Zermelo Fraenkel Set Theory in Higher-Order Logic

Lawrence C. Paulson
Computer Laboratory
University of Cambridge

October 13, 2025

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 . . . $\approx$ *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 . . . $\lesssim$ (*UNIV::nat set*) $\times$ *A*
by (*rule Sigma-lepoll-mono*) (*auto simp*: *lists-n-lepoll-self assms*)
also have . . . $\lesssim$ *A* $\times$ *A*
by (*metis assms infinite-le-lepoll order-refl subset-imp-lepoll times-lepoll-mono*)
also have . . . $\approx$ *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*

unbundle *lattice-syntax*

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* $\Rightarrow$ *bool*
where *small* $X \equiv \exists$ *V-of* :: $'a \Rightarrow V$. *inj-on V-of* $X \wedge$ *V-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*)

lemma *small-eqpoll*: *small* $A \longleftrightarrow (\exists x.$ *elts* $x \approx A)$
unfolding *small-def* **by** (*metis UNIV-I bij-betw-def eqpoll-def eqpoll-sym imageE image-eqI*)

Small classes can be mapped to sets.

definition *set* :: V *set* $\Rightarrow V$
where *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 (λ*x*. *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 $B :: 'a \Rightarrow V\ set$
  assumes *X*: *small X* **and** *B*: $\bigwedge x.\ x \in X \Longrightarrow small\ (B\ x)$
  shows $small\ (\bigcup x{\in}X.\ B\ x)$
proof −
  have $(\bigcup\ (elts\ `\ (\lambda x.\ ZFC\text{-}in\text{-}HOL.set\ (B\ x))\ `\ X)) = (\bigcup\ (B\ `\ X))$
    using *B* **by** *force*
  then show *?thesis*
    using *Union* [*OF replacement* [*OF X*, *of* $\lambda x.\ ZFC\text{-}in\text{-}HOL.set\ (B\ x)$]] **by** *simp*
qed

definition *vinsert* **where** $vinsert\ x\ y \equiv set\ (insert\ x\ (elts\ y))$

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

definition *succ* **where** $succ\ x \equiv vinsert\ x\ x$

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* $V\text{-}of\ `\ X = elts\ u$
    by (*meson small-def image-iff*)
  show *?case*
    unfolding *small-def*
  proof (*intro exI conjI*)
    show $inj\text{-}on\ (V\text{-}of(a{:=}u))\ (insert\ a\ X)$
      using *u*
      apply (*clarsimp simp add*: *inj-on-def*)
      by (*metis image-eqI mem-not-refl*)
    have $(V\text{-}of(a{:=}u))\ `\ insert\ a\ X = elts\ (vinsert\ u\ u)$
      using *insert.hyps*(*2*) *u*(*2*) **by** *auto*
    then show $(V\text{-}of(a{:=}u))\ `\ insert\ a\ X \in range\ elts$
      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* $\subseteq$ *A* **shows** *small B*
  using *assms*
  by (*metis down elts-of-set image-mono small-def small-iff-range inj-on-subset*)

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

lemma *small-iff*: *small X* $\longleftrightarrow$ ($\exists$ *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* $\cup$ *elts y*)
  using *Union* [*OF small-upair*] **by** *auto*

lemma *small-eqcong*: $[\![$*small X*; *X* $\approx$ *Y*$]\!]$ $\Longrightarrow$ *small Y*
  by (*metis bij-betw-imp-surj-on eqpoll-def replacement*)

lemma *lepoll-small*: $[\![$*small Y*; *X* $\lesssim$ *Y*$]\!]$ $\Longrightarrow$ *small X*
    by (*meson lepoll-iff replacement smaller-than-small*)

lemma *big-UNIV* [*simp*]: $\neg$ *small* (*UNIV*::*V set*) (**is** $\neg$ *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-theorem 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*]: $[\![small\ X;\ small\ Y]\!] \Longrightarrow set\ X = set\ Y \longleftrightarrow X{=}Y$
  by (*metis elts-of-set*)

1.2 Type classes and other basic setup

instantiation *V* :: *zero*
begin
definition *zero-V* **where** $0 \equiv set\ \{\}$
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** $1 \equiv succ\ 0$
instance ..
end

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

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

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

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

definition *inf-V* **where** *inf-V* $x\ y \equiv$ *set* (*elts* $x \cap$ *elts* y)

definition *sup-V* **where** *sup-V* $x\ y \equiv$ *set* (*elts* $x \cup$ *elts* y)

definition *less-eq-V* **where** *less-eq-V* $x\ y \equiv$ *elts* $x \subseteq$ *elts* y

definition *less-V* **where** *less-V* $x\ y \equiv$ *less-eq* $x\ y \wedge x \neq (y{::}V)$

instance
proof
  show $(x < y) = (x \leq y \wedge \neg\ y \leq x)$ **for** $x :: V$ **and** $y :: V$
    using *ext less-V-def less-eq-V-def* **by** *auto*
  show $x \leq x$ **for** $x :: V$
    by (*simp add*: *less-eq-V-def*)
  show $x \leq z$ **if** $x \leq y\ y \leq z$ **for** $x\ y\ z :: V$
    using *that* **by** (*auto simp*: *less-eq-V-def*)
  show $x = y$ **if** $x \leq y\ y \leq x$ **for** $x\ y :: V$
    using *that* **by** (*simp add*: *ext less-eq-V-def*)
  show *inf* $x\ y \leq x$ **for** $x\ y :: V$
    by (*metis down elts-of-set inf-V-def inf-sup-ord*(*1*) *less-eq-V-def*)
  show *inf* $x\ y \leq y$ **for** $x\ y :: V$
    by (*metis Int-lower2 down elts-of-set inf-V-def less-eq-V-def*)
  show $x \leq$ *inf* $y\ z$ **if** $x \leq y\ x \leq z$ **for** $x\ y\ z :: V$
  proof −
    have *small* (*elts* $y \cap$ *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 $x \leq x \sqcup y\ y \leq x \sqcup y$ **for** $x\ y :: V$
    by (*simp-all add*: *less-eq-V-def small-Un-elts sup-V-def*)
  show *sup* $y\ z \leq x$ **if** $y \leq x\ z \leq x$ **for** $x\ y\ z :: V$
    using *less-eq-V-def sup-V-def that* **by** *auto*
  show *sup* x (*inf* $y\ z$) $=$ *inf* $(x \sqcup y)$ (*sup* $x\ z$) **for** $x\ y\ z :: V$
  proof −
    have *small* (*elts* $y \cap$ *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*]: $(\bigwedge x.\ x \in$ *elts* $a \longleftrightarrow x \in$ *elts* $b) \Longrightarrow a = b$
  by (*meson dual-order.antisym less-eq-V-def subsetI*)

lemma *vsubsetI* [*intro*!]: $(\bigwedge x.\ x \in$ *elts* $a \Longrightarrow x \in$ *elts* $b) \Longrightarrow a \leq b$
  by (*simp add*: *less-eq-V-def subsetI*)

lemma *vsubsetD* [*elim*, *intro?*]: $a \leq b \Longrightarrow c \in elts\ a \Longrightarrow c \in elts\ b$
  using *less-eq-V-def* **by** *auto*

lemma *rev-vsubsetD*: $c \in elts\ a \Longrightarrow a \leq b \Longrightarrow c \in elts\ b$
  — 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* ⇒ *V*
  where *VPow x* ≡ *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-V X* ≡ *if small X then set* (*Union* (*elts* ‘ *X*)) *else 0*
instance ..
end

instantiation *V* :: *Inf*
begin
definition *Inf-V* **where** *Inf-V X* ≡ *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** $\forall x \in X.\ x \leq a$
      using *that* ‹*bdd-above X*› *bdd-above-def* **by** *blast*
    then show *small X*
      by (*meson VPow-iff* ‹$\forall x \in X.\ x \leq a$› *down subsetI*)
  qed
  show *bdd-above X*
    if *small X*
  proof −
    have $\forall x \in X.\ x \leq \bigsqcup X$
      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* $\equiv X \neq \{\}$

instance
  proof
  show $\bigsqcap X \leq x$ **if** $x \in X$ *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 $z \leq \bigsqcap X$
    if $X \neq \{\}$ $\bigwedge x.\ x \in X \Longrightarrow z \leq x$
    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 $x \leq \bigsqcup X$ **if** $x \in X$ **and** *bdd-above X* **for** *x* :: *V* **and** *X* :: *V set*
    using *that Sup-V-def* **by** *auto*
  show $\bigsqcup X \leq (z::V)$ **if** $X \neq \{\}$ $\bigwedge x.\ x \in X \Longrightarrow x \leq z$ **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*: $[\![x \in A;\ small\ A]\!] \Longrightarrow x \leq \bigsqcup A$ **for** *A*::*V set*

by (*auto simp*: *Sup-V-def SUP-upper Union less-eq-V-def*)

lemma *Sup-least*:
fixes $z::V$ **shows** $(\bigwedge x.\ x \in A \Longrightarrow x \le z) \Longrightarrow \bigsqcup A \le z$
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 $small\ A \Longrightarrow (\bigsqcup x{\in}A.\ f\ x \sqcup g\ x) = \bigsqcup\ (f\ `\ A) \sqcup \bigsqcup\ (g\ `\ A)$
by (*force simp*:)

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

lemma *cSUP-subset-mono*:
fixes $f :: 'a \Rightarrow V\ set$ **and** $g :: 'a \Rightarrow V\ set$
shows $[\![A \subseteq B;\ \bigwedge x.\ x \in A \Longrightarrow f\ x \le g\ x]\!] \Longrightarrow \bigsqcup\ (f\ `\ A) \le \bigsqcup\ (g\ `\ B)$
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 $\bigsqcup(f \text{ ` } (\bigcup x{\in}A.\ B\ x)) \le \bigsqcup((\lambda x.\ \bigsqcup(f \text{ ` } B\ x)) \text{ ` } A)$
using *assms* **by** (*fastforce simp add*: *intro*!: *cSUP-least intro*: *cSUP-upper2 simp*: *bdd2 bdd*)
moreover have $\bigsqcup((\lambda x.\ \bigsqcup(f \text{ ` } B\ x)) \text{ ` } A) \le \bigsqcup(f \text{ ` } (\bigcup x{\in}A.\ B\ x))$
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* $A \le$ *Sup* B **for** $A{::}V$ *set*
by *auto*

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

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

lemma *Sup-eq-0-iff* [*simp*]: $\bigsqcup A = 0 \longleftrightarrow A \subseteq \{0\} \lor \neg$ *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 $\bigsqcup\ (\bigcup x{\in}A.\ f\ x) = (\bigsqcup x{\in}A.\ \bigsqcup\ (f\ x))$
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** $*$: $\bigwedge x.\ x \in A \Longrightarrow \exists y \in B.\ x \le y$
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* $\neq$ *0* **and** *zero-not-succ* [*simp*]: *0* $\neq$ *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* $\neq$ *succ i*
  by (*metis elts-succ insertI1 mem-not-refl*)

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

lemma *le-succE*: *succ i* $\leq$ *succ j* $\Longrightarrow$ *i* $\leq$ *j*
  using *less-eq-V-def mem-not-sym* **by** *auto*

lemma *succ-inject-iff* [*iff*]: *succ i* = *succ j* $\longleftrightarrow$ *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* $\neq$ *0*
  by (*metis elts-0 elts-succ insert-not-empty*)

definition *pred* **where** *pred i* $\equiv$ *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* $\equiv$ $\forall$ *y* $\in$ *elts x. y* $\leq$ *x*

definition *Ord* **where** *Ord x* $\equiv$ *Transset x* $\wedge$ ($\forall$ *y* $\in$ *elts x. Transset y*)

abbreviation *ON* **where** $ON \equiv Collect\ Ord$

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* β
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 *less-succ-self*: $x < succ\ x$
  by (*simp add*: *less-eq-V-def order-neq-le-trans subset-insertI*)

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

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

lemma *Sup-ω*: $\bigsqcup(\textit{elts}\ \omega) = \omega$
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*]: $0 \in \textit{elts}\ \omega$
by (*metis ω-def elts-of-set inf ord-of-nat.simps*(*1*) *rangeI*)

lemma *succ-in-omega* [*simp*]: $n \in \textit{elts}\ \omega \Longrightarrow \textit{succ}\ n \in \textit{elts}\ \omega$
by (*metis ω-def elts-of-set image-iff small-image-nat-V ord-of-nat.simps*(*2*) *rangeI*)

lemma *ord-of-eq-0*: $\textit{ord-of-nat}\ j = 0 \Longrightarrow j = 0$
by (*induct j*) (*auto simp*: *succ-neq-zero*)

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

lemma *ord-of-eq-0-iff* [*simp*]: $\textit{ord-of-nat}\ n = 0 \longleftrightarrow n=0$
by (*auto simp*: *ord-of-eq-0*)

lemma *ord-of-nat-inject* [*iff*]: $\textit{ord-of-nat}\ i = \textit{ord-of-nat}\ j \longleftrightarrow i=j$
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 $X \subseteq \textit{range}\ (\textit{from-nat-into}\ X)$
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*]: $\neg$ *Limit*(*succ i*)
by (*metis Limit-def Ord-mem-iff-lt elts-succ insertI1 less-irrefl*)

lemma *Limit-is-Ord*: *Limit* $\xi \Longrightarrow$ *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 $k = 0$ | *l* **where** *Ord l succ l* = *k* | *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* $\alpha \Longrightarrow P\ \xi]\!] \Longrightarrow P\ (\bigsqcup\xi \in$ *elts* $\alpha.\ \xi)$
  shows *P* α
  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* $\wedge$ *P i*) **and** *Ord-Least-le*: (*LEAST i*. *Ord i* $\wedge$ *P i*) $\le$ *k*
proof −
  have *P* (*LEAST i*. *Ord i* $\wedge$ *P i*) $\wedge$ (*LEAST i*. *Ord i* $\wedge$ *P i*) $\le$ *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*: $\neg$ (*P* (*LEAST a*. *Ord a* $\wedge$ *P a*) $\wedge$ (*LEAST a*. *Ord a* $\wedge$ *P a*) $\le$ *x*)
      have $\bigwedge y$. *Ord y* $\wedge$ *P y* $\Longrightarrow$ *x* $\le$ *y*
      proof (*rule classical*)
        fix *y*
        assume *y*: *Ord y* $\wedge$ *P y* $\neg$ *x* $\le$ *y*
        with *step* **obtain** *P* (*LEAST a*. *Ord a* $\wedge$ *P a*) **and** *le*: (*LEAST a*. *Ord a* $\wedge$ *P a*) $\le$ *y*
          by (*meson Ord-linear2 Ord-mem-iff-lt*)
        with *assm* **have** *x* $<$ (*LEAST a*. *Ord a* $\wedge$ *P a*)
          by (*meson Ord-linear-le y order.trans* ‹*Ord x*›)
        then show *x* $\le$ *y*
          using *le* **by** *auto*
      qed
      then have *Least*: (*LEAST a*. *Ord a* $\wedge$ *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* $\wedge$ *P i*) **and** (*LEAST i*. *Ord i* $\wedge$ *P i*) $\le$ *k* **by** *auto*
qed

lemma *Ord-Least*:
  assumes *Ord k P k*
  shows *Ord* (*LEAST i*. *Ord i* $\wedge$ *P i*)

proof −
  have $Ord\ (LEAST\ i.\ Ord\ i \wedge (Ord\ i \wedge P\ i))$
    using *Ord-LeastI* [**where** $P = \lambda i.\ Ord\ i \wedge P\ i$] *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 $\bigwedge a.\ [\![\ P\ a;\ \forall\, b.\ Ord\ b \wedge P\ b \longrightarrow a \le b\ ]\!] \Longrightarrow Q\ a$
  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** $(LEAST\ i.\ Ord\ i \wedge P\ i) \le y$
    by (*simp add*: *Ord-Least-le*)
next
  fix x **assume** $Ord\ x \wedge P\ x\ \forall\, y.\ Ord\ y \wedge P\ y \longrightarrow x \le y$ **thus** $Q\ x$
    by (*simp add*: *assms*(*3*))
qed

lemma *Ord-LeastI2-wellorder-ex*:
  assumes $\exists\, x.\ Ord\ x \wedge P\ x$
  and $\bigwedge a.\ [\![\ P\ a;\ \forall\, b.\ Ord\ b \wedge P\ b \longrightarrow a \le b\ ]\!] \Longrightarrow Q\ a$
  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* α *P* α **by** *blast*
let *?x* = *LEAST* α. *Ord* $\alpha \wedge$ *P* α
have *Ord ?x*
by (*metis Ord-Least* α)
moreover
{ fix β **assume** *m*: $\beta <$ *?x Ord* β
from *not-less-Ord-Least*[*OF m*] **have** $\neg$ *P* β **. }**
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 D h*
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* μ. *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* :: $((V \Rightarrow 'a) \Rightarrow V \Rightarrow 'a) \Rightarrow V \Rightarrow 'a$
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 $a \le b$ **shows** $Vfrom\ a\ i \le Vfrom\ b\ i$
proof (*induction i rule*: *eps-induct*)
  case (*step i*)
  then have $a \sqcup (\bigsqcup j{\in}elts\ i.\ VPow\ (Vfrom\ a\ j)) \le b \sqcup (\bigsqcup j{\in}elts\ i.\ VPow\ (Vfrom\ b\ j))$
    by (*intro sup-mono cSUP-subset-mono* ‹$a \le b$›) *auto*
  then show *?case*
    by (*metis Vfrom*)
qed

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

lemma *Vfrom-mono*: $[\![Ord\ i;\ a{\le}b;\ i{\le}j]\!] \Longrightarrow Vfrom\ a\ i \le Vfrom\ b\ j$
 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 $Vfrom\ a\ (i \sqcup j) \le Vfrom\ a\ i \sqcup Vfrom\ a\ j$
    by (*simp add*: *Vfrom* [*of a i* $\sqcup$ *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 ∗: $(\bigsqcup x{\in}elts\ i.\ VPow\ (Vfrom\ a\ x)) \leq VPow\ (Vfrom\ a\ i)$
  proof (*rule cSup-least*)
    show $(\lambda x.\ VPow\ (Vfrom\ a\ x))$ ‘ $elts\ i \neq \{\}$
      using *False* **by** *auto*
    show $x \leq VPow\ (Vfrom\ a\ i)$ **if** $x \in (\lambda x.\ VPow\ (Vfrom\ a\ x))$ ‘ $elts\ i$ **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 $a \sqcup (\bigsqcup j{\in}elts\ (succ\ i).\ VPow\ (Vfrom\ a\ j)) = a \sqcup VPow\ (Vfrom\ a\ i)$
      using *assms*
      by (*intro sup-mono order-antisym*) (*auto simp*: *Sup-V-insert* ∗)
  qed
qed

lemma *Vset-succ*: $Ord\ i \Longrightarrow Vset(succ(i)) = VPow(Vset(i))$
  by (*simp add*: *Vfrom-succ-Ord*)

lemma *Vfrom-Sup*:
  assumes $X \neq \{\}$ *small X*
  shows $Vfrom\ a\ (Sup\ X) = (\bigsqcup y{\in}X.\ Vfrom\ a\ y)$
proof (*rule order-antisym*)
  have $Vfrom\ a\ (\bigsqcup\ X) = a \sqcup (\bigsqcup j{\in}elts\ (\bigsqcup\ X).\ VPow\ (Vfrom\ a\ j))$
    by (*metis Vfrom*)
  also have $\ldots \leq \bigsqcup\ (Vfrom\ a$ ‘ $X)$
  proof −
    have $a \leq \bigsqcup\ (Vfrom\ a$ ‘ $X)$
      by (*metis Vfrom all-not-in-conv assms bdd-above-iff-small cSUP-upper2 replacement sup-ge1*)
    moreover have $(\bigsqcup j{\in}elts\ (\bigsqcup\ X).\ VPow\ (Vfrom\ a\ j)) \leq \bigsqcup\ (Vfrom\ a$ ‘ $X)$
    proof −
      have $VPow\ (Vfrom\ a\ x) \leq \bigsqcup\ (Vfrom\ a$ ‘ $X)$
        if $y \in X\ x \in elts\ y$ **for** $x\ y$
      proof −

have *VPow* (*Vfrom a x*) $\le$ *Vfrom a y*
by (*metis Vfrom bdd-above-iff-small cSUP-upper2 le-supI2 order-refl replacement small-elts that*(*2*))
also have ... $\le \bigsqcup$ (*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* ($\bigsqcup$ *X*) $\le \bigsqcup$ (*Vfrom a* ` *X*) **.**
have $\bigwedge x.\ x \in X \Longrightarrow$
$a \sqcup (\bigsqcup j{\in}elts\ x.\ VPow\ (Vfrom\ a\ j))$
$\le a \sqcup (\bigsqcup j{\in}elts\ (\bigsqcup\ X).\ VPow\ (Vfrom\ a\ j))$
using *cSUP-subset-mono* ‹*small X*› **by** *auto*
then show $\bigsqcup$ (*Vfrom a* ` *X*) $\le$ *Vfrom a* ($\bigsqcup$ *X*)
by (*metis Vfrom assms*(*1*) *cSUP-least*)
qed

lemma *Limit-Vfrom-eq*:
Limit(*i*) $\Longrightarrow$ *Vfrom a i* = ($\bigsqcup y \in elts\ i.\ Vfrom\ a\ y$)
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* $\Rightarrow$ *V*]]

