VWF* ‘ *U*))
using † *U* **by** (*simp add*: *ordermap* [*OF wf-VWF*, *of* - α] *down succ-def vinsert-def*)
also have ... = *succ* (*ordertype U VWF*)
by (*simp add*: *ordertype-def*)
finally show *?thesis* **.**
qed

lemma *finite-ordertype-le-card*:
assumes *finite A wf r trans r*
shows *ordertype A r* $\leq$ *ord-of-nat* (*card A*)
proof −
have *Ord* (*ordertype A r*)
by (*simp add*: *wf-Ord-ordertype assms*)
moreover have *ordermap A r* ‘ *A* = *elts* (*ordertype A r*)
by (*simp add*: *ordertype-def finite-imp-small* ‹*finite A*›)
moreover have *card* (*ordermap A r* ‘ *A*) $\leq$ *card A*
using ‹*finite A*› *card-image-le* **by** *blast*
ultimately show *?thesis*
by (*metis Ord-linear-le Ord-ord-of-nat* ‹*finite A*› *card-ord-of-nat card-seteq finite-imageI less-eq-V-def*)
qed

lemma *ordertype-VWF-*ω:
assumes *finite A*
shows *ordertype A VWF* $\in$ *elts* ω
proof −
have *finite* (*ordermap A VWF* ‘ *A*)
using *assms* **by** *blast*
then have *ordertype A VWF* $<$ ω
by (*meson Ord-*ω *OrdmemD trans-VWF wf-VWF assms finite-ordertype-le-card le-less-trans ord-of-nat-*ω)
then show *?thesis*
by (*simp add*: *Ord-mem-iff-lt*)
qed

lemma *ordertype-VWF-finite-nat*:
assumes *finite A*
shows *ordertype A VWF* = *ord-of-nat* (*card A*)
by (*metis finite-imp-small ordermap-bij total-on-VWF wf-VWF* ω*-def assms bij-betw-same-card card-ord-of-nat elts-of-set f-inv-into-f inf ordertype-VWF-*ω)

lemma *finite-ordertype-eq-card*:
assumes *small A wf r trans r total-on A r*
shows *ordertype A r* = *ord-of-nat m* $\longleftrightarrow$ *finite A* $\wedge$ *card A* = *m*

```
  using ordermap-bij [OF ‹wf r›]
proof −
  have ∗: bij-betw (ordermap A r) A (elts (ordertype A r))
    by (simp add: assms ordermap-bij)
  moreover have card (ordermap A r ‘ A) = card A
    by (meson bij-betw-def ∗ card-image)
  ultimately show ?thesis
   using assms bij-betw-finite bij-betw-imp-surj-on finite-Ord-omega ordertype-VWF-finite-nat
wf-Ord-ordertype by fastforce
qed


lemma ex-bij-betw-strict-mono-card:
  assumes finite M M ⊆ ON
  obtains h where bij-betw h {..<card M} M and strict-mono-on h {..<card M}
proof −
  have bij: bij-betw (ordermap M VWF) M (elts (card M))
    using Finite-V ‹finite M› ordermap-bij ordertype-VWF-finite-nat by fastforce
  let ?h = (inv-into M (ordermap M VWF)) ∘ ord-of-nat
  show thesis
  proof
    show bijh: bij-betw ?h {..<card M} M
    proof (rule bij-betw-trans)
      show bij-betw ord-of-nat {..<card M} (elts (card M))
        by (simp add: bij-betw-def elts-ord-of-nat inj-on-def)
      show bij-betw (inv-into M (ordermap M VWF)) (elts (card M)) M
      using Finite-V assms bij-betw-inv-into ordermap-bij ordertype-VWF-finite-nat
by fastforce
    qed
    show strict-mono-on ?h {..<card M}
    proof −
      have ?h m < ?h n
        if m < n n < card M for m n
      proof (rule ccontr)
        obtain mn: m ∈ elts (ordertype M VWF) n ∈ elts (ordertype M VWF)
         using ‹m < n› ‹n < card M› ‹finite M› ordertype-VWF-finite-nat by auto
        have ord: Ord (?h m) Ord (?h n)
          using bijh assms(2) bij-betwE that by fastforce+
        moreover
        assume ¬ ?h m < ?h n
        ultimately consider ?h m = ?h n | ?h m > ?h n
          using Ord-linear-lt by blast
        then show False
        proof cases
          case 1
          then have m = n
            by (metis inv-ordermap-mono-eq mn comp-apply ord-of-nat-inject)
          with ‹m < n› show False by blast
        next
```

```
        case 2
        then have ord-of-nat n ≤ ord-of-nat m
            by (metis Finite-V mn assms comp-def inv-ordermap-VWF-mono-le
less-imp-le)
        then show ?thesis
          using leD ‹m < n› by blast
       qed
      qed
      with assms show ?thesis
        by (auto simp: strict-mono-on-def)
    qed
  qed
qed

lemma ordertype-finite-less-than [simp]:
  assumes finite A shows ordertype A less-than = card A
proof −
  let ?M = ord-of-nat ‘ A
  obtain M: finite ?M ?M ⊆ ON
    using Ord-ord-of-nat assms by blast
  have ordertype A less-than = ordertype ?M VWF
    by (rule ordertype-inc-eq [symmetric]) (use assms finite-imp-small total-on-def
in ‹force+›)
  also have ... = card A
  proof (subst ordertype-eq-iff)
    let ?M = ord-of-nat ‘ A
    obtain h where bijh: bij-betw h {..<card A} ?M and smh: strict-mono-on h
{..<card A}
     by (metis M card-image ex-bij-betw-strict-mono-card inj-on-def ord-of-nat-inject)
    define f where f ≡ ord-of-nat ∘ inv-into {..<card A} h
   show ∃f. bij-betw f ?M (elts (card A)) ∧ (∀x∈?M. ∀y∈?M. f x < f y ⟷ ((x,
y) ∈ VWF))
    proof (intro exI conjI ballI)
     have bij-betw (ord-of-nat ∘ inv-into {..<card A} h) (ord-of-nat ‘ A) (ord-of-nat
‘ {..<card A})
      by (meson UNIV-I bijh bij-betw-def bij-betw-inv-into bij-betw-subset bij-betw-trans
inj-ord-of-nat subsetI)
      then show bij-betw f ?M (elts (card A))
        by (metis elts-ord-of-nat f-def)
    next
      fix x y
      assume xy: x ∈ ?M y ∈ ?M
      then obtain m n where x = ord-of-nat m y = ord-of-nat n
        by auto
     have (f x < f y) ⟷ ((h ∘ inv-into {..<card A} h) x < (h ∘ inv-into {..<card
A} h) y)
        unfolding f-def using smh bij-betw-imp-surj-on [OF bijh]
        apply simp
        by (metis (mono-tags, lifting) inv-into-into not-less-iff-gr-or-eq order.asym
```

strict-mono-onD xy)
also have ... = (*x* < *y*)
using *bijh*
by (*simp add*: *bij-betw-inv-into-right xy*)
also have ... ⟷ ((*x*, *y*) ∈ *VWF*)
using *M*(*2*) *ON-imp-Ord xy* **by** *auto*
finally show (*f x* < *f y*) ⟷ ((*x*, *y*) ∈ *VWF*) **.**
qed
qed *auto*
finally show *?thesis* **.**
qed

2.10 Cardinality of a set

definition
vcard :: *V*⇒*V*
where *vcard a* ≡ (*LEAST i*. *Ord i* ∧ *elts i* ≈ *elts a*)

definition
Card:: *V*⇒*bool* **where** *Card i* ≡ *i* = *vcard i*

abbreviation *CARD* **where** *CARD* ≡ *Collect Card*

lemma *cardinal-cong*: *elts x* ≈ *elts y* ⟹ *vcard x* = *vcard y*
unfolding *vcard-def* **by** (*meson eqpoll-sym eqpoll-trans*)

lemma *Card-cardinal-eq*: *Card* κ ⟹ *vcard* κ = κ
by (*simp add*: *Card-def*)

lemma *Card-is-Ord*:
assumes *Card* κ **shows** *Ord* κ
proof −
obtain α **where** *Ord* α *elts* α ≈ *elts* κ
using *Ord-ordertype ordertype-eqpoll* **by** *blast*
then have *Ord* (*LEAST i*. *Ord i* ∧ *elts i* ≈ *elts* κ)
by (*metis Ord-Least*)
then show *?thesis*
using *Card-def vcard-def assms* **by** *auto*
qed

lemma *cardinal-eqpoll*: *elts* (*vcard a*) ≈ *elts a*
unfolding *vcard-def*
using *ordertype-eqpoll* [*of elts a*] *Ord-LeastI* **by** (*meson Ord-ordertype small-elts*)

lemma *inj-into-vcard*:
obtains *f* **where** *f* ∈ *elts A* → *elts* (*vcard A*) *inj-on f* (*elts A*)
using *cardinal-eqpoll* [*of A*] *inj-on-the-inv-into the-inv-into-onto*
by (*fastforce simp*: *Pi-iff bij-betw-def eqpoll-def*)

lemma *cardinal-idem* [*simp*]: *vcard* (*vcard a*) = *vcard a*
using *cardinal-cong cardinal-eqpoll* **by** *blast*

every natural number is a (finite) cardinal

lemma *nat-into-Card*:
assumes $\alpha \in elts\ \omega$ **shows** *Card*(α)
proof (*unfold Card-def vcard-def*, *rule sym*)
obtain *n* **where** *n*: $\alpha = ord\text{-}of\text{-}nat\ n$
by (*metis ω-def assms elts-of-set imageE inf*)
have *Ord*(α) **using** *assms* **by** *auto*
moreover
{ fix β
assume $\beta < \alpha$ *Ord* β *elts* $\beta \approx elts\ \alpha$
with *n* **have** *elts* $\beta \approx \{..<n\}$
by (*simp add*: *ord-of-nat-eq-initial* [*of n*] *eqpoll-trans inj-on-def inj-on-image-eqpoll-self*)
hence *False* **using** *assms n* ‹*Ord* β› ‹$\beta < \alpha$› ‹*Ord*(α)›
by (*metis* ‹*elts* $\beta \approx$ *elts* α› *card-seteq eqpoll-finite-iff eqpoll-iff-card finite-lessThan less-eq-V-def less-le-not-le order-refl*)
}
ultimately
show (*LEAST i*. *Ord i* ∧ *elts i* ≈ *elts* α) = α
by (*metis* (*no-types*, *lifting*) *Least-equality Ord-linear-le eqpoll-refl less-le-not-le*)
qed

lemma *Card-ord-of-nat* [*simp*]: *Card* (*ord-of-nat n*)
by (*simp add*: *ω-def nat-into-Card*)

lemma *Card-0* [*iff*]: *Card 0*
by (*simp add*: *nat-into-Card*)

lemma *CardI*: ⟦*Ord i*; ⋀*j*. ⟦*j* < *i*; *Ord j*⟧ ⟹ ¬ *elts j* ≈ *elts i*⟧ ⟹ *Card i*
unfolding *Card-def vcard-def*
by (*metis Ord-Least Ord-linear-lt cardinal-eqpoll eqpoll-refl not-less-Ord-Least vcard-def*)

lemma *vcard-0* [*simp*]: *vcard 0* = *0*
using *Card-0 Card-def* **by** *auto*

lemma *Ord-cardinal* [*simp*,*intro*!]: *Ord*(*vcard a*)
unfolding *vcard-def* **by** (*metis Card-def Card-is-Ord cardinal-cong cardinal-eqpoll vcard-def*)

The cardinals are the initial ordinals.

lemma *Card-iff-initial*: *Card* $\kappa \longleftrightarrow$ *Ord* $\kappa \wedge (\forall \alpha.\ Ord\ \alpha \wedge \alpha < \kappa \longrightarrow \sim elts\ \alpha \approx elts\ \kappa)$
proof −
{ fix *j*
assume κ: *Card* κ *elts j* ≈ *elts* κ *Ord j*
assume *j* < κ

```
    also have ... = (LEAST i. Ord i ∧ elts i ≈ elts κ) using κ
      by (simp add: Card-def vcard-def)
    finally have j < (LEAST i. Ord i ∧ elts i ≈ elts κ) .
    hence False using κ
      using not-less-Ord-Least by fastforce
  }
  then show ?thesis
    by (blast intro: CardI Card-is-Ord)
qed

lemma Card-ω [iff]: Card ω
proof −
  have ⋀α f. ⟦α ∈ elts ω; bij-betw f (elts α) (elts ω)⟧ ⟹ False
  using bij-betw-finite finite-Ord-omega infinite-ω by blast
  then show ?thesis
    by (meson CardI Ord-ω Ord-mem-iff-lt eqpoll-def)
qed

lemma lt-Card-imp-lesspoll: ⟦i < a; Card a; Ord i⟧ ⟹ elts i ≺ elts a
  by (meson Card-iff-initial less-eq-V-def less-imp-le lesspoll-def subset-imp-lepoll)

lemma lepoll-imp-Card-le:
  assumes elts a ≲ elts b shows vcard a ≤ vcard b
using Ord-cardinal [of a] Ord-cardinal [of b]
proof (cases rule: Ord-linear-le)
  case le thus ?thesis .
next
  case ge
  have elts b ≈ elts (vcard b)
    by (simp add: cardinal-eqpoll eqpoll-sym)
  also have ... ≲ elts (vcard a)
    by (meson ge less-eq-V-def subset-imp-lepoll)
  also have ... ≈ elts a
    by (simp add: cardinal-eqpoll)
  finally have elts b ≲ elts a  .
  hence elts a ≈ elts b
    using assms lepoll-antisym by blast
  hence vcard a = vcard b
    by (rule cardinal-cong)
  thus ?thesis by simp
qed

lemma lepoll-cardinal-le: ⟦elts A ≲ elts i; Ord i⟧ ⟹ vcard A ≤ i
   by (metis Ord-Least Ord-linear2 dual-order.trans eqpoll-refl lepoll-imp-Card-le
not-less-Ord-Least vcard-def)

lemma cardinal-le-lepoll: vcard A ≤ α ⟹ elts A ≲ elts α
  by (meson cardinal-eqpoll eqpoll-sym lepoll-trans1 less-eq-V-def subset-imp-lepoll)
```

lemma *lesspoll-imp-Card-less*:
assumes *elts a* $\prec$ *elts b* **shows** *vcard a* $<$ *vcard b*
by (*metis assms cardinal-eqpoll eqpoll-sym eqpoll-trans le-neq-trans lepoll-imp-Card-le lesspoll-def*)

lemma *Card-Union* [*simp*,*intro*]:
assumes *A*: $\bigwedge x.\ x \in A \Longrightarrow Card(x)$ **shows** $Card(\bigsqcup A)$
proof (*rule CardI*)
show $Ord(\bigsqcup A)$ **using** *A*
by (*simp add*: *Card-is-Ord Ord-Sup*)
next
fix *j*
assume *j*: $j < \bigsqcup A$ *Ord j*
hence $\exists\, c{\in}A.\ j < c \wedge Card(c)$ **using** *A*
by (*meson Card-is-Ord Ord-linear2 ZFC-in-HOL.Sup-least leD*)
then obtain *c* **where** *c*: $c{\in}A$ $j < c$ $Card(c)$
by *blast*
hence *jls*: *elts j* $\prec$ *elts c*
using *j*(*2*) *lt-Card-imp-lesspoll* **by** *blast*
{ assume *eqp*: *elts j* $\approx$ *elts* $(\bigsqcup A)$
have *elts c* $\lesssim$ *elts* $(\bigsqcup A)$ **using** *c*
using *Sup-V-def ZFC-in-HOL.Sup-upper j*(*1*) *less-eq-V-def subset-imp-lepoll*
by *fastforce*
also have ... $\approx$ *elts j* **by** (*rule eqpoll-sym* [*OF eqp*])
also have ... $\prec$ *elts c* **by** (*rule jls*)
finally have *elts c* $\prec$ *elts c* **.**
hence *False*
by *auto*
} thus $\neg$ *elts j* $\approx$ *elts* $(\bigsqcup A)$ **by** *blast*
qed

lemma *Card-UN*: $(\bigwedge x.\ x \in A \Longrightarrow Card(K\ x)) ==> Card(Sup\ (K\ `\ A))$
by *blast*

2.11 Transfinite recursion for definitions based on the three cases of ordinals

definition
transrec3 :: [*V*, [*V*,*V*]⇒*V*, [*V*,*V*⇒*V*]⇒*V*, *V*] ⇒ *V* **where**
transrec3 a b c ≡
transrec (λ*r x*.
if x=0 then a
else if Limit x then c x (λ*y* ∈ *elts x*. *r y*)
else b(*pred x*) (*r* (*pred x*)))

lemma *transrec3-0* [*simp*]: *transrec3 a b c 0* = *a*
by (*simp add*: *transrec transrec3-def*)

lemma *transrec3-succ* [*simp*]:
transrec3 a b c (*succ i*) = *b i* (*transrec3 a b c i*)
by (*simp add*: *transrec transrec3-def*)

lemma *transrec3-Limit* [*simp*]:
Limit i ⟹ *transrec3 a b c i* = *c i* (λ*j* ∈ *elts i*. *transrec3 a b c j*)
unfolding *transrec3-def*
by (*subst transrec*) *auto*

2.12 Cardinal Addition

definition *cadd* :: [*V*, *V*]⇒ *V* (**infixl** ‹⊕› *65*)
where *κ* ⊕ *μ* ≡ *vcard* (*κ* ⨄ *μ*)

2.12.1 Cardinal addition is commutative

lemma *vsum-commute-eqpoll*: *elts* (*a*⨄ *b*) ≈ *elts* (*b*⨄ *a*)
proof −
have *bij-betw* (λ*z* ∈ *elts* (*a*⨄ *b*). *sum-case Inr Inl z*) (*elts* (*a*⨄ *b*)) (*elts* (*b*⨄ *a*))
unfolding *bij-betw-def*
proof (*intro conjI inj-onI*)
show *restrict* (*sum-case Inr Inl*) (*elts* (*a* ⨄ *b*)) ‘ *elts* (*a* ⨄ *b*) = *elts* (*b* ⨄ *a*)
apply *auto*
apply (*metis* (*no-types*) *imageI sum-case-Inr sum-iff*)
by (*metis Inl-in-sum-iff imageI sum-case-Inl*)
qed *auto*
then show *?thesis*
using *eqpoll-def* **by** *blast*
qed

lemma *cadd-commute*: *i* ⊕ *j* = *j* ⊕ *i*
by (*simp add*: *cadd-def cardinal-cong vsum-commute-eqpoll*)

2.12.2 Cardinal addition is associative

lemma *sum-assoc-bij*:
bij-betw (λ*z* ∈ *elts* ((*a*⨄ *b*)⨄ *c*). *sum-case*(*sum-case Inl* (λ*y*. *Inr*(*Inl y*))) (λ*y*. *Inr*(*Inr y*)) *z*)
(*elts* ((*a*⨄ *b*)⨄ *c*)) (*elts* (*a*⨄ (*b*⨄ *c*)))
by (*rule-tac f′* = *sum-case* (λ*x*. *Inl* (*Inl x*)) (*sum-case* (λ*x*. *Inl* (*Inr x*)) *Inr*)
in *bij-betw-byWitness*) *auto*

lemma *sum-assoc-eqpoll*: *elts* ((*a*⨄ *b*)⨄ *c*) ≈ *elts* (*a*⨄ (*b*⨄ *c*))
unfolding *eqpoll-def* **by** (*metis sum-assoc-bij*)

lemma *elts-vcard-vsum-eqpoll*: *elts* (*vcard* (*i* ⨄ *j*)) ≈ *Inl* ‘ *elts i* ∪ *Inr* ‘ *elts j*
proof −
have *elts* (*i* ⨄ *j*) ≈ *Inl* ‘ *elts i* ∪ *Inr* ‘ *elts j*
by (*simp add*: *elts-vsum*)
then show *?thesis*

using *cardinal-eqpoll eqpoll-trans* **by** *blast*
qed

lemma *cadd-assoc*: $(i \oplus j) \oplus k = i \oplus (j \oplus k)$
proof (*unfold cadd-def*, *rule cardinal-cong*)
have *elts* (*vcard*($i \uplus j$) $\uplus$ k) $\approx$ *elts* (($i \uplus j$) $\uplus$ k)
by (*auto simp*: *disjnt-def elts-vsum elts-vcard-vsum-eqpoll intro*: *Un-eqpoll-cong*)
also have ... $\approx$ *elts* ($i \uplus$ ($j \uplus k$))
by (*rule sum-assoc-eqpoll*)
also have ... $\approx$ *elts* ($i \uplus$ *vcard*($j \uplus k$))
by (*auto simp*: *disjnt-def elts-vsum elts-vcard-vsum-eqpoll* [*THEN eqpoll-sym*] *intro*: *Un-eqpoll-cong*)
finally show *elts* (*vcard* ($i \uplus j$) $\uplus$ k) $\approx$ *elts* ($i \uplus$ *vcard* ($j \uplus k$)) **.**
qed

0 is the identity for addition

lemma *vsum-0-eqpoll*: *elts* ($0 \uplus a$) $\approx$ *elts* a
by (*simp add*: *elts-vsum*)

lemma *cadd-0* [*simp*]: *Card* $\kappa \Longrightarrow 0 \oplus \kappa = \kappa$
by (*metis Card-def cadd-def cardinal-cong vsum-0-eqpoll*)

lemma *cadd-0-right* [*simp*]: *Card* $\kappa \Longrightarrow \kappa \oplus 0 = \kappa$
by (*simp add*: *cadd-commute*)

lemma *vsum-lepoll-self*: *elts* $a \lesssim$ *elts* ($a \uplus b$)
unfolding *elts-vsum* **by** (*meson Inl-iff Un-upper1 inj-onI lepoll-def*)

lemma *cadd-le-self*:
assumes κ: *Card* κ **shows** $\kappa \leq \kappa \oplus a$
proof (*unfold cadd-def*)
have $\kappa \leq$ *vcard* κ
using *Card-def* κ **by** *auto*
also have ... $\leq$ *vcard* ($\kappa \uplus a$)
by (*simp add*: *lepoll-imp-Card-le vsum-lepoll-self*)
finally show $\kappa \leq$ *vcard* ($\kappa \uplus a$) **.**
qed

Monotonicity of addition

lemma *cadd-le-mono*: $[\![\kappa' \leq \kappa;\ \mu' \leq \mu]\!] \Longrightarrow \kappa' \oplus \mu' \leq \kappa \oplus \mu$
unfolding *cadd-def*
by (*metis* (*no-types*) *lepoll-imp-Card-le less-eq-V-def subset-imp-lepoll sum-subset-iff*)

2.13 Cardinal multiplication

definition *cmult* :: $[V,V] \Rightarrow V$ (**infixl** $\langle\otimes\rangle$ *70*)
where $\kappa \otimes \mu \equiv$ *vcard* (*VSigma* κ ($\lambda z.\ \mu$))

2.13.1 Cardinal multiplication is commutative

lemma *prod-bij*: $[\![bij\text{-}betw\ f\ A\ C;\ bij\text{-}betw\ g\ B\ D]\!]$
$\Longrightarrow bij\text{-}betw\ (\lambda(x,\ y).\ (f\ x,\ g\ y))\ (A \times B)\ (C \times D)$
apply (*rule bij-betw-byWitness* [**where** $f' = \lambda(x,y).$ (*inv-into A f x*, *inv-into B g y*)])
apply (*auto simp*: *bij-betw-inv-into-left bij-betw-inv-into-right bij-betwE*)
using *bij-betwE bij-betw-inv-into* **apply** *blast+*
done

lemma *cmult-commute*: $i \otimes j = j \otimes i$
proof −
have $(\lambda(x,\ y).\ \langle x,\ y\rangle)$ ‘ $(elts\ i \times elts\ j) \approx (\lambda(x,\ y).\ \langle x,\ y\rangle)$ ‘ $(elts\ j \times elts\ i)$
by (*simp add*: *inj-on-vpair times-commute-eqpoll*)
then show *?thesis*
unfolding *cmult-def*
using *cardinal-cong elts-VSigma* **by** *auto*
qed

2.13.2 Cardinal multiplication is associative

lemma *elts-vcard-VSigma-eqpoll*: $elts\ (vcard\ (vtimes\ i\ j)) \approx elts\ i \times elts\ j$
proof −
have $elts\ (vtimes\ i\ j) \approx elts\ i \times elts\ j$
by (*simp add*: *elts-VSigma*)
then show *?thesis*
using *cardinal-eqpoll eqpoll-trans* **by** *blast*
qed

lemma *cmult-assoc*: $(i \otimes j) \otimes k = i \otimes (j \otimes k)$
unfolding *cmult-def*
proof (*rule cardinal-cong*)
have $elts\ (vcard\ (vtimes\ i\ j)) \times elts\ k \approx (elts\ i \times elts\ j) \times elts\ k$
by (*blast intro*: *times-eqpoll-cong elts-vcard-VSigma-eqpoll cardinal-eqpoll*)
also have $\ldots \approx elts\ i \times (elts\ j \times elts\ k)$
by (*rule times-assoc-eqpoll*)
also have $\ldots \approx elts\ i \times elts\ (vcard\ (vtimes\ j\ k))$
by (*blast intro*: *times-eqpoll-cong elts-vcard-VSigma-eqpoll cardinal-eqpoll eqpoll-sym*)
finally show $elts\ (VSigma\ (vcard\ (vtimes\ i\ j))\ (\lambda z.\ k)) \approx elts\ (VSigma\ i\ (\lambda z.\ vcard\ (vtimes\ j\ k)))$
by (*simp add*: *elts-VSigma*)
qed

2.13.3 Cardinal multiplication distributes over addition

lemma *cadd-cmult-distrib*: $(i \oplus j) \otimes k = (i \otimes k) \oplus (j \otimes k)$
unfolding *cadd-def cmult-def*
proof (*rule cardinal-cong*)
have $elts\ (vtimes\ (vcard\ (i \uplus j))\ k) \approx elts\ (vcard\ (vsum\ i\ j)) \times elts\ k$
using *cardinal-eqpoll elts-vcard-VSigma-eqpoll eqpoll-sym eqpoll-trans* **by** *blast*

also have ... $\approx$ (*Inl* ' *elts i* $\cup$ *Inr* ' *elts j*) $\times$ *elts k*
using *elts-vcard-vsum-eqpoll times-eqpoll-cong* **by** *blast*
also have ... $\approx$ (*Inl* ' *elts i*) $\times$ *elts k* $\cup$ (*Inr* ' *elts j*) $\times$ *elts k*
by (*simp add*: *Sigma-Un-distrib1*)
also have ... $\approx$ *elts* (*vtimes i k* $\uplus$ *vtimes j k*)
unfolding *Plus-def*
by (*auto simp*: *elts-vsum elts-VSigma disjnt-iff intro*!: *Un-eqpoll-cong times-eqpoll-cong*)
also have ... $\approx$ *elts* (*vcard* (*vtimes i k* $\uplus$ *vtimes j k*))
by (*simp add*: *cardinal-eqpoll eqpoll-sym*)
also have ... $\approx$ *elts* (*vcard* (*vtimes i k*) $\uplus$ *vcard* (*vtimes j k*))
by (*metis cadd-assoc cadd-def cardinal-cong cardinal-eqpoll vsum-0-eqpoll vsum-commute-eqpoll*)
finally show *elts* (*VSigma* (*vcard* (*i* $\uplus$ *j*)) ($\lambda z.$ *k*))
$\approx$ *elts* (*vcard* (*vtimes i k*) $\uplus$ *vcard* (*vtimes j k*)) .
qed

Multiplication by 0 yields 0

lemma *cmult-0* [*simp*]: *0* $\otimes$ *i* = *0*
using *Card-0 Card-def cmult-def* **by** *auto*

1 is the identity for multiplication

lemma *cmult-1* [*simp*]: **assumes** *Card* κ **shows** *1* $\otimes$ κ = κ
proof −
have *elts* (*vtimes* (*set* {*0*}) κ) $\approx$ *elts* κ
by (*auto simp*: *elts-VSigma intro*!: *times-singleton-eqpoll*)
then show *?thesis*
by (*metis Card-def assms cardinal-cong cmult-def elts-1 set-of-elts*)
qed

2.14 Some inequalities for multiplication

lemma *cmult-square-le*: **assumes** *Card* κ **shows** $\kappa \leq \kappa \otimes \kappa$
proof −
have *elts* $\kappa \lesssim$ *elts* ($\kappa \otimes \kappa$)
using *times-square-lepoll* [*of elts* κ] *cmult-def elts-vcard-VSigma-eqpoll eqpoll-sym lepoll-trans2*
by *fastforce*
then show *?thesis*
using *Card-def assms cmult-def lepoll-cardinal-le* **by** *fastforce*
qed

Multiplication by a non-empty set

lemma *cmult-le-self*: **assumes** *Card* κ $\alpha \neq$ *0* **shows** $\kappa \leq \kappa \otimes \alpha$
proof −
have κ = *vcard* κ
using *Card-def* ‹*Card* κ› **by** *blast*
also have ... $\leq$ *vcard* (*vtimes* κ α)
apply (*rule lepoll-imp-Card-le*)
apply (*simp add*: *elts-VSigma*)
by (*metis ZFC-in-HOL.ext* ‹$\alpha \neq$ *0*› *elts-0 lepoll-times1*)

also have $\ldots = \kappa \otimes \alpha$
by (*simp add*: *cmult-def*)
finally show *?thesis* **.**
qed

Monotonicity of multiplication

lemma *cmult-le-mono*: $[\![\kappa' \leq \kappa;\ \mu' \leq \mu]\!] \Longrightarrow \kappa' \otimes \mu' \leq \kappa \otimes \mu$
unfolding *cmult-def*
by (*auto simp*: *elts-VSigma intro*!: *lepoll-imp-Card-le times-lepoll-mono subset-imp-lepoll*)

2.15 The finite cardinals

lemma *succ-lepoll-succD*: *elts* (*succ*(*m*)) $\lesssim$ *elts* (*succ*(*n*)) $\Longrightarrow$ *elts m* $\lesssim$ *elts n*
by (*simp add*: *insert-lepoll-insertD*)

Congruence law for *succ* under equipollence

lemma *succ-eqpoll-cong*: *elts a* $\approx$ *elts b* $\Longrightarrow$ *elts* (*succ*(*a*)) $\approx$ *elts* (*succ*(*b*))
by (*simp add*: *succ-def insert-eqpoll-cong*)

lemma *sum-succ-eqpoll*: *elts* (*succ a* $\uplus$ *b*) $\approx$ *elts* (*succ*(*a*$\uplus$ *b*))
unfolding *eqpoll-def*
proof (*rule exI*)
let *?f* = $\lambda z.$ *if z=Inl a then a*$\uplus$ *b else z*
let *?g* = $\lambda z.$ *if z=a*$\uplus$ *b then Inl a else z*
show *bij-betw ?f* (*elts* (*succ a* $\uplus$ *b*)) (*elts* (*succ* (*a* $\uplus$ *b*)))
apply (*rule bij-betw-byWitness* [**where** *f'* = *?g*], *auto*)
apply (*metis Inl-in-sum-iff mem-not-refl*)
by (*metis Inr-in-sum-iff mem-not-refl*)
qed

lemma *cadd-succ*: *succ m* $\oplus$ *n* = *vcard* (*succ*(*m* $\oplus$ *n*))
proof (*unfold cadd-def*)
have [*intro*]: *elts* (*m* $\uplus$ *n*) $\approx$ *elts* (*vcard* (*m* $\uplus$ *n*))
using *cardinal-eqpoll eqpoll-sym* **by** *blast*
have *vcard* (*succ m* $\uplus$ *n*) = *vcard* (*succ*(*m* $\uplus$ *n*))
by (*rule sum-succ-eqpoll* [*THEN cardinal-cong*])
also have $\ldots$ = *vcard* (*succ*(*vcard* (*m* $\uplus$ *n*)))
by (*blast intro*: *succ-eqpoll-cong cardinal-cong*)
finally show *vcard* (*succ m* $\uplus$ *n*) = *vcard* (*succ*(*vcard* (*m* $\uplus$ *n*))) **.**
qed

lemma *nat-cadd-eq-add*: *ord-of-nat m* $\oplus$ *ord-of-nat n* = *ord-of-nat* (*m* + *n*)
proof (*induct m*)
case (*Suc m*) **thus** *?case*
by (*metis Card-def Card-ord-of-nat add-Suc cadd-succ ord-of-nat.simps*(*2*))
qed *auto*

lemma *vcard-disjoint-sup*:
assumes *x* $\sqcap$ *y* = *0* **shows** *vcard* (*x* $\sqcup$ *y*) = *vcard x* $\oplus$ *vcard y*

proof −
have *elts* $(x \sqcup y) \approx$ *elts* $(x \uplus y)$
unfolding *eqpoll-def*
proof (*rule exI*)
let *?f* $= \lambda z.$ *if* $z \in$ *elts x then Inl z else Inr z*
let *?g* $=$ *sum-case id id*
show *bij-betw ?f* (*elts* $(x \sqcup y)$) (*elts* $(x \uplus y)$)
by (*rule bij-betw-byWitness* [**where** $f' =$ *?g*]) (*use assms V-disjoint-iff* **in** *auto*)
qed
then show *?thesis*
by (*metis cadd-commute cadd-def cardinal-cong cardinal-idem vsum-0-eqpoll cadd-assoc*)
qed