2.1 Ordered Pairs

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

lemma *doubleton-eq-iff*: *set* {*a*,*b*} = *set* {*c*,*d*} $\longleftrightarrow$ (*a*=*c* $\wedge$ *b*=*d*) $\vee$ (*a*=*d* $\wedge$ *b*=*c*)
by (*simp add*: *Set.doubleton-eq-iff*)

definition *vpair* :: *V* $\Rightarrow$ *V* $\Rightarrow$ *V*

where *vpair a b* = *set* {*set* {*a*},*set* {*a*,*b*}}

definition *vfst* :: $V \Rightarrow V$
where *vfst p* $\equiv$ *THE x*. $\exists$ *y*. *p* = *vpair x y*

definition *vsnd* :: $V \Rightarrow V$
where *vsnd p* $\equiv$ *THE y*. $\exists$ *x*. *p* = *vpair x y*

definition *vsplit* :: $[[V, V] \Rightarrow 'a, V] \Rightarrow 'a$::{} — for pattern-matching
where *vsplit c* $\equiv$ λ*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* (‹⟨(-,/ -)⟩›)
-*hpattern* :: [*pttrn*, *patterns*] $\Rightarrow$ *pttrn* (‹⟨-,/ -⟩›)
syntax-consts
-*Enum* -*Tuple* $\rightleftharpoons$ *vpair* **and**
-*hpattern* $\rightleftharpoons$ *vsplit*
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*(λ*x* <*y*,*zs*>. *b*)
λ<*x*,*y*>. *b* $\rightleftharpoons$ *CONST vsplit*(λ*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* ⟨*a*,*b*⟩ = *a*
by (*simp add*: *vfst-def*)

lemma *vsnd-conv* [*simp*]: *vsnd* ⟨*a*,*b*⟩ = *b*
by (*simp add*: *vsnd-def*)

lemma *vsplit* [*simp*]: *vsplit c* ⟨*a*,*b*⟩ = *c a b*
by (*simp add*: *vsplit-def*)

lemma *vpair-neq-fst*: ⟨*a*,*b*⟩ $\neq$ *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* $(\lambda(x,\ y).\ \langle x,\ y \rangle)\ A$
by (*auto simp*: *inj-on-def*)

2.2 Generalized Cartesian product

definition *VSigma* :: $V \Rightarrow (V \Rightarrow V) \Rightarrow V$
where $VSigma\ A\ B \equiv set(\bigcup x \in elts\ A.\ \bigcup y \in elts\ (B\ x).\ \{\langle x,y \rangle\})$

abbreviation *vtimes* **where** $vtimes\ A\ B \equiv VSigma\ A\ (\lambda x.\ B)$

definition *pairs* :: $V \Rightarrow (V * V)set$
where $pairs\ r \equiv \{(x,y).\ \langle x,y \rangle \in elts\ r\}$

lemma *pairs-iff-elts*: $(x,y) \in pairs\ z \longleftrightarrow \langle x,y \rangle \in elts\ z$
by (*simp add*: *pairs-def*)

lemma *VSigma-iff* [*simp*]: $\langle a,b \rangle \in elts\ (VSigma\ A\ B) \longleftrightarrow a \in elts\ A \wedge b \in elts\ (B\ a)$
by (*auto simp*: *VSigma-def UNION-singleton-eq-range*)

lemma *VSigmaI* [*intro*!]: $[\![\ a \in elts\ A;\ \ b \in elts\ (B\ a)]\!] \Longrightarrow \langle a,b \rangle \in elts\ (VSigma\ A\ B)$
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 $c \in elts\ (VSigma\ A\ B)$
obtains *x y* **where** $x \in elts\ A\ \ y \in elts\ (B\ x)\ \ c=\langle x,y \rangle$
using *assms* **by** (*auto simp*: *VSigma-def split*: *if-split-asm*)

lemma *VSigmaE2* [*elim*!]:
assumes $\langle a,b \rangle \in elts\ (VSigma\ A\ B)$ **obtains** $a \in elts\ A$ **and** $b \in elts\ (B\ a)$
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* ⟷ *A*=*0* ∨ (∀ *x* ∈ *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-Sigma* [*simp*]:
  assumes *A*: *small A* **and** *B*: ⋀*x*. *x* ∈ *A* ⟹ *small* (*B x*)
  shows *small* (*Sigma A B*)
proof −
  obtain *a* **where** *elts a* ≈ *A*
    by (*meson assms small-eqpoll*)
  then obtain *f* **where** *f*: *bij-betw f* (*elts a*) *A*
    using *eqpoll-def* **by** *blast*
  have ∃ *y*. *elts y* ≈ *B x* **if** *x* ∈ *A* **for** *x*
    using *B small-eqpoll that* **by** *blast*
  then obtain *g* **where** *g*: ⋀*x*. *x* ∈ *A* ⟹ *elts* (*g x*) ≈ *B x*
    by *metis*
  with *f* **have** *elts* (*VSigma a* (*g* ∘ *f*)) ≈ *Sigma A B*
    by (*simp add*: *elts-VSigma Sigma-eqpoll-cong bij-betwE*)
  then show *?thesis*
    using *small-eqpoll* **by** *blast*
qed

lemma *small-Times* [*simp*]:
  assumes *small A small B* **shows** *small* (*A* × *B*)
  by (*simp add*: *assms*)

lemma *small-Times-iff*: *small* (*A* × *B*) ⟷ *small A* ∧ *small B* ∨ *A*={} ∨ *B*={}
(**is** - = *?rhs*)
proof
  assume ∗: *small* (*A* × *B*)
  { have *small A* ∧ *small B* **if** *x* ∈ *A y* ∈ *B* **for** *x y*
    proof −
      have *A* ⊆ *fst* ‘ (*A* × *B*) *B* ⊆ *snd* ‘ (*A* × *B*)
        using *that* **by** *auto*

with *that* **show** *?thesis*
by (*metis* ∗ *replacement smaller-than-small*)
qed **}**
then show *?rhs*
by (*metis equals0I*)
next
assume *?rhs*
then show *small* ($A \times B$)
by *auto*
qed

2.3 Disjoint Sum

definition *vsum* :: $V \Rightarrow V \Rightarrow V$ (**infixl** ‹⊎› *65*) **where**
$A \uplus B \equiv (VSigma\ (set\ \{0\})\ (\lambda x.\ A)) \sqcup (VSigma\ (set\ \{1\})\ (\lambda x.\ B))$

definition *Inl* :: $V \Rightarrow V$ **where**
$Inl\ a \equiv \langle 0, a\rangle$

definition *Inr* :: $V \Rightarrow V$ **where**
$Inr\ b \equiv \langle 1, b\rangle$

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*]: $Inl\ a \in elts\ (A \uplus B) \longleftrightarrow a \in elts\ A$
by (*auto simp*: *sum-defs*)

lemma *Inr-in-sum-iff* [*iff*]: $Inr\ b \in elts\ (A \uplus B) \longleftrightarrow b \in elts\ B$
by (*auto simp*: *sum-defs*)

lemma *sumE* [*elim*!]:
assumes *u*: $u \in elts\ (A \uplus B)$
obtains *x* **where** $x \in elts\ A$ $u{=}Inl\ x$ | *y* **where** $y \in elts\ B$ $u{=}Inr\ y$ **using** *u*
by (*auto simp*: *sum-defs*)

2.3.2 Injection and freeness equivalences, for rewriting

lemma *Inl-iff* [*iff*]: $Inl\ a{=}Inl\ b \longleftrightarrow a{=}b$
by (*simp add*: *sum-defs*)

lemma *Inr-iff* [*iff*]: $Inr\ a{=}Inr\ b \longleftrightarrow a{=}b$
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* $\longleftrightarrow$ *False*
by (*simp add*: *sum-defs*)

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

lemma *sum-empty* [*simp*]: $0 \uplus 0 = 0$
by *auto*

lemma *elts-vsum*: $elts\ (a \uplus b) = Inl\ `\ (elts\ a) \cup Inr\ `\ (elts\ b)$
by *auto*

lemma *sum-iff*: $u \in elts\ (A \uplus B) \longleftrightarrow (\exists x.\ x \in elts\ A \wedge u{=}Inl\ x) \vee (\exists y.\ y \in elts\ B \wedge u{=}Inr\ y)$
by *blast*

lemma *sum-subset-iff*: $A \uplus B \le C \uplus D \longleftrightarrow A \le C \wedge B \le D$
by (*auto simp*: *less-eq-V-def*)

lemma *sum-equal-iff*:
fixes $A :: V$ **shows** $A \uplus B = C \uplus D \longleftrightarrow A{=}C \wedge B{=}D$
by (*simp add*: *eq-iff sum-subset-iff*)

definition *is-sum* :: $V \Rightarrow bool$
where $is\text{-}sum\ z = (\exists x.\ z = Inl\ x \vee z = Inr\ x)$

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

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*]: $\neg$ *is-sum a* $\Longrightarrow$ *sum-case f g a* = *undefined*
by (*simp add*: *sum-case-def is-sum-def*)

lemma *is-sum-cases*: $(\exists x.\ z = Inl\ x \vee z = Inr\ x) \vee \neg\ is\text{-}sum\ z$

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* ($X \cup Y$)
proof –
obtain *x y* **where** *elts x* $\approx$ *X elts y* $\approx$ *Y*
by (*meson assms small-eqpoll*)
then have $X \cup Y \lesssim$ *Inl* ` (*elts x*) $\cup$ *Inr* ` (*elts y*)
by (*metis* (*mono-tags*, *lifting*) *Inr-Inl-iff Un-lepoll-mono disjnt-iff eqpoll-imp-lepoll eqpoll-sym f-inv-into-f inj-on-Inl inj-on-Inr inj-on-image-lepoll-2*)
then show *?thesis*
by (*metis lepoll-iff replacement small-elts small-sup-iff smaller-than-small*)
qed

lemma *small-UN* [*simp*,*intro*]:
assumes *A*: *small A* **and** *B*: $\bigwedge x.\ x \in A \Longrightarrow$ *small* (*B x*)
shows *small* ($\bigcup x{\in}A.\ B\ x$)
proof –
obtain *a* **where** *elts a* $\approx$ *A*
by (*meson assms small-eqpoll*)
then obtain *f* **where** *f*: *bij-betw f* (*elts a*) *A*
using *eqpoll-def* **by** *blast*
have $\exists\, y.$ *elts y* $\approx$ *B x* **if** $x \in A$ **for** *x*
using *B small-eqpoll that* **by** *blast*
then obtain *g* **where** *g*: $\bigwedge x.\ x \in A \Longrightarrow$ *elts* (*g x*) $\approx$ *B x*
by *metis*
have *sm*: *small* (*Sigma* (*elts a*) (*elts* $\circ$ *g* $\circ$ *f*))
by *simp*
have ($\bigcup x{\in}A.\ B\ x$) $\lesssim$ *Sigma A B*
by (*metis image-lepoll snd-image-Sigma*)
also have ... $\lesssim$ *Sigma* (*elts a*) (*elts* $\circ$ *g* $\circ$ *f*)
by (*smt* (*verit*) *Sigma-eqpoll-cong bij-betw-iff-bijections comp-apply eqpoll-imp-lepoll eqpoll-sym f g*)
finally show *?thesis*
using *lepoll-small sm* **by** *blast*

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* $\equiv$ *set* (($\lambda x.\ \langle x,b\ x\rangle$) ' *elts A*)

definition *app* :: $[V,V] \Rightarrow V$
  where *app f x* $\equiv$ *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* $\equiv$ *set* $\{f \in$ *elts* (*VPow*(*VSigma A B*)). *elts A* $\leq$ *Domain* (*pairs* *f*) $\wedge$ *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* ∈ *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* ≤ *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* ∈ *elts* (*TC x*) **if** *v* ∈ *elts u u* ∈ *elts x* **for** *u v*
    using *that* **unfolding** *TC* [*of x*]
    using *arg-subset-TC* **by** *fastforce*
  have *2*: *v* ∈ *elts* (*TC x*) **if** *v* ∈ *elts u* ∃*x*∈*elts x*. *u* ∈ *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*≤*x*$]\!]$ $\Longrightarrow$ *TC a* ≤ *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 $x \sqsubseteq y \equiv x \sqsubset y \lor x{=}y$

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*: $x \sqsubset y \longleftrightarrow (\exists z \in elts\ y.\ x \sqsubseteq z)$
  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*: $[\![x \sqsubset y;\ y \sqsubset x]\!] \Longrightarrow False$
 by (*metis TC-least Transset-TC Transset-def antisym-conv less-TC-def less-TC-imp-not-le order-refl*)

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

lemma *less-le-TC*: $x \sqsubset y \longleftrightarrow x \sqsubseteq y \land x \neq y$

using *le-TC-def 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 *le-TC-trans* [*trans*]: $[\![x \sqsubseteq y; y \sqsubseteq z]\!] \Longrightarrow x \sqsubseteq z$
by (*smt* (*verit*, *best*) *TC-least Transset-TC Transset-def le-TC-def less-TC-def vsubsetD*)

context *order*
begin

lemma *nless-le-TC*: $(\neg\ a \sqsubset b) \longleftrightarrow (\neg\ a \sqsubseteq b) \vee a = b$
using *le-TC-def less-asym-TC* **by** *blast*

lemma *eq-refl-TC*: $x = y \Longrightarrow x \sqsubseteq y$
by *simp*

local-setup ‹
HOL-Order-Tac.declare-order {
ops = {*eq* = @{*term* ‹(=) :: *V* ⇒ *V* ⇒ *bool*›}, *le* = @{*term* ‹(⊑)›}, *lt* = @{*term* ‹(⊏)›}},
thms = {*trans* = @{*thm le-TC-trans*}, *refl* = @{*thm le-TC-refl*}, *eqD1* = @{*thm eq-refl-TC*},
eqD2 = @{*thm eq-refl-TC*[*OF sym*]}, *antisym* = @{*thm le-antisym-TC*}, *contr* = @{*thm notE*}},
conv-thms = {*less-le* = @{*thm eq-reflection*[*OF less-le-TC*]},
nless-le = @{*thm eq-reflection*[*OF nless-le-TC*]}}
}
›

end

lemma *less-TC-trans* [*trans*]: $[\![x \sqsubset y; y \sqsubset z]\!] \Longrightarrow x \sqsubset z$
and *less-le-TC-trans*: $[\![x \sqsubset y; y \sqsubseteq z]\!] \Longrightarrow x \sqsubset z$
and *le-less-TC-trans* [*trans*]: $[\![x \sqsubseteq y; y \sqsubset z]\!] \Longrightarrow x \sqsubset z$
by *simp-all*

lemma *TC-sup-distrib*: $TC\ (x \sqcup y) = TC\ x \sqcup TC\ y$
by (*simp add*: *Sup-Un-distrib TC* [*of* $x \sqcup y$] *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 $\bigsqcup X \le \bigsqcup (TC \text{ ` } X)$
using *arg-subset-TC* **by** *fastforce*
moreover have $\bigsqcup (\bigcup x{\in}X.\ TC \text{ ` } elts\ x) \le \bigsqcup (TC \text{ ` } X)$
using *assms*
by *clarsimp* (*meson TC-least Transset-TC Transset-def arg-subset-TC replacement vsubsetD*)
ultimately
have $\bigsqcup X \sqcup \bigsqcup (\bigcup x{\in}X.\ TC \text{ ` } elts\ x) \le \bigsqcup (TC \text{ ` } X)$
by *simp*
moreover have $\bigsqcup (TC \text{ ` } X) \le \bigsqcup X \sqcup \bigsqcup (\bigcup x{\in}X.\ TC \text{ ` } elts\ x)$
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*: $b \le d$
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 $b \le TC\ b \sqcap d$
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*: $rank\ a = \bigsqcup ((\lambda y.\ succ(rank\ y))\ \text{‘}\ elts\ a)$
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*: $Ord\ i \Longrightarrow rank\ i = i$
by (*induction rule*: *Ord-induct*) (*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*: $a \in elts\ b \Longrightarrow rank\ a < rank\ b$
by (*metis Ord-linear2 Ord-rank ZFC-in-HOL.SUP-le-iff rank-Sup replacement small-elts succ-le-iff order.irrefl*)

lemma *rank-0* [*simp*]: $rank\ 0 = 0$
using *transrec Ord-0 rank-def rank-of-Ord* **by** *presburger*