2.16 Infinite cardinals

definition *InfCard* :: $V \Rightarrow$*bool*
where *InfCard* $\kappa \equiv$ *Card* $\kappa \wedge \omega \leq \kappa$

lemma *InfCard-iff*: *InfCard* $\kappa \longleftrightarrow$ *Card* $\kappa \wedge$ *infinite* (*elts* κ)
proof (*cases* $\omega \leq \kappa$)
case *True*
then show *?thesis*
using *inj-ord-of-nat lepoll-def less-eq-V-def*
by (*auto simp*: *InfCard-def ω-def infinite-le-lepoll*)
next
case *False*
then show *?thesis*
using *Card-iff-initial InfCard-def infinite-Ord-omega* **by** *blast*
qed

lemma *InfCard-ge-ord-of-nat*:
assumes *InfCard* κ **shows** *ord-of-nat* $n \leq \kappa$
using *InfCard-def assms ord-of-nat-le-omega* **by** *blast*

lemma *InfCard-not-0*[*iff*]: $\neg$ *InfCard 0*
by (*simp add*: *InfCard-iff*)

definition
csucc :: $V \Rightarrow V$
where *csucc* $\kappa \equiv$ *LEAST* κ'. *Ord* $\kappa' \wedge$ (*Card* $\kappa' \wedge \kappa < \kappa'$)

lemma *less-vcard-VPow*: *vcard A* $<$ *vcard* (*VPow A*)
proof (*rule lesspoll-imp-Card-less*)
show *elts A* $\prec$ *elts* (*VPow A*)
by (*simp add*: *elts-VPow down inj-on-def lesspoll-Pow-self*)
qed

lemma *greater-Card*:
assumes *Card* κ **shows** $\kappa <$ *vcard* (*VPow* κ)
proof −
have $\kappa =$ *vcard* κ
using *Card-def assms* **by** *blast*
also have $\ldots <$ *vcard* (*VPow* κ)
proof (*rule lesspoll-imp-Card-less*)
show *elts* $\kappa \prec$ *elts* (*VPow* κ)
by (*simp add*: *elts-VPow down inj-on-def lesspoll-Pow-self*)
qed
finally show *?thesis* **.**
qed

lemma
assumes *Card* κ
shows *Card-csucc* [*simp*]: *Card* (*csucc* κ) **and** *less-csucc* [*simp*]: $\kappa <$ *csucc* κ
proof −
have *Card* (*csucc* κ) $\wedge$ $\kappa <$ *csucc* κ
unfolding *csucc-def*
proof (*rule Ord-LeastI2*)
show *Card* (*vcard* (*VPow* κ)) $\wedge$ $\kappa <$ (*vcard* (*VPow* κ))
using *Card-def assms greater-Card* **by** *auto*
qed *auto*
then show *Card* (*csucc* κ) $\kappa <$ *csucc* κ
by *auto*
qed

lemma *le-csucc*:
assumes *Card* κ **shows** $\kappa \leq$ *csucc* κ
by (*simp add*: *assms less-csucc less-imp-le*)

lemma *csucc-le*: $[\![$*Card* μ; $\kappa \in$ *elts* $\mu]\!] \Longrightarrow$ *csucc* $\kappa \leq \mu$
unfolding *csucc-def*
by (*simp add*: *Card-is-Ord Ord-Least-le OrdmemD*)

lemma *finite-csucc*: $a \in$ *elts* $\omega \Longrightarrow$ *csucc* $a =$ *succ* a
unfolding *csucc-def*
proof (*rule Least-equality*)
show *Ord* (*ZFC-in-HOL.succ a*) $\wedge$ *Card* (*ZFC-in-HOL.succ a*) $\wedge$ $a <$ *ZFC-in-HOL.succ a*
if $a \in$ *elts* ω
using *that* **by** (*auto simp*: *less-V-def less-eq-V-def nat-into-Card*)
show *ZFC-in-HOL.succ* $a \leq y$
if $a \in$ *elts* ω
and *Ord* $y \wedge$ *Card* $y \wedge$ $a < y$
for $y :: V$
using *that*

using *Ord-mem-iff-lt dual-order.strict-implies-order* **by** *fastforce*
qed

lemma *Finite-imp-cardinal-cons* [*simp*]:
assumes *FA*: *finite A* **and** *a*: $a \notin A$
shows *vcard* (*set* (*insert a A*)) = *csucc*(*vcard* (*set A*))
proof −
show *?thesis*
unfolding *csucc-def*
proof (*rule Least-equality* [*THEN sym*])
have *small A*
by (*simp add*: *FA Finite-V*)
then have ¬ *elts* (*set A*) ≈ *elts* (*set* (*insert a A*))
using *FA a eqpoll-imp-lepoll eqpoll-sym finite-insert-lepoll* **by** *fastforce*
then show *Ord* (*vcard* (*set* (*insert a A*))) ∧ *Card* (*vcard* (*set* (*insert a A*))) ∧ *vcard* (*set A*) < *vcard* (*set* (*insert a A*))
by (*simp add*: *Card-def lesspoll-imp-Card-less lesspoll-def subset-imp-lepoll subset-insertI*)
show *vcard* (*set* (*insert a A*)) ≤ *i*
if *Ord i* ∧ *Card i* ∧ *vcard* (*set A*) < *i* **for** *i*
proof −
have *elts* (*vcard* (*set A*)) ≈ *A*
by (*metis FA finite-imp-small cardinal-eqpoll elts-of-set*)
then have *less*: *A* ≺ *elts i*
using *eq-lesspoll-trans eqpoll-sym lt-Card-imp-lesspoll that* **by** *blast*
show *?thesis*
using *that less* **by** (*auto simp*: *less-imp-insert-lepoll lepoll-cardinal-le*)
qed
qed
qed

lemma *vcard-finite-set*: *finite A* ⟹ *vcard* (*set A*) = *ord-of-nat* (*card A*)
by (*induction A rule*: *finite-induct*) (*auto simp*: *set-empty ω-def finite-csucc*)

lemma *lt-csucc-iff*:
assumes *Ord α Card κ*
shows $\alpha < csucc\ \kappa \longleftrightarrow vcard\ \alpha \leq \kappa$
proof
show *vcard* $\alpha \leq \kappa$ **if** $\alpha < csucc\ \kappa$
proof −
have *vcard* $\alpha \leq csucc\ \kappa$
by (*meson* ‹*Ord α*› *dual-order.trans lepoll-cardinal-le lepoll-refl less-le-not-le that*)
then show *?thesis*
by (*metis* (*no-types*) *Card-def Card-iff-initial Ord-linear2 Ord-mem-iff-lt assms cardinal-eqpoll cardinal-idem csucc-le eq-iff eqpoll-sym that*)
qed
show $\alpha < csucc\ \kappa$ **if** *vcard* $\alpha \leq \kappa$
proof −

have $\neg$ *csucc* $\kappa \leq \alpha$
using *that*
by (*metis Card-csucc Card-def assms*(*2*) *le-less-trans lepoll-imp-Card-le less-csucc less-eq-V-def less-le-not-le subset-imp-lepoll*)
then show *?thesis*
by (*meson Card-csucc Card-is-Ord Ord-linear2 assms*)
qed
qed

lemma *Card-lt-csucc-iff*: $[\![Card\ \kappa'; Card\ \kappa]\!] \Longrightarrow (\kappa' < csucc\ \kappa) = (\kappa' \leq \kappa)$
by (*simp add*: *lt-csucc-iff Card-cardinal-eq Card-is-Ord*)

lemma *InfCard-csucc*: *InfCard* $\kappa \Longrightarrow$ *InfCard* (*csucc* κ)
using *InfCard-def le-csucc* **by** *auto*

Kunen's Lemma 10.11

lemma *InfCard-is-Limit*:
assumes *InfCard* κ **shows** *Limit* κ
proof (*rule non-succ-LimitI*)
show $\kappa \neq 0$
using *InfCard-def assms mem-not-refl* **by** *blast*
show *Ord* κ
using *Card-is-Ord InfCard-def assms* **by** *blast*
show *ZFC-in-HOL.succ y* $\neq \kappa$ **for** *y*
proof
assume *succ y* $= \kappa$
then have *Card* (*succ y*)
using *InfCard-def assms* **by** *auto*
moreover have $\omega \leq y$
by (*metis InfCard-iff Ord-in-Ord* ‹*Ord* κ› ‹*ZFC-in-HOL.succ y* $= \kappa$› *assms elts-succ finite-insert infinite-Ord-omega insertI1*)
moreover have *elts y* $\approx$ *elts* (*succ y*)
using *InfCard-iff* ‹*ZFC-in-HOL.succ y* $= \kappa$› *assms eqpoll-sym infinite-insert-eqpoll* **by** *fastforce*
ultimately show *False*
by (*metis Card-iff-initial Ord-in-Ord OrdmemD elts-succ insertI1*)
qed
qed

2.17 Toward's Kunen's Corollary 10.13 (1)

Kunen's Theorem 10.12

lemma *InfCard-csquare-eq*:
assumes *InfCard*(κ) **shows** $\kappa \otimes \kappa = \kappa$
using *infinite-times-eqpoll-self* [*of elts* κ] *assms*
unfolding *InfCard-iff Card-def*
by (*metis cardinal-cong cardinal-eqpoll cmult-def elts-vcard-VSigma-eqpoll eqpoll-trans*)

lemma *InfCard-le-cmult-eq*:

assumes *InfCard* κ $\mu \leq \kappa$ $\mu \neq 0$
shows $\kappa \otimes \mu = \kappa$
proof (*rule order-antisym*)
have $\kappa \otimes \mu \leq \kappa \otimes \kappa$
by (*simp add*: *assms*(*2*) *cmult-le-mono*)
also have $\ldots \leq \kappa$
by (*simp add*: *InfCard-csquare-eq assms*(*1*))
finally show $\kappa \otimes \mu \leq \kappa$ **.**
show $\kappa \leq \kappa \otimes \mu$
using *InfCard-def assms*(*1*) *assms*(*3*) *cmult-le-self* **by** *auto*
qed

Kunen's Corollary 10.13 (1), for cardinal multiplication

lemma *InfCard-cmult-eq*: $[\![\mathit{InfCard}\ \kappa; \mathit{InfCard}\ \mu]\!] \Longrightarrow \kappa \otimes \mu = \kappa \sqcup \mu$
by (*metis Card-is-Ord InfCard-def InfCard-le-cmult-eq Ord-linear-le cmult-commute inf-sup-aci*(*5*) *mem-not-refl sup.orderE sup-V-0-right zero-in-omega*)

lemma *cmult-succ*:
$\mathit{succ}(m) \otimes n = n \oplus (m \otimes n)$
unfolding *cmult-def cadd-def*
proof (*rule cardinal-cong*)
have *elts* (*vtimes* (*ZFC-in-HOL.succ m*) *n*) $\approx$ *elts n* <+> *elts m* $\times$ *elts n*
by (*simp add*: *elts-VSigma prod-insert-eqpoll*)
also have $\ldots \approx$ *elts* (*n* $\biguplus$ *vcard* (*vtimes m n*))
unfolding *elts-VSigma elts-vsum Plus-def*
proof (*rule Un-eqpoll-cong*)
show (*Sum-Type.Inr* ' (*elts m* $\times$ *elts n*)::(*V* + *V* $\times$ *V*) *set*) $\approx$ *Inr* ' *elts* (*vcard* (*vtimes m n*))
by (*simp add*: *elts-vcard-VSigma-eqpoll eqpoll-sym*)
qed (*auto simp*: *disjnt-def*)
finally show *elts* (*vtimes* (*ZFC-in-HOL.succ m*) *n*) $\approx$ *elts* (*n* $\biguplus$ *vcard* (*vtimes m n*)) **.**
qed

lemma *cmult-2*:
assumes *Card n* **shows** *ord-of-nat 2* $\otimes$ *n* = *n* $\oplus$ *n*
proof −
have *ord-of-nat 2* = *succ* (*succ 0*)
by *force*
then show *?thesis*
by (*simp add*: *cmult-succ assms*)
qed

lemma *InfCard-cdouble-eq*:
assumes *InfCard* κ **shows** $\kappa \oplus \kappa = \kappa$
proof −
have $\kappa \oplus \kappa = \kappa \otimes$ *ord-of-nat 2*
using *InfCard-def assms cmult-2 cmult-commute* **by** *auto*
also have $\ldots = \kappa$

by (*simp add*: *InfCard-le-cmult-eq InfCard-ge-ord-of-nat assms*)
finally show *?thesis* **.**
qed

Corollary 10.13 (1), for cardinal addition

lemma *InfCard-le-cadd-eq*: $[\![InfCard\ \kappa;\ \mu \leq \kappa]\!] \Longrightarrow \kappa \oplus \mu = \kappa$
by (*metis InfCard-cdouble-eq InfCard-def antisym cadd-le-mono cadd-le-self*)

lemma *InfCard-cadd-eq*: $[\![InfCard\ \kappa;\ InfCard\ \mu]\!] \Longrightarrow \kappa \oplus \mu = \kappa \sqcup \mu$
by (*metis Card-iff-initial InfCard-def InfCard-le-cadd-eq Ord-linear-le cadd-commute sup.absorb2 sup.orderE*)

2.18 The Aleph-seqence

This is the well-known transfinite enumeration of the cardinal numbers.

definition
$Aleph :: V \Rightarrow V \quad (\langle\aleph\text{-}\rangle\ [90]\ 90)$
where $Aleph \equiv transrec3\ \omega\ (\lambda x\ r.\ csucc(r))\ (\lambda i\ r\ .\ \bigsqcup\ (r\ `\ elts\ i))$

lemma *Card-Aleph* [*simp*, *intro*]:
$Ord\ \alpha \Longrightarrow Card(Aleph\ \alpha)$
by (*induction* α *rule*: *Ord-induct3*) (*auto simp*: *Aleph-def*)

lemma *Aleph-0* [*simp*]: $\aleph 0 = \omega$
by (*simp add*: *Aleph-def*)

lemma *Aleph-succ* [*simp*]: $\aleph(succ\ x) = csucc\ (\aleph\ x)$
by (*simp add*: *Aleph-def*)

lemma *Aleph-Limit*: $Limit\ \gamma \Longrightarrow \aleph\ \gamma = \bigsqcup\ (Aleph\ `\ elts\ \gamma)$
by (*simp add*: *Aleph-def*)

lemma *mem-Aleph-succ*: $Ord\ \alpha \Longrightarrow \aleph(\alpha) \in elts\ (\aleph(succ\ \alpha))$
by (*simp add*: *Card-is-Ord Ord-mem-iff-lt*)

lemma *Aleph-increasing*:
assumes *ab*: $\alpha < \beta\ Ord\ \alpha\ Ord\ \beta$ **shows** $Aleph(\alpha) < Aleph(\beta)$
proof −
{ fix x
have $[\![Ord\ x;\ x \in elts\ \beta]\!] \Longrightarrow Aleph(x) \in elts\ (Aleph\ \beta)$
using $\langle Ord\ \beta\rangle$
proof (*induct* β *arbitrary*: x *rule*: *Ord-induct3*)
case *0* **thus** *?case* **by** *simp*
next
case (*succ* β)
then consider $x = \beta\ |x \in elts\ \beta$
using *OrdmemD* **by** *auto*
then show *?case*
proof *cases*

```
      case 1
      then show ?thesis
        by (simp add: Card-is-Ord Ord-mem-iff-lt succ.hyps(1))
    next
      case 2
      with succ show ?thesis
        by (metis Aleph-succ Card-Aleph le-csucc vsubsetD)
    qed
  next
    case (Limit γ)
    hence sc: succ x ∈ elts γ
      by (simp add: Limit-def Ord-mem-iff-lt)
    hence ℵ x ∈ elts (⨆ (Aleph ‘ elts γ))
      using Limit
      by blast
    thus ?case using Limit
      by (simp add: Aleph-Limit)
  qed
 } thus ?thesis using ab
  by (simp add: Card-is-Ord Ord-mem-iff-lt)
qed

lemma countable-iff-le-Aleph0: countable (elts A) ⟷ vcard A ≤ ℵ0
proof
 show vcard A ≤ ℵ0
  if countable (elts A)
 proof (cases finite (elts A))
  case True
  then show ?thesis
   using vcard-finite-set by fastforce
 next
  case False
  then have elts ω ≈ elts A
   using countableE-infinite [OF that]
   by (simp add: eqpoll-def ω-def) (meson bij-betw-def bij-betw-inv bij-betw-trans
inj-ord-of-nat)
  then show ?thesis
   using Card-ω Card-def cardinal-cong vcard-def by auto
 qed
 show countable (elts A)
  if vcard A ≤ ℵ0
 proof −
  have elts A ≲ elts ω
   using cardinal-le-lepoll [OF that] by simp
  then show ?thesis
   by (simp add: countable-iff-lepoll ω-def inj-ord-of-nat)
 qed
qed
```

2.19 The ordinal $\omega 1$

abbreviation $\omega 1 \equiv Aleph\ 1$

lemma *Ord-ω1* [*simp*]: *Ord ω1*
by (*simp add*: *Card-is-Ord*)

lemma *omega-ω1* [*iff*]: $\omega \in elts\ \omega 1$
using *mem-Aleph-succ one-V-def* **by** *fastforce*

lemma *ord-of-nat-ω1* [*iff*]: *ord-of-nat n ∈ elts ω1*
using *Ord-ω1 Ord-trans* **by** *blast*

lemma *countable-iff-less-ω1*:
assumes *Ord α*
shows *countable* (*elts α*) $\longleftrightarrow \alpha < \omega 1$
by (*simp add*: *assms countable-iff-le-Aleph0 lt-csucc-iff one-V-def*)

lemma *less-ω1-imp-countable*:
assumes $\alpha \in elts\ \omega 1$
shows *countable* (*elts α*)
using *Ord-ω1 Ord-in-Ord OrdmemD assms countable-iff-less-ω1* **by** *blast*

lemma *ω1-gt0* [*simp*]: $\omega 1 > 0$
using *Ord-ω1 Ord-trans OrdmemD* **by** *blast*

lemma *ω1-gt1* [*simp*]: $\omega 1 > 1$
using *Ord-ω1 OrdmemD ω-gt1 less-trans* **by** *blast*

lemma *Limit-ω1* [*simp*]: *Limit ω1*
by (*simp add*: *InfCard-def InfCard-is-Limit le-csucc one-V-def*)

end

3 Addition and Multiplication of Sets

theory *Kirby*
imports *ZFC-Cardinals*

begin

3.1 Generalised Addition

Source: Laurence Kirby, Addition and multiplication of sets Math. Log. Quart. 53, No. 1, 52-65 (2007) / DOI 10.1002/malq.200610026 http://faculty.baruch.cuny.edu/lkirby/mlqarticlejan2007.pdf

3.1.1 Addition is a monoid

instantiation $V :: plus$
begin

This definition is credited to Tarski

definition $plus\text{-}V :: V \Rightarrow V \Rightarrow V$
  where $plus\text{-}V\ x \equiv transrec\ (\lambda f\ z.\ x \sqcup set\ (f\ `\ elts\ z))$

instance ..
end

definition $lift :: V \Rightarrow V \Rightarrow V$
  where $lift\ x\ y \equiv set\ (plus\ x\ `\ elts\ y)$

lemma *plus*: $x + y = x \sqcup set\ ((+)x\ `\ elts\ y)$
  unfolding *plus-V-def* **by** (*subst transrec*) *auto*

lemma *plus-eq-lift*: $x + y = x \sqcup lift\ x\ y$
  unfolding *lift-def* **using** *plus* **by** *blast*

Lemma 3.2

lemma *lift-sup-distrib*: $lift\ x\ (a \sqcup b) = lift\ x\ a \sqcup lift\ x\ b$
  by (*simp add*: *image-Un lift-def sup-V-def*)

lemma *lift-Sup-distrib*: $small\ Y \Longrightarrow lift\ x\ (\bigsqcup\ Y) = \bigsqcup\ (lift\ x\ `\ Y)$
  by (*auto simp*: *lift-def Sup-V-def image-Union*)

lemma *add-Sup-distrib*:
  fixes $x::V$ **shows** $y \neq 0 \Longrightarrow x + (SUP\ z{\in}elts\ y.\ f\ z) = (SUP\ z{\in}elts\ y.\ x + f\ z)$
  by (*auto simp*: *plus-eq-lift SUP-sup-distrib lift-Sup-distrib image-image*)

lemma *Limit-add-Sup-distrib*:
  fixes $x::V$ **shows** $Limit\ \alpha \Longrightarrow x + (SUP\ z{\in}elts\ \alpha.\ f\ z) = (SUP\ z{\in}elts\ \alpha.\ x + f\ z)$
  using *add-Sup-distrib* **by** *force*

Proposition 3.3(ii)

instantiation $V :: monoid\text{-}add$
begin
instance
proof
  show $a + b + c = a + (b + c)$ **for** $a\ b\ c :: V$
  proof (*induction c rule*: *eps-induct*)
    case (*step c*)
    have $(a{+}b) + c = a + b \sqcup set\ ((+)\ (a + b)\ `\ elts\ c)$
      by (*metis plus*)
    also have $\ldots = a \sqcup lift\ a\ b \sqcup set\ ((\lambda u.\ a + (b{+}u))\ `\ elts\ c)$
      using *plus-eq-lift step.IH* **by** *auto*

```
    also have ... = a ⊔ lift a (b + c)
    proof −
      have lift a b ⊔ set ((λu. a + (b + u)) ‘ elts c) = lift a (b + c)
        unfolding lift-def
        by (metis elts-of-set image-image lift-def lift-sup-distrib plus-eq-lift replace-
ment small-elts)
      then show ?thesis
        by (simp add: sup-assoc)
    qed
    also have ... = a + (b + c)
      using plus-eq-lift by auto
    finally show ?case .
  qed
  show 0 + x = x for x :: V
  proof (induction rule: eps-induct)
    case (step x)
    then show ?case
      by (subst plus) auto
  qed
  show x + 0 = x for x :: V
    by (subst plus) auto
qed
end

lemma lift-0 [simp]: lift 0 x = x
  by (simp add: lift-def)

lemma lift-by0 [simp]: lift x 0 = 0
  by (simp add: lift-def)

lemma lift-by1 [simp]: lift x 1 = set{x}
  by (simp add: lift-def)

lemma add-eq-0-iff [simp]:
  fixes x y::V
  shows x+y = 0 ⟷ x=0 ∧ y=0
  proof safe
  show x = 0 if x + y = 0
    by (metis that le-imp-less-or-eq not-less-0 plus sup-ge1)
  then show y = 0 if x + y = 0
    using that by auto
qed auto

lemma plus-vinsert: x + vinsert z y = vinsert (x+z) (x + y)
proof −
  have f1: elts (x + y) = elts x ∪ (+) x ‘ elts y
   by (metis elts-of-set lift-def plus-eq-lift replacement small-Un small-elts sup-V-def)
  moreover have lift x (vinsert z y) = set ((+) x ‘ elts (set (insert z (elts y))))
    using vinsert-def lift-def by presburger
```

ultimately show *?thesis*
by (*simp add*: *vinsert-def plus-eq-lift sup-V-def*)
qed

lemma *plus-V-succ-right*: $x + succ\ y = succ\ (x + y)$
by (*metis plus-vinsert succ-def*)

lemma *succ-eq-add1*: $succ\ x = x + 1$
by (*simp add*: *plus-V-succ-right one-V-def*)

lemma *ord-of-nat-add*: $ord\text{-}of\text{-}nat\ (m+n) = ord\text{-}of\text{-}nat\ m + ord\text{-}of\text{-}nat\ n$
by (*induction n*) (*auto simp*: *plus-V-succ-right*)

lemma *succ-0-plus-eq* [*simp*]:
assumes $\alpha \in elts\ \omega$
shows $succ\ 0 + \alpha = succ\ \alpha$
proof −
obtain *n* **where** $\alpha = ord\text{-}of\text{-}nat\ n$
using *assms elts-ω* **by** *blast*
then show *?thesis*
by (*metis One-nat-def ord-of-nat.simps ord-of-nat-add plus-1-eq-Suc*)
qed

lemma *omega-closed-add* [*intro*]:
assumes $\alpha \in elts\ \omega\ \beta \in elts\ \omega$ **shows** $\alpha+\beta \in elts\ \omega$
proof −
obtain *m n* **where** $\alpha = ord\text{-}of\text{-}nat\ m\ \beta = ord\text{-}of\text{-}nat\ n$
using *assms elts-ω* **by** *auto*
then have $\alpha+\beta = ord\text{-}of\text{-}nat\ (m+n)$
using *ord-of-nat-add* **by** *auto*
then show *?thesis*
by (*simp add*: *ω-def*)
qed

lemma *mem-plus-V-E*:
assumes *l*: $l \in elts\ (x + y)$
obtains $l \in elts\ x \mid z$ **where** $z \in elts\ y\ l = x + z$
using *l* **by** (*auto simp*: *plus* [*of x y*] *split*: *if-split-asm*)

lemma *not-add-less-right*: **assumes** *Ord y* **shows** $\neg\ (x + y < x)$
using *assms*
proof (*induction rule*: *Ord-induct*)
case (*step i*)
then show *?case*
by (*metis less-le-not-le plus sup-ge1*)
qed

lemma *not-add-mem-right*: $\neg\ (x + y \in elts\ x)$
by (*metis sup-ge1 mem-not-refl plus vsubsetD*)

Proposition 3.3(iii)

lemma *add-not-less-TC-self*: $\neg\ x + y \sqsubset x$
proof (*induction y arbitrary*: *x rule*: *eps-induct*)
  case (*step y*)
  then show *?case*
    using *less-TC-imp-not-le plus-eq-lift* **by** *fastforce*
qed

lemma *TC-sup-lift*: $TC\ x \sqcap lift\ x\ y = 0$
proof −
  have $elts\ (TC\ x) \cap elts\ (set\ ((+)\ x\ `\ elts\ y)) = \{\}$
    using *add-not-less-TC-self* **by** (*auto simp*: *less-TC-def*)
  then have $TC\ x \sqcap set\ ((+)\ x\ `\ elts\ y) = set\ \{\}$
    by (*metis inf-V-def*)
  then show *?thesis*
    using *lift-def* **by** *auto*
qed

lemma *lift-lift*: $lift\ x\ (lift\ y\ z) = lift\ (x+y)\ z$
  using *add.assoc* **by** (*auto simp*: *lift-def*)

lemma *lift-self-disjoint*: $x \sqcap lift\ x\ u = 0$
 by (*metis TC-sup-lift arg-subset-TC inf.absorb-iff2 inf-assoc inf-sup-aci*(*3*) *lift-0*)

lemma *sup-lift-eq-lift*:
  assumes $x \sqcup lift\ x\ u = x \sqcup lift\ x\ v$
  shows $lift\ x\ u = lift\ x\ v$
 by (*metis* (*no-types*) *assms inf-sup-absorb inf-sup-distrib2 lift-self-disjoint sup-commute sup-inf-absorb*)

3.1.2 Deeper properties of addition

Proposition 3.4(i)

proposition *lift-eq-lift*: $lift\ x\ y = lift\ x\ z \Longrightarrow y = z$
proof (*induction y arbitrary*: *z rule*: *eps-induct*)
  case (*step y*)
  show *?case*
  proof (*intro vsubsetI order-antisym*)
    show $u \in elts\ z$ **if** $u \in elts\ y$ **for** *u*
    proof −
      have $x+u \in elts\ (lift\ x\ z)$
        using *lift-def step.prems that* **by** *fastforce*
      then obtain *v* **where** $v \in elts\ z\ x+u = x+v$
        using *lift-def* **by** *auto*
      then have $lift\ x\ u = lift\ x\ v$
        using *sup-lift-eq-lift* **by** (*simp add*: *plus-eq-lift*)
      then have $u=v$
        using *step.IH that* **by** *blast*

```
      then show ?thesis
        using ‹v ∈ elts z› by blast
    qed
    show u ∈ elts y if u ∈ elts z for u
    proof −
      have x+u ∈ elts (lift x y)
        using lift-def step.prems that by fastforce
      then obtain v where v ∈ elts y x+u = x+v
        using lift-def by auto
      then have lift x u = lift x v
        using sup-lift-eq-lift by (simp add: plus-eq-lift)
      then have u=v
        using step.IH by (metis ‹v ∈ elts y›)
      then show ?thesis
        using ‹v ∈ elts y› by auto
    qed
  qed
qed

corollary inj-lift: inj-on (lift x) A
  by (auto simp: inj-on-def dest: lift-eq-lift)

corollary add-right-cancel [iff]:
  fixes x y z::V shows x+y = x+z ⟷ y=z
  by (metis lift-eq-lift plus-eq-lift sup-lift-eq-lift)

corollary add-mem-right-cancel [iff]:
  fixes x y z::V shows x+y ∈ elts (x+z) ⟷ y ∈ elts z
  apply safe
   apply (metis mem-plus-V-E not-add-mem-right add-right-cancel)
  by (metis ZFC-in-HOL.ext dual-order.antisym elts-vinsert insert-subset order-refl
plus-vinsert)

corollary add-le-cancel-left [iff]:
  fixes x y z::V shows x+y ≤ x+z ⟷ y≤z
  by auto (metis add-mem-right-cancel mem-plus-V-E plus sup-ge1 vsubsetD)

corollary add-less-cancel-left [iff]:
  fixes x y z::V shows x+y < x+z ⟷ y<z
  by (simp add: less-le-not-le)

corollary lift-le-self [simp]: lift x y ≤ x ⟷ y = 0
  by (auto simp: inf.absorb-iff2 lift-eq-lift lift-self-disjoint)

lemma succ-less-ω-imp: succ x < ω ⟹ x < ω
  by (metis add-le-cancel-left add.right-neutral le-0 le-less-trans succ-eq-add1)
```

Proposition 3.5

```
lemma card-lift: vcard (lift x y) = vcard y
```

proof (*rule cardinal-cong*)
have *bij-betw* ((+)*x*) (*elts y*) (*elts* (*lift x y*))
unfolding *bij-betw-def*
by (*simp add*: *inj-on-def lift-def*)
then show *elts* (*lift x y*) ≈ *elts y*
using *eqpoll-def eqpoll-sym* **by** *blast*
qed

lemma *eqpoll-lift*: *elts* (*lift x y*) ≈ *elts y*
by (*metis card-lift cardinal-eqpoll eqpoll-sym eqpoll-trans*)

lemma *vcard-add*: *vcard* (*x* + *y*) = *vcard x* ⊕ *vcard y*
using *card-lift* [*of x y*] *lift-self-disjoint* [*of x*]
by (*simp add*: *plus-eq-lift vcard-disjoint-sup*)

lemma *countable-add*:
assumes *countable* (*elts A*) *countable* (*elts B*)
shows *countable* (*elts* (*A*+*B*))
proof −
have *vcard A* ≤ ℵ*0 vcard B* ≤ ℵ*0*
using *assms countable-iff-le-Aleph0* **by** *blast*+
then have *vcard* (*A*+*B*) ≤ ℵ*0*
unfolding *vcard-add*
by (*metis Aleph-0 Card-ω InfCard-cdouble-eq InfCard-def cadd-le-mono order-refl*)
then show *?thesis*
by (*simp add*: *countable-iff-le-Aleph0*)
qed

Proposition 3.6

proposition *TC-add*: *TC* (*x* + *y*) = *TC x* ⊔ *lift x* (*TC y*)
proof (*induction y rule*: *eps-induct*)
case (*step y*)
have ∗: ⨆ (*TC* ‘ (+) *x* ‘ *elts y*) = *TC x* ⊔ (*SUP u*∈*elts y*. *TC* (*set* ((+) *x* ‘ *elts u*)))
if *elts y* ≠ {}
proof −
obtain *w* **where** *w* ∈ *elts y*
using ‹*elts y* ≠ {}› **by** *blast*
then have *TC x* ≤ *TC* (*x* + *w*)
by (*simp add*: *step.IH*)
then have †: *TC x* ≤ (*SUP w*∈*elts y*. *TC* (*x* + *w*))
using ‹*w* ∈ *elts y*› **by** *blast*
show *?thesis*
using *that*
apply (*intro conjI ballI impI order-antisym*; *clarsimp simp add*: *image-comp* †)
apply(*metis TC-sup-distrib Un-iff elts-sup-iff plus*)
by (*metis TC-least Transset-TC arg-subset-TC le-sup-iff plus vsubsetD*)

qed
have *TC* (*x* + *y*) = (*x* + *y*) ⊔ ⨆ (*TC* ‘ *elts* (*x* + *y*))
using *TC* **by** *blast*
also have ... = *x* ⊔ *lift x y* ⊔ ⨆ (*TC* ‘ *elts x*) ⊔ ⨆ ((λ*u*. *TC* (*x*+*u*)) ‘ *elts y*)
apply (*simp add*: *plus-eq-lift image-Un Sup-Un-distrib sup.left-commute sup-assoc TC-sup-distrib SUP-sup-distrib*)
apply (*simp add*: *lift-def sup.commute sup-aci* ∗)
done
also have ... = *x* ⊔ ⨆ (*TC* ‘ *elts x*) ⊔ *lift x y* ⊔ ⨆ ((λ*u*. *TC x* ⊔ *lift x* (*TC u*)) ‘ *elts y*)
by (*simp add*: *sup-aci step.IH*)
also have ... = *TC x* ⊔ *lift x y* ⊔ ⨆ ((λ*u*. *lift x* (*TC u*)) ‘ *elts y*)
by (*simp add*: *sup-aci SUP-sup-distrib flip*: *TC* [*of x*])
also have ... = *TC x* ⊔ *lift x* (*y* ⊔ ⨆ (*TC* ‘ *elts y*))
by (*metis* (*no-types*) *elts-of-set lift-Sup-distrib image-image lift-sup-distrib replacement small-elts sup-assoc*)
also have ... = *TC x* ⊔ *lift x* (*TC y*)
by (*simp add*: *TC* [*of y*])
finally show *?case* **.**
qed

corollary *TC-add′*: *z* ⊏ *x* + *y* ⟷ *z* ⊏ *x* ∨ (∃ *v*. *v* ⊏ *y* ∧ *z* = *x* + *v*)
using *TC-add* **by** (*force simp*: *less-TC-def lift-def*)

Corollary 3.7

corollary *vcard-TC-add*: *vcard* (*TC* (*x*+*y*)) = *vcard* (*TC x*) ⊕ *vcard* (*TC y*)
by (*simp add*: *TC-add TC-sup-lift card-lift vcard-disjoint-sup*)

Corollary 3.8

corollary *TC-lift*:
assumes *y* ≠ *0*
shows *TC* (*lift x y*) = *TC x* ⊔ *lift x* (*TC y*)
proof −
have *TC* (*lift x y*) = *lift x y* ⊔ ⨆ ((λ*u*. *TC*(*x*+*u*)) ‘ *elts y*)
unfolding *TC* [*of lift x y*] **by** (*simp add*: *lift-def image-image*)
also have ... = *lift x y* ⊔ (*SUP u*∈*elts y*. *TC x* ⊔ *lift x* (*TC u*))
by (*simp add*: *TC-add*)
also have ... = *lift x y* ⊔ *TC x* ⊔ (*SUP u*∈*elts y*. *lift x* (*TC u*))
using *assms* **by** (*auto simp*: *SUP-sup-distrib*)
also have ... = *TC x* ⊔ *lift x* (*TC y*)
by (*simp add*: *TC* [*of y*] *sup-aci image-image lift-sup-distrib lift-Sup-distrib*)
finally show *?thesis* **.**
qed

proposition *rank-add-distrib*: *rank* (*x*+*y*) = *rank x* + *rank y*
proof (*induction y rule*: *eps-induct*)
case (*step y*)
show *?case*
proof (*cases y*=*0*)

```
    case False
    then obtain e where e: e ∈ elts y
      by fastforce
    have rank (x+y) = (SUP u∈elts (x ⊔ ZFC-in-HOL.set ((+) x ‘ elts y)). succ
(rank u))
      by (metis plus rank-Sup)
    also have ... = (SUP x∈elts x. succ (rank x)) ⊔ (SUP z∈elts y. succ (rank x
+ rank z))
      apply (simp add: Sup-Un-distrib image-Un image-image)
      apply (simp add: step cong: SUP-cong-simp)
      done
    also have ... = (SUP z ∈ elts y. rank x + succ (rank z))
    proof −
      have rank x ≤ (SUP z∈elts y. ZFC-in-HOL.succ (rank x + rank z))
        using ‹y ≠ 0›
        by (auto simp: plus-eq-lift intro: order-trans [OF - cSUP-upper [OF e]])
      then show ?thesis
          by (force simp: plus-V-succ-right simp flip: rank-Sup [of x] intro!: or-
der-antisym)
    qed
    also have ... = rank x + (SUP z ∈ elts y. succ (rank z))
      by (simp add: add-Sup-distrib False)
    also have ... = rank x + rank y
      by (simp add: rank-Sup [of y])
    finally show ?thesis .
  qed auto
qed

lemma Ord-add [simp]: ⟦Ord x; Ord y⟧ ⟹ Ord (x+y)
proof (induction y rule: eps-induct)
  case (step y)
  then show ?case
    by (metis Ord-rank rank-add-distrib rank-of-Ord)
qed

lemma add-Sup-distrib-id: A ≠ 0 ⟹ x + ⨆(elts A) = (SUP z∈elts A. x + z)
  by (metis add-Sup-distrib image-ident image-image)

lemma add-Limit: Limit α ⟹ x + α = (SUP z∈elts α. x + z)
  by (metis Limit-add-Sup-distrib Limit-eq-Sup-self image-ident image-image)

lemma add-le-left:
  assumes Ord α Ord β shows β ≤ α+β
  using ‹Ord β›
proof (induction rule: Ord-induct3)
  case 0
  then show ?case
    by auto
next
```

case (*succ* α)
then show *?case*
by (*auto simp*: *plus-V-succ-right Ord-mem-iff-lt assms*(*1*))
next
case (*Limit* μ)
then have *k*: $\mu = (SUP\ \beta \in elts\ \mu.\ \beta)$
by (*simp add*: *Limit-eq-Sup-self*)
also have $\ldots \leq (SUP\ \beta \in elts\ \mu.\ \alpha + \beta)$
using *Limit.IH* **by** *auto*
also have $\ldots = \alpha + (SUP\ \beta \in elts\ \mu.\ \beta)$
using *Limit.hyps Limit-add-Sup-distrib* **by** *presburger*
finally show *?case*
using *k* **by** *simp*
qed

lemma *plus-ω-equals-ω*:
assumes $\alpha \in elts\ \omega$ **shows** $\alpha + \omega = \omega$
proof (*rule antisym*)
show $\alpha + \omega \leq \omega$
using *Ord-trans assms* **by** (*auto simp*: *elim*!: *mem-plus-V-E*)
show $\omega \leq \alpha + \omega$
by (*simp add*: *add-le-left assms*)
qed

lemma *one-plus-ω-equals-ω* [*simp*]: $1 + \omega = \omega$
by (*simp add*: *one-V-def plus-ω-equals-ω*)

3.1.3 Cancellation / set subtraction

definition *vle* :: $V \Rightarrow V \Rightarrow bool$ (**infix** $\trianglelefteq$ *50*)
where $x \trianglelefteq y \equiv \exists z{::}V.\ x{+}z = y$

lemma *vle-refl* [*iff*]: $x \trianglelefteq x$
by (*metis* (*no-types*) *add.right-neutral vle-def*)

lemma *vle-antisym*: $[\![x \trianglelefteq y;\ y \trianglelefteq x]\!] \Longrightarrow x = y$
by (*metis V-equalityI plus-eq-lift sup-ge1 vle-def vsubsetD*)

lemma *vle-trans* [*trans*]: $[\![x \trianglelefteq y;\ y \trianglelefteq z]\!] \Longrightarrow x \trianglelefteq z$
by (*metis add.assoc vle-def*)

definition *vle-comparable* :: $V \Rightarrow V \Rightarrow bool$
where *vle-comparable x y* $\equiv x \trianglelefteq y \vee y \trianglelefteq x$

Lemma 3.13

lemma *comparable*:
assumes $a{+}b = c{+}d$
shows *vle-comparable a c*
unfolding *vle-comparable-def*

proof (*rule ccontr*)
assume *non*: $\neg\ (a \trianglelefteq c \vee c \trianglelefteq a)$
let $?\varphi = \lambda x.\ \forall z.\ a{+}x \neq c{+}z$
have $?\varphi\ x$ **for** x
proof (*induction x rule*: *eps-induct*)
case (*step x*)
show *?case*
proof (*cases x=0*)
case *True*
with *non nonzero-less-TC* **show** *?thesis*
using *vle-def* **by** *auto*
next
case *False*
then obtain v **where** $v \in elts\ x$
using *trad-foundation* **by** *blast*
show *?thesis*
proof *clarsimp*
fix z
assume *eq*: $a + x = c + z$
then have $z \neq 0$
using *vle-def non* **by** *auto*
have *av*: $a{+}v \in elts\ (a{+}x)$
by (*simp add*: ‹$v \in elts\ x$›)
moreover have $a{+}x = c \sqcup lift\ c\ z$
using *eq plus-eq-lift* **by** *fastforce*
ultimately have $a{+}v \in elts\ (c \sqcup lift\ c\ z)$
by *simp*
moreover
define u **where** $u \equiv set\ (elts\ x - \{v\})$
have *u*: $v \notin elts\ u$ **and** *xeq*: $x = vinsert\ v\ u$
using ‹$v \in elts\ x$› **by** (*auto simp*: *u-def intro*: *order-antisym*)
have *case1*: $a{+}v \notin elts\ c$
proof
assume *avc*: $a + v \in elts\ c$
then have $a \leq c$
by *clarify* (*metis Un-iff elts-sup-iff eq mem-not-sym mem-plus-V-E plus-eq-lift*)
moreover have $a \sqcup lift\ a\ x = c \sqcup lift\ c\ z$
using *eq* **by** (*simp add*: *plus-eq-lift*)
ultimately have $lift\ c\ z \leq lift\ a\ x$
by (*metis inf.absorb-iff2 inf-commute inf-sup-absorb inf-sup-distrib2 lift-self-disjoint sup.commute*)
also have $\ldots = vinsert\ (a{+}v)\ (lift\ a\ u)$
by (*simp add*: *lift-def vinsert-def xeq*)
finally have $*$: $lift\ c\ z \leq vinsert\ (a + v)\ (lift\ a\ u)$ **.**
have $lift\ c\ z \leq lift\ a\ u$
proof −
have $a + v \notin elts\ (lift\ c\ z)$
using *lift-self-disjoint* [*of c z*] *avc V-disjoint-iff* **by** *auto*

```
        then show ?thesis
          using ∗ less-eq-V-def by auto
      qed
      { fix e
        assume e ∈ elts z
        then have c+e ∈ elts (lift c z)
          by (simp add: lift-def)
        then have c+e ∈ elts (lift a u)
          using ‹lift c z ≤ lift a u› by blast
        then obtain y where y ∈ elts u c+e = a+y
          using lift-def by auto
        then have False
          by (metis elts-vinsert insert-iff step.IH xeq)
      }
      then show False
        using ‹z ≠ 0› by fastforce
    qed
    ultimately show False
      by (metis (no-types) ‹v ∈ elts x› av case1 eq mem-plus-V-E step.IH)
  qed
 qed
 then show False
  using assms by blast
qed

lemma vle1: x ⊴ y ⟹ x ≤ y
 using vle-def plus-eq-lift by auto

lemma vle2: x ⊴ y ⟹ x ⊑ y
 by (metis (full-types) TC-add' add.right-neutral le-TC-def vle-def nonzero-less-TC)

lemma vle-iff-le-Ord:
 assumes Ord α Ord β
 shows α ⊴ β ⟷ α ≤ β
proof
 show α ≤ β if α ⊴ β
  using that by (simp add: vle1)
 show α ⊴ β if α ≤ β
  using ‹Ord α› ‹Ord β› that
 proof (induction α arbitrary: β rule: Ord-induct)
  case (step γ)
  then show ?case
   unfolding vle-def
  by (metis Ord-add Ord-linear add-le-left mem-not-refl mem-plus-V-E vsubsetD)
 qed
qed

lemma add-le-cancel-left0 [iff]:
```

fixes $x::V$ **shows** $x \leq x+z$
by (*simp add*: *vle1 vle-def*)

lemma *add-less-cancel-left0* [*iff*]:
fixes $x::V$ **shows** $x < x+z \longleftrightarrow 0<z$
by (*metis add-less-cancel-left add.right-neutral*)

lemma *le-Ord-diff*:
assumes $\alpha \leq \beta$ *Ord* α *Ord* β
obtains γ **where** $\alpha+\gamma = \beta$ $\gamma \leq \beta$ *Ord* γ
proof −
obtain γ **where** γ: $\alpha+\gamma = \beta$ $\gamma \leq \beta$
by (*metis add-le-cancel-left add-le-left assms vle-def vle-iff-le-Ord*)
then have *Ord* γ
using *Ord-def Transset-def* ‹*Ord* β› **by** *force*
with γ *that* **show** *thesis* **by** *blast*
qed

lemma *plus-Ord-le*:
assumes $\alpha \in$ *elts* ω *Ord* β **shows** $\alpha+\beta \leq \beta+\alpha$
proof (*cases* $\beta \in$ *elts* ω)
case *True*
with *assms* **have** $\alpha+\beta = \beta+\alpha$
by (*auto simp*: *elts-ω add.commute ord-of-nat-add* [*symmetric*])
then show *?thesis* **by** *simp*
next
case *False*
then have $\omega \leq \beta$
using *Ord-linear2 Ord-mem-iff-lt* ‹*Ord* β› **by** *auto*
then obtain γ **where** $\omega+\gamma = \beta$ $\gamma \leq \beta$ *Ord* γ
using ‹*Ord* β› *le-Ord-diff* **by** *auto*
then have $\alpha+\beta = \beta$
by (*metis add.assoc assms*(*1*) *plus-ω-equals-ω*)
then show *?thesis*
by *simp*
qed

lemma *add-right-mono*: ⟦$\alpha \leq \beta$; *Ord* α; *Ord* β; *Ord* γ⟧ ⟹ $\alpha+\gamma \leq \beta+\gamma$
by (*metis add-le-cancel-left add.assoc add-le-left le-Ord-diff*)

lemma *add-strict-mono*: ⟦$\alpha < \beta$; $\gamma < \delta$; *Ord* α; *Ord* β; *Ord* γ; *Ord* δ⟧ ⟹ $\alpha+\gamma$ $< \beta+\delta$
by (*metis order.strict-implies-order add-less-cancel-left add-right-mono le-less-trans*)

lemma *add-right-strict-mono*: ⟦$\alpha \leq \beta$; $\gamma < \delta$; *Ord* α; *Ord* β; *Ord* γ; *Ord* δ⟧ ⟹ $\alpha+\gamma < \beta+\delta$
using *add-strict-mono le-imp-less-or-eq* **by** *blast*

lemma *Limit-add-Limit* [*simp*]:

assumes *Limit* μ *Ord* β **shows** *Limit* $(\beta + \mu)$
unfolding *Limit-def*
proof (*intro conjI allI impI*)
show *Ord* $(\beta + \mu)$
using *Limit-def assms* **by** *auto*
show $0 \in$ *elts* $(\beta + \mu)$
using *Limit-def add-le-left assms* **by** *auto*
next
fix γ
assume $\gamma \in$ *elts* $(\beta + \mu)$
then consider $\gamma \in$ *elts* $\beta \mid \xi$ **where** $\xi \in$ *elts* μ $\gamma = \beta + \xi$
using *mem-plus-V-E* **by** *blast*
then show *succ* $\gamma \in$ *elts* $(\beta + \mu)$
proof *cases*
case *1*
then show *?thesis*
by (*metis Kirby.add-strict-mono Limit-def Ord-add Ord-in-Ord Ord-mem-iff-lt assms one-V-def succ-eq-add1*)
next
case *2*
then show *?thesis*
by (*metis Limit-def add-mem-right-cancel assms*(*1*) *plus-V-succ-right*)
qed
qed

3.2 Generalised Difference

definition *odiff* **where** *odiff* y $x \equiv$ *THE* z::V. $(x{+}z = y) \vee (z{=}0 \wedge \neg\ x \trianglelefteq y)$

lemma *vle-imp-odiff-eq*: $x \trianglelefteq y \Longrightarrow x +$ (*odiff* y x) $= y$
by (*auto simp*: *vle-def odiff-def*)

lemma *not-vle-imp-odiff-0*: $\neg\ x \trianglelefteq y \Longrightarrow$ (*odiff* y x) $= 0$
by (*auto simp*: *vle-def odiff-def*)

lemma *Ord-odiff-eq*:
assumes $\alpha \leq \beta$ *Ord* α *Ord* β
shows $\alpha +$ *odiff* β $\alpha = \beta$
by (*simp add*: *assms vle-iff-le-Ord vle-imp-odiff-eq*)

lemma *Ord-odiff*:
assumes *Ord* α *Ord* β **shows** *Ord* (*odiff* β α)
proof (*cases* $\alpha \trianglelefteq \beta$)
case *True*
then show *?thesis*
by (*metis add-right-cancel assms le-Ord-diff vle1 vle-imp-odiff-eq*)
next
case *False*
then show *?thesis*

by (*simp add*: *odiff-def vle-def*)
qed

lemma *Ord-odiff-le*:
assumes *Ord* α *Ord* β **shows** *odiff* β $\alpha \leq \beta$
proof (*cases* $\alpha \trianglelefteq \beta$)
case *True*
then show *?thesis*
by (*metis add-right-cancel assms le-Ord-diff vle1 vle-imp-odiff-eq*)
next
case *False*
then show *?thesis*
by (*simp add*: *odiff-def vle-def*)
qed

lemma *odiff-0-right* [*simp*]: *odiff x 0* $= x$
by (*metis add.left-neutral vle-def vle-imp-odiff-eq*)

lemma *odiff-succ*: $y \trianglelefteq x \Longrightarrow$ *odiff* (*succ x*) $y =$ *succ* (*odiff x y*)
unfolding *odiff-def*
by (*metis add-right-cancel odiff-def plus-V-succ-right vle-def vle-imp-odiff-eq*)

lemma *odiff-eq-iff*: $z \trianglelefteq x \Longrightarrow$ *odiff x z* $= y \longleftrightarrow x = z + y$
by (*auto simp*: *odiff-def vle-def*)

lemma *odiff-le-iff*: $z \trianglelefteq x \Longrightarrow$ *odiff x z* $\leq y \longleftrightarrow x \leq z + y$
by (*auto simp*: *odiff-def vle-def*)

lemma *odiff-less-iff*: $z \trianglelefteq x \Longrightarrow$ *odiff x z* $< y \longleftrightarrow x < z + y$
by (*auto simp*: *odiff-def vle-def*)

lemma *odiff-ge-iff*: $z \trianglelefteq x \Longrightarrow$ *odiff x z* $\geq y \longleftrightarrow x \geq z + y$
by (*auto simp*: *odiff-def vle-def*)

lemma *Ord-odiff-le-iff*: $[\![\alpha \leq x; \textit{Ord } x; \textit{Ord } \alpha]\!] \Longrightarrow$ *odiff x* $\alpha \leq y \longleftrightarrow x \leq \alpha + y$
by (*simp add*: *odiff-le-iff vle-iff-le-Ord*)

lemma *odiff-le-odiff*:
assumes $x \trianglelefteq y$ **shows** *odiff x z* $\leq$ *odiff y z*
proof (*cases* $z \trianglelefteq x$)
case *True*
then show *?thesis*
using *assms odiff-le-iff vle1 vle-imp-odiff-eq vle-trans* **by** *presburger*
next
case *False*
then show *?thesis*
by (*simp add*: *not-vle-imp-odiff-0*)
qed

lemma *Ord-odiff-le-odiff*: $[\![x \leq y;\ Ord\ x;\ Ord\ y]\!] \Longrightarrow odiff\ x\ \alpha \leq odiff\ y\ \alpha$
by (*simp add*: *odiff-le-odiff vle-iff-le-Ord*)

lemma *Ord-odiff-less-odiff*: $[\![\alpha \leq x;\ x < y;\ Ord\ x;\ Ord\ y;\ Ord\ \alpha]\!] \Longrightarrow odiff\ x\ \alpha < odiff\ y\ \alpha$
by (*metis Ord-odiff-eq Ord-odiff-le-odiff dual-order.strict-trans less-V-def*)

lemma *Ord-odiff-less-imp-less*: $[\![odiff\ x\ \alpha < odiff\ y\ \alpha;\ Ord\ x;\ Ord\ y]\!] \Longrightarrow x < y$
by (*meson Ord-linear2 leD odiff-le-odiff vle-iff-le-Ord*)

lemma *odiff-add-cancel* [*simp*]: $odiff\ (x + y)\ x = y$
by (*simp add*: *odiff-eq-iff vle-def*)

lemma *odiff-add-cancel-0* [*simp*]: $odiff\ x\ x = 0$
by (*simp add*: *odiff-eq-iff*)

lemma *odiff-add-cancel-both* [*simp*]: $odiff\ (x + y)\ (x + z) = odiff\ y\ z$
by (*simp add*: *add.assoc odiff-def vle-def*)

3.3 Generalised Multiplication

Credited to Dana Scott

instantiation $V :: times$
begin

This definition is credited to Tarski

definition $times\text{-}V :: V \Rightarrow V \Rightarrow V$
where $times\text{-}V\ x \equiv transrec\ (\lambda f\ y.\ \bigsqcup\ ((\lambda u.\ lift\ (f\ u)\ x)\ `\ elts\ y))$

instance ..
end

lemma *mult*: $x * y = (SUP\ u{\in}elts\ y.\ lift\ (x * u)\ x)$
unfolding *times-V-def* **by** (*subst transrec*) (*force simp*:)

lemma *elts-multE*:
assumes $z \in elts\ (x * y)$
obtains $u\ v$ **where** $u \in elts\ x\ v \in elts\ y\ z = x{*}v + u$
using *mult* [*of x y*] *lift-def assms* **by** *auto*

Lemma 4.2

lemma *mult-zero-right* [*simp*]:
fixes $x{::}V$ **shows** $x * 0 = 0$
by (*metis ZFC-in-HOL.Sup-empty elts-0 image-empty mult*)

lemma *mult-insert*: $x * (vinsert\ y\ z) = x{*}z \sqcup lift\ (x{*}y)\ x$
by (*metis* (*no-types*, *lifting*) *elts-vinsert image-insert replacement small-elts sup-commute mult Sup-V-insert*)

lemma *mult-succ*: $x * succ\ y = x*y + x$
by (*simp add*: *mult-insert plus-eq-lift succ-def*)

lemma *ord-of-nat-mult*: *ord-of-nat* $(m*n) =$ *ord-of-nat* $m *$ *ord-of-nat* n
proof (*induction n*)
case (*Suc n*)
then show *?case*
by (*simp add*: *add.commute* [*of m*]) (*simp add*: *ord-of-nat-add mult-succ*)
qed *auto*

lemma *omega-closed-mult* [*intro*]:
assumes $\alpha \in elts\ \omega$ $\beta \in elts\ \omega$ **shows** $\alpha*\beta \in elts\ \omega$
proof −
obtain $m\ n$ **where** $\alpha =$ *ord-of-nat* m $\beta =$ *ord-of-nat* n
using *assms elts-ω* **by** *auto*
then have $\alpha*\beta =$ *ord-of-nat* $(m*n)$
by (*simp add*: *ord-of-nat-mult*)
then show *?thesis*
by (*simp add*: *ω-def*)
qed

lemma *zero-imp-le-mult*: $0 \in elts\ y \Longrightarrow x \leq x*y$
by (*auto simp*: *mult* [*of x y*])

3.3.1 Proposition 4.3

lemma *mult-zero-left* [*simp*]:
fixes $x :: V$ **shows** $0 * x = 0$
proof (*induction x rule*: *eps-induct*)
case (*step x*)
then show *?case*
by (*subst mult*) *auto*
qed

lemma *mult-sup-distrib*:
fixes $x :: V$ **shows** $x * (y \sqcup z) = x*y \sqcup x*z$
unfolding *mult* [*of x y* ⊔ *z*] *mult* [*of x y*] *mult* [*of x z*]
by (*simp add*: *Sup-Un-distrib image-Un*)

lemma *mult-Sup-distrib*: *small* $Y \Longrightarrow x * (\bigsqcup Y) = \bigsqcup\ ((*)\ x\ `\ Y)$ **for** $Y :: V\ set$
unfolding *mult* [*of x* ⨆ *Y*]
by (*simp add*: *cSUP-UNION*) (*metis mult*)

lemma *mult-lift-imp-distrib*: $x * (lift\ y\ z) = lift\ (x*y)\ (x*z) \Longrightarrow x * (y+z) = x*y + x*z$
by (*simp add*: *mult-sup-distrib plus-eq-lift*)

lemma *mult-lift*: $x * (lift\ y\ z) = lift\ (x*y)\ (x*z)$

proof (*induction z rule*: *eps-induct*)
case (*step z*)
have $x * lift\ y\ z = (SUP\ u{\in}elts\ (lift\ y\ z).\ lift\ (x * u)\ x)$
using *mult* **by** *blast*
also have $\ldots = (SUP\ v{\in}elts\ z.\ lift\ (x * (y + v))\ x)$
using *lift-def* **by** *auto*
also have $\ldots = (SUP\ v{\in}elts\ z.\ lift\ (x * y + x * v)\ x)$
using *mult-lift-imp-distrib step.IH* **by** *auto*
also have $\ldots = (SUP\ v{\in}elts\ z.\ lift\ (x * y)\ (lift\ (x * v)\ x))$
by (*simp add*: *lift-lift*)
also have $\ldots = lift\ (x * y)\ (SUP\ v{\in}elts\ z.\ lift\ (x * v)\ x)$
by (*simp add*: *image-image lift-Sup-distrib*)
also have $\ldots = lift\ (x{*}y)\ (x{*}z)$
by (*metis mult*)
finally show *?case* **.**
qed

lemma *mult-Limit*: $Limit\ \gamma \Longrightarrow x * \gamma = \bigsqcup\ ((*)\ x\ `\ elts\ \gamma)$
by (*metis Limit-eq-Sup-self mult-Sup-distrib small-elts*)

lemma *add-mult-distrib*: $x * (y{+}z) = x{*}y + x{*}z$ **for** $x{::}V$
by (*simp add*: *mult-lift mult-lift-imp-distrib*)

instantiation $V :: monoid\text{-}mult$
begin
instance
proof
show $1 * x = x$ **for** $x :: V$
proof (*induction x rule*: *eps-induct*)
case (*step x*)
then show *?case*
by (*subst mult*) *auto*
qed
show $x * 1 = x$ **for** $x :: V$
by (*subst mult*) *auto*
show $(x * y) * z = x * (y * z)$ **for** $x\ y\ z{::}V$
proof (*induction z rule*: *eps-induct*)
case (*step z*)
have $(x * y) * z = (SUP\ u{\in}elts\ z.\ lift\ (x * y * u)\ (x * y))$
using *mult* **by** *blast*
also have $\ldots = (SUP\ u{\in}elts\ z.\ lift\ (x * (y * u))\ (x * y))$
using *step.IH* **by** *auto*
also have $\ldots = (SUP\ u{\in}elts\ z.\ x * lift\ (y * u)\ y)$
using *mult-lift* **by** *auto*
also have $\ldots = x * (SUP\ u{\in}elts\ z.\ lift\ (y * u)\ y)$
by (*simp add*: *image-image mult-Sup-distrib*)
also have $\ldots = x * (y * z)$
by (*metis mult*)
finally show *?case* **.**

```
  qed
qed

end

lemma le-mult:
  assumes Ord β β ≠ 0 shows α ≤ α ∗ β
  using assms
proof (induction rule: Ord-induct3)
  case (succ α)
  then show ?case
    using mult-insert succ-def by fastforce
next
  case (Limit μ)
  have α ∈ (∗) α ‘ elts μ
    using Limit.hyps Limit-def one-V-def by (metis imageI mult.right-neutral)
  then have α ≤ ⋃ ((∗) α ‘ elts μ)
    by auto
  then show ?case
    by (simp add: Limit.hyps mult-Limit)
qed auto

lemma mult-sing-1 [simp]:
  fixes x::V shows x ∗ set{1} = lift x x
  by (subst mult) auto

lemma mult-2-right [simp]:
  fixes x::V shows x ∗ set{0,1} = x+x
  by (subst mult) (auto simp: Sup-V-insert plus-eq-lift)

lemma Ord-mult [simp]: ⟦Ord y; Ord x⟧ ⟹ Ord (x∗y)
proof (induction y rule: Ord-induct3)
  case 0
  then show ?case
    by auto
next
  case (succ k)
  then show ?case
    by (simp add: mult-succ)
next
  case (Limit k)
  then have Ord (x ∗ ⋃ (elts k))
    by (metis Ord-Sup imageE mult-Sup-distrib small-elts)
  then show ?case
    using Limit.hyps Limit-eq-Sup-self by auto
qed
```

3.3.2 Proposition 4.4-5

proposition *rank-mult-distrib*: *rank* (*x*∗*y*) = *rank x* ∗ *rank y*
proof (*induction y rule*: *eps-induct*)
case (*step y*)
have *rank* (*x*∗*y*) = (*SUP y*∈*elts* (*SUP u*∈*elts y*. *lift* (*x* ∗ *u*) *x*). *succ* (*rank y*))
by (*metis rank-Sup mult*)
also have ... = (*SUP u*∈*elts y*. *SUP r*∈*elts x*. *succ* (*rank* (*x* ∗ *u* + *r*)))
apply (*simp add*: *lift-def image-image image-UN*)
apply (*simp add*: *Sup-V-def*)
done
also have ... = (*SUP u*∈*elts y*. *SUP r*∈*elts x*. *succ* (*rank* (*x* ∗ *u*) + *rank r*))
using *rank-add-distrib* **by** *auto*
also have ... = (*SUP u*∈*elts y*. *SUP r*∈*elts x*. *succ* (*rank x* ∗ *rank u* + *rank r*))
using *step arg-cong* [**where** *f* = *Sup*] **by** *auto*
also have ... = (*SUP u*∈*elts y*. *rank x* ∗ *rank u* + *rank x*)
proof (*rule SUP-cong*)
show (*SUP r*∈*elts x*. *succ* (*rank x* ∗ *rank u* + *rank r*)) = *rank x* ∗ *rank u* + *rank x*
if *u* ∈ *elts y* **for** *u*
proof (*cases x*=*0*)
case *False*
have (*SUP r*∈*elts x*. *succ* (*rank x* ∗ *rank u* + *rank r*)) = *rank x* ∗ *rank u* + (*SUP y*∈*elts x*. *succ* (*rank y*))
proof (*rule order-antisym*)
show (*SUP r*∈*elts x*. *succ* (*rank x* ∗ *rank u* + *rank r*)) ≤ *rank x* ∗ *rank u* + (*SUP y*∈*elts x*. *succ* (*rank y*))
by (*auto simp*: *Sup-le-iff simp flip*: *plus-V-succ-right*)
have *rank x* ∗ *rank u* + (*SUP y*∈*elts x*. *succ* (*rank y*)) = (*SUP y*∈*elts x*. *rank x* ∗ *rank u* + *succ* (*rank y*))
by (*simp add*: *add-Sup-distrib False*)
also have ... ≤ (*SUP r*∈*elts x*. *succ* (*rank x* ∗ *rank u* + *rank r*))
using *plus-V-succ-right* **by** *auto*
finally show *rank x* ∗ *rank u* + (*SUP y*∈*elts x*. *succ* (*rank y*)) ≤ (*SUP r*∈*elts x*. *succ* (*rank x* ∗ *rank u* + *rank r*)) **.**
qed
also have ... = *rank x* ∗ *rank u* + *rank x*
by (*metis rank-Sup*)
finally show *?thesis* **.**
qed *auto*
qed *auto*
also have ... = *rank x* ∗ *rank y*
by (*simp add*: *rank-Sup* [*of y*] *mult-Sup-distrib mult-succ image-image*)
finally show *?case* **.**
qed

lemma *mult-le1*:
fixes *y*::*V* **assumes** *y* ≠ *0* **shows** *x* ⊑ *x* ∗ *y*
proof (*cases x* = *0*)
case *False*

then obtain r **where** r: $r \in elts\ x$
by *fastforce*
from ‹$y \neq 0$› **show** *?thesis*
proof (*induction y rule*: *eps-induct*)
case (*step y*)
show *?case*
proof (*cases y = 1*)
case *False*
with ‹$y \neq 0$› **obtain** p **where** p: $p \in elts\ y\ p \neq 0$
by (*metis V-equalityI elts-1 insertI1 singletonD trad-foundation*)
then have $x{*}p + r \in elts\ (lift\ (x{*}p)\ x)$
by (*simp add*: *lift-def r*)
moreover have $lift\ (x{*}p)\ x \leq x{*}y$
by (*metis bdd-above-iff-small cSUP-upper2 order-refl* ‹$p \in elts\ y$› *replacement small-elts mult*)
ultimately have $x{*}p + r \in elts\ (x{*}y)$
by *blast*
moreover have $x{*}p \sqsubseteq x{*}p + r$
by (*metis TC-add′ V-equalityI add.right-neutral eps-induct le-TC-refl less-TC-iff less-imp-le-TC*)
ultimately show *?thesis*
using *step.IH* [*OF p*] *le-TC-trans less-TC-iff* **by** *blast*
qed *auto*
qed
qed *auto*

lemma *mult-eq-0-iff* [*simp*]:
fixes $y{::}V$ **shows** $x * y = 0 \longleftrightarrow x{=}0 \vee y{=}0$
proof
show $x = 0 \vee y = 0$ **if** $x * y = 0$
by (*metis le-0 le-TC-def less-TC-imp-not-le mult-le1 that*)
qed *auto*

lemma *lift-lemma*:
assumes $x \neq 0\ y \neq 0$ **shows** $\neg\ lift\ (x * y)\ x \leq x$
using *assms mult-le1* [*of concl*: *x y*]
by (*auto simp*: *le-TC-def TC-lift less-TC-def less-TC-imp-not-le*)

lemma *mult-le2*:
fixes $y{::}V$ **assumes** $x \neq 0\ y \neq 0\ y \neq 1$ **shows** $x \sqsubset x * y$
proof −
obtain v **where** v: $v \in elts\ y\ v \neq 0$
using *assms* **by** *fastforce*
have $x \neq x * y$
using *lift-lemma* [*of x v*]
by (*metis* ‹$x \neq 0$› *bdd-above-iff-small cSUP-upper2 order-refl replacement small-elts mult v*)
then show *?thesis*
using *assms mult-le1* [*of y x*]

by (*auto simp*: *le-TC-def*)
qed

lemma *elts-mult-ωE*:
assumes $x \in elts\ (y * \omega)$
obtains n **where** $n \neq 0\ x \in elts\ (y * ord\text{-}of\text{-}nat\ n)\ \bigwedge m.\ m < n \Longrightarrow x \notin elts\ (y * ord\text{-}of\text{-}nat\ m)$
proof −
obtain k **where** k: $k \neq 0 \wedge x \in elts\ (y * ord\text{-}of\text{-}nat\ k)$
using *assms*
apply (*simp add*: *mult-Limit elts-ω*)
by (*metis mult-eq-0-iff elts-0 ex-in-conv ord-of-eq-0-iff that*)
define n **where** $n \equiv (LEAST\ k.\ k \neq 0 \wedge x \in elts\ (y * ord\text{-}of\text{-}nat\ k))$
show *thesis*
proof
show $n \neq 0\ x \in elts\ (y * ord\text{-}of\text{-}nat\ n)$
unfolding *n-def* **by** (*metis* (*mono-tags*, *lifting*) *LeastI-ex k*)+
show $\bigwedge m.\ m < n \Longrightarrow x \notin elts\ (y * ord\text{-}of\text{-}nat\ m)$
by (*metis* (*mono-tags*, *lifting*) *mult-eq-0-iff elts-0 empty-iff n-def not-less-Least ord-of-eq-0-iff*)
qed
qed