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

lemma *rank-mono*: $a \le b \Longrightarrow rank\ a \le rank\ b$
using *rank* [*of* a] *rank* [*of* b] *small-UN* **by** *force*

```
lemma VsetI: rank b ⊏ i ⟹ b ∈ elts (Vset i)
proof (induction i arbitrary: b rule: eps-induct)
  case (step x)
  then consider rank b ∈ elts x | (∃ y∈elts x. rank b ∈ elts (TC y))
    using le-TC-def less-TC-def less-TC-iff by fastforce
  then have ∃ y∈elts x. b ≤ Vset y
  proof cases
    case 1
    then have b ≤ 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⟧ ⟹ b ∈ 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 ≤ Vset(rank a)
  by (simp add: Ord-VsetI rank-lt vsubsetI)

lemma two-in-Vset:
  obtains α where  x ∈ elts (Vset α) y ∈ 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 ⟷ 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) ≤ (Vset i) if x ∈ 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 ⊆ 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*: $rank(\bigsqcup A) = \bigsqcup (rank \text{ ` } A)$
proof (*rule order-antisym*)
have $elts (\bigsqcup y{\in}elts (\bigsqcup A).\ succ (rank\ y)) \subseteq elts (\bigsqcup (rank \text{ ` } A))$
by *clarsimp* (*meson Ord-mem-iff-lt Ord-rank less-V-def rank-lt vsubsetD*)
then show $rank (\bigsqcup A) \le \bigsqcup (rank \text{ ` } A)$
by (*metis less-eq-V-def rank-Sup*)
show $\bigsqcup (rank \text{ ` } A) \le rank (\bigsqcup A)$
proof (*cases small A*)
case *True*
then show *?thesis*
by (*simp add*: *ZFC-in-HOL.SUP-le-iff ZFC-in-HOL.Sup-upper rank-mono*)
next
case *False*
then have $\neg$ *small* (*rank* ` *A*)
using *small-ranks-imp-small* **by** *blast*
then show *?thesis*
by *blast*
qed
qed

lemma *small-bounded-rank*: *small* $\{x.\ rank\ x \in elts\ a\}$
proof −
have $\{x.\ rank\ x \in elts\ a\} \subseteq \{x.\ rank\ x \sqsubset a\}$
using *less-TC-iff* **by** *auto*
also have $\ldots \subseteq elts\ (Vset\ a)$
using *VsetI* **by** *blast*
finally show *?thesis*
using *down* **by** *simp*
qed

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

lemma *TC-rank-lt*: $a \sqsubset b \Longrightarrow rank\ a < rank\ b$
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*
using *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) $\leq$ *Vfrom* A x
proof (*induction* x *rule*: *eps-induct*)
case (*step* x)
have ($\bigsqcup j\in$*elts* (*rank* x). *VPow* (*Vfrom* A j)) $\leq$ ($\bigsqcup j\in$*elts* x. *VPow* (*Vfrom* A j))
apply (*rule Sup-least*)
apply (*clarsimp simp add*: *rank* [*of* x])
by (*meson Ord-in-Ord Ord-rank OrdmemD Vfrom-mono order.trans less-imp-le order.refl step*)
then show *?case*
by (*simp add*: *Vfrom* [*of* - x] *Vfrom* [*of* - *rank*(x)] *sup.coboundedI2*)
qed
show *Vfrom* A $x \leq$ *Vfrom* A (*rank* x)
proof (*induction* x *rule*: *eps-induct*)
case (*step* x)
have ($\bigsqcup j\in$*elts* x. *VPow* (*Vfrom* A j)) $\leq$ ($\bigsqcup j\in$*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)) = $A \sqcup$ *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-in-HOL.succ* k)) $u \sqsubset x$
shows $u \in$ *elts* (*Vset* k)

using *assms*
using *TC-least Transset-Vfrom Vfrom-succ 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' | *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*: $x \sqsubseteq y \Longrightarrow (x,y) \in VWO$
by (*metis Diff-iff Linear-order-VWO Linear-order-in-diff-Id UNIV-I VWO le-TC-def less-TC-imp-VWO*)

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

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

lemma *le-TC-succ*: $x \sqsubseteq succ\ \beta \longleftrightarrow x \sqsubseteq \beta \lor x = succ\ \beta$
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 DiffI IdD Linear-order-VWO Linear-order-in-diff-Id Ord-linear Ord-mem-iff-less-TC VWO iso-tuple-UNIV-I le-TC-def mem-imp-VWO*)
qed

lemma *VWO-iff-Ord-le* [*simp*]: $[\![Ord\ \alpha;\ Ord\ \beta]\!] \Longrightarrow (\beta,\ \alpha) \in VWO \longleftrightarrow \beta \le \alpha$

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 wf-VWF wf-imp-asym*)

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*:
$small\ A \Longrightarrow ordermap\ A\ r \in A \rightarrow elts\ (ordertype\ A\ r)$
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 $Transset\ (set\ (ordermap\ A\ r\ `\ \{y \in A.\ (y, u) \in r\}))$
proof (*clarsimp simp add*: *Transset-def*)
show $x \in ordermap\ A\ r\ `\ \{y \in A.\ (y, u) \in r\}$
if $small\ (ordermap\ A\ r\ `\ \{y \in A.\ (y, u) \in r\})$
and x: $x \in elts\ (ordermap\ A\ r\ y)$ **and** $y \in A\ (y, u) \in r$ **for** *x y*
proof −
have $ordermap\ A\ r\ y = ZFC\text{-}in\text{-}HOL.set\ (ordermap\ A\ r\ `\ \{a \in A.\ (a, y) \in r\})$
using *ordermap assms*(*1*) **by** *force*
then have $x \in ordermap\ A\ r\ `\ \{z \in A.\ (z, y) \in r\}$
by (*metis* (*no-types*, *lifting*) *elts-of-set empty-iff x*)
then have $\exists v.\ v \in A \wedge (v, u) \in r \wedge x = ordermap\ A\ r\ v$
using *that transD* [*OF* ‹*trans r*›] **by** *blast*
then show *?thesis*
by *blast*
qed
qed
moreover have *Ord x*
if $x \in elts\ (set\ (ordermap\ A\ r\ `\ \{y \in A.\ (y, u) \in r\}))$ **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 $y \le set$ (*ordermap A r ‘ A*)
if $y =$ *ordermap A r x* $x \in A$ *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 $z \le y$ **if** $y \in$ *ordermap A r ‘ A* $z \in$ *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 †: $\{y.\ y = x \wedge (y, x) \in r\} = \{\}$
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*: $(w, x) \in r$ **and** *wf r* $w \in A$ *small A*
shows *ordermap A r w* $\in$ *elts* (*ordermap A r x*)
proof −
have *small* $\{a \in A.\ (a, x) \in r\} \wedge w \in A \wedge (w, x) \in r$
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* $\in$ *elts* (*ordermap A r x*) *wf r total-on A r* $x \in A$ $y \in A$ *small A*

shows $(y, x) \in r$
proof (*cases* $x = y$)
case *True*
then show *?thesis*
using *assms*(*1*) *mem-not-refl* **by** *blast*
next
case *False*
then consider $(x,y) \in r \mid (y,x) \in r$
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* $x \in A$ $y \in A$ *small A*
shows *ordermap A r y* $\in$ *elts* (*ordermap A r x*) $\longleftrightarrow$ $(y, x) \in r$
by (*metis assms converse-ordermap-mono ordermap-mono*)

lemma *ordermap-surj*: *elts* (*ordertype A r*) $\subseteq$ *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*]:
$[\![x \in A;\ y \in A;$ *wf r*; *total-on A r*; *small A*$]\!] \Longrightarrow$ *ordermap A r x* = *ordermap A r y* $\longleftrightarrow x = y$
by (*metis bij-betw-iff-bijections ordermap-bij*)

lemma *inv-into-ordermap*: $\alpha \in$ *elts* (*ordertype A r*) $\Longrightarrow$ *inv-into A* (*ordermap A r*) $\alpha \in A$
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 $A \approx$ *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*) $\approx$ *A*
  using *assms eqpoll-def eqpoll-sym ordermap-bij* **by** *blast*

lemma *ordertype-eqpoll*:
  assumes *small A*
  shows *elts* (*ordertype A VWF*) $\approx$ *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*]: $\llbracket$*small X*; *wf r*$\rrbracket$ $\Longrightarrow$ *ordertype X r* = *0* $\longleftrightarrow$ *X* = {}
  by (*metis ordertype-def elts-of-set replacement image-is-empty zero-V-def*)

lemma *ordermap-mono-less*:
  assumes (*w*, *x*) $\in$ *r*
    and *wf r trans r*
    and *w* $\in$ *A x* $\in$ *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*) $\in$ *r* $\vee$ *w*=*x*
    and *wf r trans r*
    and *w* $\in$ *A x* $\in$ *A*
    and *small A*
   shows *ordermap A r w* $\leq$ *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* $\leq$ *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* $\leq$ *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* ∈ *X* **and** *zx*: (*z*,*x*) ∈ *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* ⊆ *Y small Y*
  shows *ordertype X VWF* ≤ *ordertype Y VWF*
  using *assms* **by** (*simp add*: *ordertype-mono*)

lemma *ordertype-UNION-ge*:
  assumes *A* ∈ $\mathcal{A}$ *wf r trans r* $\mathcal{A}$ ⊆ *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: α ∈ *elts* (*ordertype M r*) **and** β: β ∈ *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 ... $\leq$ *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*) $\beta \longleftrightarrow \alpha \leq \beta$
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*) $\beta \longleftrightarrow \alpha < \beta$
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*) $\rightarrow$ *M strict-mono-on* (*elts* (*ordertype M VWF*)) *f*
proof
show *inv-into M* (*ordermap M VWF*) $\in$ *elts* (*ordertype M VWF*) $\rightarrow M$
by (*meson Pi-I' in-mono inv-into-into ordermap-surj*)

show *strict-mono-on* (*elts* (*ordertype M VWF*)) (*inv-into M* (*ordermap 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* ` ($\pi$ ` $A \cap \{y.\ (y,\ \pi\ x) \in s\}$) = ($f \circ \pi$) ` ($A \cap \{y.\ (y,\ x) \in r\}$) **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 $A \subseteq ON$ π ' $A \subseteq ON$ *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* $\leq$ *ordertype B s* $\longleftrightarrow$
($\exists f \in A \to B$. *inj-on f A* $\wedge$ ($\forall x \in A.\ \forall y \in A.\ ((x,y) \in r \longrightarrow (f\,x, f\,y) \in s)$))
(**is** *?lhs* = *?rhs*)
proof
assume *L*: *?lhs*
define *f* **where** $f \equiv$ *inv-into B* (*ordermap B s*) $\circ$ *ordermap A r*
show *?rhs*
proof (*intro bexI conjI ballI impI*)
have *AB*: *elts* (*ordertype A r*) $\subseteq$ *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*) $\in B$ **if** $i \in A$ **for** *i*
by (*meson L* ‹*small A*› *inv-into-into ordermap-in-ordertype ordermap-surj subsetD that vsubsetD*)
then show $f \in A \to B$
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 $x \in A$ $y \in A$
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* ‹$x \in A$› ‹$y \in A$› *bijA bij-betw-inv-into-left*)
qed
next
fix *x y*
assume $x \in A$ $y \in A$ **and** $(x, y) \in r$
have ‡: *ordermap A r y* $\in$ *ordermap B s* ‘ *B*
by (*meson L* ‹$y \in A$› ‹*small A*› *in-mono ordermap-in-ordertype ordermap-surj vsubsetD*)
moreover have †: $\bigwedge x$. *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* $\in$ *ordermap B s* ‘ *B*
by (*meson L* ‹$x \in A$› ‹*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* ‹$(x, y) \in r$› ‹$x \in A$› ‹*small A*› *ordermap-mono*)
ultimately show $(f\,x, f\,y) \in s$
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* = *α* ⟷
(∃ *f*. *bij-betw f A* (*elts α*) ∧ (∀ *x* ∈ *A*. ∀ *y* ∈ *A*. *f x* < *f y* ⟷ (*x*,*y*) ∈ *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 ⊆ elts α
  shows ordertype X VWF ≤ α
  by (metis assms ordertype-VWF-mono ordertype-eq-Ord small-elts)

lemma ordertype-inc-le-Ord:
  assumes small A Ord α
    and π: ⋀x y. ⟦x∈A; y∈A; (x,y) ∈ r⟧ ⟹ π x < π y
    and wf r total-on A r
    and sub: π ` A ⊆ elts α
  shows ordertype A r ≤ α
proof −
  have ⋀x y. ⟦x∈A; y∈A; (x,y) ∈ r⟧ ⟹ (π x, π y) ∈ 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: β ≤ α ordertype H VWF = α and small H Ord β
  obtains G where G ⊆ H ordertype G VWF = β
proof (intro exI conjI that)
  let ?f = ordermap H VWF
  show ‡: inv-into H ?f ` elts β ⊆ H
    unfolding image-subset-iff
    by (metis α inv-into-into ordermap-surj subsetD vsubsetD)
  have ∃f. bij-betw f (inv-into H ?f ` elts β) (elts β) ∧ (∀x∈inv-into H ?f ` elts β.
∀y∈inv-into H ?f ` elts β. (f x < f y) = ((x, y) ∈ 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 ∈ inv-into H ?f ` elts β
      and y: y ∈ inv-into H ?f ` elts β
    show ?f x < ?f y if (x,y) ∈ VWF
      using that ‡ ‹small H› in-mono ordermap-mono-less x y by fastforce
    show (x,y) ∈ 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) ∘ 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: ⋀α. α ∈ elts γ ⟹ ordermap B s (inv-into B (ordermap B s) α)
= α
      by (meson bijB bij-betw-inv-into-right)
```

have *ordermap-A-γ*: $\bigwedge a.\ a \in A \Longrightarrow$ *ordermap A r a* $\in$ *elts* γ
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 ∃ *y*∈*A*. (*x*,*y*) ∈ *r* **if** *x* ∈ *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-Int1 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* ⟷ (∃ *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* $y \le \alpha$ **and** *U*: $U \subseteq$ *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*: $\{u \in U.\ (u, y) \in VWF\}$ = *elts y* $\cap$ *U*
apply (*simp add*: *set-eq-iff*)
by (*meson Ord-in-Ord Ord-mem-iff-lt VWF-iff-Ord-less assms subsetD*)
have *2*: $\{u \in$ *insert* α $U.\ (u, y) \in VWF\}$ = *elts y* $\cap$ *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*: $U \subseteq$ *elts* α
shows *ordertype* (*insert* α *U*) *VWF* = *succ* (*ordertype U VWF*)
proof −
have †: $\{y \in$ *insert* α $U.\ (y, \alpha) \in VWF\} = U$ $\{y \in U.\ (y, \alpha) \in VWF\} = U$
using *Ord-in-Ord OrdmemD assms* **by** *auto*
have *eq*: $\bigwedge x.\ x \in U \Longrightarrow$ *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 ordermap-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 {..<card M} h
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 {..<card M} ?h
    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
{..<card A} h
     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 an arbitrary HOL set

```
definition gcard :: 'a set ⇒ V
  where gcard X ≡ if small X then (LEAST i. Ord i ∧ elts i ≈ X) else 0
```

2.11 Cardinality of a set

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

lemma gcard-eq-vcard [simp]: gcard (elts x) = vcard x
  by (simp add: vcard-def gcard-def)

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 gcardinal-cong:
  assumes X ≈ Y shows gcard X = gcard Y
proof −
  have (LEAST i. Ord i ∧ elts i ≈ X) = (LEAST i. Ord i ∧ elts i ≈ Y)
    by (meson eqpoll-sym eqpoll-trans assms)
  then show ?thesis
    unfolding gcard-def
    by (meson eqpoll-sym small-eqcong assms)
qed

lemma vcard-set-image: inj-on f (elts x) ⟹ vcard (set (f ‘ elts x)) = vcard x
  by (simp add: cardinal-cong)

lemma gcard-image: inj-on f X ⟹ gcard (f ‘ X) = gcard X
  by (simp add: gcardinal-cong)

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* $\wedge$ *elts i* $\approx$ *elts* κ)
  by (*metis Ord-Least*)
then show *?thesis*
  using *Card-def vcard-def assms* **by** *auto*
qed

lemma *cardinal-eqpoll*: *elts* (*vcard a*) $\approx$ *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* $\in$ *elts A* $\rightarrow$ *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*

lemma *subset-smaller-vcard*:
assumes $\kappa \leq$ *vcard x Card* κ
obtains *y* **where** $y \leq x$ *vcard y* = κ
proof −
obtain φ **where** φ: *bij-betw* φ (*elts* (*vcard x*)) (*elts x*)
  using *cardinal-eqpoll eqpoll-def* **by** *blast*
show *thesis*
proof
  let *?y* = *ZFC-in-HOL.set* (φ ‘ *elts* κ)
  show *?y* $\leq x$
    by (*meson* φ *assms bij-betwE set-image-le-iff small-elts vsubsetD*)
  show *vcard ?y* = κ
    by (*metis vcard-set-image Card-def assms bij-betw-def bij-betw-subset* φ *less-eq-V-def*)
qed
qed

every natural number is a (finite) cardinal

lemma *nat-into-Card*:
assumes $\alpha \in$ *elts* ω **shows** *Card*(α)
proof (*unfold Card-def vcard-def*, *rule sym*)
obtain *n* **where** *n*: α = *ord-of-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* α
  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*)

lemma *gcard-big-0*: ¬ *small X* ⟹ *gcard X* = *0*
by (*simp add*: *gcard-def*)

lemma *gcard-eq-card*:
assumes *finite X* **shows** *gcard X* = *ord-of-nat* (*card X*)
proof −
have ⋀*y*. *Ord y* ∧ *elts y* ≈ *X* ⟹ *ord-of-nat* (*card X*) ≤ *y*
by (*metis assms eqpoll-finite-iff eqpoll-iff-card order-le-less ordertype-VWF-finite-nat ordertype-eq-Ord*)
then have (*LEAST i. Ord i* ∧ *elts i* ≈ *X*) = *ord-of-nat* (*card X*)
by (*simp add*: *assms eqpoll-iff-card finite-Ord-omega Least-equality*)
with *assms* **show** *?thesis*
by (*simp add*: *finite-imp-small gcard-def*)
qed

lemma *gcard-empty-0* [*simp*]: *gcard* {} = *0*
by (*simp add*: *gcard-eq-card*)

lemma *gcard-single-1* [*simp*]: *gcard* {*x*} = *1*
by (*simp add*: *gcard-eq-card one-V-def*)

lemma *Card-gcard* [*iff*]: *Card* (*gcard X*)
by (*metis Card-0 Card-def cardinal-idem gcard-big-0 gcardinal-cong small-eqpoll gcard-eq-vcard*)

lemma *gcard-eqpoll*: *small* $X \Longrightarrow$ *elts* (*gcard* X) $\approx X$
by (*metis cardinal-eqpoll eqpoll-trans gcard-eq-vcard gcardinal-cong small-eqpoll*)

lemma *lepoll-imp-gcard-le*:
assumes $A \lesssim B$ *small* B
shows *gcard* $A \leq$ *gcard* B
proof −
have *elts* (*gcard* A) $\approx A$ *elts* (*gcard* B) $\approx B$
by (*meson assms gcard-eqpoll lepoll-small*)+
with ‹$A \lesssim B$› **show** *?thesis*
by (*metis Ord-cardinal Ord-linear2 eqpoll-sym gcard-eq-vcard gcardinal-cong lepoll-antisym*
lepoll-trans2 less-V-def less-eq-V-def subset-imp-lepoll)
qed

lemma *gcard-image-le*:
assumes *small* A **shows** *gcard* (f ‘ A) $\leq$ *gcard* A
using *assms image-lepoll lepoll-imp-gcard-le* **by** *blast*

lemma *subset-imp-gcard-le*:
assumes $A \subseteq B$ *small* B
shows *gcard* $A \leq$ *gcard* B
by (*simp add*: *assms lepoll-imp-gcard-le subset-imp-lepoll*)

lemma *gcard-le-lepoll*: ⟦*gcard* $A \leq \alpha$; *small* A⟧ $\Longrightarrow A \lesssim$ *elts* α
by (*meson eqpoll-sym gcard-eqpoll lepoll-trans1 less-eq-V-def subset-imp-lepoll*)

2.12 Cardinality of a set

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* $\kappa)$
by (*metis CardI Card-def Card-is-Ord not-less-Ord-Least vcard-def*)

lemma *Card-ω* [*iff*]: *Card* ω
by (*meson CardI Ord-ω eqpoll-finite-iff infinite-Ord-omega infinite-ω leD*)

lemma *lt-Card-imp-lesspoll*: ⟦$i < a$; *Card* a; *Ord* i⟧ $\Longrightarrow$ *elts* $i \prec$ *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* $a \lesssim$ *elts* b **shows** *vcard* $a \leq$ *vcard* b
using *assms lepoll-imp-gcard-le* **by** *fastforce*

lemma *lepoll-cardinal-le*: ⟦*elts* $A \lesssim$ *elts* i; *Ord* i⟧ $\Longrightarrow$ *vcard* $A \leq i$
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*: $vcard\ A \le \alpha \Longrightarrow elts\ A \lesssim elts\ \alpha$
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 lepoll-imp-Card-le less-V-def 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
by (*metis Card-def Sup-V-def ZFC-in-HOL.Sup-upper cardinal-le-lepoll j*(*1*) *not-less-0*)
also have $\ldots \approx elts\ j$ **by** (*rule eqpoll-sym* [*OF eqp*])
also have $\ldots \prec elts\ c$ **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*: $(\bigwedge x.\ x \in A \Longrightarrow Card(K\ x))$ ==> $Card(Sup\ (K\ \text{‘}\ A))$
by *blast*

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

definition
transrec3 :: $[V,\ [V,V]{\Rightarrow}V,\ [V,V{\Rightarrow}V]{\Rightarrow}V,\ V] \Rightarrow V$ **where**
transrec3 a b c $\equiv$
transrec ($\lambda r\ x.$
if x=0 then a
else if Limit x then c x ($\lambda y \in elts\ x.\ r\ y$)
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 $\Longrightarrow$ *transrec3 a b c i* = *c i* ($\lambda j \in$ *elts i*. *transrec3 a b c j*)
  unfolding *transrec3-def*
  by (*subst transrec*) *auto*

2.14 Cardinal Addition

definition *cadd* :: $[V,V] \Rightarrow V$ (**infixl** ‹⊕› *65*)
  where $\kappa \oplus \mu \equiv$ *vcard* ($\kappa \uplus \mu$)

2.14.1 Cardinal addition is commutative

lemma *vsum-commute-eqpoll*: *elts* ($a \uplus b$) $\approx$ *elts* ($b \uplus a$)
proof −
  have *bij-betw* ($\lambda z \in$ *elts* ($a \uplus b$). *sum-case Inr Inl z*) (*elts* ($a \uplus b$)) (*elts* ($b \uplus a$))
    unfolding *bij-betw-def*
  proof (*intro conjI inj-onI*)
    show *restrict* (*sum-case Inr Inl*) (*elts* ($a \uplus b$)) ‘ *elts* ($a \uplus b$) = *elts* ($b \uplus a$)
     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*: $i \oplus j = j \oplus i$
  by (*simp add*: *cadd-def cardinal-cong vsum-commute-eqpoll*)

2.14.2 Cardinal addition is associative

lemma *sum-assoc-bij*:
   bij-betw ($\lambda z \in$ *elts* (($a \uplus b$)$\uplus c$). *sum-case*(*sum-case Inl* (λy. *Inr*(*Inl y*))) (λy. *Inr*(*Inr y*)) *z*)
      (*elts* (($a \uplus b$)$\uplus c$)) (*elts* ($a \uplus$ ($b \uplus c$)))
  by (*rule-tac f′* = *sum-case* (λx. *Inl* (*Inl x*)) (*sum-case* (λx. *Inl* (*Inr x*)) *Inr*)
      in *bij-betw-byWitness*) *auto*

lemma *sum-assoc-eqpoll*: *elts* (($a \uplus b$)$\uplus c$) $\approx$ *elts* ($a \uplus$ ($b \uplus c$))
  unfolding *eqpoll-def* **by** (*metis sum-assoc-bij*)

lemma *elts-vcard-vsum-eqpoll*: *elts* (*vcard* ($i \uplus j$)) $\approx$ *Inl* ‘ *elts i* $\cup$ *Inr* ‘ *elts j*
proof −

have *elts* $(i \uplus j) \approx$ *Inl* ‘ *elts* $i \cup$ *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 $\ldots \approx$ *elts* $(i \uplus (j \uplus k))$
    by (*rule sum-assoc-eqpoll*)
  also have $\ldots \approx$ *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

lemma *cadd-left-commute*: $j \oplus (i \oplus k) = i \oplus (j \oplus k)$
  using *cadd-assoc cadd-commute* **by** *force*

lemmas *cadd-ac* = *cadd-assoc cadd-commute cadd-left-commute*

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

Monotonicity of addition

lemma *cadd-le-mono*: $[\![\kappa' \le \kappa; \mu' \le \mu]\!] \Longrightarrow \kappa' \oplus \mu' \le \kappa \oplus \mu$
  unfolding *cadd-def*

by (*metis* (*no-types*) *lepoll-imp-Card-le less-eq-V-def subset-imp-lepoll sum-subset-iff*)

2.15 Cardinal multiplication

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

2.15.1 Cardinal multiplication is commutative

lemma *prod-bij*: ⟦*bij-betw f A C*; *bij-betw g B D*⟧
$\Longrightarrow$ *bij-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* × *elts j*) ≈ ($\lambda(x,\ y).\ \langle x,\ y\rangle$) ‘ (*elts j* × *elts i*)
by (*simp add*: *times-commute-eqpoll*)
then show *?thesis*
unfolding *cmult-def*
using *cardinal-cong elts-VSigma* **by** *auto*
qed

2.15.2 Cardinal multiplication is associative

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

lemma *elts-cmult*: *elts* ($\kappa' \otimes \kappa$) ≈ *elts* κ' × *elts* κ
by (*simp add*: *cmult-def elts-vcard-VSigma-eqpoll*)

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*)) × *elts k* ≈ (*elts i* × *elts j*) × *elts k*
by (*blast intro*: *times-eqpoll-cong elts-vcard-VSigma-eqpoll cardinal-eqpoll*)
also have ... ≈ *elts i* × (*elts j* × *elts k*)
by (*rule times-assoc-eqpoll*)
also have ... ≈ *elts i* × *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*)) ($\lambda z.\ k$)) ≈ *elts* (*VSigma i* ($\lambda z.$ *vcard* (*vtimes j k*)))

by (*simp add*: *elts-VSigma*)
qed

2.15.3 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** $1 \otimes \kappa = \kappa$
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

2.16 Some inequalities for multiplication

lemma *cmult-square-le*: **assumes** *Card* κ **shows** $\kappa \le \kappa \otimes \kappa$
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 α ≠ 0 shows κ ≤ κ ⊗ α
proof −
  have κ = vcard κ
    using Card-def ‹Card κ› by blast
  also have ... ≤ vcard (vtimes κ α)
    apply (rule lepoll-imp-Card-le)
    apply (simp add: elts-VSigma)
    by (metis ZFC-in-HOL.ext ‹α ≠ 0› elts-0 lepoll-times1)
  also have ... = κ ⊗ α
    by (simp add: cmult-def)
  finally show ?thesis .
qed
```

Monotonicity of multiplication