3.3.3 Theorem 4.6

theorem *mult-eq-imp-0*:
assumes $a*x = a*y + b\ b \sqsubset a$
shows *b=0*
proof (*cases a=0* ∨ *x=0*)
case *True*
with *assms* **show** *?thesis*
by (*metis add-le-cancel-left mult-eq-0-iff eq-iff le-0*)
next
case *False*
then have *a≠0 x≠0*
by *auto*
then show *?thesis*
proof (*cases y=0*)
case *True*
then show *?thesis*
using *assms less-asym-TC mult-le2* **by** *force*
next
case *False*
have *b=0* **if** *Ord α x* ∈ *elts* (*Vset α*) *y* ∈ *elts* (*Vset α*) **for** *α*
using *that assms*
proof (*induction α arbitrary*: *x y b rule*: *Ord-induct3*)
case *0*
then show *?case* **by** *auto*
next

case (*succ k*)
define Φ **where** $\Phi \equiv \lambda x\ y.\ \exists r.\ 0 \sqsubset r \wedge r \sqsubset a \wedge a{*}x = a{*}y + r$
show *?case*
proof (*rule ccontr*)
assume $b \neq 0$
then have $0 \sqsubset b$
by (*metis nonzero-less-TC*)
then have $\Phi\ x\ y$
unfolding Φ-*def* **using** *succ.prems* **by** *blast*
then obtain x' **where** $\Phi\ x'\ y\ x' \sqsubseteq x$ **and** *min*: $\bigwedge x''.\ x'' \sqsubset x' \Longrightarrow \neg\ \Phi\ x''\ y$
using *less-TC-minimal* [*of* $\lambda x.\ \Phi\ x\ y\ x$] **by** *blast*
then obtain b' **where** $0 \sqsubset b'\ b' \sqsubset a$ **and** *eq*: $a{*}x' = a{*}y + b'$
using Φ-*def* **by** *blast*
have $a{*}y \sqsubset a{*}x'$
using *TC-add'* ‹$0 \sqsubset b'$› *eq* **by** *auto*
then obtain p **where** $p \in elts\ (a * x')\ a * y \sqsubseteq p$
using *less-TC-iff* **by** *blast*
then have $p \notin elts\ (a * y)$
using *less-TC-iff less-irrefl-TC* **by** *blast*
then have $p \in \bigcup\ (elts\ `\ (\lambda v.\ lift\ (a * v)\ a)\ `\ elts\ x')$
by (*metis* ‹$p \in elts\ (a * x')$› *elts-Sup replacement small-elts mult*)
then obtain $u\ c$ **where** $u \in elts\ x'\ c \in elts\ a\ p = a{*}u + c$
using *lift-def* **by** *auto*
then have $p \in elts\ (lift\ (a{*}y)\ b')$
using ‹$p \in elts\ (a * x')$› ‹$p \notin elts\ (a * y)$› *eq plus-eq-lift* **by** *auto*
then obtain d **where** *d*: $d \in elts\ b'\ p = a{*}y + d\ p = a{*}u + c$
by (*metis* ‹$p = a * u + c$› ‹$p \in elts\ (a * x')$› ‹$p \notin elts\ (a * y)$› *eq mem-plus-V-E*)
have *noteq*: $a{*}y \neq a{*}u$
proof
assume $a{*}y = a{*}u$
then have $lift\ (a{*}y)\ a = lift\ (a{*}u)\ a$
by *metis*
also have $\ldots \leq a{*}x'$
unfolding *mult* [*of* - x'] **using** ‹$u \in elts\ x'$› **by** (*auto intro*: *cSUP-upper*)
also have $\ldots = a{*}y \sqcup lift\ (a{*}y)\ b'$
by (*simp add*: *eq plus-eq-lift*)
finally have $lift\ (a{*}y)\ a \leq a{*}y \sqcup lift\ (a{*}y)\ b'$ **.**
then have $lift\ (a{*}y)\ a \leq lift\ (a{*}y)\ b'$
using *add-le-cancel-left less-TC-imp-not-le plus-eq-lift* ‹$b' \sqsubset a$› **by** *auto*
then have $a \leq b'$
by (*simp add*: *le-iff-sup lift-eq-lift lift-sup-distrib*)
then show *False*
using ‹$b' \sqsubset a$› *less-TC-imp-not-le* **by** *auto*
qed
consider $a{*}y \trianglelefteq a{*}u \mid a{*}u \trianglelefteq a{*}y$
using *d comparable vle-comparable-def* **by** *auto*
then show *False*
proof *cases*

case *1*
then obtain *e* **where** *e*: $a*u = a*y + e$ $e \neq 0$
by (*metis add.right-neutral noteq vle-def*)
moreover have $e + c = d$
by (*metis e add-right-cancel* ‹$p = a * u + c$› ‹$p = a * y + d$› *add.assoc*)
with ‹$d \in elts\ b'$› ‹$b' \sqsubset a$› **have** $e \sqsubset a$
by (*meson less-TC-iff less-TC-trans vle2 vle-def*)
ultimately show *False*
— contradicts minimality of x'
using *min* **unfolding** Φ*-def* **by** (*meson* ‹$u \in elts\ x'$› *le-TC-def less-TC-iff nonzero-less-TC*)
next
case *2*
then obtain *e* **where** *e*: $a*y = a*u + e$ $e \neq 0$
by (*metis add.right-neutral noteq vle-def*)
moreover have $e + d = c$
by (*metis e add-right-cancel* ‹$p = a * u + c$› ‹$p = a * y + d$› *add.assoc*)
with ‹$d \in elts\ b'$› ‹$b' \sqsubset a$› **have** $e \sqsubset a$
by (*metis* ‹$c \in elts\ a$› *less-TC-iff vle2 vle-def*)
ultimately have $\Phi\ y\ u$
unfolding Φ*-def* **using** *nonzero-less-TC* **by** *blast*
then obtain y' **where** $\Phi\ y'\ u\ y' \sqsubseteq y$ **and** *min*: $\bigwedge x''.\ x'' \sqsubset y' \Longrightarrow \neg\ \Phi\ x''\ u$
using *less-TC-minimal* [*of* $\lambda x.\ \Phi\ x\ u\ y$] **by** *blast*
then obtain b' **where** $0 \sqsubset b'$ $b' \sqsubset a$ **and** *eq*: $a*y' = a*u + b'$
using Φ*-def* **by** *blast*
have *u-k*: $u \in elts\ (Vset\ k)$
using ‹$u \in elts\ x'$› ‹$x' \sqsubseteq x$› *succ Vset-succ-TC less-TC-iff less-le-TC-trans* **by** *blast*
have $a*u \sqsubset a*y'$
using *TC-add'* ‹$0 \sqsubset b'$› *eq* **by** *auto*
then obtain *p* **where** $p \in elts\ (a * y')$ $a * u \sqsubseteq p$
using *less-TC-iff* **by** *blast*
then have $p \notin elts\ (a * u)$
using *less-TC-iff less-irrefl-TC* **by** *blast*
then have $p \in \bigcup\ (elts\ `\ (\lambda v.\ lift\ (a * v)\ a)\ `\ elts\ y')$
by (*metis* ‹$p \in elts\ (a * y')$› *elts-Sup replacement small-elts mult*)
then obtain *v c* **where** $v \in elts\ y'$ $c \in elts\ a$ $p = a*v + c$
using *lift-def* **by** *auto*
then have $p \in elts\ (lift\ (a*u)\ b')$
using ‹$p \in elts\ (a * y')$› ‹$p \notin elts\ (a * u)$› *eq plus-eq-lift* **by** *auto*
then obtain *d* **where** *d*: $d \in elts\ b'$ $p = a*u + d$ $p = a*v + c$
by (*metis* ‹$p = a * v + c$› ‹$p \in elts\ (a * y')$› ‹$p \notin elts\ (a * u)$› *eq mem-plus-V-E*)
have *v-k*: $v \in elts\ (Vset\ k)$
using *Vset-succ-TC* ‹$v \in elts\ y'$› ‹$y' \sqsubseteq y$› *less-TC-iff less-le-TC-trans succ.hyps succ.prems*(*2*) **by** *blast*
have *noteq*: $a*u \neq a*v$
proof

```
            assume a*u = a*v
            then have lift (a*v) a ≤ a*y'
           unfolding mult [of - y'] using ‹v ∈ elts y'› by (auto intro: cSUP-upper)
            also have ... = a*u ⊔ lift (a*u) b'
              by (simp add: eq plus-eq-lift)
            finally have lift (a*v) a ≤ a*u ⊔ lift (a*u) b' .
            then have lift (a*u) a ≤ lift (a*u) b'
              by (metis ‹a * u = a * v› le-iff-sup lift-sup-distrib sup-left-commute
sup-lift-eq-lift)
            then have a ≤ b'
              by (simp add: le-iff-sup lift-eq-lift lift-sup-distrib)
            then show False
              using ‹b' ⊏ a› less-TC-imp-not-le by auto
          qed
        consider a*u ⊴ a*v | a*v ⊴ a*u
          using d comparable vle-comparable-def by auto
        then show False
        proof cases
          case 1
          then obtain e where e: a*v = a*u + e e ≠ 0
            by (metis add.right-neutral noteq vle-def)
          moreover have e + c = d
            by (metis add-right-cancel ‹p = a * u + d› ‹p = a * v + c› add.assoc
e)
          with ‹d ∈ elts b'› ‹b' ⊏ a› have e ⊏ a
            by (meson less-TC-iff less-TC-trans vle2 vle-def)
          ultimately show False
            using succ.IH u-k v-k by blast
        next
          case 2
          then obtain e where e: a*u = a*v + e e ≠ 0
            by (metis add.right-neutral noteq vle-def)
          moreover have e + d = c
            by (metis add-right-cancel add.assoc d e)
          with ‹d ∈ elts b'› ‹b' ⊏ a› have e ⊏ a
            by (metis ‹c ∈ elts a› less-TC-iff vle2 vle-def)
         ultimately show False
           using succ.IH u-k v-k by blast
        qed
      qed
    qed
  next
    case (Limit k)
    obtain i j where k: i ∈ elts k j ∈ elts k
      and x: x ∈ elts (Vset i)
      and y: y ∈ elts (Vset j)
      using that Limit by (auto simp: Limit-Vfrom-eq)
    show ?case
    proof (rule Limit.IH [of i ⊔ j])
```

```
      show i ⊔ j ∈ elts k
        by (meson k x y Limit.hyps Limit-def Ord-in-Ord Ord-mem-iff-lt Ord-sup
union-less-iff)
      show x ∈ elts (Vset (i ⊔ j)) y ∈ elts (Vset (i ⊔ j))
        using x y by (auto simp: Vfrom-sup)
    qed (use Limit.prems in auto)
  qed
  then show ?thesis
    by (metis two-in-Vset Ord-rank Ord-VsetI rank-lt)
 qed
qed
```

3.3.4 Theorem 4.7

```
lemma mult-cancellation-half:
 assumes a*x + r ≤ a*y + s r ⊏ a s ⊏ a
 shows x ≤ y
proof −
 have x ≤ y if Ord α x ∈ elts (Vset α) y ∈ elts (Vset α) for α
  using that assms
 proof (induction α arbitrary: x y r s rule: Ord-induct3)
  case 0
  then show ?case
   by auto
 next
  case (succ k)
  show ?case
  proof
   fix u
   assume u: u ∈ elts x
   have u-k: u ∈ elts (Vset k)
    using Vset-succ succ.hyps succ.prems(1) u by auto
   obtain r' where r' ∈ elts a r ⊑ r'
    using less-TC-iff succ.prems(4) by blast
   have a*u + r' ∈ elts (lift (a*u) a)
    by (simp add: ‹r' ∈ elts a› lift-def)
   also have ... ≤ elts (a*x)
    using u by (force simp: mult [of - x])
   also have ... ≤ elts (a*y + s)
    using plus-eq-lift succ.prems(3) by auto
   also have ... = elts (a*y) ∪ elts (lift (a*y) s)
    by (simp add: plus-eq-lift)
   finally have a * u + r' ∈ elts (a * y) ∪ elts (lift (a * y) s) .
   then show u ∈ elts y
   proof
    assume *: a * u + r' ∈ elts (a * y)
    show u ∈ elts y
    proof −
     obtain v e where v: v ∈ elts y e ∈ elts a a * u + r' = a * v + e
```

```
      using ∗ by (auto simp: mult [of - y] lift-def)
    then have v-k: v ∈ elts (Vset k)
      using Vset-succ-TC less-TC-iff succ.prems(2) by blast
    then show ?thesis
      by (metis ‹r′ ∈ elts a› antisym le-TC-refl less-TC-iff order-refl succ.IH
u-k v)
  qed
 next
  assume a ∗ u + r′ ∈ elts (lift (a ∗ y) s)
  then obtain t where t ∈ elts s and t: a ∗ u + r′ = a ∗ y + t
    using lift-def by auto
  have noteq: a∗y ≠ a∗u
  proof
    assume a∗y = a∗u
    then have lift (a∗y) a = lift (a∗u) a
      by metis
    also have ... ≤ a∗x
      unfolding mult [of - x] using ‹u ∈ elts x› by (auto intro: cSUP-upper)
    also have ... ≤ a∗y ⊔ lift (a∗y) s
      using ‹elts (a ∗ x) ⊆ elts (a ∗ y + s)› plus-eq-lift by auto
    finally have lift (a∗y) a ≤ a∗y ⊔ lift (a∗y) s .
    then have lift (a∗y) a ≤ lift (a∗y) s
      using add-le-cancel-left less-TC-imp-not-le plus-eq-lift ‹s ⊏ a› by auto
    then have a ≤ s
      by (simp add: le-iff-sup lift-eq-lift lift-sup-distrib)
    then show False
      using ‹s ⊏ a› less-TC-imp-not-le by auto
  qed
  consider a ∗ u ⊴ a ∗ y | a ∗ y ⊴ a ∗ u
    using t comparable vle-comparable-def by blast
  then have False
  proof cases
    case 1
    then obtain c where a∗y = a∗u + c
      by (metis vle-def)
    then have c+t = r′
      by (metis add-right-cancel add.assoc t)
    then have c ⊏ a
      using ‹r′ ∈ elts a› less-TC-iff vle2 vle-def by force
    moreover have c ≠ 0
      using ‹a ∗ y = a ∗ u + c› noteq by auto
    ultimately show ?thesis
      using ‹a ∗ y = a ∗ u + c› mult-eq-imp-0 by blast
  next
    case 2
    then obtain c where a∗u = a∗y + c
      by (metis vle-def)
    then have c+r′ = t
      by (metis add-right-cancel add.assoc t)
```

```
        then have c ⊏ a
          by (metis ‹t ∈ elts s› less-TC-iff less-TC-trans ‹s ⊏ a› vle2 vle-def)
        moreover have c ≠ 0
          using ‹a * u = a * y + c› noteq by auto
        ultimately show ?thesis
          using ‹a * u = a * y + c› mult-eq-imp-0 by blast
      qed
      then show u ∈ elts y ..
    qed
  qed
 next
  case (Limit k)
  obtain i j where k: i ∈ elts k j ∈ elts k
    and x: x ∈ elts (Vset i) and y: y ∈ elts (Vset j)
    using that Limit by (auto simp: Limit-Vfrom-eq)
  show ?case
  proof (rule Limit.IH [of i ⊔ j])
    show i ⊔ j ∈ elts k
      by (meson k x y Limit.hyps Limit-def Ord-in-Ord Ord-mem-iff-lt Ord-sup
union-less-iff)
    show x ∈ elts (Vset (i ⊔ j)) y ∈ elts (Vset (i ⊔ j))
      using x y by (auto simp: Vfrom-sup)
    show a * x + r ≤ a * y + s
      by (simp add: Limit.prems)
  qed (auto simp: Limit.prems)
 qed
 then show ?thesis
  by (metis two-in-Vset Ord-rank Ord-VsetI rank-lt)
qed

theorem mult-cancellation-lemma:
 assumes a*x + r = a*y + s r ⊏ a s ⊏ a
 shows x=y ∧ r=s
 by (metis assms leD less-V-def mult-cancellation-half odiff-add-cancel order-refl)

corollary mult-cancellation [simp]:
 fixes a::V
 assumes a ≠ 0
 shows a*x = a*y ⟷ x=y
 by (metis assms nonzero-less-TC mult-cancellation-lemma)

corollary mult-cancellation-less:
 assumes lt: a*x + r < a*y + s and r ⊏ a s ⊏ a
 obtains x < y | x = y r < s
proof −
 have x ≤ y
  by (meson assms dual-order.strict-implies-order mult-cancellation-half)
 then consider x < y | x = y
  using less-V-def by blast
```

with *lt that* **show** *?thesis* **by** *blast*
qed

corollary *lift-mult-TC-disjoint*:
fixes $x::V$
assumes $x \neq y$
shows *lift* $(a*x)$ $(TC\ a) \sqcap$ *lift* $(a*y)$ $(TC\ a) = 0$
apply (*rule V-equalityI*)
using *assms*
by (*auto simp*: *less-TC-def inf-V-def lift-def image-iff dest*: *mult-cancellation-lemma*)

corollary *lift-mult-disjoint*:
fixes $x::V$
assumes $x \neq y$
shows *lift* $(a*x)$ $a \sqcap$ *lift* $(a*y)$ $a = 0$
proof −
have *lift* $(a*x)$ $a \sqcap$ *lift* $(a*y)$ $a \leq$ *lift* $(a*x)$ $(TC\ a) \sqcap$ *lift* $(a*y)$ $(TC\ a)$
by (*metis TC′ inf-mono lift-sup-distrib sup-ge1*)
then show *?thesis*
using *assms lift-mult-TC-disjoint* **by** *auto*
qed

lemma *mult-add-mem*:
assumes $a*x + r \in$ *elts* $(a*y)$ $r \sqsubset a$
shows $x \in$ *elts* y $r \in$ *elts* a
proof −
obtain v s **where** v: $a * x + r = a * v + s$ $v \in$ *elts* y $s \in$ *elts* a
using *assms* **unfolding** *mult* [*of a y*] *lift-def* **by** *auto*
then show $x \in$ *elts* y
by (*metis arg-subset-TC assms*(*2*) *less-TC-def mult-cancellation-lemma vsubsetD*)
show $r \in$ *elts* a
by (*metis arg-subset-TC assms*(*2*) *less-TC-def mult-cancellation-lemma v*(*1*) *v*(*3*) *vsubsetD*)
qed

lemma *mult-add-mem-0* [*simp*]: $a*x \in$ *elts* $(a*y) \longleftrightarrow x \in$ *elts* $y \wedge 0 \in$ *elts* a
proof −
have $x \in$ *elts* y
if $a * x \in$ *elts* $(a * y) \wedge 0 \in$ *elts* a
using *that* **using** *mult-add-mem* [*of a x 0*]
using *nonzero-less-TC* **by** *force*
moreover have $a * x \in$ *elts* $(a * y)$
if $x \in$ *elts* y $0 \in$ *elts* a
using *that* **by** (*force simp*: *image-iff mult* [*of a y*] *lift-def*)
ultimately show *?thesis*
by (*metis mult-eq-0-iff add.right-neutral mult-add-mem*(*2*) *nonzero-less-TC*)
qed

lemma *zero-mem-mult-iff*: $0 \in elts\ (x*y) \longleftrightarrow 0 \in elts\ x \wedge 0 \in elts\ y$
by (*metis Kirby.mult-zero-right mult-add-mem-0*)

lemma *zero-less-mult-iff* [*simp*]: $0 < x*y \longleftrightarrow 0 < x \wedge 0 < y$ **if** *Ord x*
using *Kirby.mult-eq-0-iff ZFC-in-HOL.neq0-conv* **by** *blast*

lemma *mult-cancel-less-iff* [*simp*]:
$[\![Ord\ \alpha;\ Ord\ \beta;\ Ord\ \gamma]\!] \Longrightarrow \alpha*\beta < \alpha*\gamma \longleftrightarrow \beta < \gamma \wedge 0 < \alpha$
using *mult-add-mem-0* [*of* $\alpha\ \beta\ \gamma$]
by (*meson Ord-0 Ord-mem-iff-lt Ord-mult*)

lemma *mult-cancel-le-iff* [*simp*]:
$[\![Ord\ \alpha;\ Ord\ \beta;\ Ord\ \gamma]\!] \Longrightarrow \alpha*\beta \leq \alpha*\gamma \longleftrightarrow \beta \leq \gamma \vee \alpha=0$
by (*metis Ord-linear2 Ord-mult eq-iff leD mult-cancel-less-iff mult-cancellation*)

lemma *mult-Suc-add-less*: $[\![\alpha < \gamma;\ \beta < \gamma;\ Ord\ \alpha;\ Ord\ \beta;\ Ord\ \gamma]\!] \Longrightarrow \gamma * ord\text{-}of\text{-}nat\ m + \alpha < \gamma * ord\text{-}of\text{-}nat\ (Suc\ m) + \beta$
apply (*simp add*: *mult-succ add.assoc*)
by (*meson Ord-add Ord-linear2 le-less-trans not-add-less-right*)

lemma *mult-nat-less-add-less*:
assumes $m < n\ \alpha < \gamma\ \beta < \gamma$ **and** *ord*: *Ord* α *Ord* β *Ord* γ
shows $\gamma * ord\text{-}of\text{-}nat\ m + \alpha < \gamma * ord\text{-}of\text{-}nat\ n + \beta$
proof −
have $Suc\ m \leq n$
using ‹$m < n$› **by** *auto*
have $\gamma * ord\text{-}of\text{-}nat\ m + \alpha < \gamma * ord\text{-}of\text{-}nat\ (Suc\ m) + \beta$
using *assms mult-Suc-add-less* **by** *blast*
also have $\ldots \leq \gamma * ord\text{-}of\text{-}nat\ n + \beta$
using *Ord-mult Ord-ord-of-nat add-right-mono* ‹$Suc\ m \leq n$› *ord mult-cancel-le-iff ord-of-nat-mono-iff* **by** *presburger*
finally show *?thesis* **.**
qed

lemma *add-mult-less-add-mult*:
assumes $x < y\ x \in elts\ \beta\ y \in elts\ \beta\ \mu \in elts\ \alpha\ \nu \in elts\ \alpha$ *Ord* α *Ord* β
shows $\alpha*x + \mu < \alpha*y + \nu$
proof −
obtain *Ord x Ord y*
using *Ord-in-Ord assms* **by** *blast*
then obtain δ **where** $0 \in elts\ \delta\ y = x + \delta$
by (*metis add.right-neutral* ‹$x < y$› *le-Ord-diff less-V-def mem-0-Ord*)
then show *?thesis*
apply (*simp add*: *add-mult-distrib add.assoc*)
by (*meson OrdmemD add-le-cancel-left0* ‹$\mu \in elts\ \alpha$› ‹*Ord* α› *less-le-trans zero-imp-le-mult*)
qed

lemma *add-mult-less*:

assumes $\gamma \in elts\ \alpha\ \nu \in elts\ \beta\ Ord\ \alpha\ Ord\ \beta$
shows $\alpha * \nu + \gamma \in elts\ (\alpha * \beta)$
proof −
have $Ord\ \nu$
using *Ord-in-Ord assms* **by** *blast*
with *assms* **show** *?thesis*
by (*metis Ord-mem-iff-lt Ord-succ add-mem-right-cancel mult-cancel-le-iff mult-succ succ-le-iff vsubsetD*)
qed

lemma *Ord-add-mult-iff*:
assumes $\beta \in elts\ \gamma\ \beta' \in elts\ \gamma\ Ord\ \alpha\ Ord\ \alpha'\ Ord\ \gamma$
shows $\gamma * \alpha + \beta \in elts\ (\gamma * \alpha' + \beta') \longleftrightarrow \alpha \in elts\ \alpha' \vee \alpha = \alpha' \wedge \beta \in elts\ \beta'$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *L*: *?lhs*
show *?rhs*
proof (*cases* $\alpha \in elts\ \alpha'$)
case *False*
with *assms* **have** $\alpha = \alpha'$
by (*meson L Ord-linear Ord-mult Ord-trans add-mult-less not-add-mem-right*)
then show *?thesis*
using *L less-V-def* **by** *auto*
qed *auto*
next
assume *R*: *?rhs*
then show *?lhs*
proof
assume $\alpha \in elts\ \alpha'$
then obtain δ **where** $\alpha' = \alpha + \delta$
by (*metis OrdmemD assms*(*3*) *assms*(*4*) *le-Ord-diff less-V-def*)
show *?lhs*
using *assms*
by (*meson* ‹$\alpha \in elts\ \alpha'$› *add-le-cancel-left0 add-mult-less vsubsetD*)
next
assume $\alpha = \alpha' \wedge \beta \in elts\ \beta'$
then show *?lhs*
using *less-V-def* **by** *auto*
qed
qed

lemma *vcard-mult*: $vcard\ (x * y) = vcard\ x \otimes vcard\ y$
proof −
have *1*: $elts\ (lift\ (x * u)\ x) \approx elts\ x$ **if** $u \in elts\ y$ **for** u
by (*metis cardinal-eqpoll eqpoll-sym eqpoll-trans card-lift*)
have *2*: *pairwise* $(\lambda u\ u'.\ disjnt\ (elts\ (lift\ (x * u)\ x))\ (elts\ (lift\ (x * u')\ x)))\ (elts\ y)$
by (*simp add*: *pairwise-def disjnt-def*) (*metis V-disjoint-iff lift-mult-disjoint*)
have $x * y = (SUP\ u{\in}elts\ y.\ lift\ (x * u)\ x)$

using *mult* **by** *blast*
then have *elts* $(x * y) \approx (\bigcup u{\in}elts\ y.\ elts\ (lift\ (x * u)\ x))$
by *simp*
also have $\ldots \approx$ *elts* $y \times$ *elts* x
using *Union-eqpoll-Times* [*OF 1 2*] **.**
also have $\ldots \approx$ *elts* $x \times$ *elts* y
by (*simp add*: *times-commute-eqpoll*)
also have $\ldots \approx$ *elts* (*vcard* x) $\times$ *elts* (*vcard* y)
using *cardinal-eqpoll eqpoll-sym times-eqpoll-cong* **by** *blast*
also have $\ldots \approx$ *elts* (*vcard* $x \otimes$ *vcard* y)
by (*simp add*: *cmult-def elts-vcard-VSigma-eqpoll eqpoll-sym*)
finally have *elts* $(x * y) \approx$ *elts* (*vcard* $x \otimes$ *vcard* y) **.**
then show *?thesis*
by (*metis cadd-cmult-distrib cadd-def cardinal-cong cardinal-idem vsum-0-eqpoll*)
qed

proposition *TC-mult*: $TC(x * y) = (SUP\ r \in elts\ (TC\ x).\ SUP\ u \in elts\ (TC\ y).\ set\{x * u + r\})$
proof (*cases* $x = 0$)
case *False*
have $*$: $TC(x * y) = (SUP\ u \in elts\ (TC\ y).\ lift\ (x * u)\ (TC\ x))$ **for** y
proof (*induction y rule*: *eps-induct*)
case (*step y*)
have $TC(x * y) = (SUP\ u \in elts\ y.\ TC\ (lift\ (x * u)\ x))$
by (*simp add*: *mult* [*of x y*] *TC-Sup-distrib image-image*)
also have $\ldots = (SUP\ u \in elts\ y.\ TC(x * u) \sqcup lift\ (x * u)\ (TC\ x))$
by (*simp add*: *TC-lift False*)
also have $\ldots = (SUP\ u \in elts\ y.\ (SUP\ z \in elts\ (TC\ u).\ lift\ (x * z)\ (TC\ x)) \sqcup lift\ (x * u)\ (TC\ x))$
by (*simp add*: *step*)
also have $\ldots = (SUP\ u \in elts\ (TC\ y).\ lift\ (x * u)\ (TC\ x))$
by (*auto simp*: *TC′* [*of y*] *image-Un Sup-Un-distrib TC-Sup-distrib cSUP-UNION SUP-sup-distrib*)
finally show *?case* **.**
qed
show *?thesis*
by (*force simp*: $*$ *lift-def*)
qed *auto*

corollary *vcard-TC-mult*: *vcard* $(TC(x * y))$ = *vcard* $(TC\ x) \otimes$ *vcard* $(TC\ y)$
proof −
have $(\bigcup u{\in}elts\ (TC\ x).\ \bigcup v{\in}elts\ (TC\ y).\ \{x * v + u\}) = (\bigcup u{\in}elts\ (TC\ x).\ (\lambda v.\ x * v + u)\ `\ elts\ (TC\ y))$
by (*simp add*: *UNION-singleton-eq-range*)
also have $\ldots \approx (\bigcup x{\in}elts\ (TC\ x).\ elts\ (lift\ (TC\ y * x)\ (TC\ y)))$
proof (*rule UN-eqpoll-UN*)
show $(\lambda v.\ x * v + u)\ `\ elts\ (TC\ y) \approx elts\ (lift\ (TC\ y * u)\ (TC\ y))$
if $u \in elts\ (TC\ x)$ **for** u

proof −
have *inj-on* (λ*v*. *x* ∗ *v* + *u*) (*elts* (*TC y*))
by (*meson inj-onI less-TC-def mult-cancellation-lemma that*)
then have (λ*v*. *x* ∗ *v* + *u*) ‘ *elts* (*TC y*) ≈ *elts* (*TC y*)
by (*rule inj-on-image-eqpoll-self*)
also have . . . ≈ *elts* (*lift* (*TC y* ∗ *u*) (*TC y*))
by (*simp add*: *eqpoll-lift eqpoll-sym*)
finally show *?thesis* **.**
qed
show *pairwise* (λ*u ya*. *disjnt* ((λ*v*. *x* ∗ *v* + *u*) ‘ *elts* (*TC y*)) ((λ*v*. *x* ∗ *v* + *ya*) ‘ *elts* (*TC y*))) (*elts* (*TC x*))
apply (*auto simp*: *pairwise-def disjnt-def*)
using *less-TC-def mult-cancellation-lemma* **by** *blast*
show *pairwise* (λ*u ya*. *disjnt* (*elts* (*lift* (*TC y* ∗ *u*) (*TC y*))) (*elts* (*lift* (*TC y* ∗ *ya*) (*TC y*)))) (*elts* (*TC x*))
apply (*auto simp*: *pairwise-def disjnt-def*)
by (*metis Int-iff V-disjoint-iff empty-iff lift-mult-disjoint*)
qed
also have . . . = *elts* (*TC y* ∗ *TC x*)
by (*metis elts-Sup image-image mult replacement small-elts*)
finally have (⋃ *u*∈*elts* (*TC x*). ⋃ *v*∈*elts* (*TC y*). {*x* ∗ *v* + *u*}) ≈ *elts* (*TC y* ∗ *TC x*) **.**
then show *?thesis*
apply (*subst cmult-commute*)
by (*simp add*: *TC-mult cardinal-cong flip*: *vcard-mult*)
qed

lemma *countable-mult*:
assumes *countable* (*elts A*) *countable* (*elts B*)
shows *countable* (*elts* (*A*∗*B*))
proof −
have *vcard A* ≤ ℵ*0 vcard B* ≤ ℵ*0*
using *assms countable-iff-le-Aleph0* **by** *blast*+
then have *vcard* (*A*∗*B*) ≤ ℵ*0*
unfolding *vcard-mult*
by (*metis InfCard-csquare-eq cmult-le-mono Aleph-0 Card-ω InfCard-def order-refl*)
then show *?thesis*
by (*simp add*: *countable-iff-le-Aleph0*)
qed