```
lemma cmult-le-mono: ⟦κ' ≤ κ; μ' ≤ μ⟧ ⟹ κ' ⊗ μ' ≤ κ ⊗ μ
  unfolding cmult-def
 by (auto simp: elts-VSigma intro!: lepoll-imp-Card-le times-lepoll-mono subset-imp-lepoll)

lemma vcard-Sup-le-cmult:
  assumes small U and κ: ⋀x. x ∈ U ⟹ vcard x ≤ κ
  shows vcard (⨆ U) ≤ vcard (set U) ⊗ κ
proof −
  have ∃f. f ∈ elts x → elts κ ∧ inj-on f (elts x) if x ∈ U for x
   using κ [OF that] by (metis cardinal-le-lepoll image-subset-iff-funcset lepoll-def)
  then obtain φ where φ: ⋀x. x ∈ U ⟹ (φ x) ∈ elts x → elts κ ∧ inj-on (φ
x) (elts x)
    by metis
  define u where u ≡ λy. @x. x ∈ U ∧ y ∈ elts x
  have u: u y ∈ U ∧ y ∈ elts (u y) if y ∈ ⋃(elts ` U) for y
    unfolding u-def by (metis (mono-tags, lifting)that someI2-ex UN-iff)
  define ψ where ψ ≡ λy. (u y, φ (u y) y)
  have U: elts (vcard (set U)) ≈ U
    by (metis ‹small U› cardinal-eqpoll elts-of-set)
  have elts (⨆ U) = ⋃(elts ` U)
    using ‹small U› by blast
  also have ... ≲ U × elts κ
    unfolding lepoll-def
  proof (intro exI conjI)
    show inj-on ψ (⋃ (elts ` U))
      using φ u by (smt (verit) ψ-def inj-on-def prod.inject)
    show ψ ` ⋃ (elts ` U) ⊆ U × elts κ
      using φ u by (auto simp: ψ-def)
  qed
  also have ... ≈ elts (vcard (set U) ⊗ κ)
    using U elts-cmult eqpoll-sym eqpoll-trans times-eqpoll-cong by blast
  finally have elts (⨆ U) ≲ elts (vcard (set U) ⊗ κ) .
  then show ?thesis
    by (simp add: cmult-def lepoll-cardinal-le)
```

qed

2.17 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* = λz. *if z=Inl a then a*$\uplus$*b else z*
  let *?g* = λz. *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 ... = *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* = λz. *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

lemma *vcard-sup*: *vcard* $(x \sqcup y) \leq$ *vcard* $x \oplus$ *vcard* y
proof $-$
  have *elts* $(x \sqcup y) \lesssim$ *elts* $(x \uplus y)$
    unfolding *lepoll-def*
  proof (*intro exI conjI*)
    let $?f = \lambda z.$ *if* $z \in$ *elts* x *then Inl* z *else Inr* z
    show *inj-on* $?f$ (*elts* $(x \sqcup y)$)
      by (*simp add*: *inj-on-def*)
    show $?f$ ‘ *elts* $(x \sqcup y) \subseteq$ *elts* $(x \uplus y)$
      by *force*
  qed
  then show *?thesis*
    using *cadd-ac*
    by (*metis cadd-def cardinal-cong cardinal-idem lepoll-imp-Card-le vsum-0-eqpoll*)
qed

2.18 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* ≺ *elts* (*VPow A*)
    by (*simp add*: *elts-VPow down inj-on-def lesspoll-Pow-self*)
qed

lemma *greater-Card*:
  assumes *Card* κ **shows** κ < *vcard* (*VPow* κ)
proof −
  have κ = *vcard* κ
    using *Card-def assms* **by** *blast*
  also have . . . < *vcard* (*VPow* κ)
  proof (*rule lesspoll-imp-Card-less*)
    show *elts* κ ≺ *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*]: κ < *csucc* κ
proof −
  have *Card* (*csucc* κ) ∧ κ < *csucc* κ
    unfolding *csucc-def*
  proof (*rule Ord-LeastI2*)
    show *Card* (*vcard* (*VPow* κ)) ∧ κ < (*vcard* (*VPow* κ))
      using *Card-def assms greater-Card* **by** *auto*
  qed *auto*
  then show *Card* (*csucc* κ) κ < *csucc* κ
    by *auto*
qed

lemma *le-csucc*:
  assumes *Card* κ **shows** κ ≤ *csucc* κ
  by (*simp add*: *assms less-csucc less-imp-le*)

lemma *csucc-le*: ⟦*Card* μ; κ ∈ *elts* μ⟧ ⟹ *csucc* κ ≤ μ
  unfolding *csucc-def*
  by (*simp add*: *Card-is-Ord Ord-Least-le OrdmemD*)

lemma *finite-csucc*: *a* ∈ *elts* ω ⟹ *csucc a* = *succ a*
  unfolding *csucc-def*
  proof (*rule Least-equality*)
 show *Ord* (*ZFC-in-HOL.succ a*) ∧ *Card* (*ZFC-in-HOL.succ a*) ∧ *a* < *ZFC-in-HOL.succ*
a

```
    if a ∈ elts ω
    using that by (auto simp: less-V-def less-eq-V-def nat-into-Card)
  show ZFC-in-HOL.succ a ≤ y
    if a ∈ elts ω
      and Ord y ∧ Card y ∧ 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 ∉ 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 α < csucc κ ⟷ vcard α ≤ κ
proof
  show vcard α ≤ κ if α < csucc κ
  proof −
    have vcard α ≤ csucc κ
      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* κ **if** *vcard* $\alpha \le \kappa$
proof −
have ¬ *csucc* $\kappa \le \alpha$
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* κ'; *Card* $\kappa]\!] \Longrightarrow (\kappa' <$ *csucc* $\kappa) = (\kappa' \le \kappa)$
by (*simp add*: *lt-csucc-iff Card-cardinal-eq Card-is-Ord*)

lemma *csucc-lt-csucc-iff*: $[\![$*Card* κ'; *Card* $\kappa]\!] \Longrightarrow$ (*csucc* $\kappa' <$ *csucc* $\kappa) = (\kappa' < \kappa)$
by (*metis Card-csucc Card-is-Ord Card-lt-csucc-iff Ord-not-less*)

lemma *csucc-le-csucc-iff*: $[\![$*Card* κ'; *Card* $\kappa]\!] \Longrightarrow$ (*csucc* $\kappa' \le$ *csucc* $\kappa) = (\kappa' \le \kappa)$
by (*metis Card-csucc Card-is-Ord Card-lt-csucc-iff Ord-not-less*)

lemma *csucc-0* [*simp*]: *csucc 0* = *1*
by (*simp add*: *finite-csucc one-V-def*)

lemma *Card-Un* [*simp*,*intro*]:
assumes *Card x Card y* **shows** *Card*($x \sqcup y$)
by (*metis Card-is-Ord Ord-linear-le assms sup.absorb2 sup.orderE*)

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 $\omega \le y$

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

2.19 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 $\mu \le \kappa$ $\mu \neq 0$
shows $\kappa \otimes \mu = \kappa$
proof (*rule order-antisym*)
have $\kappa \otimes \mu \le \kappa \otimes \kappa$
by (*simp add*: *assms*(*2*) *cmult-le-mono*)
also have $\ldots \le \kappa$
by (*simp add*: *InfCard-csquare-eq assms*(*1*))
finally show $\kappa \otimes \mu \le \kappa$ **.**
show $\kappa \le \kappa \otimes \mu$
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*: $[\![$*InfCard* κ; *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*:
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* ($n \uplus$ *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* $\uplus$ *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 ... = κ
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* κ; $\mu \le \kappa]\!] \Longrightarrow \kappa \oplus \mu = \kappa$
by (*metis InfCard-cdouble-eq InfCard-def antisym cadd-le-mono cadd-le-self*)

lemma *InfCard-cadd-eq*: $[\![$*InfCard* κ; *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*)

lemma *csucc-le-Card-iff*: $[\![$*Card* κ'; *Card* $\kappa]\!] \Longrightarrow$ *csucc* $\kappa' \le \kappa \longleftrightarrow \kappa' < \kappa$
by (*metis Card-csucc Card-is-Ord Card-lt-csucc-iff Ord-not-le*)

lemma *cadd-InfCard-le*:
assumes $\alpha \le \kappa$ $\beta \le \kappa$ *InfCard* κ
shows $\alpha \oplus \beta \le \kappa$
using *assms* **by** (*metis InfCard-cdouble-eq cadd-le-mono*)

lemma *cmult-InfCard-le*:
assumes $\alpha \le \kappa$ $\beta \le \kappa$ *InfCard* κ
shows $\alpha \otimes \beta \le \kappa$
using *assms*
by (*metis InfCard-csquare-eq cmult-le-mono*)

2.20 The Aleph-seqence

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

definition *Aleph* :: $V \Rightarrow V$ (‹ℵ-› [*90*] *90*)
where *Aleph* $\equiv$ *transrec* ($\lambda f\ x.\ \omega \sqcup \bigsqcup((\lambda y.\ csucc(f\ y))$ ‘ *elts x*))

lemma *Aleph*: *Aleph* $\alpha = \omega \sqcup (\bigsqcup y{\in}elts\ \alpha.\ csucc\ (Aleph\ y))$
by (*simp add*: *Aleph-def transrec*[*of* - α])

lemma *InfCard-Aleph* [*simp*, *intro*]: *InfCard*(*Aleph x*)
proof (*induction x rule*: *eps-induct*)
case (*step x*)
then show *?case*
by (*simp add*: *Aleph* [*of x*] *InfCard-def Card-UN step*)
qed

lemma *Card-Aleph* [*simp*, *intro*]: *Card*(*Aleph x*)
using *InfCard-def* **by** *auto*

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

lemma *mem-Aleph-succ*: $\aleph\alpha \in$ *elts* (*Aleph* (*succ* α))
apply (*simp add*: *Aleph* [*of succ* α])
by (*meson InfCard-Aleph Card-csucc Card-is-Ord InfCard-def Ord-mem-iff-lt less-csucc*)

lemma *Aleph-lt-succD* [*simp*]: $\aleph\alpha <$ *Aleph* (*succ* α)
by (*simp add*: *InfCard-is-Limit Limit-is-Ord OrdmemD mem-Aleph-succ*)

lemma *Aleph-succ* [*simp*]: *Aleph* (*succ x*) = *csucc* (*Aleph x*)
proof (*rule antisym*)
show *Aleph* (*ZFC-in-HOL.succ x*) $\leq$ *csucc* (*Aleph x*)
apply (*simp add*: *Aleph* [*of succ x*])
by (*metis Aleph InfCard-Aleph InfCard-def Sup-V-insert le-csucc le-sup-iff order-refl*
replacement small-elts)
show *csucc* (*Aleph x*) $\leq$ *Aleph* (*ZFC-in-HOL.succ x*)
by (*force simp add*: *Aleph* [*of succ x*])
qed

lemma *csucc-Aleph-le-Aleph*: $\alpha \in$ *elts* $\beta \Longrightarrow$ *csucc* ($\aleph\alpha$) $\leq \aleph\beta$
by (*metis Aleph ZFC-in-HOL.SUP-le-iff replacement small-elts sup-ge2*)

lemma *Aleph-in-Aleph*: $\alpha \in$ *elts* $\beta \Longrightarrow \aleph\alpha \in$ *elts* ($\aleph\beta$)
using *csucc-Aleph-le-Aleph mem-Aleph-succ* **by** *auto*

lemma *Aleph-Limit*:
assumes *Limit* γ
shows *Aleph* $\gamma = \bigsqcup$ (*Aleph* ‘ *elts* γ)
proof –
have [*simp*]: $\gamma \neq 0$

```
    using assms by auto
  show ?thesis
  proof (rule antisym)
    show Aleph γ ≤ ⨆ (Aleph ‘ elts γ)
      apply (simp add: Aleph [of γ])
     by (metis (mono-tags, lifting) Aleph-0 Aleph-succ Limit-def ZFC-in-HOL.SUP-le-iff

          ZFC-in-HOL.Sup-upper assms imageI replacement small-elts)
    show ⨆ (Aleph ‘ elts γ) ≤ Aleph γ
      apply (simp add: cSup-le-iff)
     by (meson InfCard-Aleph InfCard-def csucc-Aleph-le-Aleph le-csucc order-trans)
  qed
qed

lemma Aleph-increasing:
  assumes ab: α < β Ord α Ord β shows ℵα < ℵβ
 by (meson Aleph-in-Aleph InfCard-Aleph Card-iff-initial InfCard-def Ord-mem-iff-lt
assms)

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 ≤ Aleph 0
  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

lemma Aleph-csquare-eq [simp]: ℵα ⊗ ℵα = ℵα
  using InfCard-csquare-eq by auto
```

lemma *vcard-Aleph* [*simp*]: *vcard* ($\aleph\alpha$) = $\aleph\alpha$
  using *Card-def InfCard-Aleph InfCard-def* **by** *auto*

lemma *omega-le-Aleph* [*simp*]: $\omega \leq \aleph\alpha$
  using *InfCard-def* **by** *auto*

lemma *finite-iff-less-Aleph0*: *finite* (*elts x*) $\longleftrightarrow$ *vcard x* $< \omega$
proof
  show *finite* (*elts x*) $\Longrightarrow$ *vcard x* $< \omega$
   by (*metis Card-ω Card-def finite-lesspoll-infinite infinite-ω lesspoll-imp-Card-less*)
  show *vcard x* $< \omega \Longrightarrow$ *finite* (*elts x*)
   by (*meson Ord-cardinal cardinal-eqpoll eqpoll-finite-iff infinite-Ord-omega less-le-not-le*)
qed

lemma *countable-iff-vcard-less1*: *countable* (*elts x*) $\longleftrightarrow$ *vcard x* $< \aleph 1$
  by (*simp add*: *countable-iff-le-Aleph0 lt-csucc-iff one-V-def*)

lemma *countable-infinite-vcard*: *countable* (*elts x*) $\wedge$ *infinite* (*elts x*) $\longleftrightarrow$ *vcard x*
= $\aleph 0$
 by (*metis Aleph-0 countable-iff-le-Aleph0 dual-order.refl finite-iff-less-Aleph0 less-V-def*)

2.21 The ordinal $\omega 1$

abbreviation $\omega 1 \equiv$ *Aleph 1*

lemma *Ord-ω1* [*simp*]: *Ord* $\omega 1$
  by (*metis Ord-cardinal vcard-Aleph*)

lemma *omega-ω1* [*iff*]: $\omega \in$ *elts* $\omega 1$
  by (*metis Aleph-0 mem-Aleph-succ one-V-def*)

lemma *ord-of-nat-ω1* [*iff*]: *ord-of-nat n* $\in$ *elts* $\omega 1$
  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-V* :: $V \Rightarrow V \Rightarrow V$
  where *plus-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 ≠ 0 ⟹ x + (⨆z∈elts y. f z) = (⨆z∈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 α ⟹ x + (⨆z∈elts α. f z) = (⨆z∈elts α. x + f z)
  using add-Sup-distrib by force
```

Proposition 3.3(ii)

```
instantiation V :: monoid-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 ⊔ set ((+) (a + b) ‘ elts c)
      by (metis plus)
    also have ... = a ⊔ lift a b ⊔ set ((λu. 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-of-nat* (*m*+*n*) = *ord-of-nat m* + *ord-of-nat n*
  by (*induction n*) (*auto simp*: *plus-V-succ-right*)

lemma *succ-0-plus-eq* [*simp*]:
  assumes *α* ∈ *elts ω*
  shows *succ 0* + *α* = *succ α*
proof −
  obtain *n* **where** *α* = *ord-of-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 *α* ∈ *elts ω β* ∈ *elts ω* **shows** *α*+*β* ∈ *elts ω*
proof −
  obtain *m n* **where** *α* = *ord-of-nat m β* = *ord-of-nat n*
    using *assms elts-ω* **by** *auto*
  then have *α*+*β* = *ord-of-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 $l \in elts\ x$ | *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 ⟹ y = z
proof (induction y arbitrary: z rule: eps-induct)
  case (step y)
  show ?case
  proof (intro vsubsetI order-antisym)
    show u ∈ elts z if u ∈ elts y for u
    proof −
      have x+u ∈ elts (lift x z)
        using lift-def step.prems that by fastforce
      then obtain v where v ∈ 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** $x{+}y \le x{+}z \longleftrightarrow y{\le}z$
  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** $x{+}y < x{+}z \longleftrightarrow y{<}z$
  by (*simp add*: *less-le-not-le*)

corollary *lift-le-self* [*simp*]: *lift* $x\ y \le x \longleftrightarrow y = 0$
  by (*auto simp*: *inf.absorb-iff2 lift-eq-lift lift-self-disjoint*)

lemma *succ-less-ω-imp*: *succ* $x < \omega \Longrightarrow x < \omega$
  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* $x\ y$) $\approx$ *elts* y
    using *eqpoll-def eqpoll-sym* **by** *blast*
qed

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

lemma *vcard-add*: *vcard* $(x + y)$ = *vcard* $x \oplus$ *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* $A \le \aleph 0$ *vcard* $B \le \aleph 0$
    using *assms countable-iff-le-Aleph0* **by** *blast+*
  then have *vcard* $(A{+}B) \le \aleph 0$
    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* $x \sqcup$ *lift* x (*TC* y)

proof (*induction y rule*: *eps-induct*)
case (*step y*)
have ∗: $\bigsqcup$ (*TC* ‘ (+) *x* ‘ *elts y*) = *TC x* ⊔ ($\bigsqcup$*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* ≤ ($\bigsqcup$*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*) ⊔ $\bigsqcup$ (*TC* ‘ *elts* (*x* + *y*))
using *TC* **by** *blast*
also have ... = *x* ⊔ *lift x y* ⊔ $\bigsqcup$ (*TC* ‘ *elts x*) ⊔ $\bigsqcup$ ((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* ⊔ $\bigsqcup$ (*TC* ‘ *elts x*) ⊔ *lift x y* ⊔ $\bigsqcup$ ((λ*u*. *TC x* ⊔ *lift x* (*TC u*)) ‘ *elts y*)
by (*simp add*: *sup-aci step.IH*)
also have ... = *TC x* ⊔ *lift x y* ⊔ $\bigsqcup$ ((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* ⊔ $\bigsqcup$ (*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* ⊔ (⨆*u*∈*elts y*. TC *x* ⊔ *lift x* (TC *u*))
    by (*simp add*: *TC-add*)
  also have ... = *lift x y* ⊔ TC *x* ⊔ (⨆*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*) = (⨆*u*∈*elts* (*x* ⊔ *ZFC-in-HOL.set* ((+) *x* ‘ *elts y*)). *succ* (*rank u*))
      by (*metis plus rank-Sup*)
    also have ... = (⨆*x*∈*elts x*. *succ* (*rank x*)) ⊔ (⨆*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 ... = (⨆*z* ∈ *elts y*. *rank x* + *succ* (*rank z*))
    proof −
      have *rank x* ≤ (⨆*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*!: *order-antisym*)
    qed
    also have ... = *rank x* + (⨆*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*: $A \neq 0 \Longrightarrow x + \bigsqcup(elts\ A) = (\bigsqcup z{\in}elts\ A.\ x + z)$
 by (*metis add-Sup-distrib image-ident image-image*)

lemma *add-Limit*: $Limit\ \alpha \Longrightarrow x + \alpha = (\bigsqcup z{\in}elts\ \alpha.\ x + z)$
 by (*metis Limit-add-Sup-distrib Limit-eq-Sup-self image-ident image-image*)

lemma *add-le-left*:
 assumes $Ord\ \alpha$ $Ord\ \beta$ **shows** $\beta \leq \alpha{+}\beta$
 using ‹$Ord\ \beta$›
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*: $\mu = (\bigsqcup \beta \in elts\ \mu.\ \beta)$
  by (*simp add*: *Limit-eq-Sup-self*)
 also have $\ldots \leq (\bigsqcup \beta \in elts\ \mu.\ \alpha + \beta)$
  using *Limit.IH* **by** *auto*
 also have $\ldots = \alpha + (\bigsqcup \beta \in elts\ \mu.\ \beta)$
  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 ⊴ x
  by (metis (no-types) add.right-neutral vle-def)

lemma vle-antisym: ⟦x ⊴ y; y ⊴ x⟧ ⟹ x = y
  by (metis V-equalityI plus-eq-lift sup-ge1 vle-def vsubsetD)

lemma vle-trans [trans]: ⟦x ⊴ y; y ⊴ z⟧ ⟹ x ⊴ z
  by (metis add.assoc vle-def)

definition vle-comparable :: V ⇒ V ⇒ bool
  where vle-comparable x y ≡ x ⊴ y ∨ y ⊴ 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: ¬ (a ⊴ c ∨ c ⊴ a)
  let ?φ = λx. ∀z. a+x ≠ c+z
  have ?φ 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 ∈ elts x
        using trad-foundation by blast
      show ?thesis
      proof clarsimp
        fix z
        assume eq: a + x = c + z
        then have z ≠ 0
          using vle-def non by auto
        have av: a+v ∈ elts (a+x)
          by (simp add: ‹v ∈ elts x›)
        moreover have a+x = c ⊔ lift c z
          using eq plus-eq-lift by fastforce
        ultimately have a+v ∈ elts (c ⊔ lift c z)
          by simp
        moreover
        define u where u ≡ set (elts x − {v})
        have u: v ∉ elts u and xeq: x = vinsert v u
          using ‹v ∈ elts x› by (auto simp: u-def intro: order-antisym)
        have case1: a+v ∉ elts c
```

```
        proof
          assume avc: a + v ∈ elts c
          then have a ≤ c
             by clarify (metis Un-iff elts-sup-iff eq mem-not-sym mem-plus-V-E
plus-eq-lift)
          moreover have a ⊔ lift a x = c ⊔ lift c z
            using eq by (simp add: plus-eq-lift)
          ultimately have lift c z ≤ lift a x
             by (metis inf.absorb-iff2 inf-commute inf-sup-absorb inf-sup-distrib2
lift-self-disjoint sup.commute)
          also have ... = vinsert (a+v) (lift a u)
            by (simp add: lift-def vinsert-def xeq)
          finally have *: lift c z ≤ vinsert (a + v) (lift a u) .
          have lift c z ≤ lift a u
          proof −
            have a + v ∉ 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
  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 $\alpha \trianglelefteq \beta \longleftrightarrow \alpha \leq \beta$
proof
  show $\alpha \leq \beta$ **if** $\alpha \trianglelefteq \beta$
    using *that* **by** (*simp add*: *vle1*)
  show $\alpha \trianglelefteq \beta$ **if** $\alpha \leq \beta$
    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*: $[\![\alpha \leq \beta;\ Ord\ \alpha;\ Ord\ \beta;\ Ord\ \gamma]\!] \Longrightarrow \alpha+\gamma \leq \beta+\gamma$
by (*metis add-le-cancel-left add.assoc add-le-left le-Ord-diff*)

lemma *add-strict-mono*: $[\![\alpha < \beta;\ \gamma < \delta;\ Ord\ \alpha;\ Ord\ \beta;\ Ord\ \gamma;\ Ord\ \delta]\!] \Longrightarrow \alpha+\gamma < \beta+\delta$
by (*metis order.strict-implies-order add-less-cancel-left add-right-mono le-less-trans*)

lemma *add-right-strict-mono*: $[\![\alpha \leq \beta;\ \gamma < \delta;\ Ord\ \alpha;\ Ord\ \beta;\ Ord\ \gamma;\ Ord\ \delta]\!] \Longrightarrow \alpha+\gamma < \beta+\delta$
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** $\xi \in elts\ \mu\ \gamma = \beta + \xi$
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 $\alpha + odiff\ \beta\ \alpha = \beta$
  by (*simp add*: *assms vle-iff-le-Ord vle-imp-odiff-eq*)

lemma *Ord-odiff*:
  assumes *Ord* α *Ord* β **shows** *Ord* $(odiff\ \beta\ \alpha)$
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** $odiff\ \beta\ \alpha \leq \beta$
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*: $[\![\alpha \leq x;\ Ord\ x;\ Ord\ \alpha]\!] \Longrightarrow odiff\ x\ \alpha \leq y \longleftrightarrow x \leq \alpha + y$
  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*: $x * y = (\bigsqcup u{\in}elts\ y.\ lift\ (x * u)\ x)$
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*: $ord\text{-}of\text{-}nat\ (m{*}n) = ord\text{-}of\text{-}nat\ m * ord\text{-}of\text{-}nat\ n$
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** $\alpha = ord\text{-}of\text{-}nat\ m\ \beta = ord\text{-}of\text{-}nat\ n$
using *assms elts-ω* **by** *auto*
then have $\alpha{*}\beta = ord\text{-}of\text{-}nat\ (m{*}n)$
by (*simp add*: *ord-of-nat-mult*)
then show *?thesis*
by (*simp add*: *ω-def*)
qed

lemma *zero-imp-le-mult*: $0 \in elts\ y \Longrightarrow x \le x{*}y$
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 $y \sqcup z$] *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 $\bigsqcup Y$]
by (*simp add*: *cSUP-UNION*) (*metis mult*)

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

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

lemma *mult-Limit*: *Limit* $\gamma \Longrightarrow x * \gamma = \bigsqcup ((*)\ x\ `\ \mathit{elts}\ \gamma)$
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-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 $(x * y) * z = (\bigsqcup u{\in}elts\ z.\ lift\ (x * y * u)\ (x * y))$
using *mult* **by** *blast*
also have $\ldots = (\bigsqcup u{\in}elts\ z.\ lift\ (x * (y * u))\ (x * y))$
using *step.IH* **by** *auto*
also have $\ldots = (\bigsqcup u{\in}elts\ z.\ x * lift\ (y * u)\ y)$
using *mult-lift* **by** *auto*
also have $\ldots = x * (\bigsqcup u{\in}elts\ z.\ lift\ (y * u)\ y)$
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 $Ord\ \beta\ \beta \neq 0$ **shows** $\alpha \leq \alpha * \beta$
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 $\alpha \in (*)\ \alpha$ ‘ $elts\ \mu$
using *Limit.hyps Limit-def one-V-def* **by** (*metis imageI mult.right-neutral*)
then have $\alpha \leq \bigsqcup\ ((*)\ \alpha$ ‘ $elts\ \mu)$
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*]: $[\![Ord\ y;\ Ord\ x]\!] \Longrightarrow Ord\ (x{*}y)$
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*) = (⨆*y*∈*elts* (⨆*u*∈*elts y*. *lift* (*x* ∗ *u*) *x*). *succ* (*rank y*))
by (*metis rank-Sup mult*)
also have . . . = (⨆*u*∈*elts y*. ⨆*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 . . . = (⨆*u*∈*elts y*. ⨆*r*∈*elts x*. *succ* (*rank* (*x* ∗ *u*) + *rank r*))
using *rank-add-distrib* **by** *auto*
also have . . . = (⨆*u*∈*elts y*. ⨆*r*∈*elts x*. *succ* (*rank x* ∗ *rank u* + *rank r*))
using *step arg-cong* [**where** *f* = *Sup*] **by** *auto*
also have . . . = (⨆*u*∈*elts y*. *rank x* ∗ *rank u* + *rank x*)
proof (*rule SUP-cong*)
show (⨆*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 (⨆*r*∈*elts x*. *succ* (*rank x* ∗ *rank u* + *rank r*)) = *rank x* ∗ *rank u* +
(⨆*y*∈*elts x*. *succ* (*rank y*))
proof (*rule order-antisym*)
show (⨆*r*∈*elts x*. *succ* (*rank x* ∗ *rank u* + *rank r*)) ≤ *rank x* ∗ *rank u* +
(⨆*y*∈*elts x*. *succ* (*rank y*))
by (*auto simp*: *Sup-le-iff simp flip*: *plus-V-succ-right*)
have *rank x* ∗ *rank u* + (⨆*y*∈*elts x*. *succ* (*rank y*)) = (⨆*y*∈*elts x*. *rank x* ∗
rank u + *succ* (*rank y*))
by (*simp add*: *add-Sup-distrib False*)
also have . . . ≤ (⨆*r*∈*elts x*. *succ* (*rank x* ∗ *rank u* + *rank r*))
using *plus-V-succ-right* **by** *auto*
finally show *rank x* ∗ *rank u* + (⨆*y*∈*elts x*. *succ* (*rank y*)) ≤ (⨆*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* ∈ *elts x*
    by *fastforce*
  from ‹*y* ≠ *0*› **show** *?thesis*
  proof (*induction y rule*: *eps-induct*)
    case (*step y*)
    show *?case*
    proof (*cases y* = *1*)
      case *False*
      with ‹*y* ≠ *0*› **obtain** *p* **where** *p*: *p* ∈ *elts y p* ≠ *0*
        by (*metis V-equalityI elts-1 insertI1 singletonD trad-foundation*)
      then have *x*∗*p* + *r* ∈ *elts* (*lift* (*x*∗*p*) *x*)
        by (*simp add*: *lift-def r*)
      moreover have *lift* (*x*∗*p*) *x* ≤ *x*∗*y*
        by (*metis bdd-above-iff-small cSUP-upper2 order-refl* ‹*p* ∈ *elts y*› *replacement small-elts mult*)
      ultimately have *x*∗*p* + *r* ∈ *elts* (*x*∗*y*)
        by *blast*
      moreover have *x*∗*p* ⊑ *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* ⟷ *x*=*0* ∨ *y*=*0*
proof
  show *x* = *0* ∨ *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-of-nat* $n)$ $\bigwedge m.\ m < n \Longrightarrow x \notin$ *elts* $(y *$ *ord-of-nat* $m)$
proof −
  obtain *k* **where** *k*: $k \neq 0 \wedge x \in$ *elts* $(y *$ *ord-of-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-of-nat* $k))$
  show *thesis*
  proof
    show $n \neq 0$ $x \in$ *elts* $(y *$ *ord-of-nat* $n)$
      unfolding *n-def* **by** (*metis* (*mono-tags*, *lifting*) *LeastI-ex k*)+
    show $\bigwedge m.\ m < n \Longrightarrow x \notin$ *elts* $(y *$ *ord-of-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* $a=0 \vee x=0$)
  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 Φ ≡ λx y. ∃r. 0 ⊏ r ∧ r ⊏ a ∧ a∗x = a∗y + r
      show ?case
      proof (rule ccontr)
        assume b ≠ 0
        then have 0 ⊏ b
          by (metis nonzero-less-TC)
        then have Φ x y
          unfolding Φ-def using succ.prems by blast
       then obtain x′ where Φ x′ y x′ ⊑ x and min: ⋀x′′. x′′ ⊏ x′ =⇒ ¬ Φ x′′ y
          using less-TC-minimal [of λx. Φ x y x] by blast
        then obtain b′ where 0 ⊏ b′ b′ ⊏ a and eq: a∗x′ = a∗y + b′
          using Φ-def by blast
        have a∗y ⊏ a∗x′
          using TC-add′ ‹0 ⊏ b′› eq by auto
        then obtain p where p ∈ elts (a ∗ x′) a ∗ y ⊑ p
          using less-TC-iff by blast
        then have p ∉ elts (a ∗ y)
          using less-TC-iff less-irrefl-TC by blast
        then have p ∈ ⋃ (elts ‘ (λv. lift (a ∗ v) a) ‘ elts x′)
          by (metis ‹p ∈ elts (a ∗ x′)› elts-Sup replacement small-elts mult)
        then obtain u c where u ∈ elts x′ c ∈ elts a p = a∗u + c
          using lift-def by auto
        then have p ∈ elts (lift (a∗y) b′)
          using ‹p ∈ elts (a ∗ x′)› ‹p ∉ elts (a ∗ y)› eq plus-eq-lift by auto
        then obtain d where d: d ∈ elts b′ p = a∗y + d p = a∗u + c
            by (metis ‹p = a ∗ u + c› ‹p ∈ elts (a ∗ x′)› ‹p ∉ elts (a ∗ y)› eq
mem-plus-V-E)
        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 ... $\leq$ $a{*}x'$
unfolding *mult* [*of* - x'] **using** ‹$u \in$ *elts* x'› **by** (*auto intro*: *cSUP-upper*)
also have ... $=$ $a{*}y \sqcup$ *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 Φ y u
unfolding Φ-*def* **using** *nonzero-less-TC* **by** *blast*
then obtain y' **where** Φ 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* $(a{*}v)\ a \leq a{*}y'$
unfolding *mult* [*of* - y'] **using** ‹$v \in elts\ y'$› **by** (*auto intro*: *cSUP-upper*)
also have $\ldots = a{*}u \sqcup lift\ (a{*}u)\ b'$
by (*simp add*: *eq plus-eq-lift*)
finally have *lift* $(a{*}v)\ a \leq a{*}u \sqcup lift\ (a{*}u)\ b'$ **.**
then have *lift* $(a{*}u)\ a \leq lift\ (a{*}u)\ b'$
by (*metis* ‹$a * u = a * v$› *le-iff-sup lift-sup-distrib sup-left-commute sup-lift-eq-lift*)
then have $a \leq b'$
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 $a{*}u \trianglelefteq a{*}v \mid a{*}v \trianglelefteq a{*}u$
using *d comparable vle-comparable-def* **by** *auto*
then show *False*
proof *cases*
case *1*
then obtain e **where** e: $a{*}v = a{*}u + e\ e \neq 0$
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 ‹$d \in elts\ b'$› ‹$b' \sqsubset a$› **have** $e \sqsubset a$
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: $a{*}u = a{*}v + e\ e \neq 0$
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 $a \neq 0$
shows $a{*}x = a{*}y \longleftrightarrow x{=}y$
by (*metis assms nonzero-less-TC mult-cancellation-lemma*)

corollary *mult-cancellation-less*:
assumes *lt*: $a{*}x + r < a{*}y + s$ **and** $r \sqsubset a$ $s \sqsubset a$
obtains $x < y$ | $x = y$ $r < s$
proof −
have $x \leq y$
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* ∈ *elts* (*a*∗*y*) ⟷ *x* ∈ *elts y* ∧ *0* ∈ *elts a*
proof −
have *x* ∈ *elts y*
if *a* ∗ *x* ∈ *elts* (*a* ∗ *y*) ∧ *0* ∈ *elts a*
using *that* **using** *mult-add-mem* [*of a x 0*]
using *nonzero-less-TC* **by** *force*
moreover have *a* ∗ *x* ∈ *elts* (*a* ∗ *y*)
if *x* ∈ *elts y 0* ∈ *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* ∈ *elts* (*x*∗*y*) ⟷ *0* ∈ *elts x* ∧ *0* ∈ *elts y*
by (*metis Kirby.mult-zero-right mult-add-mem-0*)

lemma *zero-less-mult-iff* [*simp*]: *0* < *x*∗*y* ⟷ *0* < *x* ∧ *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* α; *Ord* β; *Ord* γ⟧ ⟹ α∗β < α∗γ ⟷ β < γ ∧ *0* < α
using *mult-add-mem-0* [*of* α β γ]
by (*meson Ord-0 Ord-mem-iff-lt Ord-mult*)

lemma *mult-cancel-le-iff* [*simp*]:
⟦*Ord* α; *Ord* β; *Ord* γ⟧ ⟹ α∗β ≤ α∗γ ⟷ β ≤ γ ∨ α=*0*
by (*metis Ord-linear2 Ord-mult eq-iff leD mult-cancel-less-iff mult-cancellation*)

lemma *mult-Suc-add-less*: ⟦α < γ; β < γ; *Ord* α; *Ord* β; *Ord* γ⟧ ⟹ γ ∗ *ord-of-nat m* + α < γ ∗ *ord-of-nat* (*Suc m*) + β
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* α < γ β < γ **and** *ord*: *Ord* α *Ord* β *Ord* γ
shows γ ∗ *ord-of-nat m* + α < γ ∗ *ord-of-nat n* + β
proof −
have *Suc m* ≤ *n*
using ‹*m* < *n*› **by** *auto*
have γ ∗ *ord-of-nat m* + α < γ ∗ *ord-of-nat* (*Suc m*) + β
using *assms mult-Suc-add-less* **by** *blast*
also have ... ≤ γ ∗ *ord-of-nat n* + β
using *Ord-mult Ord-ord-of-nat add-right-mono* ‹*Suc m* ≤ *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* β $y \in$ *elts* β $\mu \in$ *elts* α $\nu \in$ *elts* α *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* δ $y = x + \delta$
    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* α› ‹*Ord* α› *less-le-trans*
zero-imp-le-mult)
qed

lemma *add-mult-less*:
  assumes $\gamma \in$ *elts* α $\nu \in$ *elts* β *Ord* α *Ord* β
    shows $\alpha * \nu + \gamma \in$ *elts* $(\alpha * \beta)$
proof −
  have *Ord* ν
    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* γ $\beta' \in$ *elts* γ *Ord* α *Ord* α' *Ord* γ
  shows $\gamma * \alpha + \beta \in$ *elts* $(\gamma * \alpha' + \beta') \longleftrightarrow \alpha \in$ *elts* $\alpha' \vee \alpha = \alpha' \wedge \beta \in$ *elts* β'
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
  assume *L*: *?lhs*
  show *?rhs*
  proof (*cases* $\alpha \in$ *elts* α')
    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* α'
    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* α'› *add-le-cancel-left0 add-mult-less vsubsetD*)
next
assume $\alpha = \alpha' \wedge \beta \in$ *elts* β'
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 $x * y = (\bigsqcup u \in$ *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*: $TC(x * y) = (\bigsqcup r \in$ *elts* $(TC\ x)$. $\bigsqcup u \in$ *elts* $(TC\ y)$. *set*$\{x * u + r\}$)
proof (*cases* $x = 0$)
case *False*
have $*$: $TC(x * y) = (\bigsqcup u \in$ *elts* $(TC\ y)$. *lift* $(x * u)$ $(TC\ x)$) **for** y
proof (*induction* y *rule*: *eps-induct*)
case (*step* y)
have $TC(x * y) = (\bigsqcup u \in$ *elts* y. TC (*lift* $(x * u)$ x))
by (*simp add*: *mult* [*of* x y] *TC-Sup-distrib image-image*)
also have $\ldots = (\bigsqcup u \in$ *elts* y. $TC(x * u) \sqcup$ *lift* $(x * u)$ $(TC\ x)$)
by (*simp add*: *TC-lift False*)
also have $\ldots = (\bigsqcup u \in$ *elts* y. $(\bigsqcup z \in$ *elts* $(TC\ u)$. *lift* $(x * z)$ $(TC\ x)) \sqcup$ *lift* $(x * u)$ $(TC\ x)$)
by (*simp add*: *step*)
also have $\ldots = (\bigsqcup u \in$ *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*) ⊗ *vcard* (*TC y*)
proof −
have (⋃ *u*∈*elts* (*TC x*). ⋃ *v*∈*elts* (*TC y*). {*x* ∗ *v* + *u*}) = (⋃ *u*∈*elts* (*TC x*). (λ*v*. *x* ∗ *v* + *u*) ‘ *elts* (*TC y*))
by (*simp add*: *UNION-singleton-eq-range*)
also have ... ≈ (⋃ *x*∈*elts* (*TC x*). *elts* (*lift* (*TC y* ∗ *x*) (*TC y*)))
proof (*rule UN-eqpoll-UN*)
show (λ*v*. *x* ∗ *v* + *u*) ‘ *elts* (*TC y*) ≈ *elts* (*lift* (*TC y* ∗ *u*) (*TC y*))
if *u* ∈ *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* $(A*B) \leq \aleph 0$
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*: $(u + x, u + y) \in$ *VWF* **if** $x \in$ *elts* α $y \in$ *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* α $y \in$ *elts* α $(u + x, u + y) \in$ *VWF* **for** *x y*
using *that* **by** (*metis Ord-in-Ord Ord-linear-lt VWF-asym assms*)
show $\exists f$. *bij-betw f* ((+) *u* ‘ *elts* α) (*elts* α) $\wedge$ ($\forall x \in$(+) *u* ‘ *elts* α. $\forall y \in$(+) *u* ‘ *elts* α. $(f\,x < f\,y) = ((x, y) \in$ *VWF*))
using *1 2* **unfolding** *bij-betw-def inj-on-def*
by (*rule-tac x*=λx. *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* α *Ord* α
shows *ordertype* (*elts* α − *elts* β) *VWF* = δ
proof −
have $*$: *elts* α − *elts* β = ((+)β) ‘ *elts* δ
proof
show *elts* α − *elts* $\beta \subseteq$ (+) β ‘ *elts* δ
by *clarsimp* (*metis assms*(*1*) *image-iff mem-plus-V-E*)
show (+) β ‘ *elts* $\delta \subseteq$ *elts* α − *elts* β
using *assms*(*1*) *not-add-mem-right* **by** *force*
qed
have *ordertype* ((+) β ‘ *elts* δ) *VWF* = δ
proof (*subst ordertype-VWF-inc-eq*)
show *elts* $\delta \subseteq$ *ON ordertype* (*elts* δ) *VWF* = δ
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* α **and** β: *Ord* β
shows *ordertype* ($\{\alpha\ ..< \alpha+\beta\} \cap$ *ON*) *VWF* = β

proof −
  have *ON*: (+) α ‘ *elts* $\beta \subseteq$ *ON*
    using *assms Ord-add Ord-in-Ord* **by** *blast*
  have ($\{\alpha$ ..< $\alpha+\beta\} \cap$ *ON*) = (+) α ‘ *elts* β
    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* β) *VWF* = *ordertype* ((+) α ‘ *elts* β) *VWF*
    using *ON* β *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*) ⊆ *f* ‘ (*A* × *B*)
proof (*clarsimp simp*: *f-def image-iff elim* !: *elts-multE split*: *prod.split*)
fix *u v*
assume *u*: *u* ∈ *elts* (*?β*) **and** *v*: *v* ∈ *elts ?α*
have *inv-into B* (*ordermap B s*) *u* ∈ *B*
by (*simp add*: *inv-into-ordermap u*)
moreover have *inv-into A* (*ordermap A r*) *v* ∈ *A*
by (*simp add*: *inv-into-ordermap v*)
ultimately show ∃ *x*∈*A*. ∃ *y*∈*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 ... $\leq$ *?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 $u = x \wedge (v, y) \in s$
then show $f\ p < f\ q$
using § *assms* **by** (*fastforce simp*: *f-def split*: *prod.split intro*: *ordermap-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** ‹↑› *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* $\bigsqcup \xi \in$ *elts x*. $f\ \xi$
else f ($\bigsqcup$(*elts x*)) $*$ *a*) *b*