3.4 Ordertype properties

lemma *ordertype-image-plus*:
assumes *Ord* α
shows *ordertype* ((+) *u* ‘ *elts* α) *VWF* = α
proof (*subst ordertype-VWF-eq-iff*)
have *1*: (*u* + *x*, *u* + *y*) ∈ *VWF* **if** *x* ∈ *elts* α *y* ∈ *elts* α *x* < *y* **for** *x y*
using *that*

by (*meson Ord-in-Ord Ord-mem-iff-lt add-mem-right-cancel assms mem-imp-VWF*)
then have *2*: $x < y$
if $x \in elts\ \alpha\ y \in elts\ \alpha\ (u + x,\ u + y) \in VWF$ **for** $x\ y$
using *that* **by** (*metis Ord-in-Ord Ord-linear-lt VWF-asym assms*)
show $\exists f.\ bij\text{-}betw\ f\ ((+)\ u\ `\ elts\ \alpha)\ (elts\ \alpha) \wedge (\forall x{\in}(+)\ u\ `\ elts\ \alpha.\ \forall y{\in}(+)\ u\ `\ elts\ \alpha.\ (f\ x < f\ y) = ((x,\ y) \in VWF))$
using *1 2* **unfolding** *bij-betw-def inj-on-def*
by (*rule-tac* $x{=}\lambda x.\ odiff\ x\ u$ **in** *exI*) (*auto simp*: *image-iff*)
qed (*use assms* **in** *auto*)

lemma *ordertype-diff*:
assumes $\beta + \delta = \alpha$ **and** α: $\delta \in elts\ \alpha\ Ord\ \alpha$
shows $ordertype\ (elts\ \alpha - elts\ \beta)\ VWF = \delta$
proof −
have $*$: $elts\ \alpha - elts\ \beta = ((+)\beta)\ `\ elts\ \delta$
proof
show $elts\ \alpha - elts\ \beta \subseteq (+)\ \beta\ `\ elts\ \delta$
by *clarsimp* (*metis assms*(*1*) *image-iff mem-plus-V-E*)
show $(+)\ \beta\ `\ elts\ \delta \subseteq elts\ \alpha - elts\ \beta$
using *assms*(*1*) *not-add-mem-right* **by** *force*
qed
have $ordertype\ ((+)\ \beta\ `\ elts\ \delta)\ VWF = \delta$
proof (*subst ordertype-VWF-inc-eq*)
show $elts\ \delta \subseteq ON\ ordertype\ (elts\ \delta)\ VWF = \delta$
using α *elts-subset-ON ordertype-eq-Ord* **by** *blast*+
qed (*use* $*$ *assms elts-subset-ON* **in** *auto*)
then show *?thesis*
by (*simp add*: $*$)
qed

lemma *ordertype-interval-eq*:
assumes α: $Ord\ \alpha$ **and** β: $Ord\ \beta$
shows $ordertype\ (\{\alpha\ ..{<}\ \alpha{+}\beta\} \cap ON)\ VWF = \beta$
proof −
have ON: $(+)\ \alpha\ `\ elts\ \beta \subseteq ON$
using *assms Ord-add Ord-in-Ord* **by** *blast*
have $(\{\alpha\ ..{<}\ \alpha{+}\beta\} \cap ON) = (+)\ \alpha\ `\ elts\ \beta$
using *assms*
apply (*simp add*: *image-def set-eq-iff*)
by (*metis add-less-cancel-left Ord-add Ord-in-Ord Ord-linear2 Ord-mem-iff-lt le-Ord-diff not-add-less-right*)
moreover have $ordertype\ (elts\ \beta)\ VWF = ordertype\ ((+)\ \alpha\ `\ elts\ \beta)\ VWF$
using $ON\ \beta$ *elts-subset-ON ordertype-VWF-inc-eq* **by** *auto*
ultimately show *?thesis*
using β **by** *auto*
qed

lemma *ordertype-Times*:
assumes *small A small B* **and** r: *wf r trans r total-on A r* **and** s: *wf s trans s*

total-on B s
shows *ordertype* $(A \times B)$ $(r <*lex*> s) =$ *ordertype B s* $*$ *ordertype A r* (**is** - = *?β* $*$ *?α*)
proof (*subst ordertype-eq-iff*)
show *Ord* (*?β* $*$ *?α*)
by (*intro wf-Ord-ordertype Ord-mult r s*; *simp*)
define *f* **where** $f \equiv \lambda(x,y).$ *?β* $*$ *ordermap A r x* + (*ordermap B s y*)
show $\exists f.$ *bij-betw f* $(A \times B)$ (*elts* (*?β* $*$ *?α*)) $\wedge$ $(\forall x{\in}A \times B.\ \forall y{\in}A \times B.\ (f\ x < f\ y) = ((x,\ y) \in (r <*lex*> s)))$
unfolding *bij-betw-def*
proof (*intro exI conjI strip*)
show *inj-on f* $(A \times B)$
proof (*clarsimp simp*: *f-def inj-on-def*)
fix *x y x′ y′*
assume $x \in A$ $y \in B$ $x' \in A$ $y' \in B$
and *eq*: *?β* $*$ *ordermap A r x* + *ordermap B s y* = *?β* $*$ *ordermap A r x′* + *ordermap B s y′*
have *ordermap A r x* = *ordermap A r x′* $\wedge$
ordermap B s y = *ordermap B s y′*
proof (*rule mult-cancellation-lemma* [*OF eq*])
show *ordermap B s y* $\sqsubset$ *?β*
using *ordermap-in-ordertype* [*OF* ‹*y* $\in$ *B*›, *of s*] *less-TC-iff* ‹*small B*› **by** *blast*
show *ordermap B s y′* $\sqsubset$ *?β*
using *ordermap-in-ordertype* [*OF* ‹*y′* $\in$ *B*›, *of s*] *less-TC-iff* ‹*small B*› **by** *blast*
qed
then show $x = x' \wedge y = y'$
using ‹*x* $\in$ *A*› ‹*x′* $\in$ *A*› ‹*y* $\in$ *B*› ‹*y′* $\in$ *B*› *r s* ‹*small A*› ‹*small B*› **by** *auto*
qed
show *f* ‘ $(A \times B)$ = *elts* (*?β* $*$ *?α*) (**is** *?lhs* = *?rhs*)
proof
show *f* ‘ $(A \times B) \subseteq$ *elts* (*?β* $*$ *?α*)
apply (*auto simp*: *f-def add-mult-less ordermap-in-ordertype wf-Ord-ordertype r s*)
by (*simp add*: *add-mult-less assms ordermap-in-ordertype wf-Ord-ordertype*)
show *elts* (*?β* $*$ *?α*) $\subseteq$ *f* ‘ $(A \times B)$
proof (*clarsimp simp*: *f-def image-iff elim* !: *elts-multE split*: *prod.split*)
fix *u v*
assume *u*: $u \in$ *elts* (*?β*) **and** *v*: $v \in$ *elts ?α*
have *inv-into B* (*ordermap B s*) $u \in B$
by (*simp add*: *inv-into-ordermap u*)
moreover have *inv-into A* (*ordermap A r*) $v \in A$
by (*simp add*: *inv-into-ordermap v*)
ultimately show $\exists x{\in}A.\ \exists y{\in}B.$ *?β* $*$ *v* + *u* = *?β* $*$ *ordermap A r x* + *ordermap B s y*
by (*metis* ‹*small A*› ‹*small B*› *bij-betw-inv-into-right ordermap-bij r*(*1*) *r*(*3*) *s*(*1*) *s*(*3*) *u v*)
qed

```
    qed
  next
    fix p q
    assume p ∈ A × B and q ∈ A × B
    then obtain u v x y where §: p = (u,v) u ∈ A v ∈ B q = (x,y) x ∈ A y ∈ B
      by blast
    show ((f p) < f q) = ((p, q) ∈ (r <*lex*> s))
    proof
      assume f p < f q
      with § assms have (u, x) ∈ r ∨ u=x ∧ (v, y) ∈ s
        apply (simp add: f-def)
      by (metis Ord-add Ord-add-mult-iff Ord-mem-iff-lt Ord-mult wf-Ord-ordermap
converse-ordermap-mono
          ordermap-eq-iff ordermap-in-ordertype wf-Ord-ordertype)
      then show (p,q) ∈ (r <*lex*> s)
        by (simp add: §)
    next
      assume (p,q) ∈ (r <*lex*> s)
      then have (u, x) ∈ r ∨ u = x ∧ (v, y) ∈ s
        by (simp add: §)
      then show f p < f q
      proof
        assume ux: (u, x) ∈ r
        have oo: ⋀x. Ord (ordermap A r x) ⋀y. Ord (ordermap B s y)
          by (simp-all add: r s)
        show f p < f q
        proof (clarsimp simp: f-def split: prod.split)
          fix a b a′ b′
          assume p = (a, b) and q = (a′, b′)
          then have ?β * ordermap A r a + ordermap B s b < ?β * ordermap A r
a′
            using ux assms §
           by (metis Ord-mult wf-Ord-ordermap OrdmemD Pair-inject add-mult-less
ordermap-in-ordertype ordermap-mono wf-Ord-ordertype)
          also have ... ≤ ?β * ordermap A r a′ + ordermap B s b′
            by simp
          finally show ?β * ordermap A r a + ordermap B s b < ?β * ordermap A
r a′ + ordermap B s b′ .
        qed
      next
        assume u = x ∧ (v, y) ∈ s
        then show f p < f q
              using § assms by (fastforce simp: f-def split: prod.split intro: or-
dermap-mono-less)
      qed
    qed
  qed
qed (use assms small-Times in auto)
```

end

4 Exponentiation of ordinals

theory *Ordinal-Exp*
imports *Kirby*

begin

Source: Schlöder, Julian. Ordinal Arithmetic; available online at http://www.math.uni-bonn.de/ag/logik/teaching/2012WS/Set%20theory/oa.pdf

definition *oexp* :: $[V, V] \Rightarrow V$ (**infixr** $\uparrow$ *80*)
where *oexp a b* $\equiv$ *transrec* ($\lambda f\ x$. *if x=0 then 1*
else if Limit x then if a=0 then 0 else SUP $\xi \in$ *elts x*. *f* ξ
else f ($\bigsqcup$(*elts x*)) $*$ *a*) *b*

0 $\uparrow$ ω = *1* if we don't make a special case for Limit ordinals and zero

lemma *oexp-0-right* [*simp*]: $\alpha\uparrow 0 = 1$
by (*simp add*: *def-transrec* [*OF oexp-def*])

lemma *oexp-succ* [*simp*]: *Ord* $\beta \Longrightarrow \alpha\uparrow(succ\ \beta) = \alpha\uparrow\beta * \alpha$
by (*simp add*: *def-transrec* [*OF oexp-def*])

lemma *oexp-Limit*: *Limit* $\beta \Longrightarrow \alpha\uparrow\beta$ = (*if* α=*0 then 0 else SUP* $\xi \in$ *elts* β. $\alpha\uparrow\xi$)
by (*auto simp*: *def-transrec* [*OF oexp-def*, *of - β*])

lemma *oexp-1-right* [*simp*]: $\alpha\uparrow 1 = \alpha$
using *one-V-def oexp-succ* **by** *fastforce*

lemma *oexp-1* [*simp*]: *Ord* $\alpha \Longrightarrow 1\uparrow\alpha = 1$
by (*induction rule*: *Ord-induct3*) (*use Limit-def oexp-Limit* **in** *auto*)

lemma *oexp-0* [*simp*]: *Ord* $\alpha \Longrightarrow 0\uparrow\alpha$ = (*if* $\alpha = 0$ *then 1 else 0*)
by (*induction rule*: *Ord-induct3*) (*use Limit-def oexp-Limit* **in** *auto*)

lemma *oexp-eq-0-iff* [*simp*]:
assumes *Ord* β **shows** $\alpha\uparrow\beta = 0 \longleftrightarrow \alpha$=*0* $\wedge$ $\beta\neq$*0*
using ‹*Ord* β›
proof (*induction rule*: *Ord-induct3*)
case (*Limit* μ)
then show *?case*
using *Limit-def oexp-Limit* **by** *auto*
qed *auto*

lemma *oexp-gt-0-iff* [*simp*]:
assumes *Ord* β **shows** $\alpha\uparrow\beta > 0 \longleftrightarrow \alpha$>*0* $\vee$ β=*0*
by (*simp add*: *assms less-V-def*)

lemma *ord-of-nat-oexp*: *ord-of-nat* $(m\widehat{\ }n) = ord\text{-}of\text{-}nat\ m{\uparrow}ord\text{-}of\text{-}nat\ n$
proof (*induction n*)
  case (*Suc n*)
  then show *?case*
    by (*simp add*: *mult.commute* [*of m*]) (*simp add*: *ord-of-nat-mult*)
qed *auto*

lemma *omega-closed-oexp* [*intro*]:
  assumes $\alpha \in elts\ \omega\ \beta \in elts\ \omega$ **shows** $\alpha{\uparrow}\beta \in elts\ \omega$
proof −
  obtain *m n* **where** $\alpha = ord\text{-}of\text{-}nat\ m\ \beta = ord\text{-}of\text{-}nat\ n$
    using *assms elts-ω* **by** *auto*
  then have $\alpha{\uparrow}\beta = ord\text{-}of\text{-}nat\ (m\widehat{\ }n)$
    by (*simp add*: *ord-of-nat-oexp*)
  then show *?thesis*
    by (*simp add*: *ω-def*)
qed

lemma *Ord-oexp* [*simp*]:
  assumes *Ord* α *Ord* β **shows** *Ord* $(\alpha{\uparrow}\beta)$
  using ‹*Ord* β›
proof (*induction rule*: *Ord-induct3*)
  case (*Limit* α)
  then show *?case*
    by (*auto simp*: *oexp-Limit image-iff intro*: *Ord-Sup*)
qed (*auto intro*: *Ord-mult assms*)

Lemma 3.19

lemma *le-oexp*:
  assumes *Ord* α *Ord* β $\beta \neq 0$ **shows** $\alpha \leq \alpha{\uparrow}\beta$
  using ‹*Ord* β› ‹$\beta \neq 0$›
proof (*induction rule*: *Ord-induct3*)
  case (*succ* β)
  then show *?case*
    by *simp* (*metis* ‹*Ord* α› *le-0 le-mult mult.left-neutral oexp-0-right order-refl order-trans*)
next
  case (*Limit* μ)
  then show *?case*
    by (*metis Limit-def Limit-eq-Sup-self ZFC-in-HOL.Sup-upper eq-iff image-eqI image-ident oexp-1-right oexp-Limit replacement small-elts one-V-def*)
qed *auto*

Lemma 3.20

lemma *le-oexp′*:
  assumes *Ord* α $1 < \alpha$ *Ord* β **shows** $\beta \leq \alpha{\uparrow}\beta$
proof (*cases* $\beta = 0$)

```
  case True
  then show ?thesis
    by auto
next
  case False
  show ?thesis
    using ‹Ord β›
  proof (induction rule: Ord-induct3)
    case 0
    then show ?case
      by auto
  next
    case (succ γ)
    then have α↑γ * 1 < α↑γ * α
      using ‹Ord α› ‹1 < α›
       by (metis le-mult less-V-def mult.right-neutral mult-cancellation not-less-0
oexp-eq-0-iff succ.hyps)
    then have  γ < α↑succ γ
      using succ.IH succ.hyps by auto
    then show ?case
      using False ‹Ord α› ‹1 < α› succ
     by (metis Ord-mem-iff-lt Ord-oexp Ord-succ elts-succ insert-subset less-eq-V-def
less-imp-le)
  next
    case (Limit μ)
    with False ‹1 < α› show ?case
      by (force simp: Limit-def oexp-Limit intro: elts-succ)
  qed
qed


lemma oexp-Limit-le:
  assumes β < γ Limit γ Ord β α > 0 shows α↑β ≤ α↑γ
proof −
  have Ord γ
    using Limit-def assms(2) by blast
  with assms show ?thesis
    using Ord-mem-iff-lt ZFC-in-HOL.Sup-upper oexp-Limit by auto
qed

proposition oexp-less:
  assumes β: β ∈ elts γ and Ord γ and α: α > 1 Ord α shows α↑β < α↑γ
proof −
  obtain β < γ Ord β
    using Ord-in-Ord OrdmemD assms by auto
  have gt0: α↑β > 0
    using ‹Ord β› α dual-order.order-iff-strict by auto
  show ?thesis
    using ‹Ord γ› β
```

```
  proof (induction rule: Ord-induct3)
    case 0
    then show ?case
      by auto
  next
    case (succ δ)
    then consider β = δ | β < δ
      using OrdmemD elts-succ by blast
    then show ?case
    proof cases
      case 1
      then have (α↑β) * 1 < (α↑δ) * α
        using Ord-1 Ord-oexp α gt0 mult-cancel-less-iff succ.hyps by metis
      then show ?thesis
        by (simp add: succ.hyps)
    next
      case 2
      then have (α↑δ) * 1 < (α↑δ) * α
     by (meson Ord-1 Ord-mem-iff-lt Ord-oexp ‹Ord β› α gt0 less-trans mult-cancel-less-iff
succ)
      with 2 show ?thesis
        using Ord-mem-iff-lt ‹Ord β› succ by auto
    qed
  next
    case (Limit γ)
    then obtain Ord γ succ β < γ
      using Limit-def Ord-in-Ord OrdmemD assms by auto
    have α↑β = (α↑β) * 1
      by simp
    also have ... < (α↑β) * α
      using Ord-oexp ‹Ord β› assms gt0 mult-cancel-less-iff by blast
    also have ... = α↑succ β
      by (simp add: ‹Ord β›)
    also have ... ≤ (SUP ξ ∈ elts γ. α↑ξ)
    proof −
      have succ β ∈ elts γ
        using Limit.hyps Limit.prems Limit-def by auto
      then show ?thesis
        by (simp add: ZFC-in-HOL.Sup-upper)
    qed
    finally
    have α↑β < (SUP ξ ∈ elts γ. α↑ξ) .
    then show ?case
      using Limit.hyps oexp-Limit ‹α > 1› by auto
  qed
qed

corollary oexp-less-iff:
  assumes α > 0 Ord α Ord β Ord γ shows α↑β < α↑γ ⟷ β ∈ elts γ ∧ α > 1
```

```
proof safe
  show β ∈ elts γ 1 < α
    if α↑β < α↑γ
  proof −
    show α > 1
    proof (rule ccontr)
      assume ¬ α > 1
      then consider α=0 | α=1
        using ‹Ord α› less-V-def mem-0-Ord by fastforce
      then show False
        by cases (use that ‹α > 0› ‹Ord β› ‹Ord γ› in ‹auto split: if-split-asm›)
    qed
    show β: β ∈ elts γ
    proof (rule ccontr)
      assume β ∉ elts γ
      then have γ ≤ β
        by (meson Ord-linear-le Ord-mem-iff-lt assms less-le-not-le)
      then consider γ = β | γ < β
        using less-V-def by blast
      then show False
      proof cases
        case 1
        then show ?thesis
          using that by blast
      next
        case 2
        with ‹α > 1› have α↑γ < α↑β
          by (simp add: Ord-mem-iff-lt assms oexp-less)
        with that show ?thesis
          by auto
      qed
    qed
  qed
  show α↑β < α↑γ if β ∈ elts γ 1 < α
    using that by (simp add: assms oexp-less)
qed

lemma ω-oexp-iff [simp]: ⟦Ord α; Ord β⟧ ⟹ ω↑α = ω↑β ⟷ α=β
  by (metis Ord-ω Ord-linear ω-gt1 less-irrefl oexp-less)

lemma Limit-oexp:
  assumes Limit γ Ord α α > 1 shows Limit (α↑γ)
  unfolding Limit-def
proof safe
  show Oαγ: Ord (α↑γ)
    using Limit-def Ord-oexp ‹Limit γ› assms(2) by blast
  show 0: 0 ∈ elts (α↑γ)
    using Limit-def oexp-Limit ‹Limit γ› ‹α > 1› by fastforce
  have Ord γ
```

```
    using Limit-def ‹Limit γ› by blast
  fix x
  assume x: x ∈ elts (α↑γ)
  with ‹Limit γ› ‹α > 1›
  obtain β where β < γ Ord β Ord x and xβ: x ∈ elts (α↑β)
    apply (simp add: oexp-Limit split: if-split-asm)
    using Ord-in-Ord OrdmemD ‹Ord γ› Oαγ x by blast
  then have Oαβ: Ord (α↑β)
    using Ord-oexp assms(2) by blast
  have β ∈ elts γ
    by (simp add: Ord-mem-iff-lt ‹Ord β› ‹Ord γ› ‹β < γ›)
  moreover have α ≠ 0
    using ‹α > 1› by blast
  ultimately have αβγ: α↑β ≤ α↑γ
    by (simp add: Sup-upper oexp-Limit ‹Limit γ›)
  have succ x ≤ α↑β
    by (simp add: OrdmemD Oαβ ‹Ord x› succ-le-iff xβ)
  then consider succ x < α↑β | succ x = α↑β
    using le-neq-trans by blast
  then show succ x ∈ elts (α↑γ)
  proof cases
    case 1
    with αβγ show ?thesis
      using Oαβ Ord-mem-iff-lt ‹Ord x› by blast
  next
    case 2
    then have succ β < γ
      using Limit-def OrdmemD ‹β ∈ elts γ› assms(1) by auto
    have ge1: 1 ≤ α↑β
      by (metis 2 Ord-0 ‹Ord x› le-0 le-succ-iff one-V-def)
    have succ x < succ (α↑β)
      using 2 Oαβ succ-le-iff by auto
    also have ... ≤ (α↑β) + (α↑β)
      using ge1 by (simp add: succ-eq-add1)
    also have ... = (α↑β) * succ (succ 0)
      by (simp add: mult-succ)
    also have ... ≤ (α↑β) * α
      using Oαβ Ord-succ assms(2) assms(3) one-V-def succ-le-iff by auto
    also have ... = α↑succ β
      by (simp add: ‹Ord β›)
    also have ... ≤ α↑γ
      by (meson Limit-def ‹β ∈ elts γ› assms dual-order.order-iff-strict oexp-less)
  finally show ?thesis
    by (simp add: 2 Oαβ Oαγ Ord-mem-iff-lt)
  qed
qed
```

lemma *oexp-mono*:
assumes α: *Ord* α $\alpha \neq 0$ **and** β: *Ord* β $\gamma \sqsubseteq \beta$ **shows** $\alpha\uparrow\gamma \leq \alpha\uparrow\beta$
using β
proof (*induction rule*: *Ord-induct3*)
case *0*
then show *?case*
by *simp*
next
case (*succ* β)
with α *le-mult* **show** *?case*
by (*auto simp*: *le-TC-succ*)
next
case (*Limit* μ)
then have $\alpha\uparrow\gamma \leq \bigsqcup$ ((↑) α ‘ *elts* μ)
using *Limit.hyps Ord-less-TC-mem* ‹$\alpha \neq 0$› *le-TC-def* **by** (*auto simp*: *oexp-Limit Limit-def*)
then show *?case*
using α **by** (*simp add*: *oexp-Limit Limit.hyps*)
qed

lemma *oexp-mono-le*:
assumes $\gamma \leq \beta$ $\alpha \neq 0$ *Ord* α *Ord* β *Ord* γ **shows** $\alpha\uparrow\gamma \leq \alpha\uparrow\beta$
by (*simp add*: *assms oexp-mono vle2 vle-iff-le-Ord*)

lemma *oexp-sup*:
assumes $\alpha \neq 0$ *Ord* α *Ord* β *Ord* γ **shows** $\alpha\uparrow(\beta \sqcup \gamma) = \alpha\uparrow\beta \sqcup \alpha\uparrow\gamma$
by (*metis Ord-linear-le assms oexp-mono-le sup.absorb2 sup.orderE*)

lemma *oexp-Sup*:
assumes α: $\alpha \neq 0$ *Ord* α **and** X: $X \subseteq$ *ON small X X* $\neq \{\}$ **shows** $\alpha\uparrow\bigsqcup X = \bigsqcup$ ((↑) α ‘ X)
proof (*rule order-antisym*)
show $\bigsqcup$ ((↑) α ‘ X) $\leq \alpha\uparrow\bigsqcup X$
by (*metis ON-imp-Ord Ord-Sup ZFC-in-HOL.Sup-upper assms cSUP-least oexp-mono-le*)
next
have *Ord* (*Sup X*)
using *Ord-Sup X* **by** *auto*
then show $\alpha\uparrow\bigsqcup X \leq \bigsqcup$ ((↑) α ‘ X)
proof (*cases rule*: *Ord-cases*)
case *0*
then show *?thesis*
using *X dual-order.antisym* **by** *fastforce*
next
case (*succ* β)
then show *?thesis*
using *ZFC-in-HOL.Sup-upper X succ-in-Sup-Ord* **by** *auto*
next
case *limit*

```
    show ?thesis
    proof (clarsimp simp: assms oexp-Limit limit)
      fix x y z
      assume x: x ∈ elts (α ↑ y) and z ∈ X y ∈ elts z
      then have α ↑ y ≤ α ↑ z
      by (meson ON-imp-Ord Ord-in-Ord OrdmemD α ‹X ⊆ ON› le-less oexp-mono-le)
      with x have x ∈ elts (α ↑ z) by blast
      then show ∃ u∈X. x ∈ elts (α ↑ u)
        using ‹z ∈ X› by blast
    qed
  qed
qed


lemma omega-le-Limit:
  assumes Limit μ shows ω ≤ μ
proof
  fix ϱ
  assume ϱ ∈ elts ω
  then obtain n where ϱ = ord-of-nat n
    using elts-ω by auto
  have ord-of-nat n ∈ elts μ
    by (induction n) (use Limit-def assms in auto)
  then show ϱ ∈ elts μ
    using ‹ϱ = ord-of-nat n› by auto
qed

lemma finite-omega-power [simp]:
  assumes 1 < n n ∈ elts ω shows n↑ω = ω
proof (rule order-antisym)
  have ⋃ ((↑) (ord-of-nat k) ‘ elts ω) ≤ ω for k
  proof (induction k)
    case 0
    then show ?case
      by auto
  next
    case (Suc k)
    then show ?case
      by (metis Ord-ω OrdmemD Sup-eq-0-iff ZFC-in-HOL.SUP-le-iff le-0 le-less
omega-closed-oexp ord-of-nat-ω)
  qed
  then show n↑ω ≤ ω
    using assms
    by (simp add: elts-ω oexp-Limit) metis
  show ω ≤ n↑ω
    using Ord-in-Ord assms le-oexp′ by blast
qed
```

proposition *oexp-add*:
assumes *Ord* α *Ord* β *Ord* γ **shows** $\alpha\uparrow(\beta + \gamma) = \alpha\uparrow\beta * \alpha\uparrow\gamma$
proof (*cases* ‹$\alpha = 0$›)
case *True*
then show *?thesis*
using *assms* **by** *simp*
next
case *False*
show *?thesis*
using ‹*Ord* γ›
proof (*induction rule*: *Ord-induct3*)
case *0*
then show *?case*
by *auto*
next
case (*succ* ξ)
then show *?case*
using ‹*Ord* β› **by** (*auto simp*: *plus-V-succ-right mult.assoc*)
next
case (*Limit* μ)
have $\alpha\uparrow(\beta + (SUP\ \xi{\in}elts\ \mu.\ \xi)) = (SUP\ \xi{\in}elts\ (\beta + \mu).\ \alpha\uparrow\xi)$
by (*simp add*: *Limit.hyps oexp-Limit assms False*)
also have $\ldots = (SUP\ \xi \in \{\xi.\ Ord\ \xi \wedge \beta + \xi < \beta + \mu\}.\ \alpha\uparrow(\beta + \xi))$
proof (*rule Sup-eq-Sup*)
show $(\lambda\xi.\ \alpha\uparrow(\beta + \xi))$ ‘ $\{\xi.\ Ord\ \xi \wedge \beta + \xi < \beta + \mu\} \subseteq (\uparrow)\ \alpha$ ‘ *elts* $(\beta + \mu)$
using *Limit.hyps Limit-def Ord-mem-iff-lt imageI* **by** *blast*
fix x
assume $x \in (\uparrow)\ \alpha$ ‘ *elts* $(\beta + \mu)$
then obtain ξ **where** ξ: $\xi \in elts\ (\beta + \mu)$ **and** x: $x = \alpha\uparrow\xi$
by *auto*
have $\exists\gamma.\ Ord\ \gamma \wedge \gamma < \mu \wedge \alpha\uparrow\xi \leq \alpha\uparrow(\beta + \gamma)$
proof (*rule mem-plus-V-E* [*OF* ξ])
assume $\xi \in elts\ \beta$
then have $\alpha\uparrow\xi \leq \alpha\uparrow\beta$
by (*meson arg-subset-TC assms False le-TC-def less-TC-def oexp-mono vsubsetD*)
with *zero-less-Limit* [*OF* ‹*Limit* μ›]
show $\exists\gamma.\ Ord\ \gamma \wedge \gamma < \mu \wedge \alpha\uparrow\xi \leq \alpha\uparrow(\beta + \gamma)$
by *force*
next
fix δ
assume $\delta \in elts\ \mu$ **and** $\xi = \beta + \delta$
have *Ord* δ
using *Limit.hyps Limit-def Ord-in-Ord* ‹$\delta \in elts\ \mu$› **by** *blast*
moreover have $\delta < \mu$
using *Limit.hyps Limit-def OrdmemD* ‹$\delta \in elts\ \mu$› **by** *auto*
ultimately show $\exists\gamma.\ Ord\ \gamma \wedge \gamma < \mu \wedge \alpha\uparrow\xi \leq \alpha\uparrow(\beta + \gamma)$
using ‹$\xi = \beta + \delta$› **by** *blast*
qed

```
    then show ∃ y∈(λξ. α↑(β + ξ)) ` {ξ. Ord ξ ∧ β + ξ < β + μ}. x ≤ y
      using x by auto
  qed auto
  also have ... = (SUP ξ∈elts μ. α↑(β + ξ))
    using ‹Limit μ›
    by (simp add: Ord-Collect-lt Limit-def)
  also have ... = (SUP ξ∈elts μ. α↑β * α↑ξ)
    using Limit.IH by auto
  also have ... = α↑β * α↑(SUP ξ∈elts μ. ξ)
    using ‹α ≠ 0› Limit.hyps
    by (simp add: image-image oexp-Limit mult-Sup-distrib)
  finally show ?case .
 qed
qed

proposition oexp-mult:
 assumes Ord α Ord β Ord γ shows α↑(β * γ) = (α↑β)↑γ
proof (cases α = 0 ∨ β = 0)
 case True
 then show ?thesis
  by (auto simp: ‹Ord β› ‹Ord γ›)
next
 case False
 show ?thesis
  using ‹Ord γ›
 proof (induction rule: Ord-induct3)
  case 0
  then show ?case
   by auto
 next
  case succ
  then show ?case
   using assms by (auto simp: mult-succ oexp-add)
 next
  case (Limit μ)
  have Lim: Limit (⨆ ((*) β ` elts μ))
   unfolding Limit-def
  proof (intro conjI allI impI)
   show Ord (⨆ ((*) β ` elts μ))
    using Limit.hyps Limit-def Ord-in-Ord ‹Ord β› by (auto intro: Ord-Sup)
   have succ 0 ∈ elts μ
    using Limit.hyps Limit-def by blast
   then show 0 ∈ elts (⨆ ((*) β ` elts μ))
    using False ‹Ord β› mem-0-Ord by force
   show succ y ∈ elts (⨆ ((*) β ` elts μ))
    if y ∈ elts (⨆ ((*) β ` elts μ)) for y
    using that False Limit.hyps
    apply (clarsimp simp: Limit-def)
   by (metis Ord-in-Ord Ord-linear Ord-mem-iff-lt Ord-mult Ord-succ assms(2)
```

less-V-def mult-cancellation mult-succ not-add-mem-right succ-le-iff succ-ne-self)
  qed
  have $\alpha\uparrow(\beta * (SUP\ \xi{\in}elts\ \mu.\ \xi)) = \alpha\uparrow\bigsqcup\ ((*)\ \beta\ `\ elts\ \mu)$
   by (*simp add*: *mult-Sup-distrib*)
  also have $\ldots = \bigsqcup\ (\bigcup x{\in}elts\ \mu.\ (\uparrow)\ \alpha\ `\ elts\ (\beta * x))$
   using *False Lim oexp-Limit* **by** *fastforce*
  also have $\ldots = (SUP\ x{\in}elts\ \mu.\ \alpha\uparrow(\beta * x))$
  proof (*rule Sup-eq-Sup*)
   show $(\lambda x.\ \alpha\uparrow(\beta * x))\ `\ elts\ \mu \subseteq (\bigcup x{\in}elts\ \mu.\ (\uparrow)\ \alpha\ `\ elts\ (\beta * x))$
    using ‹*Ord* α› ‹*Ord* β› *False Limit*
    apply *clarsimp*
    by (*metis Limit-def elts-succ imageI insertI1 mem-0-Ord mult-add-mem-0*)
   show $\exists\, y{\in}(\lambda x.\ \alpha\uparrow(\beta * x))\ `\ elts\ \mu.\ x \leq y$
    if $x \in (\bigcup x{\in}elts\ \mu.\ (\uparrow)\ \alpha\ `\ elts\ (\beta * x))$ **for** x
    using *that* ‹*Ord* α› ‹*Ord* β› *False Limit*
   by *clarsimp* (*metis Limit-def Ord-in-Ord Ord-mult VWO-TC-le mem-imp-VWO oexp-mono*)
  qed *auto*
  also have $\ldots = \bigsqcup\ ((\uparrow)\ (\alpha\uparrow\beta)\ `\ elts\ (SUP\ \xi{\in}elts\ \mu.\ \xi))$
   using *Limit.IH Limit.hyps* **by** *auto*
  also have $\ldots = (\alpha\uparrow\beta)\uparrow(SUP\ \xi{\in}elts\ \mu.\ \xi)$
   using *False Limit.hyps oexp-Limit* ‹*Ord* β› **by** *auto*
  finally show *?case* **.**
 qed
qed

lemma *Limit-omega-oexp*:
 assumes *Ord* δ $\delta \neq 0$
 shows *Limit* $(\omega\uparrow\delta)$
 using *assms*
proof (*cases* δ *rule*: *Ord-cases*)
 case *0*
 then show *?thesis*
  using *assms*(*2*) **by** *blast*
next
 case (*succ l*)
 have $*$: *succ* $\beta \in$ *elts* $(\omega\uparrow l * n + \omega\uparrow l)$
  if n: $n \in$ *elts* ω **and** β: $\beta \in$ *elts* $(\omega\uparrow l * n)$ **for** n β
 proof −
  obtain *Ord n Ord* β
   by (*meson Ord-ω Ord-in-Ord Ord-mult Ord-oexp* β *n succ*(*1*))
  obtain *oo*: *Ord* $(\omega\uparrow l)$ *Ord* $(\omega\uparrow l * n)$
   by (*simp add*: ‹*Ord n*› *succ*(*1*))
  moreover have *f4*: $\beta < \omega\uparrow l * n$
   using *oo Ord-mem-iff-lt* ‹*Ord* β› ‹$\beta \in$ *elts* $(\omega\uparrow l * n)$› **by** *blast*
  moreover have *f5*: *Ord* (*succ* β)
   using ‹*Ord* β› **by** *blast*
  moreover have $\omega\uparrow l \neq 0$
   using *oexp-eq-0-iff omega-nonzero succ*(*1*) **by** *blast*

ultimately show *?thesis*
by (*metis add-less-cancel-left Ord-ω Ord-add Ord-mem-iff-lt OrdmemD ‹Ord β› add.right-neutral dual-order.strict-trans2 oexp-gt-0-iff succ*(*1*) *succ-le-iff zero-in-omega*)
qed
show *?thesis*
using *succ*
apply (*clarsimp simp*: *Limit-def mem-0-Ord*)
apply (*simp add*: *mult-Limit*)
by (*metis* ∗ *mult-succ succ-in-omega*)
next
case *limit*
then show *?thesis*
by (*metis Limit-oexp Ord-ω OrdmemD one-V-def succ-in-omega zero-in-omega*)
qed

lemma *oexp-mult-commute*:
fixes *j*::*nat*
assumes *Ord α*
shows (*α* ↑ *j*) ∗ *α* = *α* ∗ (*α* ↑ *j*)
proof −
have (*α* ↑ *j*) ∗ *α* = *α* ↑ (*1* + *ord-of-nat j*)
by (*simp add*: *one-V-def*)
also have ... = *α* ∗ (*α* ↑ *j*)
by (*simp add*: *assms oexp-add*)
finally show *?thesis* **.**
qed

lemma *oexp-ω-Limit*: *Limit β* ⟹ *ω*↑*β* = (*SUP ξ* ∈ *elts β*. *ω*↑*ξ*)
by (*simp add*: *oexp-Limit*)

lemma *ω-power-succ-gtr*: *Ord α* ⟹ *ω* ↑ *α* ∗ *ord-of-nat n* < *ω* ↑ *succ α*
by (*simp add*: *OrdmemD*)

lemma *countable-oexp*:
assumes *ν*: *α* ∈ *elts ω1*
shows *ω* ↑ *α* ∈ *elts ω1*
proof −
have *Ord α*
using *Ord-ω1 Ord-in-Ord assms* **by** *blast*
then show *?thesis*
using *assms*
proof (*induction rule*: *Ord-induct3*)
case *0*
then show *?case*
by (*simp add*: *Ord-mem-iff-lt*)
next
case (*succ α*)
then have *countable* (*elts* (*ω* ↑ *α* ∗ *ω*))
by (*simp add*: *succ-in-Limit-iff countable-mult less-ω1-imp-countable*)

then show *?case*
using *Ord-mem-iff-lt countable-iff-less-ω1 succ.hyps* **by** *auto*
next
case (*Limit α*)
with *Ord-ω1* **have** *countable* (⋃*β*∈*elts α*. *elts* (*ω* ↑ *β*)) *Ord* (*ω* ↑ ⨆ (*elts α*))
by (*force simp*: *Limit-def intro*: *Ord-trans less-ω1-imp-countable*)+
then have *ω* ↑ ⨆ (*elts α*) < *ω1*
using *Limit.hyps countable-iff-less-ω1 oexp-Limit* **by** *fastforce*
then show *?case*
using *Limit.hyps Limit-def Ord-mem-iff-lt* **by** *auto*
qed
qed

end

5 Cantor Normal Form

theory *Cantor-NF*
imports *Ordinal-Exp*
begin

5.1 Cantor normal form

Lemma 5.1

lemma *cnf-1*:
assumes *α*: *α* ∈ *elts β Ord β* **and** *m* > *0*
shows *ω*↑*α* ∗ *ord-of-nat n* < *ω*↑*β* ∗ *ord-of-nat m*
proof −
have †: *ω*↑*succ α* ≤ *ω*↑*β*
using *Ord-mem-iff-less-TC assms oexp-mono succ-le-TC-iff* **by** *auto*
have *ω*↑*α* ∗ *ord-of-nat n* < *ω*↑*α* ∗ *ω*
using *Ord-in-Ord OrdmemD assms* **by** *auto*
also have ... = *ω*↑*succ α*
using *Ord-in-Ord α* **by** *auto*
also have ... ≤ *ω*↑*β*
using † **by** *blast*
also have ... ≤ *ω*↑*β* ∗ *ord-of-nat m*
using ‹*m* > *0*› *le-mult* **by** *auto*
finally show *?thesis* **.**
qed

fun *Cantor-sum* **where**
Cantor-sum-Nil: *Cantor-sum* [] *ms* = *0*
| *Cantor-sum-Nil2*: *Cantor-sum* (*α*#*αs*) [] = *0*
| *Cantor-sum-Cons*: *Cantor-sum* (*α*#*αs*) (*m*#*ms*) = (*ω*↑*α*) ∗ *ord-of-nat m* + *Cantor-sum αs ms*

abbreviation *Cantor-dec* :: *V list* ⇒ *bool* **where**
Cantor-dec ≡ *sorted-wrt* (>)

lemma *Ord-Cantor-sum*:
assumes *List.set* $\alpha s \subseteq ON$
shows *Ord* (*Cantor-sum* αs *ms*)
using *assms*
proof (*induction* αs *arbitrary*: *ms*)
case (*Cons a* αs *ms*)
then show *?case*
by (*cases ms*) *auto*
qed *auto*

lemma *Cantor-dec-Cons-iff* [*simp*]: *Cantor-dec* ($\alpha \# \beta \# \beta s$) ⟷ $\beta < \alpha \wedge$ *Cantor-dec* ($\beta \# \beta s$)
by *auto*

Lemma 5.2. The second and third premises aren't really necessary, but their removal requires quite a lot of work.

lemma *cnf-2*:
assumes *List.set* ($\alpha \# \alpha s$) ⊆ *ON list.set ms* ⊆ {*0*<..} *length* αs = *length ms*
and *Cantor-dec* ($\alpha \# \alpha s$)
shows $\omega \uparrow \alpha >$ *Cantor-sum* αs *ms*
using *assms*
proof (*induction ms arbitrary*: α αs)
case *Nil*
then obtain $\alpha 0$ **where** $\alpha 0$: ($\alpha \# \alpha s$) = [$\alpha 0$]
by (*metis length-0-conv*)
then have *Ord* $\alpha 0$
using *Nil.prems*(*1*) **by** *auto*
then show *?case*
using $\alpha 0$ *zero-less-Limit* **by** *auto*
next
case (*Cons m1 ms*)
then obtain $\alpha 0$ $\alpha 1$ $\alpha s'$ **where** $\alpha 01$: ($\alpha \# \alpha s$) = $\alpha 0 \# \alpha 1 \# \alpha s'$
by (*metis* (*no-types*, *lifting*) *Cons.prems*(*3*) *Suc-length-conv*)
then have *Ord* $\alpha 0$ *Ord* $\alpha 1$
using *Cons.prems*(*1*) $\alpha 01$ **by** *auto*
have ∗: $\omega \uparrow \alpha 0 *$ *ord-of-nat 1* > $\omega \uparrow \alpha 1 *$ *ord-of-nat m1*
proof (*rule cnf-1*)
show $\alpha 1 \in$ *elts* $\alpha 0$
using *Cons.prems* $\alpha 01$ **by** (*simp add*: *Ord-mem-iff-lt* ‹*Ord* $\alpha 0$› ‹*Ord* $\alpha 1$›)
qed (*use* ‹*Ord* $\alpha 0$› **in** *auto*)
show *?case*
proof (*cases ms*)
case *Nil*
then show *?thesis*
using ∗ *one-V-def Cons.prems*(*3*) $\alpha 01$ **by** *auto*

```
  next
    case (Cons m2 ms')
    then obtain α2 αs'' where α02: (α#αs) = α0#α1#α2#αs''
      by (metis Cons.prems(3) Suc-length-conv α01 length-tl list.sel(3))
    then have Ord α2
      using Cons.prems(1) by auto
    have m1 > 0 m2 > 0
      using Cons.prems Cons by auto
     have ω↑α1 * ord-of-nat m1 + ω↑α1 * ord-of-nat m1 = (ω↑α1 * ord-of-nat
m1) * ord-of-nat 2
      by (simp add: mult-succ eval-nat-numeral)
    also have ... < ω↑α0
      using cnf-1 [of concl: α1 m1 * 2 α0 1] Cons.prems α01 one-V-def
      by (simp add: mult.assoc ord-of-nat-mult Ord-mem-iff-lt)
    finally have II: ω↑α1 * ord-of-nat m1 + ω↑α1 * ord-of-nat m1 < ω↑α0
      by simp
    have Cantor-sum (tl αs) ms < ω↑hd αs
    proof (rule Cons.IH)
      show Cantor-dec (hd αs # tl αs)
        using ‹Cantor-dec (α#αs)› α01 by auto
    qed (use Cons.prems α01 in auto)
    then have Cantor-sum (α2 # αs'') ms < ω↑α1
      using α02 by auto
    also have ... ≤ ω↑α1 * ord-of-nat m1
      by (simp add: ‹0 < m1› le-mult)
    finally show ?thesis
      using II α02 dual-order.strict-trans by fastforce
  qed
qed

proposition Cantor-nf-exists:
  assumes Ord α
  obtains αs ms where List.set αs ⊆ ON list.set ms ⊆ {0<..} length αs = length
ms
    and Cantor-dec αs
    and α = Cantor-sum αs ms
  using assms
proof (induction α arbitrary: thesis rule: Ord-induct)
  case (step α)
  show ?case
  proof (cases α = 0)
    case True
    have Cantor-sum [] [] = 0
      by simp
    with True show ?thesis
      using length-pos-if-in-set step.prems subset-eq
      by (metis length-0-conv not-gr-zero sorted-wrt.simps(1))
  next
    case False
```

define αhat **where** $\alpha hat \equiv Sup\ \{\gamma \in ON.\ \omega{\uparrow}\gamma \leq \alpha\}$
then have $Ord\ \alpha hat$
using *Ord-Sup assms* **by** *fastforce*
have $\bigwedge\xi.\ [\![Ord\ \xi; \omega{\uparrow}\xi \leq \alpha]\!] \Longrightarrow \xi \leq \omega{\uparrow}\alpha$
by (*metis Ord-ω OrdmemD le-oexp′ order-trans step.hyps one-V-def succ-in-omega zero-in-omega*)
then have $\{\gamma \in ON.\ \omega{\uparrow}\gamma \leq \alpha\} \subseteq elts\ (succ\ (\omega{\uparrow}\alpha))$
using *Ord-mem-iff-lt step.hyps* **by** *force*
then have *sma*: $small\ \{\gamma \in ON.\ \omega{\uparrow}\gamma \leq \alpha\}$
by (*meson down*)
have *le*: $\omega{\uparrow}\alpha hat \leq \alpha$
proof (*rule ccontr*)
assume $\neg\ \omega{\uparrow}\alpha hat \leq \alpha$
then have †: $\alpha \in elts\ (\omega{\uparrow}\alpha hat)$
by (*meson Ord-ω Ord-linear2 Ord-mem-iff-lt Ord-oexp ‹Ord αhat› step.hyps*)
obtain γ **where** $Ord\ \gamma\ \omega{\uparrow}\gamma \leq \alpha\ \alpha < \gamma$
using ‹*Ord αhat*›
proof (*cases αhat rule*: *Ord-cases*)
case *0*
with † **show** *thesis*
by (*auto simp*: *False*)
next
case (*succ β*)
have $succ\ \beta \in \{\gamma \in ON.\ \omega{\uparrow}\gamma \leq \alpha\}$
by (*rule succ-in-Sup-Ord*) (*use succ αhat-def sma* **in** *auto*)
then have $\omega{\uparrow}succ\ \beta \leq \alpha$
by *blast*
with † **show** *thesis*
using ‹$\neg\ \omega{\uparrow}\alpha hat \leq \alpha$› *succ* **by** *blast*
next
case *limit*
with † **show** *thesis*
apply (*clarsimp simp*: *oexp-Limit αhat-def*)
by (*meson Ord-ω Ord-in-Ord Ord-linear-le mem-not-refl oexp-mono-le omega-nonzero vsubsetD*)
qed
then show *False*
by (*metis Ord-ω OrdmemD leD le-less-trans le-oexp′ one-V-def succ-in-omega zero-in-omega*)
qed
have *False* **if** $\nexists M.\ \alpha < \omega{\uparrow}\alpha hat * ord\text{-}of\text{-}nat\ M$
proof −
have †: $\omega{\uparrow}\alpha hat * ord\text{-}of\text{-}nat\ M \leq \alpha$ **for** M
by (*meson that Ord-ω Ord-linear2 Ord-mult Ord-oexp Ord-ord-of-nat ‹Ord αhat› step.hyps*)
have $\neg\ \omega{\uparrow}succ\ \alpha hat \leq \alpha$
using *Ord-mem-iff-lt αhat-def ‹Ord αhat› sma elts-succ* **by** *blast*
then have $\alpha < \omega{\uparrow}succ\ \alpha hat$
by (*meson Ord-ω Ord-linear2 Ord-oexp Ord-succ ‹Ord αhat› step.hyps*)

```
    also have ... = ω↑αhat * ω
      using ‹Ord αhat› oexp-succ by blast
    also have ... = Sup (range (λm. ω↑αhat * ord-of-nat m))
      by (simp add: mult-Limit) (auto simp: ω-def image-image)
    also have ... ≤ α
      using † by blast
    finally show False
      by simp
  qed
  then obtain M where M: ω↑αhat * ord-of-nat M > α
    by blast
  have bound: i ≤ M if ω↑αhat * ord-of-nat i ≤ α for i
  proof −
    have ω↑αhat * ord-of-nat i < ω↑αhat * ord-of-nat M
      using M dual-order.strict-trans2 that by blast
    then show ?thesis
      using ‹Ord αhat› less-V-def by auto
  qed
  define mhat where mhat ≡ Greatest (λm. ω↑αhat * ord-of-nat m ≤ α)
  have mhat-ge: m ≤ mhat if ω↑αhat * ord-of-nat m ≤ α for m
    unfolding mhat-def
    by (metis (mono-tags, lifting) Greatest-le-nat bound that)
  have mhat: ω↑αhat * ord-of-nat mhat ≤ α
    unfolding mhat-def
    by (rule GreatestI-nat [where k=0 and b=M]) (use bound in auto)
  then obtain ξ where Ord ξ ξ ≤ α and ξ: α = ω↑αhat * ord-of-nat mhat + ξ
      by (metis Ord-ω Ord-mult Ord-oexp Ord-ord-of-nat ‹Ord αhat› step.hyps
le-Ord-diff)
  have False if ξ = α
  proof −
    have ξ ≥ ω↑αhat
      by (simp add: le that)
    then obtain ζ where Ord ζ ζ ≤ ξ and ζ: ξ = ω↑αhat + ζ
      by (metis Ord-ω Ord-oexp ‹Ord αhat› ‹Ord ξ› le-Ord-diff)
    then have α = ω↑αhat * ord-of-nat mhat + ω↑αhat + ζ
      by (simp add: ξ add.assoc)
    then have ω↑αhat * ord-of-nat (Suc mhat) ≤ α
     by (metis add-le-cancel-left add.right-neutral le-0 mult-succ ord-of-nat.simps(2))
    then show False
      using Suc-n-not-le-n mhat-ge by blast
  qed
  then have ξinα: ξ ∈ elts α
    using Ord-mem-iff-lt ‹Ord ξ› ‹ξ ≤ α› less-V-def step.hyps by auto
  show thesis
  proof (cases ξ = 0)
    case True
    show thesis
    proof (rule step.prems)
      show α = Cantor-sum [αhat] [mhat]
```

```
      by (simp add: True ξ)
    qed (use ξ True ‹α ≠ 0› ‹Ord αhat› in auto)
  next
    case False
    obtain βs ns where sub: List.set βs ⊆ ON list.set ns ⊆ {0<..}
      and len-eq: length βs = length ns
      and dec: Cantor-dec βs
      and ξeq: ξ = Cantor-sum βs ns
      using step.IH [OF ξinα] by blast
    then have length βs > 0 length ns > 0
      using False Cantor-sum.simps(1) ‹ξ = Cantor-sum βs ns› by auto
    then obtain β0 n0 βs' ns' where β0: βs = β0 # βs' and Ord β0
      and n0: ns = n0 # ns' and n0 > 0
      using sub by (auto simp: neq-Nil-conv)
    moreover have False if β0 > αhat
    proof −
      have ω↑β0 ≤ ω↑β0 * ord-of-nat n0 + u for u
        using ‹n0 > 0›
       by (metis add-le-cancel-left Ord-ord-of-nat add.right-neutral dual-order.trans
gr-implies-not-zero le-0 le-mult ord-of-eq-0-iff)
      moreover have ω↑β0 > α
        using that ‹Ord β0›
      by (metis (no-types, lifting) Ord-ω Ord-linear2 Ord-oexp Sup-upper αhat-def
leD mem-Collect-eq sma step.hyps)
      ultimately have ξ ≥ ω↑β0
        by (simp add: ξeq β0 n0)
      then show ?thesis
        using ‹α < ω↑β0› ‹ξ ≤ α› by auto
    qed
    ultimately have β0 ≤ αhat
      using Ord-linear2 ‹Ord αhat› by auto
    then consider β0 < αhat | β0 = αhat
      using dual-order.order-iff-strict by auto
    then show ?thesis
    proof cases
      case 1
      show ?thesis
      proof (rule step.prems)
        show list.set (αhat#βs) ⊆ ON
          using sub by (auto simp: ‹Ord αhat›)
        show list.set (mhat#ns) ⊆ {0::nat<..}
          using sub using ‹ξ = α ⟹ False› ξ by fastforce
        show Cantor-dec (αhat#βs)
            using that ‹β0 < αhat› ‹Ord αhat› ‹Ord β0› Ord-mem-iff-lt β0 dec
less-Suc-eq-0-disj
          by (force simp: β0 n0)
        show length (αhat#βs) = length (mhat#ns)
          by (auto simp: len-eq)
        show α = Cantor-sum (αhat#βs) (mhat#ns)
```

```
          by (simp add: ξ ξeq β0 n0)
        qed
      next
        case 2
        show ?thesis
        proof (rule step.prems)
          show list.set βs ⊆ ON
            by (simp add: sub(1))
          show list.set ((n0+mhat)#ns') ⊆ {0::nat<..}
            using n0 sub(2) by auto
          show length (βs::V list) = length ((n0+mhat)#ns')
            by (simp add: len-eq n0)
          show Cantor-dec βs
            using that β0 dec by auto
          show α = Cantor-sum βs ((n0+mhat)#ns')
            using 2
              by (simp add: add-mult-distrib β0 ξ ξeq add.assoc add.commute n0
ord-of-nat-add)
        qed
      qed
    qed
  qed
qed

lemma Cantor-sum-0E:
  assumes Cantor-sum αs ms = 0 List.set αs ⊆ ON list.set ms ⊆ {0<..} length
αs = length ms
  shows αs = []
  using assms
proof (induction αs arbitrary: ms)
  case Nil
  then show ?case
    by auto
next
  case (Cons a list)
  then obtain m ms' where ms = m#ms' m ≠ 0 list.set ms' ⊆ {0<..}
    by simp (metis Suc-length-conv greaterThan-iff insert-subset list.set(2))
  with Cons show ?case by auto
qed


lemma Cantor-nf-unique-aux:
  assumes Ord α
    and αsON: List.set αs ⊆ ON
    and βsON: List.set βs ⊆ ON
    and ms: list.set ms ⊆ {0<..}
    and ns: list.set ns ⊆ {0<..}
    and mseq: length αs = length ms
    and nseq: length βs = length ns
```

```
    and αsdec: Cantor-dec αs
    and βsdec: Cantor-dec βs
    and αseq: α = Cantor-sum αs ms
    and βseq: α = Cantor-sum βs ns
  shows αs = βs ∧ ms = ns
  using assms
proof (induction α arbitrary: αs ms βs ns rule: Ord-induct)
  case (step α)
  show ?case
  proof (cases α = 0)
    case True
    then show ?thesis
      using step.prems by (metis Cantor-sum-0E length-0-conv)
  next
    case False
    then obtain α0 αs' β0 βs' where αs: αs = α0 # αs' and βs: βs = β0 #
βs'
      by (metis Cantor-sum.simps(1) min-list.cases step.prems(9,10))
    then have ON: Ord α0 list.set αs' ⊆ ON Ord β0 list.set βs' ⊆ ON
      using αs βs step.prems(1,2) by auto
    then obtain m0 ms' n0 ns' where ms: ms = m0 # ms' and ns: ns = n0 #
ns'
      by (metis αs βs length-0-conv list.distinct(1) list.exhaust step.prems(5,6))
    then have nz: m0 ≠ 0 list.set ms' ⊆ {0<..} n0 ≠ 0 list.set ns' ⊆ {0<..}
      using ms ns step.prems(3,4) by auto
    have False if β0 < α0
    proof −
      have Ordc: Ord (Cantor-sum βs ns) Ord (ω↑α0)
        using Ord-oexp ‹Ord α0› step.hyps step.prems(10) by blast+
      have ∗: Cantor-sum βs ns < ω↑α0
        using step.prems(2−6) ‹Ord α0› ‹Cantor-dec βs› that βs cnf-2
        by (metis Cantor-dec-Cons-iff insert-subset list.set(2) mem-Collect-eq)
      then show False
        by (metis Cantor-sum-Cons Ord-mem-iff-lt Ord-ord-of-nat Ordc αs ‹m0 ≠
0› ∗ le-mult ms not-add-mem-right ord-of-eq-0 step.prems(9,10) vsubsetD)
    qed
    moreover
    have False if α0 < β0
    proof −
      have Ordc: Ord (Cantor-sum αs ms) Ord (ω↑β0)
        using Ord-oexp ‹Ord β0› step.hyps step.prems(9) by blast+
      have ∗: Cantor-sum αs ms < ω↑β0
        using step.prems(1−5) ‹Ord β0› ‹Cantor-dec αs› that αs cnf-2
        by (metis Cantor-dec-Cons-iff βs insert-subset list.set(2))
      then show False
        by (metis Cantor-sum-Cons Ord-mem-iff-lt Ord-ord-of-nat Ordc βs ‹n0 ≠
0› ∗ le-mult not-add-mem-right ns ord-of-eq-0 step.prems(9,10) vsubsetD)
    qed
    ultimately have 1: α0 = β0
```

using *Ord-linear-lt* ‹*Ord* $\alpha 0$› ‹*Ord* $\beta 0$› **by** *blast*
have *False* **if** $m0 < n0$
proof −
have $\omega\uparrow\alpha 0 >$ *Cantor-sum* $\alpha s'$ ms'
using αs ‹*list.set* $ms' \subseteq \{0<..\}$› *cnf-2 ms step.prems*(1,5,7) **by** *auto*
then have $\alpha < \omega\uparrow\alpha 0 *$ *ord-of-nat m0* $+ \omega\uparrow\alpha 0$
by (*simp add*: αs *ms step.prems*(9))
also have $\ldots =$ $\omega\uparrow\alpha 0 *$ *ord-of-nat* (*Suc m0*)
by (*simp add*: *mult-succ*)
also have $\ldots \leq$ $\omega\uparrow\alpha 0 *$ *ord-of-nat n0*
by (*meson Ord-ω Ord-oexp Ord-ord-of-nat Suc-leI* ‹*Ord* $\alpha 0$› *mult-cancel-le-iff ord-of-nat-mono-iff that*)
also have $\ldots \leq \alpha$
by (*metis Cantor-sum-Cons add-le-cancel-left* βs ‹$\alpha 0 = \beta 0$› *add.right-neutral le-0 ns step.prems*(10))
finally show *False*
by *blast*
qed
moreover have *False* **if** $n0 < m0$
proof −
have $\omega\uparrow\beta 0 >$ *Cantor-sum* $\beta s'$ ns'
using βs ‹*list.set* $ns' \subseteq \{0<..\}$› *cnf-2 ns step.prems*(2,6,8) **by** *auto*
then have $\alpha < \omega\uparrow\beta 0 *$ *ord-of-nat n0* $+ \omega\uparrow\beta 0$
by (*simp add*: βs *ns step.prems*(10))
also have $\ldots =$ $\omega\uparrow\beta 0 *$ *ord-of-nat* (*Suc n0*)
by (*simp add*: *mult-succ*)
also have $\ldots \leq$ $\omega\uparrow\beta 0 *$ *ord-of-nat m0*
by (*meson Ord-ω Ord-oexp Ord-ord-of-nat Suc-leI* ‹*Ord* $\beta 0$› *mult-cancel-le-iff ord-of-nat-mono-iff that*)
also have $\ldots \leq \alpha$
by (*metis Cantor-sum-Cons add-le-cancel-left* αs ‹$\alpha 0 = \beta 0$› *add.right-neutral le-0 ms step.prems*(9))
finally show *False*
by *blast*
qed
ultimately have *2*: $m0 = n0$
using *nat-neq-iff* **by** *blast*
have $\alpha s' = \beta s' \wedge ms' = ns'$
proof (*rule step.IH*)
have *Cantor-sum* $\alpha s'$ $ms' < \omega\uparrow\alpha 0$
using αs *cnf-2 ms nz*(2) *step.prems*(1) *step.prems*(5) *step.prems*(7) **by** *auto*
also have $\ldots \leq$ *Cantor-sum* αs *ms*
apply (*simp add*: αs βs *ms ns*)
by (*metis Cantor-sum-Cons add-less-cancel-left ON*(1) *Ord-ω Ord-linear2 Ord-oexp Ord-ord-of-nat* αs *add.right-neutral dual-order.strict-trans1 le-mult ms not-less-0 nz*(1) *ord-of-eq-0 step.hyps step.prems*(9))
finally show *Cantor-sum* $\alpha s'$ $ms' \in$ *elts* α
using *ON*(2) *Ord-Cantor-sum Ord-mem-iff-lt step.hyps step.prems*(9) **by** *blast*

```
    show length αs' = length ms' length βs' = length ns'
      using αs ms βs ns step.prems by auto
    show Cantor-dec αs' Cantor-dec βs'
      using αs βs step.prems(7,8) by auto
    have Cantor-sum αs ms = Cantor-sum βs ns
      using step.prems(9,10) by auto
    then show Cantor-sum αs' ms' = Cantor-sum βs' ns'
      using 1 2 by (simp add: αs βs ms ns)
   qed (use ON nz in auto)
   then show ?thesis
    using 1 2 by (simp add: αs βs ms ns)
  qed
qed


proposition Cantor-nf-unique:
  assumes Cantor-sum αs ms = Cantor-sum βs ns
    and αsON: List.set αs ⊆ ON
    and βsON: List.set βs ⊆ ON
    and ms: list.set ms ⊆ {0<..}
    and ns: list.set ns ⊆ {0<..}
    and mseq: length αs = length ms
    and nseq: length βs = length ns
    and αsdec: Cantor-dec αs
    and βsdec: Cantor-dec βs
  shows αs = βs ∧ ms = ns
  using Cantor-nf-unique-aux Ord-Cantor-sum assms  by auto


lemma less-ω-power:
  assumes Ord α1 Ord β
    and α2: α2 ∈ elts α1 and β: β < ω↑α2
    and m1 > 0 m2 > 0
  shows ω↑α2 * ord-of-nat m2 + β < ω↑α1 * ord-of-nat m1 + (ω↑α2 * ord-of-nat
m2 + β)
       (is ?lhs < ?rhs)
proof −
  obtain oo: Ord (ω↑α1) Ord (ω↑α2)
    using Ord-in-Ord Ord-oexp assms by blast
  moreover obtain ord-of-nat m2 ≠ 0
    using assms ord-of-eq-0 by blast
  ultimately have β < ω↑α2 * ord-of-nat m2
    by (meson Ord-ord-of-nat β dual-order.strict-trans1 le-mult)
  with oo assms have ?lhs ≠ ?rhs
   by (metis Ord-mult Ord-ord-of-nat add-strict-mono add.assoc cnf-1 not-add-less-right
oo)
  then show ?thesis
    by (simp add: add-le-left ‹Ord β› less-V-def oo)
qed
```

lemma *Cantor-sum-ge*:
assumes *List.set* $(\alpha\#\alpha s) \subseteq ON$ *list.set ms* $\subseteq \{0<..\}$ *length ms* > 0
shows $\omega \uparrow \alpha \leq$ *Cantor-sum* $(\alpha\#\alpha s)$ *ms*
proof −
obtain *m ns* **where** *ms*: *ms* = *Cons m ns*
by (*meson assms*(3) *list.set-cases nth-mem*)
then have $\omega \uparrow \alpha \leq \omega \uparrow \alpha *$ *ord-of-nat m*
using *assms*(2) *le-mult* **by** *auto*
then show *?thesis*
using *dual-order.trans ms* **by** *auto*
qed