$0 \uparrow \omega = 1$ 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* $\alpha=0$ *then 0 else* $\bigsqcup \xi \in$ *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* $\alpha \Longrightarrow 0\uparrow\alpha = (\mathit{if}\ \alpha = 0\ \mathit{then}\ 1\ \mathit{else}\ 0)$
  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** $\alpha\uparrow\beta > 0 \longleftrightarrow \alpha>0 \vee \beta=0$
  by (*simp add*: *assms less-V-def*)

lemma *ord-of-nat-oexp*: *ord-of-nat* $(m\hat{\ }n)$ = *ord-of-nat* $m\uparrow$*ord-of-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* ω $\beta \in$ *elts* ω **shows** $\alpha\uparrow\beta \in$ *elts* ω
proof −
  obtain m n **where** α = *ord-of-nat* m β = *ord-of-nat* n
    using *assms elts-*ω **by** *auto*
  then have $\alpha\uparrow\beta$ = *ord-of-nat* $(m\hat{\ }n)$
    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* $<$ α *Ord* β **shows** $\beta \le \alpha\uparrow\beta$
proof (*cases* β = *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 $\alpha\uparrow\gamma * 1 < \alpha\uparrow\gamma * \alpha$
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 $\gamma < \alpha\uparrow$*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 $\beta < \gamma$ *Limit* γ *Ord* β $\alpha > 0$ **shows** $\alpha\uparrow\beta \le \alpha\uparrow\gamma$
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: $\beta \in$ *elts* γ **and** *Ord* γ **and** α: $\alpha > 1$ *Ord* α **shows** $\alpha\uparrow\beta < \alpha\uparrow\gamma$
proof −
obtain $\beta < \gamma$ *Ord* β
using *Ord-in-Ord OrdmemD assms* **by** *auto*
have *gt0*: $\alpha\uparrow\beta > 0$
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 $\beta = \delta \mid \beta < \delta$
using *OrdmemD elts-succ* **by** *blast*
then show *?case*
proof *cases*
case *1*
then have $(\alpha\uparrow\beta) * 1 < (\alpha\uparrow\delta) * \alpha$
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 $(\alpha\uparrow\delta) * 1 < (\alpha\uparrow\delta) * \alpha$
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* $\beta < \gamma$
using *Limit-def Ord-in-Ord OrdmemD assms* **by** *auto*
have $\alpha\uparrow\beta = (\alpha\uparrow\beta) * 1$
by *simp*
also have $\ldots < (\alpha\uparrow\beta) * \alpha$
using *Ord-oexp* ‹*Ord* β› *assms gt0 mult-cancel-less-iff* **by** *blast*
also have $\ldots = \alpha\uparrow$*succ* β
by (*simp add*: ‹*Ord* β›)
also have $\ldots \le (\bigsqcup\xi \in$ *elts* $\gamma.\ \alpha\uparrow\xi)$

```
    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 < (⨆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 α ≠ 0 and β: Ord β γ ⊑ β shows α↑γ ≤ α↑β
  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 α↑γ ≤ ⨆ ((↑) α ‘ elts μ)
   using Limit.hyps Ord-less-TC-mem ‹α ≠ 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 γ ≤ β α ≠ 0 Ord α Ord β Ord γ shows α↑γ ≤ α↑β
  by (simp add: assms oexp-mono vle2 vle-iff-le-Ord)

lemma oexp-sup:
  assumes α ≠ 0 Ord α Ord β Ord γ shows α↑(β ⊔ γ) = α↑β ⊔ α↑γ
  by (metis Ord-linear-le assms oexp-mono-le sup.absorb2 sup.orderE)

lemma oexp-Sup:
  assumes α: α ≠ 0 Ord α and X: X ⊆ ON small X X ≠ {} shows α↑⨆ X =
⨆ ((↑) α ‘ X)
proof (rule order-antisym)
  show ⨆ ((↑) α ‘ X) ≤ α↑⨆ 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 $\alpha\uparrow\bigsqcup X \leq \bigsqcup ((\uparrow)\ \alpha\ `\ X)$
  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*: $x \in$ *elts* $(\alpha \uparrow y)$ **and** $z \in X$ $y \in$ *elts* z
      then have $\alpha \uparrow y \leq \alpha \uparrow z$
      by (*meson ON-imp-Ord Ord-in-Ord OrdmemD* α ‹$X \subseteq ON$› *le-less oexp-mono-le*)
      with *x* **have** $x \in$ *elts* $(\alpha \uparrow z)$ **by** *blast*
      then show $\exists u \in X.\ x \in$ *elts* $(\alpha \uparrow u)$
        using ‹$z \in X$› **by** *blast*
    qed
  qed
qed

lemma *omega-le-Limit*:
  assumes *Limit* μ **shows** $\omega \leq \mu$
proof
  fix ϱ
  assume $\varrho \in$ *elts* ω
  then obtain *n* **where** $\varrho =$ *ord-of-nat n*
    using *elts-*ω **by** *auto*
  have *ord-of-nat n* $\in$ *elts* μ
    by (*induction n*) (*use Limit-def assms* **in** *auto*)
  then show $\varrho \in$ *elts* μ
    using ‹$\varrho =$ *ord-of-nat n*› **by** *auto*
qed

lemma *finite-omega-power* [*simp*]:
  assumes $1 < n$ $n \in$ *elts* ω **shows** $n\uparrow\omega = \omega$
proof (*rule order-antisym*)
  have $\bigsqcup ((\uparrow)$ (*ord-of-nat k*) ‘ *elts* $\omega) \leq \omega$ **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 $n\uparrow\omega \le \omega$
using *assms*
by (*simp add*: *elts-ω oexp-Limit*) *metis*
show $\omega \le n\uparrow\omega$
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 $\alpha\uparrow(\beta + (\bigsqcup\xi\in elts\ \mu.\ \xi)) = (\bigsqcup\xi\in elts\ (\beta + \mu).\ \alpha\uparrow\xi)$
by (*simp add*: *Limit.hyps oexp-Limit assms False*)
also have $\ldots = (\bigsqcup\xi \in \{\xi.\ Ord\ \xi \wedge \beta + \xi < \beta + \mu\}.\ \alpha\uparrow(\beta + \xi))$
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 $\exists\gamma.\ Ord\ \gamma \wedge \gamma < \mu \wedge \alpha\uparrow\xi \le \alpha\uparrow(\beta + \gamma)$
proof (*rule mem-plus-V-E* [*OF* ξ])
assume $\xi \in elts\ \beta$
then have $\alpha\uparrow\xi \le \alpha\uparrow\beta$

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 $\exists y \in (\lambda\xi.\ \alpha\uparrow(\beta + \xi))$ ‘ $\{\xi.\ Ord\ \xi \wedge \beta + \xi < \beta + \mu\}.\ x \leq y$
using x **by** *auto*
qed *auto*
also have $\ldots = (\bigsqcup\xi\in elts\ \mu.\ \alpha\uparrow(\beta + \xi))$
using ‹*Limit* μ›
by (*simp add*: *Ord-Collect-lt Limit-def*)
also have $\ldots = (\bigsqcup\xi\in elts\ \mu.\ \alpha\uparrow\beta * \alpha\uparrow\xi)$
using *Limit.IH* **by** *auto*
also have $\ldots = \alpha\uparrow\beta * \alpha\uparrow(\bigsqcup\xi\in elts\ \mu.\ \xi)$
using ‹$\alpha \neq 0$› *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** $\alpha\uparrow(\beta * \gamma) = (\alpha\uparrow\beta)\uparrow\gamma$
proof (*cases* $\alpha = 0 \vee \beta = 0$)
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 α↑(β ∗ (⨆ξ∈elts μ. ξ)) = α↑⨆ ((∗) β ‘ elts μ)
      by (simp add: mult-Sup-distrib)
    also have ... = ⨆ (⋃x∈elts μ. (↑) α ‘ elts (β ∗ x))
      using False Lim oexp-Limit by fastforce
    also have ... = (⨆x∈elts μ. α↑(β ∗ x))
    proof (rule Sup-eq-Sup)
      show (λx. α↑(β ∗ x)) ‘ elts μ ⊆ (⋃x∈elts μ. (↑) α ‘ elts (β ∗ 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 ∃y∈(λx. α↑(β ∗ x)) ‘ elts μ. x ≤ y
        if x ∈ (⋃x∈elts μ. (↑) α ‘ elts (β ∗ 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 ... = ⨆ ((↑) (α↑β) ‘ elts (⨆ξ∈elts μ. ξ))
      using Limit.IH Limit.hyps by auto
    also have ... = (α↑β)↑(⨆ξ∈elts μ. ξ)
      using False Limit.hyps oexp-Limit ‹Ord β› by auto
    finally show ?case .
  qed
qed

lemma Limit-omega-oexp:
  assumes Ord δ δ ≠ 0
  shows Limit (ω↑δ)
  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 $(\alpha \uparrow j) * \alpha = \alpha * (\alpha \uparrow j)$
proof $-$
  have $(\alpha \uparrow j) * \alpha = \alpha \uparrow (1 +$ *ord-of-nat j*$)$
    by (*simp add*: *one-V-def*)
  also have ... $= \alpha * (\alpha \uparrow j)$
    by (*simp add*: *assms oexp-add*)
  finally show *?thesis* **.**
qed

lemma *oexp-ω-Limit*: *Limit* $\beta \Longrightarrow \omega\uparrow\beta = (\bigsqcup \xi \in$ *elts* $\beta.\ \omega\uparrow\xi)$
  by (*simp add*: *oexp-Limit*)

lemma *ω-power-succ-gtr*: *Ord* $\alpha \Longrightarrow \omega \uparrow \alpha *$ *ord-of-nat n* $< \omega \uparrow$ *succ* α
  by (*simp add*: *OrdmemD*)

lemma *countable-oexp*:

assumes ν: $\alpha \in$ *elts* $\omega 1$
shows $\omega \uparrow \alpha \in$ *elts* $\omega 1$
proof –
have *Ord* α
using *Ord-*ω*1 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* ($\omega \uparrow \alpha * \omega$))
by (*simp add*: *succ-in-Limit-iff countable-mult less-*ω*1-imp-countable*)
then show *?case*
using *Ord-mem-iff-lt countable-iff-less-*ω*1 succ.hyps* **by** *auto*
next
case (*Limit* α)
with *Ord-*ω*1* **have** *countable* ($\bigcup\beta\in$*elts* α. *elts* ($\omega \uparrow \beta$)) *Ord* ($\omega \uparrow \bigsqcup$ (*elts* α))
by (*force simp*: *Limit-def intro*: *Ord-trans less-*ω*1-imp-countable*)+
then have $\omega \uparrow \bigsqcup$ (*elts* α) $< \omega 1$
using *Limit.hyps countable-iff-less-*ω*1 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: $\alpha \in$ *elts* β *Ord* β **and** $m > 0$
shows $\omega{\uparrow}\alpha *$ *ord-of-nat* $n < \omega{\uparrow}\beta *$ *ord-of-nat* m
proof –
have †: $\omega{\uparrow}$*succ* $\alpha \le \omega{\uparrow}\beta$
using *Ord-mem-iff-less-TC assms oexp-mono succ-le-TC-iff* **by** *auto*
have $\omega{\uparrow}\alpha *$ *ord-of-nat* $n < \omega{\uparrow}\alpha * \omega$
using *Ord-in-Ord OrdmemD assms* **by** *auto*
also have ... = $\omega{\uparrow}$*succ* α

using *Ord-in-Ord* α **by** *auto*
also have ... $\leq \omega\uparrow\beta$
using † **by** *blast*
also have ... $\leq \omega\uparrow\beta * $ *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* ($\alpha\#\alpha s$) [] = *0*
| *Cantor-sum-Cons*: *Cantor-sum* ($\alpha\#\alpha s$) ($m\#ms$) = ($\omega\uparrow\alpha$) $*$ *ord-of-nat m* + *Cantor-sum* αs *ms*

abbreviation *Cantor-dec* :: *V list* $\Rightarrow$ *bool* **where**
Cantor-dec $\equiv$ *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* ($\alpha\#\beta\#\beta s$) $\longleftrightarrow \beta < \alpha \wedge$ *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* ($\alpha\#\alpha s$) $\subseteq$ *ON list.set ms* $\subseteq \{0<..\}$ *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 α0 α1 αs' where α01: (α#αs) = α0#α1#αs'
    by (metis (no-types, lifting) Cons.prems(3) Suc-length-conv)
  then have Ord α0 Ord α1
    using Cons.prems(1) α01 by auto
  have *: ω↑α0 * ord-of-nat 1 > ω↑α1 * ord-of-nat m1
  proof (rule cnf-1)
    show α1 ∈ elts α0
      using Cons.prems α01 by (simp add: Ord-mem-iff-lt ‹Ord α0› ‹Ord α1›)
  qed (use ‹Ord α0› in auto)
  show ?case
  proof (cases ms)
    case Nil
    then show ?thesis
      using * one-V-def Cons.prems(3) α01 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 αhat ≡ Sup {γ ∈ ON. ω↑γ ≤ α}
    then have Ord αhat
      using Ord-Sup assms by fastforce
    have ⋀ξ. ⟦Ord ξ; ω↑ξ ≤ α⟧ ⟹ ξ ≤ ω↑α
     by (metis Ord-ω OrdmemD le-oexp' order-trans step.hyps one-V-def succ-in-omega
zero-in-omega)
    then have {γ ∈ ON. ω↑γ ≤ α} ⊆ elts (succ (ω↑α))
      using Ord-mem-iff-lt step.hyps by force
    then have sma: small {γ ∈ ON. ω↑γ ≤ α}
      by (meson down)
    have le: ω↑αhat ≤ α
    proof (rule ccontr)
      assume ¬ ω↑αhat ≤ α
      then have †: α ∈ elts (ω↑αhat)
       by (meson Ord-ω Ord-linear2 Ord-mem-iff-lt Ord-oexp ‹Ord αhat› step.hyps)
      obtain γ where Ord γ ω↑γ ≤ α α < γ
        using ‹Ord αhat›
      proof (cases αhat rule: Ord-cases)
        case 0
        with † show thesis
          by (auto simp: False)
      next
        case (succ β)
        have succ β ∈ {γ ∈ ON. ω↑γ ≤ α}
          by (rule succ-in-Sup-Ord) (use succ αhat-def sma in auto)
        then have ω↑succ β ≤ α
          by blast
        with † show thesis
          using ‹¬ ω↑αhat ≤ α› 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 †: $\omega\uparrow\alpha hat * ord\text{-}of\text{-}nat\ M \le \alpha$ **for** *M*
   by (*meson that Ord-ω Ord-linear2 Ord-mult Ord-oexp Ord-ord-of-nat* ‹*Ord αhat*› *step.hyps*)
  have ¬ $\omega\uparrow succ\ \alpha hat \le \alpha$
   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 $\ldots = \omega\uparrow\alpha hat * \omega$
   using ‹*Ord αhat*› *oexp-succ* **by** *blast*
  also have $\ldots = Sup\ (range\ (\lambda m.\ \omega\uparrow\alpha hat * ord\text{-}of\text{-}nat\ m))$
   by (*simp add*: *mult-Limit*) (*auto simp*: *ω-def image-image*)
  also have $\ldots \le \alpha$
   using † **by** *blast*
  finally show *False*
   by *simp*
 qed
 then obtain *M* **where** *M*: $\omega\uparrow\alpha hat * ord\text{-}of\text{-}nat\ M > \alpha$
  by *blast*
 have *bound*: $i \le M$ **if** $\omega\uparrow\alpha hat * ord\text{-}of\text{-}nat\ i \le \alpha$ **for** *i*
 proof −
  have $\omega\uparrow\alpha hat * ord\text{-}of\text{-}nat\ i < \omega\uparrow\alpha hat * ord\text{-}of\text{-}nat\ M$
   using *M dual-order.strict-trans2 that* **by** *blast*
  then show *?thesis*
   using ‹*Ord αhat*› *less-V-def* **by** *auto*
 qed
 define *mhat* **where** $mhat \equiv Greatest\ (\lambda m.\ \omega\uparrow\alpha hat * ord\text{-}of\text{-}nat\ m \le \alpha)$
 have *mhat-ge*: $m \le mhat$ **if** $\omega\uparrow\alpha hat * ord\text{-}of\text{-}nat\ m \le \alpha$ **for** *m*
  unfolding *mhat-def*
  by (*metis* (*mono-tags*, *lifting*) *Greatest-le-nat bound that*)
 have *mhat*: $\omega\uparrow\alpha hat * ord\text{-}of\text{-}nat\ mhat \le \alpha$
  unfolding *mhat-def*
  by (*rule GreatestI-nat* [**where** *k=0* **and** *b=M*]) (*use bound* **in** *auto*)
 then obtain *ξ* **where** $Ord\ \xi\ \xi \le \alpha$ **and** *ξ*: $\alpha = \omega\uparrow\alpha hat * ord\text{-}of\text{-}nat\ mhat + \xi$
  by (*metis Ord-ω Ord-mult Ord-oexp Ord-ord-of-nat* ‹*Ord αhat*› *step.hyps le-Ord-diff*)
 have *False* **if** $\xi = \alpha$
 proof −
  have $\xi \ge \omega\uparrow\alpha hat$
   by (*simp add*: *le that*)
  then obtain *ζ* **where** $Ord\ \zeta\ \zeta \le \xi$ **and** *ζ*: $\xi = \omega\uparrow\alpha hat + \zeta$
   by (*metis Ord-ω Ord-oexp* ‹*Ord αhat*› ‹*Ord ξ*› *le-Ord-diff*)

then have $\alpha = \omega\uparrow\alpha hat * ord\text{-}of\text{-}nat\ mhat + \omega\uparrow\alpha hat + \zeta$
by (*simp add*: ξ *add.assoc*)
then have $\omega\uparrow\alpha hat * ord\text{-}of\text{-}nat\ (Suc\ mhat) \le \alpha$
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 $\xi in\alpha$: $\xi \in elts\ \alpha$
using *Ord-mem-iff-lt* ‹*Ord* ξ› ‹$\xi \le \alpha$› *less-V-def step.hyps* **by** *auto*
show *thesis*
proof (*cases* $\xi = 0$)
case *True*
show *thesis*
proof (*rule step.prems*)
show $\alpha = Cantor\text{-}sum\ [\alpha hat]\ [mhat]$
by (*simp add*: *True* ξ)
qed (*use* ξ *True* ‹$\alpha \neq 0$› ‹*Ord* αhat› **in** *auto*)
next
case *False*
obtain βs *ns* **where** *sub*: *List.set* $\beta s \subseteq ON$ *list.set ns* $\subseteq \{0<..\}$
and *len-eq*: *length* βs = *length ns*
and *dec*: *Cantor-dec* βs
and ξeq: $\xi = Cantor\text{-}sum\ \beta s\ ns$
using *step.IH* [*OF* $\xi in\alpha$] **by** *blast*
then have *length* $\beta s > 0$ *length ns* > 0
using *False Cantor-sum.simps*(*1*) ‹$\xi = Cantor\text{-}sum\ \beta s\ ns$› **by** *auto*
then obtain $\beta 0$ *n0* $\beta s'$ *ns'* **where** $\beta 0$: $\beta s = \beta 0\ \#\ \beta s'$ **and** *Ord* $\beta 0$
and *n0*: $ns = n0\ \#\ ns'$ **and** $n0 > 0$
using *sub* **by** (*auto simp*: *neq-Nil-conv*)
moreover have *False* **if** $\beta 0 > \alpha hat$
proof −
have $\omega\uparrow\beta 0 \le \omega\uparrow\beta 0 * ord\text{-}of\text{-}nat\ n0 + u$ **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 $\omega\uparrow\beta 0 > \alpha$
using *that* ‹*Ord* $\beta 0$›
by (*metis* (*no-types*, *lifting*) *Ord-ω Ord-linear2 Ord-oexp Sup-upper* αhat*-def leD mem-Collect-eq sma step.hyps*)
ultimately have $\xi \ge \omega\uparrow\beta 0$
by (*simp add*: ξeq $\beta 0$ *n0*)
then show *?thesis*
using ‹$\alpha < \omega\uparrow\beta 0$› ‹$\xi \le \alpha$› **by** *auto*
qed
ultimately have $\beta 0 \le \alpha hat$
using *Ord-linear2* ‹*Ord* αhat› **by** *auto*
then consider $\beta 0 < \alpha hat \mid \beta 0 = \alpha hat$
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** $\alpha 0 < \beta 0$
proof −
have *Ordc*: *Ord* (*Cantor-sum* αs *ms*) *Ord* ($\omega\uparrow\beta 0$)
using *Ord-oexp* ‹*Ord* $\beta 0$› *step.hyps step.prems*(*9*) **by** *blast*+
have ∗: *Cantor-sum* αs *ms* < $\omega\uparrow\beta 0$
using *step.prems*(*1*−*5*) ‹*Ord* $\beta 0$› ‹*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*: $\alpha 0 = \beta 0$
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'* ⊆ {*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 ... = $\omega\uparrow\alpha 0 *$ *ord-of-nat* (*Suc m0*)
by (*simp add*: *mult-succ*)
also have ... ≤ $\omega\uparrow\alpha 0 *$ *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 ... ≤ α
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'* ⊆ {*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 ... = $\omega\uparrow\beta 0 *$ *ord-of-nat* (*Suc n0*)
by (*simp add*: *mult-succ*)
also have ... ≤ $\omega\uparrow\beta 0 *$ *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 ... ≤ α
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* $\alpha s'$ = *length* ms' *length* $\beta s'$ = *length* ns'
using αs *ms* βs *ns step.prems* **by** *auto*
show *Cantor-dec* $\alpha s'$ *Cantor-dec* $\beta s'$
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* $\alpha s'$ ms' = *Cantor-sum* $\beta s'$ 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* $\alpha s \subseteq ON$
and βsON: *List.set* $\beta s \subseteq ON$
and *ms*: *list.set ms* $\subseteq \{0<..\}$
and *ns*: *list.set ns* $\subseteq \{0<..\}$
and *mseq*: *length* αs = *length ms*
and *nseq*: *length* βs = *length ns*
and $\alpha sdec$: *Cantor-dec* αs
and $\beta sdec$: *Cantor-dec* βs
shows $\alpha s = \beta s \wedge ms = ns$
using *Cantor-nf-unique-aux Ord-Cantor-sum assms* **by** *auto*

lemma *less-*ω*-power*:
assumes *Ord* $\alpha 1$ *Ord* β
and $\alpha 2$: $\alpha 2 \in$ *elts* $\alpha 1$ **and** β: $\beta < \omega\uparrow\alpha 2$
and $m1 > 0$ $m2 > 0$
shows $\omega\uparrow\alpha 2 *$ *ord-of-nat* $m2 + \beta < \omega\uparrow\alpha 1 *$ *ord-of-nat* $m1 + (\omega\uparrow\alpha 2 *$ *ord-of-nat*

$m2 + \beta)$
(**is** $?lhs < ?rhs$)
proof −
obtain *oo*: *Ord* $(\omega\uparrow\alpha 1)$ *Ord* $(\omega\uparrow\alpha 2)$
using *Ord-in-Ord Ord-oexp assms* **by** *blast*
moreover obtain *ord-of-nat* $m2 \neq 0$
using *assms ord-of-eq-0* **by** *blast*
ultimately have $\beta < \omega\uparrow\alpha 2 *$ *ord-of-nat m2*
by (*meson Ord-ord-of-nat* β *dual-order.strict-trans1 le-mult*)
with *oo assms* **have** $?lhs \neq ?rhs$
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* $(\alpha\#\alpha s) = (\omega\uparrow\alpha) + \omega$*-sum* αs

abbreviation ω*-dec* :: *V list* $\Rightarrow$ *bool* **where**
ω*-dec* $\equiv$ *sorted-wrt* $(\geq)$

lemma *Ord-*ω*-sum* [*simp*]: *List.set* $\alpha s \subseteq ON \Longrightarrow$ *Ord* $(\omega$*-sum* $\alpha s)$
by (*induction* αs) *auto*

lemma ω*-dec-Cons-iff* [*simp*]: ω*-dec* $(\alpha\#\beta\#\beta s) \longleftrightarrow \beta \leq \alpha \wedge \omega$*-dec* $(\beta\#\beta s)$
by *auto*

lemma ω*-sum-0E*:
assumes ω*-sum* $\alpha s = 0$ *List.set* $\alpha s \subseteq ON$
shows $\alpha s = []$
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* (*α*#*αs*) [] = []
| *ω-of-Cantor-Cons*: *ω-of-Cantor* (*α*#*αs*) (*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*) = *ω* ↑ *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*) ⊆ *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* ≤ *α*
using *Cons.prems*(*1*) *order.strict-implies-order* **by** *auto*

```
  moreover have ∀x∈list.set (ω-of-Cantor αs ns). x ≤ a
    using Cons ns ‹a ≤ α›
    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 $\delta \in$ *elts* (*ω-sum αs*)
shows $\exists k<$*length αs*. $\exists \gamma\in$*elts* $(\omega \uparrow (\alpha s!k))$. $\delta = \omega$-*sum* (*take k αs*) + γ
using *assms*
proof (*induction αs arbitrary*: δ)
case (*Cons α αs*)
then have $\delta \in$ *elts* $(\omega \uparrow \alpha + \omega$-*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 αs* $\subseteq$ *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 αs* $\subseteq$ *ON*
by *simp*
with *Cons.IH* **have** $\bigwedge k\ x.\ k <$ *length αs* $\Longrightarrow \omega \uparrow \alpha s \ !\ k \leq \omega \uparrow a + \omega$-*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 αs* $< \omega\uparrow\alpha + \omega$-*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 α1 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* ($\alpha\#\alpha s$) *List.set* ($\alpha\#\alpha s$) $\subseteq$ *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* $\alpha s < \omega\uparrow\alpha * \omega$
using *Cons* **by** *force*
then have $\omega\uparrow\beta$ + *ω-sum* $\alpha s < \omega\uparrow\alpha + \omega\uparrow\alpha * \omega$
using *Cons.prems add-right-strict-mono oexp-mono-le* **by** *auto*
also have $\ldots = \omega\uparrow\alpha * \omega$
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* $\alpha s \subseteq$ *ON*
and *βsON*: *List.set* $\beta s \subseteq$ *ON*
and *αsdec*: *ω-dec* αs
and *βsdec*: *ω-dec* βs
and *αseq*: α = *ω-sum* αs
and *βseq*: α = *ω-sum* βs
shows $\alpha s = \beta s$
using *assms*
proof (*induction* α *arbitrary*: αs βs *rule*: *Ord-induct*)
case (*step* α)
show *?case*
proof (*cases* $\alpha = 0$)
case *True*
then show *?thesis*
using *step.prems* **by** (*metis ω-sum-0E*)
next
case *False*
then obtain $\alpha 0$ $\alpha s'$ $\beta 0$ $\beta s'$ **where** *αs*: $\alpha s = \alpha 0 \ \# \ \alpha s'$ **and** *βs*: $\beta s = \beta 0 \ \# \ \beta s'$

by (*metis* ω*-sum.elims step.prems*(*5*,*6*))
then have *ON*: *Ord* α*0 list.set* $\alpha s' \subseteq$ *ON Ord* β*0 list.set* $\beta s' \subseteq$ *ON*
using αs βs *step.prems*(*1*,*2*) **by** *auto*
have *False* **if** $\beta 0 < \alpha 0$
proof −
have *Ordc*: *Ord* (ω*-sum* βs) *Ord* ($\omega\uparrow\alpha 0$)
using *Ord-oexp* ‹*Ord* $\alpha 0$› *step.hyps step.prems*(*6*) **by** *blast+*
have ω*-sum* $\beta s < \omega\uparrow\beta 0 * \omega$
by (*rule* ω*-sum-less-*ω*-power*) (*use* βs *step.prems ON* **in** *auto*)
also have ... $\leq \omega\uparrow\alpha 0$
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** $\alpha 0 < \beta 0$
proof −
have *Ordc*: *Ord* (ω*-sum* αs) *Ord* ($\omega\uparrow\beta 0$)
using *Ord-oexp* ‹*Ord* $\beta 0$› *step.hyps step.prems*(*5*) **by** *blast+*
have ω*-sum* $\alpha s < \omega\uparrow\alpha 0 * \omega$
by (*rule* ω*-sum-less-*ω*-power*) (*use* αs *step.prems ON* **in** *auto*)
also have ... $\leq \omega\uparrow\beta 0$
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*: ‹$\alpha = \omega$*-sum* αs› *leD*)
qed
ultimately have †: $\alpha 0 = \beta 0$
using *Ord-linear-lt* ‹*Ord* $\alpha 0$› ‹*Ord* $\beta 0$› **by** *blast*
moreover have $\alpha s' = \beta s'$
proof (*rule step.IH*)
show ω*-sum* $\alpha s' \in$ *elts* α
using *step.prems* αs
by (*simp add*: *Ord-mem-iff-lt* ω*-sum-less-self*)
show ω*-dec* $\alpha s'$ ω*-dec* $\beta s'$
using αs βs *step.prems*(*3*,*4*) **by** *auto*
have ω*-sum* $\alpha s = \omega$*-sum* βs
using *step.prems*(*5*,*6*) **by** *auto*
then show ω*-sum* $\alpha s' = \omega$*-sum* $\beta s'$
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* $\alpha \equiv$ *Ord* $\alpha \wedge (\forall \beta \in$ *elts* $\alpha.\ \forall \gamma \in$ *elts* $\alpha.\ \beta+\gamma \in$ *elts* $\alpha)$

lemma *indecomposableD*:
$[\![$*indecomposable* α; $\beta < \alpha$; $\gamma < \alpha$; *Ord* β; *Ord* $\gamma]\!] \Longrightarrow \beta+\gamma < \alpha$
by (*meson Ord-mem-iff-lt OrdmemD indecomposable-def*)