5.2 Simplified Cantor normal form

No coefficients, and the exponents decreasing non-strictly

fun *ω-sum* **where**
ω-sum-Nil: *ω-sum* [] = *0*
| *ω-sum-Cons*: *ω-sum* $(\alpha\#\alpha s)$ = $(\omega\uparrow\alpha)$ + *ω-sum* αs

abbreviation *ω-dec* :: *V list* ⇒ *bool* **where**
ω-dec ≡ *sorted-wrt* (≥)

lemma *Ord-ω-sum* [*simp*]: *List.set* $\alpha s \subseteq ON \Longrightarrow$ *Ord* (*ω-sum* αs)
by (*induction* αs) *auto*

lemma *ω-dec-Cons-iff* [*simp*]: *ω-dec* $(\alpha\#\beta\#\beta s) \longleftrightarrow \beta \leq \alpha \wedge$ *ω-dec* $(\beta\#\beta s)$
by *auto*

lemma *ω-sum-0E*:
assumes *ω-sum* $\alpha s = 0$ *List.set* $\alpha s \subseteq ON$
shows αs = []
using *assms*
by (*induction* αs) *auto*

fun *ω-of-Cantor* **where**
ω-of-Cantor-Nil: *ω-of-Cantor* [] *ms* = []
| *ω-of-Cantor-Nil2*: *ω-of-Cantor* $(\alpha\#\alpha s)$ [] = []
| *ω-of-Cantor-Cons*: *ω-of-Cantor* $(\alpha\#\alpha s)$ $(m\#ms)$ = *replicate m* α @ *ω-of-Cantor* αs *ms*

lemma *ω-sum-append* [*simp*]: *ω-sum* (*xs* @ *ys*) = *ω-sum xs* + *ω-sum ys*
by (*induction xs*) (*auto simp*: *add.assoc*)

lemma *ω-sum-replicate* [*simp*]: *ω-sum* (*replicate m a*) = $\omega \uparrow a *$ *ord-of-nat m*
by (*induction m*) (*auto simp*: *mult-succ simp flip*: *replicate-append-same*)

lemma *ω-sum-of-Cantor* [*simp*]: *ω-sum* (*ω-of-Cantor* αs *ms*) = *Cantor-sum* αs

ms
proof (*induction* αs *arbitrary*: *ms*)
case (*Cons a* αs *ms*)
then show *?case*
by (*cases ms*) *auto*
qed *auto*

lemma ω*-of-Cantor-subset*: *List.set* (ω*-of-Cantor* αs *ms*) $\subseteq$ *List.set* αs
proof (*induction* αs *arbitrary*: *ms*)
case (*Cons a* αs *ms*)
then show *?case*
by (*cases ms*) *auto*
qed *auto*

lemma ω*-dec-replicate*: ω*-dec* (*replicate m* α @ αs) = (*if m=0 then* ω*-dec* αs *else* ω*-dec* (α#αs))
by (*induction m arbitrary*: αs) (*simp-all flip*: *replicate-append-same*)

lemma ω*-dec-of-Cantor-aux*:
assumes *Cantor-dec* (α#αs) *length* αs = *length ms*
shows ω*-dec* (ω*-of-Cantor* (α#αs) (*m*#*ms*))
using *assms*
proof (*induction* αs *arbitrary*: *ms*)
case *Nil*
then show *?case*
using *sorted-wrt-iff-nth-less* **by** *fastforce*
next
case (*Cons a* αs *ms*)
then obtain *n ns* **where** *ns*: *ms* = *n*#*ns*
by (*metis length-Suc-conv*)
then have $a \leq \alpha$
using *Cons.prems*(*1*) *order.strict-implies-order* **by** *auto*
moreover have $\forall x \in$*list.set* (ω*-of-Cantor* αs *ns*). $x \leq a$
using *Cons ns* ‹$a \leq \alpha$›
apply (*simp add*: ω*-dec-replicate*)
by (*meson* ω*-of-Cantor-subset order.strict-implies-order subsetD*)
ultimately show *?case*
using *Cons ns* **by** (*force simp*: ω*-dec-replicate*)
qed

lemma ω*-dec-of-Cantor*:
assumes *Cantor-dec* αs *length* αs = *length ms*
shows ω*-dec* (ω*-of-Cantor* αs *ms*)
proof (*cases* αs)
case *Nil*
then have *ms* = []
using *assms* **by** *auto*

with *Nil* **show** *?thesis*
by *simp*
next
case (*Cons a list*)
then show *?thesis*
by (*metis ω-dec-of-Cantor-aux assms length-Suc-conv*)
qed

proposition *ω-nf-exists*:
assumes *Ord α*
obtains *αs* **where** *List.set αs ⊆ ON* **and** *ω-dec αs* **and** *α = ω-sum αs*
proof −
obtain *αs ms* **where** *List.set αs ⊆ ON list.set ms ⊆ {0<..}* **and** *length*: *length αs = length ms*
and *Cantor-dec αs*
and *α*: *α = Cantor-sum αs ms*
using *Cantor-nf-exists assms* **by** *blast*
then show *thesis*
by (*metis ω-dec-of-Cantor ω-of-Cantor-subset ω-sum-of-Cantor order-trans that*)
qed

lemma *ω-sum-take-drop*: *ω-sum αs = ω-sum* (*take k αs*) + *ω-sum* (*drop k αs*)
proof (*induction k arbitrary*: *αs*)
case *0*
then show *?case*
by *simp*
next
case (*Suc k*)
then show *?case*
proof (*cases αs*)
case *Nil*
then show *?thesis*
by *simp*
next
case (*Cons a list*)
with *Suc.prems* **show** *?thesis*
by (*simp add*: *add.assoc flip*: *Suc.IH*)
qed
qed

lemma *in-elts-ω-sum*:
assumes *δ ∈ elts* (*ω-sum αs*)
shows *∃k<length αs*. *∃γ∈elts* (*ω ↑* (*αs!k*)). *δ = ω-sum* (*take k αs*) + *γ*
using *assms*
proof (*induction αs arbitrary*: *δ*)
case (*Cons α αs*)
then have *δ ∈ elts* (*ω ↑ α* + *ω-sum αs*)
by *simp*

then show *?case*
proof (*rule mem-plus-V-E*)
fix η
assume η: $\eta \in$ *elts* (ω-*sum* αs) **and** δ: $\delta = \omega \uparrow \alpha + \eta$
then obtain k γ **where** k<*length* αs $\gamma \in$ *elts* ($\omega \uparrow (\alpha s!k)$) $\eta = \omega$-*sum* (*take* k αs) + γ
using *Cons.IH* **by** *blast*
then show *?case*
by (*rule-tac x*=*Suc k* **in** *exI*) (*simp add*: δ *add.assoc*)
qed *auto*
qed *auto*

lemma ω-*le-*ω-*sum*: $[\![k <$ *length* αs; *List.set* $\alpha s \subseteq ON]\!] \Longrightarrow \omega \uparrow (\alpha s!k) \leq \omega$-*sum* αs
proof (*induction* αs *arbitrary*: k)
case (*Cons a* αs)
then obtain *Ord a list.set* $\alpha s \subseteq ON$
by *simp*
with *Cons.IH* **have** $\bigwedge k\ x.\ k <$ *length* $\alpha s \Longrightarrow \omega \uparrow \alpha s\ !\ k \leq \omega \uparrow a + \omega$-*sum* αs
by (*meson Ord-*ω *Ord-*ω*-sum Ord-oexp add-le-left order-trans*)
then show *?case*
using *Cons* **by** (*simp add*: *nth-Cons split*: *nat.split*)
qed *auto*

lemma ω-*sum-less-self*:
assumes *List.set* (α#αs) $\subseteq$ *ON* **and** ω-*dec* (α#αs)
shows ω-*sum* $\alpha s < \omega\uparrow\alpha + \omega$-*sum* αs
using *assms*
proof (*induction* αs *arbitrary*: α)
case *Nil*
then show *?case*
using *ZFC-in-HOL.neq0-conv* **by** *fastforce*
next
case (*Cons* $\alpha 1$ αs)
then show *?case*
by (*simp add*: *add-right-strict-mono oexp-mono-le*)
qed

Something like Lemma 5.2 for ω-*sum*

lemma ω-*sum-less-*ω-*power*:
assumes ω-*dec* (α#αs) *List.set* (α#αs) $\subseteq$ *ON*
shows ω-*sum* $\alpha s < \omega\uparrow\alpha * \omega$
using *assms*
proof (*induction* αs)
case *Nil*
then show *?case*
by (*simp add*: ω-*gt0*)
next
case (*Cons* β αs)

```
  then have Ord α
    by auto
  have ω-sum αs < ω↑α * ω
    using Cons by force
  then have ω↑β + ω-sum αs < ω↑α + ω↑α * ω
    using Cons.prems add-right-strict-mono oexp-mono-le by auto
  also have ... = ω↑α * ω
    by (metis Kirby.add-mult-distrib mult.right-neutral one-plus-ω-equals-ω)
  finally show ?case
    by simp
qed


lemma ω-sum-nf-unique-aux:
  assumes Ord α
    and αsON: List.set αs ⊆ ON
    and βsON: List.set βs ⊆ ON
    and αsdec: ω-dec αs
    and βsdec: ω-dec βs
    and αseq: α = ω-sum αs
    and βseq: α = ω-sum βs
  shows αs = βs
  using assms
proof (induction α arbitrary: αs βs rule: Ord-induct)
  case (step α)
  show ?case
  proof (cases α = 0)
    case True
    then show ?thesis
      using step.prems by (metis ω-sum-0E)
  next
    case False
    then obtain α0 αs' β0 βs' where αs: αs = α0 # αs' and βs: βs = β0 #
βs'
      by (metis ω-sum.elims step.prems(5,6))
    then have ON: Ord α0 list.set αs' ⊆ ON Ord β0 list.set βs' ⊆ ON
      using αs βs step.prems(1,2) by auto
    have False if β0 < α0
    proof −
      have Ordc: Ord (ω-sum βs) Ord (ω↑α0)
        using Ord-oexp ‹Ord α0› step.hyps step.prems(6) by blast+
      have ω-sum βs < ω↑β0 * ω
        by (rule ω-sum-less-ω-power) (use βs step.prems ON in auto)
      also have ... ≤ ω↑α0
       using ON by (metis Ord-ω Ord-succ oexp-mono-le oexp-succ omega-nonzero
succ-le-iff that)
      finally show False
        using αs leD step.prems(5,6) by auto
    qed
```

```
    moreover
    have False if α0 < β0
    proof −
      have Ordc: Ord (ω-sum αs) Ord (ω↑β0)
        using Ord-oexp ‹Ord β0› step.hyps step.prems(5) by blast+
      have ω-sum αs < ω↑α0 * ω
        by (rule ω-sum-less-ω-power) (use αs step.prems ON in auto)
      also have ... ≤ ω↑β0
       using ON by (metis Ord-ω Ord-succ oexp-mono-le oexp-succ omega-nonzero
succ-le-iff that)
      finally show False
        using βs leD step.prems(5,6)
        by (simp add: ‹α = ω-sum αs› leD)
    qed
    ultimately have †: α0 = β0
      using Ord-linear-lt ‹Ord α0› ‹Ord β0› by blast
    moreover have αs′ = βs′
    proof (rule step.IH)
      show ω-sum αs′ ∈ elts α
        using step.prems αs
        by (simp add: Ord-mem-iff-lt ω-sum-less-self)
      show ω-dec αs′ ω-dec βs′
        using αs βs step.prems(3,4) by auto
      have ω-sum αs = ω-sum βs
        using step.prems(5,6) by auto
      then show ω-sum αs′ = ω-sum βs′
        by (simp add: † αs βs)
    qed (use ON in auto)
    ultimately show ?thesis
      by (simp add: αs βs)
  qed
qed
```

5.3 Indecomposable ordinals

Cf exercise 5 on page 43 of Kunen

```
definition indecomposable
  where indecomposable α ≡ Ord α ∧ (∀ β ∈ elts α. ∀ γ ∈ elts α. β+γ ∈ elts α)

lemma indecomposableD:
  ⟦indecomposable α; β < α; γ < α; Ord β; Ord γ⟧ ⟹ β+γ < α
  by (meson Ord-mem-iff-lt OrdmemD indecomposable-def)

lemma indecomposable-imp-Ord:
  indecomposable α ⟹ Ord α
  using indecomposable-def by blast

lemma indecomposable-1: indecomposable 1
  by (auto simp: indecomposable-def)
```

lemma *indecomposable-0*: *indecomposable 0*
by (*auto simp*: *indecomposable-def*)

lemma *indecomposable-succ* [*simp*]: *indecomposable* (*succ* α) $\longleftrightarrow$ $\alpha = 0$
using *not-add-mem-right*
apply (*auto simp*: *indecomposable-def*)
apply (*metis add-right-cancel add.right-neutral*)
done

lemma *indecomposable-alt*:
assumes *ord*: *Ord* α *Ord* β **and** β: $\beta < \alpha$ **and** *minor*: $\bigwedge\beta\ \gamma.\ [\![\beta < \alpha; \gamma < \alpha; Ord\ \gamma]\!] \Longrightarrow \beta{+}\gamma < \alpha$
shows $\beta{+}\alpha = \alpha$
proof −
have ¬ $\beta{+}\alpha < \alpha$
by (*simp add*: *add-le-left ord leD*)
moreover have ¬ $\alpha < \beta{+}\alpha$
by (*metis assms le-Ord-diff less-V-def*)
ultimately show *?thesis*
by (*simp add*: *add-le-left less-V-def ord*)
qed

lemma *indecomposable-imp-eq*:
assumes *indecomposable* α *Ord* β $\beta < \alpha$
shows $\beta{+}\alpha = \alpha$
by (*metis assms indecomposableD indecomposable-def le-Ord-diff less-V-def less-irrefl*)

lemma *indecomposable2*:
assumes y: $y < x$ **and** z: $z < x$ **and** *minor*: $\bigwedge y{::}V.\ y < x \Longrightarrow y{+}x = x$
shows $y{+}z < x$
by (*metis add-less-cancel-left y z minor*)

lemma *indecomposable-imp-Limit*:
assumes *indec*: *indecomposable* α **and** $\alpha > 1$
shows *Limit* α
using *indecomposable-imp-Ord* [*OF indec*]
proof (*cases rule*: *Ord-cases*)
case (*succ* β)
then show *?thesis*
using *assms one-V-def* **by** *auto*
qed (*use assms* **in** *auto*)

lemma *eq-imp-indecomposable*:
assumes *Ord* α $\bigwedge\beta{::}V.\ \beta \in elts\ \alpha \Longrightarrow \beta{+}\alpha = \alpha$
shows *indecomposable* α
by (*metis add-mem-right-cancel assms indecomposable-def*)

lemma *indecomposable-ω-power*:

```
  assumes Ord δ
  shows indecomposable (ω↑δ)
  unfolding indecomposable-def
proof (intro conjI ballI)
  show Ord (ω↑δ)
    by (simp add: ‹Ord δ›)
next
  fix β γ
  assume asm: β ∈ elts (ω↑δ) γ ∈ elts (ω↑δ)
  then obtain ord: Ord β Ord γ and β: β < ω↑δ and γ: γ < ω↑δ
    by (meson Ord-ω Ord-in-Ord Ord-oexp OrdmemD ‹Ord δ›)
  show β + γ ∈ elts (ω↑δ)
    using ‹Ord δ›
  proof (cases δ rule: Ord-cases)
    case 0
    then show ?thesis
      using ‹Ord δ› asm by auto
  next
    case (succ l)
    have ∃ x∈elts ω. β + γ ∈ elts (ω↑l * x)
      if x: x ∈ elts ω β ∈ elts (ω↑l * x) and y: y ∈ elts ω γ ∈ elts (ω↑l * y)
      for x y
    proof −
      obtain Ord x Ord y Ord (ω↑l * x) Ord (ω↑l * y)
        using Ord-ω Ord-mult Ord-oexp x y nat-into-Ord succ(1) by presburger
      then have β + γ ∈ elts (ω↑l * (x+y))
        using add-mult-distrib Ord-add Ord-mem-iff-lt add-strict-mono ord x y by
presburger
      then show ?thesis
        using x y by blast
    qed
    then show ?thesis
    using ‹Ord δ› succ ord β γ by (auto simp: mult-Limit simp flip: Ord-mem-iff-lt)
  next
    case limit
    have Ord (ω↑δ)
      by (simp add: ‹Ord δ›)
    then obtain x y where x: x ∈ elts δ Ord x β ∈ elts (ω↑x)
      and y: y ∈ elts δ Ord y γ ∈ elts (ω↑y)
      using ‹Ord δ› limit ord β γ oexp-Limit
      by (auto simp flip: Ord-mem-iff-lt intro: Ord-in-Ord)
    then have succ (x ⊔ y) ∈ elts δ
      by (metis Limit-def Ord-linear-le limit sup.absorb2 sup.orderE)
    moreover have β + γ ∈ elts (ω↑succ (x ⊔ y))
    proof −
      have oxy: Ord (x ⊔ y)
        using Ord-sup x y by blast
      then obtain ω↑x ≤ ω↑(x ⊔ y) ω↑y ≤ ω↑(x ⊔ y)
       by (metis Ord-ω Ord-linear-le Ord-mem-iff-less-TC Ord-mem-iff-lt le-TC-def
```

less-le-not-le oexp-mono omega-nonzero sup.absorb2 sup.orderE x(2) y(2))
then have $\beta \in elts\ (\omega\uparrow(x \sqcup y))\ \gamma \in elts\ (\omega\uparrow(x \sqcup y))$
using *x y* **by** *blast+*
then have $\beta + \gamma \in elts\ (\omega\uparrow(x \sqcup y) * succ\ (succ\ 0))$
by (*metis Ord-ω Ord-add Ord-mem-iff-lt Ord-oexp Ord-sup add-strict-mono mult.right-neutral mult-succ ord one-V-def x(2) y(2)*)
then show *?thesis*
apply (*simp add*: *oxy*)
using *Ord-ω Ord-mult Ord-oexp Ord-trans mem-0-Ord mult-add-mem-0 oexp-eq-0-iff omega-nonzero oxy succ-in-omega* **by** *presburger*
qed
ultimately show *?thesis*
using *ord* ‹*Ord* $(\omega\uparrow\delta)$› *limit oexp-Limit* **by** *auto*
qed
qed

lemma *ω-power-imp-eq*:
assumes $\beta < \omega\uparrow\delta$ *Ord* β *Ord* δ $\delta \neq 0$
shows $\beta + \omega\uparrow\delta = \omega\uparrow\delta$
by (*simp add*: *assms indecomposable-ω-power indecomposable-imp-eq*)

lemma *mult-oexp-indec*: $[\![Ord\ \alpha; Limit\ \mu; indecomposable\ \mu]\!] \Longrightarrow \alpha * (\alpha \uparrow \mu) = (\alpha \uparrow \mu)$
by (*metis Limit-def Ord-1 OrdmemD indecomposable-imp-eq oexp-1-right oexp-add one-V-def*)

lemma *mult-oexp-ω*: *Ord* $\alpha \Longrightarrow \alpha * (\alpha \uparrow \omega) = (\alpha \uparrow \omega)$
by (*metis Ord-1 Ord-ω oexp-1-right oexp-add one-plus-ω-equals-ω*)

lemma *type-imp-indecomposable*:
assumes α: *Ord* α
and *minor*: $\bigwedge X.\ X \subseteq elts\ \alpha \Longrightarrow ordertype\ X\ VWF = \alpha \vee ordertype\ (elts\ \alpha - X)\ VWF = \alpha$
shows *indecomposable* α
unfolding *indecomposable-def*
proof (*intro conjI ballI*)
fix $\beta\ \gamma$
assume β: $\beta \in elts\ \alpha$ **and** γ: $\gamma \in elts\ \alpha$
then obtain $\beta\gamma$: *elts* $\beta \subseteq elts\ \alpha$ *elts* $\gamma \subseteq elts\ \alpha$ *Ord* β *Ord* γ
using α *Ord-in-Ord Ord-trans* **by** *blast*
then have *oeq*: *ordertype* (*elts* β) *VWF* $= \beta$
by *auto*
show $\beta + \gamma \in elts\ \alpha$
proof (*rule ccontr*)
assume $\beta + \gamma \notin elts\ \alpha$
then obtain δ **where** δ: *Ord* δ $\beta + \delta = \alpha$
by (*metis Ord-ordertype βγ(1) le-Ord-diff less-eq-V-def minor oeq*)
then have $\delta \in elts\ \alpha$
using *Ord-linear* $\beta\gamma$ γ ‹$\beta + \gamma \notin elts\ \alpha$› **by** *blast*

```
    then have ordertype (elts α − elts β) VWF = δ
      using δ ordertype-diff Limit-def α ‹Ord β› by blast
    then show False
      by (metis β ‹δ ∈ elts α› ‹elts β ⊆ elts α› oeq mem-not-refl minor)
  qed
qed (use assms in auto)
```

This proof uses Cantor normal form, yet still is rather long

```
proposition indecomposable-is-ω-power:
  assumes inc: indecomposable μ
  obtains μ = 0 | δ where Ord δ μ = ω↑δ
proof (cases μ = 0)
  case True
  then show thesis
    by (simp add: that)
next
  case False
  obtain Ord μ
    using Limit-def assms indecomposable-def by blast
  then obtain αs ms where Cantor: List.set αs ⊆ ON list.set ms ⊆ {0<..}
                                  length αs = length ms Cantor-dec αs
    and μ: μ = Cantor-sum αs ms
    using Cantor-nf-exists by blast
  consider (0) length αs = 0 | (1) length αs = 1 | (2) length αs ≥ 2
    by linarith
  then show thesis
  proof cases
    case 0
    then show ?thesis
      using μ assms False indecomposable-def by auto
  next
    case 1
    then obtain α m where αm: αs = [α] ms = [m]
     by (metis One-nat-def ‹length αs = length ms› length-0-conv length-Suc-conv)
    then obtain Ord α m ≠ 0 Ord (ω↑α)
      using ‹list.set αs ⊆ ON› ‹list.set ms ⊆ {0<..}› by auto
    have μ: μ = ω↑α ∗ ord-of-nat m
      using αm by (simp add: μ)
    moreover have m = 1
    proof (rule ccontr)
      assume m ≠ 1
      then have 2: m ≥ 2
        using ‹m ≠ 0› by linarith
      then have m = Suc 0 + Suc 0 + (m−2)
        by simp
      then have ord-of-nat m = 1 + 1 + ord-of-nat (m−2)
           by (metis add.left-neutral mult.left-neutral mult-succ ord-of-nat.simps
ord-of-nat-add)
      then have μeq: μ = ω↑α + ω↑α + ω↑α ∗ ord-of-nat (m−2)
```

```
      using μ by (simp add: add-mult-distrib)
    moreover have less: ω↑α < μ
    by (metis Ord-ω OrdmemD μeq ‹Ord α› add-le-cancel-left0 add-less-cancel-left0
le-less-trans less-V-def oexp-gt-0-iff zero-in-omega)
    moreover have ω↑α + ω↑α * ord-of-nat (m−2) < μ
      using 2 μ ‹Ord α› assms less indecomposableD less-V-def by auto
    ultimately show False
      using indecomposableD [OF inc, of ω↑α ω↑α + ω↑α * ord-of-nat (m−2)]
      by (simp add: ‹Ord (ω↑α)› add.assoc)
  qed
  moreover have Ord α
    using ‹List.set αs ⊆ ON› by (simp add: ‹αs = [α]›)
  ultimately show ?thesis
    by (metis One-nat-def mult.right-neutral ord-of-nat.simps one-V-def that(2))
 next
  case 2
   then obtain α1 α2 αs' m1 m2 ms' where αm: αs = α1#α2#αs' ms =
m1#m2#ms'
  by (metis Cantor(3) One-nat-def Suc-1 impossible-Cons length-Cons list.size(3)
not-numeral-le-zero remdups-adj.cases)
  then obtain Ord α1 Ord α2 m1 ≠ 0 m2 ≠ 0 Ord (ω↑α1) Ord (ω↑α2)
          list.set αs' ⊆ ON list.set ms' ⊆ {0<..}
    using ‹list.set αs ⊆ ON› ‹list.set ms ⊆ {0<..}› by auto
  have oCs: Ord (Cantor-sum αs' ms')
    by (simp add: Ord-Cantor-sum ‹list.set αs' ⊆ ON›)
  have α21: α2 ∈ elts α1
    using Cantor-dec-Cons-iff αm(1) ‹Cantor-dec αs›
    by (simp add: Ord-mem-iff-lt ‹Ord α1› ‹Ord α2›)
  have ω↑α2 ≠ 0
    by (simp add: ‹Ord α2›)
  then have *: (ω↑α2 * ord-of-nat m2 + Cantor-sum αs' ms') > 0
    by (simp add: OrdmemD ‹Ord (ω↑α2)› ‹m2 ≠ 0› mem-0-Ord oCs)
  have μ: μ = ω↑α1 * ord-of-nat m1 + (ω↑α2 * ord-of-nat m2 + Cantor-sum
αs' ms')
    (is μ = ?α + ?β)
    using αm by (simp add: μ)
  moreover
  have ω↑α2 * ord-of-nat m2 + Cantor-sum αs' ms' < ω↑α1 * ord-of-nat m1
+ (ω↑α2 * ord-of-nat m2 + Cantor-sum αs' ms')
    if α2 ∈ elts α1
  proof (rule less-ω-power)
    show Cantor-sum αs' ms' < ω↑α2
      using αm Cantor cnf-2 by auto
  qed (use oCs ‹Ord α1› ‹m1 ≠ 0› ‹m2 ≠ 0› that in auto)
  then have ?β < μ
    using α21 by (simp add: μ αm)
  moreover have less: ?α < μ
    using oCs by (metis μ * add-less-cancel-left add.right-neutral)
  ultimately have False
```

```
    using indecomposableD [OF inc, of ?α ?β]
    by (simp add: ‹Ord (ω↑α1)› ‹Ord (ω↑α2)› oCs)
   then show ?thesis ..
  qed
qed

corollary indecomposable-iff-ω-power:
  indecomposable μ ⟷ μ = 0 ∨ (∃δ. μ = ω↑δ ∧ Ord δ)
 by (meson indecomposable-0 indecomposable-ω-power indecomposable-is-ω-power)

theorem indecomposable-imp-type:
  fixes X :: bool ⇒ V set
  assumes γ: indecomposable γ
    and ⋀b. ordertype (X b) VWF ≤ γ ⋀b. small (X b) ⋀b. X b ⊆ ON
    and elts γ ⊆ (UN b. X b)
  shows ∃b. ordertype (X b) VWF = γ
  using γ [THEN indecomposable-imp-Ord] assms
proof (induction arbitrary: X)
  case (succ β)
  show ?case
  proof (cases β = 0)
    case True
    then have ∃b. 0 ∈ X b
      using succ.prems(5) by blast
    then have ∃b. ordertype (X b) VWF ≠ 0
      using succ.prems(3) by auto
    then have ∃b. ordertype (X b) VWF ≥ succ 0
      by (meson Ord-0 Ord-linear2 Ord-ordertype less-eq-V-0-iff succ-le-iff)
    then show ?thesis
      using True succ.prems(2) by blast
  next
    case False
    then show ?thesis
      using succ.prems by auto
  qed
next
  case (Limit γ)
  then obtain δ where δ: γ = ω↑δ and δ ≠ 0 Ord δ
   by (metis Limit-eq-Sup-self image-ident indecomposable-is-ω-power not-succ-Limit
oexp-0-right one-V-def zero-not-Limit)
  show ?case
  proof (cases Limit δ)
    case True
    have ot: ∃b. ordertype (X b ∩ elts (ω↑α)) VWF = ω↑α
      if α ∈ elts δ for α
    proof (rule Limit.IH)
      have Ord α
        using Ord-in-Ord ‹Ord δ› that by blast
      then show ω↑α ∈ elts γ
```

```
      by (simp add: Ord-mem-iff-lt δ ω-gt1 ‹Ord δ› oexp-less that)
    show indecomposable (ω↑α)
      using ‹Ord α› indecomposable-1 indecomposable-ω-power by fastforce
    show small (X b ∩ elts (ω↑α)) for b
      by (meson down inf-le2)
    show ordertype (X b ∩ elts (ω ↑ α)) VWF ≤ ω ↑ α for b
      by (simp add: ‹Ord α› ordertype-le-Ord)
    show X b ∩ elts (ω ↑ α) ⊆ ON for b
      by (simp add: Limit.prems inf.coboundedI1)
    show elts (ω ↑ α) ⊆ (⋃ b. X b ∩ elts (ω ↑ α))
      using Limit.prems Limit.hyps ‹ω ↑ α ∈ elts γ›
      by clarsimp (metis Ord-trans UN-E indecomposable-imp-Ord subset-eq)
  qed
  define A where A ≡ λb. {α ∈ elts δ. ordertype (X b ∩ elts (ω↑α)) VWF ≥
ω↑α}
  have Asmall: small (A b) for b
    by (simp add: A-def)
  have AON: A b ⊆ ON for b
    using A-def ‹Ord δ› elts-subset-ON by blast
  have eq: elts δ = (⋃ b. A b)
    by (auto simp: A-def) (metis ot eq-refl)
  then obtain b where b: Sup (A b) = δ
    using ‹Limit δ›
    apply (auto simp: UN-bool-eq)
  by (metis AON ON-imp-Ord Ord-Sup Ord-linear-le Limit-eq-Sup-self Sup-Un-distrib
Asmall sup.absorb2 sup.orderE)
  have ω↑α ≤ ordertype (X b) VWF if α ∈ A b for α
  proof −
    have (ω↑α) = ordertype ((X b) ∩ elts (ω↑α)) VWF
      using ‹Ord δ› that by (simp add: A-def Ord-in-Ord dual-order.antisym
ordertype-le-Ord)
    also have ... ≤ ordertype (X b) VWF
      by (simp add: Limit.prems ordertype-VWF-mono)
    finally show ?thesis .
  qed
  then have ordertype (X b) VWF ≥ Sup ((λα. ω↑α) ‘ A b)
    by blast
  moreover have Sup ((λα. ω↑α) ‘ A b) = ω ↑ Sup (A b)
    by (metis b Ord-ω ZFC-in-HOL.Sup-empty AON ‹δ ≠ 0› Asmall oexp-Sup
omega-nonzero)
  ultimately show ?thesis
    using Limit.hyps Limit.prems δ b by auto
next
  case False
  then obtain β where β: δ = succ β Ord β
    using Ord-cases ‹δ ≠ 0› ‹Ord δ› by auto
  then have Ordωβ: Ord (ω↑β)
    using Ord-oexp by blast
  have subX12: elts (ω↑β ∗ ω) ⊆ (⋃ b. X b)
```

using *Limit* β δ **by** *auto*
define E **where** $E \equiv \lambda n.\ \{\omega\uparrow\beta * \textit{ord-of-nat}\ n\ ..<\ \omega\uparrow\beta * \textit{ord-of-nat}\ (\textit{Suc}\ n)\} \cap \textit{ON}$
have *EON*: $E\ n \subseteq \textit{ON}$ **for** n
using *E-def* **by** *blast*
have *E-imp-less*: $x < y$ **if** $i < j\ x \in E\ i\ y \in E\ j$ **for** $x\ y\ i\ j$
proof −
have $\textit{succ}\ (i) \leq \textit{ord-of-nat}\ j$
using *that*(*1*) **by** *force*
then have $\neg\ y \leq x$
using *that*
apply (*auto simp*: *E-def*)
by (*metis Ordωβ Ord-ord-of-nat leD mult-cancel-le-iff ord-of-nat.simps*(*2*) *order-trans*)
with *that* **show** *?thesis*
by (*meson EON ON-imp-Ord Ord-linear2*)
qed
then have *djE*: *disjnt* (*E i*) (*E j*) **if** $i \neq j$ **for** $i\ j$
using *that nat-neq-iff* **unfolding** *disjnt-def* **by** *auto*
have *less-imp-E*: $i \leq j$ **if** $x < y\ x \in E\ i\ y \in E\ j$ **for** $x\ y\ i\ j$
using *that E-imp-less* [*OF* - ‹*y* ∈ *E j*› ‹*x* ∈ *E i*›] *leI less-asym* **by** *blast*
have *inc*: *indecomposable* ($\omega\uparrow\beta$)
using β *indecomposable-1 indecomposable-ω-power* **by** *fastforce*
have *in-En*: $\omega\uparrow\beta * \textit{ord-of-nat}\ n + x \in E\ n$ **if** $x \in \textit{elts}\ (\omega\uparrow\beta)$ **for** $x\ n$
using *that Ordωβ Ord-in-Ord* [*OF Ordωβ*] **by** (*auto simp*: *E-def Ordωβ OrdmemD mult-succ*)
have $*$: $\textit{elts}\ \gamma = \bigcup(\textit{range}\ E)$
proof
have $\exists\, m.\ \omega\uparrow\beta * m \leq x \wedge x < \omega\uparrow\beta * \textit{succ}\ (\textit{ord-of-nat}\ m)$
if $x \in \textit{elts}\ (\omega\uparrow\beta * \textit{ord-of-nat}\ n)$ **for** $x\ n$
using *that*
apply (*clarsimp simp add*: *mult* [*of* - *ord-of-nat n*] *lift-def*)
by (*metis add-less-cancel-left OrdmemD inc indecomposable-imp-Ord mult-succ plus sup-ge1*)
moreover have *Ord x* **if** $x \in \textit{elts}\ (\omega\uparrow\beta * \textit{ord-of-nat}\ n)$ **for** $x\ n$
by (*meson Ordωβ Ord-in-Ord Ord-mult Ord-ord-of-nat that*)
ultimately show $\textit{elts}\ \gamma \subseteq \bigcup(\textit{range}\ E)$
by (*auto simp*: δ β *E-def mult-Limit elts-ω*)
have $x \in \textit{elts}\ (\omega\uparrow\beta * \textit{succ}(\textit{ord-of-nat}\ n))$
if *Ord x* $x < \omega\uparrow\beta * \textit{succ}\ (n)$ **for** $x\ n$
by (*metis that Ord-mem-iff-lt Ord-mult Ord-ord-of-nat inc indecomposable-imp-Ord ord-of-nat.simps*(*2*))
then show $\bigcup(\textit{range}\ E) \subseteq \textit{elts}\ \gamma$
by (*force simp*: δ β *E-def Limit.prems mult-Limit*)
qed
have *smE*: *small* (*E n*) **for** n
by (*metis* $*$ *complete-lattice-class.Sup-upper down rangeI*)
have *otE*: *ordertype* (*E n*) *VWF* = $\omega\uparrow\beta$ **for** n
by (*simp add*: *E-def inc indecomposable-imp-Ord mult-succ ordertype-interval-eq*)