lemma *indecomposable-imp-Ord*:
indecomposable $\alpha \Longrightarrow$ *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 $\neg\ \beta+\alpha < \alpha$
by (*simp add*: *add-le-left ord leD*)
moreover have $\neg\ \alpha < \beta+\alpha$
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 α > 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 ⋀β::V. β ∈ elts α ⟹ β+α = α
  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 $Ord\ (\omega\uparrow\delta)$
by (*simp add*: ‹$Ord\ \delta$›)
then obtain $x\ y$ **where** x: $x \in elts\ \delta\ \ Ord\ x\ \ \beta \in elts\ (\omega\uparrow x)$
and y: $y \in elts\ \delta\ \ Ord\ y\ \ \gamma \in elts\ (\omega\uparrow y)$
using ‹$Ord\ \delta$› *limit ord* $\beta\ \gamma$ *oexp-Limit*
by (*auto simp flip*: *Ord-mem-iff-lt intro*: *Ord-in-Ord*)
then have $succ\ (x \sqcup y) \in elts\ \delta$
by (*metis Limit-def Ord-linear-le limit sup.absorb2 sup.orderE*)
moreover have $\beta + \gamma \in elts\ (\omega\uparrow succ\ (x \sqcup y))$
proof −
have *oxy*: $Ord\ (x \sqcup y)$
using *Ord-sup* $x\ y$ **by** *blast*
then obtain $\omega\uparrow x \le \omega\uparrow(x \sqcup y)\ \ \omega\uparrow y \le \omega\uparrow(x \sqcup y)$
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 $\beta < \omega\uparrow\delta\ \ Ord\ \beta\ \ Ord\ \delta\ \ \delta \ne 0$
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: $Ord\ \alpha$
and *minor*: $\bigwedge X.\ X \subseteq elts\ \alpha \Longrightarrow ordertype\ X\ VWF = \alpha \lor ordertype\ (elts\ \alpha - X)\ VWF = \alpha$

```
  shows indecomposable α
  unfolding indecomposable-def
proof (intro conjI ballI)
  fix β γ
  assume β: β ∈ elts α and γ: γ ∈ elts α
  then obtain βγ: elts β ⊆ elts α elts γ ⊆ elts α Ord β Ord γ
    using α Ord-in-Ord Ord-trans by blast
 then have oeq: ordertype (elts β) VWF = β
    by auto
  show β + γ ∈ elts α
  proof (rule ccontr)
    assume β + γ ∉ elts α
    then obtain δ where δ: Ord δ β + δ = α
      by (metis Ord-ordertype βγ(1) le-Ord-diff less-eq-V-def minor oeq)
    then have δ ∈ elts α
      using Ord-linear βγ γ ‹β + γ ∉ elts α› 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: $\gamma = \omega\uparrow\delta$ **and** $\delta \neq 0$ *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*: $\exists\, b.$ *ordertype* (X b $\cap$ *elts* ($\omega\uparrow\alpha$)) *VWF* $= \omega\uparrow\alpha$
if $\alpha \in$ *elts* δ **for** α
proof (*rule Limit.IH*)
have *Ord* α
using *Ord-in-Ord* ‹*Ord* δ› *that* **by** *blast*
then show $\omega\uparrow\alpha \in$ *elts* γ
by (*simp add*: *Ord-mem-iff-lt* δ ω*-gt1* ‹*Ord* δ› *oexp-less that*)
show *indecomposable* ($\omega\uparrow\alpha$)
using ‹*Ord* α› *indecomposable-1 indecomposable-*ω*-power* **by** *fastforce*
show *small* (X b $\cap$ *elts* ($\omega\uparrow\alpha$)) **for** b
by (*meson down inf-le2*)
show *ordertype* (X b $\cap$ *elts* ($\omega \uparrow \alpha$)) *VWF* $\leq \omega \uparrow \alpha$ **for** b
by (*simp add*: ‹*Ord* α› *ordertype-le-Ord*)
show X b $\cap$ *elts* ($\omega \uparrow \alpha$) $\subseteq$ *ON* **for** b
by (*simp add*: *Limit.prems inf.coboundedI1*)
show *elts* ($\omega \uparrow \alpha$) $\subseteq$ ($\bigcup b.$ X b $\cap$ *elts* ($\omega \uparrow \alpha$))
using *Limit.prems Limit.hyps* ‹$\omega \uparrow \alpha \in$ *elts* γ›
by *clarsimp* (*metis Ord-trans UN-E indecomposable-imp-Ord subset-eq*)
qed
define A **where** $A \equiv \lambda b.$ $\{\alpha \in$ *elts* $\delta.$ *ordertype* (X b $\cap$ *elts* ($\omega\uparrow\alpha$)) *VWF* $\geq \omega\uparrow\alpha\}$
have *Asmall*: *small* (A b) **for** b
by (*simp add*: *A-def*)
have *AON*: A b $\subseteq$ *ON* **for** b
using *A-def* ‹*Ord* δ› *elts-subset-ON* **by** *blast*
have *eq*: *elts* $\delta = (\bigcup\ b.\ A\ b)$
by (*auto simp*: *A-def*) (*metis ot eq-refl*)
then obtain b **where** b: *Sup* (A b) $= \delta$
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 $\omega\uparrow\alpha \leq$ *ordertype* (X b) *VWF* **if** $\alpha \in A$ b **for** α
proof $-$
have ($\omega\uparrow\alpha$) $=$ *ordertype* ((X b) $\cap$ *elts* ($\omega\uparrow\alpha$)) *VWF*
using ‹*Ord* δ› *that* **by** (*simp add*: *A-def Ord-in-Ord dual-order.antisym ordertype-le-Ord*)
also have $\ldots \leq$ *ordertype* (X b) *VWF*
by (*simp add*: *Limit.prems ordertype-VWF-mono*)

finally show *?thesis* **.**
qed
then have *ordertype* $(X\ b)$ *VWF* $\geq$ *Sup* $((\lambda\alpha.\ \omega\uparrow\alpha)$ ‘ $A\ b)$
by *blast*
moreover have *Sup* $((\lambda\alpha.\ \omega\uparrow\alpha)$ ‘ $A\ b) = \omega \uparrow$ *Sup* $(A\ b)$
by (*metis b Ord-ω ZFC-in-HOL.Sup-empty AON* ‹$\delta \neq 0$› *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: $\delta =$ *succ* β *Ord* β
using *Ord-cases* ‹$\delta \neq 0$› ‹*Ord* δ› **by** *auto*
then have *Ordωβ*: *Ord* $(\omega\uparrow\beta)$
using *Ord-oexp* **by** *blast*
have *subX12*: *elts* $(\omega\uparrow\beta * \omega) \subseteq (\bigcup b.\ X\ b)$
using *Limit* β δ **by** *auto*
define E **where** $E \equiv \lambda n.\ \{\omega\uparrow\beta *$ *ord-of-nat* n ..< $\omega\uparrow\beta *$ *ord-of-nat* $(Suc\ n)\}$ $\cap$ *ON*
have *EON*: $E\ n \subseteq$ *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 *succ* $(i) \leq$ *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* - ‹$y \in E\ j$› ‹$x \in E\ i$›] *leI less-asym* **by** *blast*
have *inc*: *indecomposable* $(\omega\uparrow\beta)$
using β *indecomposable-1 indecomposable-ω-power* **by** *fastforce*
have *in-En*: $\omega\uparrow\beta *$ *ord-of-nat* $n + x \in E\ n$ **if** $x \in$ *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 $*$: *elts* $\gamma = \bigcup$(*range* E)
proof
have $\exists\, m.\ \omega\uparrow\beta * m \leq x \land x < \omega\uparrow\beta *$ *succ* (*ord-of-nat* m)
if $x \in$ *elts* $(\omega\uparrow\beta *$ *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** $x \in$ *elts* ($\omega\uparrow\beta$ $*$ *ord-of-nat n*) **for** *x n*
by (*meson Ordωβ Ord-in-Ord Ord-mult Ord-ord-of-nat that*)
ultimately show *elts* $\gamma \subseteq \bigcup$(*range E*)
by (*auto simp*: δ β *E-def mult-Limit elts-ω*)
have $x \in$ *elts* ($\omega\uparrow\beta$ $*$ *succ*(*ord-of-nat n*))
if *Ord x* $x < \omega\uparrow\beta$ $*$ *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 $\bigcup$(*range E*) $\subseteq$ *elts* γ
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* $\equiv \lambda n\ x.$ *odiff x* ($\omega\uparrow\beta$ $*$ *ord-of-nat n*)
have *cutON*: *cut n* ‘ *X* $\subseteq$ *ON* **if** *X* $\subseteq$ *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* ($\omega \uparrow \beta$ $*$ *ord-of-nat n* + *x*) = *x* **for** *x n*
by (*auto simp*: *cut-def*)
have *cuteq*: $x \in$ *cut n* ‘ (*X* $\cap$ *E n*) $\longleftrightarrow$ $\omega\uparrow\beta$ $*$ *ord-of-nat n* + $x \in X$
if *x*: $x \in$ *elts* ($\omega\uparrow\beta$) **for** *x X n*
proof
show $\omega\uparrow\beta$ $*$ *ord-of-nat n* + $x \in X$ **if** $x \in$ *cut n* ‘ (*X* $\cap$ *E n*)
using *E-def Ordωβ Ord-odiff-eq image-iff local.cut-def that* **by** *auto*
show $x \in$ *cut n* ‘ (*X* $\cap$ *E n*)
if $\omega\uparrow\beta$ $*$ *ord-of-nat n* + $x \in X$
by (*metis* (*full-types*) *IntI cut image-iff in-En that x*)
qed
have *ot-cuteq*: *ordertype* (*cut n* ‘ (*X* $\cap$ *E n*)) *VWF* = *ordertype* (*X* $\cap$ *E n*) *VWF* **for** *n X*
proof (*rule ordertype-VWF-inc-eq*)
show *X* $\cap$ *E n* $\subseteq$ *ON*
using *E-def* **by** *blast*
then show *cut n* ‘ (*X* $\cap$ *E n*) $\subseteq$ *ON*
by (*simp add*: *cutON*)
show *small* (*X* $\cap$ *E n*)
by (*meson Int-lower2 smE smaller-than-small*)
show *cut n x* < *cut n y*
if $x \in X \cap E\ n$ $y \in X \cap E\ n$ $x < y$ **for** *x y*
using *that* ‹*X* $\cap$ *E n* $\subseteq$ *ON*› **by**(*simp add*: *E-def Ordωβ Ord-odiff-less-odiff local.cut-def*)
qed

define *N* **where** *N* $\equiv \lambda b.$ {*n. ordertype* (*X b* $\cap$ *E n*) *VWF* = $\omega\uparrow\beta$}
have $\exists$ *b. infinite* (*N b*)
proof (*rule ccontr*)

assume $\nexists\, b.$ *infinite* $(N\ b)$
then obtain n **where** $\bigwedge b.\ n \notin N\ b$
apply (*simp add*: *ex-bool-eq*)
by (*metis* (*full-types*) *finite-nat-set-iff-bounded not-less-iff-gr-or-eq*)
moreover
have $\exists\, b.$ *ordertype* (*cut* n ‘ $(X\ b \cap E\ n)$) $VWF = \omega\uparrow\beta$
proof (*rule Limit.IH*)
show $\omega\uparrow\beta \in$ *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* $(\omega\uparrow\beta)$
by (*simp add*: *inc*)
show *ordertype* (*cut* n ‘ $(X\ b \cap E\ n)$) $VWF \leq \omega\uparrow\beta$ **for** b
by (*metis otE inf-le2 ordertype-VWF-mono ot-cuteq smE*)
show *small* (*cut* n ‘ $(X\ b \cap E\ n)$) **for** b
using *smE subset-iff-less-eq-V*
by (*meson inf-le2 replacement*)
show *cut* n ‘ $(X\ b \cap E\ n) \subseteq ON$ **for** b
using *E-def cutON* **by** *auto*
have *elts* $(\omega\uparrow\beta *$ *succ* $n) \subseteq \bigcup($*range* $X)$
by (*metis Ord*$\omega\beta$ *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* $(\omega\uparrow\beta) \subseteq (\bigcup b.$ *cut* n ‘ $(X\ b \cap E\ n))$
by (*auto simp*: *mult-succ mult-Limit UN-subset-iff cuteq UN-bool-eq*)
qed
then have $\exists\, b.$ *ordertype* $(X\ b \cap E\ n)$ $VWF = \omega\uparrow\beta$
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* $\Rightarrow$ *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* $(X\ b \cap E\ (\varphi\ n))$ $VWF = \omega\uparrow\beta$ **for** n
using *N-def bij-betw-imp-surj-on* **by** *blast*
moreover have *small* $(X\ b \cap E\ (\varphi\ n))$ **for** n
by (*meson inf-le2 smE subset-iff-less-eq-V*)
ultimately have $\exists f.$ *bij-betw* f $(X\ b \cap E\ (\varphi\ n))$ (*elts* $(\omega\uparrow\beta)) \wedge (\forall x \in X\ b \cap E\ (\varphi\ n).\ \forall y \in X\ b \cap E\ (\varphi\ n).\ f\,x < f\,y \longleftrightarrow (x,y) \in VWF)$
for n **by** (*metis Ord-ordertype ordertype-VWF-eq-iff*)
then obtain F **where** *bijF*: $\bigwedge n.$ *bij-betw* $(F\ n)$ $(X\ b \cap E\ (\varphi\ n))$ (*elts* $(\omega\uparrow\beta)$)
and F: $\bigwedge n.\ \forall x \in X\ b \cap E\ (\varphi\ n).\ \forall y \in X\ b \cap E\ (\varphi\ n).\ F\ n\ x < F\ n\ y \longleftrightarrow (x,y) \in VWF$
by *metis*
then have *F-bound*: $\bigwedge n.\ \forall x \in X\ b \cap E\ (\varphi\ n).\ F\ n\ x < \omega\uparrow\beta$
by (*metis Ord*-ω *Ord-oexp OrdmemD* $\beta(2)$ *bij-betw-imp-surj-on image-eqI*)
have *F-Ord*: $\bigwedge n.\ \forall x \in X\ b \cap E\ (\varphi\ n).$ *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* $\gamma - X$) $\subseteq$ *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*: $B \subseteq A$ *small A*
shows *ordertype B VWF* = $\alpha \vee$ *ordertype* ($A{-}B$) *VWF* = α
proof (*rule ccontr*)
assume $\neg$ (*ordertype B VWF* = $\alpha \vee$ *ordertype* ($A - B$) *VWF* = α)
moreover have *ordertype* (*ordermap A VWF* ‘ *B*) *VWF* = *ordertype B VWF*
using ‹$B \subseteq A$› **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* ‘ $B \subseteq$ *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* $\geq \alpha$ **and** *small*: *small A small B*
shows *ordertype B VWF* $\geq \alpha \vee$ *ordertype* ($A{-}B$) *VWF* $\geq \alpha$
proof −
obtain A' **where** $A' \subseteq A$ *ordertype* A' *VWF* = α
by (*meson* α ‹*small A*› *indec indecomposable-def le-ordertype-obtains-subset*)
then have *ordertype* ($B \cap A'$) *VWF* = $\alpha \vee$ *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* ($B \cap A'$) *VWF* $\leq$ *ordertype B VWF*
by (*meson Int-lower1 small ordertype-VWF-mono smaller-than-small*)
moreover have *ordertype* ($A'{-}B$) *VWF* $\leq$ *ordertype* ($A{-}B$) *VWF*
by (*meson Diff-mono Diff-subset* ‹$A' \subseteq A$› ‹*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 $\mathcal{A}$: *finite* $\mathcal{A}$ *pairwise disjnt* $\mathcal{A}$ $\bigcup\mathcal{A} = A$ $\mathcal{A} \neq \{\}$ *ordertype* A $VWF = \alpha$ *small* A
shows $\exists\, X \in \mathcal{A}.$ *ordertype* X $VWF = \alpha$
using $\mathcal{A}$
proof (*induction arbitrary*: A)
case (*insert* X $\mathcal{A}$)
show *?case*
proof (*cases* $\mathcal{A} = \{\}$)
case *True*
then show *?thesis*
using *insert.prems* **by** *blast*
next
case *False*
have *smA*: *small* ($\bigcup\mathcal{A}$)
using *insert.prems* **by** *auto*
show *?thesis*
proof (*cases* $\exists\, X \in \mathcal{A}.$ *ordertype* X $VWF = \alpha$)
case *True*
then show *?thesis*
using *insert.prems* **by** *blast*
next
case *False*
have $X = A - \bigcup\mathcal{A}$
using *insert.hyps insert.prems* **by** (*auto simp*: *pairwise-insert disjnt-iff*)
then have *ordertype* X $VWF = \alpha$
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 $\mathcal{A}$: *finite* $\mathcal{A}$ $A \subseteq \bigcup\mathcal{A}$ $\mathcal{A} \neq \{\}$ *ordertype* A $VWF \geq \alpha$ *small* ($\bigcup\mathcal{A}$)
shows $\exists\, X \in \mathcal{A}.$ *ordertype* X $VWF \geq \alpha$
using $\mathcal{A}$
proof (*induction arbitrary*: A)
case (*insert* X $\mathcal{A}$)
show *?case*
proof (*cases* $\mathcal{A} = \{\}$)
case *True*
then have $\alpha \leq$ *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* $\exists X \in \mathcal{A}$. *ordertype* X *VWF* $\geq \alpha$)
case *True*
then show *?thesis*
using *insert.prems* **by** *blast*
next
case *False*
moreover have *small* $(X \cup \bigcup \mathcal{A})$
using *insert.prems* **by** *auto*
moreover have *ordertype* $(\bigcup (insert\ X\ \mathcal{A}))$ *VWF* $\geq \alpha$
using *insert.prems ordertype-VWF-mono* **by** *blast*
ultimately have *ordertype* X *VWF* $\geq \alpha$
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*: $\exists$ *V-of* :: $'a \Rightarrow V$. *inj V-of*

context *countable*
begin

subclass *embeddable*
proof −
have *inj* (*ord-of-nat* $\circ$ *to-nat*) **if** *inj to-nat*
for *to-nat* :: $'a \Rightarrow nat$
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 $f :: 'a \Rightarrow V$
  assumes $\bigwedge x\ y.\ f\ x = f\ y \Longrightarrow x = y$
  shows $OFCLASS('a,\ embeddable\text{-}class)$
proof (*intro-classes*, *rule exI*)
  show *inj f*
    by (*rule injI* [*OF assms*]) *assumption*
qed

instance *V* :: *embeddable*
  by (*intro-classes*) (*meson inj-on-id*)

instance *prod* :: (*embeddable*,*embeddable*) *embeddable*
proof −
  have *inj* ($\lambda(x,y).\ \langle V\text{-}of1\ x,\ V\text{-}of2\ y\rangle$) **if** *inj V-of1 inj V-of2*
    for $V\text{-}of1 :: 'a \Rightarrow V$ **and** $V\text{-}of2 :: 'b \Rightarrow V$
    using *that* **by** (*auto simp*: *inj-on-def*)
  then show $OFCLASS('a \times 'b,\ embeddable\text{-}class)$
    by *intro-classes* (*meson embeddable-class.ex-inj*)
qed

instance *sum* :: (*embeddable*,*embeddable*) *embeddable*
proof −
  have *inj* (*case-sum* ($Inl \circ V\text{-}of1$) ($Inr \circ V\text{-}of2$)) **if** *inj V-of1 inj V-of2*
    for $V\text{-}of1 :: 'a \Rightarrow V$ **and** $V\text{-}of2 :: 'b \Rightarrow V$
    using *that* **by** (*auto simp*: *inj-on-def split*: *sum.split-asm*)
  then show $OFCLASS('a + 'b,\ embeddable\text{-}class)$
    by *intro-classes* (*meson embeddable-class.ex-inj*)
qed

instance *option* :: (*embeddable*) *embeddable*
proof −
  have *inj* (*case-option 0* ($\lambda x.\ ZFC\text{-}in\text{-}HOL.set\{V\text{-}of\ x\}$)) **if** *inj V-of*
    for $V\text{-}of :: 'a \Rightarrow V$
    using *that* **by** (*auto simp*: *inj-on-def split*: *option.split-asm*)
  then show $OFCLASS('a\ option,\ embeddable\text{-}class)$
    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*: $X \lesssim$ (*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 $X \lesssim A$
  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* $(\lambda(x,y).\ \langle V\text{-}of1\ x,\ V\text{-}of2\ y\rangle)$
range $(\lambda(x,y).\ \langle V\text{-}of1\ x,\ V\text{-}of2\ y\rangle) \le elts\ (VSigma\ A\ (\lambda x.\ B))$
if *inj V-of1 inj V-of2 range V-of1* $\le$ *elts A range V-of2* $\le$ *elts B*
for *V-of1* :: $'a \Rightarrow V$ **and** *V-of2* :: $'b \Rightarrow V$ **and** *A B*
using *that* **by** (*auto simp*: *inj-on-def*)
with *small* [**where** $'a='a$] *small* [**where** $'a='b$]
show *OFCLASS*($'a \times 'b$, *small-class*)
by *intro-classes* (*smt* (*verit*) *down-raw f-inv-into-f set-eq-subset small-def*)
qed

instance *sum* :: (*small*,*small*) *small*
proof −
have *inj* (*case-sum* (*Inl* $\circ$ *V-of1*) (*Inr* $\circ$ *V-of2*))
range (*case-sum* (*Inl* $\circ$ *V-of1*) (*Inr* $\circ$ *V-of2*)) $\le$ *elts* ($A \uplus B$)
if *inj V-of1 inj V-of2 range V-of1* $\le$ *elts A range V-of2* $\le$ *elts B*
for *V-of1* :: $'a \Rightarrow V$ **and** *V-of2* :: $'b \Rightarrow V$ **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*)
by *intro-classes* (*metis down-raw replacement set-eq-subset small-def small-iff*)
qed

instance *option* :: (*small*) *small*
proof −
have *inj* ($\lambda x.$ *case x of None* $\Rightarrow$ *0* | *Some x* $\Rightarrow$ *ZFC-in-HOL.set* {*V-of x*})
range ($\lambda x.$ *case x of None* $\Rightarrow$ *0* | *Some x* $\Rightarrow$ *ZFC-in-HOL.set* {*V-of x*}) $\le$ *insert 0* (*elts* (*VPow A*))
if *inj V-of range V-of* $\le$ *elts A*
for *V-of* :: $'a \Rightarrow V$ **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*)
by *intro-classes* (*smt* (*verit*) *down order.refl ex-inj small-iff small-image-iff small-insert*)
qed

instance *list* :: (*small*) *small*
proof −
have *small* (*range* (*V-of-list V-of*))
if *inj V-of range V-of* $\le$ *elts A*
for *V-of* :: $'a \Rightarrow V$ **and** *A*
proof −
have *range* (*V-of-list V-of*) $\approx$ (*UNIV* :: $'a$ *list set*)
using *that* **by** (*simp add*: *inj-V-of-list*)
also have ... $\approx$ *lists* (*UNIV* :: $'a$ *set*)
by *simp*
also have ... $\lesssim$ (*UNIV* :: $'a$ *set*) $\times$ (*UNIV* :: *nat set*)
proof (*cases finite* (*range* (*V-of*::$'a \Rightarrow V$)))
case *True*

then have *lists* (*UNIV* :: *'a set*) $\lesssim$ (*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*) $\lesssim$ (*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*)
by *intro-classes* (*metis inj-V-of-list order.refl small-def small-iff 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* $\le$ *elts A*
for *V-of1* :: *'a* $\Rightarrow$ *V* **and** *V-of2* :: *'b* $\Rightarrow$ *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* $\Rightarrow$ *'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* $\Rightarrow$ *'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*)))) $\le$ *elts* (*VPi A* (λx. *B*))
if ∗: *inj V-of1 inj V-of2 range V-of1* $\le$ *elts A* **and** *range V-of2* $\le$ *elts B*
for *V-of1* :: *'a* $\Rightarrow$ *V* **and** *V-of2* :: *'b* $\Rightarrow$ *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 $f\,f'$ :: $'a \Rightarrow 'b$ **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 ?*$\varphi \le$ *elts* (*VPi A* ($\lambda x.$ *B*))
using *that* **by** (*simp add*: *VPi-I subset-eq*)
qed
with *small* [**where** $'a{=}'a$] *small* [**where** $'a{=}'b$]
show *OFCLASS*($'a \Rightarrow 'b$, *small-class*)
by *intro-classes* (*smt* (*verit*, *best*) *down-raw order-refl imageE small-def*)
qed

instance *set* :: (*small*) *small*
proof −
have *1*: *inj* ($\lambda x.$ *ZFC-in-HOL.set* (*V-of* ‘ *x*))
if *inj V-of* **for** *V-of* :: $'a \Rightarrow V$
by (*simp add*: *inj-on-def inj-image-eq-iff* [*OF that*])
have *2*: *range* ($\lambda x.$ *ZFC-in-HOL.set* (*V-of* ‘ *x*)) $\le$ *elts* (*VPow A*)
if *range V-of* $\le$ *elts A* **for** *V-of* :: $'a \Rightarrow V$ **and** *A*
using *that* **by** (*auto simp*: *inj-on-def image-subset-iff*)
from *small* [**where** $'a{=}'a$]
show *OFCLASS*($'a$ *set*, *small-class*)
by *intro-classes* (*metis 1 2 down-raw imageE small-def order-refl*)
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 $\bigwedge c.\ c \in$ *range* ($\lambda(x,y).$ *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 ZF sets corresponding to $\mathbb{R}$ and $\mathbb{C}$ and the cardinality of the contimuum

theory *General-Cardinals*
imports *ZFC-Typeclasses HOL−Analysis.Continuum-Not-Denumerable*

begin

7.1 Making the embedding from the type class explicit

definition *V-of* :: $'a$::*embeddable* $\Rightarrow$ *V*
  where *V-of* $\equiv$ *SOME f. inj f*

lemma *inj-V-of*: *inj V-of*
  unfolding *V-of-def* **by** (*metis embeddable-class.ex-inj some-eq-imp*)

declare *inv-f-f* [*OF inj-V-of*, *simp*]

lemma *inv-V-of-image-eq* [*simp*]: *inv V-of* ‘ (*V-of* ‘ *X*) = *X*
  by (*auto simp*: *image-comp*)

lemma *infinite-V-of*: *infinite* (*UNIV*::$'a$ *set*) $\Longrightarrow$ *infinite* (*range* (*V-of*::$'a$::*embeddable*$\Rightarrow$*V*))
  using *finite-imageD inj-V-of* **by** *blast*

lemma *countable-V-of*: *countable* (*range* (*V-of*::$'a$::*countable*$\Rightarrow$*V*))
  by *blast*

lemma *elts-set-V-of*: *small X* $\Longrightarrow$ *elts* (*ZFC-in-HOL.set* (*V-of* ‘ *X*)) $\approx$ *X*
  by (*metis inj-V-of inj-eq inj-on-def inj-on-image-eqpoll-self replacement set-of-elts small-iff*)

lemma *V-of-image-times*: *V-of* ‘ (*X* $\times$ *Y*) $\approx$ (*V-of* ‘ *X*) $\times$ (*V-of* ‘ *Y*)
proof −
  have *V-of* ‘ (*X* $\times$ *Y*) $\approx$ *X* $\times$ *Y*
    by (*meson inj-V-of inj-eq inj-onI inj-on-image-eqpoll-self*)
  also have . . . $\approx$ (*V-of* ‘ *X*) $\times$ (*V-of* ‘ *Y*)
   by (*metis eqpoll-sym inj-V-of inj-eq inj-onI inj-on-image-eqpoll-self times-eqpoll-cong*)
  finally show *?thesis* **.**
qed

7.2 The cardinality of the continuum

definition *Real-set* $\equiv$ *ZFC-in-HOL.set* (*range* (*V-of*::*real*$\Rightarrow$*V*))
definition *Complex-set* $\equiv$ *ZFC-in-HOL.set* (*range* (*V-of*::*complex*$\Rightarrow$*V*))
definition *C-continuum* $\equiv$ *vcard Real-set*

lemma *V-of-Real-set*: *bij-betw V-of* (*UNIV*::*real set*) (*elts Real-set*)
  by (*simp add*: *Real-set-def bij-betw-def inj-V-of*)

lemma *uncountable-Real-set*: *uncountable* (*elts Real-set*)
  using *V-of-Real-set countable-iff-bij uncountable-UNIV-real* **by** *blast*

lemma *Card C-continuum*
  by (*simp add*: *C-continuum-def Card-def*)

lemma *C-continuum-ge*: *C-continuum* $\geq$ *ℵ1*
by (*metis C-continuum-def Ord-ω1 Ord-cardinal Ord-linear2 countable-iff-vcard-less1*

uncountable-Real-set)

lemma *V-of-Complex-set*: *bij-betw V-of* (*UNIV*::*complex set*) (*elts Complex-set*)
by (*simp add*: *Complex-set-def bij-betw-def inj-V-of*)

lemma *uncountable-Complex-set*: *uncountable* (*elts Complex-set*)
using *V-of-Complex-set countableI-bij2 uncountable-UNIV-complex* **by** *blast*

lemma *Complex-vcard*: *vcard Complex-set* = *C-continuum*
proof −
define *emb1* **where** *emb1* $\equiv$ *V-of o complex-of-real o inv V-of*
have *elts Real-set* $\approx$ *elts Complex-set*
proof (*rule lepoll-antisym*)
show *elts Real-set* $\lesssim$ *elts Complex-set*
unfolding *lepoll-def*
proof (*intro conjI exI*)
show *inj-on emb1* (*elts Real-set*)
unfolding *emb1-def Real-set-def*
by (*simp add*: *inj-V-of inj-compose inj-of-real inj-on-imageI*)
show *emb1* ‘ *elts Real-set* $\subseteq$ *elts Complex-set*
by (*simp add*: *emb1-def Real-set-def Complex-set-def image-subset-iff*)
qed
define *emb2* **where** *emb2* $\equiv$ (λz. (*V-of* (*Re z*), *V-of* (*Im z*))) *o inv V-of*
have *elts Complex-set* $\lesssim$ *elts Real-set* $\times$ *elts Real-set*
unfolding *lepoll-def*
proof (*intro conjI exI*)
show *inj-on emb2* (*elts Complex-set*)
unfolding *emb2-def Complex-set-def inj-on-def*
by (*simp add*: *complex.expand inj-V-of inj-eq*)
show *emb2* ‘ *elts Complex-set* $\subseteq$ *elts Real-set* $\times$ *elts Real-set*
by (*simp add*: *emb2-def Real-set-def Complex-set-def image-subset-iff*)
qed
also have . . . $\approx$ *elts Real-set*
by (*simp add*: *infinite-times-eqpoll-self uncountable-Real-set uncountable-infinite*)
finally show *elts Complex-set* $\lesssim$ *elts Real-set* **.**
qed
then show *?thesis*
by (*simp add*: *C-continuum-def cardinal-cong*)
qed

lemma *gcard-Union-le-cmult*:
assumes *small U* **and** κ: $\bigwedge x.\ x \in U \Longrightarrow$ *gcard x* $\leq \kappa$ **and** *sm*: $\bigwedge x.\ x \in U \Longrightarrow$ *small x*
shows *gcard* ($\bigcup U$) $\leq$ *gcard U* $\otimes \kappa$
proof −
have $\exists f.\ f \in x \to$ *elts* $\kappa \wedge$ *inj-on f x* **if** $x \in U$ **for** x

using κ [*OF that*] *gcard-le-lepoll* **by** (*metis image-subset-iff-funcset lepoll-def sm that*)
then obtain φ **where** φ: $\bigwedge x.\ x \in U \Longrightarrow (\varphi\ x) \in x \to elts\ \kappa \wedge inj\text{-}on\ (\varphi\ x)\ x$
by *metis*
define u **where** $u \equiv \lambda y.\ @x.\ x \in U \wedge y \in x$
have u: $u\ y \in U \wedge y \in\ \ (u\ y)$ **if** $y \in \bigcup(\ U)$ **for** y
unfolding *u-def* **using** *that* **by** (*fast intro*!: *someI2*)
define ψ **where** $\psi \equiv \lambda y.\ (u\ y,\ \varphi\ (u\ y)\ y)$
have U: $elts\ (gcard\ U) \approx U$
using *assms* **by** (*simp add*: *gcard-eqpoll*)
have $\bigcup U \lesssim U \times elts\ \kappa$
unfolding *lepoll-def*
proof (*intro exI conjI*)
show $inj\text{-}on\ \psi\ (\bigcup\ U)$
using φ u **by** (*smt* (*verit*) ψ*-def inj-on-def prod.inject*)
show ψ ‘ $\bigcup\ U \subseteq U \times elts\ \kappa$
using φ u **by** (*auto simp*: ψ*-def*)
qed
also have $\ldots\ \approx elts\ (gcard\ U \otimes \kappa)$
using *U elts-cmult eqpoll-sym eqpoll-trans times-eqpoll-cong* **by** *blast*
finally have $(\bigcup U) \lesssim elts\ (gcard\ U \otimes \kappa)$ **.**
then show *?thesis*
by (*metis cardinal-idem cmult-def gcard-eq-vcard lepoll-imp-gcard-le small-elts*)
qed

lemma *gcard-Times* [*simp*]: $gcard\ (X \times Y) = gcard\ X \otimes gcard\ Y$
proof (*cases small* $X \wedge$ *small* Y)
case *True*
have $elts\ (gcard\ (X \times Y)) \approx X \times Y$
by (*simp add*: *True gcard-eqpoll*)
also have ... $\approx elts\ (gcard\ X) \times elts\ (gcard\ Y)$
by (*simp add*: *True eqpoll-sym gcard-eqpoll times-eqpoll-cong*)
also have ... $\approx elts\ (gcard\ X \otimes gcard\ Y)$
by (*simp add*: *elts-cmult eqpoll-sym*)
finally show *?thesis*
using *Card-cardinal-eq cmult-def gcardinal-cong* **by** *force*
next
case *False*
have $gcard\ (X \times Y) = 0$
by (*metis False Times-empty gcard-big-0 gcard-empty-0 small-Times-iff*)
then show *?thesis*
by (*metis False cmult-0 cmult-commute gcard-big-0*)
qed

7.3 Countable and uncountable sets

lemma *countable-iff-g-le-Aleph0*:
assumes *small* X
shows *countable* $X \longleftrightarrow gcard\ X \leq \aleph 0$

proof −
  have *countable X* $\longleftrightarrow$ *X* $\lesssim$ *elts* ω
    by (*simp add*: ω*-def countable-iff-lepoll inj-ord-of-nat*)
  also have ... $\longleftrightarrow$ *gcard X* $\leq$ $\aleph$*0*
    using *Card-*ω *Card-def assms gcard-le-lepoll lepoll-imp-gcard-le* **by** *fastforce*
  finally show *?thesis* **.**
qed

lemma *countable-imp-g-le-Aleph0*: *countable X* $\Longrightarrow$ *gcard X* $\leq$ $\aleph$*0*
  by (*meson countable countable-iff-g-le-Aleph0*)

lemma *finite-iff-g-le-Aleph0*: *small X* $\Longrightarrow$ *finite X* $\longleftrightarrow$ *gcard X* $<$ $\aleph$*0*
  by (*metis Aleph-0 eqpoll-finite-iff finite-iff-less-Aleph0 gcard-eq-vcard gcard-eqpoll gcardinal-cong*)

lemma *finite-imp-g-le-Aleph0*: *finite X* $\Longrightarrow$ *gcard X* $<$ $\aleph$*0*
  by (*meson finite-iff-g-le-Aleph0 finite-imp-small*)

lemma *countable-infinite-gcard*: *countable X* $\wedge$ *infinite X* $\longleftrightarrow$ *gcard X* $=$ $\aleph$*0*
proof −
  have *gcard X* $= \omega$
    if *countable X* **and** *infinite X*
    using *that countable countable-imp-g-le-Aleph0 finite-iff-g-le-Aleph0 less-V-def* **by** *auto*
  moreover have *countable X* **if** *gcard X* $= \omega$
    by (*metis Aleph-0 countable-iff-g-le-Aleph0 dual-order.refl gcard-big-0 omega-nonzero that*)
  moreover have *False* **if** *gcard X* $= \omega$ **and** *finite X*
    by (*metis Aleph-0 dual-order.irrefl finite-iff-g-le-Aleph0 finite-imp-small that*)
  ultimately show *?thesis*
    by *auto*
qed

lemma *uncountable-gcard*: *small X* $\Longrightarrow$ *uncountable X* $\longleftrightarrow$ *gcard X* $>$ $\aleph$*0*
  by (*simp add*: *Card-is-Ord Ord-not-le countable-iff-g-le-Aleph0*)

lemma *uncountable-gcard-ge*: *small X* $\Longrightarrow$ *uncountable X* $\longleftrightarrow$ *gcard X* $\geq$ $\aleph$*1*
  by (*simp add*: *uncountable-gcard csucc-le-Card-iff one-V-def*)

lemma *subset-smaller-gcard*:
  assumes κ: $\kappa \leq$ *gcard X Card* κ
  obtains *Y* **where** *Y* $\subseteq$ *X gcard Y* $= \kappa$
proof (*cases small X*)
  case *True*
  then have *elts* $\kappa \lesssim X$
    by (*meson assms*(*1*) *eqpoll-imp-lepoll gcard-eqpoll lepoll-trans less-eq-V-def subset-imp-lepoll*)
  then obtain *Y* **where** *Y* $\subseteq$ *X elts* $\kappa \approx Y$
    by (*metis bij-betw-def eqpoll-def lepoll-def*)

then show *?thesis*
using *Card-def* ‹*Card* κ› *gcardinal-cong that* **by** *force*
next
case *False*
with *assms* **show** *?thesis*
by (*metis antisym gcard-big-0 le-0 order-refl that*)
qed

lemma *Real-gcard*: *gcard* (*UNIV*::*real set*) = *C-continuum*
by (*metis C-continuum-def V-of-Real-set bij-betw-def gcard-eq-vcard gcard-image*)

lemma *Complex-gcard*: *gcard* (*UNIV*::*complex set*) = *C-continuum*
by (*metis Complex-vcard V-of-Complex-set bij-betw-def gcard-eq-vcard gcard-image*)

end

8 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.