define *cut* **where** *cut* ≡ λ*n x*. *odiff x* (ω↑β ∗ *ord-of-nat n*)
have *cutON*: *cut n* ‘ *X* ⊆ *ON* **if** *X* ⊆ *ON* **for** *n X*
using *that* **by** (*simp add*: *image-subset-iff cut-def ON-imp-Ord Ordωβ Ord-odiff*)
have *cut* [*simp*]: *cut n* (ω ↑ β ∗ *ord-of-nat n* + *x*) = *x* **for** *x n*
by (*auto simp*: *cut-def*)
have *cuteq*: *x* ∈ *cut n* ‘ (*X* ∩ *E n*) ⟷ ω↑β ∗ *ord-of-nat n* + *x* ∈ *X*
if *x*: *x* ∈ *elts* (ω↑β) **for** *x X n*
proof
show ω↑β ∗ *ord-of-nat n* + *x* ∈ *X* **if** *x* ∈ *cut n* ‘ (*X* ∩ *E n*)
using *E-def Ordωβ Ord-odiff-eq image-iff local.cut-def that* **by** *auto*
show *x* ∈ *cut n* ‘ (*X* ∩ *E n*)
if ω↑β ∗ *ord-of-nat n* + *x* ∈ *X*
by (*metis* (*full-types*) *IntI cut image-iff in-En that x*)
qed
have *ot-cuteq*: *ordertype* (*cut n* ‘ (*X* ∩ *E n*)) *VWF* = *ordertype* (*X* ∩ *E n*) *VWF* **for** *n X*
proof (*rule ordertype-VWF-inc-eq*)
show *X* ∩ *E n* ⊆ *ON*
using *E-def* **by** *blast*
then show *cut n* ‘ (*X* ∩ *E n*) ⊆ *ON*
by (*simp add*: *cutON*)
show *small* (*X* ∩ *E n*)
by (*meson Int-lower2 smE smaller-than-small*)
show *cut n x* < *cut n y*
if *x* ∈ *X* ∩ *E n y* ∈ *X* ∩ *E n x* < *y* **for** *x y*
using *that* ‹*X* ∩ *E n* ⊆ *ON*› **by**(*simp add*: *E-def Ordωβ Ord-odiff-less-odiff local.cut-def*)
qed

define *N* **where** *N* ≡ λ*b*. {*n*. *ordertype* (*X b* ∩ *E n*) *VWF* = ω↑β}
have ∃ *b*. *infinite* (*N b*)
proof (*rule ccontr*)
assume ∄ *b*. *infinite* (*N b*)
then obtain *n* **where** ⋀*b*. *n* ∉ *N b*
apply (*simp add*: *ex-bool-eq*)
by (*metis* (*full-types*) *finite-nat-set-iff-bounded not-less-iff-gr-or-eq*)
moreover
have ∃ *b*. *ordertype* (*cut n* ‘ (*X b* ∩ *E n*)) *VWF* = ω↑β
proof (*rule Limit.IH*)
show ω↑β ∈ *elts* γ
by (*metis Limit.hyps Limit-def Limit-omega Ord-mem-iff-less-TC* β δ *mult-le2 not-succ-Limit oexp-succ omega-nonzero one-V-def*)
show *indecomposable* (ω↑β)
by (*simp add*: *inc*)
show *ordertype* (*cut n* ‘ (*X b* ∩ *E n*)) *VWF* ≤ ω↑β **for** *b*
by (*metis otE inf-le2 ordertype-VWF-mono ot-cuteq smE*)
show *small* (*cut n* ‘ (*X b* ∩ *E n*)) **for** *b*
using *smE subset-iff-less-eq-V*

by (*meson inf-le2 replacement*)
show *cut n ‘ (X b ∩ E n) ⊆ ON* **for** *b*
using *E-def cutON* **by** *auto*
have *elts (ω↑β ∗ succ n) ⊆ ⋃(range X)*
by (*metis Ordωβ Ord-ω Ord-ord-of-nat less-eq-V-def mult-cancel-le-iff ord-of-nat.simps(2) ord-of-nat-le-omega order-trans subX12*)
then show *elts (ω↑β) ⊆ (⋃b. cut n ‘ (X b ∩ E n))*
by (*auto simp*: *mult-succ mult-Limit UN-subset-iff cuteq UN-bool-eq*)
qed
then have *∃b. ordertype (X b ∩ E n) VWF = ω↑β*
by (*simp add*: *ot-cuteq*)
ultimately show *False*
by (*simp add*: *N-def*)
qed
then obtain *b* **where** *b*: *infinite (N b)*
by *blast*
then obtain *φ* :: *nat ⇒ nat* **where** *φ*: *bij-betw φ UNIV (N b)* **and** *mono*: *strict-mono φ*
by (*meson bij-enumerate enumerate-mono strict-mono-def*)
then have *ordertype (X b ∩ E (φ n)) VWF = ω↑β* **for** *n*
using *N-def bij-betw-imp-surj-on* **by** *blast*
moreover have *small (X b ∩ E (φ n))* **for** *n*
by (*meson inf-le2 smE subset-iff-less-eq-V*)
ultimately have *∃f. bij-betw f (X b ∩ E (φ n)) (elts (ω↑β)) ∧ (∀x ∈ X b ∩ E (φ n). ∀y ∈ X b ∩ E (φ n). f x < f y ⟷ (x,y) ∈ VWF)*
for *n* **by** (*metis Ord-ordertype ordertype-VWF-eq-iff*)
then obtain *F* **where** *bijF*: *⋀n. bij-betw (F n) (X b ∩ E (φ n)) (elts (ω↑β))*
and *F*: *⋀n. ∀x ∈ X b ∩ E (φ n). ∀y ∈ X b ∩ E (φ n). F n x < F n y ⟷ (x,y) ∈ VWF*
by *metis*
then have *F-bound*: *⋀n. ∀x ∈ X b ∩ E (φ n). F n x < ω↑β*
by (*metis Ord-ω Ord-oexp OrdmemD β(2) bij-betw-imp-surj-on image-eqI*)
have *F-Ord*: *⋀n. ∀x ∈ X b ∩ E (φ n). Ord (F n x)*
by (*metis otE ON-imp-Ord Ord-ordertype bijF bij-betw-def elts-subset-ON imageI*)

have *inc*: *φ n ≥ n* **for** *n*
by (*simp add*: *mono strict-mono-imp-increasing*)

have *djφ*: *disjnt (E (φ i)) (E (φ j))* **if** *i ≠ j* **for** *i j*
by (*rule djE*) (*use φ that* **in** ‹*auto simp*: *bij-betw-def inj-def*›)
define *Y* **where** *Y ≡ (⋃n. E (φ n))*
have *∃n. y ∈ E (φ n)* **if** *y ∈ Y* **for** *y*
using *Y-def that* **by** *blast*
then obtain *ι* **where** *ι*: *⋀y. y ∈ Y ⟹ y ∈ E (φ (ι y))*
by *metis*
have *Y ⊆ ON*
by (*auto simp*: *Y-def E-def*)
have *ιle*: *ι x ≤ ι y* **if** *x < y x ∈ Y y ∈ Y* **for** *x y*

```
      using less-imp-E strict-mono-less-eq that ι [OF ‹x ∈ Y›] ι [OF ‹y ∈ Y›]
mono
      unfolding Y-def by blast
    have eqι: x ∈ E (φ k) ⟹ ι x = k for x k
      using ι unfolding Y-def
      by (meson UN-I disjnt-iff djφ iso-tuple-UNIV-I)

    have upper: ω↑β * ord-of-nat (ι x) ≤ x if x ∈ Y for x
      using that
    proof (clarsimp simp add: Y-def eqι)
      fix u v
      assume u: u ∈ elts (ω↑β * ord-of-nat v) and v: x ∈ E (φ v)
      then have u < ω↑β * ord-of-nat v
        by (simp add: OrdmemD β(2))
      also have ... ≤ ω↑β * ord-of-nat (φ v)
        by (simp add: β(2) inc)
      also have ... ≤ x
        using v by (simp add: E-def)
      finally show u ∈ elts x
        using ‹Y ⊆ ON›
     by (meson ON-imp-Ord Ord-ω Ord-in-Ord Ord-mem-iff-lt Ord-mult Ord-oexp
Ord-ord-of-nat β(2) that u)
    qed

    define G where G ≡ λx. ω↑β * ord-of-nat (ι x) + F (ι x) x
    have G-strict-mono: G x < G y if x < y x ∈ X b ∩ Y y ∈ X b ∩ Y for x y
    proof (cases ι x = ι y)
      case True
      then show ?thesis
        using that unfolding G-def
            by (metis F Int-iff add-less-cancel-left Limit.prems(4) ON-imp-Ord
VWF-iff-Ord-less ι)
    next
      case False
      then have ι x < ι y
        by (meson IntE ιle le-less that)
      then show ?thesis
     using that by (simp add: G-def F-Ord F-bound Ordωβ ι mult-nat-less-add-less)
    qed

    have ordertype (X b ∩ Y) VWF = (ω↑β) * ω
    proof (rule ordertype-VWF-eq-iff [THEN iffD2])
      show Ord (ω↑β * ω)
        by (simp add: β)
      show small (X b ∩ Y)
        by (meson Limit.prems(3) inf-le1 subset-iff-less-eq-V)
      have bij-betw G (X b ∩ Y) (elts (ω↑β * ω))
      proof (rule bij-betw-imageI)
        show inj-on G (X b ∩ Y)
```

```
      proof (rule linorder-inj-onI)
        fix x y
        assume xy: x < y x ∈ (X b ∩ Y) y ∈ (X b ∩ Y)
        show G x ≠ G y
          using G-strict-mono xy by force
      next
        show x ≤ y ∨ y ≤ x
          if x ∈ (X b ∩ Y) y ∈ (X b ∩ Y) for x y
         using that ‹X b ⊆ ON› by (clarsimp simp: Y-def) (metis ON-imp-Ord
Ord-linear Ord-trans)
      qed
      show G ‘ (X b ∩ Y) = elts (ω↑β * ω)
      proof
        show G ‘ (X b ∩ Y) ⊆ elts (ω↑β * ω)
          using ‹X b ⊆ ON›
          apply (clarsimp simp: G-def mult-Limit Y-def eqι)
         by (metis IntI add-mem-right-cancel bijF bij-betw-imp-surj-on image-eqI
mult-succ ord-of-nat-ω succ-in-omega)
        show elts (ω↑β * ω) ⊆ G ‘ (X b ∩ Y)
        proof
          fix x
          assume x: x ∈ elts (ω↑β * ω)
          then obtain k where n: x ∈ elts (ω↑β * ord-of-nat (Suc k))
                          and minim: ⋀m. m < Suc k ⟹ x ∉ elts (ω↑β * ord-of-nat
m)
            using elts-mult-ωE
            by (metis old.nat.exhaust)
         then obtain y where y: y ∈ elts (ω↑β) and xeq: x = ω↑β * ord-of-nat
k + y
            using x by (auto simp: mult-succ elim: mem-plus-V-E)
          then have 1: inv-into (X b ∩ E (φ k)) (F k) y ∈ (X b ∩ E (φ k))
            by (metis bijF bij-betw-def inv-into-into)
          then have (inv-into (X b ∩ E (φ k)) (F k) y) ∈ X b ∩ Y
            by (force simp: Y-def)
          moreover have G (inv-into (X b ∩ E (φ k)) (F k) y) = x
            by (metis 1 G-def Int-iff bijF bij-betw-inv-into-right eqι xeq y)
            ultimately show x ∈ G ‘ (X b ∩ Y)
              by blast
        qed
      qed
    qed
    moreover have (x,y) ∈ VWF
      if x ∈ X b x ∈ Y y ∈ X b y ∈ Y G x < G y for x y
    proof −
      have x < y
        using that by (metis G-strict-mono Int-iff Limit.prems(4) ON-imp-Ord
Ord-linear-lt less-asym)
      then show ?thesis
        using ON-imp-Ord ‹Y ⊆ ON› that by auto
```

```
    qed
    moreover have G x < G y
      if x ∈ X b x ∈ Y y ∈ X b y ∈ Y (x, y) ∈ VWF for x y
    proof −
      have x < y
        using that ON-imp-Ord ‹Y ⊆ ON› by auto
      then show ?thesis
        by (simp add: G-strict-mono that)
    qed
    ultimately show ∃f. bij-betw f (X b ∩ Y) (elts (ω↑β ∗ ω)) ∧ (∀x∈(X b ∩
Y). ∀y∈(X b ∩ Y). f x < f y ⟷ ((x, y) ∈ VWF))
      by blast
  qed
   moreover have ordertype (⋃n. X b ∩ E (φ n)) VWF ≤ ordertype (X b)
VWF
    using Limit.prems(3) ordertype-VWF-mono by auto
  ultimately have ordertype (X b) VWF = (ω↑β) ∗ ω
    using Limit.hyps Limit.prems(2) β δ
    using Y-def by auto
  then show ?thesis
    using Limit.hyps β δ by auto
 qed
qed auto

corollary indecomposable-imp-type2:
  assumes α: indecomposable γ X ⊆ elts γ
  shows ordertype X VWF = γ ∨ ordertype (elts γ − X) VWF = γ
proof −
  have Ord γ
    using assms indecomposable-imp-Ord by blast
  have ∃b. ordertype (if b then X else elts γ − X) VWF = γ
  proof (rule indecomposable-imp-type)
    show ordertype (if b then X else elts γ − X) VWF ≤ γ for b
      by (simp add: ‹Ord γ› assms ordertype-le-Ord)
    show (if b then X else elts γ − X) ⊆ ON for b
      using ‹Ord γ› assms elts-subset-ON by auto
  qed (use assms down in auto)
  then show ?thesis
    by (metis (full-types))
qed
```

5.4 From ordinals to order types

```
lemma indecomposable-ordertype-eq:
  assumes indec: indecomposable α and α: ordertype A VWF = α and A: B ⊆
A small A
  shows ordertype B VWF = α ∨ ordertype (A−B) VWF = α
proof (rule ccontr)
  assume ¬ (ordertype B VWF = α ∨ ordertype (A − B) VWF = α)
```

moreover have *ordertype* (*ordermap A VWF ‘ B*) *VWF* = *ordertype B VWF*
using ‹*B* ⊆ *A*› **by** (*auto intro*: *ordertype-image-ordermap* [*OF* ‹*small A*›])
moreover have *ordertype* (*elts α* − *ordermap A VWF ‘ B*) *VWF* = *ordertype* (*A* − *B*) *VWF*
by (*metis ordertype-map-image α A elts-of-set ordertype-def replacement*)
moreover have *ordermap A VWF ‘ B* ⊆ *elts α*
using *α A* **by** *blast*
ultimately show *False*
using *indecomposable-imp-type2* [*OF* ‹*indecomposable α*›] ‹*small A*› **by** *metis*
qed

lemma *indecomposable-ordertype-ge*:
assumes *indec*: *indecomposable α* **and** *α*: *ordertype A VWF* ≥ *α* **and** *small*: *small A small B*
shows *ordertype B VWF* ≥ *α* ∨ *ordertype* (*A*−*B*) *VWF* ≥ *α*
proof −
obtain *A′* **where** *A′* ⊆ *A ordertype A′ VWF* = *α*
by (*meson α* ‹*small A*› *indec indecomposable-def le-ordertype-obtains-subset*)
then have *ordertype* (*B* ∩ *A′*) *VWF* = *α* ∨ *ordertype* (*A′*−*B*) *VWF* = *α*
by (*metis Diff-Diff-Int Diff-subset Int-commute* ‹*small A*› *indecomposable-ordertype-eq indec smaller-than-small*)
moreover have *ordertype* (*B* ∩ *A′*) *VWF* ≤ *ordertype B VWF*
by (*meson Int-lower1 small ordertype-VWF-mono smaller-than-small*)
moreover have *ordertype* (*A′*−*B*) *VWF* ≤ *ordertype* (*A*−*B*) *VWF*
by (*meson Diff-mono Diff-subset* ‹*A′* ⊆ *A*› ‹*small A*› *order-refl ordertype-VWF-mono smaller-than-small*)
ultimately show *?thesis*
by *blast*
qed

now for finite partitions

lemma *indecomposable-ordertype-finite-eq*:
assumes *indecomposable α*
and *𝒜*: *finite 𝒜 pairwise disjnt 𝒜* ⋃*𝒜* = *A 𝒜* ≠ {} *ordertype A VWF* = *α small A*
shows ∃ *X* ∈ *𝒜*. *ordertype X VWF* = *α*
using *𝒜*
proof (*induction arbitrary*: *A*)
case (*insert X 𝒜*)
show *?case*
proof (*cases 𝒜* = {})
case *True*
then show *?thesis*
using *insert.prems* **by** *blast*
next
case *False*
have *smA*: *small* (⋃*𝒜*)
using *insert.prems* **by** *auto*
show *?thesis*

```
    proof (cases ∃X ∈ 𝒜. ordertype X VWF = α)
      case True
      then show ?thesis
        using insert.prems by blast
    next
      case False
      have X = A − ⋃𝒜
        using insert.hyps insert.prems by (auto simp: pairwise-insert disjnt-iff)
      then have ordertype X VWF = α
        using indecomposable-ordertype-eq assms insert False
        by (metis Union-mono cSup-singleton pairwise-insert smA subset-insertI)
      then show ?thesis
        using insert.prems by blast
    qed
  qed
qed auto

lemma indecomposable-ordertype-finite-ge:
  assumes indec: indecomposable α
    and 𝒜: finite 𝒜 A ⊆ ⋃𝒜 𝒜 ≠ {} ordertype A VWF ≥ α small (⋃𝒜)
  shows ∃X ∈ 𝒜. ordertype X VWF ≥ α
  using 𝒜
proof (induction arbitrary: A)
  case (insert X 𝒜)
  show ?case
  proof (cases 𝒜 = {})
    case True
    then have α ≤ ordertype X VWF
      using insert.prems
      by (simp add: order.trans ordertype-VWF-mono)
    then show ?thesis
      using insert.prems by blast
  next
    case False
    show ?thesis
    proof (cases ∃X ∈ 𝒜. ordertype X VWF ≥ α)
      case True
      then show ?thesis
        using insert.prems by blast
    next
      case False
      moreover have small (X ∪ ⋃𝒜)
        using insert.prems by auto
      moreover have ordertype (⋃(insert X 𝒜)) VWF ≥ α
        using insert.prems ordertype-VWF-mono by blast
      ultimately have ordertype X VWF ≥ α
        using indecomposable-ordertype-ge [OF indec]
        by (metis Diff-subset-conv Sup-insert cSup-singleton insert.IH small-sup-iff
subset-refl)
```

```
    then show ?thesis
      using insert.prems by blast
  qed
 qed
qed auto

end
```

6 Type Classes for ZFC

```
theory ZFC-Typeclasses
 imports ZFC-Cardinals Complex-Main

begin
```

6.1 The class of embeddable types

```
class embeddable =
 assumes ex-inj: ∃ V-of :: 'a ⇒ V. inj V-of

context countable
begin

subclass embeddable
proof −
 have inj (ord-of-nat ∘ to-nat) if inj to-nat
   for to-nat :: 'a ⇒ nat
   using that by (simp add: inj-compose inj-ord-of-nat)
 then show class.embeddable TYPE('a)
   by intro-classes (meson local.ex-inj)
qed

end

instance unit :: embeddable ..
instance bool :: embeddable ..
instance nat :: embeddable ..
instance int :: embeddable ..
instance rat :: embeddable ..
instance char :: embeddable ..
instance String.literal :: embeddable ..
instance typerep :: embeddable ..


lemma embeddable-classI:
 fixes f :: 'a ⇒ V
 assumes ⋀x y. f x = f y ⟹ x = y
 shows OFCLASS('a, embeddable-class)
proof (intro-classes, rule exI)
```

```
  show inj f
    by (rule injI [OF assms]) assumption
qed

instance V :: embeddable
  by (rule embeddable-classI [where f=id]) auto

instance prod :: (embeddable,embeddable) embeddable
proof −
  have inj (λ(x,y). ⟨V-of1 x, V-of2 y⟩) if inj V-of1 inj V-of2
    for V-of1 :: 'a ⇒ V and V-of2 :: 'b ⇒ V
    using that by (auto simp: inj-on-def)
  then show OFCLASS('a × 'b, embeddable-class)
    by intro-classes (meson embeddable-class.ex-inj)
qed

instance sum :: (embeddable,embeddable) embeddable
proof −
  have inj (case-sum (Inl ∘ V-of1) (Inr ∘ V-of2)) if inj V-of1 inj V-of2
    for V-of1 :: 'a ⇒ V and V-of2 :: 'b ⇒ V
    using that by (auto simp: inj-on-def split: sum.split-asm)
  then show OFCLASS('a + 'b, embeddable-class)
    by intro-classes (meson embeddable-class.ex-inj)
qed

instance option :: (embeddable) embeddable
proof −
  have inj (case-option 0 (λx. ZFC-in-HOL.set{V-of x})) if inj V-of
    for V-of :: 'a ⇒ V
    using that by (auto simp: inj-on-def split: option.split-asm)
  then show OFCLASS('a option, embeddable-class)
    by intro-classes (meson embeddable-class.ex-inj)
qed

primrec V-of-list where
  V-of-list V-of Nil = 0
| V-of-list V-of (x#xs) = ⟨V-of x, V-of-list V-of xs⟩

lemma inj-V-of-list:
  assumes inj V-of
  shows inj (V-of-list V-of)
proof −
  note inj-eq [OF assms, simp]
  have x = y if V-of-list V-of x = V-of-list V-of y for x y
    using that
  proof (induction x arbitrary: y)
    case Nil
    then show ?case
      by (cases y) auto
```

```
  next
    case (Cons a x)
    then show ?case
      by (cases y) auto
  qed
  then show ?thesis
    by (auto simp: inj-on-def)
qed

instance list :: (embeddable) embeddable
proof −
  have inj (rec-list 0 (λx xs r. ⟨V-of x, r⟩)) (is inj ?f)
    if V-of: inj V-of for V-of :: 'a ⇒ V
  proof −
    note inj-eq [OF V-of, simp]
    have x = y if ?f x = ?f y for x y
      using that
    proof (induction x arbitrary: y)
      case Nil
      then show ?case
        by (cases y) auto
    next
      case (Cons a x)
      then show ?case
        by (cases y) auto
    qed
    then show ?thesis
      by (auto simp: inj-on-def)
  qed
  then show OFCLASS('a list, embeddable-class)
    by intro-classes (meson embeddable-class.ex-inj)
qed
```

6.2 The class of small types

```
class small =
  assumes small: small (UNIV::'a set)
begin

subclass embeddable
  by intro-classes (meson local.small small-def)

lemma TC-small [iff]:
  fixes A :: 'a set
  shows small A
  using small smaller-than-small by blast

end
```

context *countable*
begin

subclass *small*
proof −
  have ∗: *inj* (*ord-of-nat* ∘ *to-nat*) **if** *inj to-nat*
    for *to-nat* :: *'a* ⇒ *nat*
    using *that* **by** (*simp add*: *inj-compose inj-ord-of-nat*)
  then show *class.small TYPE*(*'a*)
    by *intro-classes* (*metis small-image-nat local.ex-inj the-inv-into-onto*)
qed

end

lemma *lepoll-UNIV-imp-small*: *X* ≲ (*UNIV*::*'a*::*small set*) ⟹ *small X*
  by (*meson lepoll-iff replacement small smaller-than-small*)

lemma *lepoll-imp-small*:
  fixes *A* :: *'a*::*small set*
  assumes *X* ≲ *A*
  shows *small X*
  by (*metis lepoll-UNIV-imp-small UNIV-I assms lepoll-def subsetI*)

instance *unit* :: *small* **..**
instance *bool* :: *small* **..**
instance *nat* :: *small* **..**
instance *int* :: *small* **..**
instance *rat* :: *small* **..**
instance *char* :: *small* **..**
instance *String.literal* :: *small* **..**
instance *typerep* :: *small* **..**

instance *prod* :: (*small*,*small*) *small*
proof −
  have *inj* (λ(*x*,*y*). ⟨*V-of1 x*, *V-of2 y*⟩)
      range (λ(*x*,*y*). ⟨*V-of1 x*, *V-of2 y*⟩) ≤ *elts* (*VSigma A* (λ*x*. *B*))
    if *inj V-of1 inj V-of2 range V-of1* ≤ *elts A range V-of2* ≤ *elts B*
    for *V-of1* :: *'a* ⇒ *V* **and** *V-of2* :: *'b* ⇒ *V* **and** *A B*
    using *that* **by** (*auto simp*: *inj-on-def*)
  with *small* [**where** *'a*=*'a*] *small* [**where** *'a*=*'b*]
  show *OFCLASS*(*'a* × *'b*, *small-class*)
    apply *intro-classes*
    unfolding *small-def*
    apply *clarify*
    by (*metis down-raw dual-order.refl*)
qed

instance *sum* :: (*small*,*small*) *small*
proof −

```
  have inj (case-sum (Inl ∘ V-of1) (Inr ∘ V-of2))
      range (case-sum (Inl ∘ V-of1) (Inr ∘ V-of2)) ≤ elts (A ⊎ B)
    if inj V-of1 inj V-of2 range V-of1 ≤ elts A range V-of2 ≤ elts B
    for V-of1 :: 'a ⇒ V and V-of2 :: 'b ⇒ V and A B
    using that by (force simp: inj-on-def split: sum.split)+
  with small [where 'a='a] small [where 'a='b]
  show OFCLASS('a + 'b, small-class)
    apply intro-classes
    unfolding small-def
    apply clarify
    by (metis down-raw dual-order.refl)
qed

instance option :: (small) small
proof −
  have inj (λx. case x of None ⇒ 0 | Some x ⇒ ZFC-in-HOL.set {V-of x})
       range (λx. case x of None ⇒ 0 | Some x ⇒ ZFC-in-HOL.set {V-of x}) ≤
insert 0 (elts (VPow A))
    if inj V-of range V-of ≤ elts A
    for V-of :: 'a ⇒ V and A
    using that  by (auto simp: inj-on-def split: option.split-asm)
  with small [where 'a='a]
  show OFCLASS('a option, small-class)
    apply intro-classes
    unfolding small-def
    apply clarify
    by (metis down-raw elts-vinsert subset-insertI)
qed

instance list :: (small) small
proof −
  have small (range (V-of-list V-of))
    if inj V-of range V-of ≤ elts A
    for V-of :: 'a ⇒ V and A
  proof −
    have range (V-of-list V-of) ≈ (UNIV :: 'a list set)
      using that by (simp add: inj-V-of-list)
    also have ... ≈ lists (UNIV :: 'a set)
      by simp
    also have ... ≲ (UNIV :: 'a set) × (UNIV :: nat set)
    proof (cases finite (range (V-of::'a ⇒ V)))
      case True
      then have lists (UNIV :: 'a set) ≲ (UNIV :: nat set)
       using countable-iff-lepoll countable-image-inj-on that(1) uncountable-infinite
by blast
      then show ?thesis
        by (blast intro: lepoll-trans [OF - lepoll-times2])
    next
      case False
```

```
    then have lists (UNIV :: 'a set) ≲ (UNIV :: 'a set)
      using ‹infinite (range V-of)› eqpoll-imp-lepoll infinite-eqpoll-lists by blast
    then show ?thesis
      using lepoll-times1 lepoll-trans by fastforce
  qed
  finally show ?thesis
    by (simp add: lepoll-imp-small)
 qed
 with small [where 'a='a]
 show OFCLASS('a list, small-class)
  apply intro-classes
  unfolding small-def
  apply clarify
  by (metis inj-V-of-list order-refl small-def small-iff-range)
qed

instance fun :: (small,embeddable) embeddable
proof −
 have inj (λf. VLambda A (λx. V-of2 (f (inv V-of1 x))))
  if ∗: inj V-of1 inj V-of2 range V-of1 ≤ elts A
  for V-of1 :: 'a ⇒ V and V-of2 :: 'b ⇒ V and A
 proof −
  have f u = f' u
   if VLambda A (λu. V-of2 (f (inv V-of1 u))) = VLambda A (λx. V-of2 (f'
(inv V-of1 x)))
   for f f' :: 'a ⇒ 'b and u
   by (metis inv-f-f rangeI subsetD VLambda-eq-D2 [OF that, of V-of1 u] ∗)
  then show ?thesis
   by (auto simp: inj-on-def)
 qed
 then show OFCLASS('a ⇒ 'b, embeddable-class)
   by intro-classes (metis embeddable-class.ex-inj small order-refl replacement
small-iff)
qed

instance fun :: (small,small) small
proof −
 have inj (λf. VLambda A (λx. V-of2 (f (inv V-of1 x))))  (is inj ?φ)
    range (λf. VLambda A (λx. V-of2 (f (inv V-of1 x)))) ≤ elts (VPi A (λx. B))
  if ∗: inj V-of1 inj V-of2 range V-of1 ≤ elts A and range V-of2 ≤ elts B
  for V-of1 :: 'a ⇒ V and V-of2 :: 'b ⇒ V and A B
 proof −
  have f u = f' u
   if VLambda A (λu. V-of2 (f (inv V-of1 u))) = VLambda A (λx. V-of2 (f'
(inv V-of1 x)))
   for f f' :: 'a ⇒ 'b and u
   by (metis inv-f-f rangeI subsetD VLambda-eq-D2 [OF that, of V-of1 u] ∗)
  then show inj ?φ
   by (auto simp: inj-on-def)
```

```
    show range ?φ ≤ elts (VPi A (λx. B))
      using that by (simp add: VPi-I subset-eq)
  qed
  with small [where 'a='a] small [where 'a='b]
  show OFCLASS('a ⇒ 'b, small-class)
    apply intro-classes
    unfolding small-def
    apply clarify
    by (metis down-raw dual-order.refl)
qed

instance set :: (small) small
proof −
  have 1: inj (λx. ZFC-in-HOL.set (V-of ‘ x))
    if inj V-of for V-of :: 'a ⇒ V
    by (simp add: inj-on-def inj-image-eq-iff [OF that])
  have 2: range (λx. ZFC-in-HOL.set (V-of ‘ x)) ≤ elts (VPow A)
    if range V-of ≤ elts A for V-of :: 'a ⇒ V and A
    using that by (auto simp: inj-on-def image-subset-iff)
  from small [where 'a='a]
  show OFCLASS('a set, small-class)
    apply intro-classes
    unfolding small-def
    apply clarify
    by (metis 1 2 down-raw subsetI)
qed

instance real :: small
proof −
  have small (range (Rep-real))
    by simp
  then show OFCLASS(real, small-class)
    by intro-classes (metis Rep-real-inverse image-inv-f-f inj-on-def replacement)
qed

instance complex :: small
proof −
  have ⋀c. c ∈ range (λ(x,y). Complex x y)
    by (metis case-prod-conv complex.exhaust-sel rangeI)
  then show OFCLASS(complex, small-class)
    by intro-classes (meson TC-small replacement smaller-than-small subset-eq)
qed


end
```

7 Acknowledgements

The author was supported by the ERC Advanced Grant ALEXANDRIA (Project 742178) funded by the European Research Council.

References

[1] L. Kirby. Addition and multiplication of sets. *Mathematical Logic Quarterly*, 53(1):52–65, 2007.

[2] S. Obua. Partizan games in Isabelle/HOLZF. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *Theoretical Aspects of Computing — ICTAC 2006*, pages 272–286. Springer, 2006.